

Grado en Ingeniería Informática

IA para el Desarrollo de Videojuegos

Proyecto final

Documentación



Vladyslav Grechyshkin
Nicolás Fuentes Turpín
Jose Miguel Sánchez Almagro

24 de mayo de 2020

Índice general

1. Introducción	1
2. Software utilizado	1
3. Interfaz de usuario	2
3.1. Escenarios de laboratorio	2
3.2. Escenario final	2
3.2.1. Control de usuario	2
3.2.2. Interfaz gráfica	3
4. Bloque I. Movimiento	6
4.1. Jerarquía de clases	6
4.2. Scripts adicionales	6
4.3. Steerings básicos	7
4.3.1. Escenario propuesto	7
4.3.2. Steerings	8
4.4. Steerings delegados	9
4.4.1. Escenario propuesto	9
4.4.2. Steerings	10
4.5. Steerings cinemáticos	11
4.5.1. Escenario propuesto	11
4.5.2. Steerings	12
4.6. Detección de obstáculos	12
4.6.1. Escenario propuesto	12
4.6.2. Steerings	13
4.7. Steerings de grupo	13
4.7.1. Escenario propuesto	13
4.7.2. Steerings	14

4.8.	Formaciones	15
4.8.1.	Escenario propuesto	15
4.8.2.	Steerings	16
4.9.	Pathfinding	18
4.9.1.	Escenario propuesto	18
4.9.2.	Steerings	19
4.10.	Modo de depuración	21
5.	Bloque II. IA Estratégica y táctica	23
5.1.	Tipos de unidades y terrenos	23
5.1.1.	Unidades	23
5.1.2.	Terrenos	25
5.1.3.	<i>Pathfinding</i> táctico individual	25
5.2.	Escenario desarrollado	26
5.3.	Estados	27
5.3.1.	Conjunto de estados	27
5.3.2.	Diagrama de estados	30
5.3.3.	Modos de Combate	31
5.4.	Clases implementadas	31
5.4.1.	Interfaz gráfica	32
5.4.2.	Estrategia	32
5.4.3.	Máquina de estados	33
5.5.	Elementos adicionales	33

1. Introducción

Este documento contiene la memoria del proyecto final evaluable de la asignatura, atendiendo a las especificaciones de formato, estructura y contenido indicadas en el guion de la misma. El proyecto en cuestión radica en la implementación de diversos elementos de inteligencia artificial en entornos de videojuegos de guerra en tiempo real. Estos elementos se componen, siguiendo la estructura del documento de prácticas, en dos bloques fundamentales: movimiento (IA reactiva) y estrategia (IA estratégica y táctica).

Así pues, siguiendo el orden cronológico de la asignatura y los apartados indicados en el documento, los detalles de implementación y diseño más relevantes de dichos bloques son los siguientes:

2. Software utilizado

Para el desarrollo de este proyecto se ha utilizado exclusivamente el motor gráfico Unity, utilizando la misma versión que la disponible en los laboratorios de la facultad: 2019.2.17f1. No se destaca ninguna plataforma, librería o lenguaje adicional diferente a los utilizados en Unity.

3. Interfaz de usuario

Manteniendo el orden y estructura del documento, la explicación de la interfaz de usuario se desglosará según cada bloque, aunque hayan múltiples elementos comunes entre ambos.

3.1. Escenarios de laboratorio

Para todas las escenas de demostración de los elementos del primer bloque se utiliza la misma interfaz y controles de usuario, a saber:

- **Control de unidades.** El usuario podrá seleccionar aquellas unidades indicadas como seleccionables haciendo uso del click izquierdo del ratón. Este uso permite la selección individual de unidades haciendo click izquierdo sobre la unidad deseada y la selección múltiple haciendo click izquierdo y arrastrando una región rectangular que incluya las unidades a seleccionar. Asimismo, se podrá añadir elementos a la selección actual utilizando cualquiera de las formas descritas en conjunción a la tecla **SHIFT** izquierdo. Las unidades seleccionadas se iluminarán con un borde de color verde claro. Una vez seleccionadas las unidades de interés, el usuario podrá indicar mediante el click derecho del ratón una región de la escena a la cual deberán desplazarse las unidades seleccionadas. Estas unidades se desplazarán haciendo uso de los *steering* aplicados a ellas.
- **Control de cámara.** El usuario podrá desplazarse en la escena moviendo la cámara horizontal y verticalmente con las teclas **WASD**, rotar la cámara en incrementos/decrementos de 90° con las teclas **Q** y **E** y, finalmente, podrá acercar y alejar la cámara con la rueda del ratón.

3.2. Escenario final

La totalidad de los elementos a implementar del bloque II se incluye en el escenario final desarrollado (descrito en la sección del bloque en cuestión). Por tanto, la interfaz de usuario especificada a continuación quedará referida exclusivamente a dicho escenario.

3.2.1. Control de usuario

El usuario dispondrá de exactamente los mismos controles (salvo la rotación) que para los escenarios de laboratorio. Adicionalmente, podrá presionar la tecla **G** para añadir unidades a un mismo grupo y así visualizar el comportamiento táctico grupal del *pathfinding* implementado. Las unidades seleccionadas que pertenezcan a un grupo tendrán un borde de color amarillento. Si de todas las unidades seleccionadas existen aquellas que pertenecen al mismo grupo, los demás componentes del grupo sin seleccionar se iluminarán con un borde de color naranja. No obstante, esto no significa que estas últimas unidades estén seleccionadas. Asimismo, para eliminar cualquier

unidad seleccionada de un grupo, si pertenecen a uno, se podrá presionar la tecla **C**. Por último, presionando la tecla **M** se puede activar y desactivar la música de fondo. Existen las siguientes restricciones y consideraciones a la hora de crear grupos:

- Una unidad podrá pertenecer únicamente a un grupo. Si un usuario asigna una unidad perteneciente a un grupo a otro grupo, esta unidad dejará de pertenecer al anterior.
- Las unidades de un grupo deberán pertenecer al mismo equipo.
- Los grupos estarán constituidos como mínimo por dos unidades. Si un grupo se reduce a un único miembro, el grupo dejará de existir.

Del mismo modo, es preciso destacar que, a diferencia de los escenarios de laboratorio, el usuario únicamente podrá desplazar una única unidad a la vez. Por tanto, la selección de unidades múltiples quedará limitada exclusivamente para la creación de grupos. Otras restricciones de movimiento incluyen hacer click derecho sobre partes del mapa inalcanzables o zonas seguras.

3.2.2. Interfaz gráfica

Los elementos gráficos del escenario final se pueden clasificar en dos categorías diferentes: aquellos asociados a una unidad y aquellos asociados al juego en general. En la Figura 1 se muestran estos elementos en ejecución simultáneamente.

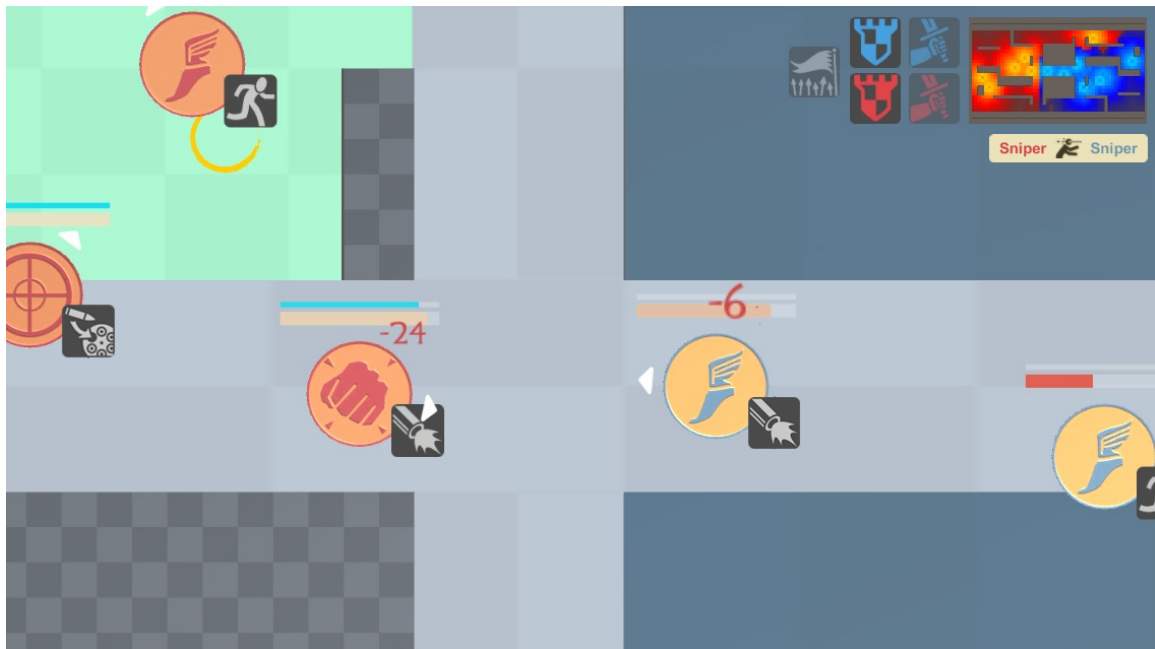


Figura 1: Ejemplo de ejecución del escenario final.

En cuanto a las unidades, estas se componen de los siguientes elementos visuales:

- **Ficha circular.** Define a la unidad, su equipo (rojo o azul) y el tipo de unidad que es según la paleta de colores e iconos utilizados. El icono de zapato con alas se corresponde con la unidad *scout*, el puño con el *heavy*, el símbolo más con el *medic* y la mira de francotirador con el *sniper*.
- **Barra de vida y munición.** Situadas en la parte superior de la unidad, indican la vida y munición actual de toda unidad respecto al máximo establecido.
- **Orientación.** Representada mediante un icono triangular alrededor de la ficha circular descrita, indica la orientación actual de la unidad. Este elemento es meramente estético y no influye en el comportamiento de la unidad.
- **Estado actual.** Denotado por el icono cuadricular en la parte inferior derecha de la unidad, muestra el estado actual de la unidad correspondiente con los estados descritos en la sección de IA estratégica y táctica.
- **Cambio de vida.** Cuando una unidad es curada o recibe cierta cantidad de daño, se muestra numéricamente en una posición al azar alrededor de la unidad. Si la cantidad de vida es curada, el indicador numérico es verde y si es daño, el indicador es rojo. Si la cantidad recibida es un golpe crítico, se muestra textualmente encima de la unidad receptora.

Respecto a los elementos asociados al juego en general, se distinguen:

- **Mini mapa.** En la parte superior derecha de la pantalla se muestra el mapa de influencia utilizado para el *pathfinding* táctico. Los colores cálidos se corresponden con la influencia del equipo rojo y los colores fríos se corresponden con la influencia del equipo azul.
- **Modos de combate.** Justo a la izquierda de dicho mini mapa se encuentran los botones para poder cambiar los modos de combate de cada equipo (denotados por colores). Estos modos son los indicados en el documento de prácticas. El modo defensivo se corresponde con los botones cuyo icono es un escudo y el ofensivo con los botones cuyo icono es una espada. Asimismo, el botón cuyo icono es un estandarte con lanzas representa el modo de guerra total. Al presionarlo, los modos anteriores se desactivarán y ambos equipos entrarán en el modo de guerra total.
- **Registro de muertes.** Situado debajo del mini mapa, muestra las muertes por combate generadas durante el desarrollo de la partida. Estas muertes son mostradas durante varios segundos e imitan el formato utilizado en el juego tomado de referencia para el desarrollo de este proyecto (Team Fortress 2). De izquierda a derecha, se indica el nombre del atacante (en el color de su equipo), el arma que se ha utilizado (un icono diferente para cada clase y ataque a melee o distancia), si ha sido por causa de golpe crítico (mediante un difuminado rojo) y el nombre de la víctima (en el color de su equipo). En todo momento se muestran como mucho cuatro registros en orden cronológico.

- **Estado de captura.** No visible en la figura referenciada hasta ahora, ambos puntos de captura existentes para la victoria de cada equipo disponen de una barra similar a las de vida y munición para indicar el progreso de la captura.
- **Victoria.** Cuando un equipo gana, se muestra el equipo ganador en el centro de la pantalla y se pausa el juego.

4. Bloque I. Movimiento

El primer bloque se compone por el desarrollo de un sistema *steering* en un escenario general que, a su vez, se compone de múltiples *steerings* individuales. Para su realización se ha seguido la jerarquía de clases propuesta por el profesorado y el pseudocódigo de la segunda edición del libro “Artificial Intelligence for Games” [1].

Como el pseudocódigo sobre el que se desarrolla cada *steering* individual está extensivamente explicado tanto en la teoría de la asignatura como en el propio libro, únicamente se indicarán matices de implementación que difieran o sean suficientemente significativos, siempre que se haya seguido el pseudocódigo correspondiente.

Asimismo, por comodidad, la explicación de los *steerings* implementados se organiza según a la escena “de laboratorio” a la cual pertenecen. Estas escenas, salvo que se indique lo contrario, dispondrán de los siguientes elementos:

- **Controles:** El usuario podrá desplazar la cámara y aquellas unidades indicadas como seleccionables según la especificación de la sección de “Interfaz de usuario”.
- **Unidades:** Quedará incluida una o más unidades de aspecto triangular por cada *steering* contenido en la escena. Estas unidades estarán denotadas por un color claramente distinguible a indicar en esta misma memoria.

Cualquier otra modificación a la estructura general de la escena se indicará en los correspondientes apartados.

4.1. Jerarquía de clases

Las únicas diferencias a destacar respecto a la jerarquía original es la inclusión directa de la clase **Cuerpo Físico** en la clase **Agent** y la incorporación necesaria para los *steerings* basados en velocidad. Esta incorporación se desglosa en tres elementos: una estructura, **KinematicSteering** (velocidad y rotación), un método adicional en la clase **SteeringBehaviour** llamado **GetKinematicSteering** y su correspondiente aplicación en la clase **AgentNPC**, **applyKinematicSteering**. Para determinar si el agente estará basado en *steerings* de velocidad o no, se utiliza un booleano.

De este modo, cada *steering* individual puede optar a implementar su versión basada en velocidad sin la creación de múltiples clases agente y *steering*. En el caso en el que no se desee incluir dicha versión o no sea conceptualmente posible, es suficiente con devolver un *steering* nulo para representar que el método está sin implementar.

4.2. Scripts adicionales

Para la construcción de los escenarios de prueba se ha hecho uso de una serie de *scripts* auxiliares, notablemente:

- **CameraController**. Permite el control de la cámara según la especificación correspondiente.
- **InputManager**. Gestiona la interacción del usuario con las unidades seleccionables, imitando el funcionamiento de un juego de estrategia.
- **DebugManager**. Utilizado para los *steerings* básicos y delegados. Asigna automáticamente los **target** de aquellos *steerings* cuyo objetivo sea un vector y se quiere que sea otro agente. Paralelamente, inicializa el camino a seguir por el *steering* de seguimiento de caminos.
- **GroupController**. Utilizado para los *steerings* de grupo. Inicializa los *steerings* de cada una de las unidades al ejecutar la escena.

4.3. Steerings básicos

En este apartado quedan englobados todos los *steerings* de movimiento uniformemente acelerado que se corresponden con la escena **Basic Steerings**.

4.3.1. Escenario propuesto

El escenario de demostración es el que se muestra en la Figura 2. Nótese que las indicaciones textuales han sido añadidas a posteriori para facilitar la comprensión y no forman parte de la escena en sí. El funcionamiento configurado de cada una de las unidades es el siguiente:

- Las unidades que contienen los *steerings* **Seek** y **Arrive** son seleccionables y controlables por el usuario.
- La unidad **VelocityMatch** imita la velocidad de la unidad **Seek**.
- La unidad **Align** imita la orientación de la unidad **Seek** y **AntiAlign** igualmente pero con orientación contraria. Para ello, se incluye un *steering* de **Look Where You're Going** a la unidad **Seek** para poder modificar su orientación, aunque no sea un *steering* básico.
- La unidad **Flee** huye de la unidad **Seek** si esta se encuentra dentro de un radio arbitrariamente establecido.



Figura 2: Elementos disponibles en la escena de Basic Steerings.

4.3.2. Steerings

Seek

En la escena, este *steering* está asociado a la unidad de color verde claro. La única diferencia respecto al pseudocódigo es la inserción de una variable `threshold` para evitar el resbalado o derrape inherente a este tipo de *steering*. Si el agente se encuentra a una distancia menor que `threshold`, se aplica una aceleración negativa equivalente a la actual del agente.

Flee

Este *steering* está asociado a la unidad de color naranja. Análogo a su *steering* contrario, Seek, también se apoya en el uso de una variable `threshold` para evitar el derrape de la unidad.

Arrive

Este *steering* tiene asociada la unidad de color verde oscuro. Para este *steering*, se deja como libre interpretación el significado de devolver un *steering* “nulo” en el caso en el que la unidad haya llegado a su destino. La interpretación elegida es la misma que para los *steerings* anteriores; cuando el agente haya llegado a su objetivo, aplicar negativamente la aceleración actual del agente.

Align

La unidad asociada a este *steering* es de color azul. Conceptualmente, no hay ninguna diferencia respecto al pseudocódigo, aunque la conversión al intervalo $(-\pi, \pi)$ es

explícita y no mediante una función externa.

Anti-Align

Este *steering* se aplica a la unidad de color rojo. Para obtener la orientación opuesta a **Align**, es suficiente con añadir π a la rotación calculada; el resto del código es idéntico al de **Align**.

Velocity Matching

La unidad de color blanco es la que representa este *steering*. Para su implementación no se ha realizado ninguna mejora o cambio al pseudocódigo original.

4.4. Steerings delegados

Similarmente, en este apartado se incluyen aquellos *steerings* basados (herencia de clases) en los básicos anteriores, correspondientes con la escena **Delegate Steerings**.

4.4.1. Escenario propuesto

El funcionamiento configurado para esta escena (véase la Figura 3) es el siguiente:

- No hay ninguna unidad seleccionable y controlable por el usuario.
- La unidad **Pursue** persigue, a una distancia de dos unidades, a la unidad **Wander**, mientras que esta se desplaza libremente por el escenario.
- La unidad **PathFollowing** “patrulla” el camino indicado por los cubos de color blanco.
- La unidad **Face** se orienta mirando a la unidad **PathFollowing**.
- Las unidades **Pursue**, **PathFollowing** y **Wander** se orientan en la dirección en la que se desplazan mediante el uso del *steering* **LookWhereYoureGoing**.

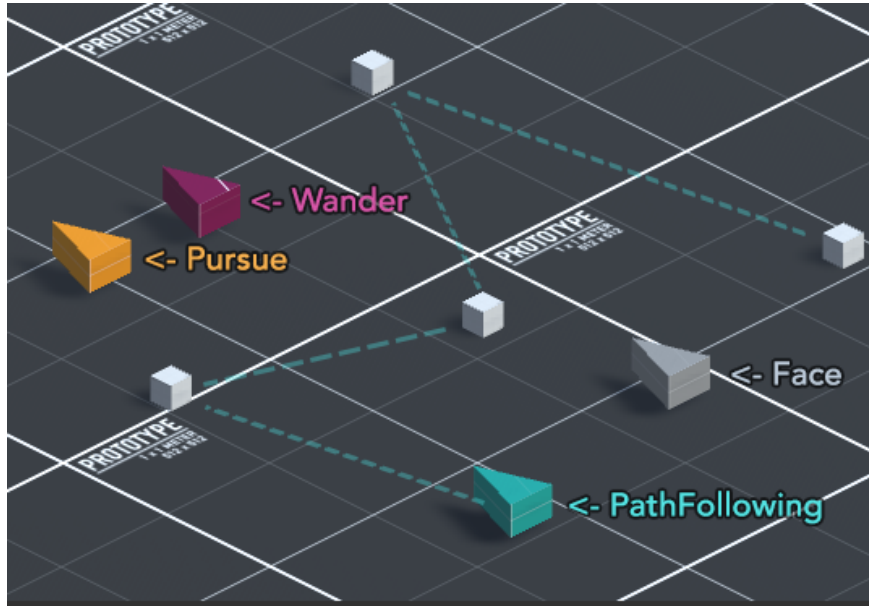


Figura 3: Contenidos de la escena Delegate Steerings.

4.4.2. Steerings

Pursue

La unidad de color naranja es la que representa este *steering*. Para su implementación no se ha realizado ninguna mejora o cambio al pseudocódigo original.

Face

Este *steering* se aplica a la unidad de color gris. En esta implementación tan solo se ha añadido una comprobación para cambiar la orientación solo si tenemos un **Target** al que mirar.

Look Where You're Going

Este *steering* está asociado a las unidades Wander, PathFollowing y Pursue. Interpretamos la llamada **return** del pseudocódigo como un **return none** al igual que en **Arrive**. Esto se consigue devolviendo la orientación actual como la de destino.

Wander

En la escena, este *steering* está asociado a la unidad de color vino. Nuevamente, para este *steering* se ha optado seguir el pseudocódigo propuesto (heredando de la clase **Face**). No hay ninguna modificación ni aportación significativa al código implementado.

Path Following

La unidad asociada a este *steering* es de color turquesa. Siguiendo la estructura sugerida en el libro, este *steering* se compone en dos clases complementarias: **Path-Following** y **Path**.

La primera no presenta ninguna diferencia sobre el pseudocódigo. En comparación, para la segunda solamente se ofrece una interfaz y el programador debe elegir su propia estrategia para determinar la posición del agente en el camino a seguir. Así pues, la implementación elegida es mediante segmentos de línea. Los detalles concretos de cada método descrito en la interfaz son:

- **GetParam**: Devuelve un entero que simboliza el punto actual del agente en el camino en función de un punto anterior pasado de parámetro. Si la posición anterior se encuentra fuera de los límites, la posición devuelta es el límite correspondiente. En caso contrario, el criterio utilizado para obtener el punto actual es el siguiente: si el agente se encuentra más cerca (distancia euclídea) del punto anterior o posterior y a menor distancia de un radio establecido, se devuelve dicho punto. De lo contrario, se devuelve el punto actual.
- **GetPosition**: Devuelve la posición en la escena asociada al entero obtenido en **GetParam**.
- **AppendPointToPath**: Añade un punto al final del camino actual.
- **ClearPath**: Elimina todos los puntos del camino e inicializa el camino a la posición actual del objeto asociado a este *steering*.

4.5. Steerings cinemáticos

En este apartado se recogen todos los *steerings* cinemáticos correspondientes con la escena **Kinematic Steerings**

4.5.1. Escenario propuesto

El funcionamiento configurado para esta escena (véase la Figura 4) es el siguiente:

- La unidad que contiene el *steering* **Seek** es seleccionable y controlable por el usuario.
- La unidad **Wander** se desplaza libremente por el escenario.
- La unidad **PathFollowing** “patrulla” el camino indicado por los cubos de color blanco.
- La unidad **Flee** huye de la unidad **Seek** si esta se encuentra a menos de una distancia arbitrariamente establecida.

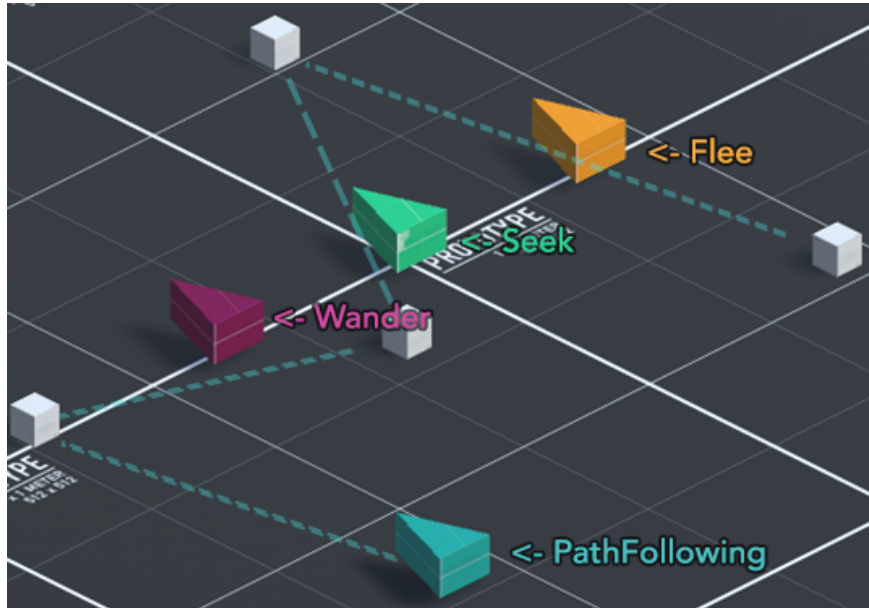


Figura 4: Contenidos de la escena Kinematic Steerings.

4.5.2. Steerings

El código implementado en estos *steerings* cinemáticos es análogo a sus versiones basadas en aceleración en el caso de **Seek**, **Flee** y **PathFollowing**. En el caso de **Wander**, se sigue el pseudocódigo correspondiente.

4.6. Detección de obstáculos

En este apartado se incluye el *steering* delegado restante: **ObstacleAvoidance**. El principal y único motivo de separar este *steering* del resto de *steerings* delegados es reducir la carga de contenidos de cada escena de demostración.

4.6.1. Escenario propuesto

La disposición y configuración de los elementos de la escena construida (Figura 5) es:

- Una única unidad seleccionable y controlable por el usuario con **Seek**, **ObstacleAvoidance** y **BlendedSteering**.
- Múltiples obstáculos de color naranja para visualizar el funcionamiento del *steering* **ObstacleAvoidance**.

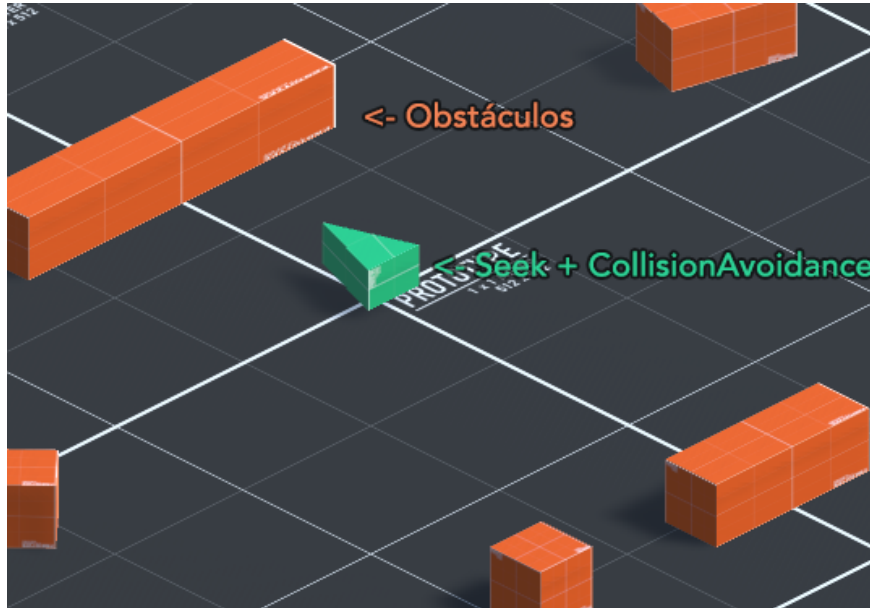


Figura 5: Contenidos de la escena `Obstacle Avoidance`.

4.6.2. Steerings

Nótese que, a pesar de que para la demostración del funcionamiento de `ObstacleAvoidance` se hace uso de dos *steerings* adicionales, estos se explican en sus respectivos apartados.

Obstacle Avoidance

Asociado a la unidad de color verde claro, la estrategia seguida para la implementación es el uso de las normales obtenidas por tres rayos de colisión: uno principal frontal y dos auxiliares de menor tamaño posicionados con un ángulo modificable respecto al principal.

Así pues, se repite el procedimiento descrito en el pseudocódigo para cada uno de los rayos de colisión. Por lo demás, el código es el mismo.

4.7. Steerings de grupo

Los *steering* descritos en este apartado se corresponden con los vistos en la asignatura necesarios para la incorporación de un comportamiento de *flocking*.

4.7.1. Escenario propuesto

Los elementos disponibles en la escena (Figura 6) y la configuración de su comportamiento son los siguientes:

- No hay ninguna unidad seleccionable y controlable por el usuario.

- Tres grupos de tres unidades con comportamiento de **Alignment**, **Separation** y **Cohesion** respectivamente.
- Un grupo de doce unidades con comportamiento combinado de **Alignment**, **Separation** y **Cohesion** mediante **BlendedSteering** para conseguir un efecto de *flocking*.

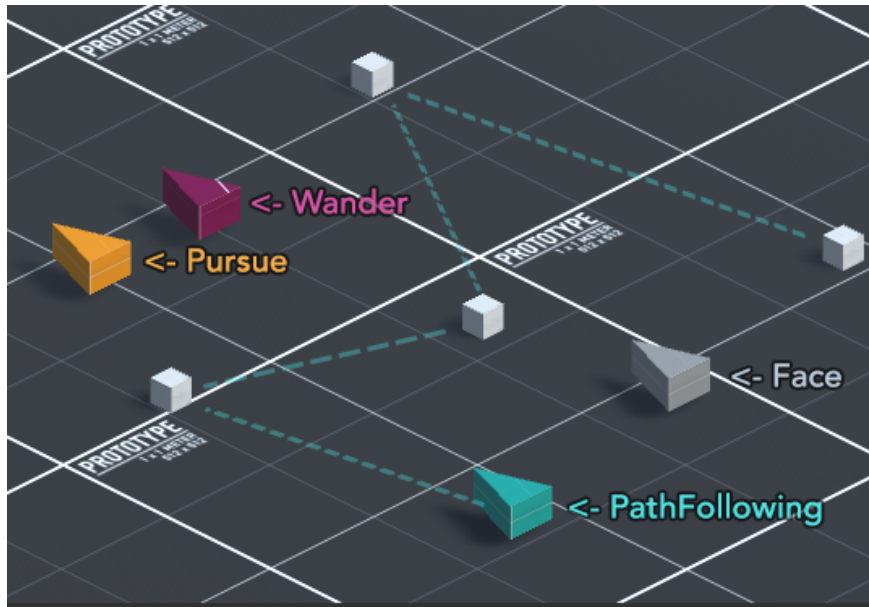


Figura 6: Cambiar imagen luego. Group Steerings.

4.7.2. Steerings

Blended Steering, Separation y Cohesion

No hay aportaciones significativas respecto al pseudocódigo disponible.

Alignment

Este *steering* deriva de la clase **Align** y siguiendo el pseudocódigo el funcionamiento del *steering* no se corresponde con el deseado. Esto se debe a que hay un fallo en el pseudocódigo original: la instrucción `Heading -= character.Heading()` carece de sentido; no guarda ninguna relación con el cálculo de la media previo a esta instrucción.

Flocking

El *flocking* reúne a cuatro *steerings* ya comentados: **Wander**, **Separation**, **Cohesion** y **Alignment** mediante una mezcla ponderada aplicada con **BlendedSteering**.

En este también se han realizado diversas modificaciones respecto al pseudocódigo de las transparencias. Se sustituyen las limitaciones de las aceleraciones que en el

pseudocódigo aparecen realizadas con la función máximo. Se realiza la función mínimo, dado que es algo incoherente en si mismo limitar un valor escogiendo el valor máximo de entre dos. Por otro lado, en el atributo `linear` del *steering* no se realiza el mínimo simplemente, con ese valor obtenido se multiplica al vector normalizado para obtener la aceleración deseada.

4.8. Formaciones

En este apartado quedan descritas las clases diseñadas para garantizar la generación de formaciones de estructuras fijas y escalables en dos niveles.

4.8.1. Escenario propuesto

Puesto que solo es preciso demostrar la transición de un estado desordenado a un estado “ordenado”, se presentan todos las formaciones desarrolladas en dos escenas (Figura 7), separando según sean formaciones fijas o escalables. La descripción completa de los elementos integrados en las escenas es la que procede:

- No hay ninguna unidad seleccionable y controlable por el usuario en la escena para las formaciones escalables. Para la escena de formaciones fijas, el usuario podrá controlar los líderes de cada formación (indicados mediante un círculo amarillo en la figura correspondiente) y, mediante la tecla Z del teclado, hacer que los demás miembros de la formación vayan a sus posiciones en cuestión.
- Se incluyen siete grupos de unidades en total, identificadas cada uno por un color diferente. Dos de los grupos tienen asociado una formación de estructura fija diferente, a saber: `FingerPattern` y `BoxPattern`. Los cuatro grupos restantes presentan formaciones escalables; un grupo se ordena mediante una formación de `LinePattern` y las tres restantes mediante `DefensiveCirclePattern` (con números de unidades diferentes).
- Al inicializar las escenas, los grupos de unidades harán una transición a su estado “ordenado” en función del correspondiente líder o centro de masas calculado.

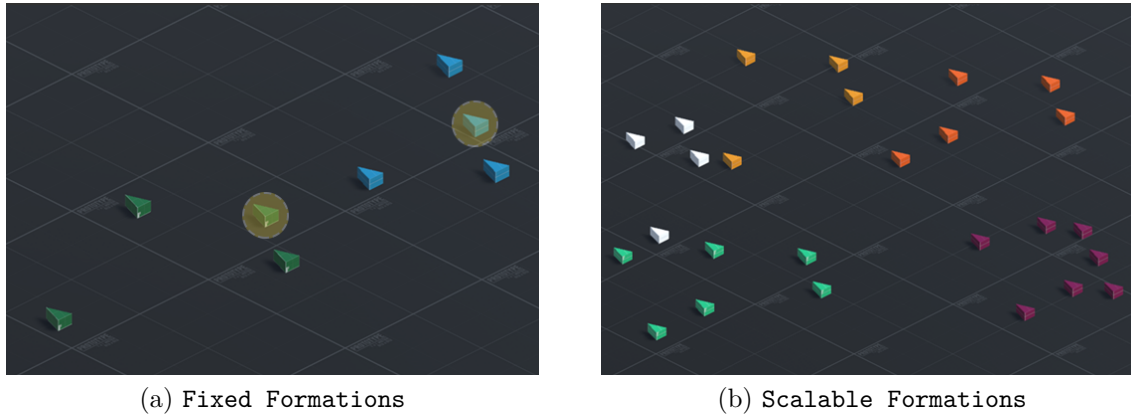


Figura 7: Escenas para la demostración del funcionamiento de las formaciones desarrolladas.

4.8.2. Steerings

De por sí, las estructuras de formaciones no presentan un *steering* nuevo, sino que hacen uso de *steerings* anteriores para generar una estructura fija o emergente. Las clases utilizadas para representar estas estructuras son similares pero con diferencias destacables. El diagrama de la Figura 8 muestra la jerarquía de clases utilizada.

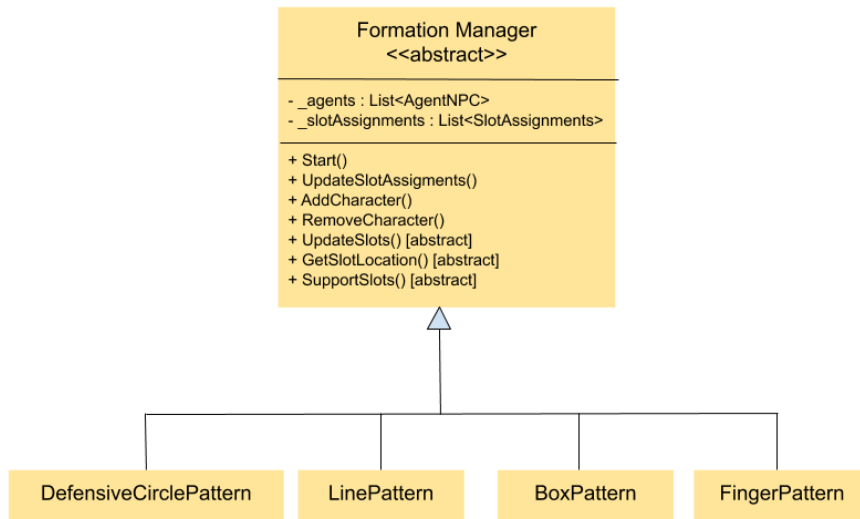


Figura 8: Diagrama de clases para las formaciones.

En esta jerarquía, se presenta una clase abstracta llamada **FormationManager** de la cual heredan todas las diferentes formaciones concretas. Asimismo, dentro de la clase abstracta se hace uso de dos estructuras diferentes para sintetizar el manejo de las variables involucradas: **Location**, representando una posición y orientación determinada de un miembro de la formación, y **SlotAssignment**, representando el

agente y su posición dentro de la formación. En cuanto a la funcionalidad, por un lado los métodos que deben implementar cada una de las subclases para obtener generar la formación deseada son:

- **UpdateSlots.** Puede considerarse el núcleo de la clase. Su función es asignar una posición y una orientación a cada elemento que conforma la formación.
- **GetSlotLocation.** Dado un número de slot que representa a un elemento de la formación devuelve su respectivo **Location**.
- **SupportSlots.** Devuelve un booleano indicando si se permite cierto número de unidades en la formación. En las formaciones escalables siempre será *true*.

Por otro lado, los métodos restantes que hacen uso de los métodos abstractos son:

- **UpdateSlotAssignments.** Asigna un valor numérico a cada posición o *slot* de la formación.
- **AddCharacter.** Añade un agente a la formación (en función de **SupportSlots**) y actualiza los valores de cada posición mediante **UpdateSlotAssignments**.
- **RemoveCharacter.** Elimina un agente de la formación (si pertenece a ella) y actualiza los valores de cada posición mediante **UpdateSlotAssignments**.

Two abreast in cover

Programada en la clase **BoxPattern** y representada mediante las unidades de color azul, se trata de una formación fija de cuatro unidades en aspecto de “caja” defensiva. Para su implementación, se hace uso de un líder (obtenido mediante la función auxiliar **GetLeader**); todas las posiciones de la formación son en relación al líder. Una vez obtenido el líder y todas las **Location** correspondientes, se itera sobre los miembros de la formación y se asigna la posición y rotación de destino de cada miembro.

Finger four

Implementada en la clase **FingerPattern** e integrada en las unidades de color verde oscuro, su funcionamiento es el mismo que el de la formación anterior. La única diferencia, evidentemente, es la posición concreta de cada miembro de la formación generada por **GetSlotLocation**. El resultado final es una formación con aspecto de los dedos de una mano (excluyendo el pulgar).

Línea

Disponible en la clase **LinePattern**, es la primera de las dos formaciones escalables programadas (no tiene un máximo de unidades que pueden ser asignadas a la misma). En la escena se corresponde con las unidades de color amarillo. Produce una formación en la que todos los miembros se posicionan paralelamente en una línea mirando en la

misma dirección. A diferencia de las formaciones anteriores, no se hace uso de un líder sino de una estructura de dos niveles; el punto de anclaje utilizado es la posición y orientación media de los miembros de la formación (obtenido mediante `GetAnchor`). Nótese, no obstante, que la orientación podría parametrizarse si se desea que la línea se oriente en una dirección concreta.

Para obtener las posiciones de cada uno de los miembros de la formación (esto es, la función `GetSlotLocation`) se realiza una sencilla operación aritmética de desplazamiento de posición.

Círculo defensivo

Finalmente, la última formación implementada se encuentra en la clase `DefensiveCirclePattern`. Para su desarrollo se ha seguido el pseudocódigo disponible en la bibliografía utilizada, a excepción del *drift*. La única diferencia sobre el código original es la consideración de la posición en la escena de los miembros para la obtención del punto de anclaje. En la escena en la que se incluye, las unidades (agrupadas por color) con este tipo de formación son las de color blanco, naranja, verde claro y vino.

4.9. Pathfinding

En este apartado se incluye el último *steering* desarrollado: la búsqueda de caminos o *pathfinding*. Tal y como se especifica en el documento de prácticas, las variantes concretas a implementar son LRTA* (obligatoriamente para el primer bloque) y A* (opcionalmente para el segundo bloque).

4.9.1. Escenario propuesto

Se distinguen dos escenarios diferentes, uno para cada variante. El primer escenario (Figura 9), descrito en este apartado, sirve para demostrar el funcionamiento de LRTA* y sus contenidos se describen a continuación. Contrariamente, como el uso de A* quedaría reflejado en el segundo bloque, el escenario de demostración sería el propio escenario general a desarrollar en dicho bloque. No obstante, aunque la intención inicial era implementar A* y la jerarquía de clases propuesta acomoda ambas variantes, al final la decisión ha sido únicamente utilizar LRTA*.

Así pues, los contenidos son:

- Dos unidades con **Pathfinding** seleccionables y controlables por el usuario, uno cinemático y otro basado en aceleraciones. Por defecto, únicamente la unidad cinemática está activada puesto que visualiza el camino obtenido sin depender de derrapes generados por aceleraciones. No obstante, si el usuario lo desea, puede utilizar la otra unidad activando el objeto `Acceleration Pathfinder` desde la escena.
- Un escenario dividido por celdas cuadriculares con obstáculos por el cual la

unidad podrá desplazarse. Los límites del escenario quedan denotados por el tinte amarillo aplicado a las celdas visitables. El objetivo de este escenario es visualizar las limitaciones de “visión” de LRTA*.

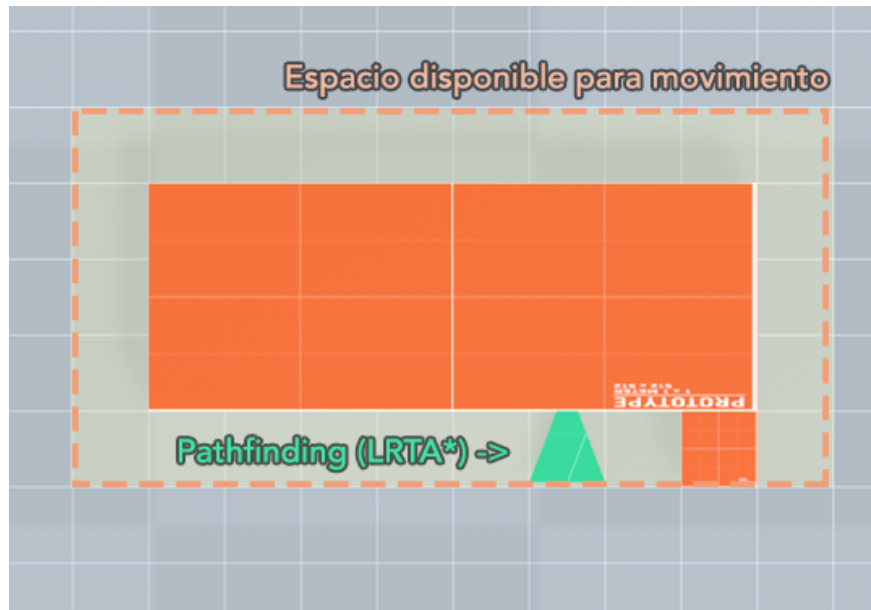


Figura 9: Contenidos de la escena Pathfinding.

4.9.2. Steerings

Para la implementación de este *steering* se ha seguido una estructura de clases completamente ajena a la referenciada en la bibliografía utilizada hasta ahora. Destacamos dos elementos fundamentales para el funcionamiento de este *steering*: el mapa cuadrícula (*grid*) y el *pathfinding*.

Grid

La estructura y funcionamiento del mapa cuadrícula se basa en los tutoriales del canal de Sharp Accent [2]. Las clases específicas implementadas son **Tile** y **GridMap**.

La primera, **Tile**, simboliza una celda individual y su información asociada, esto es, la posición de la celda en la cuadrícula, posición global, tipo de terreno de la celda y si se trata de una celda sobre la cual una unidad puede pasar.

La segunda, **GridMap**, representa la cuadrícula completa compuesta de **Tile** individuales y permite la obtención de cualquier celda, sus adyacentes y la transformación de posiciones de celda a posiciones de escena y viceversa. Para la creación de la cuadrícula se hace uso de dos métodos:

- **ReadLevel**. Determina los límites y tamaño de la cuadrícula en función de dos marcadores posicionados en la escena.

- **CreateGrid**. Llamada al final de **ReadLevel**, genera una cuadrícula de tamaño dado inicializando cada celda individual. Para cada celda, se comprueba el tipo de terreno y obstáculos observando los objetos que residen dentro de la celda. Si se encuentra un obstáculo, se marca la celda como infranqueable. De lo contrario, se le asigna el tipo de terreno correspondiente.

Pathfinding

Por decisión completamente subjetiva, se ha optado implementar la búsqueda de caminos sobre el seguimiento de caminos descrito anteriormente. La jerarquía resultante se descompone en tres clases: **PathFinding**, superclase abstracta de la cual heredan las dos siguientes; **LRTA** y **A** (no implementada).

PathFinding hereda de la clase **PathFollowing** e incluye la información común a ambas implementaciones concretas de *pathfinding* desarrolladas. Esta información es el cálculo de las heurísticas de una celda a otra en función de diferentes tipos de distancias (Manhattan, Chebyshev y euclídea). La construcción de cada uno de los métodos para el cálculo de las distancias está basada en las explicaciones de Amit Patel [3].

Luego, **LRTA** y **A** (si hubiera sido implementada) heredan de la clase **PathFinding** e implementan el método abstracto **FindPathToPosition** que, dada una celda de comienzo y fin, buscan y generan el camino de un extremo al otro. Este método es utilizado por la superclase en forma de co-rutina para evitar el cálculo secuencial por parte del agente.

La implementación del algoritmo de LRTA* ha sido de manera estricta y fiel a la bibliografía de la cual procede el pseudocódigo desarrollado en las transparencias de la asignatura [4]. Por tanto, la estructura y métodos concretos utilizados se corresponden directamente con el pseudocódigo original. No obstante, cabe destacar los siguientes detalles de implementación empleados para la mejora de eficiencia y memoria del algoritmo:

- El algoritmo implementado permite cualquier espacio de búsqueda local y no se limita a la versión **LookAheadOne**, cumpliendo pues con uno de los objetivos opcionales propuestos. Los espacios de búsqueda concretos desarrollados son **LookAheadOne** y una búsqueda en anchura (**BreadthFirstSearch**). Esta búsqueda en anchura añade celdas hasta que se satisfaga el tamaño establecido en el orden característico de este tipo de búsqueda; primero se añaden todos los vecinos de la celda actual y luego los vecinos de los vecinos en el orden que fuesen añadidos.
- El uso de una estructura interna llamada **TileInfo** para almacenar información heurística relativa al algoritmo y una copia perezosa del *grid* original con la información añadida. Este enfoque garantiza la concurrencia de agentes de *pathfinding* eliminando secciones críticas al almacenar individualmente cada agente

sus valores heurísticos. Similarmente, al ser una copia perezosa del mapa cuadrangular, solamente se inicializa la información adicional de las celdas una vez las visita el agente.

- El uso de marcas de tiempo o *timestamps* para reducir el tiempo de inicialización del algoritmo entre una ejecución y otra. Cuando el agente termina su ejecución e inicia otra completamente diferente, los valores heurísticos actualizados de cada celda dejan de ser válidos. En vez de reiniciar todos los valores heurísticos (lo cual supone un coste equivalente al tamaño de celdas del mapa visitadas hasta el momento), cada **TileInfo** se actualiza cuando la marca de tiempo en la que se ejecutó el algoritmo no se corresponde con la almacenada por dicho **TileInfo**.
- Como el algoritmo no necesita conocer la estructura interna del *grid* al solicitar las celdas adyacentes o “vecinos” mediante el método **GetAdjacentTiles** de la clase **GridMap**, teóricamente podría utilizarse cualquier grid, sea cuadrangular o no. Asimismo, las búsquedas de los vecinos de coste mínimo dependen del orden en el que se reciban estos vecinos. En el caso particular del mapa implementado, es en el sentido del reloj, empezando por el vecino izquierdo. Cualquier empate que se produzca se resolverá en función de ese orden.
- Por último, dado que tanto este algoritmo como A* únicamente genera el camino y no dicta cómo debe ser atravesado, cualquier posible mejora como *string pulling* puede ser añadida. Del mismo modo, al heredar de la clase **PathFollowing**, el agente puede desplazarse mediante movimientos cinemáticos y no basados en aceleración.

4.10. Modo de depuración

A continuación se detallan los distintos *Gizmos* que se muestran en el modo depuración. Todo lo explicado en esta sección se puede activar desde el inspector de Unity mediante un booleano:

- **Agent**. Todos los agentes pueden mostrar sus radios internos y externos (usados en **Arrive**) así como sus ángulos internos y externos (usados en **Align** y **AntiAlign**). Asimismo, pueden mostrar su vector velocidad.
- **AgentNPC**. Este tipo de agente puede mostrar adicionalmente su vector aceleración (siempre que el *steering* aplicado no sea cinemático).
- **ObstacleAvoidance**. En este *steering* se puede mostrar la distancia que nuestra unidad mantiene con los obstáculos (en forma de esfera) y los vectores que comprueban si existe una posible colisión (tanto el principal como sus bigotes).
- **Path**. La clase **Path** permite mostrar el radio de llegada de todos los puntos del camino, es decir, a la distancia mínima que ha de estar nuestra unidad para que se considere que ha llegado a ese punto.

- **Pathfinding.** Se pueden mostrar todas las celdas visitadas por la unidad y sus heurísticas (mientras éstas se van actualizando).
- **GridMap.** Se puede visualizar todas las celdas existentes, franqueables (color gris) o no (color rojo). Para el segundo bloque también se distinguen celdas no franqueables pero que sí permiten visibilidad (color azul).

5. Bloque II. IA Estratégica y táctica

El segundo y último bloque se compone en el desarrollo de elementos de inteligencia artificial en un entorno de juego de guerra en tiempo real. Siguiendo los aspectos y consideraciones del documento de prácticas, para su desarrollo se ha optado por tomar de referencia un juego multijugador de disparos en primera persona: Team Fortress 2. Todos los tipos de unidades y su información estática están basados directamente en este juego. Asimismo, los iconos referentes a cada unidad y otros elementos visuales también están directamente extraídos de este o, en su defecto, inspirados en él.

A excepción de los elementos obligatorios especificados en el guion de prácticas, el formato de la estructura y lógica del juego es totalmente subjetivo según el juicio de los componentes del grupo.

5.1. Tipos de unidades y terrenos

El entorno desarrollado dispone de cuatro tipos de unidades diferentes y tres tipos de terreno que influyen su *pathfinding* táctico.

5.1.1. Unidades

Scout. Este tipo de unidad tiene la mayor velocidad de movimiento de todos. Su arma a distancia es una escopeta que utiliza varias balas de daño reducido en cada disparo y su velocidad de ataque cuerpo a cuerpo y captura es el doble que la del resto de unidades.



(a) Red Scout



(b) Blu Scout

Figura 10: Icono de unidad Scout

Heavy. Este tipo de unidad tiene una velocidad de movimiento reducida pero una gran cantidad de vida base. Su arma a distancia es una ametralladora con una alta cadencia de disparo y un cargador de gran tamaño. Este tipo de unidad es la única que patrulla durante la partida.



(a) Red Heavy



(b) Blu Heavy

Figura 11: Icono de unidad Heavy

Medic. Este es el único tipo de unidad capaz de curar. Tiene una cantidad de vida ligeramente incrementada y su arma a distancia dispara dardos que hieren a los enemigos y curan a los aliados, pero tiene un cargador muy pequeño.



(a) Red Medic



(b) Blu Medic

Figura 12: Icono de unidad Medic

Sniper. La característica principal es su arma a distancia, un francotirador de largo alcance con un gran tiempo de carga pero daño explosivo.



(a) Red Sniper







(b) Blu Sniper

Figura 13: Icono de unidad Sniper

5.1.2. Terrenos

En la siguiente tabla se muestran los multiplicadores que se le aplica a la velocidad base de cada unidad dependiendo del tipo de terreno de la celda en la que se encuentren.

Unidad	Bosque	Pradera	Carretera
	0.9	1	1.25
	0.75	1	1.5
	1	1	1.5
	1	1.1	1.25

5.1.3. *Pathfinding* táctico individual

Para garantizar el comportamiento táctico en el *pathfinding* desarrollado se incluyen modificaciones al cálculo de los costes de cada celda. De este modo, se incluye una variable que indica si se aplica *pathfinding* táctico o no. Si esta se activa, se aplican los costes asociados al terreno, mapa de influencia [5] y visibilidad. Los costes de terreno y de visibilidad se suman directamente al coste final, mientras que los costes de influencia dependen del equipo al que pertenezca la unidad que quiere crear el camino. Si pertenece al equipo rojo, ignora influencias positivas e intenta evitar las negativas y si pertenece al equipo azul lo realiza de manera inversa.

5.2. Escenario desarrollado

El escenario desarrollado para el segundo bloque es el siguiente:

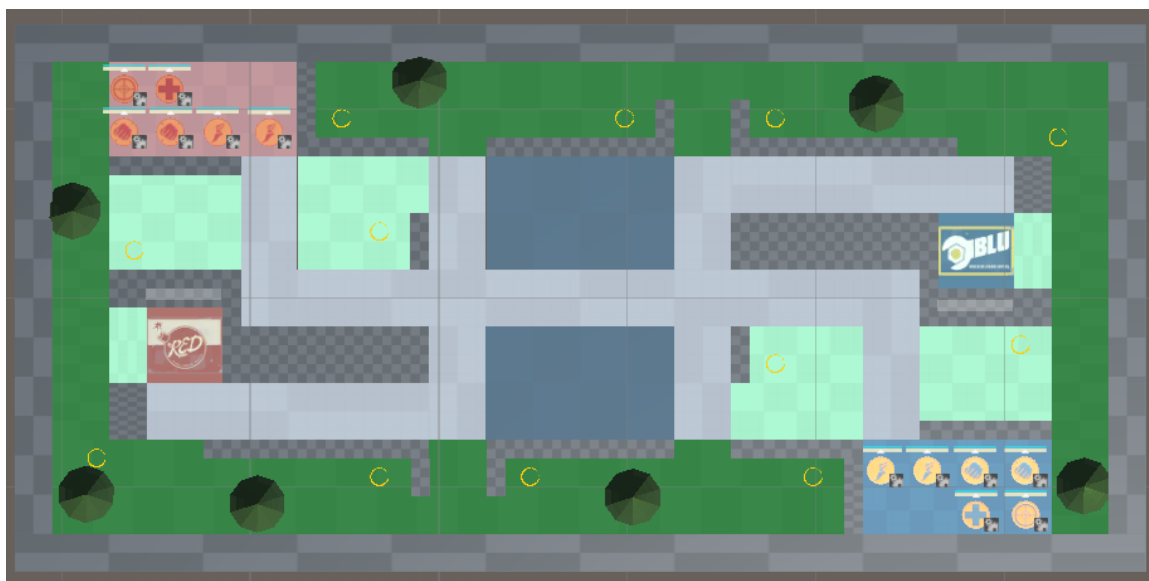


Figura 14: Contenidos de la escena **Strategy Scene**.

Este escenario se compone por los diferentes tipos de terrenos expuestos en el apartado anterior, junto a los *waypoints* de cobertura (los círculos de color amarillento), bases y puntos de captura de ambos equipos, obstáculos y elementos decorativos. Haciendo un desglose por colores, las partes del mapa de color gris claro se corresponden con carreteras (*Road* en el código), las de color verde claro con praderas (*Grassland*), las de color verde oscuro con bosques (*Forest*), las de color azul con agua (zonas infranqueables pero que sí permiten visibilidad) y, finalmente, las partes grisáceas en forma de cuadrícula se corresponden con obstáculos (infranqueables y no permiten visibilidad).




En cuanto a las bases, estas son del color correspondientes a cada equipo (rojo y azul) y se encuentran en las esquinas superior izquierda e inferior derecha del escenario. Del mismo modo, los puntos de captura son aquellas partes del mapa que contienen el logo de cada equipo.





Por la disposición de los elementos del escenario, el mapa queda dividido en dos países (tal como y se indica en la especificación de la práctica), dejando únicamente tres pasos diferentes entre ambas regiones del mapa. Esta disposición conforma un rico escenario que ayuda a las unidades a funcionar de una manera más estratégica e inteligente y asegura que todas las partidas sean diferentes entre ellas, debido a la variedad de situaciones posibles que pueden llegar a producirse.






5.3. Estados

La funcionalidad y el comportamiento de las unidades se modela con una máquina de estados. Esta máquina está compuesta por un número determinado de estados y conexiones definidas para garantizar el comportamiento táctico y estratégico de las unidades.

5.3.1. Conjunto de estados

Icono	Estado	Descripción
	Idle	Es el estado de inicio de todas las unidades desde el cual se puede hacer una transición a cualquier otro. Cuando el personaje se encuentra en este estado no realiza ninguna acción, únicamente determina el estado al cual pasar. Por tanto, todas las unidades solamente estarán intermitentemente en este estado para ver qué acción tomar.
	Dead	Este estado es al que van las unidades cuando mueren. Cuando están en él no realizan ninguna acción, simplemente esperan una cantidad de tiempo prefijada para poder revivir. Cuando reviven, recuperan toda la vida y munición.
	Escape	Cuando una unidad tiene baja vida o está en peligro pasa a este estado. Una vez en él, determinará si escapar a base, a un aliado (en el caso de que esté en peligro) o un médico (en el caso de que tenga poca vida y el médico pueda curarlo) en función de su distancia a estas tres opciones. Si elige huir hacia un aliado, cuando la unidad llega a dicha posición comprueba si el aliado sigue ahí. En caso negativo, huye a base (recuperando toda la vida y munición). Si se ha activado el modo de guerra total, las unidades nunca huyen.

	Capture	El objetivo del juego es capturar la zona enemiga. Por tanto, cuando una unidad cumpla las condiciones adecuadas, accederá a este estado e intentará avanzar al checkpoint enemigo y capturarlo, para así ganar la partida. Las condiciones de entrada son que la unidad en cuestión no deba patrullar (se ignora si es modo de guerra total), no hayan enemigos cerca, hayan suficientes aliados capturando (dependiente de cada unidad y modo de combate) y que tenga la vida suficiente para hacerlo. Asimismo, si una unidad enemiga alcanza el punto de captura aliado, saldrá de este estado si su distancia a la zona de captura aliada es menor que la zona de captura enemiga.
	Defend	Por el contrario al estado anterior, cuando una unidad enemiga alcanza el punto de captura aliado, acudirá a defenderlo siempre que no esté capturando ya el punto enemigo y tenga vida suficiente (nuevamente, esto último se ignora si el modo de guerra total está activado).
	MeleeAttack	Permite el combate cuerpo a cuerpo de una unidad. Cualquier unidad pasará a este estado siempre que no pueda atacar a distancia (por no tener munición). Para realizar un ataque cuerpo a cuerpo, primero intentará acercarse (una única vez) a su objetivo. Al llegar, procederá a atacar repetidas veces a su objetivo hasta matarlo o perder demasiada vida (se ignora en el modo de guerra total).
	RangedAttack	Permite el combate a distancia a una unidad. Cualquier unidad pasará a este estado siempre que vea a un enemigo en su campo de visión y tenga munición. Al entrar en el estado, se cargarán disparos consecutivos contra el objetivo siempre que este siga en su campo de visión y tenga munición y vida suficiente (una vez más, se ignora lo último en el modo de guerra total). En el caso particular de los médicos, estos podrán utilizar su munición para “disparar” a un aliado y curarlo.

	Reload	Cuando una unidad se queda sin munición, debe recargar su arma. Procederá a hacerlo siempre que no tenga una acción más importante a realizar y no hayan enemigos cerca. Al entrar en el estado, buscará el punto de cobertura más cercano para ir a recargar. Si en el proceso de desplazarse al punto de cobertura se encuentra con enemigos, volverá directamente a base para recargar.
	User	Tal y como se indica en la interfaz de usuario, todas las unidades son seleccionables y movibles. Cuando se produce dicha acción, la unidad pasa directamente a este estado en el que solo obedece las órdenes del jugador. Cuando la unidad llega a su destino sale automáticamente de este estado.
	Patrol	Cualquier unidad designada para patrullar realizará esta acción siempre que no deba hacer ninguna de las acciones anteriores. El modo de patrulla consiste únicamente en desplazarse linealmente entre dos puntos prefijados.
	Roam	Si una unidad no cumple ninguna de las condiciones de los estados anteriores, optará por deambular por puntos de interés de su “país”.
	Heal	Cualquier unidad permanecerá en este estado mientras está siendo curada, ya sea por un médico o cuando se encuentra en base (curándose automáticamente). La unidad saldrá del estado cuando recupere suficiente vida en el caso de ser curado por un médico o cuando recupere la vida al completo en base. En este último caso, también recuperará la munición.

5.3.2. Diagrama de estados

Así pues, los estados descritos y sus condiciones quedan reflejados en el siguiente diagrama de estados:

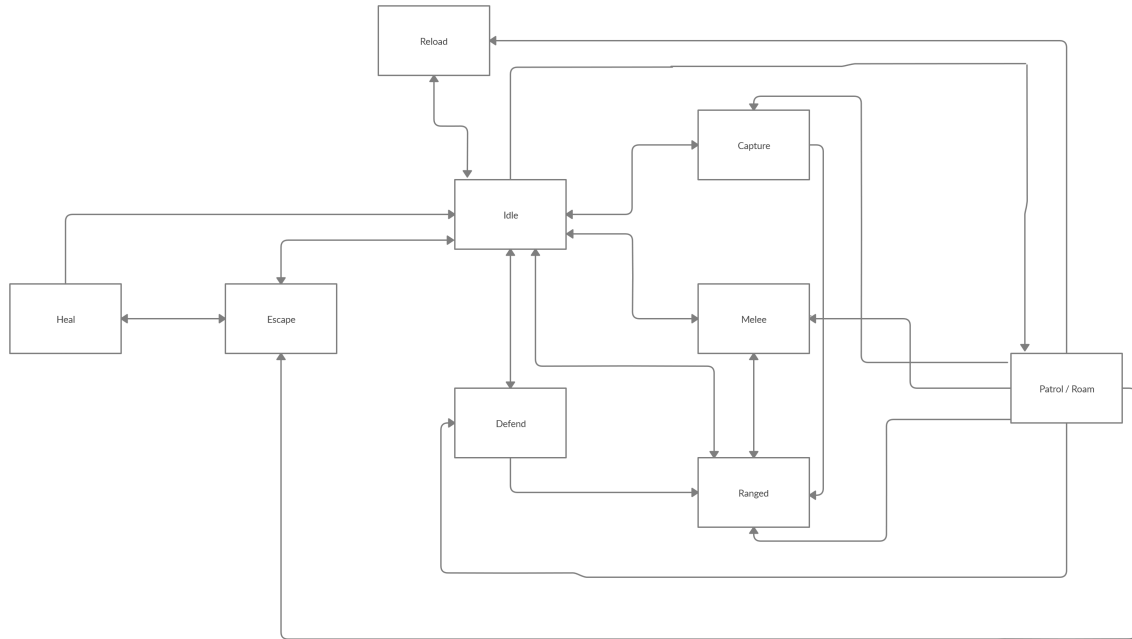


Figura 15: Diagrama de estados de las unidades del juego.




Nótese que hay dos estados que no se incluyen en el diagrama por simplicidad:

- **Dead**. Todos los estados pueden pasar a **Dead**, de modo que únicamente dificultaría la comprensión del diagrama. Cuando una unidad revive, únicamente puede pasar a **Idle**.
- **User**. Similarmente al estado anterior, todos los estados pueden pasar a **User** y desde este solo se puede volver a **Idle**.

Observando el diagrama, se solidifica la noción de que **Idle** actúa de estado central desde el cual se pueden producir todas las demás acciones. No obstante, no significa que la única manera de acceder a cierto estado es a través de **Idle**. Similarmente, los estados **Patrol** y **Roam** tienen un comportamiento similar. Como ambos estados se acceden únicamente si las demás acciones no son posibles, se simplifica el flujo de las unidades al hacer que estos estados puedan comprobar todos los demás. Por último, cabe destacar que **Heal** es únicamente accesible desde **Escape** puesto que en todos los casos para poder curarse, la unidad debe huir.

5.3.3. Modos de Combate

Estas transiciones entre estados pueden variar dependiendo del modo de combate que este activo durante la partida. Existen 3 diferentes modos que se detallan a continuación.

Icono	Modo	Descripción
	Defensive Mode	Es el modo predeterminado. Mientras está activo se prioriza la defensa del punto de captura aliado mediante la disminución del número de enemigos máximos para huir y atacar a melee, aumentando la importancia de la influencia enemiga en el <i>pathfinding</i> e incrementando el número mínimo de aliados para atacar a melee así como reduciendo los valores LowHealth y Healthy de las unidades. Además, en este estado las unidades deciden ir a capturar si tienen un cierto número de aliados en el punto de captura enemigo, a excepción del Scout .
	Offensive Mode	Este modo prioriza la conquista del punto de captura enemigo. Esto se consigue aumentando el número máximo de enemigos que deben de encontrarse cerca para que una unidad decida huir o atacar a melee. Asimismo se disminuye el número de aliados necesarios para atacar a melee y las estadísticas <i>LowHealth</i> y <i>Healthy</i> . En este modo todas las unidades son capaces de capturar sin restricción de aliados y se aumenta la importancia del mapa de visibilidad respecto al de influencia en el <i>pathfinding</i> .
	Total War Mode	En este modo prevalece la “valentía” por encima de cualquier objetivo, eliminando por completo la mecánica de escape. Asimismo, las unidades que deben de patrullar dejan de hacerlo. Los valores de LowHealth y Healthy decrecen de manera significativa, así como el número de aliados mínimo para entablar combate cuerpo a cuerpo. En este modo el <i>pathfinding</i> táctico no tiene en cuenta la influencia enemiga o la visibilidad y todas las unidades pueden capturar sin restricciones. El efecto deseado es buscar las condiciones de victoria a cualquier coste.

5.4. Clases implementadas

Se destacan tres conjuntos de clases necesarios para el funcionamiento de los elementos relacionados con el segundo bloque.

5.4.1. Interfaz gráfica

En este conjunto quedan englobadas todas las clases relacionadas con el funcionamiento de la interfaz gráfica, a saber:

- **GUIManager**. Maneja todas las llamadas relacionadas con la actualización de los elementos visuales de la escena (animaciones, cambios de estado, carga de imágenes, etc).
- **GUIKill**. Constituye un registro de muerte individual, encargándose de inicializarse con los valores adecuados según la muerte que haya ocurrido durante un combate.
- **GUIKillFeed**. Compone el registro de muertes completo, gestionando la generación y actualización de múltiples **GUIKill**.
- **GUICaller**. Utilizado estrictamente para poder hacer llamadas a eventos durante la animación del cambio de estado.
- **FloatingText**. Encargada de la inicialización y destrucción de los indicadores numéricos generados durante cambios de vida.

5.4.2. Estrategia

Este segundo conjunto se corresponde con las clases utilizadas para el control del estado de juego, la entrada del usuario y de las interacciones entre unidades. Estas clases son:

- **CombatManager**. Actuando de patrón fachada para el manejo del combate del juego, dispone de los métodos necesarios para poder realizar los diferentes tipos de ataque disponibles (melee y a distancia).
- **StrategyInfluenceManager**. Fundamentalmente, es una copia extendida de **InputManager**. Con el fin de no aglutinar una clase que ya es utilizada para los escenarios de laboratorio, esta clase adopta todas las funcionalidades anteriores e incluye la generación de grupos descrita en la interfaz de usuario.
- **GroupManager**. Clase estática auxiliar utilizada por **StrategyInfluenceManager** para llevar un registro correcto de todos los grupos existentes. Alternativamente, podría incluirse directamente dentro de dicha clase.
- **GameManager**. Constituye el núcleo principal del flujo del juego. Los cambios de modo de ataque y la obtención de información táctica (enemigos atacando, aliados defendiendo, etc) se realizan a través de esta clase.
- **UnitsManager**. Clase estática auxiliar para el cálculo de información táctica referida estrictamente a las diferentes posiciones de las unidades como por ejemplo el número de enemigos o aliados cercanos de una unidad.

- **Waypoint.** Almacena la información estática de los diferentes *waypoints* disponibles en el juego. En el caso de los puntos de captura, también se actualiza el indicador visual correspondiente.
- **WaypointManager.** Como su nombre indica, maneja la obtención de información referida a los *waypoints* disponibles.

5.4.3. Máquina de estados

El último conjunto incluye todas las clases referidas a lógica individual de cada unidad participante en la escena. Estas clases son:

- **NPC.** Contiene toda la información estática y dinámica referida a cada unidad. Asimismo, controla la ejecución de la máquina de estados implementada y sus elementos visuales.
- **State.** Super-clase abstracta que define toda la información y lógica de un estado, así como la definición de las comprobaciones necesarias para la transición de un estado a otro.
- Finalmente, las clases de cada uno de los estados contemplados en los apartados anteriores. Sus nombres y comportamiento son los ya descritos.

5.5. Elementos adicionales

Para cumplir con los requerimientos necesarios para la evaluación de la práctica en caso de ser un grupo de tres personas, se ha optado por introducir los siguientes elementos adicionales de los especificados en el guion de la práctica:

- *Pathfinding* táctico a nivel de grupo. Haciendo uso de la creación de grupos especificada en la sección de interfaz de usuario, se puede modificar el *pathfinding* táctico utilizado por cada unidad. Como no se especifica la forma en la que deben influir los grupos en el *pathfinding*, se ha optado únicamente por modificar la velocidad y costes de terreno de las unidades pertenecientes a un grupo tal que compartan todos los valores de la unidad más lenta. Esto es, si se hace un grupo entre un *heavy* y un *scout*, el *scout* irá a la velocidad del *heavy* y tendrá sus costes de terreno. Opcionalmente, también se podría modificar los pesos utilizados para el comportamiento táctico en función de la “peor” unidad.
- Visibilidad. Se ha implementado un mapa de visibilidad que es utilizado en los costes del *pathfinding* táctico. La generación de este mapa de visibilidad se realiza al inicio del juego y, para cada celda, se almacena, aproximadamente, el número de celdas visibles desde esa celda. Cabe destacar que el valor almacenado en cada celda del mapa de visibilidad está normalizado e invertido para que las celdas con un valor de visibilidad más “pequeño” sean las más favorables. El motivo de esta modificación es simplificar y mantener la uniformidad de los cálculos realizados desde el *pathfinding* táctico.

- Mezcla de niveles y otras propiedades de información táctica. Nuevamente, este apartado es altamente genérico y subjetivo, de modo que existen múltiples enfoques para el mismo proyecto. La información adicional implementada es el uso de puntos de cobertura para recargar munición, la búsqueda de unidades médicas para recibir curación si estas se encuentran más cerca que la base aliada, la consideración de peligro para la toma de decisiones (en forma del número de enemigos y aliados cercanos para decidir si huir o atacar).

Referencias

- [1] Ian Millington y John Funge. *Artificial Intelligence for Games, Second Edition*. 2nd. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2009. ISBN: 0123747317.
- [2] Sharp Accent. *(NEW) Turn Based Game Part 1 Grid Creation - Unity Tutorial (Advanced)*. Youtube. 2019. URL: <https://www.youtube.com/watch?v=x7drd0tSftk>.
- [3] Amit Patel. *Heuristics - Amit's Thoughts on Pathfinding*. 1997. URL: <https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>.
- [4] Stefan Edelkamp y Stefan Schroedl. *Heuristic search: theory and applications*. Elsevier, 2011.
- [5] Ferdinand Joseph Fernandez. *Influence Maps in Unity 3d*. 2012. URL: <https://bitbucket.org/AnomalousUnderdog/influencemapsunity3d/src/Dev-Main/InfluenceMapUnity/>.