

**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

<b>I</b>	<b>Bloque 2</b>	<b>3</b>
<b>1.</b>	<b>Mapa táctico</b>	<b>3</b>
1.1.	Mini mapas . . . . .	3
1.2.	Comportamiento táctico: Map de Influencia . . . . .	4
<b>2.</b>	<b>Interfaz gráfica y Jugabilidad</b>	<b>5</b>
<b>3.</b>	<b>Campo de batalla</b>	<b>6</b>
<b>4.</b>	<b>Tipos de Unidades</b>	<b>7</b>
<b>5.</b>	<b>Comportamiento táctico</b>	<b>11</b>
5.1.	Condiciones . . . . .	11
5.2.	Acciones . . . . .	12
5.3.	Comportamiento de los agentes . . . . .	13
5.3.1.	Infantería . . . . .	14
5.4.	Lancero . . . . .	14
5.5.	Arquero . . . . .	15
5.6.	Caballería . . . . .	16
<b>6.</b>	<b>Comportamiento Estratégico</b>	<b>18</b>
6.1.	Modos de comportamiento . . . . .	18
	<b>Referencias</b>	<b>20</b>

## Índice de figuras

1.	Interfaz gráfica estrategia . . . . .	5
2.	Interfaz gráfica cambio de mapa . . . . .	5
3.	Terreno de juego . . . . .	7
4.	Árbol de comportamiento de infantería pesada . . . . .	14
5.	Árbol de comportamiento de lancero . . . . .	15
6.	Árbol de comportamiento de arquero . . . . .	16
7.	Árbol de comportamiento de caballería . . . . .	17

## 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. 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 verá los distintos mapas implementados y su influencia táctica.

### 1.1. Mini mapas

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.

## 1.2. Comportamiento táctico: Map 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.

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

## 2. 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

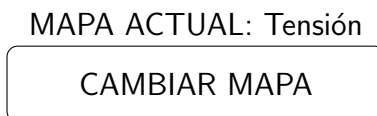


Figura 2: Interfaz gráfica cambio de mapa

### 3. 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. Sin embargo, el agua más clara y menos profunda puede ser atravesada por unidades montadas o unidades ligeras, ya que las más pesadas pueden ser susceptibles de hundirse aquí.
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.

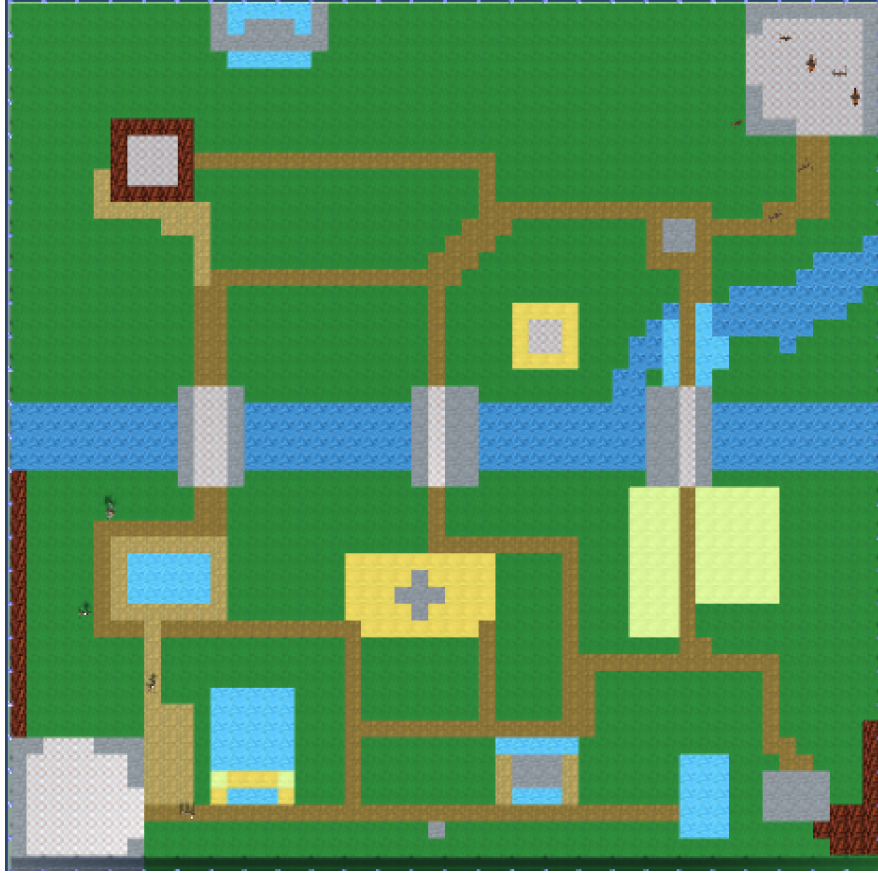


Figura 3: Terreno de juego

## 4. 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 a caballo
- Unidad arquera
- 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 = 3f;
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 2. 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.

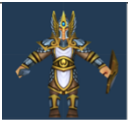
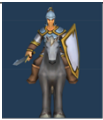












						
			1.5	1.5	1.5	1.5
			10	10	3	3
			$\infty$	$\infty$	$\infty$	$\infty$
			5	1	5	1
			1	5	1	1
			15	15	15	15

Tabla 1: Tabla de Influencias

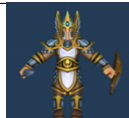
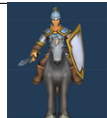












						
			75 %	75 %	90 %	90 %
			25 %	25 %	60 %	60 %
			0 %	0 %	0 %	0 %
			50 %	100 %	45 %	100 %
			100 %	50 %	100 %	100 %
			5 %	5 %	10 %	2 %

Tabla 2: Tabla de velocidades

## 5. 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.

### 5.1. 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 esta categoría pertenecen los nodos `NearToEnemyBase` y `NearToTeamBase`.
  - **Comprobación de lejanía:** en estos nodos se comprueba si el agente está lejos de una estructura. A esta 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.

## 5.2. 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.

```

411 }
412 _hpCurrent -= damage;
413 if (_hpCurrent <= 0)
414 {
415     Debug.Log("Personaje muerto");
416     DeadNPC();
417 }
418 }
419
420 /**
421 * @brief Atacar a un enemigo.
422 * @param[in] enemy Enemigo.
423 */
424 public void AttackEnemy(AgentNPC enemy)
425 {
426     if (Mathf.Approximately(_timerAttack, _attackSpeed))
427     {
428         this.RemoveAllSteeringsExcept(new List<string>()
429         {
430             SteeringNames.LookingWhereYoureGoing
431         });
432
433         Random random = new Random();
434         float damage = ((float) random.NextDouble() * (1f - 0.8f) + 0.8f) *
         _baseDamage;

```

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

```

679 /**
680 * @brief Comprueba si el agente está cerca de la base enemiga.
681 * @return true si está cerca.
682 */
683 public bool NearToEnemyBase()
684 {
685     return NearToBuilding(enemyBase);
686 }
687
688
689 /**
690 * @brief El agente aplica una cantidad de puntos de captura cada cierto tiempo a
        la base enemiga.
691 */
692 public void CaptureEnemyBase()

```

```

693 {
694     if (Mathf.Approximately(captureTimer, captureSpeed))
695     {
696         this.RemoveAllSteeringsExcept(new List<string>()
697         {
698             SteeringNames.LookingWhereYoureGoing
699         });

```

- **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.

```

526 }
527
528 /**
529  * @brief Se mueve al punto de curación.
530  */
531 public void GoHealing()
532 {
533     MoveToTarget(healingPoint);
534 }
535
536 /**
537  * @brief El personaje recibe vida cada cierto tiempo.
538  */
539 public void Heal()
540 {
541     if (_hpCurrent >= _hpMax)
542     {
543         return;
544     }
545
546     if (Mathf.Approximately(healTimer, healSpeed))
547     {
548         this.RemoveAllSteeringsExcept(new List<string>()
549         {
550             SteeringNames.LookingWhereYoureGoing
551         });

```

- **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).

```

343 {
344     // Calcula el steeringbehaviour
345     Steering kinematic = behavior.GetSteering(this);
346
347     // Usar kinematic con el árbitro desesado para combinar todos los
348     // Llamaremos kinematicFinal a la aceleraciones finales.
349     Steering kinematicFinal = Arbitro.GetSteering(behavior, kinematic);
350
351     // El resultado final se guarda para ser aplicado en el siguiente frame.
352     this.steer += kinematicFinal;
353 }
354
355 // Moverse hacia una estructura.
356 protected void MoveToTarget(Building building)
357 {
358     this.RemoveAllSteeringsExcept(new List<string>()

```

### 5.3. Comportamiento de los agentes

Ahora que ya hemos visto cómo serían todos los elementos del comportamiento táctico de los personajes, podemos ver cómo se han juntado todos estos elementos para crear

los árboles de comportamiento de los distintos tipos de unidades.

### 5.3.1. Infantería

Esta unidad la hemos 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** se actuará como en ataque pero sin pensar en curarse.

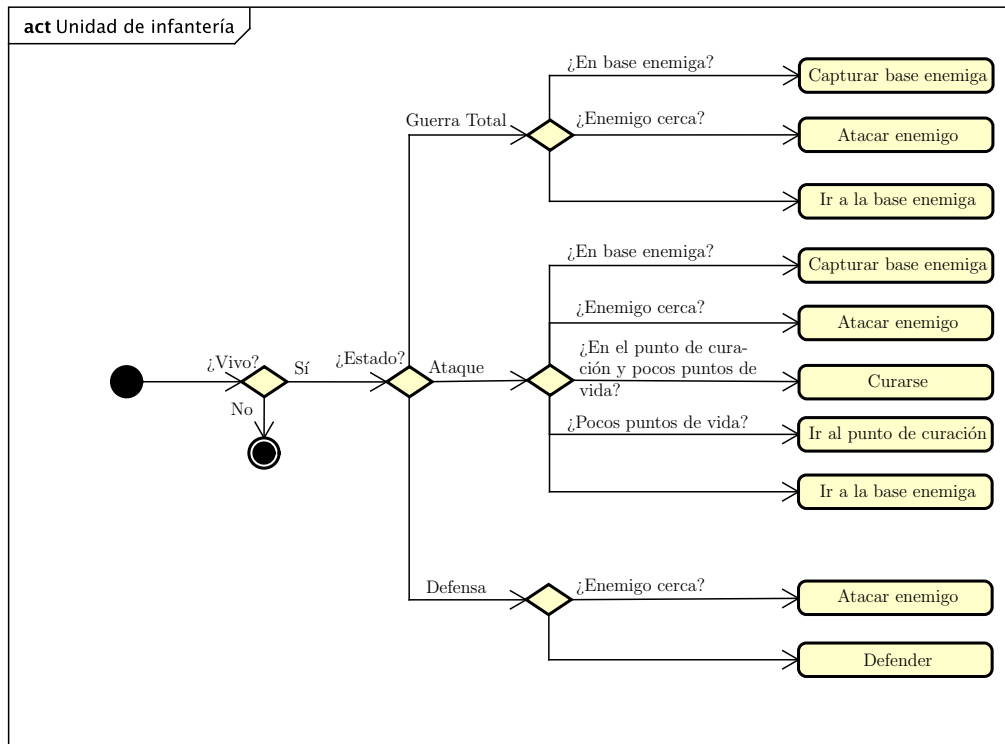


Figura 4: Árbol de comportamiento de infantería pesada

## 5.4. Lancero

Esta unidad funciona de forma similar a la infantería, solo que la hemos ideado 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.

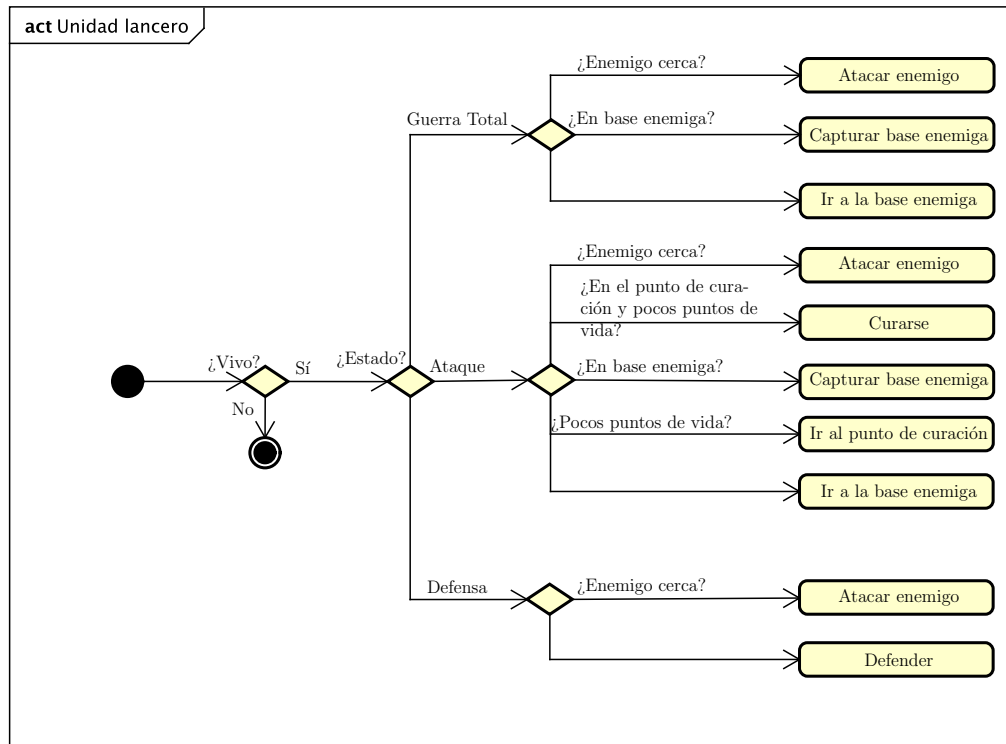


Figura 5: Árbol de comportamiento de lancero

## 5.5. Arquero

Para la unidad de tipo arquero, hemos ideado que sea una unidad “cobarde”, esta prioriza su vida antes que el trabajo en equipo. A diferencia de las otras unidades esta en el **modo ataque** buscará atacar y capturar la base enemiga, pero siempre que su vida baje peligrosamente huirá a curarse. En el **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.



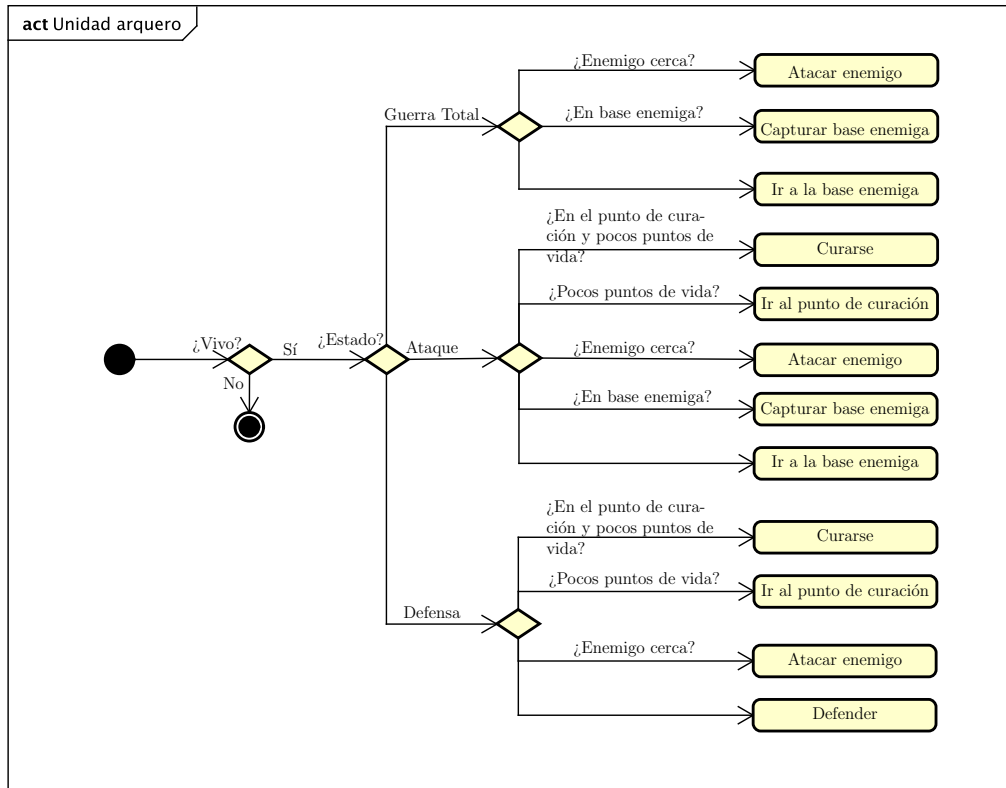


Figura 6: Árbol de comportamiento de arquero

## 5.6. Caballería

Esta unidad la podríamos considerar la mas distintas. de las 4. Su comportamiento se basa en desplazarse en modo de "patrullazendo y viniendo a la base. Irá compronbando que se aleje demasiado de su base para volver, y si se encuentra a cualquier enemigo le atacará. Este comportamiendo lo mantendrá siempre que esté **modo ataque** y en **modo defensa**. Pero si le ponemos **modo guerra total** simplemente atacara la base. Cabe recalcar que esta unidad al centrarse en patrullar, nunca llegará a atacar la base enemiga, ni a defender su propia base.

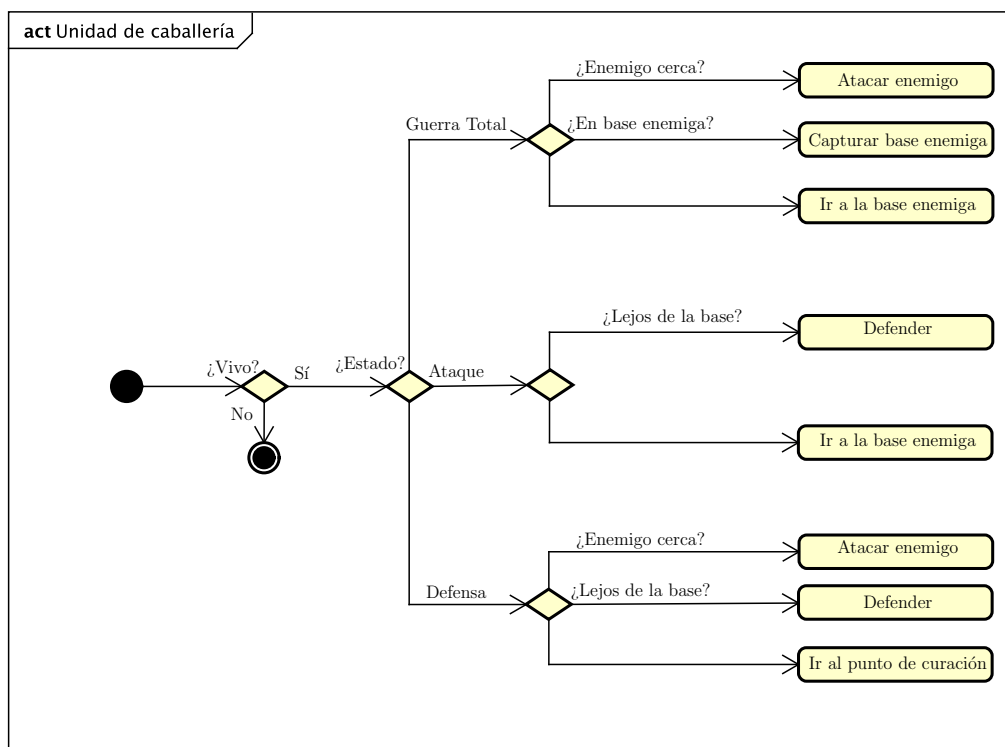


Figura 7: Árbol de comportamiento de caballería

## 6. Comportamiento Estratégico

### 6.1. 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 public void AttackModeA()
106 {
107     SwitchTeamMode(Teams.TeamA, State.Attack);
108 }
113 public void AttackModeB()
114 {
115     SwitchTeamMode(Teams.TeamB, State.Attack);
116 }
121 public void DefenseModeA()
122 {
123     SwitchTeamMode(Teams.TeamA, State.Defense);
124 }
129 public void DefenseModeB()
130 {
131     SwitchTeamMode(Teams.TeamB, State.Defense);
132 }
137 public void TotalWar()
138 {
139     SwitchTeamMode(Teams.TeamA, State.TotalWar);
140     SwitchTeamMode(Teams.TeamB, State.TotalWar);
141 }
```

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 protected void SwitchTeamMode(Teams team, State state)
90 {
91     List<AgentNPC> teamAgents = GameObject.FindGameObjectsWithTag("NPC").ToList()
92         .Select(a => a.GetComponent<AgentNPC>())
93         .Where(a => a.Team.Equals(team))
94         .ToList();
95
96     foreach (AgentNPC agent in teamAgents)
97     {
98         agent.State = state;
99     }
100 }
```

## 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>.