UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
GRADUADO EN INGENIERÍA DEL SOFTWARE

Desarrollo de videojuego de estrategia por turnos

Turn-Based game development

Realizado por

Adrián Guerrero Álvarez

Tutorizado por
David Bueno Vallejo

Departamento
Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA
MÁLAGA, SEPTIEMBRE DE 2023

Fecha defensa: Septiembre de 2023

Abstract

The end of degree project that is being introduced has the objective to show the work plan that has been planned from the beginning of the project, we followed this plan to carry out the development of a turn-based game using the game engine Unity, following a Agile Project Management Methodology, like Scrum base on Sprints, we have done six Sprints, each one of them with an average length of two weeks, where we split the work to keep a continuous work along the whole project.

Inside our videogame, we have implemented different actions, these actions can be used for all available characters, some of these actions make use of special algorithm that we have developed, for example, we have a pathfinding that can get the most efficient way to reach a position when needed, therefore we have explained and implemented algorithm A*, that allow us to get a way base on heuretics.

Considering this as a single player game, we have built an artificial intelligence, because of that we made an investigation to decide witch artificial intelligence behaviour fits better with the final behaviour we want, one that allowed us to have real enemies, we discussed others kinds of artificial intelligence, like "State Machines" or "Behaviour Trees", but at the end, we selected "Utility AI as the main AI, the one that gives us the chances to select an action base on different scores.

KEYWORDS: Videogame, Turn-Based game, Artificial Intelligence, Path-finding, Players Actions

Resumen

El trabajo de Fin de Grado, que se presenta a continuación, tiene como objetivo mostrar el plan que se ha planteado en un inicio y cómo hemos seguido dicho plan para realizar el desarrollo de un videojuego de estrategia usando, para ello, el motor de videojuegos Unity y siguiendo una metodología Agile como es Scrum, basado en Sprints.

Hemos realizado seis Sprints que han tenido una duración media de dos semanas y donde hemos dividido el trabajo para seguir un desarrollo continuo a lo largo del proyecto.

Dentro del videojuego hemos implementado diferentes acciones las cuales podrán ser usadas por todos los personajes disponibles y, alguna de éstas, hacen uso de distintos algoritmos que hemos desarrollado para lograr obtener una búsqueda de caminos que obtenga, siempre que sea posible, al menos, un camino y que sea el más eficiente para cuando sea necesario. Por ello, hemos explicado e implementado el Algoritmo A* que nos permite obtener un camino basado en la heurística.

Dado que es un videojuego de un jugador hemos creado una Inteligencia Artificial. Para ello, hemos realizado una investigación que nos llevará a tomar una decisión sobre qué Inteligencia Artificial se acopla, de mejor manera, a lo que queremos lograr y que nos permita dar vida a los enemigos.

También comentaremos otras como pueden ser las Máquinas de Estados o los Árboles de Decisión, pero eligiendo, entre todas las opciones disponibles, la "IA de Utilidad" que nos permitirá elegir una acción basada en puntuaciones.

Palabras claves: Videojuego, Juego por Turnos, Inteligencia Artífical, Búsqueda de Caminos, Acciones de los personajes

Índice

1. Introducción

1.1. Motivación

El sector de los videojuegos está en completo auge y será, si no lo es ya, uno de los sectores más relevantes que habrá en el futuro.

Por ello, gracias a los avances que se han realizado dentro de los sectores de la Inteligencia Artificial es posible realizar, cada vez más, juegos que tengan una mayor complejidad en el ámbito de las Inteligencias Artificiales y juegos que permitan la interacción y la posibilidad de crear experiencias que puedan ser satisfactorias.

Por ello, es una buena oportunidad académica de poder aplicar no sólo los conocimientos de programación sino también conocimientos en la gestión de proyectos y, todo ello, para poder explorar nuevas tecnologías y técnicas en el campo de videojuegos.

1.2. Objetivos

Por una parte, el Trabajo de Fin de Grado tiene como objetivo la gestión de un proyecto de mediana dimensión. Esto se llevará a cabo, como hemos comentado anteriormente, mediante una metodología Agile basada en Scrum, dividida en seis Sprints que nos permiten medir los objetivos para poder evaluar el progreso y el éxito de los diferentes objetivos que se han realizado a lo largo del proyecto y que funcionan como puntos de referencia para saber la dirección en la que se encuentra dicho proyecto.

Por otra parte, el proyecto consiste en la realización de un videojuego de estrategia por turnos que nos permita desarrollar los conocimientos en el desarrollo de videojuegos y que, por tanto, nos permita también investigar e implementar en otros campos que están relacionados. En nuestro caso, en el ámbito de la Inteligencia Artificial, los caminos de búsqueda y otros ámbitos en el desarrollo de videojuegos como pueden ser los sonidos, las interfaces, cámaras dentro del juego y las animaciones de los personajes.

1.3. Estructura del documento

La estructura del documento está dividida en secciones que, a su vez, se dividen en otras subsecciones que nos permiten estructurar el proyecto de manera organizada. El documento está estructurado de la siguiente forma:

- **Introducción:** Proporcionamos una visión general del proyecto, incluyendo la motivación y los objetivos, además hablaremos sobre la estructura del documento y la metodología que hemos usado.
- **Desarrollo del sistema de mallas:** Comentaremos cómo hemos desarrollado el sistema de mallas dentro del videojuego y cómo hemos implementado, dentro del sistema, el algoritmo de búsqueda A* para obtener el mejor camino posible hacia una ubicación.
- **Implementación de las acciones de las diferentes unidades:** Explicación del conjunto de acciones que se encuentran disponibles para todos los personajes y cómo hemos realizado su implementación mediante la interfaz baseAction.
- **Creación e implementación de un sistema de porcentajes y cobertura:** Desarrollo del sistema de coberturas y los porcentajes aplicados a la acción de disparar de los personajes.
- **Inteligencia Artificial:** Abordamos el punto fundamental del proyecto donde debatimos sobre cuál de las Inteligencias Artificiales se adecua mejor a nuestro proyecto y hablamos sobre el porqué de la elección de la IA de utilidad.
- **Diseño e implementación de menús, interfaces y unidades:**Mostramos los diseños que hemos desarrollado para los menús al igual que la interfaz que se incluirá dentro del juego.
- **Creación de niveles jugables:**Por último, mostraremos los niveles jugables que hemos incluido dentro del juego, explicando la implementación que hemos desarrollado sobre el sistema de guardado y cargado del videojuego, así como las animaciones y sonidos que hemos incluido.
- **Conclusión:** Comentamos una idea final de cómo ha ido el proyecto y las sensaciones sobre éste.

1.4. Metodología de trabajo

La metodología de trabajo que hemos usado en este proyecto está basada en una metodología Agile basada en Scrum. Hemos dividido el proyecto en diferentes Sprints y,

con ello, lo que conseguimos es que este tipo de metodología nos permita poder tomar decisiones más repentinas sobre el proyecto, pudiendo priorizar ciertos aspectos que, a lo mejor, en el inicio no parecían tan importantes como otros y según nos adentremos van tomando relevancia. Por ello, esta metodología nos permite tener una mayor adaptación y flexibilidad y ello no hubiese sido posible si hubiésemos optado por otras como pueden ser los modelos en cascada, los cuales no dan lugar a posibles cambios en el desarrollo, sino que tienen que seguir una línea hasta que se acabe.

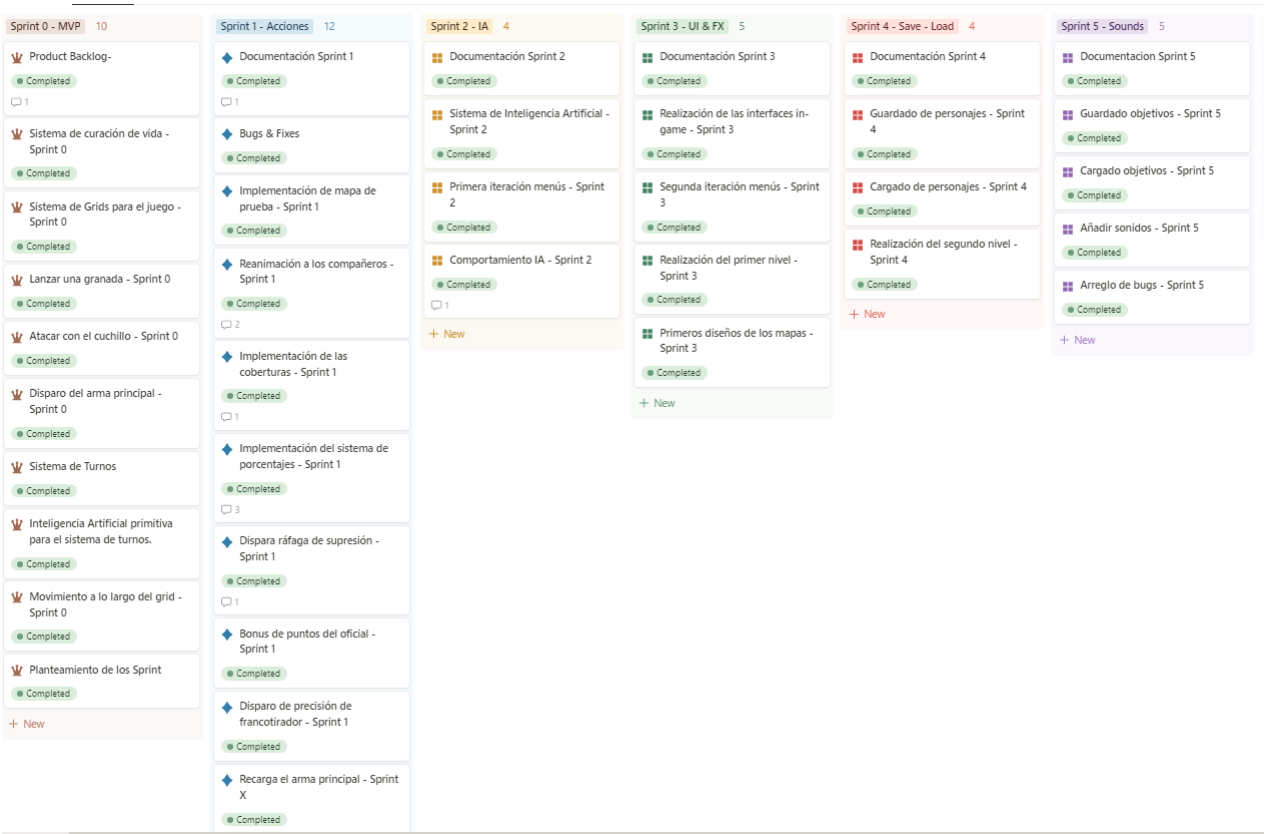

Figura 1: División de las tareas por Sprints

El proyecto ha tenido un total de seis Sprints y éstos, a su vez, han tenido una duración de trece/quince días, siendo necesario en algún Sprint aumentar el número de éstos para poder finalizar el desarrollo del sistema que se estaba creando. En este caso era la Inteligencia Artificial y ello era debido a que se requería de una mayor búsqueda de información sobre las distintas posibilidades de Inteligencia Artificial que podíamos escoger y cuál se adecuaba mejor a nuestro proyecto. Además de planificar, al inicio, un poco el desarrollo de cada Sprint al final de cada uno hemos llevado a cabo una revisión del contenido realizado y planificamos, con detalle, cómo se va a plantear el desarrollo del

siguiente Sprint.

1.4.1. Sprint 0 — Bases del proyecto — 06/02/2023 - 20/02/2023

En este Sprint se realizaron los cimientos del proyecto. Se hizo la base sobre la que se sustenta el juego, creando el sistema de mallas sobre el cual los personajes realizan sus movimientos y acciones.

Las principales acciones que se realizaron en este Sprint fueron:

- Movimiento del personaje.
- Disparo del personaje.
- Curación del personaje.
- Ataque cuerpo a cuerpo.
- Lanzamiento de granada.

En esta primera iteración no se tuvo en cuenta el Algoritmo A* y, por ello, el Path-finding tampoco está contemplado, el personaje no tenía por qué recorrer el camino con mayor rendimiento. También se implementó el sistema de turnos, por el cual el jugador podrá elegir cuando terminar el turno, pulsando el botón correspondiente.

También se realizó, para comprobar este sistema de turnos, una Inteligencia Artificial primitiva, la cual solo atacaba al enemigo para comprobar el correcto funcionamiento. El turno de los enemigos acaba cuando a éstos no les quedan más puntos de acción por gastar y, por tanto, sería, de nuevo, turno del jugador.

1.4.2. Sprint 1 — Desarrollo de los personajes — 20/02/2023 - 06/03/2023

En este segundo Sprint se han realizado la mayoría de los aspectos relacionados con las acciones de los personajes y los diferentes roles que éstos podrán tener. Por ello, en este Sprint se decidió las acciones que iban a tener cada personaje y la función que éstos cumplirían.

Las acciones que se han realizado en este Sprint fueron:

- Reanimación de aliados.
- Disparo del lanzacohetes.

- Bonus de movimiento del oficial.
- Disparo de precisión.
- Disparo de supresión.

Además de estas acciones, en la segunda mitad del Sprint, añadimos también el sistema de coberturas. Y, junto con ello, el sistema de porcentajes.

Aunque en Sprints posteriores se ha tenido que modificar dicho sistema de porcentajes ya que se añadieron otras variables a tener en cuenta, como será el disparo de supresión o la adición del selector de dificultad, con el cual se han añadido diferentes aumentos de porcentajes.

1.4.3. Sprint 2 — Inteligencia Artificial — 06/03/2023 - 20/03/2023

En el tercer Sprint, que hemos realizado, hemos hecho un avance, casi final, en el desarrollo de la Inteligencia Artificial en el juego y que está basado en el Utility AI por el cual los movimientos del enemigo están sujetos a una puntuación, basándose ésta en el resultado que obtendrían si realizasen la acción.

Para tomar la decisión de elegir este tipo de Inteligencia Artificial hemos hecho un proceso de recolección de información, buscando diferentes tipos de IA y viendo cuál se adecuaba más a nuestro tipo de juego. Hemos recopilado información de los Árboles de Comportamiento y de las Máquinas de Estado, pero la que mejor se adecuaba a lo que estábamos buscando era la IA de Utilidad.

Dado que este Sprint duró tres semanas, en vez de dos, además de la Inteligencia Artificial también se realizó la primera iteración de los menús que acompañarían al juego, aunque, más tarde, se cambiaron para adecuarlos más a la estética que estábamos buscando.

1.4.4. Sprint 3 — Primer Mapa & UI — 20/03/2023 - 03/04/2023

Este Sprint ha estado dedicado al desarrollo del mapeado de un nivel que tendremos en el juego. Además de ello, se realizaron diferentes cambios en el menú a lo largo del Sprint ya que algunos de los diseños no fueron lo suficientemente convincentes al final,

por ello se optó por el último diseño que realizamos. A la hora de tomar la decisión final se optó por elegir entre el modelo 3 y el modelo que sí hemos elegido y que veremos más adelante. Estos diseños proyectan, de mejor manera, el estilo de videojuego que estamos mostrando.

Estos son los diseños y cómo van progresando hasta el final.

Figura 2: Modelos realizados para el menú

Además, se realizaron los cambios de la interfaz de juego dentro de los niveles cambiando la que teníamos, en un inicio, que era solo funcional e incluyendo en la interfaz una parte especializada en la situación del escuadrón donde podremos ver el estado de vida en el que se encuentran, la munición que les queda disponible, las granadas restantes y, además, el número de puntos de acción disponibles. En este Sprint hemos dejado, en un buen punto, el aspecto visual del juego, cambiando, en un futuro, solo pequeñas partes que fuesen necesarias.

1.4.5. Sprint 4 — Guardado y cargado de las unidades, segundo mapa— 03/04/2023- 20/04/2023

De nuevo este Sprint ha tenido un poco más de duración debido a que la funcionalidad de guardar y cargar partida era más compleja que el resto de las tareas y, por ello, requería de una mayor recolección de información para saber cuál era la mejor forma de afrontar

cómo queríamos guardar y luego recoger la información. Se optó, por ello, por crear diferentes clases e interfaces de las cuales, más tarde, heredarían aquellas clases de las que necesitásemos guardar información simplificando, un poco, para el caso de que en un futuro queramos guardar más información.

Se tomó la decisión de toda la información que queríamos guardar dentro de los niveles y, en primer lugar, se realizó la función de guardar aquella información que estuviese relacionada con el ámbito de las unidades permitiendo, así, guardar la posición, la vida y el resto de información respecto a todas las unidades, tanto aliadas como enemigas y, dado que llevó más tiempo de lo establecido, se decidió que el sistema de guardado de los objetivos se pasaría al siguiente Sprint. Además del sistema de guardado también se realizó el segundo nivel jugable, teniendo éste las mismas funcionalidades que el primero. Es decir, deberán de cumplir ciertos objetivos interactuando con diferentes objetos o, si fuese necesario, eliminar a los enemigos restantes.

1.4.6. Sprint 5 — Guardado y cargado de los objetivos, sonidos y arreglo de errores — 20/04/2023- 02/05/2023

En este Sprint se terminó la funcionalidad de guardar y cargar partida ya que se terminó la función de guardar los objetivos. De esta manera, el jugador podrá guardar y cargar partida en todo momento que lo necesite, guardando toda la información necesaria.

Adicionalmente, también se implementaron diferentes sonidos dentro del juego, además de música para todas las posibles acciones que puedan realizar los personajes. En relación a esto, también se incluyó una funcionalidad nueva dentro del menú de pausa para que el jugador pueda aumentar o disminuir el volumen dependiendo de si éste está muy alto o muy bajo y se acomode al jugador.

Por último, para terminar el proyecto se procedió a corregir diferentes errores que podían surgir como, por ejemplo, algunos que ocurrían a la hora de guardar y cargar partida o cuando los personajes querían lanzar alguna acción, provocando que el juego se bloquease. Eliminando estos errores se redujo, en gran cantidad, el número de bugs que el jugador podría encontrarse.

2. Desarrollo del sistema de mallas

Una de las principales funciones de nuestro videojuego es el poder realizar diferentes movimientos por el mapa, teniendo total libertad a la hora de tomar nuestras propias decisiones.

Por tanto, es fundamental desarrollar un sistema de mallas y dar esta posibilidad al jugador. Para ello, creamos lo que hemos denominado el "Level Grid". Éste controla la lógica a la hora de aplicar los movimientos.

Este sistema de mallas es la base del videojuego y es que todas las posibles acciones y algoritmos usados tendrán en cuenta las diferentes posiciones, ya sea para realizar las acciones en sí mismas o para realizar diferentes revisiones para el entorno.

En primer lugar, se han tenido que hacer diferentes conversiones, ya que dentro del editor de Unity nos encontramos en un entorno 3D en el cual podemos encontrar tres coordenadas, siendo éstas la X, Y y Z. Dado que, en nuestro videojuego, no se tienen en cuenta diferentes alturas debemos eliminar aquella coordenada que se encarga del aspecto vertical del entorno. Por ello, nuestro "Level Grid" se mueve en el eje XZ y tenemos diferentes coordenadas.

Por un lado, tenemos, como la hemos denominado nosotros, la "WorldPosition", siendo la coordenada a tener en cuenta la Y (aunque siempre sea 0). Y, por otro lado, tenemos la "GridPosition" que, por decisión del proyecto, decidimos que cada celda de nuestro sistema tuviese un espacio de dos unidades. Otra parte de nuestro sistema de mallas es el "GridObject", siendo su comportamiento el de poder controlar lo que se encuentra en cada cuadrícula de nuestro sistema, pudiendo ser éste una unidad aliada, enemiga, un objeto sin ninguna utilidad y, por simple mapeado, una cobertura para que las unidades puedan cubrirse e incluso objetos que sirvan para poder cumplir los objetivos de la misión, es decir, los objetos interactuables. En cada cuadrícula se podrá situar una y solo una unidad. Pero esto no significa que una unidad no pueda pasar por una cuadrícula en la que haya otra unidad, la unidad si podrá moverse y podrá atravesar a sus aliados o enemigos, permitiendo un mayor movimiento. Pero, en ningún caso, podrá estar en la misma casilla cuando se acabe el movimiento.

Cuando una unidad sea eliminada o sea abatida, con posibilidad de ser reanimada, el GridObject lo tendrá en cuenta y dejará su posición libre para que otras unidades puedan

moverse a esa posición. Pero desde el momento en el que se reanime a la unidad, si ello es posible, la casilla volverá a quedar bloqueada hasta que la unidad, que se encuentre en ella, se mueva y/o vuelva a ser eliminada.

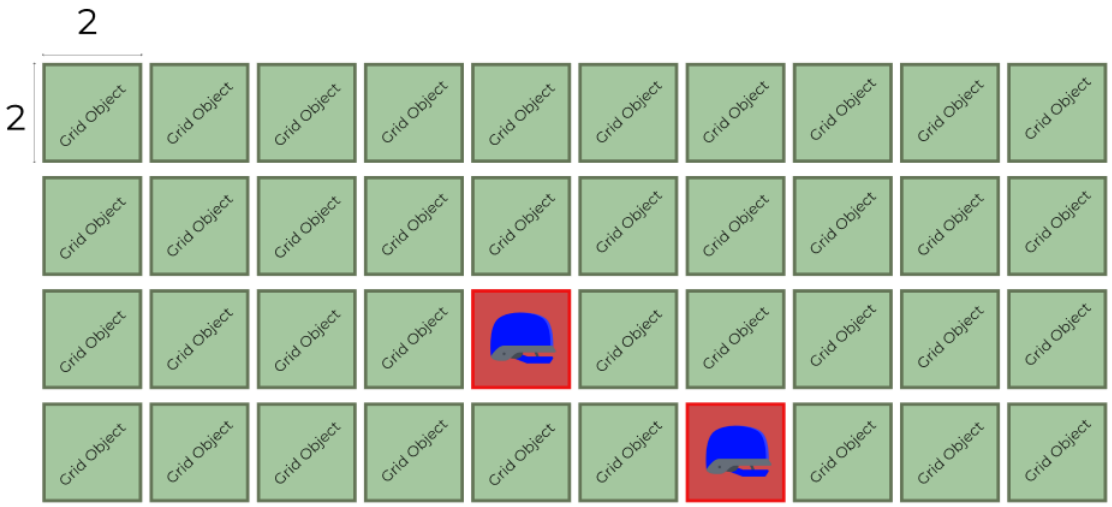

Figura 3: Ejemplo de Level Grid

Este sistema de mallas tendrá en cuenta también los obstáculos. Esto será necesario para que nuestro sistema de "Pathfinding" pueda encontrar la forma más óptima de recorrer el camino que tenga estipulado. Es, por ello, que para que el Algoritmo A* tenga el máximo rendimiento hemos creado la capa ("Layer") de obstáculos. Y, gracias a ella, podemos evitar las zonas que no queremos que sean transitables, siendo el algoritmo conocedor de todas estas situaciones. Pero todo ello lo explicaremos, con más detalle, en los siguientes puntos, en lo que respecta al "Pathfinding y el Algoritmo A*".

2.0.1. Pathfinding y el Algoritmo A*

Para que nuestro personaje decida qué camino es el más óptimo hemos hecho uso de un algoritmo de búsqueda en grafos, como es el "Algoritmo A*", que se adecua perfectamente a las propiedades y descripciones en las que se basa nuestro sistema de mallas. El Algoritmo A* está basado en el peso de grafos. Comenzando desde un nodo inicial dicho algoritmo busca el coste más bajo (entendiendo por coste el camino más óptimo para alcanzar un destino final). Este algoritmo, que realiza sus cálculos nodo a nodo, pertenece al método de búsqueda preferente por lo mejor (Best First Search) realizando una serie de

posibles caminos hasta alcanzar el final del mismo. En cada iteración del nodo en el que nos encontramos, el algoritmo busca cuál de los posibles nodos, que tenemos alrededor, es el que tiene un menor coste. Es decir, cuál de los nodos, contiguos al actual, es el que mejor se adecua para alcanzar el punto final con el menor coste posible.

Por ello, en cada "GridObject" que tenemos en nuestro sistema, hemos creado un PathNode, el cual contiene los valores para el valor de coste y heurístico en cada nodo.

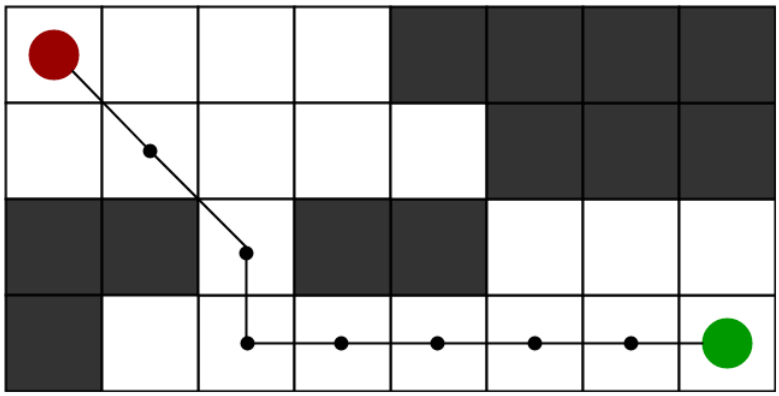

Figura 4: Algoritmo A*

De forma similar a la figura, nuestro sistema de mallas se basará en un conjunto de celdas contiguas donde podemos establecer las mismas condiciones que las necesarias para el Algoritmo A*.

En primer lugar, daremos una explicación acerca de en qué consiste el Algoritmo A* y por qué hemos tomado la decisión de usar el mismo.

Un problema que tienen algunos algoritmos, diferentes al mencionado, es que se guían por la función heurística. Esto puede dar lugar, en algunos casos, a no indicar correctamente el camino de coste más bajo y es, por ello, que no nos vale como algoritmo ya que queremos, en todo momento, la solución que tenga un mayor rendimiento.

Por ello, el Algoritmo A* tiene esto en cuenta y utiliza una función de evaluación como es:

$$f(n) = g(n) + h'(n)$$

Donde h'(n) representa el valor heurístico del nodo a evaluar, desde el actual n hasta el final, y g(n) el coste real del camino recorrido para llegar a dicho nodo n desde el nodo inicial.

El Algoritmo A* mantiene dos estructuras de datos auxiliares, es decir, g(n) es el coste del camino entre el nodo inicial y el nodo que estamos comprobando y h'(n), el cuál es el coste heurístico (un coste supuesto), desde el nodo que estamos valorando hasta el final del camino.

Para realizar esta implementación, en primer lugar, lo que planteamos es que el Path-finding pueda reconocer qué posiciones del mapa son caminables y cuales no, es decir, que pueda deducir si hay algún obstáculo en las posiciones para saber si se puede pasar por ellas. Para ello, realizamos un escáner al inicio para guardar las referencias en las posiciones.

Comprobación de terrenos caminables

```
1  public void Setup(int width, int height, float cellSize)
2  {
3      this.width = width;
4      this.height = height;
5      this.cellSize = cellSize;
6      gridSystem = new GridSystem<PathNode>(width, height, cellSize, (GridSystem<PathNode> g
         , GridPosition gridPosition) => new PathNode(gridPosition));
7      for (int x = 0; x < width; x++)
8      {
9          for (int z = 0; z < height; z++)
10         {
11             GridPosition gridPosition = new GridPosition(x, z);
12             Vector3 worldPosition = LevelGrid.Instance.GetWorldPosition(gridPosition);
13             float raycastOffsetDistance = 5f;
14             if(Physics.Raycast(worldPosition + Vector3.down * raycastOffsetDistance, Vector3.up,
       raycastOffsetDistance * 2, osbtaclesLayerMask))
15             {
16                 GetNode(x, z).SetIsWalkable(false);
17             }
18         }
19     }
20 }
```

Una vez hemos cubierto todo el terreno podemos aplicar el método FindPath por el cual, con la posición inicial y la posición final, obtenemos el camino más eficiente para alcanzar el destino. Para ello, con los cálculos que hemos comentado, el algoritmo realiza

la búsqueda comprobando los terrenos por los que el personaje podría caminar y realiza los cálculos de heurística pertinentes.

Tendremos dos listas con las cuales trabajaremos, éstas serán **openList** y **closedList**. En la primera lista estaremos almacenando aquellas casillas que estén siendo analizadas y que pueden seguir abriendo posibles caminos para alcanzar el destino. En cambio, en la segunda lista guardaremos aquellas casillas que ya han sido revisadas.

Después de inicializar estas listas aplicaremos los costes a todos los nodos para que el algoritmo pueda realizar su búsqueda. Para ello, usaremos el método de **CalculateDistance** con el cual realizaremos el cálculo heurístico, que sería el coste que podría alcanzar esa posición supuestamente. Este cálculo, en un sistema de matrices, se hace restando los ejes X y Z de la matriz, sacamos su valor absoluto y realizamos un cálculo para saber cuántos movimientos diagonales ha tenido que realizar y el resto se aplica a movimientos rectos.

Método para calcular el coste heurístico entre dos puntos del mapa

```
1  private const int MOVE_STRAIGHT_COST = 10;
2  private const int MOVE_DIAGONAL_COST = 14;
3
4  public int CalculateDistance(GridPosition gridPositionA, GridPosition gridPositionB)
5  {
6      GridPosition gridPositionDistance = gridPositionA - gridPositionB;
7      int xDistance = Mathf.Abs(gridPositionDistance.x);
8      int zDistance = Mathf.Abs(gridPositionDistance.z);
9      int remaining = Mathf.Abs(xDistance - zDistance);
10     return MOVE_DIAGONAL_COST * Mathf.Min(xDistance, zDistance) +
         MOVE_STRAIGHT_COST * remaining;
11 }
```

Tenemos también un método para poder recoger el nodo con un menor coste F, para seguir buscando por el nodo que tenga un menor coste en ese momento y seguir buscando el más óptimo.

Búsqueda del nodo con menor coste en la lista de nodos abiertos

```
1    private PathNode GetLowestFCostPathNode(List<PathNode> pathNodeList)
2    {
3        PathNode lowestFCostPathNode = pathNodeList[0];
```

```
        for(int i = 0; i < pathNodeList.Count; i++)
        {
            if(pathNodeList[i].GetFCost() < lowestFCostPathNode.GetFCost())
            {
                lowestFCostPathNode = pathNodeList[i];
            }
        }
        return lowestFCostPathNode;
    }
```

Con todos estos métodos auxiliares podemos calcular el camino que deseamos, haciendo uso del método general "FindPath", desde un punto inicial a uno final que pasamos como argumento, donde, como hemos comentado, vamos rellenando las listas, tanto la "openList" como la "closedList".

En primer lugar, inicializamos los costes de todos los nodos, tanto los costes heurísticos como el coste real, que hemos comentando anteriormente en la fórmula que aplicamos para obtener, en este caso, lo que llamamos el coste F.

Inicialización de todos los nodos del camino*

```
public List<GridPosition> FindPath(GridPosition startGridPosition, GridPosition
    endGridPosition, out int pathLength)
    {
        List<PathNode> openList = new List<PathNode>();
        List<PathNode> closedList = new List<PathNode>();

        PathNode startNode = gridSystem.GetGridObject(startGridPosition);
        PathNode endNode = gridSystem.GetGridObject(endGridPosition);
        openList.Add(startNode);
        for(int x = 0; x < gridSystem.GetWidth(); x++)
        {
            for (int z = 0; z < gridSystem.GetHeight(); z++)
            {
                GridPosition gridPosition = new GridPosition(x, z);
                PathNode pathNode = gridSystem.GetGridObject(gridPosition);
                pathNode.SetGCost(int.MaxValue);
                pathNode.SetHCost(0);
                pathNode.CalculateFCost();
                pathNode.ResetCameFromPathNode();
```

```
19          }
20      }
21      startNode.SetGCost(0);
22      startNode.SetHCost(CalculateDistance(startGridPosition, endGridPosition));
23      startNode.CalculateFCost();
```

Una vez hemos inicializado todos los nodos, realizamos comprobaciones mientras haya caminos por los que podamos seguir comprobando costes, es decir, mientras haya caminos incluidos dentro de la lista "openList". Lo primero a tener en cuenta es si el nodo que estamos comprobando es el nodo objetivo al que queremos llegar y, si es el caso, terminaríamos las comprobaciones. En otro caso, eliminamos el nodo de la lista de nodos sin comprobar, y lo añadimos a la lista de nodos cerrados que no volveremos a comprobar.

Para seguir buscando, en el caso de que no sea la posición final, hacemos una iteración por todos los nodos contiguos al nodo actual. La comprobación que realizamos es, en primer lugar, que no hayamos comprobado anteriormente ese mismo nodo y, en segundo lugar, que ese nodo sea uno transitable. En caso de que sea un nodo válido, calculamos el coste tentativo y, en caso de que el coste sea menor que el que tenía anteriormente, cambiamos los costes anteriores.

Cálculo del mejor camino mediante el Algoritmo A*

```
1       while(openList.Count > 0)
2       {
3           PathNode currentNode = GetLowestFCostPathNode(openList);
4           if(currentNode == endNode)
5           {
6               pathLength = endNode.GetFCost();
7               return CalculatePath(endNode);
8           }
9           openList.Remove(currentNode);
10          closedList.Add(currentNode);
11          foreach(PathNode neighbourNode in GetNeighbourList(currentNode))
12          {
13              if (!closedList.Contains(neighbourNode))
14              {
15                  int tentativeGCost = currentNode.GetGCost() + CalculateDistance(currentNode.
    GetGridPosition(), neighbourNode.GetGridPosition());
16                  if(tentativeGCost < neighbourNode.GetGCost())
```

```
17                  {
18                      neighbourNode.SetCameFromPathNode(currentNode);
19                      neighbourNode.SetGCost(tentativeGCost);
20                      neighbourNode.SetHCost(CalculateDistance(neighbourNode.GetGridPosition(),
    endGridPosition));
21                      neighbourNode.CalculateFCost();
22                      if (!openList.Contains(neighbourNode))
23                      {
24                          openList.Add(neighbourNode);
25                      }
26                  }
27              }
28              if (!neighbourNode.IsWalkable())
29              {
30                  closedList.Add(neighbourNode);
31              }
32          }
33      }
34      pathLength = 0;
35      return null;
36  }
```

3. Implementación de las acciones de las diferentes Unidades

En nuestro videojuego cada personaje podrá realizar diferentes acciones, dependiendo del rol que cumpla en el escuadrón. Estos roles pueden ser:

- Soldado de infantería.

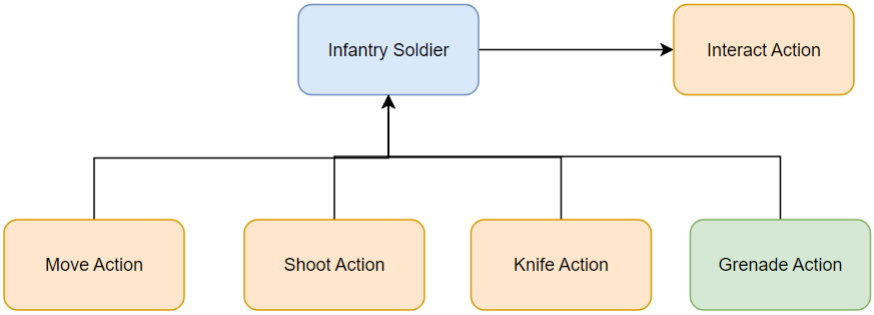

Figura 5: Soldado de Infanteria

- Soldado lanzacohetes.

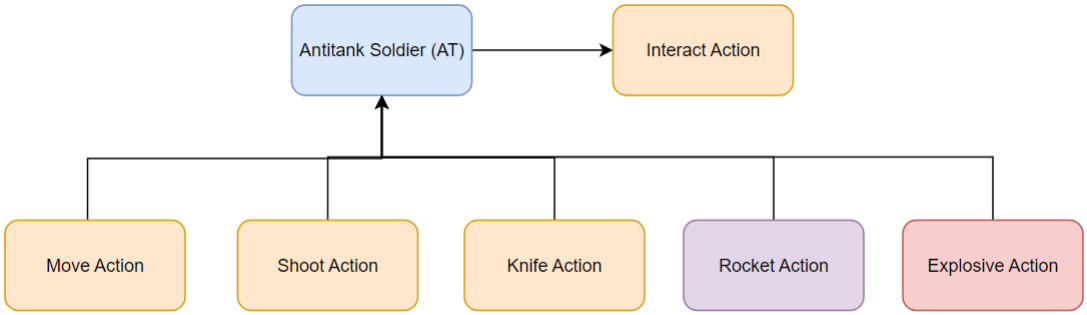

Figura 6: Soldado de Lanzacohetes

- Médico de combate.

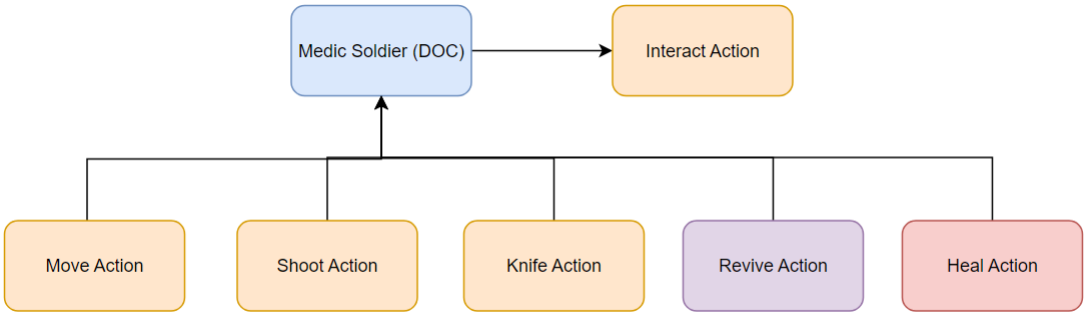

Figura 7: Médico de combate

- Oficial del escuadrón.

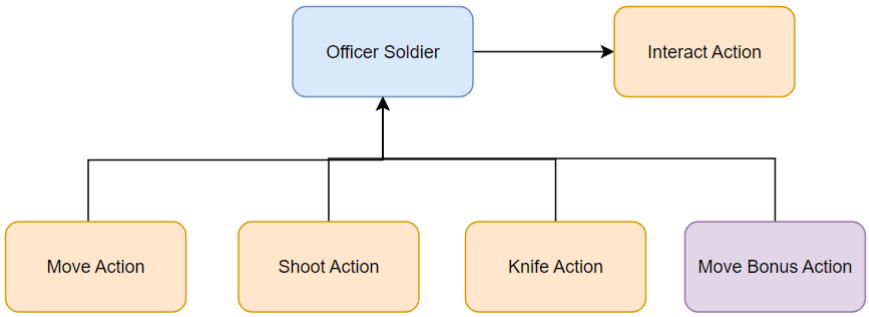

Figura 8: Oficial del escuadrón

- Soldado de reconocimiento.

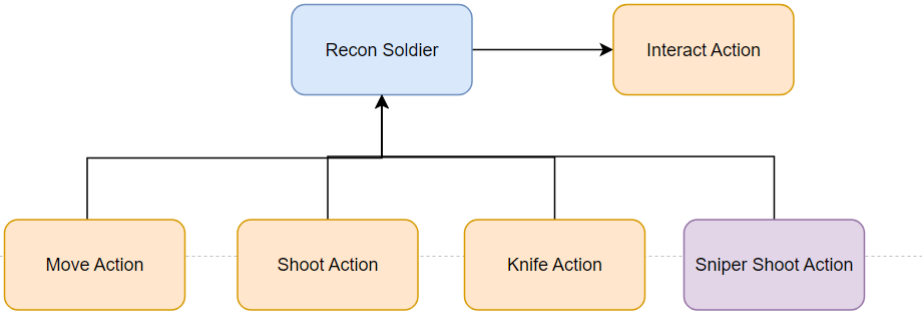

Figura 9: Soldado de reconocimiento

- Soldado de apoyo

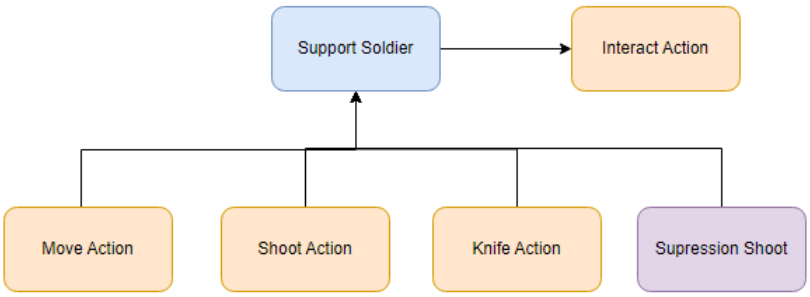

Figura 10: Soldado de apoyo.

Cada tipo de soldado estará limitado a un número de éstos. Con ello se pretende que no haya un desbalance en el juego. Por ejemplo, no se podrá permitir al jugador poder jugar con 4 soldados lanzacohetes y 2 médicos.

Por lo tanto, tendremos:

- 1 médico.
- 1 soldado de reconocimiento.
- 1 soldado lanzacohetes.
- 1 soldado de apoyo.
- 1 oficial.
- 1 soldado de infantería.

Es decir, un soldado de cada tipo. Así, de esta manera, se da una sensación de importancia para cada soldado de nuestro escuadrón. Y será una gran pérdida si, en algún momento de la batalla, causara baja uno de nuestras unidades.

Para poder hacer uso de las acciones del personaje éste debe tener puntos de acción con el coste correspondiente a cada una de ellas. Cada unidad, aliada o enemiga, tiene dos puntos de acción y dependiendo de la acción que decida el jugador usar, en cada momento, puede gastar un número determinado de puntos. Por ejemplo, el hecho de realizar la acción de disparar y poder atacar al enemigo hace que ese personaje gaste todos sus puntos de acción en lo que le resta de turno y deberá esperar al siguiente turno para poder realizar más acciones.

La implementación de estas acciones se ha realizado de manera que tenemos una acción base, de la cual heredan el resto de acciones. Otorgando, así, una mayor facilidad de incluir más acciones en un futuro, permitiendo una mayor accesibilidad a desarrollos de nuevas actualizaciones.

Dentro del código de la Acción Base incluimos el comportamiento para que la Inteligencia Artificial enemiga haga su propia elección y tome la decisión de utilizar la acción más acertada en cada momento. Para ello, buscamos por cada posición que tenga al alcance el personaje y comprobamos cuál es la opción que mejor se adecua a la situación y con la información que el personaje recoge del entorno obtiene una serie de puntuaciones y elije la opción más óptima.

Método para que los NPC elijan la opción más óptima

```
1  public EnemyAIAction GetBestEnemyAIAction(Difficult diff)
2  {
3      List<EnemyAIAction> enemyAIActionList = new List<EnemyAIAction>();
4
5      List<GridPosition> validActionGridPositionList = GetValidActionGridPositionList();
6
7      foreach(GridPosition gridPosition in validActionGridPositionList)
8      {
9          EnemyAIAction enemyAIAction = GetEnemyAIAction(gridPosition, diff);
10         enemyAIActionList.Add(enemyAIAction);
11     }
12
13     if(enemyAIActionList.Count > 0)
14     {
15         enemyAIActionList.Sort((EnemyAIAction a, EnemyAIAction b) => b.actionValue - a.
       actionValue);
16         return enemyAIActionList[0];
17     }
18     else
19     {
20         return null;
21     }
22 }
```

Vamos a proceder a ver, con más detalle, cada una de las posibles acciones que podemos realizar.

3.1. Acción de mover

La acción de mover es la que hace uso del Pathfinding y el Algoritmo A* que hemos estado comentando en puntos anteriores.

Para ello, lo que hacemos es determinar el camino más eficiente cuando el jugador elije la posición que desea. Y cuando lo hace, almacenamos todos los nodos por el que el personaje debe moverse para alcanzar la posición final y, ello, lo hacemos en una lista de posiciones. Con ella, el personaje recorre los nodos, de uno en uno, con el movimiento más eficiente posible.

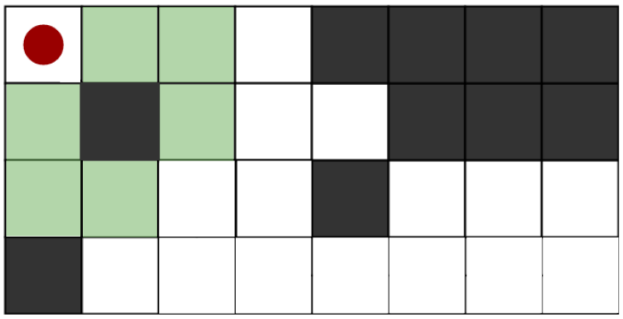

Figura 11: Acción para mover un personaje

Podemos encontrar en la imagen que las posiciones con cuadrados de color gris oscuro/negro son obstáculos. Y es, por ello, que nuestro sistema al detectar una posición como obstáculo no nos la muestra como posible alternativa para nuestro personaje.

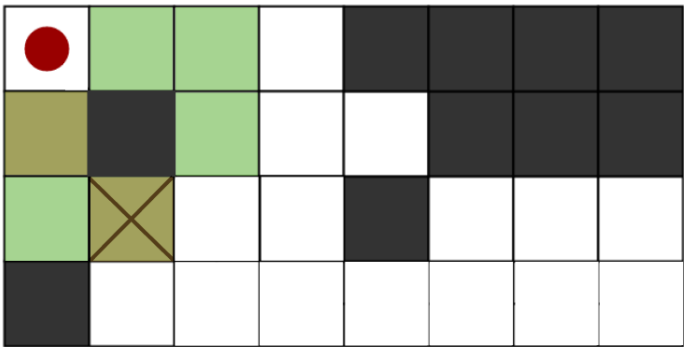

Figura 12: Pathfinding

De esta forma, en nuestra lista de nodos tendríamos dos posiciones por las cuáles el personaje se movería y, de esa manera, obtendríamos el mejor camino posible.

En cuanto al coste de puntos de acción el personaje solamente gastaría uno, independientemente del movimiento que haga. Es decir, si realiza un movimiento de dos "cuadrados" gastará los mismos puntos que si hace el máximo movimiento posible.

Para obtener la lista de posiciones válidas primero comprobamos que la unidad tenga puntos de acción disponibles. En ese caso, comprobará, a lo largo y a lo ancho, la distancia que le indicamos, siendo en este caso "maxMoveDistance = 7". Dentro de cada posición comprobamos si es una posición válida, para ello tendremos en cuenta para todas:

- Si se encuentra dentro de los límites del mapa.
- Si no es la misma posición que la actual.
- Si no hay ninguna unidad en la posición.

En caso de que sea válida, comprobamos a través del algoritmo de búsqueda de caminos si es una posición a la que se puede acceder o si hay algún obstáculo en ella. Si consigue pasar las diferentes pruebas, obtenemos todos las posibles posiciones donde se puede mover el personaje.

Código que muestra las posiciones válidas para poder realizar movimientos

```
1  public override List<GridPosition> GetValidActionGridPositionList()
2  {
3      List<GridPosition> validGridPositionList = new List<GridPosition>();
4
5      GridPosition unitGridPosition = unit.GetGridPosition();
6
7      if(unit.GetActionPoints() > 0)
8      {
9          for (int x = -maxMoveDistance; x <= maxMoveDistance; x++)
10         {
11             for (int z = -maxMoveDistance; z <= maxMoveDistance; z++)
12             {
13                 GridPosition offSetGridPosition = new GridPosition(x, z);
14                 GridPosition testGridPosition = unitGridPosition + offSetGridPosition;
15                 if (LevelGrid.Instance.IsValidGridPosition(testGridPosition)
```

```
16                  || unitGridPosition != testGridPosition
17                  || !LevelGrid.Instance.HasAnyUnitOnGridPosition(testGridPosition))
18              {
19                  if (Pathfinding.Instance.IsWalkableGridPosition(testGridPosition)
20                  && Pathfinding.Instance.HasPath(unitGridPosition, testGridPosition))
21                  {
22                      int pathFindingDistanceMultiplier = 10;
23                      if (Pathfinding.Instance.GetPathLength(unitGridPosition, testGridPosition) <=
      maxMoveDistance * pathFindingDistanceMultiplier)
24                      {
25                          validGridPositionList.Add(testGridPosition);
26                      }
27
28                  }
29              }
30          }
31      }
32  }
33  return validGridPositionList;
34 }
```

3.2. Acción de disparar y recarga

Otra de las acciones principales de nuestro videojuego es la posibilidad de poder eliminar a nuestros enemigos. Para ello, cada soldado tendrá una acción para poder disparar su arma, dependiendo de cuál sea ésta. Por ejemplo, el soldado antitanque no dispone de un arma de fuego y, en su caso, podrá disparar su lanzacohetes y no un arma convencional.

La acción de disparar dispone de un rango determinado que no tiene un alcance de X posiciones en una dirección, sino que tiene un rango que se asemeja más a un triángulo que a un cuadrado para que sea un poco más visual y no se base todo en cuadrados. Podemos ver un ejemplo en la figura siguiente donde el color rojo, con tono ligero, indica el rango en el que el personaje puede disparar.

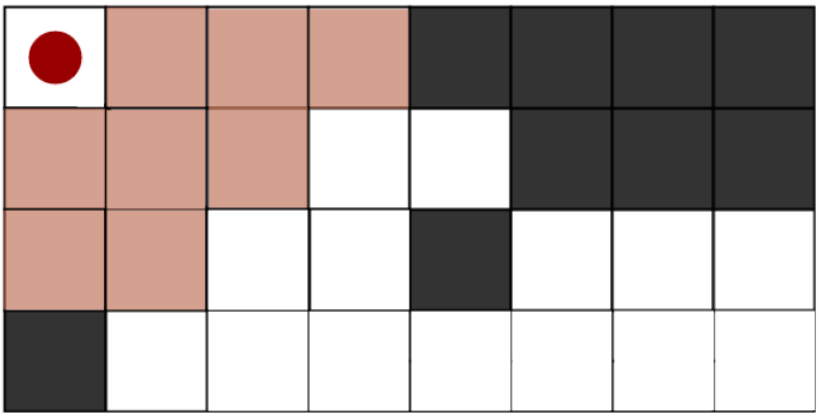

Figura 13: Rango de disparar

De igual manera que ocurre en la acción de movimiento, todas las acciones tienen en cuenta la posición de los obstáculos y, por tanto, no permiten los disparos, igual que no se puede disparar si no se encuentra ningún enemigo. Solo se podrá realizar el disparo en aquellos nodos en los que se encuentra un enemigo y estén al alcance de la unidad.

El disparo a un enemigo no siempre será acertado. Esto es debido a que, para dar una mayor sensación de dificultad y de aleatoriedad, se ha incluido un sistema de porcentajes que contiene diferentes variables a tener cuenta a la hora de disparar, como pueden ser la distancia, la cobertura, flanqueo, supresión e incluso la dificultad en la que nos encontremos. Estos porcentajes se verán con más detalles en puntos posteriores.

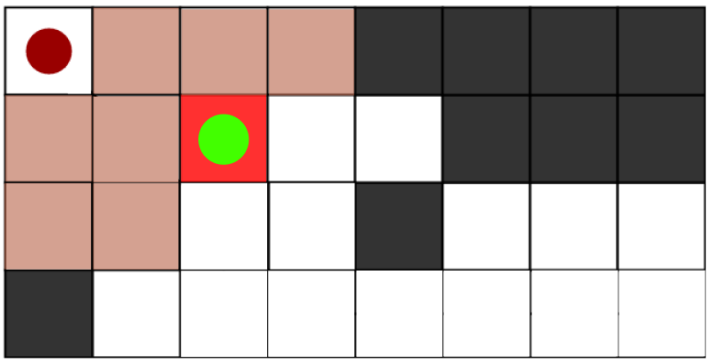

Figura 14: Disparo al enemigo

El disparo al enemigo siempre quitará la misma cantidad de vida a éste, en caso de que el disparo sea acertado. En cambio, si en vez del disparo, realizamos otras acciones

que tengan que ver con el arma de fuego, éstas no tienen por qué tener el mismo daño. Esto lo veremos en la acción de supresión.

Nuestro sistema realizará diferentes comprobaciones a la hora de verificar qué posiciones son válidas para poder realizar el disparo. Por ejemplo, no se podrá disparar a través de paredes que cubran por completo al enemigo dado que, de manera realista, no tendríamos visual completa de éste y, por tanto, no podríamos disparar al mismo. Por ello, se lanzarán diferentes "Raycast" para comprobar que podemos ver al enemigo y, por tanto, podemos disparar. Esto lo podemos ver en la siguiente figura.

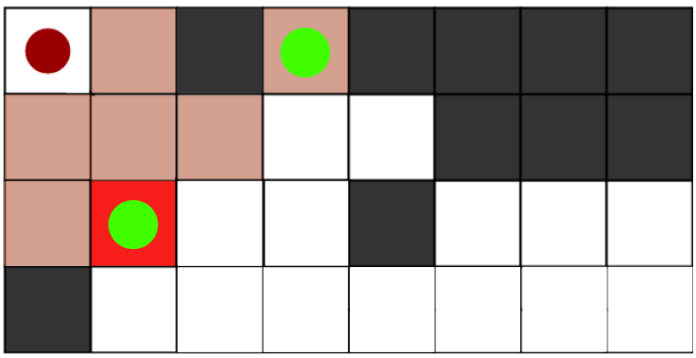

Figura 15: Visual del enemigo

El método por el cual obtenemos las posiciones válidas comparte algunas semejanzas con el resto de acciones. Como hemos explicado en la acción de mover, comprobamos los diferentes aspectos:

- La posición dentro de los límites del mapa.
- Si la posición incluye una unidad.
- Si la distancia es menor a la distancia máxima de disparo.
- Si la unidad a la que se dispara no pertenece al mismo equipo.
- Si la unidad que dispara tiene visual completa del enemigo al que dispara.
- Si la unidad que se encuentra en la posición pertenece el equipo enemigo.

Además, en este método, añadimos un nuevo argumento llamado "GetPercent" que lo usamos para poder incluir una de las funcionalidades nuevas que hemos tratado y es

que, en primer lugar, las unidades enemigas tendrán una posición de alerta. Es decir, las animaciones que estarán realizando serán diferentes, con una actitud más tranquila en el momento en el que se encuentren al alcance de alguna unidad enemiga. En este caso, éstos se alertarán y mostrarán unas animaciones diferentes. Por ello, dado que usamos este método para comprobar si están al alcance, usamos este valor booleano para no mostrar los porcentajes en ese momento de alerta. En caso de que el argumento de "GetPercent" sea true mostraremos los porcentajes, en caso contrario, no lo mostraremos.

```
1  public List<GridPosition> GetValidActionGridPositionList(GridPosition unitGridPosition, bool
       GetPercent)
2  {
3      List<GridPosition> validGridPositionList = new List<GridPosition>();
4      if(unit.GetActionPoints() > 0)
5      {
6          for(int x = -maxShootDistance; x <= maxShootDistance; x++)
7          {
8              for(int z = -maxShootDistance; z <= maxShootDistance; z++)
9              {
10                 GridPosition offSetGridPosition = new GridPosition(x, z);
11                 GridPosition testGridPosition = unitGridPosition + offSetGridPosition;
12                 if(LevelGrid.Instance.IsValidGridPosition(testGridPosition)
13                     || LevelGrid.Instance.HasAnyUnitOnGridPosition(testGridPosition))
14                 {
15                     int testDistance = Mathf.Abs(x) + Mathf.Abs(z);
16                     if (testDistance <= maxShootDistance)
17                     {
18                         Unit targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(testGridPosition);
19                         if (targetUnit.IsEnemy() != unit.IsEnemy())
20                         {
21                             float unitShoulderHeight = 2f;
22                             Vector3 unitWorldPosition = LevelGrid.Instance.GetWorldPosition(
       unitGridPosition);
23                             Vector3 shootDir = (targetUnit.GetWorldPosition() - unitWorldPosition).
       normalized;
24                             if (Physics.Raycast(unitWorldPosition + Vector3.up * unitShoulderHeight,
       shootDir, Vector3.Distance(unitWorldPosition, targetUnit.GetWorldPosition()),
25                                 obstacleLayerMask))
26                             {
```

```
                        continue;
                    }
                    if (GetPercent)
                    {
                        if (!Physics.Raycast(unitWorldPosition + Vector3.up * unitShoulderHeight,
                        shootDir, Vector3.Distance(unitWorldPosition, targetUnit.
    GetWorldPosition()),
                        coverLayerMask) && targetUnit.GetCoverType() != CoverType.None)
                        {
                            if (targetUnit.IsEnemy())
                            {
                                AddUnitToFlankedUnits(targetUnit);
                                targetUnit.ShowHitPercent(GetHitChance(targetUnit), true);
                            }
                        }
                        else
                        {
                            if (targetUnit.IsEnemy())
                            {
                                targetUnit.ShowHitPercent(GetHitChance(targetUnit), false);
                            }
                        }
                    }
                    validGridPositionList.Add(testGridPosition);
                }

            }

        }

    }
}
return validGridPositionList;
}
```

La acción de disparar lleva emparejada otra de las acciones principales, el poder recargar las armas. Se ha implementado un sistema de munición por el cual todos los personajes que dispongan de armas de fuego tienen cuatro disparos antes de que tengan la necesidad

de recargar si quisiesen disparar otra vez. Esto no quiere decir que tengan que esperar a realizar los cuatro disparos mencionados para poder recargar. La acción de recargar es una acción individual independiente de la acción de disparar. Por ello, el jugador podrá decidir, en todo momento de la partida, cuando quiere recargar. Si bien gastará los puntos de acción que tenga disponibles en ese momento.

Dicha acción de recargar también está contemplada, aunque el personaje no disponga de armas de fuego, es decir, para el soldado lanzacohetes. Pero éste tendrá que recargar cada vez que realice un disparo ya que el lanzacohetes solo dispone de un cohete para poder seguir disparando.

3.3. Acción de supresión

La acción de supresión tiene diferentes utilidades, siendo útil también para aplicar daños a los enemigos. Realmente su principal función es la de aplicar diferentes reducciones a las estadísticas de los enemigos, en este caso, a la puntería del personaje que queda suprimido. Para esta acción el comportamiento es similar a la del disparo, pero tiene un menor porcentaje de acierto, aunque, por otra parte, siempre le aplicará el estado de supresión al enemigo cuando le dispare, obteniendo éste un -0.134 de porcentaje de acierto en el siguiente turno.

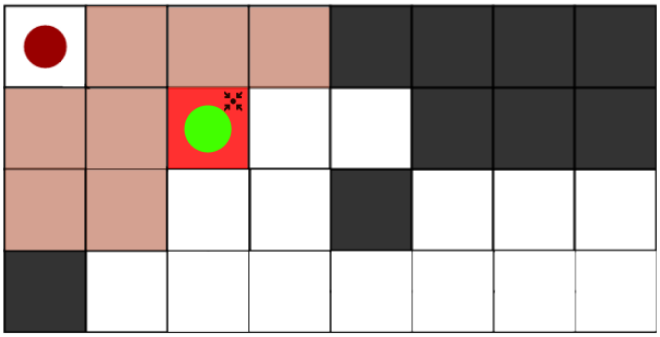

Figura 16: Disparo de supresión

3.4. Acción de curar

La labor del médico es muy importante ya que será el único capacitado para poder curar a sus compañeros. No tendrá un límite de curas, pero no, por ello, curará una gran

cantidad de puntos de vida. Para poder curar éste deberá estar cerca del compañero que esté herido. De esa manera, podrá aplicar el vendaje al mismo.

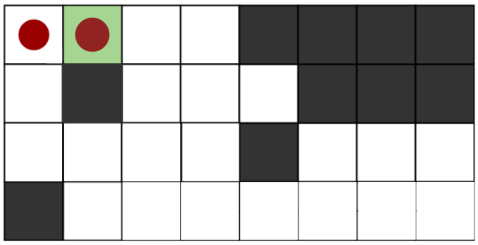

Figura 17: Acción de curar

3.5. Acción de revivir

De manera similar a la acción de curar, el médico será el único capacitado para poder revivir a los aliados. Siempre que uno de nuestros personajes alcance una vida de 0 o inferior, dicho personaje quedará incapacitado y, por ello, necesitará que sea reanimado por un médico. De esta manera, se podrá volver a manejar a dicho personaje. En el tiempo en el que esté incapacitado no se podrá realizar ninguna acción con éste. Igual que en la acción de curar, el médico requiere estar en una posición próxima al soldado para poder reanimarlo.

3.6. Acción de lanzar granada

La acción del lanzamiento de granadas se hace mediante el uso de **Curvas de Animación**. Dichas curvas sirven para controlar cómo cambian las propiedades por el tiempo. Con esto controlamos cómo se dibuja la parábola que realiza la granada y cuál será su trayectoria para acabar en el destino final.

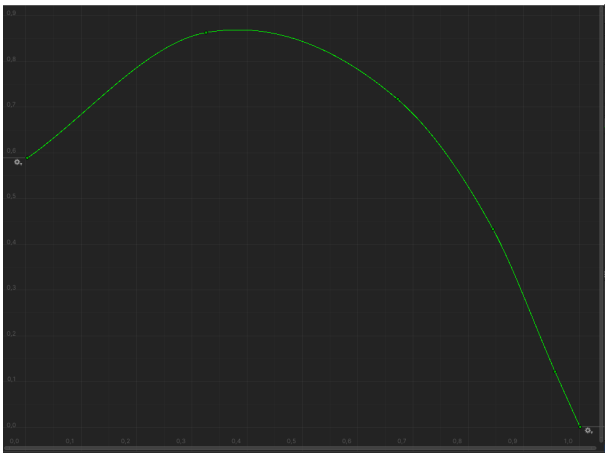

Figura 18: Curva de Animación

Si no aplicásemos dicha curva de animación la granada se movería en una línea recta hasta la posición indicada y aplicaría el mismo comportamiento. Pero éste no sería el aspecto visual que queremos en una granada.

Esta curva de animación se encuentra con números normalizados. Es decir, trabajaremos con números que se encuentren dentro del intervalo [0, 1] que servirán para manejar la altura a la que se encuentra la granada en el trayecto que ésta realiza. En primer lugar, calcularemos la distancia entre el personaje que lanza la granada y el objetivo y, en todo momento, normalizaremos dicho número. Esto lo haremos dividiendo la distancia actual en la que se encuentre la granada en X posición entre la distancia total.

distanciaNormalizada = 1 - (distancia / distanciaTotal)

Y con esto, evaluaremos la curva de animación y se aplicará la parábola que le hemos indicado. Podremos probar distintos tipos de gráficas hasta encontrar una que se adapte a lo que necesitásemos.

La granada tendrá un rango, que visto en nuestro sistema, será de un alcance de 3x3 cuadrados en el sistema de mallas. Por ello, cuando la granada alcanza el suelo comprobamos todas las unidades que se encuentran dentro del rango y si existe alguna unidad, ya sea aliada o enemiga, ésta recibirá el daño pertinente.

Movimiento de la granada a lo largo de su trayectoria

```
1  private void Update()
2  {
```

```
        Vector3 moveDir = (targetPosition - positionXZ).normalized;
        float moveSpeed = 15f;
        positionXZ += moveDir * moveSpeed * Time.deltaTime;
        float distance = Vector3.Distance(positionXZ, targetPosition);
        float distanceNormalized = 1 - distance / totalDistance;
        float maxHeight = totalDistance / 4f;
        float positionY = arcYAnimationCurve.Evaluate(distanceNormalized) * maxHeight;
        transform.position = new Vector3(positionXZ.x, positionY, positionXZ.z);
        Vector3 hitBoxSize = new Vector3(3,3,3);
        float reachedTargetDistance = .2f;
        if(Vector3.Distance(positionXZ, targetPosition) < reachedTargetDistance)
        {
            Collider[] colliderArray = Physics.OverlapBox(targetPosition, hitBoxSize);//Para saber
    que enemigos hay dentro del radio de la granada.
            foreach(Collider collider in colliderArray)
            {
                if(collider.TryGetComponent<Unit>(out Unit targerUnit))
                {
                    targerUnit.Damage(30);
                }
                if(collider.TryGetComponent<DestructibleCrate>(out DestructibleCrate
    destructibleCrate))
                {
                    destructibleCrate.Damage();
                }
            }
            OnAnyGrenadeExploded?.Invoke(this, EventArgs.Empty);
            trailRenderer.transform.parent = null;
            Instantiate(grenadeExplodeVfxPrefab, targetPosition + Vector3.up * 1f, Quaternion.
    identity);
            Destroy(gameObject);
            OnGrenadeBehaviourComplete();
        }
    }
```

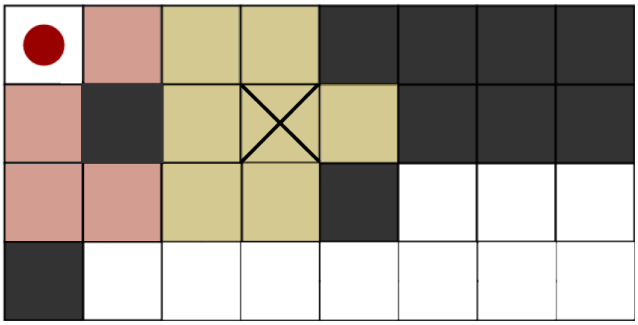

Figura 19: Acción de lanzar granada

3.7. Acción de lanzar misil

El lanzamiento del misil funciona de manera similar en lo que al comportamiento de impactar el misil se refiere. Aplicará el mismo daño que aplican las granadas, pero, en este caso, al ser un lanzacohetes lo que propulsa el misil tendrá un mayor rango.

El soldado lanzacohetes será el único capaz de utilizar esta acción ya que sólo él dispone del armamento. Por ello, su acción de recargar no será la munición normal que recargaría si fuese un soldado con arma de fuego, sino que le recargará el lanzacohetes al personaje para que éste pueda seguir disparando.

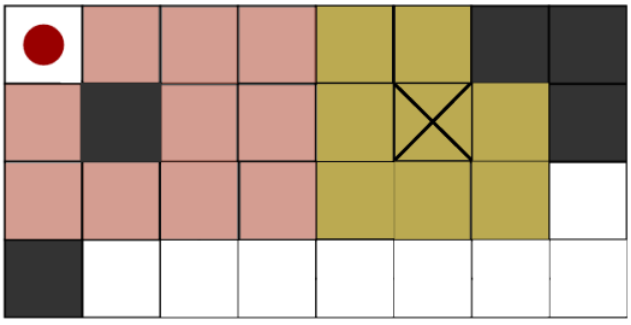

Figura 20: Acción de lanzar misil

3.8. Acción de ataque cuerpo a cuerpo

El ataque cuerpo a cuerpo requerirá que el personaje esté pegado al enemigo para poder realizar la acción. Por ello, eliminará al enemigo de un solo golpe. Solo se podrá realizar esta acción cuando tengamos a un enemigo a la distancia de 1, es decir, cuando se encuentre cerca de él, en otro caso no podremos hacer nada. Lo mismo ocurre con las

acciones de disparar armas de fuego, éstas se deben de encontrar a la distancia designada para poder realizar la acción, en otro caso no podrá disparar.

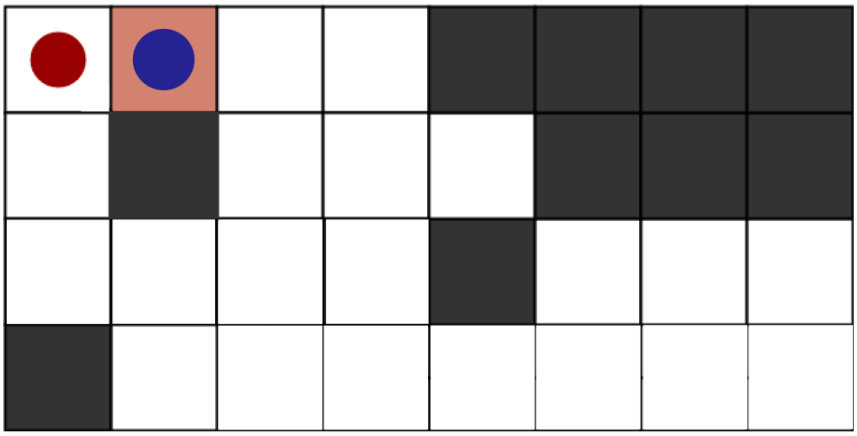

Figura 21: Acción de ataque cuerpo a cuerpo

3.9. Acción de Interactuar

La acción de interactuar será la acción principal para poder cumplir los objetivos.

Los personajes aliados podrán interactuar con diferentes objetos del mapa para poder cumplir los objetivos de la misión, solo con éstos serán con los que se puede interactuar. Y deberán estar en un cuadrado contiguo para poder realizar la acción, saliendo el cuadrado de color azul cuando pueda interactuar con algo.

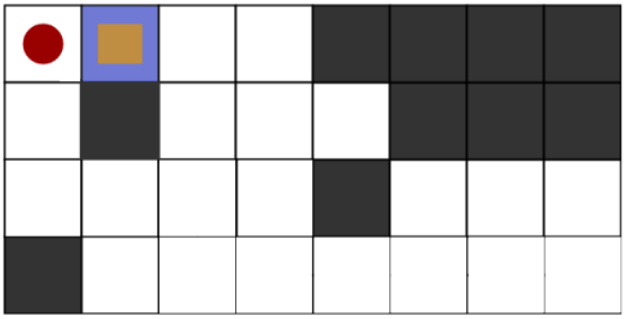

Figura 22: Acción de Interactuar

3.10. Acciónes Especiales

Podemos encontrar diferentes acciones especiales. Dichas acciones son únicas y solo se podrán encontrar en los personajes a los que les corresponda dicha acción, siendo éstas:

- Disparo de precisión.
- Bonus de acción.

El disparo de precisión será una habilidad única que tendrá disponible el soldado de reconocimiento, ya que es el único capaz de realizar un disparo de tal alcance. La función de esta acción es la de otorgar mayor alcance al disparo y tener una mayor probabilidad de acertar el disparo.

En cambio, el bonus de movimiento será una habilidad disponible en el oficial del pelotón y es capaz de otorgar un punto de acción adicional al personaje que desee. Pero, para esto, debe de encontrarse dentro del rango de alcance.

4. Creación e implementación de un sistema de porcentajes y cobertura

Como en la mayoría de juegos de este estilo, se han incluido porcentajes y coberturas para dar una mayor sensación de aleatoriedad. Estos porcentajes están aparejados a las probabilidades que tenemos de acertar un disparo a los enemigos y dependerá de diferentes aspectos a tener en cuenta, éstos pueden ser la cobertura, supresión, flanqueo...

Los porcentajes se aplicarán a todos los posibles disparos que se realicen a lo largo de la misión. En cambio, las granadas o los lanzamientos de cohetes no contarán con este tipo de probabilidades ya que no tendría sentido.

4.0.1. Coberturas

En primer lugar, hablaremos del sistema de coberturas que hemos incluido. Este sistema se basa en la posición del jugador en el mapa. Para ello, hemos creado tres posibles estados en los que se puede encontrar el personaje en un punto dado. Estos tres estados pueden ser:

- Al descubierto.
- Media cobertura.
- Cobertura completa.

Para decidir qué tipo de cobertura tiene un objeto le aplicamos el script por el cual decidimos que clase va a tener. Y, una vez aplicado, cuando el personaje se coloca en una de las posiciones contiguas al objeto le aplicaríamos el bonus que obtendría cuando le disparasen otorgándole bonificaciones y reduciendo, de esta manera, el porcentaje de aciertos de los disparos del enemigo. Las posiciones válidas de una cobertura serán las verticales y horizontales al objeto, no se estará en cobertura si nos encontramos en dirección diagonal al objeto.

Para obtener el tipo de cobertura de la unidad primero comprobamos qué posiciones están dentro de los límites del mapa y, en el caso de que esté dentro de los límites, vamos

uno por uno por las posiciones para comprobar si hubiese algún objeto con cobertura y comprobamos qué nivel tendría dicha cobertura.

Detección de cobertura hacia los personajes.

```
public CoverType GetUnitCoverType(GridPosition gridPosition)
  {
      int x = gridPosition.x;
      int z = gridPosition.z;

      GridPosition leftPosition = new GridPosition(x−1, z);
      GridPosition rightPosition = new GridPosition(x+1, z);
      GridPosition upPosition = new GridPosition(x, z+1);
      GridPosition downPosition = new GridPosition(x, z−1);

      bool validLeft = gridSystem.IsValidGridPosition(leftPosition);
      bool validRight = gridSystem.IsValidGridPosition(rightPosition);
      bool validUp = gridSystem.IsValidGridPosition(upPosition);
      bool validDown = gridSystem.IsValidGridPosition(downPosition);

      CoverType leftCover = CoverType.None;
      CoverType rightCover = CoverType.None;
      CoverType upCover = CoverType.None;
      CoverType downCover = CoverType.None;

      if (validLeft)
      {
      if (gridSystem.GetGridObject(leftPosition).GetCoverObject() != null)
            leftCover = gridSystem.GetGridObject(leftPosition).GetCoverObject().GetCoverType
    ();
      }
      if(validRight)
      if (gridSystem.GetGridObject(rightPosition).GetCoverObject() != null)
            rightCover = gridSystem.GetGridObject(rightPosition).GetCoverObject().
    GetCoverType();
      if (validUp)
      if (gridSystem.GetGridObject(upPosition).GetCoverObject() != null)
            upCover = gridSystem.GetGridObject(upPosition).GetCoverObject().GetCoverType();
      if (validDown)
         if (gridSystem.GetGridObject(downPosition).GetCoverObject() != null)
```

```
34              downCover = gridSystem.GetGridObject(downPosition).GetCoverObject().
    GetCoverType();
35
36      if(leftCover == CoverType.Full || rightCover == CoverType.Full
37          || upCover == CoverType.Full || downCover == CoverType.Full)
38
39          return CoverType.Full;
40
41      if (leftCover == CoverType.Half || rightCover == CoverType.Half
42          || upCover == CoverType.Half || downCover == CoverType.Half)
43
44          return CoverType.Half;
45      return CoverType.None;
46
47  }
```

En el caso de que alguna de las cuatro posiciones tenga una cobertura le aplicamos dicho nivel. En primer lugar, comprobamos las coberturas completas para darle prioridad en caso de que haya dos niveles distintos.

Podremos ver qué tipo de cobertura tiene cada personaje, en su momento, con el icono que tenga éste a su lado. Los iconos son como estos, para la media cobertura y la cobertura completa, respectivamente.

Figura 23: Media Cobertura

Figura 24: Cobertura Completa

Estas coberturas otorgarán un bonus de protección, pero también otorgarán un bonus al enemigo cuando nos disparen y nos encontremos en un lugar sin cobertura. Además,

las unidades podrán flanquear a los enemigos otorgando un mayor porcentaje de acierto al jugador o al enemigo en caso de disparar.

4.0.2. Porcentajes

A continuación, vamos a mostrar los porcentajes que se pueden considerar.

Estos son los aumentos de porcentajes que obtenemos cuando el personaje se encuentra en cualquier tipo de cobertura. Cuando nos referimos a la disminución de porcentaje queremos indicar que será más complicado acertar disparos a estos personajes ya que se encuentran en cobertura. Y, en caso de aumento de porcentaje, nos referimos a que es más fácil acertar disparos a estos mismos.

Estos valores han sido escogidos de manera aleatoria, probando diferentes tipos de ellos dentro del juego. Es decir, indicabábamos un valor y probábamos, de manera interna, si éste era aceptable o era demasiado alto o bajo y, por tanto, se ha ido probando de manera práctica.

Adicionalmente, también hemos incluido milésimas para que el porcentaje fuese un poco más natural y no tanto números enteros como pueden ser el 50 % o el 25 % sino que éstos fuesen por ejemplo 23,456 % y, por tanto, la suma o la resta de algún porcentaje contase un poco más.

- Cobertura Completa.
  - Disminución de porcentaje de: 0.278.
- Media Cobertura.
  - Disminución de porcentaje de: 0.152.
- Sin cobertura.
  - Aumento de porcentaje de: 0.1342.
- Flanqueo a la cobertura.
  - Aumento de porcentaje de: 0.324.

Todos estos porcentajes se tendrán en cuenta y sumarán o restarán al total del porcentaje inicial, que se basará en la distancia a la que se encuentra del enemigo. Para ello, calculamos el número de cuadrados y reducimos 0.0843 por cada uno de ellos. A partir de este número comenzamos los cálculos de probabilidades.

```
1   private float GetHitChance(Unit targetUnit)
2   {
3       int distance = CalculateDistance(targetUnit.GetGridPosition());
4       float totalPercent = 0.92f;
5
6       float distanceReduction = 0.0843f;
7       float noneCoverIncrease = 0.1342f;
8       float halfCoverReduction = 0.152f;
9       float suppressedReduction = 0.134f;
10      float fullCoverReduction = 0.278f;
11      float flankedIncrease = 0.324f;
```

También se ha incluido un aumento de porcentajes en caso de que se elijan dificultades inferiores a la dificultad difícil, para el caso de que el jugador decida tener una menor dificultad ayudarle en ese sentido. Estos porcentajes van en relación a la distancia del personaje con el enemigo y los porcentajes por escudos que hemos creado para el caso de que el jugador no consiga acertar ningún tiro de manera reiterada.

En el caso de que se elija la dificultad fácil y normal la reducción de probabilidades por distancia se aplicará solo a los enemigos. En este caso, en vez de aplicar una reducción de 0.0843 por cuadrado se reducirá 0.12 para las partidas en dificultad fácil y 0.10 en dificultad media. Los porcentajes por distancia se quedarán iguales a los disparos de los personajes aliados, permitiendo un poco de ventaja y dándole una mayor facilidad al juego.

Código para aumentar la reducción a los enemigos si están a mayor distancia en dificultades más fáciles

```
1   if (unit.IsEnemy())
2   {
3       switch (EnemyAI.Instance.GetDifficult())
4       {
5           case Difficult.Easy:
```

```
            distanceReduction = 0.12f;
            break;
        case Difficult.Medium:
            distanceReduction = 0.10f;
            break;
    }
}
totalPercent -= distance * distanceReduction;
```

	Enemigos	Aliados
Dificultad Fácil	0.12	0.0843
Dificultad Media	0.10	0.0843
Dificultad Díficil	0.0843	0.0843

Figura 25: Reducciones de porcentaje por distancia

También hemos aplicado diferentes ayudas para mantener que las dificultades menores sean más fáciles. Hemos aplicado un "escudo" que proporciona una disminución de porcentaje por cada disparo que se le aplique al aliado. Es decir, cuando un enemigo dispara a un aliado y acierta el disparo le aplicará un escudo virtual por el cual el próximo disparo que se realice sobre este personaje tenga una menor probabilidad de ser acertado. Esto permite que, en dificultades menores, el jugador se pueda permitir cometer más errores y no eliminen a sus personajes debido a ello. Igual que con las distancias, el porcentaje que disminuimos va en relación a la dificultad elegida. Esto solo se mantendrá durante el mismo turno en el que han sido atacados, no se mantendrá a lo largo de la partida.

Cálculo de escudos y ayudas al disparar

```
if (calculateChance(hitChance))
{
//Si calculateChance es true, el disparo es acertado
    unit.SetHitHelper(0);
    if (unit.IsEnemy())
    {
    //En caso de ser unidad aliada la que recibe el disparo, le otorgamos escudo
        switch (EnemyAI.Instance.GetDifficult())
```

```
9           {
10              case Difficult.Easy:
11                  targetUnit.SetShieldHelper(targetUnit.GetShieldHelper() + 10);
12                  targetUnit.StartRecordTurn();
13                  break;
14              case Difficult.Medium:
15                  targetUnit.SetShieldHelper(targetUnit.GetShieldHelper() + 5);
16                  targetUnit.StartRecordTurn();
17                  break;
18          }
19      }
20      Shoot();
21  }
22  else
23  //Disparo fallido
24  {
25  //En caso de que sea unidad aliada, aplicamos ayuda de porcentaje
26      if (!unit.IsEnemy())
27      {
28          switch (EnemyAI.Instance.GetDifficult())
29          {
30              case Difficult.Easy:
31                  unit.SetHitHelper(unit.GetHitHelper() + 10);
32                  break;
33              case Difficult.Medium:
34                  unit.SetHitHelper(unit.GetHitHelper() + 5);
35                  break;
36          }
37      }
38      //Instanciamos el missPrefab
39      onAnyMissShoot?.Invoke(this, targetUnit);
40  }
```

Podemos ver en la figura que, en el caso de dificultad fácil, disminuiremos el porcentaje de acierto enemigo en diez. En cambio, en el caso de dificultad media lo reduciremos en cinco.

	Aliados
Dificultad Fácil	10
Dificultad Media	5
Dificultad Díficil	0

Figura 26: Reducciones por impactos

De manera similar, tenemos una ayuda que se aplica cuando el jugador tiene la mala suerte de fallar disparos reiteradamente. Por ello, cuando uno de sus personajes falla un disparo le aplicamos un bonus que se activará para el siguiente disparo que éste aplique, otorgándole, de esta manera, un mayor porcentaje en el siguiente disparo. Esto no se eliminará en cada turno, seguirá aumentando hasta que el personaje consiga acertar un disparo. Por tanto, se podría dar el caso de que el personaje pueda seguir fallando disparos incluso después de haber realizado el disparo y haber recibido el bonus del que hablamos.

Los porcentajes que reciben son los mismos que hemos comentado con el escudo, recibiendo un aumento de diez en dificultades fáciles y cinco en dificultades medias.

Parte final del cálculo del porcentaje, aplicando la cobertura y los escudos

```
1
2   switch (targetUnit.GetCoverType())
3   {
4       case CoverType.None:
5           totalPercent += noneCoverIncrease;
6           break;
7       case CoverType.Half:
8           totalPercent -= halfCoverReduction;
9           break;
10      case CoverType.Full:
11          totalPercent -= fullCoverReduction;
12          break;
13  }
14
15  if (unit.GetIsSupressed())
16      totalPercent -= suppressedReduction;
17
```

```
18      if (unitsFlanked.Contains(targetUnit))
19      {
20          totalPercent += flankedIncrease;
21      }
22
23      if(unit.GetHitHelper() > 0)
24      {
25          totalPercent += unit.GetHitHelper() / 100;
26      }
27
28      if(targetUnit.GetShieldHelper() > 0)
29      {
30          totalPercent -= targetUnit.GetShieldHelper() / 100;
31      }
32
33      if (totalPercent > 1f)
34          return 1;
35      else
36          return totalPercent;
37      }
```

Además de estas ayudas, que van en relación a lo que ocurre dentro de la partida, existe un multiplicador final donde se multiplica por el porcentaje final que se obtiene sobre el enemigo. Este cálculo de multiplicar se realizará una vez calculemos el porcentaje final y será con éste con el que comparemos para ver si el disparo es acertado o no.

Cálculo final para la decisión del disparo

```
1  private bool calculateChance(float hitChance)
2  {
3      float chance = UnityEngine.Random.Range(0f, 1f);
4
5      Difficult difficulty = EnemyAI.Instance.GetDifficult();
6      float easyDiff = 1.15f;
7      float mediumDiff = 1.075f;
8
9      if (targetUnit.IsEnemy())
10     {
11         switch (difficulty)
```

```
12          {
13              case Difficult.Easy:
14                  hitChance *= easyDiff;
15                  break;
16              case Difficult.Medium:
17                  hitChance *= mediumDiff;
18                  break;
19          }
20      }
21      return chance < hitChance;
22  }
```

	Aliados
Dificultad Fácil	1.15
Dificultad Media	1.075
Dificultad Díficil	1

Figura 27: Multiplicador de porcentaje

5. Inteligencia Artificial

La Inteligencia Artificial ha avanzado, de manera gigantesca, en todos los sentidos, no solo en los videojuegos. Pero, en estos últimos, se han comenzado a ver diferentes versiones de Inteligencia Artificial que podemos aplicar en ellos, las cuales nos permiten tener NPCs (Non- player character) con una mayor inteligencia que la que podrían tener hace diez años.

En nuestro caso, para el videojuego hemos realizado una Inteligencia Artificial para que simule el comportamiento de los enemigos cuando sea el turno del contrincante. Estos NPCs serán capaces de tomar la mejor decisión que puedan para que el combate sea un desafío para el jugador o, en caso de que se elijan dificultades menores, intentará tomar una de las mejores opciones que tenga, sin necesidad de que ésta sea la mejor.

Existen diferentes tipos de Inteligencia Artificial que se podrían utilizar para el comportamiento de los personajes en un videojuego de estrategia por turnos como es el nuestro.

En nuestro caso, hemos elegido que la Inteligencia Artificial, que hemos creado, se base en la conocida IA de utilidad, la cual se basa en la puntuación de las posibles acciones y que veremos, con mayor detalle, a continuación.

Además de esta IA de utilidad también hemos desarrollado el algoritmo de búsqueda para que obtengamos, de manera óptima, el camino más corto para que un personaje llegue desde un punto origen a otro punto destino, teniendo en cuenta el terreno y los obstáculos. Este algoritmo que hemos desarrollado está basado completamente en el Algoritmo A*.

5.0.1. Otras Inteligencias Artificiales

Además de la Inteligencia Artificial, que hemos elegido, también hemos tenido en cuenta el comportamiento de otras como pueden ser los Árboles de Decisión o las Máquinas de Estados. Aunque ésta última no se asemeja tanto al comportamiento que deseamos ya que los NPCs tendrán que buscar la mejor solución a los problemas y, para ello, hemos tenido que realizar un estudio para ver qué inteligencia se asemeja mejor a lo que estábamos buscando entre los **Árboles de Decisión** y la **IA de utilidad.**

Un Árbol de Decisión es un modelo matemático para ejecutar un objetivo. Además de ser usado en los videojuegos se utiliza en otros campos importantes como pueden ser

la robótica o los campos de computación científica. El punto álgido de los Árboles de Decisión es su facilidad para poder completar objetivos complejos a partir de acciones sencillas. Su compresión es sencilla y, por ello, se volvieron muy populares en el ámbito de los videojuegos. Y es que permite crear el comportamiento de los NPCs con una estructura en árbol y cuyos nodos son las acciones, con los cuales se determina la mejor actuación del personaje.

Por ello, al ser intuitivo, fácil de diseñar y probar, junto con que tiene una mayor escalabilidad que otras posibilidades, nos permite añadir mayor número de acciones en el juego sin necesidad de cambiar todas las posibles acciones que hayamos incluido hasta el momento.

Vamos a comentar la estructura de los Árboles de Decisión. Tienen tres tipos de nodos que se clasifican en nodo raíz, nodos de control y nodos de ejecución:

- El nodo raíz no tiene padre, como en cualquier tipo de árbol, y un solo hijo.
- Los nodos de control tienen un padre y, al menos, un nodo hijo.
- Los nodos de ejecución tienen un padre también pero no tienen hijos, siendo éstos la última parte del árbol.

Los nodos de ejecución, a su vez, pueden tener diferentes estados en sí mismos. Y es que dependiendo de la decisión del Árbol, los nodos se pueden encontrar en el estado de: **En proceso**, **Éxito** y **Fracaso**.

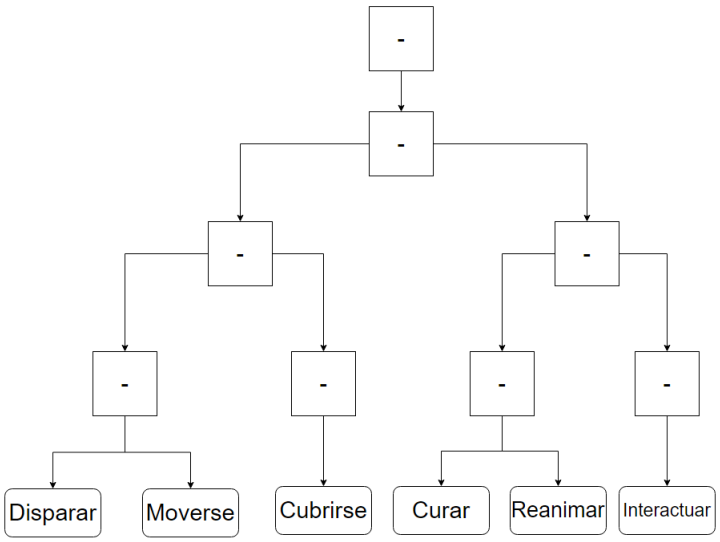

Figura 28: Árbol de decisión

Podemos ver un ejemplo de un Árbol de Decisión. El problema de estos Árboles es que se han de realizar de manera manual. Es decir, es el propio desarrollador el que tiene que tener en cuenta los posibles escenarios que se pueden dar dentro del juego y es, por ello, que se pueden dar casos dentro del juego que, a lo mejor, no se han tenido en cuenta.

A lo largo de los años y de las diferentes implementaciones que se han realizado sobre los Árboles de Decisión se han ido aumentando la eficiencia y la capacidad para satisfacer las demandas de la industria, las cuales cada vez son mayores. Los Árboles de Decisión han evolucionado hasta el punto de que se han creado los "**Event Driven Behavior Trees**", los cuales resuelven problemas de escalabilidad de los Árboles de Decisión originales.

Esto es debido a que se ha cambiado cómo el Árbol de Decisión controla la ejecución de manera interna e introduciendo un nuevo tipo de nodo que permite reaccionar a eventos y abortar los nodos de ejecución que se encuentran en el estado de "En proceso", que es uno de los estados en los que se puede encontrar el nodo. Aunque el nombre sea distinto se decidió dejar el nombre de Árboles de Decisión para su mayor simplicidad.

Dentro de estos eventos podemos encontrar los"Comportamientos estimulados". Dichos comportamientos sirven para aumentar la eficiencia de los Árboles de Decisión. Este tipo de estímulos se le dio uso en juegos famosos como es el **"Halo 2"** [Damian Isla 2005].

Para poner en contexto, en este juego cuando el líder del escuadrón era eliminado los soldados de rangos más bajos entraban en pánico y comenzaban a esconderse. Intentar realizar este tipo de comportamiento sin el uso de los estímulos tiene un problema y es la necesidad de que hay que estar comprobando, en todo momento, si dentro del Árbol se cumplen las condiciones específicas para que los soldados comiencen a esconderse aun sabiendo que el líder no ha sido eliminado.

Por ello, mediante el uso de los estímulos, este comportamiento o impulso se puede añadir dinámicamente a un punto específico del Árbol. Es mediante el uso de los eventos dentro del código que podemos saber que un personaje líder ha sido eliminado y, con ello, conseguimos que durante un corto periodo de tiempo este nuevo nodo sea añadido al Árbol de manera que se tendrá en cuenta, junto con el resto de nodos, para ejecutar la mejor acción posible. Por ello, al añadirlo dinámicamente no es necesario que tengamos que estar comprobando ese nodo del Árbol ya que no existe y se añade cuando se cumplan ciertas condiciones.

5.0.2. Utility AI y la decisión de su uso

Pero al final hemos tomado la decisión de realizar una implementación de la Inteligencia Artificial basada en la ya conocida Utility AI. Dicha decisión fue determinada porque creemos que para los juegos de estrategia por turnos se asemeja, de mejor manera, una Inteligencia Artificial la cual necesita tener en cuenta, en mayor medida, la cantidad de enemigos que hay en el entorno y diferentes variables adicionales como son la vida, la cantidad de puntos de acción u otras. Además, como hemos comentado con los Árboles de Decisión, los caminos han de ser pensados por el desarrollador. Y es que dentro de un juego de estrategia por turnos puede haber decenas de posibles casos dentro de un escenario específico. Por ello, tener todos los casos en cuenta es una tarea demasiado compleja y no merece la pena teniendo un comportamiento basado en puntuaciones como es la "IA de Utilidad".

El concepto principal detrás de la teoría de Utilidad es que toda acción puede estar descrita mediante un solo valor. Dicho valor, a menudo, se refiere a la Utilidad de la acción dentro de un contexto.

El cálculo de dichas puntuaciones debe de ser consistente, ya que al final del cálculo de todas las posibles acciones se debe tomar una decisión por la cual se elegirá, en nuestro caso, la mejor de todas ellas. Por ello, se deben de encontrar en la misma escala. Uno de los puntos de partida iniciales que se pueden usar son los "Valores normalizados".

Estos valores se encuentran entre el 0 y el 1 (ambos incluidos). Aunque todos los rangos que podamos pensar son válidos mientras que se haga el cálculo con la misma escala en el resto de acciones. En nuestro caso hemos elegido un valor que se pueda encontrar entre 0 y 150.

Podemos ver un ejemplo de estas puntuaciones con un caso donde la IA debe determinar si debe aumentar de tamaño la colonia o debe cuidar a los aldeanos. Hay tres factores diferentes, a tener en cuenta, para calcular la puntuación de ambas acciones. El primero de ellos es la cantidad de gente que se encuentra en la colonia en ese momento. Este cálculo se hace dividiendo la población actual entre el máximo de población posible. El segundo factor es la cantidad de comida que se encuentra en la colonia, dividiendo este número por 100. Y, por último, el factor en relación al espacio de la enfermería. Con éstos se hacen los cálculos de cada una de las acciones y se elige la mejor opción posible.

Podremos verlo, de forma más clara, en el esquema siguiente:

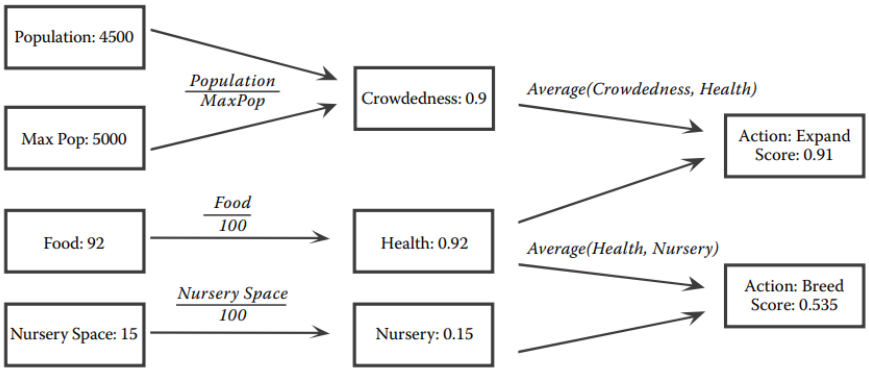

Figura 29: Ejemplo de toma de decisión con la IA de Utilidad

Podemos ver que los cálculos siguen la misma escala encontrando así valores que están en el intervalo [0,1] y obteniendo unas puntuaciones que mantienen la relación.

La clave de este tipo de Inteligencias Artificiales se basa en la comprensión de la entrada que introducimos en la función y la salida que obtenemos. Es, por ello, que lo más importante de la IA de Utilidad es la de descubrir cuál es la mejor función que se adecua a las acciones que van a realizar nuestros personajes. En el ejemplo que hemos visto sobre la colonia hemos aplicado una función lineal donde obteníamos el valor normalizado dividiendo, en todo momento, por el máximo de lo que estábamos calculando. Por ejemplo, para el caso de la población hemos dividido la población actual entre el total.

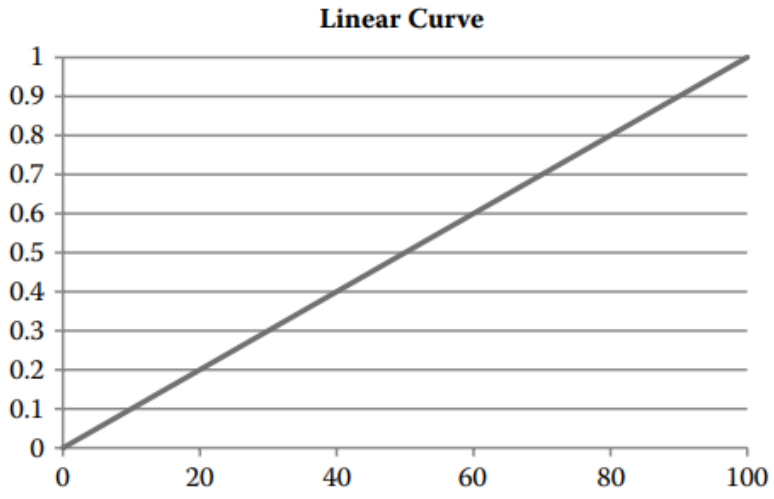

Figura 30: Curva lineal para el cálculo de puntuaciones

En otros casos, también querremos aplicar curvas cuadráticas que nos proporcionan una curva donde se comienza de manera más lenta, pero avanza más rápidamente. Esto lo conseguimos aplicando un exponente a la función lineal.

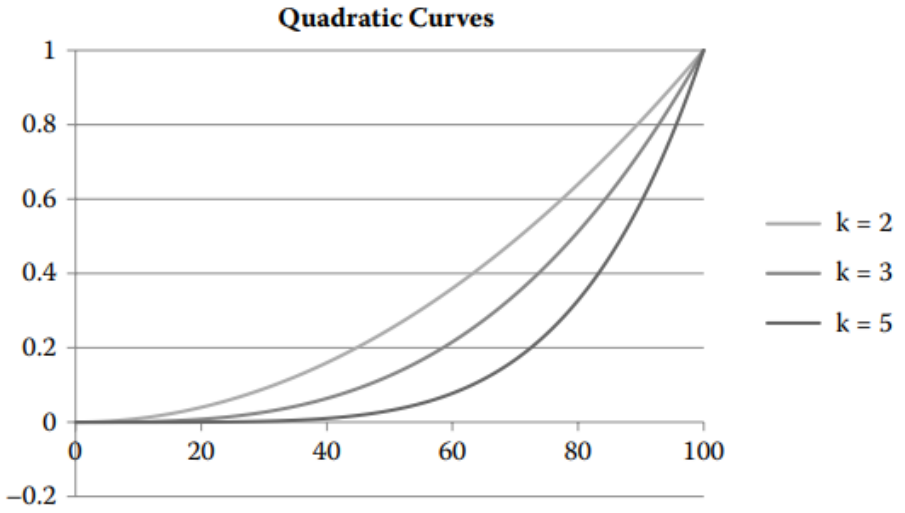

Figura 31: Curva cuadrática para el cálculo de puntuaciones

Podemos obtener una curva cuadrática si aplicamos en el exponente un número que se encuentre dentro del intervalo [0,1].

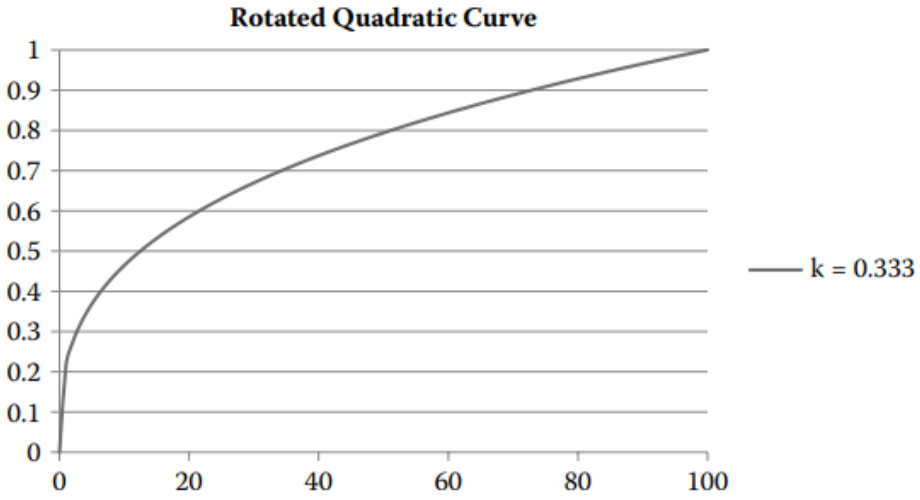

Figura 32: Curva cuadrática para el cálculo de puntuaciones

Un aspecto interesante que ocurre en todos los comportamientos de las Inteligencias Artificiales en los videojuegos es el concepto de la **Inercia**.

Este concepto es referido a que cuando el personaje, controlado por la Inteligencia Artificial, elije una acción deberá mantener ésta durante un tiempo para que el comportamiento no sea errático. Es decir, no podemos tener un NPC que esté continuamente cambiando de decisión a cada momento. Por ello, cuando el personaje tome una decisión deberá continuar con ésta hasta que termine. Una solución, a esto, es la de añadir un bonus a la acción que se esté realizando en el momento, haciendo que el personaje la realice hasta que verdaderamente aparezca una opción mucho mejor, que merezca la pena, para cambiar el comportamiento. Esto es un problema que no hemos tenido en nuestro proyecto ya que, en nuestro caso, no se puede cambiar continuamente de decisión puesto que cuando el personaje elije una de las acciones a realizar no tiene oportunidad de realizar otra, pues quedaría bloqueado y no podrá gastar más puntos hasta que la acción quede completada en su totalidad.

5.0.3. Funcionamiento dentro del proyecto

La implementación de la IA de Utilidad tendrá diferentes comportamientos dependiendo del nivel de dificultad en el que el jugador decida jugar. Tenemos una clase general "EnemyAI" desde la cual controlamos cuándo los personajes, controlados por Inteligencia Artificial, deben realizar las acciones que éstos determinen, siempre y cuando se encuentren dentro de su turno. Es por ello, que disponemos de una clase la cual controla cuándo es el turno del jugador y cuándo es el turno de la máquina.

Método Update de la clase EnemyAI

```
1  void Update()
2  {
3      if (TurnSystem.Instance.IsPlayerTurn())
4      {
5          return;
6      }
7      switch (state)
8      {
9          case State.WaitingForEnemyTurn:
10             break;
11         case State.TakingTurn:
12             timer -= Time.deltaTime;
13             if (timer <= 0f)
```

```
14            {
15                if (TryTakeEnemyAIAction(SetStateTakingTurn))
16                {
17                    state = State.Busy;
18                }
19                else
20                {
21                    TurnSystem.Instance.NextTurn();
22                    onFinishedTurn?.Invoke(this, EventArgs.Empty);
23                }
24            }
25            break;
26        case State.Busy:
27            break;
28    }
29 }
```

Una vez que el jugador ha decidido pasar turno, esta clase procederá a realizar la mejor acción posible para todas las unidades controladas por dicha clase, es decir, las unidades pertenecientes al equipo contrario. Cuando se termine de realizar el método de TryTakeEnemyAIAction significará que se han realizado todas las posibles acciones que los personajes se podrían permitir o, en el caso de que no hubiese unidades a las que realizar alguna acción, terminaría el método al instante y, por ello, se pasaría turno, dándole, de nuevo, el control al jugador para que pueda realizar acciones con sus propios personajes.

Este método de "TryTakeEnemyAIAction" tiene diferentes versiones, dependiendo de los argumentos que le pasemos al método. Este primer método que estamos usando es el encargado de realizar una iteración por todas las posibles unidades y comprobar si dentro de la unidad es posible realizar alguna acción, en el caso de que los puntos se lo permitan.

Método Update de la clase EnemyAI

```
1 private bool TryTakeEnemyAIAction(Action onEnemyAIActionComplete)
2 {
3
4     foreach(Unit enemyUnit in UnitManager.Instance.GetEnemyUnitList())
5     {
```

```
            onAnyActionOfEnemy?.Invoke(this, enemyUnit);
            if (TryTakeEnemyAIAction(enemyUnit, onEnemyAIActionComplete))
            {
                return true;
            }
        }
        return false;
    }
```

El argumento que le pasamos a este método es una Action la cual nos permite saber cuándo se va a realizar la acción y, por tanto, podemos pasar al siguiente estado.

Dentro de la iteración de cada unidad llamamos al evento "onAnyActionOfEnemy" por el cual la cámara tiene en cuenta la unidad que está llamando dicho evento y, por tanto, centramos la cámara a dicha unidad. Después de ello lo que hacemos es comprobar, ya dentro de la unidad, qué posible acción puede realizar. Devolvemos true o false porque, de esta manera, sabremos si hemos podido realizar la acción o no. En el caso de que pueda realizará la acción y devolverá un true de que ha podido realizarla. En caso contrario, solo devolverá false.

Búsqueda de la mejor acción posible

```
private bool TryTakeEnemyAIAction(Unit enemyUnit, Action onEnemyAIActionComplete)
{
    BestAction bestAction = new BestAction();
    foreach(BaseAction baseAction in enemyUnit.GetBaseActionArray())
    {
        if (enemyUnit.CanSpendActionPointsToTakeAction(baseAction))
        {
            switch (GetDifficult())
            {
                case Difficult.Easy:
                    bestAction = GetBestAction(bestAction, action, Difficult.Easy);
                    break;
                case Difficult.Medium:
                    bestAction = GetBestAction(bestAction, action, Difficult.Medium);
                    break;
                case Difficult.Hard:
                    bestAction = GetBestAction(bestAction, action, Difficult.Hard);
```

```
                    break;
                }
            }
        }

        if(bestAction.bestEnemyAIAction != null && enemyUnit.TrySpendActionPointsToTakeAction(
          bestAction.bestBaseAction))
        {
            bestAction.bestBaseAction.TakeAction(bestAction.bestEnemyAIAction.gridPosition,
          onEnemyAIActionComplete);
            return true;
        }
        else
        {
            return false;
        }
    }
```

En este caso, ya dentro de la unidad, iteramos por todas las posibles acciones que tiene el personaje. En el caso de que los puntos de acción, que tenga la unidad, no sean suficientes no se tendrá en cuenta dicha acción para que pueda ser escogida por el personaje. Por tanto, ahora con cada acción y dependiendo de la dificultad, procedemos a ver la puntuación que tiene dicha acción, pero no es solo recoger la puntuación en general de la acción, debemos realizar una comprobación por todas las posibles posiciones que estén al alcance, ya que, en el caso, por ejemplo, de querer disparar deberá comprobar entre todos los enemigos al alcance para poder decidir a cuál de éstos es mejor disparar. Por ello, llamamos al método "GetIfBest" donde recogemos, para dicha acción, la mejor posición que se le aplica.

Búsqueda de la mejor posición para realizar X acción

```
private BestAction GetBestAction(BestAction bestAction, BaseAction action, Difficult diff)
{
    if (bestAction.bestEnemyAIAction == null)
    {

        bestAction.bestEnemyAIAction = action.GetBestEnemyAIAction(diff);
        bestAction.bestBaseAction = action;
```

```
8       }
9       else
10      {
11          EnemyAIAction testEnemyAIAction = action.GetBestEnemyAIAction(diff);
12
13          if (testEnemyAIAction != null && testEnemyAIAction.actionValue >= bestAction.
        bestEnemyAIAction.actionValue)
14          {
15              bestAction.bestEnemyAIAction = testEnemyAIAction;
16              bestAction.bestBaseAction = action;
17          }
18      }
19      return bestAction;
20  }
```

En este método lo primero que hacemos es tener en cuenta si es la primera acción que estamos comprobando y la ponemos como la más óptima, por si se diese el caso de que no hubiese ninguna acción posible adicional. En otro caso, lo que hacemos es que obtenemos la mejor posición para dicha acción dependiendo de la dificultad en la que se encuentre el nivel. En el caso de que dicha posición, para la acción, sea mejor que la posición o acción anterior la sustituimos como mejor posibilidad y realizamos esta iteración hasta que obtengamos un resultado, siendo éste la mejor acción posible para dicho personaje.

Para elegir la mejor posición de dicha acción lo que hacemos es hacer uso del método que hemos creado en la interfaz de las acciones "GetBestEnemyAIAction", el cual nos devuelve un objeto de una clase de tipo "EnemyAIAction", y que lo que contiene es la puntuación y la posición de dicha acción en una posición específica. Por ello, ésta es la que comprobamos para ver la puntuación que tiene y saber cuál es mejor para el personaje.

Contenido de la clase EnemyAIAction

```
1  public class EnemyAIAction
2  {
3      public GridPosition gridPosition;
4      public int actionValue;
5  }
```

Para comprobar todas las posiciones, como hemos comentado, hacemos uso de "GetBestEnemyAIAction" donde insertamos, dentro de una lista, todas las posibles puntua-

ciones de dicha acción y, una vez, hemos comprobado todas las posiciones reordenamos la lista para obtener la que tenga una mayor puntuación.

Funcionamiento del método GetBestEnemyAIAction

```
public EnemyAIAction GetBestEnemyAIAction(Difficult diff)
{
    List<EnemyAIAction> enemyAIActionList = new List<EnemyAIAction>();
    List<GridPosition> validActionGridPositionList = GetValidActionGridPositionList();
    foreach(GridPosition gridPosition in validActionGridPositionList)
    {
        EnemyAIAction enemyAIAction = GetEnemyAIAction(gridPosition, diff);
        enemyAIActionList.Add(enemyAIAction);
    }
    if(enemyAIActionList.Count > 0)
    {
        enemyAIActionList.Sort((EnemyAIAction a, EnemyAIAction b) => b.actionValue - a.
      actionValue);
        return enemyAIActionList[0];
    }
    else
    {
        return null;
    }
}
```

Dentro del método recorremos todas las posiciones que se encuentren dentro de la lista de "validActionGridPosition". Dentro de dicha lista podemos encontrar todas las posibles posiciones a las que el personaje puede optar para poder realizar la acción que se esté comprobando. Dentro de cada posición lo que hacemos es calcular la puntuación que tendría el realizar dicha acción en ese momento y, dado que es un juego de estrategia por turnos, la cuestión del instante de la partida no es tan importante ya que el tiempo no avanza a no ser que pase turno un jugador. Por ello, calcula la puntuación teniendo en cuenta diversas propiedades dependiendo de la acción que esté analizando, no tendrá las mismas variables el disparar a otro personaje que el querer mover el personaje a una posición o recargar el arma.

Para calcular dicha puntuación utilizamos un método que hemos incluido en la interfaz. Este método es el de "GetEnemyAIAction", y que, por tanto, todas las acciones deben

de implementar, así todas podrán ser usadas por los enemigos y calculará, de forma independiente, los costes de las acciones.

Funcionamiento del método GetBestEnemyAIAction

```
1 public abstract EnemyAIAction GetEnemyAIAction(GridPosition gridPosition, Difficult diff);
```

Dentro de este método es donde podremos diferenciar entre el nivel de dificultad y, por tanto, habrá ciertas variables las cuales tendrán más o menos peso dependiendo de dicha dificultad. Por ejemplo, para el lanzamiento de granada la cantidad de unidades tendrá un mayor peso en dificultades más altas para que la Inteligencia Artificial tome la decisión de querer lanzar la granada en ese caso, pudiéndose dar el caso de que en dificultades más ligeras a lo mejor no quiera lanzar la granada porque no tiene tanto peso.

En primer lugar, vamos a ver un ejemplo de cómo se han implementado las acciones viendo el caso de la acción de disparo y realizaremos una explicación del resto de acciones y cómo hemos determinado su comportamiento.

Puntuaciones de la accion de disparo

```
1  public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition, Difficult diff)
2  {
3      Unit targetUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPosition);
4      EnemyAIAction enemyAIAction = new EnemyAIAction();
5      enemyAIAction.gridPosition = gridPosition;
6
7      switch (diff)
8      {
9          case Difficult.Easy:
10             if (calculateChanceNoHelp(50f))
11                 enemyAIAction.actionValue = (100 / Mathf.RoundToInt(GetHitChance(targetUnit) *
       100)) * 50;
12             else
13                 enemyAIAction.actionValue = Mathf.RoundToInt(GetHitChance(targetUnit) * 100);
14             break;
15         case Difficult.Medium:
16                 enemyAIAction.actionValue = (100 / Mathf.RoundToInt(GetHitChance(targetUnit) *
       100)) * 50;
17             else
18                 enemyAIAction.actionValue = Mathf.RoundToInt(GetHitChance(targetUnit) * 100);
```

```
19              break;
20          case Difficult.Hard:
21              if (unit.GetCoverType() == CoverType.None)
22              {
23                  enemyAIAction.actionValue -= 20;
24              }
25
26              int targetHealth = targetUnit.GetHealthSystem().GetUnitHealth();
27              if ((targetHealth < minDamage) || (targetHealth > minDamage && targetHealth <
        maxDamage))
28              {
29                  enemyAIAction.actionValue = (Mathf.RoundToInt(GetHitChance(targetUnit) * 100))
        + (100 - targetHealth);
30              }
31              else
32              {
33                  enemyAIAction.actionValue = Mathf.RoundToInt(GetHitChance(targetUnit) * 100);
34              }
35
36              break;
37      }
38      return enemyAIAction;
39  }
```

Realizamos la separación de las dificultades mediante un "Switch", ya que en el caso de que quisiéramos aumentar la cantidad de niveles de dificultad simplemente tendríamos que añadir un caso más y esto nos permite tener una mayor escalabilidad en el código.

En este caso, tanto para la dificultad fácil como para la normal, la Inteligencia Artificial no tendrá en cuenta la propia posición para saber si está expuesto o está a salvo.

En la dificultad fácil, para que haya una mayor sensación de menor dificultad, habrá un 50 % de probabilidad de que el personaje ataque a otro que no tenga el mayor porcentaje de acierto, permitiendo que el jugador reciba menos cantidad de disparos. En el caso de que no se cumpla el 50 % intentará disparar al personaje con mayor probabilidad, teniendo en cuenta todos los escudos y ayudas que ya hemos comentado anteriormente.

En la dificultad normal el comportamiento es similar al de la dificultad fácil, pero en lugar de ser un 50 %, la probabilidad de que dispare a otro enemigo es del 20 %.

La mayor complejidad, en cuanto al comportamiento en todas las acciones, está centrada en la dificultad difícil. En este caso, para disparar tendrá en cuenta si el propio soldado se encuentra en una cobertura y en el caso de que no lo esté la puntuación bajará. Sin embargo, no podremos indicarle que abandone la acción ya que se puede dar el caso en el que, a lo mejor, disparar es lo importante, aunque el personaje no esté en cobertura.

Después de comprobar si está en cobertura o no se realizará la comprobación de la unidad enemiga. La Inteligencia Artificial comprobará si es posible eliminar al personaje enemigo y, en el caso de que pudiese hacerlo, la puntuación recibirá un bonus dependiendo de la vida que le falte. En otro caso, solo se tendrá en cuenta la probabilidad que hay de acertar el disparo, podemos ver un ejemplo de un caso de esta decisión en la siguiente figura.

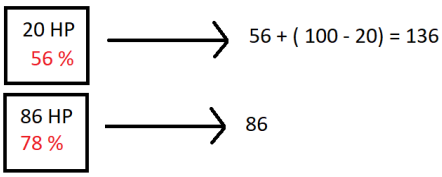

Figura 33: Shoot Action con Inteligencia Artificial

En este caso, dado de que se puede eliminar al personaje, tendría mayor prioridad aun significando esto que el porcentaje de acierto sea menor. En difícil le damos prioridad a que el enemigo pueda hacer que pierdas algunas de tus unidades.

Tanto para el lanzamiento de granadas como para el lanzamiento de misiles el comportamiento será el mismo. Para ello, primero tendrá que comprobar que el personaje tenga algún cohete cargado dentro del arma o que tenga alguna de sus granadas disponibles. Esto se debe a que los personajes que tienen dicha acción de lanzar una granada disponen de dos de ellas que no son regenerables, para que no haya un uso continuado de éstas.

Puntuaciones de la acción de lanzamiento de granada

```
1 public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition, Difficult diff)
2 {
3     EnemyAIAction enemyAIAction = new EnemyAIAction();
4     enemyAIAction.gridPosition = gridPosition;
5
```

```
6       if (grenadesLeft > 0)
7       {
8           int enemyCount = GetEnemyCountHittedByGrenades(gridPosition);
9           int v = 0;
10          switch (diff)
11          {
12              case Difficult.Easy:
13                  if (calculateChanceNoHelp(15f))
14                  {
15                      v = (enemyCount * 25);
16                      enemyAIAction.actionValue = v;
17                  }
18                  break;
19              case Difficult.Medium:
20                  if (calculateChanceNoHelp(30f))
21                  {
22                      v = (enemyCount * 30);
23                      enemyAIAction.actionValue = v;
24                  }
25                  break;
26              case Difficult.Hard:
27                  v = (enemyCount * 30);
28                  enemyAIAction.actionValue = v;
29                  break;
30          }
31      }
32      else
33      {
34          enemyAIAction.actionValue = -100;
35      }
36      return enemyAIAction;
37  }
```

En el caso de que tengan granadas disponibles se pasa a la siguiente comprobación dado que es una parte importante de cómo puede avanzar la misión. Para ello, en dificultades más ligeras que la de difícil se calcula una probabilidad de que tan siquiera se plantee lanzar la granada. Siendo la probabilidad de 15 % y 30 % de pasar esta barrera, respectivamente, para las dificultades fácil y normal. Si pasan dicha probabilidad, el comportamiento con la

dificultad difícil no tiene mucha diferencia. En todos los casos se tienen en cuenta cuántas unidades serían alcanzadas por la granada y lo multiplicamos por un factor para darle más importancia en el caso de que varios personajes sean alcanzados por la misma. Dicho factor será 25, 30 y 30 para fácil, normal y difícil.

El comportamiento en relación al ataque cuerpo a cuerpo no tiene mucha complejidad siempre que el jugador pueda realizar dicha acción. Es decir, que si tiene a un personaje enemigo en una posición contigua a la suya realizará esta acción, ya que elimina al personaje con un 100 % de probabilidad. Por ello, es una acción que si de da el caso de que pueda realizarla, la realizará en todos los casos. **Puntuaciones de la acción de lanzamiento de granada**

```
1  public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition, Difficult diff)
2  {
3      return new EnemyAIAction{
4          gridPosition = gridPosition,
5          actionValue = 200,
6      };
7  }
```

Las unidades enemigas también podrán curarse entre ellas. Por ello, también tendrán una puntuación que les permite decidir si es buen momento para curar o no. En este caso, al igual que con el ataque cuerpo a cuerpo, no diferenciamos entre niveles de dificultad, el comportamiento será siempre el mismo. Y la decisión se adecuará en relación a si tiene alguna unidad herida cerca suya y su puntuación se basará en la cantidad de puntos de vida que le falten a dicha unidad.

Puntuaciones de la acción de curación

```
1  public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition, Difficult diff)
2  {
3      EnemyAIAction enemyAIAction = new EnemyAIAction();
4      enemyAIAction.gridPosition = gridPosition;
5
6      int healTargetCountAtGridPosition = unit.GetAction<HealAction>().
         GetTargetCountAtPosition(gridPosition);
7      List<GridPosition> gridPositionList = unit.GetAction<HealAction>().
         GetWoundedGridPositionList(gridPosition);
```

```
8       Unit moreWoundedUnit = null;
9       if (gridPositionList.Count >= 1)
10      {
11          moreWoundedUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPositionList[0]);
12          for (int i = 0; i < gridPositionList.Count; i++)
13          {
14              Unit aux = LevelGrid.Instance.GetUnitAtGridPosition(gridPositionList[i]);
15              if (aux.GetHealthSystem().GetUnitHealth() < moreWoundedUnit.GetHealthSystem().
        GetUnitHealth())
16              {
17                  moreWoundedUnit = aux;
18              }
19          }
20      }
21      if (healTargetCountAtGridPosition > 0)
22      {
23          enemyAIAction.actionValue = 100 - moreWoundedUnit.GetHealthSystem().GetUnitHealth
        () + 5;
24      }
25      return enemyAIAction;
26  }
```

En primer lugar, buscará si hay alguna posición contigua a la unidad, la cual contenga alguna unidad aliada. Después se comprobará, en el caso de que haya una o más, cuál es la que se encontraría más herida y realizaríamos el cálculo teniendo en cuenta los puntos de vida de ésta.

Otras de las acciones más importantes es la de moverse por el mapa. Para ello, no solo tiene en cuenta las posiciones normales, sino que también tendrá en cuenta si la unidad se encuentra en cobertura o está en medio de la nada y, por tanto, se encuentra desprotegida. De ahí, la importancia de ponerse en una cobertura que cambia dependiendo de la dificultad, en dificultades menores será menos importante colocarse en una cobertura.

Puntuaciones de la acción de movimiento

```
1   public override EnemyAIAction GetEnemyAIAction(GridPosition gridPosition, Difficult diff)
2   {
3       EnemyAIAction enemyAIAction = new EnemyAIAction();
```

```
    enemyAIAction.gridPosition = gridPosition;

    switch (diff)
    {
        case Difficult.Easy:
            enemyAIAction.actionValue = CalculateValueEasy(gridPosition);
            break;
        case Difficult.Medium:
            enemyAIAction.actionValue = CalculateValueMediumHard(gridPosition);
            break;
        case Difficult.Hard:
            enemyAIAction.actionValue = CalculateValueMediumHard(gridPosition);
            break;
    }
    return enemyAIAction;
}
```

En esta función principal hemos creado más métodos auxiliares para que tenga una mayor legibilidad de código, ya que si no sería demasiado amplio. El comportamiento que realizamos, dentro de estos métodos auxiliares, es el de moverse en dirección de los enemigos a la vez que intenta colocarse en una posición donde tenga una cobertura.

Hay una diferencia entre estos dos métodos. Uno de los cambios, que no incluye tanta complejidad, es el de los valores que tienen en cuenta las coberturas, siendo 60 en fácil y 70, 75 en normal y difícil. Pero el cambio que tiene un mayor impacto es que en dificultades mayores no solo tendrá en cuenta las coberturas y los enemigos, también en el caso de que sea médico intentará moverse a una posición donde pueda curar a otras unidades. Para ello, tiene en cuenta a todas las unidades que tenga al alcance de su movimiento y comprueba cuál es la que tiene menor vida. En el caso de que el cálculo de su puntuación sea muy elevado el médico tomará la decisión de acercarse a esa unidad en lugar de desplazarse a una cobertura.

Puntuaciones de la acción de movimiento parte 2

```
private int CalculateValueMediumHard(GridPosition gridPosition)
{
    int targetCountAtGridPosition;
    int value = 0;
```

```
5       if (unit.GetAction<ShootAction>() != null)
6           targetCountAtGridPosition = unit.GetAction<ShootAction>().GetTargetCountAtPosition(
        gridPosition);
7       else
8           targetCountAtGridPosition = 0;
9
10      if (targetCountAtGridPosition > 0 && unit.GetCoverType() == CoverType.None)
11      {
12          switch (LevelGrid.Instance.GetUnitCoverType(gridPosition))
13          {
14              case CoverType.None:
15                  value = 20;
16                  break;
17              case CoverType.Half:
18                  value =70;
19                  break;
20              case CoverType.Full:
21                  value = 75;
22                  break;
23          }
24      }
25      else
26      {
27          Unit unidadMasCercana = getUnitWithLessDistance(gridPosition);
28          int distance = GetDistanceWithEnemyUnit(gridPosition, unidadMasCercana.
        GetGridPosition());
29          value = 200 / distance;
30      }
31
32      if (unit.GetAction<HealAction>() != null)
33      {
34          int healTargetCountAtGridPosition = unit.GetAction<HealAction().
        GetTargetCountAtPosition(gridPosition);
35          List<GridPosition> gridPositionList = unit.GetAction<HealAction().
        GetWoundedGridPositionList(gridPosition);
36          Unit moreWoundedUnit = null;
37          if (gridPositionList.Count >= 1)
38          {
39              moreWoundedUnit = LevelGrid.Instance.GetUnitAtGridPosition(gridPositionList[0]);
```

```
40            for (int i = 0; i < gridPositionList.Count; i++)
41            {
42                Unit aux = LevelGrid.Instance.GetUnitAtGridPosition(gridPositionList[i]);
43                if (aux.GetHealthSystem().GetUnitHealth() < moreWoundedUnit.GetHealthSystem().
      GetUnitHealth())
44                {
45                    moreWoundedUnit = aux;
46                }
47            }
48        }
49        if (healTargetCountAtGridPosition > 0)
50        {
51            value = 100 - moreWoundedUnit.GetHealthSystem().GetUnitHealth();
52        }
53    }
54    return value;
55 }
```

Además, hemos implementado una forma para que, en el caso de que el enemigo no esté en una posición cercana a los personajes del equipo contrario, intente realizar los movimientos en esa dirección ya que, en caso contrario, éstos pueden optar por tomar caminos que se alejan de dichos personajes y, por tanto, sería un comportamiento un poco fuera de lo común. Para ello, lo que planteamos es una búsqueda que nos permita encontrar el personaje enemigo más cercano y, por ello, vamos en su dirección mediante una puntuación que se calcula como 200 / distancia. Realizamos esta división ya que, por reglas matemáticas, cuando dividimos un número por otro mientras más alto sea el divisor menor será el resultado. Por ello, cuando calculemos la puntuación con un personaje que está muy lejos no lo tomará en cuenta ya que el valor será más pequeño que con otros personajes que estén más cerca.

6. Diseño e implementación de menús, interfaces y unidades

El juego contará con un menú inicial donde el jugador podrá elegir si empezar el juego, cambiar de dificultad, elegir el nivel al que desee jugar o salir del juego. El cambio de dificultad se tendrá en cuenta para todos los niveles y si se quisiese cambiar ésta, para algún nivel, tendríamos que hacerlo desde el menú.

Figura 34: Menú principal

Figura 35: Selector de dificultad

Dentro del menú podrá elegir entre los tres niveles disponibles, aunque en éste nos

encontraremos con un mayor número de misiones que podrían aparecer en un futuro. Los niveles jugables son:

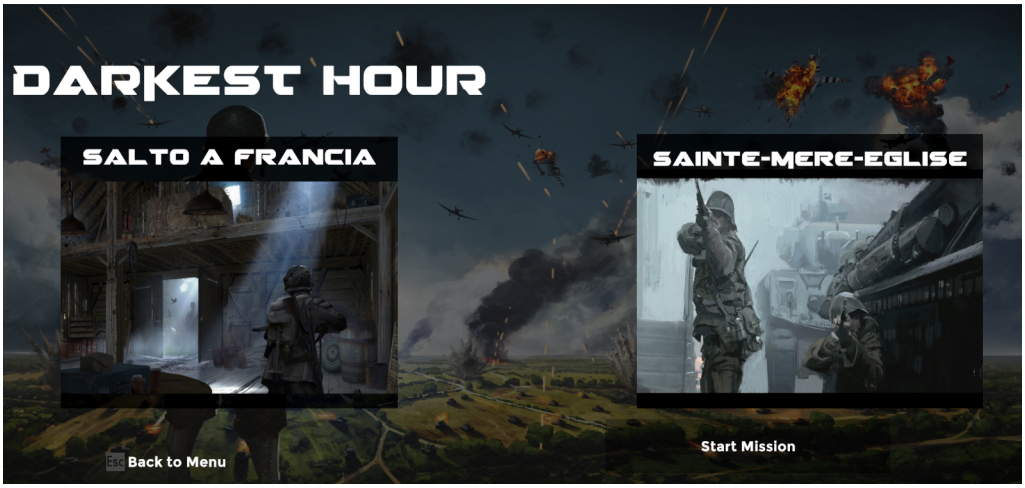

Figura 36: Selector de niveles

Dentro del nivel, el jugador dispondrá de una interfaz donde podrá ver toda la información en relación a su escuadrón y en qué situación se encuentra cada uno de ellos. Se podrá ver información con respecto a la vida que tiene cada soldado, cuántos puntos de acción tienen a su disposición, cuánta munición le queda disponible, el número de granadas restantes y, además, podrá clicar en cualquiera de sus iconos para poder acceder directamente a ellos. Podemos ver esta interfaz en la siguiente imagen indicado con el cuadrado azul.

Figura 37: Interfaz sobre el escuadrón

Además de poder ver, de forma general, a todos los soldados del escuadrón podremos seleccionar, de manera individual, a cada uno de ellos, eligiendo a la unidad y, al hacer esto, podremos realizar las acciones que deseemos, siempre y cuando nos los permita los puntos de acción. Además, en esta sección también podemos ver, de forma más clara, los puntos de acción y las granadas restantes.

Figura 38: Acciones del personaje

7. Creación de niveles jugables

Para la creación de los niveles jugables hemos tenido que realizar diferentes búsquedas de objetos, los cuales nos permitan realizar un nivel que mantenga la misma estética en todos los aspectos y no que hubiese distintos tipos de objetos con diferentes estilos. Por ello, hemos obtenido muchos "Assets" de páginas como SketchFab y, con los éstos descargados, hemos realizado tres niveles que permiten al jugador jugar tres misiones diferentes. La estética será parecida en todos ellos ya que la totalidad de los niveles se encuentran en el territorio de Francia, por ello se ha usado el mismo estilo de casas.

Un ejemplo de las casas que hemos incluido en los niveles es este, éstas formarán parte de las casas que conforman el pueblo.

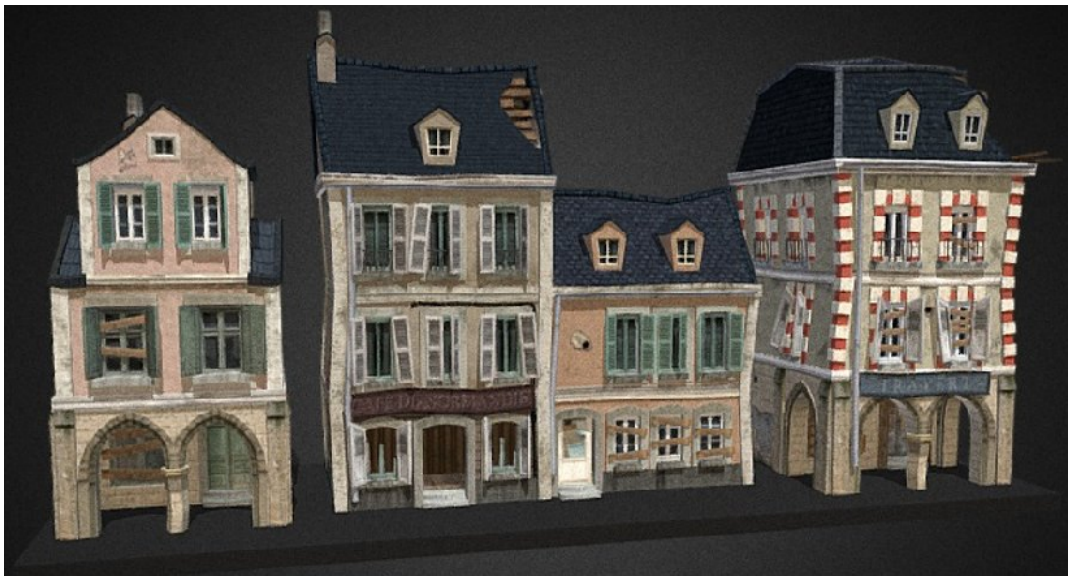

Figura 39: Assets de las casas del pueblo

Nuestro videojuego contendrá dos niveles:

El primero de ellos simulará el salto que se realizó con planeadores de manera previa al Desembarco de Normandía, es un nivel con menores dimensiones y que sirve como pequeño tutorial para que el jugador se acostumbre a las mecánicas.

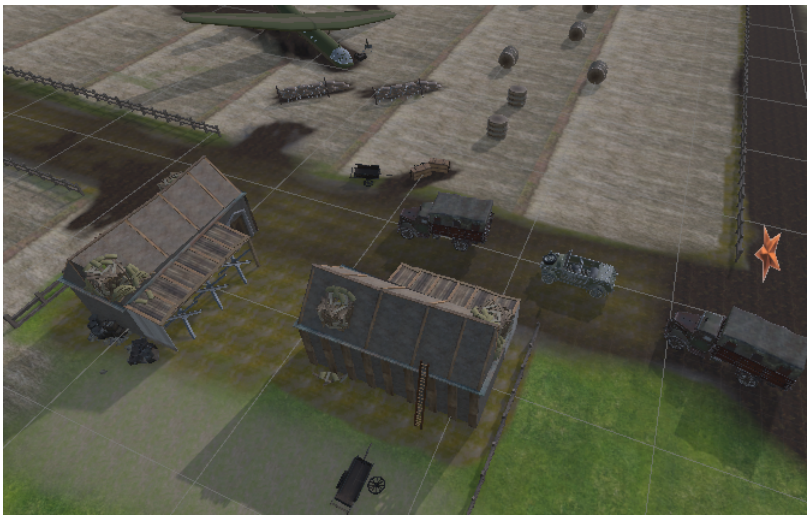

Figura 40: Nivel jugable 1

Dentro de este nivel, el jugador tendrá que eliminar a todos los enemigos que se encuentren dentro de la misión y, además, deberá interactuar con el camión para poder finalizar la misma.

De manera similar, el segundo nivel tendrá lugar en un pueblo de Francia donde el jugador tendrá que hacerse con el control del mismo para poder obtener información del enemigo. Para ello, el jugador tendrá dos objetivos diferentes:

- Destruir la posición de mortero.
- Recoger los planos del enemigo.

Todos los objetivos estarán separados para que el jugador tenga que moverse a lo largo del mapa.

Figura 41: Nivel jugable 2

El segundo nivel tendrá esta estructura y para poder completar dicho nivel, en su totalidad, deberá interactuar con el mortero y recoger los mapas del enemigo. Adicionalmente, deberá eliminar a los enemigos que queden en el pueblo cuando hayan cumplido los demás objetivos.

Aquí hacemos uso de las coberturas como ya hemos explicado y es que, de manera manual, hemos puesto posiciones de cobertura indicando qué tipo de cobertura le convenía a ese tipo de objeto. Podemos ver, por ejemplo, en la imagen diferentes sacos de arena que servirán como cobertura.

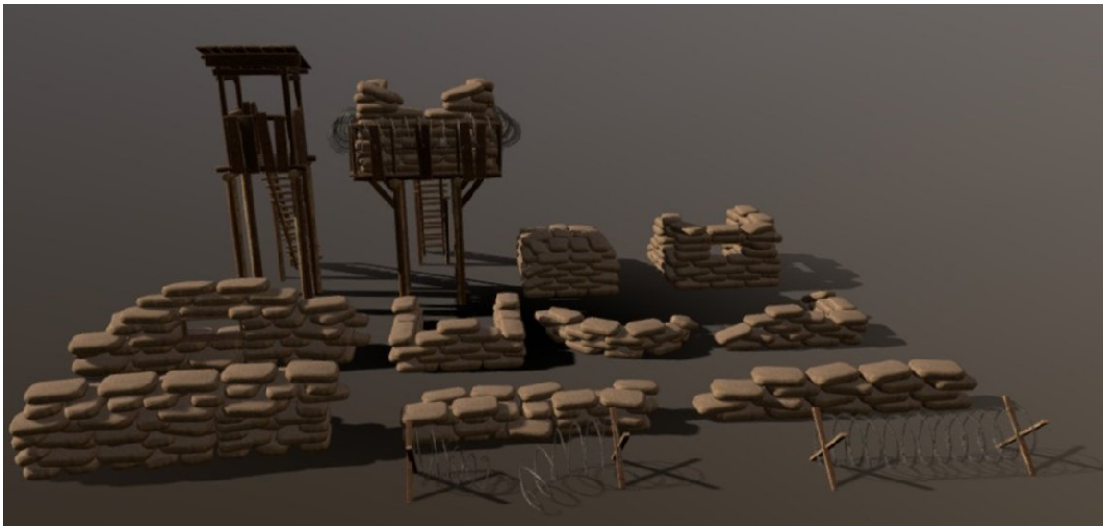

Figura 42: Diferentes coberturas incluidas en los niveles

Todos estos objetos que ponemos dentro de los niveles significarán que no se podrá pasar sobre ellos, es decir, serán obstáculos tal y como serían en la vida real ya que en ésta no podemos atravesar un edificio o una pared. Por ello, para el Algoritmo de Búsqueda A* todos estos objetos los tomará como obstáculos y tomará la mejor ruta para poder sortearlos y, aun así, obtener el mejor camino posible.

Adicionalmente, también haremos uso de algunos artículos de la Asset Store, los cuales nos servirán para dar vida a nuestros personajes y, además, la aparición de vehículos para que el mapa parezca que tiene una mayor diversidad. Para los vehículos hemos utilizado un artículo de pago de la Unity Asset Store. Dichos vehículos también podrán ser usados como cobertura, pero su principal función no será esa. Los vehículos tendrán la siguiente forma.

Figura 43: Diferentes vehículos incluidos en los niveles

Y los soldados que podremos controlar serán de esta forma, también de un Asset que hemos conseguido desde la Unity Asset Store y, por el cual, obtenemos tanto los personajes que podemos controlar como los personajes del enemigo.

Estos sí que los podremos controlar nosotros mismos y hacemos uso de la mayoría de ellos. De igual forma, podemos encontrar la misma versión de soldados en otras facciones como pueden ser los alemanes.

Figura 44: Diferentes tipos de soldados incluidos en los niveles

7.1. Implementación de Objetivos

Para que el jugador pueda completar el nivel al completo deberá realizar los objetivos, que pueden variar dependiendo del nivel en el que se encuentre. Estos objetivos pueden ser de dos tipos, puede constar en eliminar a todos los enemigos de la zona o interactuar con diferentes objetos que se encuentran a lo largo del mapa. Estos objetos podrán ser interactuados con la acción pertinente que ya se ha comentado anteriormente.

El sistema controlará el momento en el que uno de los objetos de misión sea interactuado, avisando de que dicho objeto ha sido modificado y, por tanto, se ha cumplido uno de los objetivos, cambiando el color del texto de la misión e indicando que está completo.

Estos avisos lo hacemos mediante el uso de eventos para poder comunicarnos entre nuestras clases y, por tanto, no tenemos que estar constantemente preguntando al sistema si el objetivo se encuentra completado o no, mediante los eventos podemos avisarle cuando debe cambiarlo.

Actualización de objetivo en la interfaz

```
1
2  private void missionManager_onAnyObjectiveCompleted(object sender, ObjetivoInteractuar e)
3  {
4      UpdateObjective(e);
5  }
6
7  private void UpdateObjective(ObjetivoInteractuar obj)
8  {
9      for (int i = 0; i < objetivos.Count; i++)
10     {
11         if (obj.Equals(objetivos[i]))
12         {
13             listaTexto[i].color = Color.gray;
14         }
15
16     }
17 }
```

7.2. Sistema de Guardado y Selector de dificultad

Nuestro juego contará con un sistema de guardado donde podremos elegir, en cualquier momento de la partida, cuándo queremos realizar una copia de la misma. Esta copia guardada se mantendrá en nuestro ordenador o dispositivo en el que estemos jugando, de manera que si queremos cerrar el juego podremos volver a cargar la partida desde el punto en el que nosotros deseemos.

La opción de guardar y cargar partida se podrá realizar una vez hemos abierto el menú de pausa, donde podremos reanudar la partida por el punto que lo habíamos dejado. Podremos, como hemos comentado, guardar y cargar la partida, aunque ésta solo se podrá guardar una vez. Por tanto, si guardamos otra vez se sobrescribirán los datos en el archivo que se crea al guardar.

Figura 45: Menú de pausa dentro del juego

Para realizar el guardado de los datos creamos un archivo llamado "data.game" y dentro de éste guardaremos la información en relación a las unidades. Toda esta información, de todas las unidades de la misión, la tenemos almacenada dentro de un objeto donde podremos ver diferentes listas y donde tenemos todas las unidades, tanto enemigas como aliadas, e incluso tenemos una lista para saber si tenemos unidades heridas para poder aplicarles las opciones adecuadas una vez se cargue toda la información.

Información de las unidades almacenadas en el archivo data.game

```
[System.Serializable]
public class GameData
{

    public List<string> unitList;
    public List<string> friendlyUnitList;
    public List<string> woundedUnitDic;
    public List<string> enemyUnitsDataList;
    public List<string> enemyUnitsDeadList;

    public List<string> objetivesNotCompleted;
    public List<string> objetivesCompleted;
    public bool needToClearEnemies;

    public int missionNumber;
    public GameData()
    {
        enemyUnitsDeadList = new List<string>();
        woundedUnitDic = new List<string>();
        enemyUnitsDataList = new List<string>();
        unitList = new List<string>();
        friendlyUnitList = new List<string>();
        missionNumber = 0;
        objetivesCompleted = new List<string>();
        objetivesNotCompleted = new List<string>();
        needToClearEnemies = false;
    }
}
```

Para poder almacenar toda esta información, además, hemos creado una interfaz con la que conseguimos salvar y cargar información de aquellas clases de las que queremos conservar la información, esta interfaz es:

Interfaz para guardar y cargar información

```
public interface IDataPersistence
{
    void LoadData(GameData data);
    void SaveData(GameData data);
```

```
}
```

Por tanto, en nuestro Manager, donde controlamos toda esta información, recorremos todos los objetos de los que heredan esta clase "IDataPersistence". En nuestro caso, será la clase "Unit Manager" y, además, también tendremos que guardar la información sobre el estado de la misión en relación a los objetivos que hemos cumplido hasta el momento.

Métodos en DataPersistenceManager

```
public void LoadGame()
{
    this.gameData = dataHandler.Load();

    foreach(IDataPersistence dataPersistenceObj in dataPersistenceObject)
    {
        dataPersistenceObj.LoadData(gameData);
    }
}
public void SaveGame()
{
    if (this.gameData == null)
    {
        Debug.Log("No data was found, Initializing data to defaults");
        NewGame();
    }
    this.gameData = new GameData();
    foreach (IDataPersistence dataPersistenceObj in dataPersistenceObject)
    {
        dataPersistenceObj.SaveData(gameData);
    }
    dataHandler.Save(gameData);
}
```

En este código lo que planteamos es una iteración por todos los objetos en los que tenemos que guardar información, en caso de que tuviésemos que guardar alguna más relevante. De manera más específica vamos a comentar cómo hemos guardado la información sobre las unidades.

En primer lugar, para guardar una partida almacenamos todas las listas de unidades. Para esto, recorremos todas las unidades de cada lista y recogemos la información relevante

de cada unidad. Esta información que recogemos es:

- Vida de la unidad.
- Granadas restantes.
- Munición restante.
- Tipo de soldado.
- Puntos de acción restantes.
- Posición de la unidad.
- Si la unidad está herida.

Toda esta información será almacenada en una clase llamada "UnitData" que es la información que recogeremos dentro de nuestro archivo con el formato de Json. Por ello, con la ayuda de un módulo como es "JsonUtility" guardaremos los atributos como Json directamente y guardaremos esta información dentro del archivo.

Hemos tenido que crear esta clase adicional para la información, ya que si no tuviésemos dicha clase la información que se guardaría sería la instancia de la unidad y, por tanto, luego no podríamos obtener la información de cada unidad. Por ello, guardamos los datos en una clase externa y lo almacenamos en un Json, y, por tanto, esto nos permite recoger los datos de una manera más sencilla en el momento que tengamos que cargar partida en el futuro.

Lo guardamos, de esta manera, dentro del archivo recogiendo toda la información que hemos comentado. Este sería el ejemplo de una sola unidad, por tanto, dentro del archivo tendremos diferentes listas que contendrán toda la información de las unidades como la que mostramos a continuación.

"{\n \"health\": 100,\n \"grenades\": 2,\n \"ammo\": 4,\n \"typeOfSoldier\": 3,\n \"actionPoint\": 2,\n \"position\": {\n \"x\": 52.10900115966797,\n \"y\": 4.76837158203125e-7,\n \"z\": 39.979000091552737\n },\n \"isWounded\": false\n}",

Figura 46: Formato de guardado de una unidad

Los métodos, donde guardamos el resto de listas, son de la misma forma y del mismo comportamiento, pero cambiando para adecuarse a cada uno. En el ámbito de la carga

de información de las unidades, en primer lugar, eliminaremos todas las unidades que se encuentren dentro del campo de batalla. De esta manera, no existirán conflictos sobre diferentes unidades posicionándose en el mismo lugar al instanciar de nuevo las unidades. Además, también destruimos la parte de la interfaz referente a la sección de las unidades y sus atributos para poder instanciarlas, de nuevo, con los nuevos datos. Después de eliminar todo lo referente a las antiguas unidades recogemos los datos de las unidades guardadas dentro del archivo "data.game".

```
1    private void TakeEnemyUnitsBack(List<string> jsonList)
2    {
3        foreach(string json in jsonList)
4        {
5            UnitData unitD = JsonUtility.FromJson<UnitData>(json);
6            Unit unit = SpawnEnemyUnit(unitD);
7        }
8    }
```

Y después de recoger los datos de cada unidad la instanciamos, de nuevo, dentro del mapa con toda la información. En primer lugar, detectamos qué tipo de soldado es el que estamos instanciando y, con esta información, aplicamos el método a un modelo de personaje distinto para mantener la consistencia de las unidades siempre que hayamos cargado la partida.

```
1  private Unit SpawnEnemyUnit(UnitData unitData)
2  {
3      Transform soldierPrefab = null;
4      switch (unitData.GetTypeOfSoldier())
5      {
6          case TypeOfSoldier.INF:
7              soldierPrefab = Instantiate(UnitManager.Instance.Get_GER_INF(), unitData.GetPosition
       (), Quaternion.identity);
8              break;
9          case TypeOfSoldier.MED:
10             soldierPrefab = Instantiate(UnitManager.Instance.Get_GER_MED(), unitData.
       GetPosition(), Quaternion.identity);
11             break;
12         case TypeOfSoldier.SUPP:
13             soldierPrefab = Instantiate(UnitManager.Instance.Get_GER_SUPP(), unitData.
       GetPosition(), Quaternion.identity);
```

```
            break;
        case TypeOfSoldier.AT:
            soldierPrefab = Instantiate(UnitManager.Instance.Get_GER_AT(), unitData.GetPosition()
    , Quaternion.identity);
            break;
    }
    Unit unit = soldierPrefab.GetComponent<Unit>();
    unit.Setup(unitData, true);
    return unit;
}
```

Dentro del método de Setup, lo que aplicamos son el resto de atributos en relación a la vida, las granadas, la munición y el resto de datos que hemos recogido. Además, al instanciar las unidades lo que estamos haciendo es incluirlos dentro de la clase global que tiene constancia de todas las unidades llamada UnitManager, por lo que habremos eliminado las unidades antiguas e incluido las nuevas. De igual manera ocurre con las interfaces, hemos eliminado las antiguas e incluido las nuevas de cada soldado, actualizadas con sus propios valores.

Además de guardar toda la información relevante a las unidades dentro del nivel también tendremos que guardar los datos que van en relación a los objetivos de la misión. Para ello, hacemos que la clase "MissionManager" también herede de la clase IDataPersistence, permitiendo que podamos guardar y cargar los datos de la misión y de los objetivos. Para poder guardar esta información guardamos dos listas, una para los de objetivos que no están completos y otra para los objetivos que si lo están. Cuando se guarde la partida dentro de esta lista, al igual que con las unidades, guardamos una serie de datos. En este caso lo que guardamos es el texto de la misión si está completada y el nombre del objetivo para, más tarde, poder saber qué objetivo tenemos que cargar de nuevo. Además de las dos listas, guardamos un valor booleano, por el cual sabremos si dentro del nivel tendremos que eliminar a todos los enemigos restantes.

Formato de la información respecto a los objetivos

```
"objetivesNotCompleted": ["{\n \"objectiveText\": \"- Roba el camion enemigo\",\n \"
    isCompleted\": false,\n    \"objectiveName\": \"Camion\"\n}"],
"objetivesCompleted": [],
"needToClearEnemies": true,
```

Igual que con las unidades, hemos creado una clase "ObjectiveData" donde decidimos qué información queremos guardar sobre cada objetivo. Para poder cargar los objetivos tenemos que saber qué objetivo estamos modificando, por ello, cuando recogemos los datos almacenados comprobamos el nombre del objetivo y dependiendo de qué tipo de objetivo sea, y si está completado o no, actualizamos los datos correspondientes.

Cargando elementos relacionados con los objetivos

```
1  private void LoadObjectivesNotCompleted(List<string> data)
2  {
3      foreach (string json in data)
4      {
5          ObjetiveData objectivoData = JsonUtility.FromJson<ObjetiveData>(json);
6          ObjetivoInteractuar obj = null;
7          switch (objectivoData.GetObjectiveName())
8          {
9              case "Mortero":
10                 foreach (ObjetivoInteractuar objetivo in listaObjetivosInteractuar)
11                 {
12                     if (objetivo.GetObjectiveName() == "Mortero")
13                     {
14                         objetivo.SetIsCompleted(objectivoData.isCompleted);
15                         SetStar(Mortero, objetivo);
16                         obj = objetivo;
17                     }
18                 }
19                 break;
20             case "Documentos":
21                 foreach (ObjetivoInteractuar objetivo in listaObjetivosInteractuar)
22                 {
23                     if (objetivo.GetObjectiveName() == "Documentos")
24                     {
25                         objetivo.SetIsCompleted(objectivoData.isCompleted);
26                         SetStar(mesaDocumentos, objetivo);
27                         obj = objetivo;
28                     }
29                 }
30                 break;
31             case "Camion":
```

```
32                foreach(ObjetivoInteractuar objetivo in listaObjetivosInteractuar)
33                {
34                    if(objetivo.GetObjectiveName() == "Camion")
35                    {
36                        objetivo.SetIsCompleted(objectivoData.isCompleted);
37                        SetStar(camion, objetivo);
38                        obj = objetivo;
39                    }
40                }
41                break;
42            }
43            if (obj.getIsCompleted())
44            {
45                this.listaObjetivosInteractuarCompletados.Add(obj);
46            }
47        }
48    }
```

Dentro de los menús podremos elegir entre tres niveles de dificultad diferentes, siendo éstos:

- Fácil.
- Normal.
- Díficil.

Estos niveles de dificultad significarán diferentes comportamientos de la Inteligencia Artificial y cómo estos mismos eligen qué acciones se corresponden mejor a cada situación en la que se encuentren. Además de estos comportamientos, también habrá diferentes ayudas dependiendo del tipo de dificultad que hayamos elegido que, como ya hemos comentado, son ayudas que otorgan escudos de porcentajes y ayudas para que el jugador pueda tener una mayor sensación de acierto en los disparos, ayudando cuando su personaje no logre acertar ninguno de los disparos realizados.

Este nivel de dificultad se podrá elegir sólo en el menú principal y también se tendrá que guardar en los datos para el caso de que el jugador guarde una partida y cambie la dificultad del juego. En el caso de que cargue la partida, se cargará con la dificultad que le pertenece.

Tendremos una única clase que será la que controle el nivel de dificultad y éste será uno de los tipos que se encuentran controlados mediante un Enum. De esta manera, tenemos un mayor control en el caso de que queramos añadir más dificultades en un futuro.

```
1 public enum Difficult { Easy, Medium, Hard };
```

Por tanto, cada vez que tenemos que tomar una decisión sobre algún aspecto del juego, lo haremos mediante un "Switch", por el cual tendremos en cuenta todos los posibles casos de cada una de las dificultades y haremos el comportamiento deseado.

7.3. Animaciones y sonidos

Hemos incluido dentro de nuestro videojuego animaciones y sonidos, los cuales nos permiten tener una mayor sensación de inmersión dentro de la historia.

Este conjunto de animaciones lo hemos obtenido a través de la Unity Asset Store y nos permite realizar diferentes animaciones para todas las acciones que hemos incluido, permitiendo dar vida a los personajes y otorgar fluidez al movimiento. Estas animaciones varían entre simples movimientos con las manos, con las cuales simulamos que un personaje está curando a otro, a movimientos más complejos que permiten ver cómo nuestro personaje comienza a correr dentro del nivel.

Para poder aplicar todas estas animaciones hemos creado la clase "UnitAnimator" desde la cual cada personaje controla todas sus posibles animaciones cuando sea necesario y es, dentro de esta clase, donde estamos suscritos a todos los posibles eventos que se lancen y que requieran de una animación. Es decir, en cualquier lugar se realizarán diferentes acciones y cuando se requiera de una animación de uno de los personajes éste recurrirá a la suscripción del evento determinado y realizará la animación correspondiente.

Suscripción a los diferentes eventos

```
1  private void Awake()
2      {
3          unit = GetComponent<Unit>();
4          healthSystem = GetComponent<HealthSystem>();
5          healthSystem.onDead += HealthSystem_OnDead;
6          healthSystem.onWounded += HealthSystem_OnWounded;
7          healthSystem.onDamagedNotDead += HealthSystem_OnDamagedNotDead;
8          healthSystem.onRevived += HealthSystem_OnRevived;
9          unit.onCoverChanged += Unit_OnCoverChanged;
10         Unit.onAnyUnitAlerted += Unit_OnAnyUnitAlerted;
11         unit.onActiveUnit += Unit_OnActiveUnit;
12         if (TryGetComponent<MoveAction>(out MoveAction moveAction))
13         {
14             this.moveAction = moveAction;
15             moveAction.onStartMoving += moveAction_onStartMoving;
16             moveAction.onStopMoving += moveAction_onStopMoving;
17         }
18
```

```
19        if(TryGetComponent<HealAction>(out HealAction healAction))
20        {
21            healAction.onStartHealing += healAction_onStartHealing;
22            healAction.onStopHealing += healAction_onStopHealing;
23        }
24        if(TryGetComponent<GrenadeAction>(out GrenadeAction grenadeAction))
25        {
26            grenadeAction.onGrenadeLaunched += grenadeAction_onGrenadeLaunched;
27        }
28        if(TryGetComponent<ShootAction>(out ShootAction shootAction))
29        {
30            shootAction.onShoot += shootAction_onShoot;
31            shootAction.onAnyMissShoot += ShootAction_OnAnyMissShoot;
32        }
33
34                                ...
```

Además de éstos cada unidad estaría suscrito a más eventos, pero el tamaño de este código sería demasiado alto.

De igual manera ocurre con los sonidos. Dichos sonidos los hemos obtenido de diferentes páginas webs desde las cuales hemos podido descargar los que más se adecuaban a las diferentes acciones, pero el comportamiento sería el mismo que con las animaciones.

Hemos creado una clase llamada "UnitSound" desde la cual hemos incluido todos los sonidos y, de igual manera, nos suscribimos a los eventos y hacemos sonar el sonido que se adecue a cada situación. No hemos incluido el código de las animaciones y los sonidos juntos debido a que es buena práctica separar los comportamientos que no tengan relación. Por ello, separamos las animaciones y los sonidos en diferentes clases ya que tienen diferentes comportamientos y, así, realizamos una buena práctica en el ámbito del desarrollo de videojuegos.

Diferentes audios usados dentro de la clase UnitSound

```
1 [SerializeField] private AudioClip cancion;
2 [Header("Grenade Explosion")]
3 [SerializeField] private List<AudioClip> grenadeSounds;
4 [Header("Rocket Explosion")]
5 [SerializeField] private List<AudioClip> rocketSounds;
```

```
6  [Header("Machine Gun")]
7  [SerializeField] private List<AudioClip> lmgSounds;
8  [Header("Rifle")]
9  [SerializeField] private List<AudioClip> rifleSounds;
10
11 [Header("SMG")]
12 [SerializeField] private List<AudioClip> smgSounds;
13
14 [Header("Reload")]
15 [SerializeField] private AudioClip reloadGun;
16 [Header("Knife")]
17 [SerializeField] private AudioClip knifeAction;
18 [Header("Interact")]
19 [SerializeField] private AudioClip interactAction;
20 [Header("MedKit")]
21 [SerializeField] private AudioClip medkit;
22 [Header("Rocket")]
23 [SerializeField] private AudioClip rocketLaunching;
24 [Header("Move")]
25 [SerializeField] private AudioClip running;
```

Hemos incluido gran variedad de audios aplicando entre ellos, por ejemplo, diferentes tipos de sonidos cuando las armas disparen. Para ello, creamos una lista de clips de audios y cuando sea necesario disparar generaremos un número aleatorio entre 0 y el máximo de dicha lista por el cual se escogerá una pista de audio aleatoria y le damos así una mayor variedad de sonidos y no que se vuelva algo monótono y repetitivo.

Además de los sonidos de cada unidad también hemos incluido diferentes canciones que sonarán de fondo mientras el nivel se encuentre activo. Tanto este sonido de música como el sonido de los efectos ocasionados por las unidades podrán ser modificados, dependiendo del gusto del jugador. Esto será posible activando el menú de pausa donde podremos modificar dos barras, una para cada tipo de sonido y podremos actualizar el volumen de cada uno.

Esto lo haremos dentro de la clase de "PauseMenu" donde tendremos referencias a los "Sliders" y donde, también, podremos modificar los volúmenes de cada unidad.

Figura 47: Cambio de volumen en el menú de pausa

Modificación de los volúmenes

```
1
2  [SerializeField] private Slider musicVolumeSlider;
3  [SerializeField] private Slider effectsVolumeSlider;
4
5
6  public void ApplyVolumeChanges()
7  {
8
9      float musicVolume = musicVolumeSlider.value;
10     float effectVolume = effectsVolumeSlider.value;
11     LoadValues(musicVolume, effectVolume);
12 }
13
14 public void LoadValues(float musicValue, float effectValue)
15 {
16     AudioSource musicSource = GameManager.Instance.GetAudioSource();
17     musicSource.volume = musicValue;
18     musicVolumeSlider.value = musicValue;
19     foreach(Unit unit in UnitManager.Instance.GetUnitList())
```

```
20      {
21          AudioSource unitSource = unit.GetComponent<AudioSource>();
22          unitSource.volume = effectValue;
23      }
24      effectsVolumeSlider.value = effectValue;
25  }
```

Dentro del código lo que hacemos es recoger el valor que se encuentre dentro del elemento que hemos modificado, el cual se encuentra dentro del intervalo [0, 1] y este valor se lo aplicamos al "Audio Source" correspondiente ya que éste es el componente del objeto que se encarga de cargar las pistas de audio y controlar su volumen dentro del juego. En el caso de los efectos tenemos que recorrer todas las unidades que se encuentren dentro del nivel y modificar su "Audio Source" para que se pueda aplicar este cambio de volumen a todas las unidades. En el caso de la música solo será necesario aplicar el cambio de volumen a un solo objeto encargado de la música.

8. Conclusiones y Líneas Futuras

8.1. Conclusiones

En este proyecto se han realizado diversas tareas donde hemos profundizado en el desarrollo de una Inteligencia Artificial capaz de tomar sus propias decisiones sobre qué acción sería más determinante en cada momento.

Se ha podido observar una forma de implementar una IA de Utilidad basada en puntuaciones y también uno de los usos que nos permite un algoritmo como es el Algoritmo A* que nos permite obtener un camino eficiente.

Para poder realizar el proyecto al completo, hemos tenido que abarcar la mayoría de los campos que se requieren a la hora de realizar un trabajo de tal envergadura. No solo hemos implementado los diferentes algoritmos, también hemos realizado una labor de gestión de proyectos donde hemos tenido que tomar diferentes decisiones en los diferentes Sprints que realizamos y que nos permitieron tener un control sobre todo el proyecto.

Esta gestión de proyectos que hemos realizado, junto con los diferentes campos que hemos tratado dentro del motor de videojuegos como es Unity y que hemos usado para completar el proyecto, nos ha otorgado una experiencia no solo académica sino también profesional y personal.

Hemos obtenido experiencia en el desarrollo de técnicas que se usan, en el día a día, en el campo de la Inteligencia Artificial, aprendiendo a investigar diferentes opciones y cuáles tienen las mejores características para diferentes casos. Además de este campo, hemos investigado otros campos del sector de los videojuegos abordando temas como pueden ser el sistema de animaciones y de sonidos e interfaces dentro de un videojuego.

A nivel personal, este TFG ha sido un proyecto que me ha permitido aumentar los conocimientos de un campo tan interesante y que me apasiona, permitiéndome seguir mi carrera profesional y habiendo adquirido conocimientos de todos los campos posibles para un futuro.

8.2. Líneas Futuras

Podemos encontrar diferentes líneas que nos permitirán continuar con el proyecto y que serán, entre otras, las que a continuación se detallan:

- Mejorar el sistema de mallas permitiendo, con ello, tener diferentes alturas dentro de los niveles o, también, incluir el interior de los edificios dentro de dichos niveles.
- Incluir dentro de los niveles el uso de los vehículos y, por tanto, su comportamiento dentro de la Inteligencia Artificial.
- Realizar diferentes mejoras en el comportamiento de la Inteligencia Artificial, permitiendo, con ello, que las puntuaciones tengan en cuenta un mayor número de factores.
- Posibilidad de mapas generados proceduralmente para crear niveles aleatorios teniendo, por ejemplo, unos objetivos ya concretos, y con estos que se genere un nivel aleatorio.
- Incluir un sistema de inventario dentro de cada personaje que nos permita recoger objetos, armas o munición de una manera más interactiva y, todo ello, teniendo en cuenta la munición de manera general.

El objetivo de estas líneas sería aumentar el tamaño del proyecto inicial intentando, así, tener un videojuego completo y que contenga diferentes aspectos, los cuales otorgen al producto final las características necesarias para que obtengamos el videojuego completo antes mencionado.

9. Apéndice A. Manual de Ejecución

Para poder ejecutar el proyecto y jugar al videojuego hay que realizar los siguientes pasos.

1. Descargar el archivo "Build.zip", siendo éste el archivo comprimido donde se encuentra el proyecto.

2. Descomprimir el archivo dentro de una carpeta, siendo el lugar de la descompresión de elección personal (Tamaño de 500 MB necesario).

3. Ejecutar el archivo "Project XW2.exe".

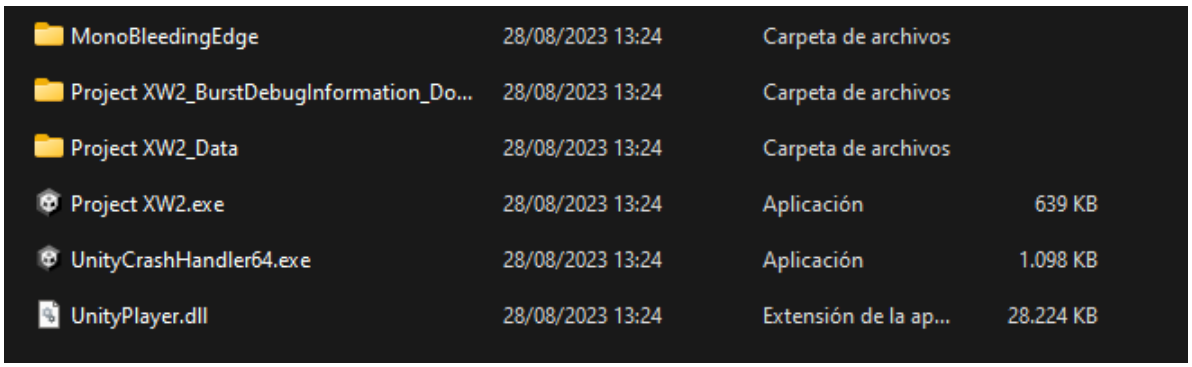

Figura 48: Manual de ejecución

10. Bibliografía

10.1. Artículos

- Damian Isla. (2005). GDC 2005 Proceeding: Handling Complexity in the Halo 2 AI. https : //www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai
- David Huebner. (2017). Indie AI Programming: From behaviour trees to utility AI. https://www.gamedeveloper.com/programming/indie-ai-programming-from-behaviour-trees-to-utility-ai#close-modal
- Alex J. Champandard, Philip Dunstan. GameAIPro_Chapter06_The_Behavior_Tree_Starter_Kit. http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter06_The_Behavior_Tree_Starter_Kit.pdf
- David "Rez" Graham. GameAIPro_Chapter09_An_Introduction_to_Utility_Theory. http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter09_An_Introduction_to_Utility_Theory.pdf
- Morgan Walkup. (2021). AI Made Easy with Utility AI. https://medium.com/@morganwalkupdev/ai-made-easy-with-utility-ai-fef94cd36161
- Jonas Erkert. (2019). Utility AI. https://jonas-erkert.de/utility-ai/
- Kevin Dill, Pat Garrity Gino Fragomeni, Eugene Ray Pursel, (2012). Design Patterns for the Configuration of Utility-Based AI. https://course.ccs.neu.edu/cs5150f13/readings/dill_designpatterns.pdf
- GeekForGeeks, (2023). A* Search Algorithm. https://www.geeksforgeeks.org/a-search-algorithm/

10.2. Texturas

- Rob Tuytel. (2019). Forest Ground 01 https://polyhaven.com/a/forrest_ground_01
- Rob Tuytel. (2021). Aerial Mud 1 https://polyhaven.com/a/aerial_mud_1
- Rob Tuytel. (2019). Reed Roof 04 https://polyhaven.com/a/reed_roof_04
- Rob Tuytel. (2020). Aerial Grass Rock https://polyhaven.com/a/aerial_grass_rock

- Rob Tuytel. (2018). Cobblestone Floor 03. https://polyhaven.com/a/cobblestone_floor_03
- Amal Kumar. (2023). Red Sandstone Pavement https://polyhaven.com/a/red_sandstone_pavement
- Dario Barresi, Dimitrios Savva. (2022). Roots. https://polyhaven.com/a/roots
- eye-cancy.xyz. (2023). Mud Forest https://polyhaven.com/a/mud_forest

10.3. Música

- Andreas Adler. (2017). Battle Call https://www.youtube.com/watch?v=CbgQkDhbjgo
- Jon Adamich. (2017). The Viking Charge https://www.youtube.com/watch?v=V_HgnP_qT00
- Evgeny Emelyanov. (2017). The End of the World https://www.youtube.com/watch?v=xUExa3g1z4g
- Mark Petrie, Andres Prahlow (2017). Reaching Horizon https://www.youtube.com/watch?v=lY1iYx26cEo
- Mark Petrie, Andres Prahlow (2017). Approaching The Summithttps://www.youtube.com/watch?v=j04UwmDzJCM
- Christopher Lennertz (2005). Dogs of warhttps://www.youtube.com/watch?v=USahlXwUjmo

10.4. Efectos de sonido

- The Recordist. (2014). Gun AK47 Machine Gun 1http://creativesounddesign.com/sound/mp3_14/Gun_AK47_Machine_Gun_1.mp3
- The Recordist. (2014). Gun AK47 Machine Gun 2http://creativesounddesign.com/sound/mp3_14/Gun_AK47_Machine_Gun_2.mp3
- The Recordist. (2014). Gun Glock 17 9mm Single Shot 1 1http://creativesounddesign.com/sound/mp3_14/Gun_Glock_17_9mm_Single_Shot_1.mp3

- The Recordist. (2014). Explosion Large Blast 1http://creativesounddesign.com/sound/mp3_14/Explosion_Large_Blast_1.mp3
- The Recordist. (2014). Gun AK47 Machine Gun 2http://creativesounddesign.com/sound/mp3_14/Explosion_Large_Blast_2.mp3
- The Recordist. (2014). Gun AK47 Machine Gun 3http://creativesounddesign.com/sound/mp3_14/Explosion_Large_Blast_3.mp3
- The Recordist. (2014). Gun Machine Gun M60E Burst 1http://creativesounddesign.com/sound/mp3_14/Gun_Machine_Gun_M60E_Burst_1.mp3
- The Recordist. (2014). Gun Machine Gun M60E Burst 2http://creativesounddesign.com/sound/mp3_14/Gun_Machine_Gun_M60E_Burst_2.mp3
- nathanaelsams. (2011). Running on grass with wet feethttps://freesound.org/people/nathanaelsams/sounds/127955/

10.5. Assets

- SilkevdSmissen. (2019). WW2 Cityscene - Carentan inspired. https://skfb.ly/6RoYD
- SilkevdSmissen. (2019). 3 Buildings - WW2 Carentan Inspired. https://skfb.ly/6Tr8F
- SilkevdSmissen. (2019). Low Poly Military Jeep - Willys MB - WW2 Scene. https://skfb.ly/6VRtJ
- Robin Vandenberghe. (2019). Small WW2 scene. https://skfb.ly/6WXQV
- Nicholas-3D. (2023). Plants Scene Free!. https://skfb.ly/oCXru
- Nicholas-3D. (2022). Farm (low poly). https://skfb.ly/otPnE
- BytesCrafter. (2017). Farm Haystack. https://skfb.ly/6pqJN
- Kesenkai. (2022). Volkswagen Typ-82. https://skfb.ly/owDuq
- BiscuitEater. (2022). Stylized Victorian Fence. https://skfb.ly/oyX8n
- danieljorge435. (2019). Single Room Building WWII. https://skfb.ly/6X8YH

- creationwasteland. (2019). Small Barracks. https://skfb.ly/6WRIA
- Sellpet. (2021).Pak 40 & Zis 3 Anti-Tank Gun. https://skfb.ly/orGNY
- mohamedhussien. (2022). Mo 45 Opel Blitz. https://skfb.ly/oyGpB
- Killian_Delias. (2021). Tank Trap. https://skfb.ly/onHTz
- Thunder. (2020). WW2 Mortar. https://skfb.ly/6SX96
- wouterDAE. (2020). Grenade Box - WW2 Carentan Scene. https://skfb.ly/6RtJz
- TraianDumbrava. (2017). Sandbag German WWI Flag. https://skfb.ly/6suEv
- Evan. (2021). Sandbags - Defense line. https://skfb.ly/opKFn
- Polygon Blacksmith. (2019). Toon Soldiers - WW2 edition. https://assetstore.unity.com/packages/3d/characters/toon-soldiers-ww2-edition-56250
- Kado3D. (2017). WWII Low Poly US Tanks Pack 2 https://assetstore.unity.com/packages/3d/vehicles/land/wwii-low-poly-us-tanks-pack-2-95689
- CraftPix. (2020). Free TDS Game UI Pixel Art. https://free-game-assets.itch.io/free-tds-game-ui-pixel-art

UNIVERSIDAD DE MÁLAGA | uma.es

E.T.S de Ingeniería Informática
Bulevar Louis Pasteur, 35
Campus de Teatinos
29071 Málaga

E.T.S. DE INGENIERÍA INFORMÁTICA