

Universidad de Murcia

FACULTAD DE INFORMÁTICA

PROYECTO DE IA PARA EL DESARROLLO VIDEOJUEGOS

*Profesores: Luis Daniel Hernández Molinero
Francisco Javier Marín-Blázquez Gómez*

Vocal: Sergio Marín Sánchez , Grupo: 1.2
Email: sergio.marins@um.es

Jose Miguel Sánchez Fernández, Grupo: 1.2
Email: josemiguel.sanchezf@um.es

Gaspar Muñoz Cava, Grupo: 1.3
Email: gaspar.munozc@um.es

Fecha: 17 de junio de 2022

Curso: 2021/22

Índice de contenidos

I Bloque 2	1
1. Interfaz gráfica y Jugabilidad	2
2. Campo de batalla	4
3. Tipos de Unidades	5
4. Comportamiento táctico	9
4.1. Condiciones	9
4.2. Acciones	10
4.3. Comportamiento de los agentes	11
4.3.1. Infantería	12
4.3.2. Lancero	12
4.3.3. Arquero	13
4.3.4. Caballería	14
5. Comportamiento Estratégico	16
5.1. Modos de comportamiento	16
6. Sistema de combate	18
6.1. Condición de victoria	19
7. Mapa Táctico	20
7.1. Minimapa	20
7.2. Creación mapa táctico	21
7.2.1. Mapa de Influencia	21
7.2.2. Mapas de Tensión y Vulnerabilidad	22
8. Pathfinding táctico individual	24
8.1. Algoritmo A*	24
8.2. Pathfinding basado en A*	25
9. Modo Depuración	27
Referencias	29

Índice de figuras

1.	Interfaz gráfica estrategia	2
2.	Interfaz gráfica cambio de mapa	2
3.	Acción del personaje	2
4.	Terreno de juego	5
5.	Árbol de comportamiento de infantería pesada	12
6.	Árbol de comportamiento de lancero	13
7.	Árbol de comportamiento de arquero	14
8.	Árbol de comportamiento de caballería	15
9.	Tipos de mapas	20
10.	Gizmos disponibles	27

Parte I

Bloque 2

Una vez implementado el comportamiento basado en el movimiento de unidades se pasará a implementar distintos comportamientos propios de una IA táctica dentro de un entorno de juego de guerra en tiempo real. En este bloque se hará un repaso por los distintos pasos y decisiones tomadas para superar los distintos apartados (obligatorios y opcionales) necesarios para implementar estos elementos.

Para esta práctica se utilizará todo lo implementado en la primera parte de la asignatura relacionado con movimiento de personajes, formaciones y colisiones.

El sistema de combate implementado consta de distintos tipos de unidades y terrenos, donde cada unidad tendrá distintos valores tanto de vida, ataque, alcance (cuerpo a cuerpo y a distancia) y velocidad de movimiento que dependerá en muchos casos del terreno por el que se mueve el personaje. El mapa se ha implementado de forma manual como un grid, teniendo dos tipos de mapas. Por un lado el mapa convencional que consta de distintos tipos de terrenos, puntos de interés, bases, personajes, etc.; y por otro lado un mapa que mostrará las distintas influencias en forma de ‘mapa de calor’.

Además existirán distintas zonas de interés táctico en el mapa como pueden ser puntos de curación, zonas de paso entre ‘regiones’, bases o puntos intermedios de cruces de caminos. Cada una de estas zonas de interés tendrá un tipo de comportamiento dependiendo del equipo, por ejemplo, las bases solo podrán ser tomadas por personajes enemigos.

El movimiento de los personajes del juego estará condicionado por distintos factores que se irán comentando en las siguientes secciones.

Tendremos en cuenta el tipo de terreno, el tipo de personaje, la interacción entre los distintos personajes con los tipos de terrenos que marcará un comportamiento distinto con atributos cambiantes como puede ser la velocidad de los personajes y por último la influencia del mapa que se utilizará también para el comportamiento de los personajes.

1. Interfaz gráfica y Jugabilidad

El juego como ya se ha comentado consistirá en ver cómo los NPCs aplican una estrategia concreta, en este caso es el jugador el que a través de botones puede interaccionar para cambiar la estrategia. En la siguiente figura podemos ver cómo queda la interfaz.



Figura 1: Interfaz gráfica estrategia

Los iconos de espadas permitirán poner el modo ofensivo al equipo que se desee mientras que el icono del escudo permite indicar al equipo que se ponga a defender. El botón de *Total War* cambia el comportamiento de todas las unidades del juego para que estas den prioridad al enfrentamiento y toma de la base rival. El juego comenzará con los botones desactivados.

Además tendremos una segunda UI a la que accederemos presionando la tecla I. Esta nos permitirá tener el mapa de influencias en grande y el mapa de juego normal en un mini mapa, teniendo la siguiente interfaz que permite cambiar el mapa que se muestra.

MAPA ACTUAL: Tensión

CAMBIAR MAPA

Figura 2: Interfaz gráfica cambio de mapa

Por otra parte, se puede activar una opción para mostrar la acción que está realizando cada personaje de forma visual en el juego. Esta opción se activa presionando la tecla E.



Figura 3: Acción del personaje

A continuación mostramos una tabla con las distintas imágenes asociadas a cada estado.







Imagen	Acción asociada
	Atacar enemigo
	Capturar la base enemiga
	Defender base propia
	Ir a curarse
	Ir a la base enemiga
	Curarse

Tabla 1: Leyenda de acciones

Esto se ha implementado mediante la clase `FollowStates`. Esta clase tiene referencias tanto del personaje como de una imagen próxima a él. Se ha definido también un enumerado, llamado `ActionState` que recoge cada una de las acciones vistas en la tabla anterior. Cada agente posee una variable de tipo este enumerado y es la que nos servirá para llevar a cabo esta labor.

```
8 public enum ActionState
9 {
10     AttackEnemy = 0,
11     CaptureBase = 1,
12     Defend = 2,
13     GoHealing = 3,
14     GoEnemyBase = 4,
15     Heal = 5
16 }
```

En cada frame se asigna la nueva posición en función del agente y se le asigna la imagen que le corresponda.

```
42 transform.position = cam.WorldToScreenPoint(lookAt.Position + offset);
43 img.texture = myTextures[(int) lookAt.ActionState];
```

2. Campo de batalla

En esta sección veremos los tipos de terrenos que se han utilizado, cómo se han usado y cual será su forma de afectar al movimiento de los personajes. El mapa de juego se ha creado en forma de *grid* donde cada celda del *grid* tendrá una etiqueta de tipo **Floor** y una textura que permita distinguir el tipo de terreno. En total se han utilizado 10 tipos distintos de terrenos:

1. Piedra gris: La piedra gris permite tanto delimitar los bordes de las bases como los pasos entre zonas. Es un material por el que a priori los personajes no tienen mayor dificultad para andar, exceptuando las unidades con montura que pueden resbalar en él por lo que deberían reducir su velocidad.
2. Camino marrón claro y marrón oscuro: Estos dos materiales son usados para crear calzadas por las que las unidades terrestres gustan andar y a priori todas podrán andar por estos caminos teniendo en cuenta que unas tendrán preferencia por el marrón y otras por el marrón claro. Las preferencias de movimiento se verán más adelante.
3. Pradera: Es una zona en donde las unidades pesadas encontrarán dificultad para moverse, pero otro tipo de unidades más ligeras, en especial las unidades a distancia encontrarán facilidad de movimiento y sitio para atacar a unidades que puedan encontrarse en los caminos.
4. Arena clara y oscura: Estas texturas formarán zonas de dunas de arena en donde la mayoría de unidades verán reducido su movimiento pero tendrán que atravesar si quieren conseguir algún sub-objetivo o zona de curación.
5. Agua: Las zonas de agua oscura serán las más profundas y que ninguna unidad será capaz de atravesar.
6. Suelo: Las zonas de suelo ajedrezadas indicarán dónde se sitúan las bases.
7. Lava: Terreno al que las unidades suelen dar muy poca preferencia ya que reduce considerablemente la velocidad de movimiento.

El terreno de juego como podemos ver en la Fig. 4 está formado por un grid de 52×52 casillas, donde se han representado 2 bases, la del equipo A en la zona inferior y el equipo B en la zona superior. Estas bases están separadas por un río que no se puede cruzar y conectadas por 3 puentes. Cada base está conectada por una serie de caminos hacia la base enemiga, encontrándose zonas de interés como zonas de curación.

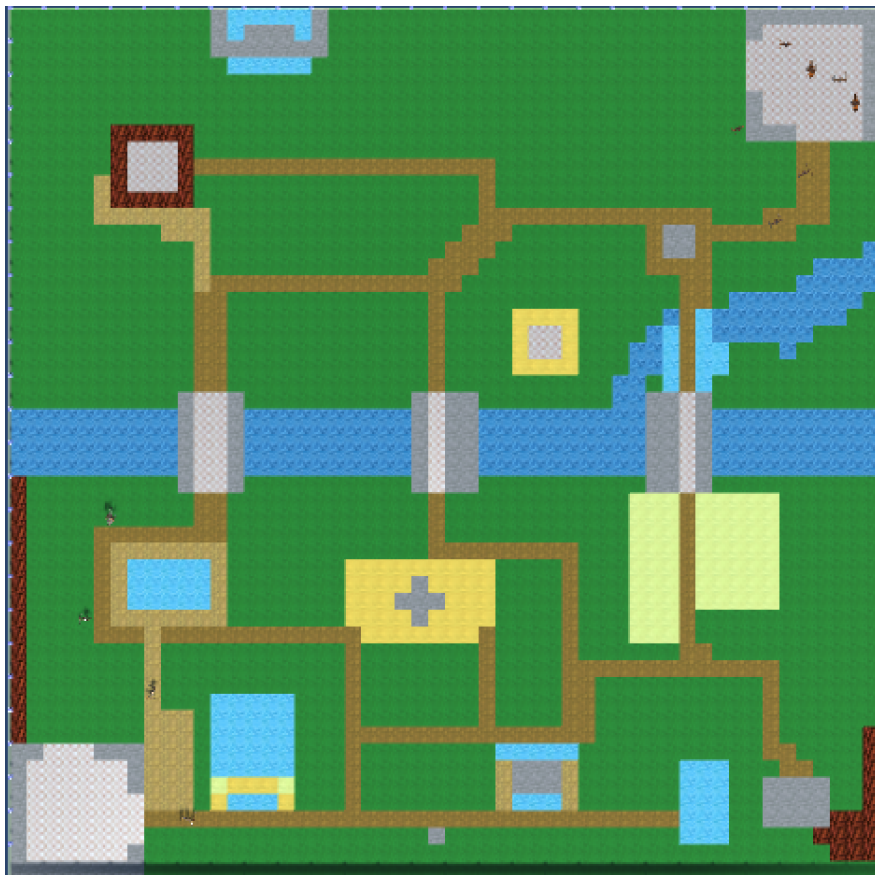


Figura 4: Terreno de juego

3. Tipos de Unidades

Al habernos basado en un juego de captura de bases tipo RTS lo suyo era tener varias unidades que tengan movimientos variados dependiendo del terreno y de los personajes que se encuentren para pelear. Es por esto que se han implementando 4 tipos de unidades:

- Unidades de infantería
- Unidad arquera
- Unidad a caballo
- Unidad lancera

Para distinguir las unidades se han usado unos assets de la store de unity [1].

Por cada unidad tendremos una clase distinta que hereda de la clase **AgentNPC**. En cada una de estas clases hija tendremos una estructura de tipo diccionario $\langle \text{Terreno}, \text{Float} \rangle$ donde a cada tipo de terreno se le asignará un multiplicador de influencia para cada personaje, los cuales se utilizarán en el comportamiento táctico de las unidades que veremos más adelante.

La clase **AgentNPC** definirá todos los atributos de un personaje como pueden ser la vida,

velocidad de movimiento, ataque, etc y serán las clases hijas las encargadas de asignar valores a estos atributos en función de las características que se les quieran asignar. Además tenemos el método `GenerateCostsDict` en cada clase de unidad, que será llamado en el método `Start`, donde se asigna a cada tipo de terreno un multiplicador de coste de movimiento para ese personaje, veamos como quedaría en el caso de la unidad lancero la asignación de influencia e inicialización de atributos:

```
15 private new void Start()
16 {
17     base.Start();
18     _mass = 3f;
19     _maxSpeed = 2f;
20     _maxRotation = 2f;
21     _maxAcceleration = 2f;
22     _maxAngularAcc = 2f;
23     _maxForce = 4f;
24     _baseDamage = 40f;
25     _attackRange = 6f;
26     _attackSpeed = 4f;
27     _hpMax = 250;
28     _hpCurrent = 250;
29
30     GenerateCostsDict();
31 }

39 private void GenerateCostsDict()
40 {
41     terrainCosts.Add(TerrainType.Floor, 1f);
42     terrainCosts.Add(TerrainType.Stones, 1.5f);
43     terrainCosts.Add(TerrainType.Sand, 3f);
44     terrainCosts.Add(TerrainType.Grass, 5f);
45     terrainCosts.Add(TerrainType.Lava, 15f);
46     terrainCosts.Add(TerrainType.Water, Mathf.Infinity);
47 }
```

En el método `ApplySteering` de la clase `AgentNPC` se multiplicará la componente lineal del steering por un factor. Este factor para cada tipo de personaje se puede ver en la Tabla 3. A continuación la función que devuelve este factor para la unidad arquero como ejemplo.

```
46 protected override float GetVelocityFromTerrain()
47 {
48     TerrainType terrain = GetActualTerrain();
49     switch (terrain)
50     {
51         case TerrainType.Stones:
52         {
53             return 0.9f;
54         }
55         case TerrainType.Sand:
56         {
57             return 0.6f;
58         }
59         case TerrainType.Water:
60         {
61             return 0f;
62         }
63         case TerrainType.Lava:
64         {
65             return 0.02f;
66         }
67         default:
68         {
69             return 1f;
```

```
70     }  
71     }  
72 }
```

El método `GetActualTerrain` simplemente devolverá el tipo de terreno en el que se encuentra el personaje.

En la siguiente tabla podemos ver una relación de los tipos de unidades y su multiplicador de velocidad según el terreno. También veremos otra tabla con los distintos valores de influencia dependiendo del terreno y la unidad.

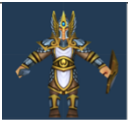
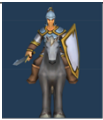












				
  	1.5	1.5	1.5	1.5
 	10	10	3	3
 	∞	∞	∞	∞
	5	1	5	1
	1	5	1	1
	15	15	15	15

Tabla 2: Tabla de Influencias

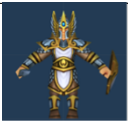
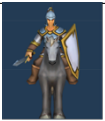












						
			75 %	75 %	90 %	90 %
			25 %	25 %	60 %	60 %
			0 %	0 %	0 %	0 %
			50 %	100 %	45 %	100 %
			100 %	50 %	100 %	100 %
			5 %	5 %	10 %	2 %

Tabla 3: Tabla de velocidades

4. Comportamiento táctico

Para el funcionamiento del juego hemos diseñado un comportamiento y características propias de cada personaje. Para esto hemos usado una herramienta llamada Behavior Trees [2], el cual nos permite crear un árbol de decisiones sobre cada unidad. Dentro de estos árboles podemos encontrar distintos nodos. Para poder entender mejor estos árboles de comportamiento primero veremos los Scripts que encontramos dentro del directorio `Assets/Scripts/Tactica/Behaviour` del proyecto, creados para las acciones y las condiciones que serán la clave del comportamiento ‘inteligente’ de cada personaje.

4.1. Condiciones

En total contamos con 13 scripts de condiciones distintos que heredan de la clase `Conditional` que modelan el comportamiento de nuestros agentes. Estos scripts en el método `IsUpdatable` se encargan de llamar a los métodos del agente definido que comprueban una condición en especial. Los bloques de código serán tanto del método `IsUpdatable` como de la clase `AgentNPC` que es la clase padre de todos nuestros personajes, encargada de implementar todos los métodos de comprobación de condiciones a los que se llaman.

- **Comprobaciones de estado:** para este cometido se tienen 3 nodos distintos. Cada uno comprueba si el estado actual del personaje es ataque (`AttackMode`), defensa (`DefenseMode`), o bien, guerra total (`Total War`).
- **Comprobación de muerte:** esta labor recae sobre un nodo (`IsAlive`) que comprueba si el agente está vivo. Esta comprobación controla la ejecución del resto del árbol.
- **Comprobaciones de salud:** tenemos dos comprobaciones sobre el estado de salud del personaje. Una de ellas (`LowHP`) comprueba si la salud del enemigo ha caído por de la mitad de los puntos totales. La otra (`NotFullHP`) comprueba si un personaje no tiene la totalidad de los puntos de salud.
- **Comprobaciones de localización del personaje:** esta es la categoría donde más nodos distintos hay. Los podemos dividir en 3 grupos, según las distancias que comprueban:
 - **Comprobación de localización exacta:** con estos nodos se comprueba si el personaje está en una estructura concreta. En esta categoría tenemos `OnHealingPoint` y `OnEnemyBase`.
 - **Comprobación de cercanía:** en estos nodos se comprueba si el agente está cerca de una estructura. A este categoría pertenecen los nodos `NearToEnemyBase` y `NearToTeamBase`.
 - **Comprobación de lejanía:** en estos nodos se comprueba si el agente está lejos de una estructura. A este categoría pertenecen los nodos `FarFromEnemyBase` y `FarFromTeamBase`.

Tanto para las comprobaciones de lejanía como de cercanía se usa la distancia de Chevychev.

Una vez vista las condiciones veamos las acciones que llevarán a cabo nuestros agentes.

4.2. Acciones

En lo que respecta a acciones tenemos 6 scripts distintos que en este caso heredarán de la clase `Action`. Estas acciones serán los nodos hoja de nuestros árboles de comportamiento, ya que se ejecutarán una vez realizadas todas las comprobaciones sobre las condiciones establecidas. En este caso la implementación de la acción requerida se encuentra en el método `OnUpdate`, veamos las diferentes implementaciones:

- **Atacar a un enemigo (AttackEnemy):** el agente ataca al enemigo más cercano que esté en su rango de ataque. Estos ataques se producen cada cierto número de segundos, que serán la velocidad de ataque del agente.

```

432 public void AttackEnemy(AgentNPC enemy)
433 {
434     if (Mathf.Approximately(_timerAttack, _attackSpeed))
435     {
436         this.RemoveAllSteeringsExcept(new List<string>()
437         {
438             SteeringNames.LookingWhereYoureGoing
439         });
440
441         Random random = new Random();
442         float damage = ((float) random.NextDouble() * (1f - 0.8f) + 0.8f) *
         _baseDamage;
443
444         enemy.GetDamage(damage);
445         _timerAttack -= Time.deltaTime;
446     }
447     else if (_timerAttack > 0 && _timerAttack < _attackSpeed)
448     {
449         _timerAttack -= Time.deltaTime;
450     }
451     else
452     {
453         _timerAttack = _attackSpeed;
454     }
455     this._actionState = ActionState.AttackEnemy;
456 }

```

- **Capturar base enemiga (CaptureEnemyBase):** el agente aporta puntos de captura a la base enemiga mientras se encuentre en ella.

```

706 public void CaptureEnemyBase()
707 {
708     if (Mathf.Approximately(captureTimer, captureSpeed))
709     {
710         this.RemoveAllSteeringsExcept(new List<string>()
711         {
712             SteeringNames.LookingWhereYoureGoing
713         });
714
715         enemyBase.GetCapturePoints(5);
716         captureTimer -= Time.deltaTime;
717     }
718     else if (captureTimer > 0f && captureTimer < captureSpeed)
719     {

```

```

720         captureTimer -= Time.deltaTime;
721     }
722     else
723     {
724         captureTimer = captureSpeed;
725     }
726     this._actionState = ActionState.CaptureBase;
727 }

```

- **Curarse (Heal):** el agente obtiene puntos de salud del punto de curación. Estos puntos de salud se recuperan en forma de pulsos cada cierto tiempo.

```

552 public void Heal()
553 {
554     this._actionState = ActionState.Heal;
555     if (_hpCurrent >= _hpMax)
556     {
557         return;
558     }
559
560     if (Mathf.Approximately(healTimer, healSpeed))
561     {
562         this.RemoveAllSteeringsExcept(new List<string>()
563         {
564             SteeringNames.LookingWhereYoureGoing
565         });
566
567         _hpCurrent = Mathf.Min(_hpCurrent + 10f, _hpMax);
568         healTimer -= Time.deltaTime;
569     }
570     else if (healTimer > 0f && healTimer < healSpeed)
571     {
572         healTimer -= Time.deltaTime;
573     }
574     else
575     {
576         healTimer = healSpeed;
577     }
578 }

```

- **Ir a una estructura:** el agente se dirige hacia una estructura como la base enemiga (GoToEnemyBase), su propia base (Defend) o el punto de curación (GoHealing).

```

363 protected void MoveToTarget(Building building)
364 {
365     this.RemoveAllSteeringsExcept(new List<string>()
366     {
367         SteeringNames.LookingWhereYoureGoing
368     });
369
370     PathFindingA steering = gameObject.AddComponent<PathFindingA>();
371     Vector2Int end = new Vector2Int(building.Center.y, building.Center.x);
372     steering.StartNode = grid.WorldPointToNode(Position);
373     steering.EndNode = grid.GridPointToNode(end - grid.Center);
374     steering.DistanceMethod = DistanceMethods.Chebychev;
375     steering.Weight = 5f;
376     steering.enabled = true;
377     listSteerings.Add(steering);
378 }

```

4.3. Comportamiento de los agentes

Ahora que ya se ha visto cómo serían todos los elementos del comportamiento táctico de los personajes, se procederá ver cómo se han unido todos estos elementos para crear

los árboles de comportamiento de los distintos tipos de unidades.

4.3.1. Infantería

Esta unidad se ha interpretado con una máxima prioridad sobre ganar la partida. Priorizará capturar la base enemiga sobre cualquier cosa. En **modo ataque** la unidad prioriza capturar la base enemiga sobre todas las cosas, seguido de atacar a los enemigos. Y después pensará en su salud, pero solo cuando no pueda capturar o atacara a un enemigo. En **modo defensa** defenderá atacando a los enemigos que vea y no pensará en curarse. Por ultimo, en **modo guerra total** actuará como en ataque pero sin pensar en curarse.

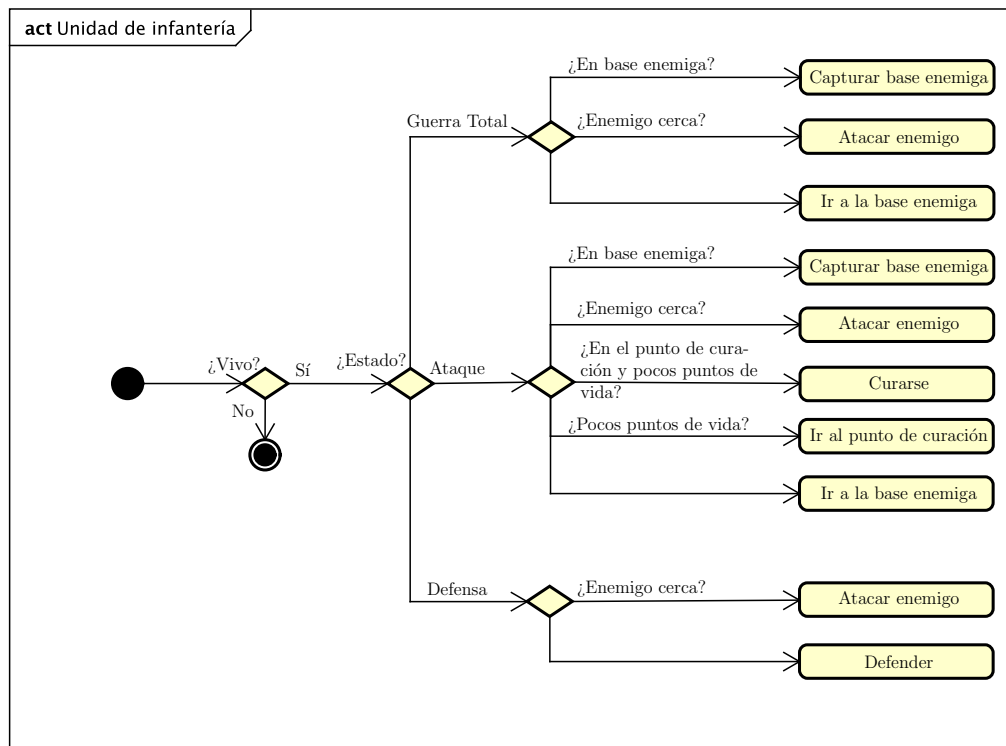


Figura 5: Árbol de comportamiento de infantería pesada

4.3.2. Lancero

Esta unidad funciona de forma similar a la infantería, solo que ha sido ideada como acompañamiento de esta. En el **modo ataque** y **modo guerra total** priorizaran atacar a enemigos antes que capturar, pero siempre priorizan atacar a los enemigos en a la base frente a su salud. En el **modo defensa** se comportará igual que en la infantería.

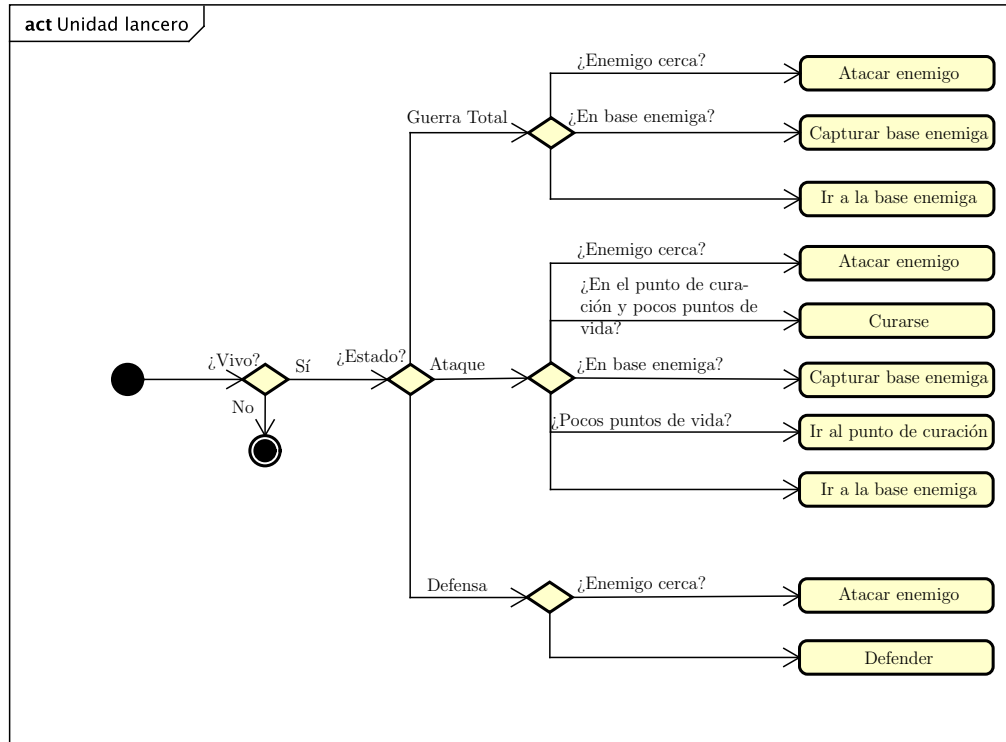


Figura 6: Árbol de comportamiento de lancero

4.3.3. Arquero

Para la unidad de tipo arquero, se ha pensado que sea una unidad “cobarde”. Esta prioriza su vida antes que el trabajo en equipo. A diferencia de las otras unidades cuando está en **modo ataque** buscará atacar y capturar la base enemiga, pero siempre que su vida baje peligrosamente huirá a curarse. En **modo defensa** también encontramos esta característica, si la unidad se ve muy dañada, irá a curarse a diferencia de las anteriores unidades que simplemente defendían.

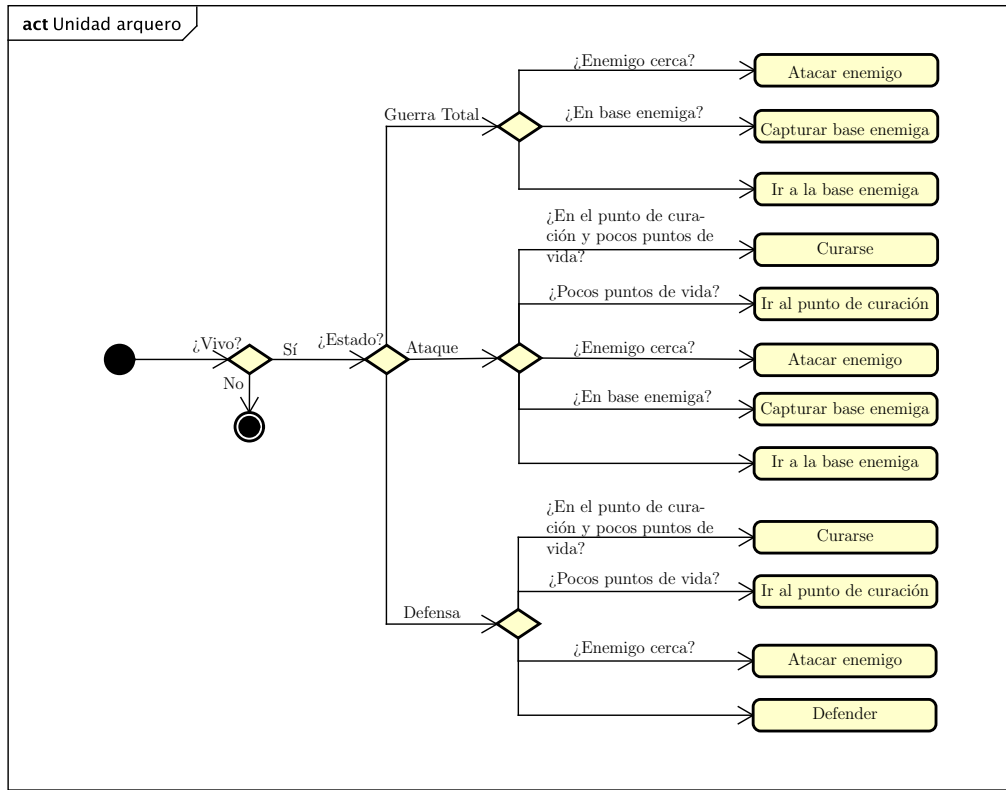


Figura 7: Árbol de comportamiento de arquero

4.3.4. Caballería

Esta unidad podría considerarse la más distinta de todas. Su comportamiento se basa en desplazarse en modo de “patrulla” yendo y viniendo de su propia base. Comprobará si está demasiado lejos de su base en todo momento, y en caso de estarlo, procederá a regresar. Hay que remarcar que si se cruza con un enemigo procederá a atacarlo.

Este comportamiento se mantendrá siempre que esté en **modo ataque** y en **modo defensa**. En caso de estar en **modo guerra total** simplemente atacará la base enemiga. Cabe recalcar que esta unidad al centrarse en patrullar, nunca llegará a atacar la base enemiga, ni a defender su propia base.

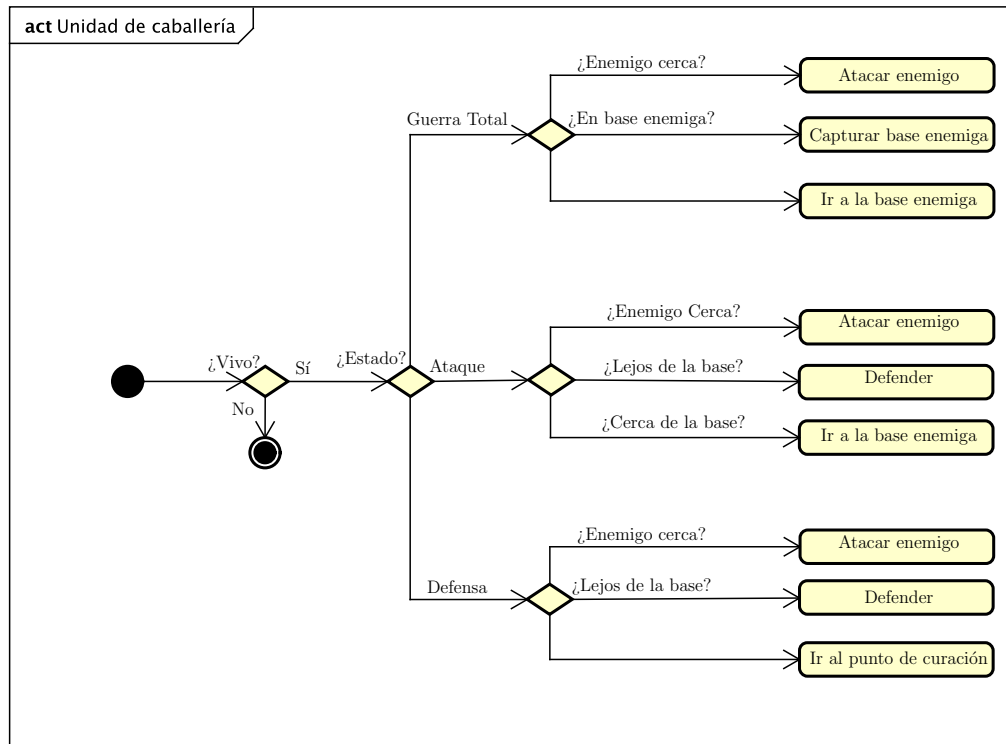


Figura 8: Árbol de comportamiento de caballería

5. Comportamiento Estratégico

5.1. Modos de comportamiento

El comportamiento estratégico implementado a nivel de bando/grupo se ha hecho de forma que tenemos varios modos.

- **Ataque:** El bando que esté en modo ataque irá a por la base enemiga y todo aquel que se encuentre por su paso lo tratará de eliminar.
- **Defensa:** Este modo es para que el equipo seleccionado se centre en defender su base de los enemigos y priorizarán también la cura de los personajes.
- **Guerra Total:** En este modo ambos equipos pasan al ataque y ganará el primero en conquistar la base enemiga

La manera en la que se adapta el comportamiento según el modo es sencilla basándonos en el uso de la librería Behaviour Trees [2]. Tenemos la clase **State** que recoge todos los modos de juego y cada personaje tendrá un atributo de este tipo.

```
5 public enum State
6 {
7     TotalWar,
8     Attack,
9     Defense
10 }
```

El cambio de modo es a través de los botones, donde cada botón llamará al método de la clase **GameController** correspondiente:

```
105 }
106
107 /**
108  * @brief Pone al equipo A en modo ataque.
109  */
110
111 }
112
113 /**
114  * @brief Pone al equipo B en modo ataque.
115  */
116
117 }
118
119 /**
120  * @brief Pone al equipo A en modo defensa.
121  */
122
123 }
124
125 /**
126  * @brief Pone al equipo B en modo defensa.
127  */
128
129 }
130
131 /**
132  * @brief Pone a todos los equipos en modo guerra total.
133  */
134
135 }
```

El método que contiene toda la lógica de los cambios de modo es **SwitchTeamMode**, que se encarga de buscar dentro de todos los NPCs que tenemos aquellos que coinciden con el equipo pasado por parámetro y el estado se modificará al que se tiene como segundo parámetro.

```
89 List<Agent> allAgents = GetNearAgents(agent, distance);
90 return allAgents.Where(a => !a.Team.Equals(agent.Team)).ToList();
91 }
92
93 // Cambia el modo del equipo team
94 protected void SwitchTeamMode(Teams team, State state)
95 {
96 List<AgentNPC> teamAgents = GameObject.FindGameObjectsWithTag("NPC").ToList()
97     .Select(a => a.GetComponent<AgentNPC>())
98     .Where(a => a.Team.Equals(team))
99     .ToList();
```

6. Sistema de combate

El sistema de combate consiste en simples ataques entre los distintos personajes donde la evolución del combate dependerá de las distintas características de las unidades. Estos para atacar necesitan estar en rango de ataque, teniendo cada uno un rango específico, por ejemplo, los arqueros tienen más rango que el resto. Veamos los atributos que toman parte en el combate y los valores que toman estos para cada personaje:

- **_baseDamage**: Daño base de la unidad, varía según el tipo.
- **_attackRange**: Distancia máxima a la que se considera que un objetivo está cerca, y por tanto se le puede atacar.
- **_attackSpeed**: Velocidad a la que ataque el agente.
- **_hpMax**: Puntos de vida máximos del agente.
- **_hpCurrent**: Puntos de vida actuales del agente.
- **healSpeed**: Velocidad a la que recupera vida el agente.
- **captureSpeed**: Velocidad de captura de la base enemiga.

En la siguiente tabla podemos ver cómo varían estos atributos según la unidad:

Unidad	Daño base	Rango de ataque	Velocidad de ataque	Vida máxima
Lancero	40	6	4	250
Infantería	10	6	4	200
Caballería	30	6	4	130
Arquero	20	14	4	100

Tabla 4: Tabla de Influencias

Una vez estamos en rango de ataque el agente hará un daño aleatorio entre su rango, este podrá ir entre el 80 – 100 % del mismo. Tras esto, para seguir atacando el NPC tendrá que esperar según su velocidad de ataque.

Es el método **AttackEnemy** el que incluye la lógica de combate entre unidades, donde podemos ver cómo se obtiene un valor aleatorio entre 0.8 y 1 que se usará como factor multiplicativo para el ataque base del agente. Posteriormente se llama al método **GetDamage** del enemigo, para que reduzca sus puntos de vida (en el caso de estar en su punto de curación el agente no recibirá daño).

```

432 public void AttackEnemy (AgentNPC enemy)
433 {
434     if (Mathf.Approximately(_timerAttack, _attackSpeed))
435     {
436         this.RemoveAllSteeringsExcept(new List<string>()
437         {
438             SteeringNames.LookingWhereYoureGoing
439         });
440
441         Random random = new Random();

```

```

442     float damage = ((float) random.NextDouble() * (1f - 0.8f) + 0.8f) *
    _baseDamage;
443
444     enemy.GetDamage(damage);
445     _timerAttack -= Time.deltaTime;
446 }
447 else if (_timerAttack > 0 && _timerAttack < _attackSpeed)
448 {
449     _timerAttack -= Time.deltaTime;
450 }
451 else
452 {
453     _timerAttack = _attackSpeed;
454 }
455 this._actionState = ActionState.AttackEnemy;
456 }

```

Una vez se ha terminado un combate hay 2 posibilidades. Si sigue vivo el personaje, seguirá con su funcionamiento definido; podrá ir a curarse, seguir atacando, defender, etc. La otra posibilidad que queda es que ese personaje haya muerto, si este es el caso reaparecerá después de un tiempo en su base.

6.1. Condición de victoria

En cuanto a las condiciones de victoria se ha decidido implementar solamente una, que será la de conseguir capturar la base enemiga ya que se supone que es la más completa a la hora de mostrar el funcionamiento de todo lo implementado en el juego pudiendo probar los comportamientos de pathFinding, influencia y el ataque entre unidades.

Será el método `CaptureEnemyBase` de la clase `AgentNPC` el que implemente este comportamiento, donde podemos ver cómo se realizan las mismas comprobaciones para que se apliquen los puntos de captura cada X segundos y dónde se llama al método de la base enemiga que le añade puntos de captura.

```

706 public void CaptureEnemyBase()
707 {
708     if (Mathf.Approximately(captureTimer, captureSpeed))
709     {
710         this.RemoveAllSteeringsExcept(new List<string>()
711         {
712             SteeringNames.LookingWhereYoureGoing
713         });
714
715         enemyBase.GetCapturePoints(5);
716         captureTimer -= Time.deltaTime;
717     }
718     else if (captureTimer > 0f && captureTimer < captureSpeed)
719     {
720         captureTimer -= Time.deltaTime;
721     }
722     else
723     {
724         captureTimer = captureSpeed;
725     }
726     this._actionState = ActionState.CaptureBase;
727 }

```

Una vez completados los puntos de captura saltará la condición de victoria, se pausará el juego y aparecerá una ventana indicando cuál ha sido el equipo ganador.

7. Mapa Táctico


En nuestro juego RTS se ha implementado un mini mapa que nos permite tener en la esquina inferior derecha los distintos mapas de influencia y además la IA táctica se puede adaptar su comportamiento según esta influencia. En este apartado se verán los distintos mapas implementados y su influencia táctica.

Este mapa táctico en un principio se puede ver en forma de minimapa que permite apreciar cómo varía la influencia en los distintos puntos del mapa. Esta visión táctica como ya se ha explicado se podrá alternar con el mapa de juego y así poder visualizarla en pantalla completa a la vez que cambiar el tipo de mapa táctico, mientras que el mapa de juego permanece en el minimapa.

Veamos primero cómo se ha implementado el minimapa y después entraremos más en detalle de los mapas tácticos.

7.1. Minimapa

Utilizando un canvas de tamaño más pequeño se ha creado un panel en la esquina inferior derecha en donde se proyectarán diferentes texturas en tiempo real. Tenemos la cámara enfocada en nuestro terreno de juego principal y otra cámara enfocada en el terreno de las influencias y las texturas en tiempo real serán las que contendrán aquello que renderice la cámara.

Si presionamos la tecla  podemos hacer que el mapa principal pase al mini mapa y así ver en grande los distintos mapas de influencias. Además en esta vista tenemos un botón que nos permite cambiar entre 3 tipos de mapa: influencia, vulnerabilidad y tensión.

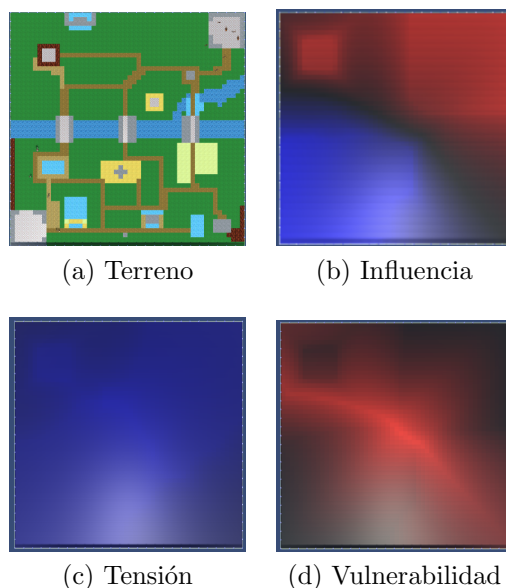


Figura 9: Tipos de mapas

7.2. Creación mapa táctico

El mapa táctico será un grid de tamaño 52×52 al igual que el mapa de juego, en este caso este tomará colores según los valores de influencia.

7.2.1. Mapa de Influencia

El mapa de influencia se basa en que los distintos NPCs y puntos de interés del juego tienen una influencia distintos, y estas influencias afectarán de distinta forma al comportamiento que tendrán los agentes a partir de la información que reciban de su entorno. Esta información usada por el agente puede ser el tipo de terreno, los agentes que tiene cerca, puntos de curación, etc. Este comportamiento influirá en donde se dirigen los personajes a la hora de atacar y defender.

La influencia se calcula periódicamente y se suma a la influencia previa que hubiera en el terreno. Del mismo modo la influencia también decae pasado un cierto tiempo.

Para calcular la influencia de los distintos equipos usamos la función `UpdateTeamsInfluence` dentro de la clase `GridController`. Esta función obtiene la influencia previa de cada equipo (al comienzo del juego será 0), y por cada personaje y estructura perteneciente a ese equipo les asigna una influencia en su posición que posteriormente será extendida por todo el mapa.

```

86 private void UpdateTeamsInfluence()
87 {
88     List<Teams> teamsList = Enum.GetValues(typeof(Teams)).Cast<Teams>().
89         Where(t => !t.Equals(Teams.None)).ToList();
90
91     Transform teamTransform;
92     float[,] newInfluence;
93     Vector2Int pos;
94
95     foreach (Teams team in teamsList)
96     {
97         teamTransform = GameObject.Find(team.ToString()).transform;
98         newInfluence = GetLastInfluence(team);
99
100        foreach (Transform child in teamTransform)
101        {
102            if (!child.gameObject.activeSelf)
103            {
104                continue;
105            }
106
107            pos = _grid.WorldToGridPoint(child.position);
108            Vector2Int auxPos = new Vector2Int(pos.y, pos.x);
109            ExtendInfluence(newInfluence, auxPos, TypeToInfluence(child.tag));
110        }
111
112        _grid.UpdateCellsInfluence(team, newInfluence, mapType);
113    }
114
115    // Actualizamos la influencia de las estructuras
116    Building[] buildings = GameObject.Find("Buildings").GetComponentsInChildren<
117    Building>();
118    foreach (Building building in buildings)
119    {
120        if (building.Team.Equals(Teams.None))
121        {
122            continue;

```



```

122     }
123
124     newInfluence = GetLastInfluence(building.Team);
125     foreach (Vector2Int buildingCell in building.Cells)
126     {
127         ExtendInfluence(newInfluence, buildingCell - _grid.Center, TypeToInfluence
128         ("Building"));
129     }
130     _grid.UpdateCellsInfluence(building.Team, newInfluence, mapType);
131 }
132 }

```

Como se puede ver en el código para extender la influencia se hace uso de la función **ExtendInfluence**. Esta función recorre todo el terreno y asigna una influencia determinada dependiendo de la distancia. La fórmula usada para calcular como se extiende la influencia es la siguiente.

$$Influencia_{i,j} = \frac{Influencia_{base}}{1 + d((i,j), (x,y))}$$

donde i, j representan a la casilla dentro del terreno, x, y al objeto que representa el centro de donde extiende, e $Influencia_{base}$ representa la influencia que desprende ese objeto. Por último, comentar que se utiliza la distancia de Chevychev para este cálculo.

$$d((i,j), (x,y)) = \max(|x - i|, |y - j|)$$

La influencia que desprende cada objeto, dependiendo de su tipo es:

Jugador	0.75
Estructura	0.25
NPC	0.5

Tabla 5: Influencia por tipo de objeto

El mapa de influencia se representará como un mapa de calor con los colores rojo y azul, donde el rojo representa influencia máxima del equipo B y el azul del A. La influencia mostrada en el mapa se calculará de la siguiente forma.

$$Influencia = Influencia_A - Influencia_B$$

donde $Influencia_A, Influencia_B \in [0, 1]$ y, por tanto, $Influencia \in [-1, 1]$.

Esta información de influencias será usada como información táctica a la hora de calcular el comportamiento táctico de los personajes

7.2.2. Mapas de Tensión y Vulnerabilidad

Estos mapas se calculan a través de los valores de influencia de los distintos equipos y aportan información visual sobre el estado de la partida actual.

$$Tension = \frac{Influencia_A + Influencia_B}{2}$$

donde $Influencia_A, Influencia_B \in [0, 1]$ y, por tanto, $Tension \in [0, 1]$.

$$Vulnerabilidad = Tension - |Influencia|$$

donde $Tension, |Influencia| \in [0, 1]$ y, por tanto, $Vulnerabilidad \in [-1, 1]$.

8. Pathfinding táctico individual

8.1. Algoritmo A*

En esta sección se explicará el funcionamiento del pathfinding táctico individual de los personajes. Para esta labor se ha hecho uso del algoritmo A* (1).

Algoritmo 1 Algoritmo A*

```
1: procedure A*(grid, inicio, final,  $h$ )    ▷  $h$  es la función heurística admisible
2:   Poner inicio en lista de ABIERTOS con  $f(\text{inicio}) = h(\text{inicio})$ 
3:   while lista de ABIERTOS no esté vacía do
4:     Obtener de la lista de ABIERTOS el nodo actual con menor  $f(\text{nodo})$ 
5:     if actual = final then                ▷ Se ha encontrado una solución
6:       break
7:     end if
8:     Conseguir todos los nodos sucesor de actual
9:     for cada sucesor de actual do
10:      Establecer  $\text{coste\_sucesor} = g(\text{actual}) + w(\text{actual}, \text{sucesor})$  ▷  $w(a, b)$ 
      es el coste del camino entre  $a$  y  $b$ 
11:      if actual está en la lista de ABIERTOS then
12:        if  $g(\text{sucesor}) \leq \text{coste\_sucesor}$  then
13:          continue
14:        end if
15:      else if sucesor está en la lista de CERRADOS then
16:        if  $g(\text{sucesor}) \leq \text{coste\_sucesor}$  then
17:          continue
18:        end if
19:      Mover sucesor de la lista de CERRADOS a la de ABIERTOS
20:    else
21:      Añadir sucesor a la lista de ABIERTOS
22:    end if
23:    Establecer  $g(\text{sucesor}) = \text{coste\_sucesor}$ 
24:    Establecer actual como nodo padre de sucesor
25:  end for
26:  Añadir actual a la lista de CERRADOS
27: end while
28: if actual  $\neq$  final then                ▷ No se ha encontrado camino
29:   Terminar con error.
30: end if
31: end procedure
```

Este algoritmo se ha implementado casi de manera literal. Su mayor cambio viene por la parte de calcular el coste del sucesor. En este caso no sólo se ha tenido en cuenta el coste de desplazarse del nodo actual al vecino, sino que se ha tenido en cuenta el tipo de terreno así como la influencia enemiga. Por lo tanto el código implementado sería:

```

37 float newCost = current.GCost + agent.GetTerrainCost(neighbour.TerrainType);
38
39 float influenceValue = neighbour.InfluenceValue;
40 switch (agent.Team)
41 {
42     case Teams.TeamA:
43     {
44         newCost -= influenceValue;
45         break;
46     }
47     case Teams.TeamB:
48     {
49         newCost += influenceValue;
50         break;
51     }
52 }

```

El camino resultante de este algoritmo se guardará en la variable `path` del agente correspondiente.

8.2. Pathfinding basado en A*

Una vez tenemos una implementado el algoritmo A*, tenemos el camino a seguir por el agente. Este camino se compone de una lista de nodos que el agente recorrerá uno por uno.

En este caso el steering de Pathfinding no tiene que generar camino a no ser que no exista uno. El steering generará un camino y posteriormente sólo irá recorriéndolo en orden.

Lo primero que hace el steering es comprobar si existe camino y lo generará si este no existe.

```

67 _path = agent.Path;
68
69 if (_path == null)
70 {
71     GeneratePath(agent);
72     _path = agent.Path;
73 }

```

La función `GeneratePath` inicializa el grafo y ejecuta el algoritmo A*.

```

123 private void GeneratePath(Agent agent)
124 {
125     InitializeCost();
126     A.AStar(_grid, _start, _end, _heuristic, agent);
127 }

```

La inicialización del grafo se hace estableciendo $g(n) = \infty$ para todos los nodos del grafo, y $h(n) = \infty$ para aquellos que no sean transitables. Por último se inicializan tanto el nodo inicial, $g(\text{inicio}) = 0$, como el final $h(\text{final}) = 0$. El nodo inicial también conserva su valor heurístico $h(\text{start}) \neq \infty$.

```

103 private void InitializeCost()
104 {
105     Node n;
106     for (int i = 0; i < _grid.NumCellsX; i++)
107     {

```

```

108     for (int j = 0; j < _grid.NumCellsY; j++)
109     {
110         n = _grid.GetNode(i, j);
111         n.GCost = Mathf.Infinity;
112         if (!n.Passable)
113         {
114             n.HCost = Mathf.Infinity;
115         }
116     }
117 }
118 _start.GCost = 0f;
119 _start.HCost = _heuristic.EstimateCost(_start);
120 _end.HCost = 0f;
121 }

```

Una vez que ya se tiene el camino, ya sea porque lo acabamos de generar, o bien, porque ya existía comprobamos si el agente ha llegado al siguiente nodo y si este es el final. En caso de no haber llegado al final, pero sí al siguiente nodo del camino se cambia el objetivo.

```

75 if (!_currentNode.Equals(_end) && AtNode(agent))
76 {
77     _currentNode = _path.ElementAt(_path.IndexOf(_currentNode) + 1);
78 }

```

Por último, el agente se mueve al nuevo nodo delegando en el steering Arrive.

```

80 // Creamos un personaje auxiliar
81 GameObject auxiliar = new GameObject("DetectPlayer");
82 Target = auxiliar.AddComponent<Agent>();
83 auxiliar.tag = "Auxiliar";
84
85 // Le asignamos la posicion necesaria
86 Target.Position = _currentNode.WorldPosition;
87
88 // Delegamos en Seek
89 steer = base.GetSteering(agent);
90
91 // Borramos al personaje auxiliar y volvemos a asignar el antiguo
92 Destroy(auxiliar);
93
94 // Retornamos el steering
95 return steer;

```

En caso de que el agente estuviera en el nodo final, el nodo objetivo no cambia y por tanto se propone moverse al mismo nodo en el que está, por lo que no se produce ningún movimiento.

9. Modo Depuración

La información respecto al movimiento de nuestros agentes la podemos consultar en el inspector en la parte derecha de unity siempre y cuando no ejecutemos el juego en pantalla completa. Para intentar hacer que algunos valores se consulten de forma más intuitiva se ha hecho uso de Gizmos y también de imágenes auxiliares.

Empezando por los Gizmos, estos son una clase especial de Unity que nos permite que sea activada y desactivada durante la ejecución a través del inspector como se ve en la figura 10 y estos serán visibles en la escena.

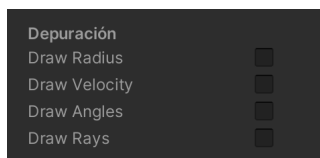


Figura 10: Gizmos disponibles

Para que una clase pueda ‘depurar’ sus atributos será necesario que esta tenga un método llamado `OnDrawGizmos`. Veamos por ejemplo el de la clase `Agent` que muestra los atributos de la figura anterior.

```

190 private void OnDrawGizmos()
191 {
192     if (drawAngles)
193     {
194         Gizmos.color = Color.red;
195         Gizmos.DrawLine(Position,
196             Position + OrientationToVector(Orientation + _interiorAngle) *
197             DefaultLineLength);
198         Gizmos.DrawLine(Position,
199             Position + OrientationToVector(Orientation - _interiorAngle) *
200             DefaultLineLength);
201
202         Gizmos.color = Color.yellow;
203         Gizmos.DrawLine(Position,
204             Position + OrientationToVector(Orientation + _exteriorAngle) *
205             DefaultLineLength);
206         Gizmos.DrawLine(Position,
207             Position + OrientationToVector(Orientation - _exteriorAngle) *
208             DefaultLineLength);
209     }
210
211     if (drawVelocity)
212     {
213         Gizmos.color = Color.blue;
214         Gizmos.DrawLine(Position, Position + Velocity);
215     }
216
217     if (drawRadius)
218     {
219         Gizmos.color = Color.magenta;
220         Gizmos.DrawWireSphere(Position, _interiorRadius);
221
222         Gizmos.color = Color.blue;
223         Gizmos.DrawWireSphere(Position, _arrivalRadius);
224     }
225
226     if (drawRays && _rays != null)

```

```
223     {
224         Gizmos.color = Color.blue;
225         foreach (Vector3 ray in _rays)
226         {
227             Gizmos.DrawLine(Position, Position + ray);
228         }
229     }
230 }
```

En este bloque de código vemos como se comprueban si están activados cada una de las casillas. Cada subbloque establece el color y la forma en la que se verán los distintos gizmos. Por ejemplo, en el subbloque de `drawRadius`, se puede ver como se dibujan dos círculos de distintos colores alrededor del agente, representando cada uno de ellos los radios del agente.

El otro elemento introducido a modo de depuración del comportamiento de los agentes, ha sido el de poder conocer el estado en el que está cada uno mediante el icono de estado que aparece a la izquierda del personaje. La implementación de este mecanismo ya se explicó en la sección 1, *Interfaz gráfica y Jugabilidad*.

Referencias

- [1] Polygon Blacksmith. Toon RTS Units. URL: <https://assetstore.unity.com/packages/3d/characters/toon-rts-units-67948>.
- [2] Naohiro Yoshida. Behavior Tree designer for Unity. Ver. 0.0.3. Dic. de 2021. URL: <https://github.com/yoshidan/UniBT>.