



Escuela  
Politécnica  
Superior

# Gestión y manejo de comportamientos grupales emergentes de entidades en videojuegos



Grado en Ingeniería Multimedia

## Trabajo Fin de Grado

Autor:

Borja Pozo Wals

Tutor/es:

Francisco José Gallego Durán

Octubre 2020



Universitat d'Alacant  
Universidad de Alicante



# Gestión y manejo de comportamientos grupales de entidades en videojuegos

---

**Subtítulo del TFG**

**Autor**

Borja Pozo Wals

**Tutor/es**

Francisco José Gallego Durán

*Departamento de Ciencia de la Computación e Inteligencia Artificial*



GRADO EN INGENIERÍA MULTIMEDIA



Escuela  
Politécnica  
Superior



Universitat d'Alacant  
Universidad de Alicante

ALICANTE, 12 de febrero de 2021



# Resumen

**título** es una herramienta para simular escenarios bélicos como los que podemos encontrar en juegos del género *Real Time Strategy (RTS)* como son ‘*They are Billions (TaB)*’ o la saga ‘*Age of Empires (AoE)*’, desarrollado completamente en C++ para Personal Computer (PC).

La “demo” se compone de un escenario, una serie de unidades controladas por el jugador que conforman el ejercito bajo sus ordenes y por último, tendremos una serie de objetivos enemigos que deberemos abatir. Una vez conseguido el reto propuesto terminará el juego pudiendo el jugador elegir entre repetir el nivel o volver al menú principal.

El proyecto desarrollado utiliza estas herramientas y está destinado a los siguientes sistemas:

- **Lenguaje:** C++17 y Makefile
- **Compilador Linux:** GCC v10.2.0
- **Compilador Windows:** MinGW vX.XX.X
- **SO:** Windows 10 y Manjaro v20.2



# Justificación y objetivos

A lo largo de la carrera son muchas las asignaturas que requieren y desarrollan habilidades relacionadas con la programación en diversos lenguajes y usos, pero no es hasta el tercer año que se me presenta la oportunidad de desarrollar un videojuego completo.

Durante la asignatura de ‘Fundamentos de los Videojuegos’ tuve por primera vez la experiencia de enfrentarme al desafío que es crear un videojuego, y fue en ese momento cuando me di cuenta de que a lo que me quería dedicar es a desarrollar juegos de forma profesional. A lo largo del cuarto curso junto a los demás integrantes de ‘*Sunlight Studio*’ desarrollamos ‘*Cyborgeddon*’, y fue en este momento que terminé de decidir que es a lo que quiero dedicarme en el futuro.

A lo largo de mi vida he jugado una considerable cantidad de juegos entre los cuales se puede apreciar una inclinación por los juegos de aventura y exploración, los Rol Play Game (RPG) y los RTS o de estrategia en general, es por esto que me hace especial ilusión desarrollar un juego de uno de estos géneros.

Teniendo en cuenta esto la elección de desarrollar un juego del tipo RTS para la realización del Trabajo Final de Grado (TFG) se debe a que según mi criterio es el que tiene mayor potencial para ayudarme a mejorar mis habilidades como programador, ya que se basa en unas mecánicas con las que no he trabajado con anterioridad.

Una vez dicho esto, los objetivos planteados para el proyecto son los siguientes:

- Aumentar mis conocimientos de C++
- Comprender mejor el funcionamiento del PC
- Aprender nuevas técnicas de Inteligencia Artificial (IA)
- Profundizar en la arquitectura Entity-Component-System (ECS)
- Desarrollar un producto mediante el cual mostrar mis habilidades



# Agradecimientos

*A mi madre Gabriela,  
por el increíble esfuerzo que realiza cada día por mí.*

*A mi hermana Raquel,  
por haber sido y ser un faro para mí.*

*A mi buen compañero Jorge Espinosa,  
por nuestra convivencia estos años.*



*Para saber hablar es preciso saber escuchar*

Plutarco.



# Índice general

<b>1</b>	<b>Introducción</b>	<b>1</b>
<b>2</b>	<b>Marco teórico</b>	<b>3</b>
2.1	Age of Empires II: The Age of Kings . . . . .	3
2.2	The Are Billions . . . . .	6
2.3	Sea Salt . . . . .	9
2.4	Técnicas de inteligencia artificial . . . . .	11
<b>3</b>	<b>Documento de Diseño del Juego (GDD)</b>	<b>13</b>
3.1	Descripción general . . . . .	13
3.2	Mecánicas . . . . .	13
3.3	Unidades . . . . .	15
3.4	Controles . . . . .	16
3.5	Pantallas . . . . .	16
3.6	Estados del juego . . . . .	17
3.7	Escenarios . . . . .	18
3.8	Mínimo producto viable . . . . .	19
<b>4</b>	<b>Metodología</b>	<b>21</b>
4.1	Metodología . . . . .	21
4.2	Estructura de una iteración . . . . .	22
<b>5</b>	<b>Desarrollo y fases del producto</b>	<b>25</b>
5.1	Fase 0: Definición del proyecto y primeros pasos . . . . .	25
5.2	Fase 1: Producto mínimo viable . . . . .	31
	<b>Bibliografía</b>	<b>35</b>



# Índice de figuras

2.1	Carátula del juego original.	4
2.2	Ciudad siendo asediada.	5
2.3	Descripción Estructura.	5
2.4	Ejemplo condición.	5
2.5	Ejemplo condición.	6
2.6	Ejemplo constantes.	6
2.7	Imagen promocional del juego.	6
2.8	Ejemplo de elementos tecnológicos del juego.	7
2.9	Infectado gigante	8
2.10	Ejemplo de captura global del nivel.	8
2.11	Menú principal	9
2.12	Criaturas de ‘Dagon’	10
2.13	Separation behavior	12
2.14	Alignment behavior	12
2.15	Cohesion behavior	12
3.1	MockUp Seguimiento de la marca del jugador.	13
3.2	MockUp formacion en fila.	14
3.3	MockUp formacion desordenada.	14
3.4	MockUp formacion en anillo.	14
3.5	MockUp inicio.	16
3.6	MockUp carga.	16
3.7	MockUp juego.	17
3.8	MockUp pausa.	17
3.9	MockUp victoria.	17
3.10	MockUp derrota.	17
3.11	MockUp flujo de ejecución.	18
3.12	Ejemplo escenario con vegetación.	18
4.1	Logo de Trello	22
4.2	Logo de Toggl	22
4.3	Diagrama de iteración	22
5.1	Movimiento rectilíneo uniforme.	28
5.2	Movimiento rectilíneo acelerado.	28
5.3	Esquema algoritmo de Bresenham.	30

5.4	Esquema acceso a vector de punteros.	32
5.5	Esquema acceso a vector de objetos.	32
5.6	Marcador dirección entidad.	33

# **Índice de tablas**

3.1 Unidades y estadísticas . . . . .	15
5.1 Resumen fase 0 . . . . .	25
5.2 Resumen fase 1 . . . . .	31



# Índice de Listados

5.1.	Resultados iteración bucle . . . . .	26
5.2.	Componente de Movimiento . . . . .	26
5.3.	Entero coma fija . . . . .	27
5.4.	Arrive behavior . . . . .	29



# Índice de Acrónimos

A lo largo del documento serán utilizadas una serie de abreviaturas con el fin de hacer más cómoda su lectura. Todos los términos están indicados a continuación:

- **TFG:** Trabajo Final de Grado
- **PC:** Personal Computer
- **RTS:** Real Time Strategy
- **RPG:** Rol Play Game
- **IA:** Inteligencia Artificial
- **NPC:** Non-Player Character
- **AoE:** Age of Empires
- **TaB:** They are Billions
- **ECS:** Entity-Component-System
- **GDD:** Game Design Document



# 1 Introducción

Este proyecto trata sobre el desarrollo de un simulador de batallas para Personal Computer (PC) desarrollado en *Linux*, escrito en C++ y haciendo uso de librerías escritas en C como TinyPTC [\[add reference\]](#).

El simulador, [\[título\]](#), se trata de una demo del género Real Time Strategy (RTS) compuesta por un escenario a través del cual tendremos que comandar y luchar junto a nuestro ejército con el fin de abatir al ejército rival. El escenario estará compuesto por estructuras y obstáculos que deberemos sortear para alcanzar nuestro objetivo.

A nivel técnico el proyecto cuenta con el desafío de desarrollar una Inteligencia Artificial (IA) grupal para los Non-Player Character (NPC) basada en el uso de técnicas como los *Steering behaviours* y el *Flocking* con el fin de emular un comportamiento coordinado entre las distintas entidades. El objetivo es crear una versión inicial sencilla mediante la cual poder profundizar en los conceptos en los cuales se basan las mencionadas técnicas con el fin de desarrollar una versión más compleja y específica para nuestro proyecto de forma que se adapte de la mejor forma posible a nuestras necesidades.

Por otro lado, como objetivo adicional fuera de la demo como tal es realizar un *port* a *Windows* con el fin de poder mostrar los resultados en ambos sistemas operativos.



## 2 Marco teórico

Cuando uno busca desarrollar un producto es una buena práctica el investigar que se ha realizado previamente en entregas similares, ver que técnicas se han utilizado y que problemas se han encontrado y como los solventaron. También es interesante ver que mecánicas se han ido conservando y que innovaciones se han ido introduciendo con el paso del tiempo.

A continuación vamos a detallar aspectos de interés encontrados en diversas entregas las cuales nos servirán de referentes.

### 2.1. Age of Empires II: The Age of Kings

En primer lugar encontramos el juego '*Age of Empires II: The Age of Kings*', en este juego encontraremos una serie de campañas a través de las cuales encarnaremos a personajes históricos como 'Juana de Arco' o 'William Wallace' y los acompañaremos en sus conquistas y batallas más icónicas. Además de grandes batallas encontraremos el deber de gestionar nuestra facción, este trabajo nos requerirá recoger recursos a lo largo del mapa mientras desarrollamos nuestras ciudades y unidades.

El desarrollo tecnológico se divide en cuatro etapas históricas: Alta Edad Media, de los Castillos, Feudal e Imperial, cada una de estas etapas trae consigo una serie de unidades, edificaciones e investigaciones nuevas las cuales nos proporcionaran escenarios más complejos a nivel estratégico, para pasar de una etapa a otra no es necesario completar al 100 % las opciones desbloqueadas, solo unos requisitos mínimos.

El juego cuenta con un total de 35 civilizaciones o facciones, todas tienen acceso al mismo árbol tecnológico pero cuentan con ligeras diferencias, cada una cuenta con una o dos tecnologías únicas y características especiales en una unidad militar<sup>1</sup>.

Todas la unidades militares<sup>2</sup> de las que podemos disponer cuentan con una serie características que las hacen más o menos fuertes en función del objetivo al que se enfrenten, podemos encontrar el ejemplo de las máquinas de asedio las cuales son más fuertes contra estructuras pero más débiles contra infantería, o las unidades a caballo que son fuertes contra arqueros e infantería pero débiles frente a lanceros.

<sup>1</sup>Normalmente alguna cualidad mejorada en relación a las demás como puede ser la velocidad o daño.

<sup>2</sup>Lista de unidades: [https://ageofempires.fandom.com/wiki/Units\\_\(Age\\_of\\_Empires\\_II\)](https://ageofempires.fandom.com/wiki/Units_(Age_of_Empires_II)).

Este tipo de mécanicas dotan al juego de una importante componente táctica en el manejo de las unidades que debemos dominar si queremos completar los diferentes niveles de forma satisfactoria.

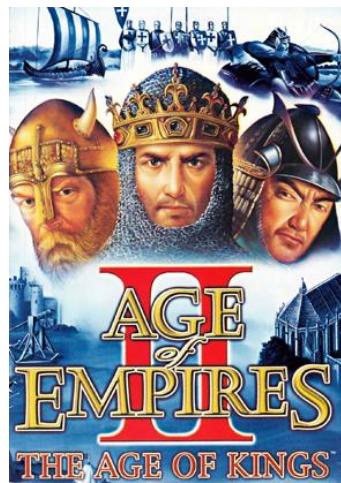


Figura 2.1: Carátula del juego original.

En lo referente a edificaciones<sup>3</sup> podemos encontrar distintos tipos según su aporte al jugador, el edificio principal es el ‘Ayuntamiento’ el cual nos permitirá crear colonos para que trabajen recogiendo recursos y/o ayudando en la construcción de nuevas estructuras, en segundo lugar podemos encontrar edificios como el ‘Cuartel’ o el ‘Campo de Tiro’ los cuales nos permiten crear unidades militares, por otro lado tenemos edificios como la ‘Herrería’ la cual nos permitirá realizar investigaciones y crear unidades de asedio.

Por último podemos encontrar estructuras de temática religiosa como el ‘Monasterio’, otras destinadas a aumentar la productividad en la explotación de recursos naturales como el ‘Molino’ y defensivas como la ‘Muralla’.

En el género *RTS* hay una tendencia al uso de una perspectiva isométrica con cámara fija la cual solo podemos modificar haciendo *zoom* o desplazandola por el entorno, esto seguramente se deba a que es la configuración que mejor nos permite observar el mapa y todas las unidades que hay en el. Además, el hecho de que sea fija nos libra de la problemática de ir ajustando la cámara según la parte del mapeado que estemos, ya que, seguramente el mapeado y los elementos visuales del juego también hayan sido diseñados teniendo esto en cuenta.

Como podemos ver en el artículo de Pritchard sobre el desarrollo de *Age of Empires (AoE) 2*, la IA del juego esta desarrollada completamente con un lenguaje de *scripting* desarrollado por ellos 2.1, de esta forma el código consiste en una serie de constantes y variables definidas desde un inicio y mediante condicionales realizar las acciones y

---

<sup>3</sup>Lista de edificios: [https://ageofempires.fandom.com/wiki/Buildings\\_\(Age\\_of\\_Empires\\_II\)](https://ageofempires.fandom.com/wiki/Buildings_(Age_of_Empires_II)).



Figura 2.2: Ciudad siendo asediada.

cambios pertinentes. Gracias a un usuario de ‘GitHub’ podemos ver en las imágenes 2.1 el código usado en el juego y confirmar como está elaborado.

```
(defrule
  (CONDITION 1)
  (CONDITION 2)
=>
  (ACTION 1)
  (ACTION 2)
)
```

Figura 2.3: Descripción Estructura.

```
(defrule
  (can-train villager)
=>
  (train villager)
)
```

Figura 2.4: Ejemplo condición.

**Fuente:** guía de scripting para IA de AoE 2 de Redmechanic.

Además de estos *scripts* el juego utiliza un sistema de *pathfinding* en dos niveles, el primero de los algoritmos calcula la ruta a seguir por las unidades sin tener en cuenta los elementos triviales como las demás unidades, mientras que la segunda pasada calcula el camino en tramos específicos con el fin de esquivar otras unidades, edificaciones, etc....

En lo referente al movimiento de las unidades, tanto por la forma en la que está programada la IA como por la sensación que dan al jugar, es posible que se trate de un “*Goto*” clásico y sencillo para evitar realizar muchos cálculos.

Además, con el tiempo el juego se ha convertido en un objetivo para “*Speedrunners*” y con una escena competitiva que se basa en ser el más rápido jugando<sup>4</sup> por lo que una reacción instantánea de las unidades es un factor importante para estos jugadores.

---

<sup>4</sup>En los enfrentamientos de alto rango se llegan a medir las acciones por minuto (APM).

```
(defrule
  (taunt-detected any-ally 33)
  (game-time > 300)
=>
  (acknowledge-taunt this-any-ally 33)
  (chat-to-allies-using-id 22157)
  (set-goal train-civ-goal 1)
)
;
```

```
;Goals
(defconst increase-town-size-goal 1); this increases town size
(defconst attack-goal 2)
(defconst strategy-goal 3)
(defconst unit-goal 4)
(defconst train-civ-goal 5);1=train villagers, !=1 no villager
(defconst control-goal 6); 6 = allow to be shot, 7 = shot, als
(defconst anti-cavalry-threat-goal 7)
```

Figura 2.5: Ejemplo condición.

Figura 2.6: Ejemplo constantes.

**Fuente:** repositorio de '*GitHub*' del usuario Andygmb.

## 2.2. The Are Billions

En segundo lugar podemos encontrar el juego '*They are Billions (TaB)*' el cual se localiza en un mundo postapocalíptico infestado de zombis donde quedan muy pocos supervivientes, el juego cuenta con dos modos:

En primer lugar encontramos el modo **horda** en el cual deberemos sobrevivir a sucesivas oleadas de zombis, la cuales serán cada vez más grandes y con unidades más poderosas. Además, deberemos mejorar nuestra colonia y sus defensas para poder lograr pasar las rondas.

En segundo lugar tenemos el modo **campaña**, el cual consistirá en una serie de misiones con el fin de ampliar el tamaño de nuestro imperio a la vez que conocemos su historia y personajes clave.



Figura 2.7: Imágen promocional del juego.

En '*TaB*' podemos ver como se reutilizan una serie de características propias del género como pueden ser: el uso de una perspectiva isométrica con cámara fija o el uso de un “árbol tecnológico” por el cual deberemos escalar si queremos desbloquear nuevas unidades y/o mejorar las ya disponibles.

Si nos fijamos en las construcciones<sup>5</sup> disponibles en el juego, encontramos como se sigue la línea de un “Ayuntamiento” como núcleo de nuestra base, el cual nos permitirá conseguir trabajadores y nos dará acceso a las primeras mejoras. Por otro lado, encontramos las edificaciones destinadas a conseguir recursos, reclutar nuevas tropas y/o defender nuestra colonia.

En este juego se introduce como novedad el recurso de la ‘energía’, el cual será utilizado para mantener nuestras maquinarias en funcionamiento y nos limitará cuanto y donde podremos construir. Para ganar energía y ampliar el terreno disponible el jugador deberá construir ‘torres de Tesla’ (o sus respectivas mejoras).

A diferencia de en ‘AoE’, en ‘TaB’ encontramos la aparición de solamente dos facciones: ‘El Nuevo Imperio’, la facción del jugador, la cual representa una serie de colonias que tendremos que desarrollar a lo largo de los niveles, y en el otro lado, encontramos las infinitas hordas de zombis que tratarán de exterminar a la raza humana.

Cada una de las facciones tiene sus propias unidades 2.8, el jugador contará con 7 unidades distintas entre las que podemos encontrar desde unas rápidas y sigilosas exploradoras hasta unidades equipadas con lanzallamas o montados en robots de combate.<sup>6</sup> Las hordas enemigas estarán compuestas en su mayoría por unidades débiles que de forma eventual serán acompañadas por gigantes 2.9 y/o otras unidades especiales.<sup>7</sup>



Figura 2.8: Ejemplo de elementos tecnológicos del juego.

A lo largo del juego podemos encontrar funciones interesantes e innovadoras como la pausa táctica, la cual nos permitirá visualizar detenidamente el estado del mapa sin tener que preocuparnos por no estar atendiendo algunos posibles eventos, como ataques a nuestras tropas por parte del enemigo.

En juegos anteriores como los ‘AoE’ es fácil encontrarnos en la situación de tener trabajadores en la ciudad sin hacer tareas un rato y, no poder mirar cuales son e ir pensando

<sup>5</sup>Lista de edificios: <https://they-are-billions.fandom.com/wiki/Category:Buildings#Buildings>.

<sup>6</sup>Lista unidades imperiales: <https://they-are-billions.fandom.com/wiki/Category:Units>.

<sup>7</sup>Lista unidades zombi: <https://they-are-billions.fandom.com/wiki/Category:Infected>.



Figura 2.9: Infectado gigante

su ocupación siguiente por estar atrapados en refriegas con otros jugadores, con este tipo de mecánicas estas situaciones se solventan en mayor o menor medida y permiten al jugador tomarse el tiempo que necesite para pensar las acciones que quiere realizar.

Otra adición interesante es la posibilidad de hacer una captura de todo el mapeado explorado 2.10 en el nivel pudiendo así mostrar el avance de nuestra ciudad y alrededores, además de la inmensidad de las oleadas de zombis que nos irán atacando.



Figura 2.10: Ejemplo de captura global del nivel.

En lo refente a la IA, podemos observar como intentan replicar la forma de actuar del zombi común. Como podemos leer en la entrevista a ‘Jesús Arribas’ y ‘Miguel Corral’<sup>8</sup> por parte de Sucasas para el portal ‘Xakata’, cada zombi cuenta con mecanismos para detectar a los humanos, ya sea por divisarlos visualmente o escucharlos moverse, además si un compañero cercano ha encontrado un objetivo, este le seguirá siguiendo una mentalidad de manada a la que se irán sumando los zombis en su recorrido.

Mientras los infectados estén sin un objetivo específico, pueden darse dos escenarios: el primero nos muestra enemigos dispersos por el mapa esperando a que el jugador se

<sup>8</sup>Responsables principales de Numantian Games y desarrolladores del juego.

aventure a explorar esa zona y sorprenderle con un ataque. El otro nos trae las oleadas que aparecen en los límites del mapa las cuales tienen como objetivo el centro de nuestra base, si los enemigos estáticos por el mapa divisan la horda se unirán a ella aumentando así su poder, esto sin duda alienta al jugador a “limpiar” el mapa con el fin de aliviar estas acometidas.

Como la intención de la IA es la de replicar la forma de “no pensar” de los zombis, en el momento en que unidades humanas se presenten delante de las hordas de enemigos estos se dispondrán a perseguir a estas personas, perdiendo así el foco en nuestra base. Esto abre las puertas a poder jugar con el “aggro” de los zombis de forma estratégica.

En paralelo a las declaraciones de los *devs*, parte de la comunidad discutía sobre si la IA del juego hacía trampas con tal de incrementar la dificultad, a lo que el usuario Amordron comenta las declaraciones de los desarrolladores y niega que la máquina haga trampas, para apoyar sus palabras nos redirige a un *post* de ‘Reddit’ donde PikachuNet nos muestra capturas de un posible editor de niveles del juego en el que se nos muestran multitud de opciones, entre las que encontramos la cantidad de enemigos, como se disponen en el mapa, o si tendrán esa iniciativa de ir al centro de la base.

### 2.3. Sea Salt

Como podemos observar en los anteriores referentes, dentro del género RTS da la sensación de que todos los recursos destinados a IA estaban reservados para la toma de decisiones y en gestionar la civilización, dejando el desplazamiento de las unidades y el combate como algo trivial.

Dicho sea que el género nació cuando no había casi capacidad de procesamiento y ,con el paso del tiempo, el hecho de intentar consevar la esencia del género a ocasionado la herencia de experiencias de juego un “poco básicas“.



Figura 2.11: Menú principal

Por otro lado, hay juegos que han aportado experiencias y mecánicas nuevas, ya sea combinando géneros y/o incluyendo técnicas más modernas que nos ofrecen resultados distintos. Este es el caso de ‘*Sea Salt*’, juego que combina estrategia, acción y cartas en el que encarnaremos a ‘Dagon’ una antigua deidad marina la cual planea castigar a la humanidad por haberse opuesto a sus designios.

El juego se compone de una serie de niveles que representan distintas partes de la ciudad donde se desarrolla la historia, cada nivel tiene a su vez distintas salas donde encontraremos diversos obstáculos y enemigos que abatir. Para abrirnos paso por el nivel manejaremos al ejército de ‘Dagon’ el cual se compone de diversas criaturas cada una con sus características únicas, a lo largo de los niveles encontraremos altares donde podremos invocar nuevas unidades para aumentar el grueso de nuestras filas.

En este caso los puntos que más nos interesa tratar es el uso de ‘*Flocking*’ y el control de las unidades que controla el jugador.



Figura 2.12: Criaturas de ‘Dagon’

En cuanto a la jugabilidad y los controles, el jugador no manejará de forma explícita las criaturas del ejército, sino que, mediante las teclas ‘w’, ‘a’, ‘s’, ‘d’ moverá una marca por el mapa la cual será seguida por las unidades. Además, con la ‘barra espaciadora’ las unidades se lanzarán al ataque de los enemigos en su rango de acción, aparentemente al más cercano. Por último encontramos la tecla ‘ctrl’ la cual nos permitirá hacer que las unidades se reagrupen en la marca del jugador y el ‘shift’ el cual hará que todas las unidades avancen a la misma velocidad.

Mediante estas acciones tendremos que arreglarnos las para matar a las unidades enemigas y esquivar sus ataques, sobretodo en las peleas con jefes al final de los niveles.

Por otro lado encontramos el uso de ‘*Flocking*’ el cual aporta un toque “errático” en el desplazamiento de las unidades que, junto a los controles, juega mucho con la temática de monstruos/criaturas malvadas.

La técnica ayuda a darle vida y sensación de individualidad a cada una de las unidades, a la vez que refuerza la visión de conjunto al ver como las unidades se encuentran en todo momento actuando de la misma forma.

## 2.4. Técnicas de inteligencia artificial

Como ha sido mencionado en la introducción 1, uno de los objetivos más importantes del proyecto recaen en el desarrollo de una IA haciendo uso de las técnicas de '*Flocking*' y '*Steering behaviors*'.

Para ello nos basaremos principalmente en las explicaciones y ejemplos que podemos encontrar en el libro (Millington, 2009, ch. 3) donde de forma extensa y detallada se nos introduce en la teoría relacionada a los algoritmos y la forma en la que estos se estructuran e interactúan entre ellos. Para dar un poco de contexto sobre el tema resumiremos brevemente las ideas que se nos presentan a lo largo del capítulo dedicado a estas técnicas.

Los '*Steering behaviors*' pueden ser entendidos como una serie de algoritmos destinados a guiar la forma en la cual los NPC se desplazan por el escenario y/o interactúan con los distintos elementos que puedan encontrarse en la escena. Siguen una filosofía de crear movimientos complejos a base de una combinación de movimientos y/o acciones simples, un ejemplo común puede ser la acción de perseguir a un objetivo mientras se sorteán obstáculos en el proceso.

En este caso no tendríamos una función llamada "*persigue-enemigo-mientras-esquivas()*" y esta encargarse de todo el trabajo, sino que, tendremos el cálculo de la velocidad y dirección necesarias para alcanzar el objetivo, la compropación para saber si hay algún tipo de obstáculo por el camino y la rectificación de la trayectoria en caso de haberlos cada uno por su lado y es la resultante de todos los pasos la que defina el movimiento final.

Esta forma de estructurar y formar actividades complejas en base a acciones más simples nos permite reutilizar y jugar con los diferentes comportamientos permitiéndonos crear con ellos un amplio espectro de resultados.

En lo referente al '*Flocking*' podemos observar como en esencia es lo mismo que los '*Steering behaviors*' pero añadiendo factores y/o componentes grupales, el origen del modelo lo podemos encontrar en las publicaciones de Reynolds (1986) donde se nos introduce el concepto de "*Boid*" como entidad genérica que simula su comportamiento bajo este algoritmo. Además, se nos introducen los tres comportamientos básicos en los que se basa la técnica para generar el movimiento emergente, que son:

La **separación** 2.13 que cada *Boid* mantendrá entre las demás entidades en su vecindad, con esto evitaremos solapamientos y respetar el espacio y movimiento de las demás entidades.

Por otro lado podemos encontrar el **alineamiento** 2.14 de la dirección del movimiento propio con las de las entidades más cercanas, de esta forma conseguimos un movimiento armónico entre los *boids* y produciremos una sensación de coordinación entre ellos.

Por último encontramos la **cohesión** 2.15 la cual se encargará de mantener a las enti-

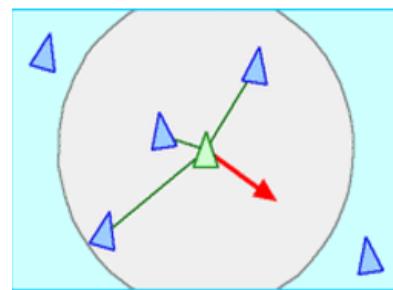


Figura 2.13: Separation behavior

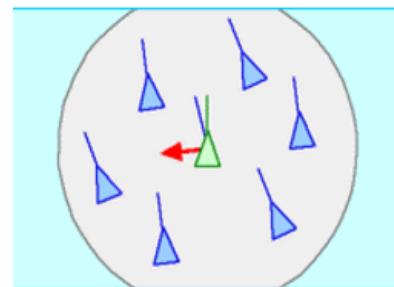


Figura 2.14: Alignment behavior

dades cercanas juntas para crear esa sensación de grupo que buscamos con el algoritmo.

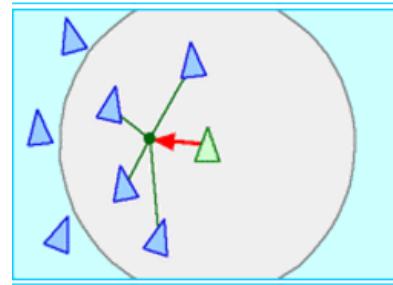


Figura 2.15: Cohesion behavior

Por otro lado, podemos ver en el artículo de ‘Raynolds’ como a lo largo de los años se ha ido modificando y ampliando el algoritmo con el fin de añadir variaciones en el comportamiento y/o introducir más factores influyentes en la decisión de los *Boids* como puede ser el olor de determinada entidad/es y/o escenario.

Esto sin duda es gracias a la versatilidad que nos proporciona el uso de los ‘*Steering behaviors*’ y jugar con la importancia de las distintas componentes a la hora de hacer la toma de decisiones.

# 3 Documento de Diseño del Juego (GDD)

## 3.1. Descripción general

El juego que se pretende desarrollar se basa en el apartado de navegación por el mapa y combate típicos en los RTS. A lo largo de los distintos niveles, el jugador deberá hacer frente a distintos desafíos como pueden ser: escoltar con sus unidades de un punto del mapa a otro a un personaje importante y/o eliminar a todas las entidades enemigas del escenario.

La ficha técnica es la siguientes:

- **Plataforma:** PC (Windows y Linux).
- **Género:** Real Time Strategy (RTS).
- **Idioma:** Inglés.

## 3.2. Mecánicas

Durante el juego podremos ejecutar una serie de órdenes sobre nuestras unidades para cambiar su distribución, ordenarles que se desplazan y/o que ataquen a enemigos. Para el desplazamiento por el mapa, el jugador contará con un marcador en el mapa el cual será capaz de desplazar para que sus unidades lo sigan mientras les sea posible.

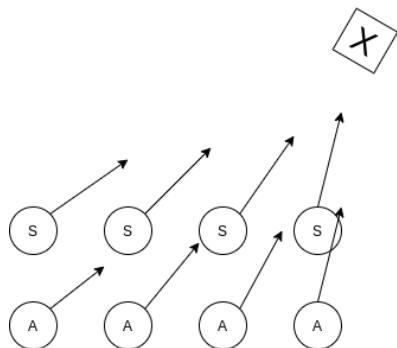


Figura 3.1: MockUp Seguimiento de la marca del jugador.

En cuanto a las posibles formaciones para nuestras unidades, encontramos las siguientes:

- **Por filas:** las unidades se dispondrán en función de su rango, siendo encabezado el pelotón por las unidades de ataque a *melee*.

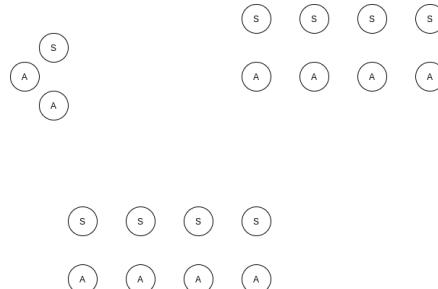


Figura 3.2: MockUp formacion en fila.

- **Sin formación:** las unidades no tendrán ningún parámetro de ordenación.

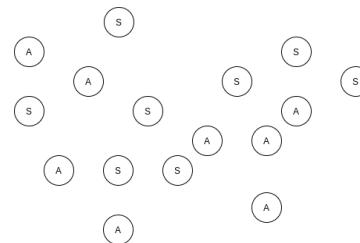


Figura 3.3: MockUp formacion desordenada.

- **En anillo:** formación circular alrededor de una unidad especial. En caso de no haber unidad escoltada el centro estará vacío.

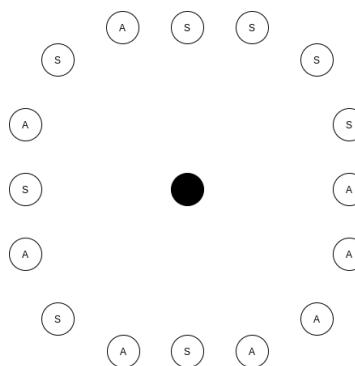


Figura 3.4: MockUp formacion en anillo.

Por último podemos ordenar a las unidades que mantengan una estrategia definida:

- **Mantener la zona:** en este modo las unidades intentarán evitar dejar su posición actual, pudiendo desplazarse ligeramente para atacar unidades enemigas. En caso de salir de combate regresará a su posición.
- **Agresivo:** en el momento en el que la tropa divisa un enemigo saldrá directo a atacarle, aunque se aleje esta le seguirá mientras siga en su rango de detección.

Aunque el jugador marque una estrategia y forma de plantear el combate, las unidades podrán decidir si seguir al jugador o revelarse. Si los soldados cercanos están siendo masacrados existe la posibilidad de que tengan miedo y huyan, si las unidades enemigas nos superan en número la unidad puede decidir si permanecer en defensa en lugar de atacar de forma activa.

### 3.3. Unidades

Entre las unidades podemos encontrar dos arquetipos con características propias que nos permitirán crear variedad en las soluciones a la hora de superar el nivel.

Los tipos son los siguientes:

- **Soldado:** es la unidad más básica que podemos encontrar en el campo de guerra, está armado con una espada y posee estadísticas bajas.
- **Arquero:** va equipado con arco y flechas para atacar a distancia a sus rivales, tiene menos resistencia que los soldados por lo que tendremos que protegerlos para asegurar su supervivencia.

	Daño	Vida	Rango
Soldado	1	5	Melee
Arquero	2	3	Largo

Tabla 3.1: Unidades y estadísticas

Valores totalmente provisionales

### 3.4. Controles

A la hora de jugar tendremos una serie de teclas asignadas a las acciones que el jugador puede realizar cuando interactúe con el juego. Para enumerarlas dividiremos las acciones en dos grupos, dependiendo de si son para navegar por los menús o si representan acciones durante el *gameplay*.

Las teclas para navegar por los menús son las siguientes:

- **Enter:** mediante esta tecla podremos avanzar por los menús una vez estemos sobre la opción deseada.
- **Retroceso:** mediante esta otra podremos ir hacia atrás por los menús.
- **Escape:** esta nos permitirá salir de la ejecución del programa. igual esto es un poco ilegal.

Las asignadas para jugar son las siguientes:

- **W/A/S/D:** mediante estas teclas podremos desplazar el puntero por el mapa.
- **Barra Espaciadora:** con esta otra podremos ordenar atacar a nuestras unidades, si no encuentran una unidades enemiga en su rango se mantendrá en seguimiento del puntero, si un aliado cercano entra en combate se unirá a la afrenta junto a él.
- **B:** activa el modo huída y las unidades dejarán de combatir y seguirán el puntero aunque tengan enemigos en su campo de visión.

### 3.5. Pantallas

Al ejecutar el programa la primera pantalla que aparecerá será la de inicio. Se compone del título del juego, la fecha de lanzamiento, el nombre del desarrollador y un mensaje que nos indica que tenemos que pulsar al tecla *entre* para continuar, esta pantalla se mantendrá hasta que el jugador presione dicha tecla.

Una vez pulsado el botón indicado se procederá a cargar el juego mientras se muestra una barra que indica el progreso de cargado.

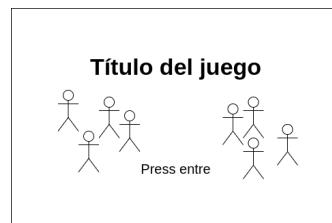


Figura 3.5: MockUp inicio.

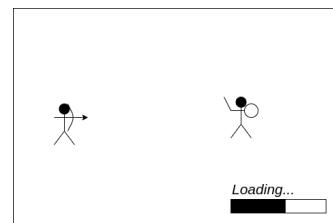


Figura 3.6: MockUp carga.

A continuación de la carga pasaremos directamente al escenario, donde se desarrollará el *gameplay* y nos mantendremos en esta pantalla hasta que termine el juego, ya sea por victoria o derrota del jugador 3.7.

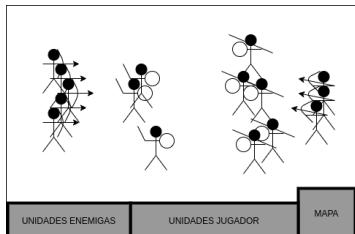


Figura 3.7: MockUp juego.

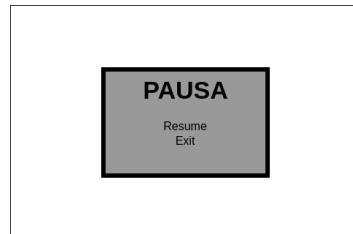


Figura 3.8: MockUp pausa.

El final de la partida nos trae dos posibles escenarios, la victoria y la derrota. Para cada uno saldrá su respectivo mensaje 3.9 3.10 y pasado un momento se nos mandará automáticamente a la pantalla de inicio.

Como última pantalla con la que el jugador podrá interactuar es la del menú de pausa 3.8, en el cual se le dará la opción de salir o de volver a la partida, mientras esta pantalla este activa la acción en el juego se paralizará hasta que el jugador decida.



Figura 3.9: MockUp victoria.

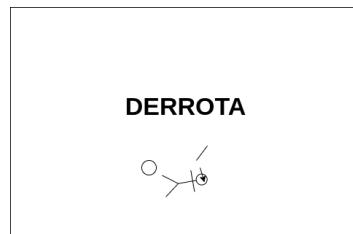


Figura 3.10: MockUp derrota.

### 3.6. Estados del juego

Una vez mostradas todas las posibles pantallas con las que podrá interactuar el jugador, es interesante dibujar un diagrama de flujo que plasme sus conexiones y posibilidades con el fin de crear una representación gráfica que sirva como esquema global.

Dicho esquema podemos encontrarlo en la figura 3.11.

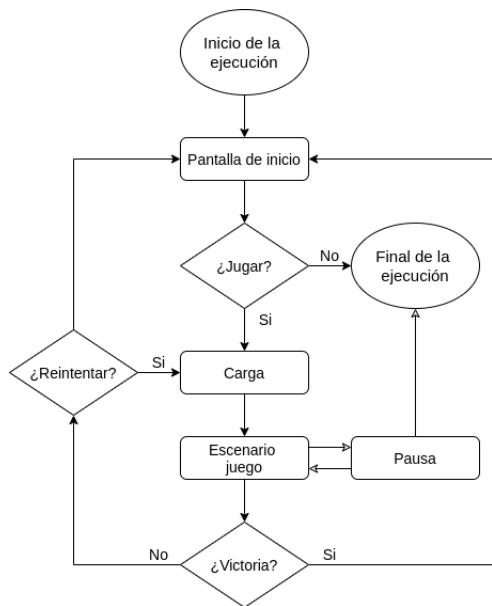


Figura 3.11: MockUp flujo de ejecución.

### 3.7. Escenarios

A lo largo de los niveles la intención es que nos encontremos con distintos biomas como pueden ser praderas, bosque o desiertos como los que podemos encontrar en AoE II.



Figura 3.12: Ejemplo escenario con vegetación.

Además de imitar el estilo artístico el juego se desarrollará empleando una cámara aerea fija manteniendo una perspectiva isométrica que nos permitirá visualizar desde un punto elevado una gran porción del escenario, esto lo haremos con el fin de dar al jugador la posibilidad de conocer lo máximo posible el territorio cercano posible y a la

vez le libramos de tener que estar preocuparse de situar la cámara para cada momento. Al mantener una posición fija podemos situar todos los elementos en la escena de forma que no obstaculicen la visión o molesten al jugador.

### 3.8. Mínimo producto viable

Una vez planteadas todas las características deseadas para el videojuego, es acosenjable decidir que partes del producto son vitales para mostrar la experiencia de juego deseable, el hecho de marcar estos objetivos nos ayudará a poder desarrollar una versión reducida del proyecto que nos permita dar la sensación de tener un producto acabado.

Es importante realizar este proceso ya que por motivos internos o externos al equipo de desarrollo, siempre existen contratiempos que nos impidan conseguir todos los objetivos propuestos en el documento.

En cuanto a las etapas en la ejecución del juego, encontramos esencial el poder finalizar el nivel, ya sea por victoria o derrota. Acto seguido volver al comienzo del nivel, dejando a un lado un posible menú inicial y de pausa.

Una vez en partida el juego constará de un nivel único compuesto por: un escenario estático mínimo, un conjunto de unidades propiedad del jugador y otro perteneciente a la máquina.

Al inicio del juego el jugador se encontrará quieto en uno de los extremos del nivel y el enemigo estará patrullando por una zona designada, dejando así a elección del jugador la distribución y el momento exacto en el que comenzará la disputa.

En lo referente a las mecánicas y jugabilidad, es importante que el jugador sea capaz de poder mover a sus unidades por el escenario y ordenarles atacar, por otro lado sería deseable poder ajustar algunos parámetros del comportamiento de su ejercito para poder pasar el nivel de más de una forma.

Otro lado la IA debe ser capaz de detectar el acercamiento del jugador y responder a la ofensiva, dejando la capacidad de tomar la iniciativa y variar su estrategia para versiones más completas. Ambos ejercitos se compondrán de soldados y arqueros.

En lo referente al apartado visual, sería deseable tener sprites para unidades y escenario pero de ser necesario podemos mantener el sistema de *render* inicial.



# 4 Metodología

## 4.1. Metodología

La metodología llevada a cabo durante el proyecto es una creación propia basada en características copiadas de metodologías ágiles como el '**Scrum**' y adaptadas a la forma de trabajar propia. La idea es estructurar todo el proyecto en diversas iteraciones y durante estas marcarse objetivos y/o tareas con el fin de añadir funcionalidades completas.

El uso de **iteraciones** para dividir la carga de trabajo y agrupar tareas nos ayudará a conseguir acotar la duración de algunas tareas y marcarnos objetivos a corto/medio plazo. Además de separar por funciones el proyecto, las tareas que conllevan su desarrollo también deberemos analizarlas e intentar reducir su tamaño y carga lo máximo posible, esto permite obtener una sensación de éxito de forma rápida lo cual ayuda a mantener una moral y motivación altas.

Para poner un ejemplo de como elaborar esta separación por subtareas, el objetivo que vamos a usar es “dibujar elementos por pantalla”. Como estamos en un proyecto basado en Entity-Component-System (ECS) rápidamente vemos que vamos a requerir mínimo un componente de dibujado (*RenderCmp*) y un “*RenderSystem*” el cual se encargará de recoger todos los elementos visuales y hacer las llamadas necesarias para dibujarlos. En caso de no tener una librería de dibujado, otra tarea podría ser el buscar una que sea capaz de realizar el trabajo requerido y/o implementar una propia <sup>1</sup>.

Para ayudar a la planificación del trabajo usaremos la herramienta ‘Trello’ <sup>2</sup> la cual nos permitirá crear diversas listas según el ámbito o el estado de desarrollo de las distintas tareas que contendrán, cada tarea se ve representada con una tarjeta la cual puede tener una serie de subtareas asociadas dentro. Esto nos permitirá tener un registro de todas las labores que quedan por hacer en cada iteración y cuales ya están terminadas completamente, además podremos conservar las listas de las tareas realizadas en iteraciones anteriores para futuras revisiones.

Otra herramienta de control que usaremos durante el desarrollo será ‘Toggl’ <sup>3</sup> la cual nos permitirá llevar a cabo un registro de las horas que dedicamos en cada tarea y agrupar tareas en función del campo de estudio o parte del desarrollador al que pertenecen.

---

<sup>1</sup>Esto supondría una serie de tareas en base a la cantidad de funcionalidades que se requieran.

<sup>2</sup>Página de ‘Trello’: <https://trello.com/es>

<sup>3</sup>Página de ‘Toggl’: <https://toggl.com>



Figura 4.1: Logo de Trello



Figura 4.2: Logo de Toggl

## 4.2. Estructura de una iteración

Con el fin de trabajar de la mejor forma posible, a lo largo de cada iteración encontramos una serie de fases/tareas organizativas que guian el flujo de trabajo.

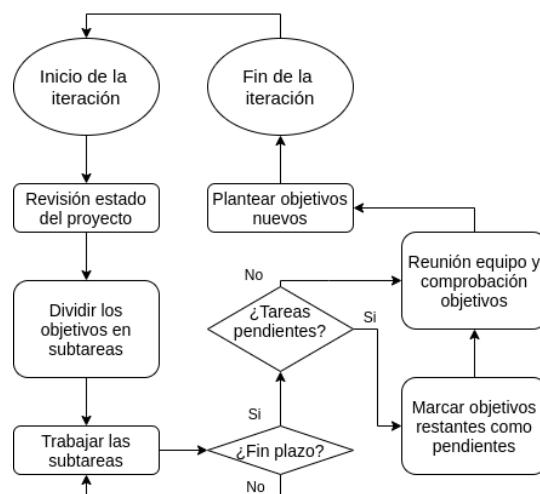


Figura 4.3: Diagrama de iteración

El inicio de la iteración trae consigo un análisis y revisado del proyecto, que lo último implementado esté en un estado aceptable y todo preparado para recibir las futuras adiciones al proyecto si es que las va a haber.

Una vez esté todo correcto procedemos al estudio y división de las futuras tareas, a la vez que las categorizamos según la temática o aspecto del proyecto que se trabaje. Con una visión global de todo el trabajo podemos organizar como vamos a abordar el desarrollo, analizar si tenemos dependencias entre tareas y acto seguido empezar. Por lo general suelen surgir inconvenientes o nuevas necesidades durante las iteraciones las cuales nos harán rehacer, posponer o simplemente añadir algunas tareas, por lo que es posible tener que ampliar de forma puntual este apartado a lo largo de la iteración.

Cuando el apartado organizativo esté terminado (sin tener en cuenta contratiempos) lo único que queda por hacer es entrar de lleno en el desarrollo, esta fase se extenderá hasta que se termine el plazo de la iteración o se acaben los objetivos propuestos.

Conforme se acerca el final de la iteración, lo último es una reunión de control donde se comprueba el porcentaje del trabajo propuesto que ha sido completado, posibles problemas encontrados durante la iteración y que soluciones se han aportado (en el caso de haber encontrado una). Una vez la discusión termina se procede a marcar los nuevos objetivos , terminando así la actual iteración y comenzando la siguiente en caso de no finalizar todavía el desarrollo del proyecto.



# 5 Desarrollo y fases del producto

A lo largo de esta sección comentaremos todo el proceso de desarrollo del prototipo y de la memoria, las diferentes iteraciones del producto serán agrupadas en diversas fases con el fin de facilitar la lectura y el entendimiento de los objetivos planteados en cada momento. Además, se contarán de forma más extensa los puntos relevantes y objetivos en este proyecto.

## 5.1. Fase 0: Definición del proyecto y primeros pasos

Iteraciones	De la 0 a la 2	Completado	Terminado en
Objetivos	<b>0.1.</b> Definir la idea.	60 %	It.4
	<b>0.2.</b> Iniciar desarrollo del prototipo Integrar TinyPTC y dibujar por pantalla.	100 %	x
	<b>0.3.</b> Adquirir hábito de escribir, preparar las herramientas y entorno de LaTex.	50 %	It.2
	<b>0.4.</b> Prototipado del movimiento usando <i>Steering behavior</i> .	30 %	It.1
	<b>1.1.</b> Creación tipo con coma fija.	100 %	x
	<b>1.2.</b> Busqueda de referentes	40 %	It.3
	<b>1.3.</b> Corregir funcionamiento de los <i>Steering behavior</i> y prototipar todos los posibles	100 %	x
	<b>1.4.</b> Sistema de debug visual: vectores.	100 %	x
	<b>1.5.</b> Herramientas control de ejecución.	50 %	$\infty$
	<b>2.1.</b> Trabajar más la memoria.	65 %	$\infty$
	<b>2.2.</b> Terminar de definir idea (GDD).	100 %	x
	<b>2.3.</b> Implementar el mínimo producto.	20 %	It.4
	<b>2.4.</b> Experimentación y análisis de resultados.	0 %	$\infty$

Tabla 5.1: Resumen fase 0

Esta primera fase engloba las tres primeras iteraciones donde comenzamos a terminar de definir la idea para el proyecto, ya que, todavía era un poco difusa la imagen del producto final que se quería desarrollar.

Para ir entrando en una dinámica productiva iniciamos el desarrollo de funcionalidades básicas como pueden ser el bucle principal del juego, en el cual usamos de la librería `<chrono>` de C++ para medir y limitar el tiempo entre ejecuciones del bucle acompañado del uso de una enumeración que contiene los posibles resultados de un ciclo del juego.

Listado 5.1: Resultados iteración bucle

```

1 enum class GameConditions : int16_t {
2     Loop      = 0u, //continuar con el bucle
3     Cerrar    = 1u, //terminar la ejecución del programa
4     Derrota   = 2u, //el jugador a perdido
5     Victoria = 3u //el jugador a ganado
6 };

```

Una vez tenemos el bucle, toca introducir algo que hacer en él, como primer sistema del juego encontramos el sistema de dibujado el cual se encargará de crear y manejar la ventana, además de pintar nuestras entidades y herramientas visuales. Para incluir todas estas funcionalidades sin añadir librerías pesadas incluiremos `<TinyPTC>`, la cual no incluye demasiadas funcionalidades u opciones pero nos permitirá trabajar rápido y de forma sencilla. Los sprites en un inicio serán simples cuadrados de colores por lo que no requeriremos mucho.

Por último, se añadieron las componentes de movimiento y IA que, junto a sus correspondientes sistemas, nos permitirán poder crear un comportamiento de patrulla básico entre puntos fijos. Además, se terminó la preparación de las herramientas de L<sup>A</sup>T<sub>E</sub>Xy ‘Jabref’ para redactar la memoria acompañado de la lectura de las directrices, normas y memorias de compañeros del año pasado.

Listado 5.2: Componente de Movimiento

```

1 struct MovementComponent : BECS::Component_t {
2
3     explicit MovementComponent(const BECS::entID entityID, const fint_t
4         <int64_t> c_X, const fint_t<int64_t> c_Y)
5         : Component_t(entityID), coords(c_X, c_Y) {}
6
7     //posicion actual.
8     fvec2<fint_t<int64_t>> coords           { { 01 }, { 01 } };
9     //desplazamiento final de la entidad.
10    fvec2<fint_t<int64_t>> dir              { { 01 }, { 01 } };
11    //acceleracion hacia la posición deseada.
12    fvec2<fint_t<int64_t>> accel_to_target { { 01 }, { 01 } };

```

```

12 // si hay entidades cercanas, fuerza que las agrupa.
13 fvec2<fint_t<int64_t>> cohesion_force { { 01 }, { 01 } };
14 // si hay entidades cercanas, fuerza que las separa.
15 fvec2<fint_t<int64_t>> separation_force { { 01 }, { 01 } };
16
17 //aux debug mode
18 fvec2<fint_t<int64_t>> sep_copy_to_draw { { 01 }, { 01 } };
19 fvec2<fint_t<int64_t>> coh_copy_to_draw { { 01 }, { 01 } };
20 };

```

A lo largo de la iteración siguiente, seguimos añadiendo funcionalidades al prototipo como puede ser el componente y sistema de *Input*, el cual nos permitirá interactuar con el juego mediante el mapeado del teclado haciendo uso de la librería  $\langle X11 \rangle$  y  $\langle TinyPTC \rangle$

Una de las herramientas implementadas en el proyecto es el tipo de dato entero con coma fija, en C++ podemos encontrar el tipo fundamental *float* creado para trabajar con números reales, pero si miramos como funcionan exactamente es posible encontrar que no es lo que deseamos exactamente. Los *floats* usan “precisión simple” lo cual implica que conforme el número sea más grande, la cantidad de bytes destinados a la representación de la parte decimal va disminuyendo ofreciendo así menor precisión. Además, trabajar con ellos es más costoso a nivel computacional<sup>1</sup> y llevan un código más pesado debido a la codificación usada al ser compilados.

El tipo con coma fija tiene dos partes, en primer lugar el valor que se quiere representar normal y corriente, y por otro la escala, número que indicará la cantidad de valores disponibles para la representación decimal. La escala la usaremos para multiplicar o dividir nuestro número para convertirlo a coma fija, devolverlo sin escalar y/o operaciones con otros números enteros.

Es importante elegir una escala adecuada para trabajar de forma eficiente, para ello escogeremos un número potencia de 2 el cual nos permitirá usar desplazamientos en multiplicaciones y divisiones, haciendo los cálculos mucho más rápidos que si tuvieramos que usar las operaciones comunes.

Listado 5.3: Entero coma fija

```

1 template <typename NumType>
2 struct fint_t {
3     fint_t() = default;
4
5     constexpr fint_t(NumType num)           noexcept; // ctor
6     constexpr explicit fint_t(float num)    noexcept; // ctor
7     constexpr fint_t(const fint_t<NumType>& num) noexcept = default;
8     constexpr fint_t(fint_t<NumType>&& num)  noexcept = default;
9

```

---

<sup>1</sup>Esto pasa sobretodo en máquinas que no tengan CPUs con unidades destinadas a cálculos en coma flotante

```

10  /* OPERATIONS */
11 // ***
12
13 /* GETTERS */
14 constexpr NumType getNoScaled() const noexcept;
15
16 /* DATA */
17     NumType number { 0 };
18 constexpr static inline NumType SCALE { 65536 };
19 };

```

Por otro lado, se comenzó a trabajar en la IA del juego creando los primeros comportamientos y herramientas para alternar entre ellos. Aquí caeremos en los primeros errores de concepto a la hora de trabajar con los '*Steering behaviors*', ya que, a la hora de implementar los usamos un plateamiento y cálculos de movimiento cinético sin aceleración, trabajando directamente con la velocidad.

En su lugar, lo planteado por la técnica es usar la aceleración de la entidad, de esta forma se logrará calcular el movimiento deseado en un momento puntual, enfrentando lo contra la velocidad acumulada. Esto nos proporcionará un movimiento continuado y dinámico en lugar de cambiar drásticamente de dirección y velocidad.



Figura 5.1: Movimiento rectilíneo uniforme.



Figura 5.2: Movimiento rectilíneo acelerado.

Uno de los comportamientos más sencillos es el '*Arrive*', el cual nos llevará a la posición objetivo de forma directa y conforme estemos llegando a esta, la aceleración disminuirá parando una vez hayamos llegado al destino. En esta función podremos apreciar como calculamos la velocidad objetivo en función de la distancia hasta llegar a la posición objetivo, una vez se calcula se compara con el desplazamiento actual de la entidad y es la resta de ambas la que nos dará la aceleración necesaria para alcanzar la velocidad objetivo, siempre controlando que la aceleración y la velocidad no exceden los límites marcados.

Además, contamos con dos modificadores adicionales como son el tiempo deseado para

alcanzar el objetivo y un límite de distancia con el objetivo para evitar solapes y/o oscilar sobre él.

Listado 5.4: Arrive behavior

```

1  template <typename Context_t>
2   constexpr bool
3   AI_System<Context_t>::arrive(Context_t& context, BECS::entID eid)
4     noexcept {
5       /* Recogida de datos */
6       //Las variables en mayúscula son constantes definidas fuera
7       //los datos terminados en '2' usan el valor al cuadrado
8
9       if(distance2 < ENT_ARRIVE_DIST2)
10      return false;
11
12      if(distance2 > ENT_SLOW_DIST2)
13        target_speed = ENT_MAX_SPEED;
14      else
15        target_speed = ENT_MAX_SPEED *
16          ( target_dir.length_fix() / ENT_SLOW_DIST );
17
18      target_dir.normalize();
19      target_dir *= target_speed;
20
21      my_accel = (target_dir - my_direct);
22      my_accel /= ENT_TIME_TO_TARGET;
23
24      if(my_accel.length2() > ENT_MAX_ACCEL2) {
25        my_accel.normalize();
26        my_accel *= ENT_MAX_ACCEL;
27      }
28
29      return true;
}

```

Durante la tercera iteración nos centramos en la mejora de algunos '*Steering behaviors*' para terminar de hacerlos más fáciles de usar y combinar, con el fin de tener un código más limpio y legible, también se mejoró el uso de la coma fija añadiendo más operaciones y ajustando el funcionamiento.

Por otro lado, se añadieron herramientas para debug del movimientos de las entidades y las diferentes componentes de este, para el dibujado de los vectores se eligió el algoritmo de 'Bresenham', ya que, nos permite dibujar rectas de una forma efectiva y sin consumir muchos recursos. Además, el algoritmo esta diseñado para trabajar con enteros, cosa que hacemos a lo largo de todo el prototipo. En el esquema 5.3 la cuadricula blanca representa los pixeles de la pantalla, en color azul la recta ideal que se busca dibujar y en amarillo la recta resultante del algoritmo.

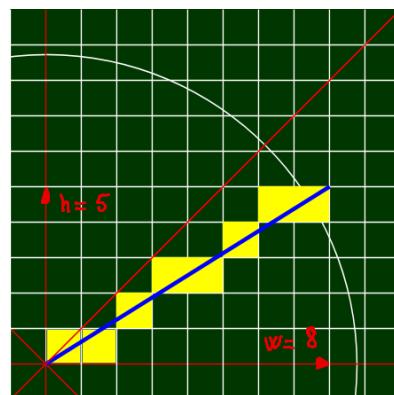
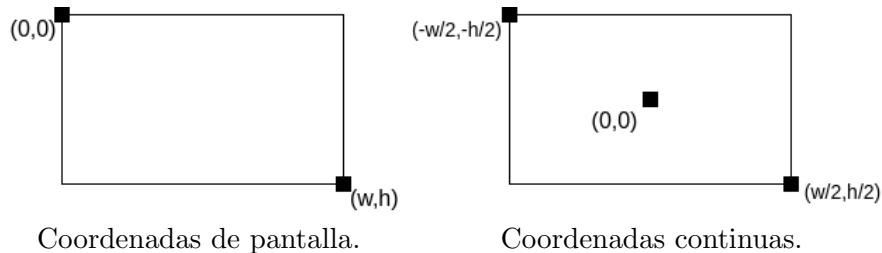


Figura 5.3: Esquema algoritmo de Bresenham.

Otra de las herramientas destinadas a poder analizar el funcionamiento del programa es el control del tiempo de ejecución y el tamaño del ‘*DeltaTime*’. Al poder disminuir el número de fotogramas por segundo, podemos visualizar más detenidamente que sucede en pantalla y ver que todo va según la previsión, por otra parte se puede aumentar la velocidad para llegar antes a un punto de la ejecución en concreto. Además, jugar con el tamaño del ‘*DeltaTime*’ nos permite experimentar con los valores de velocidad y/o otros factores interpolados para ajustarlos de forma visual.

Por último en cuanto a implementación, toda la información relacionada con el sistema de dibujado y los elementos visuales del juego se trabajan con *uint64\_t*, mientras que todo lo relacionado con el movimiento y demás cálculos físicos se trabajan con *float\_t < int64\_t >*. Por ello es necesario usar “*casts*” sobre los datos físicos con los que se quiera operar en el *Render System*<sup>2</sup>, como es una operación recurrente en el código se implementó un sistema auxiliar para convertir coordenadas continuas (usadas en la mayoría de sistemas) a coordenadas de pantalla o pixel.



En lo referente a la memoria comenzamos la redacción de una versión preliminar del Game Design Document (GDD) donde comenzamos a describir las características del producto, del Estado del Arte donde se hace mención a juegos que nos han servido de inspiración o modelo para imitar y se incluye una explicación sobre las técnicas y algoritmos se van a usar, y la explicación sobre la metodología seguida durante este desarrollo.

<sup>2</sup>Ya sea la posición en el mapa como el sistema de debug para los vectores de *Steer*.

## 5.2. Fase 1: Producto mínimo viable

Iteraciones	De la 3 a la 5	% Completado	Terminado en
Objetivos	<b>3.1.</b> Cambios en el motor ECS.	100 %	x
	<b>3.2.</b> Cambios en el almacenamiento de datos.	100 %	x
	<b>3.3.</b> Cambio tipos propios a template.	100 %	x
	<b>3.4.</b> Cambio en el sistema de clases.	100 %	x
	<b>4.1.</b> Añadir referente que use <i>Steering Behavior</i> .	100 %	x
	<b>4.2.</b> Diagrama y explicación de las fases de una iteración.	100 %	x
	<b>4.3.</b> Agrupar iteración en fases y describir el desarrollo hasta la fecha.	30 %	
	<b>4.4.</b> Implementar mínimo producto.	85 %	
	<b>4.5.</b> Análisis de resultados del MVP.	10 %	
	<b>5.1.</b> Crear unidad de ataque a distancia.	%	
	<b>5.2.</b> Adición de formación en anillo.	%	
	<b>5.3.</b> Balanceo de las unidades.	%	
	<b>5.4.</b> Profundizar los objetivos en el GDD y explicar método de balanceo.	%	
	<b>5.5.</b> Incluir tabla resumen al inicio de fase e incluir información extra como el set-up.	100 %	x

Tabla 5.2: Resumen fase 1

La fase comenzó con una iteración corta debido a las vacaciones de Navidad y final de año, en la que el foco estuvo depositado en refactorizar y dejar en mejor estado el código desarrollado hasta el momento, donde se separó el nucleo principal del motor ECS del conjunto de sistemas y herramientas en dos *namespaces* diferentes.

Por el lado del motor ECS, en un primer lugar el almacén de componentes estaba diseñado para contener punteros a componentes, esto implicaba que los datos de los componentes se guardaban en la memoria libre de forma “anárquica” 5.4, en su lugar, lo deseado es almacenar todas las componentes de forma contigua para poder utilizar la caché cuando sea posible, para poder aprovechar que es más rápida y minimizar la cantidad de veces que mandamos al sistema ir a buscar nuestros datos 5.5.

A continuación, se modificó la estructura de las entidades para componerse únicamente

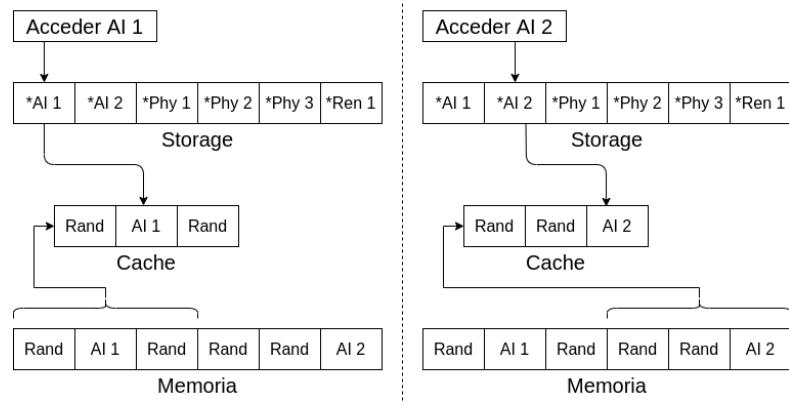


Figura 5.4: Esquema acceso a vector de punteros.

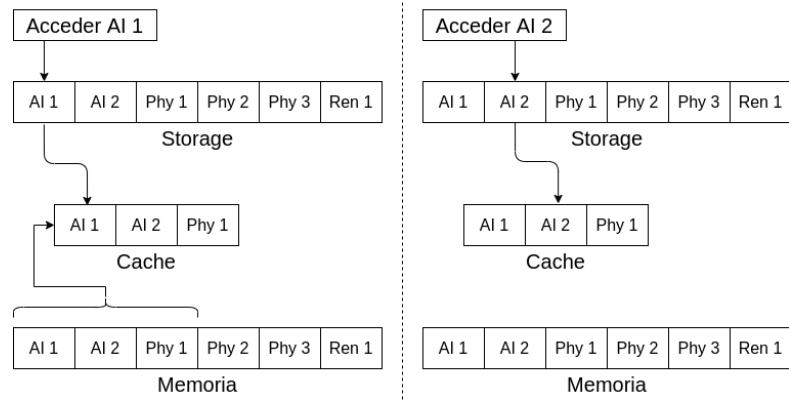


Figura 5.5: Esquema acceso a vector de objetos.

de su *Entity ID* y una serie de *Component IDs* para posibles comprobaciones de que componentes contiene la entidad, dejando así de tener un puntero a las entidades del almacén. Esto implica una serie de cambios en la fachada y uso del *Entity manager* a la hora de crear/eliminar entidades y la obtención de las componentes desde los sistemas a través del contexto.

Por último se siguieron modificando los tipos propios, tanto el  *fint\_t* como los *vec2* y *fvec2* para trabajar como plantillas, pudiendo así librarnos de la necesidad de crear un tipo completo para cada tipo básico con el que queramos usarlos.

Durante el mes de enero se implementaron diversas funcionalidades, en primer lugar abordamos el sistema de combate del juego, para ello se creó una componente para almacenar elementos como la vida de la unidad, su daño, el rango y el tiempo entre ataques, además, los sistemas de ataque para ir restando vida a las unidades conforme reciban daño, el de *cooldown* para actualizar los temporizadores para volver a atacar, y por último el sistema de muertes donde se manda a pedir que se borren las entidades

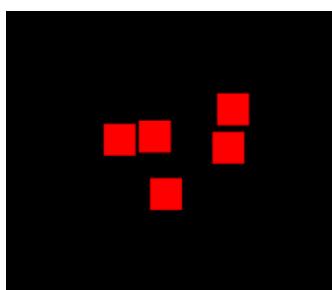
que hayan muerto y se realizan las comprobaciones de victoria y derrota de la partida.

Una vez se resuelve la partida, en el bucle de ejecución del juego se reiniciarán el nivel vaciando y volviendo a cargar los elementos del juego.

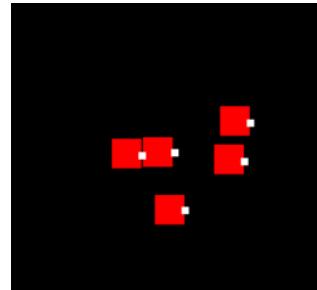
Para incluir los nuevos sistemas y funcionalidades a la rutina de las unidades, se ha ampliado el apartado de toma de decisión incluyendo métodos para detectar si hay unidades enemigas cercanas y seleccionar como objetivo a la más cercana, que en caso de morir se lanzará una nueva búsqueda. De forma similar, cuando no haya unidades enemigas cercanas se regresará al estado de patrulla (en caso de la IA) y las manejadas por el jugador volverán a la posición de su marcador en el mapa.

En cuanto a las unidades de la partida, se creó un manejador de equipos donde se gestiona que unidades pertenecen a cada facción, proporcionar métodos para obtener los *IDs* de las unidades por equipo e ir actualizando los equipos cuando haya bajas. Además de, tener métodos para crear tipos de unidades en concreto trabajando a la vez de “fábrica”.

Por otro lado, en el apartado visual, se ha añadido un marcador que indica la dirección a la que se está orientando la entidad con el fin de dar *feedback* al jugador y que se entienda mejor y de forma visual el desplazamiento o acción actual. Este marcador tiene un tamaño relativo al del sprite de la unidad y serán siempre de color blanco para que sea fácil de identificar.



Sin marcador.



Con marcador.

Figura 5.6: Marcador dirección entidad.

Por último, en cuanto a la memoria, se ha añadido en el apartado de metodología la descripción de las fases que componen una iteración junto a un esquema. Se ha introducido un tercer referente el cual basa toda su jugabilidad alrededor de los *Steering behavior*, debido a este juego hemos optado por cambiar la jugabilidad del proyecto con el fin de imitar la de este juego y con intención de innovar en los controles, como ya se refleja en el GDD, actualmente el jugador no maneja directamente a las unidades con el ratón ni las selecciona de forma individual, sino que controla un marcador verde el cual será seguido por sus tropas y mediante ordenenes activará los distintos comportamientos de sus unidades.



# Bibliografía

- Amordron (2019). TaB AI discussion. <https://steamcommunity.com/app/644930/discussions/0/1649917420753004999/>.
- Andygmb (2014). Age of empires II AI Scripting. <https://gist.github.com/Andygmb/1e3a6d9d444b2dfa8c40>.
- Bresenham (1962). Algoritmo de Bresenham. [https://es.wikipedia.org/wiki/Algoritmo\\_de\\_Bresenham](https://es.wikipedia.org/wiki/Algoritmo_de_Bresenham).
- Millington, I. (2009). *Artificial intelligence for games*. Morgan Kaufmann/Elsevier, Burlington, MA.
- Pikachunet (2018). TaB Informal level editor. [https://www.reddit.com/r/TheyAreBillions/comments/9zvffg/example\\_of\\_number\\_of\\_zombies\\_per\\_wave\\_800/](https://www.reddit.com/r/TheyAreBillions/comments/9zvffg/example_of_number_of_zombies_per_wave_800/).
- Pritchard, M. (2000). Postmortem: Ensemble Studio's Age of Empires II: Age of Kings. [https://www.gamasutra.com/view/feature/131844/postmortem\\_ensemble\\_studios\\_age\\_.php?page=1](https://www.gamasutra.com/view/feature/131844/postmortem_ensemble_studios_age_.php?page=1).
- Redmechanic (2017). A guide to scripting your own AI for AoEII. <https://steamcommunity.com/sharedfiles/filedetails/?id=1238296169>.
- Reynolds, C. (1986). Boids: Background and update. <http://www.red3d.com/cwr/boids/>.
- Sucasas, A. (2018). Entre las bambalinas de 'They Are Billions'. <https://www.xataka.com/videojuegos/entre-bambalinas-de-they-are-billions-el-nuevo-bombazo-del-videojuego-espanol>.