



UNIVERSIDAD
DE GRANADA

TRABAJO FIN DE GRADO
INGENIERÍA EN ...

Desarrollo de una demo de videojuego de simulación

La creación de Ant Simulator

Autor

Yoram Joel Slot

Directores

Nombre Apellido1 Apellido2 (tutor1)

Nombre Apellido1 Apellido2 (tutor2)

Aquí se puede incluir nombre y logo del
Departamento responsable del proyecto



Escuela Técnica Superior de Ingenierías Informática y
de Telecomunicación

—
Granada, mes de 2025

Alternativamente, el logo de la UGR puede sustituirse / complementarse con uno específico del proyecto



UNIVERSIDAD DE GRANADA

Desarrollo de una demo de videojuego de simulación

La creación de Ant Simulator

Autor

Nombre Apellido1 Apellido2 (alumno)

Directores

Nombre Apellido1 Apellido2 (tutor1)
Nombre Apellido1 Apellido2 (tutor2)

AntSim Demo: Diseño y desarrollo de un juego de simulación en 3D

Nombre Apellido1 Apellido2 (alumno)

Palabras clave: palabra clave1, palabra _clave2, _____palabra _____clave3,

Resumen

Poner aquí el resumen.

Project Title: Project Subtitle

First name, Family name (student)

Keywords: Keyword1, Keyword2, Keyword3,

Abstract

Write here the abstract in English.

Yo, **Nombre Apellido1 Apellido2**, alumno de la titulación **TITULACIÓN de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI XXXXXXXXXX, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Nombre Apellido1 Apellido2

Granada a X de mes de 201 .

Índice

1 Introducción.....	9
1.1 Motivación.....	9
2 Resumen.....	9
3 Bibliografía.....	10
4 Creación del terreno dinámico: Marching Cubes.....	10
4.1 Razonamiento.....	10
4.2 Marching Cubes, ¿Qué es?.....	11
4.3 Funcionamiento del algoritmo de Marching Cubes.....	12
4.4 Implementación del algoritmo de Marching Cubes.....	14
4.5 Edición del mapa en tiempo real.....	15
4.6 Chunks.....	16
5 Desarrollo gráfico del juego.....	18
5.1 Concepto estilístico general: low-poly.....	18
5.2 Obtención y creación de modelos.....	18
5.2.1 Hormiga trabajadora.....	18
5.2.2 Hormiga reina.....	19
5.2.3 Maíz.....	19
5.2.4 Huevo.....	21
5.3 Rigging.....	22
5.3.1 Creación del esqueleto.....	22
5.3.2 Skinning.....	23
5.4 Creación de animaciones.....	24
5.4.1 Walk cycle.....	24
5.4.2 Idle animation.....	25
5.4.3 Girar.....	25
5.4.4 Interact.....	26
5.4.5 Carrying.....	26
5.4.6 Putting down.....	26
5.4.7 Attack.....	26
5.4.8 Getting hit.....	26
5.4.9 Dying.....	27
5.4.10 Dead.....	27
5.4.11 Hatch.....	27
5.5 Implementación de las animaciones e acciones.....	28
5.5.1 Animator.....	28
5.5.2 Modelo inicial: animaciones básicas de movimiento.....	29
5.5.3 Segundo modelo: hormiga llevando objeto.....	30
5.5.4 Tercer modelo: implementación de acciones de interactuar con el entorno.....	31
5.5.5 Cuarto modelo: Implementación de guardado y cargado.....	32
5.5.6 Quinto modelo: Edad y nacimiento de la hormiga trabajadora.....	32
5.5.7 Modelo de la reina hormiga.....	33
5.6 Color del fondo: debajo o encima del terreno?.....	34
5.7 Aplicando color al terreno.....	34
5.8 Uso de partículas.....	35
5.8.1 Qué son partículas.....	35
5.8.2 Obtención de partículas.....	36
5.8.3 Representar Feromonas.....	36

5.8.4 Representar DigPoints.....	38
5.9 Representación gráfica del nido.....	38
5.9.1 Incorporación de la Universal Rendering Pipeline.....	38
5.9.2 Buffer de stencil.....	39
5.9.3 Uso de distintos colores en las partes del nido.....	40
5.9.4 Uso de Render Queue.....	40
5.10 Consecuencias de uso de URP.....	41
5.10.1 Necesidad de nuevos shaders para cada material.....	41
5.10.2 Creación de shader en URP para el terreno.....	41
6 Físicas del juego.....	42
6.1 Interacción de la hormiga con el terreno.....	42
6.1.1 Los dos estados de la hormiga.....	42
6.1.2 Gestionar movimiento sobre malla y más Raycasts.....	43
6.1.3 La orientación de la hormiga sobre la malla.....	44
6.1.4 Centro de gravedad de la hormiga.....	45
6.1.5 Sostener la hormiga sobre la superficie.....	45
6.1.6 Rotar la hormiga para seguir la malla.....	46
6.2 Interacción entre hormigas.....	47
6.2.1 Implementación inicial.....	47
6.2.2 Hormigas caminando sobre otras hormigas.....	47
6.2.3 Cambio de colisionador.....	48
6.2.4 Huevos de hormigas.....	48
6.3 Interacción entre hormiga y otros objetos.....	49
6.4 Interacción de objetos llevados por las hormigas.....	49
6.4.1 Mientras son llevados por las hormigas.....	49
6.4.2 Al ser depositados en el mapa.....	50
7 Sistema de movimiento de la hormiga.....	50
8 Sistema de pathing de la hormiga.....	52
8.1 Qué es pathfinding.....	52
8.2 Navmesh.....	53
8.3 Diseñar pathfinding y feromonas.....	53
8.3.1 Primera versión: nodos de feromonas en puntos enteros.....	53
8.3.1.1 Cambiar el sistema de detección de feromonas.....	54
8.3.1.2 Representar caminos distinguibles.....	55
8.3.1.3 Errores y soluciones.....	55
8.3.1.4 Veredicto final.....	56
8.3.2 Segunda versión: feromonas sobre la superficie.....	56
8.3.2.1 Cuando depositar feromonas.....	56
8.3.2.2 Eliminación de PheromoneNode.....	57
8.3.2.3 Detección y seguimiento de las feromonas.....	57
8.3.2.4 Llegar a una feromona.....	57
8.3.2.5 Feromonas auxiliares.....	57
8.3.2.6 Veredicto final.....	57
8.3.3 Versión final: adyacentCubes.....	58
8.3.3.1 CubeSurface.....	58
8.3.3.2 Obtener superficies adyacentes.....	58
8.3.4 CubePheromone: caminos de feromona.....	59
8.3.4.1 Creando caminos en CubePheromone.....	59
8.3.5 La simplificación del sistema de feromonas y implementación de edad.....	60
8.3.6 Usar seguimiento de goles con cubeSurface.....	61
8.3.6.1 Error de la hormiga moviendo entre dos cubos.....	61

8.4 La implementación de algoritmos de búsqueda de caminos.....	62
8.4.1 Búsqueda en anchura.....	62
8.4.2 A*.....	63
9 Sistema de excavación.....	63
9.1 División de excavación en pequeñas tareas.....	63
9.2 Desarrollo de <i>DigPoints</i>	64
9.3 Conversión de áreas del nido a <i>DigPoint</i>	65
9.3.1 Obtener los valores de los puntos: <i>GetMarchingValue()</i>	65
9.3.2 Iterar sobre los puntos que hay que modificar.....	66
9.4 Restricciones de inicialización de <i>DigPoints</i>	66
9.5 El problema de la separación del terreno.....	67
9.5.1 Orden de excavación de <i>DigPoints</i>	68
9.5.2 Excavación automática de <i>DigPoints</i> inaccesibles.....	68
10 El nido de las hormigas.....	68
10.1 Clase <i>NestPart</i>	69
10.1.1 Visualización.....	69
10.1.2 Como se coloca un túnel.....	70
10.1.3 Restricciones de la colocación del túnel.....	71
10.1.4 Como se coloca una cámara.....	71
10.1.5 Prevención de intersección de cámaras.....	71
10.1.6 Otras restricciones de colocación de una cámara.....	73
10.1.7 Cómo comprobar si ha sido excavado un <i>NestPart</i>	73
10.2 Clase <i>Nest</i>	74
10.2.1 Memoria del nido y mandar Tasks.....	74
10.2.2 Saber si se está en el nido.....	74
10.2.3 Gestionar los objetos dentro del nido.....	76
11 Inteligencia artificial.....	76
11.1 Objetivos.....	76
11.2 Fluent behaviour tree.....	77
11.2.1 Descripción.....	77
11.2.2 Funcionamiento.....	77
11.3 Implementación de Task.....	78
11.3.1 Motivo.....	78
11.3.2 Idea general.....	79
11.3.3 Tipos de acciones y tareas.....	79
11.3.4 Prevenir que múltiples hormigas reciban tareas para el mismo objeto.....	80
11.4 Versión final de la implementación del árbol de comportamiento.....	80
11.4.1 Sección de obtención de tarea <i>Task de la hormiga trabajadora</i>	81
11.4.2 Sección de ejecución de tarea <i>Task de la hormiga trabajadora</i>	84
11.4.3 Sección de obtención de tarea <i>Task de la hormiga reina</i>	86
11.4.4 Sección de ejecución de tarea <i>Task de la hormiga reina</i>	88
12 Sistema de guardado y cargado.....	89
12.1 Guardado y cargado del mapa.....	89
12.1.1 Texto plano.....	89
12.1.2 SharpSerializer.....	89
12.1.3 Codificación y decodificación.....	90
12.1.4 Redundancia del mapa.....	91
12.2 Qué es SharpSerializer.....	91
12.3 Por qué usar SharpSerializer.....	91
12.4 Serialización de los objetos con SharpSerializer.....	92
12.4.1 ¿Qué hay que guardar?.....	92

12.4.2 ¿Como se hace que una clase sea serializable?.....	92
12.4.3 Relaciones entre objetos e indexación.....	93
12.4.4 Clases Información.....	93
12.4.5 Variables en GameData.....	95
12.4.6 Guardado y cargado de mapas.....	95
12.5 Tipos de archivos guardados.....	96
12.6 Guardando información sobre cada partida.....	97
12.7 Estructura de archivos de guardado.....	97
13 Interfaz gráfica y modos de juego.....	97
13.1 Menú principal.....	98
13.1.1 Clase GameSettings.....	98
13.1.2 Submenú de selección de mapa.....	99
13.1.3 Submenú de selección de archivo de guardado.....	100
13.1.4 Submenú de creación de mapa: tipo de mapa.....	100
13.1.5 Submenú de creación de mapa plano.....	101
13.1.6 Pantalla de carga.....	101
13.2 Editor de mapas.....	102
13.3 Modo de juego.....	104
13.3.1 Botón de gestión del nido.....	105
13.3.2 Botón de visibilidad de las feromonas.....	105
13.3.3 Botón de control de hormigas.....	106
13.3.4 La sección de información.....	106
13.3.5 La sección de menú y guardado.....	107
13.3.6 La sección de notificaciones.....	107
14 Combinar simulación con juego.....	107
14.1 Requerir interacción y mantenimiento.....	108
14.1.1 La falta de necesidad de mantenimiento del nido.....	108
14.1.2 Solución: requerir la expansión constante del nido.....	108
14.1.3 La falta de interacción directa con la simulación.....	109
14.1.4 Solución: control directo de las hormigas.....	109
14.2 El problema de la espera.....	109
14.2.1 Acelerar la velocidad de excavación de la reina.....	109
14.2.2 Acelerar la velocidad de nacimiento de las primeras hormigas.....	109
14.3 La falta de un objetivo general.....	110
14.3.1 Crear una meta para el juego.....	110
14.3.2 Esconder información del jugador.....	110
15 Planificación del proyecto.....	111
15.1 Metodología usada.....	111
15.2 Preplanificación.....	111
15.3 Planificación y desarrollo.....	112

1 Introducción

2 Resumen

El objetivo de este TFG es documentar y realizar el diseño y desarrollo desde cero de un videojuego en 3D. El juego será una simulación de una colonia de hormigas, en la que el jugador tiene control indirecto sobre las acciones del nido. Se realiza en Unity, una plataforma de desarrollo de videojuegos.

Algunos de los objetivos principales del proyecto son:

- Creación de un terreno dinámico, que permitirá a las hormigas excavar en cualquier parte del mapa designado por el jugador.
- Que las hormigas sean agentes simples independientes, que compartiendo información y trabajando juntos simulen una inteligencia de colonia
- Que las hormigas exploren el mapa mediante un sistema de caminos de feromonas
- Crear animaciones 3D estilístico.
- Implementar un estilo artístico low poly
- Finalizar una demo jugable del juego

3 Bibliografía

[repositorio oficial](#) behaviour tree

Unity manual

blender manual

<https://www.zendesk.com.mx/blog/metodologia-agil-que-es/#>

[Corn cob icons created by ADI_ICONS – Flaticon](https://www.flaticon.com/free-icons/corn-cob) for corn cob icon

4 Creación del terreno dinámico: Marching Cubes

4.1 Razonamiento

Una de las mecánicas que se quiso destacar de la demo es la capacidad del jugador para poder elegir cualquier parte del mapa como la localización del nido. Esta libertad crearía mucho valor de rejugabilidad, ya que habría una infinidad de formas de jugar incluso en un mismo mapa. Por desgracia, cuanta más libertad le concedes a un jugador en un videojuego, más situaciones imprevistas, glitches y exploits pueden aparecer. Es necesario entonces encontrar una forma muy simple y flexible de representar el mapa del mundo, que puede ser editado e interactuar con las otras partes del juego fácilmente.

Una de las grandes inspiraciones de la idea del terreno dinámico fue Deep Rock Galactic, un juego cooperativo creado por Ghost Ship Games en el que hasta cuatro jugadores exploran y minan en sistemas de cuevas altamente dinámicas. Los jugadores pueden picar y perforar cualquier parte del terreno, proporcionando una variedad infinita de formas de avanzar en cada mapa. Se quiso emular esta libertad de interacción con el terreno en la demo del juego.

Ghost Ship Games nunca ha compartido demasiada información sobre el funciona-

miento de sus juegos, pero en una entrevista con los desarrolladores de Deep Rock Galactic aludieron al uso de un sistema similar al algoritmo Marching Cubes para generar su terreno y CSG (Constructive Solid Geometry) para la destrucción de este. Esto último se basa en editar mallas formadas por volumen sustrayéndoles otras mallas de volumen. No fue usado en ninguna capacidad en este proyecto dado su complejidad y falta de necesidad, pudiendo modificar el terreno directamente en el algoritmo Marching Cubes.



Figure 1: La portada del videojuego Deep Rock Galactic, creado por Ghost Ship Games

Una de las otras opciones que investigué fue el sistema de terreno por defecto de Unity. Este consiste en una malla simple con valores de altura en cada coordenada de su plano horizontal. Cada punto del plano horizontal de la malla se encuentra a la altura definida por sus coordenadas. Esto significa que no puede formar túneles, ya que estos requieren de una forma más compleja que un plano modificado. Además de que no es apto para la creación de un terreno altamente tres dimensional, modificar este para crear aperturas para cuevas en su superficie de forma realista no sería posible. Por estas razones, se descartó rápidamente la idea de usar los objetos terreno nativos de Unity y se pasó a estudiar e implementar el algoritmo Marching Cubes.

4.2 Marching Cubes, ¿Qué es?

Marching Cubes es el algoritmo que finalmente se implementó para representar la superficie del terreno. Es un algoritmo de gráficos por computadora originalmente publicado en SIGGRAPH en 1987 por Lorensen y Cline. Se usa para crear superficies poligonales como representación de una isosuperficie de un campo escalar 3D. En concreto, dado un campo tridimensional escalar de valores numéricos en el que se encuentra un volumen, en el que cada punto dentro del volumen tiene un valor mayor o igual que la constante isolevel y cada punto fuera del volumen tiene un valor de menos de isolevel; el algoritmo puede crear la superficie aproximada entre el volumen y las afuera colocando una malla de triángulos entre ellos. Este proceso es simplificado dividiendo el espacio en cubos que comparten caras, donde una arista de un cubo será cortado por la superficie si de los dos vértices que lo forman, 1 se encuentra dentro del volumen y el otro fuera. Así, dado los 8 vértices del cubo, hay 256 (2^8) posibles combinaciones de polígonos dentro del cubo que corten sus aristas.

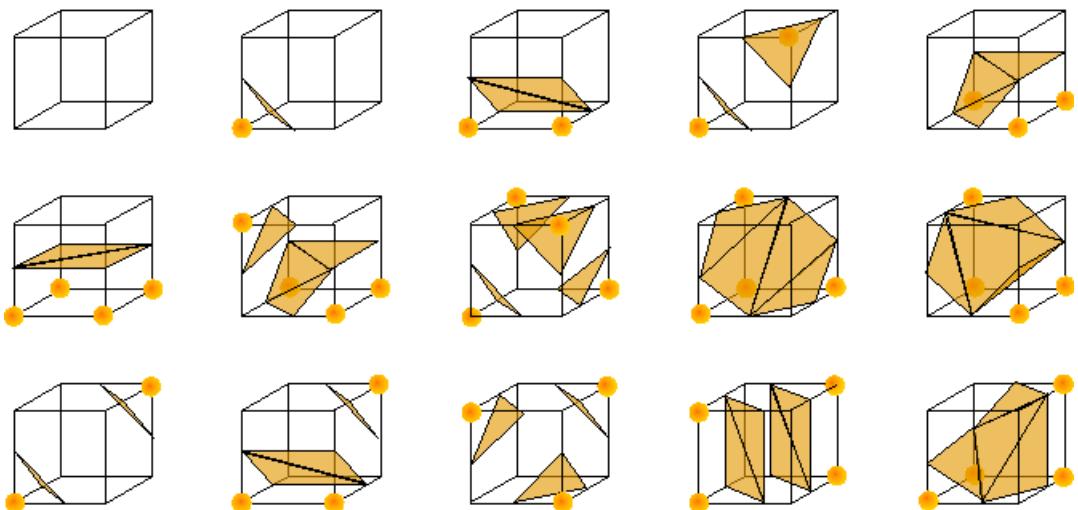


Figure 2: Los distintas casos posibles en un cubo sin tener en cuenta distintas orientaciones

La dificultad a la que se enfrenta al representar así un volumen es la gran cantidad de combinaciones posibles y la necesidad de que aquellas combinaciones conecten correctamente entre los cubos para formar la malla. Calcular y generar los triángulos de cada cubo requeriría además bastante computación. Sin embargo, el algoritmo Marching Cubes toma ventaja del número limitado de combinaciones posibles en cada cubo, guardando todas estas posibilidades en lookup tables. Estas lookup tables guardan las 256 combinaciones posibles de triángulos y eliminan la necesidad de su cálculo y generación, ahorrando mucho tiempo de cómputo y acelerando la generación de las mallas.

4.3 Funcionamiento del algoritmo de Marching Cubes

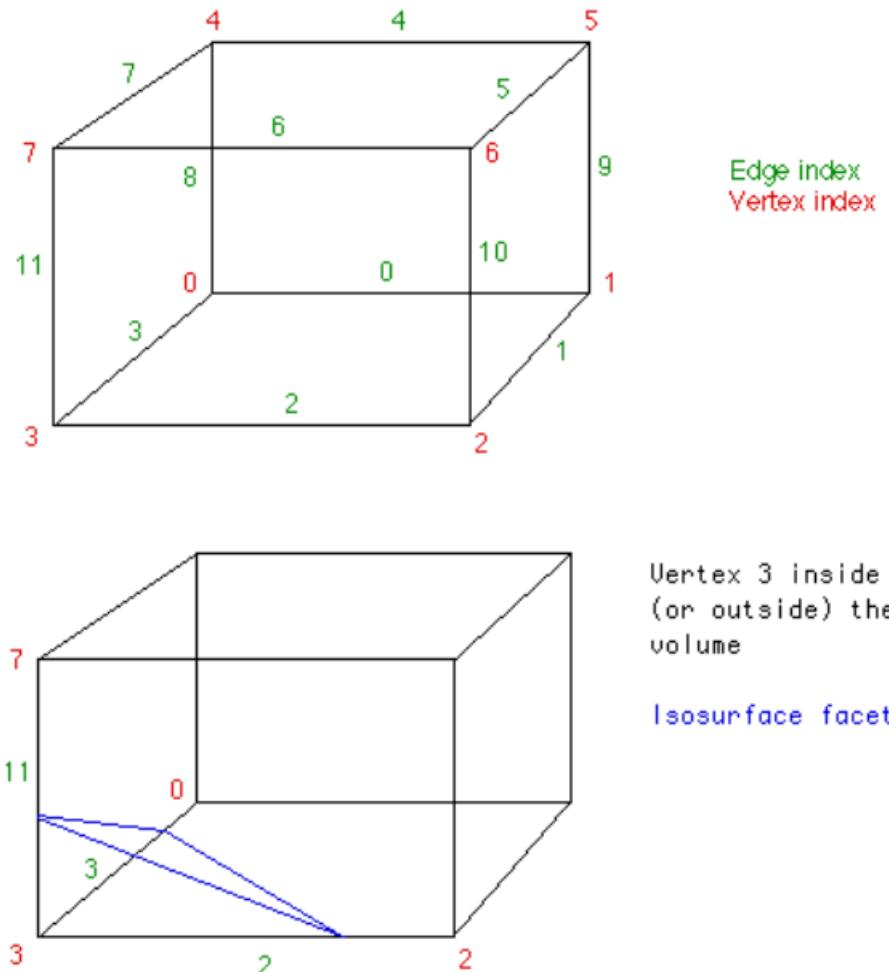


Figure 3: Ejemplo de un cubo en el que la superficie corta los lados 2, 3 y 11

Los pasos que sigue el algoritmo para formar la malla son los siguientes:

1. Para cada cubo, se obtienen los ejes cortados por la isosuperficie usando los valores de los vértices del cubo. Dado los vértices que se encuentran debajo del volumen, se obtiene un índice que se usa en la tabla de ejes para conseguir los ejes que la superficie del volumen corta. Por ejemplo, dada la figura 3, el índice correspondiente sería 0000 1000 o 8 en binario. Se obtiene de la siguiente forma:

```

cubeindex = 0;
if (cube.corner[0] < isolevel) cubeindex |= 1;
if (cube.corner[1] < isolevel) cubeindex |= 2;
if (cube.corner[2] < isolevel) cubeindex |= 4;
if (cube.corner[3] < isolevel) cubeindex |= 8;
if (cube.corner[4] < isolevel) cubeindex |= 16;
if (cube.corner[5] < isolevel) cubeindex |= 32;
if (cube.corner[6] < isolevel) cubeindex |= 64;
if (cube.corner[7] < isolevel) cubeindex |= 128;

```

CodeBlock 1: Código que obtiene el índice para un cubo

La tabla de ejes devuelve un número de 12 bits que representan los 12 ejes del

cubo. Para cada bit, si su valor es 1, significa que el eje respectivo es cortado por la superficie. Si su valor es 0, es que no es cortado. Volviendo al ejemplo de la figura 3, al buscar en la tabla de ejes usando el índice 8, se obtendría el número 1000 0000 1100. Esto significa que los ejes cortados son el 2, el 3 y el 11 (el primer bit es el eje 0).

2. Se calculan los puntos de intersección de la malla en los ejes cortados. Esto se hace usando interpolación lineal a base de los valores de los dos vértices que forman el eje. Permite representar con más precisión el volumen, ya que no limita la malla a solo cortar por los centros de los ejes de los cubos. El impacto de la interpolación lineal sobre la malla es enorme. Sin ella, no es factible representar terreno de forma realista, como se puede ver en la figura 4. Dado los puntos P1 y P2 que forman el eje cortado y sus valores escalares V1 y V2 respectivamente, el punto de intersección en el eje viene dado por:

$$P = P1 + (\text{isolevel} - V1)(P2 - P1)/(V2 - V1).$$

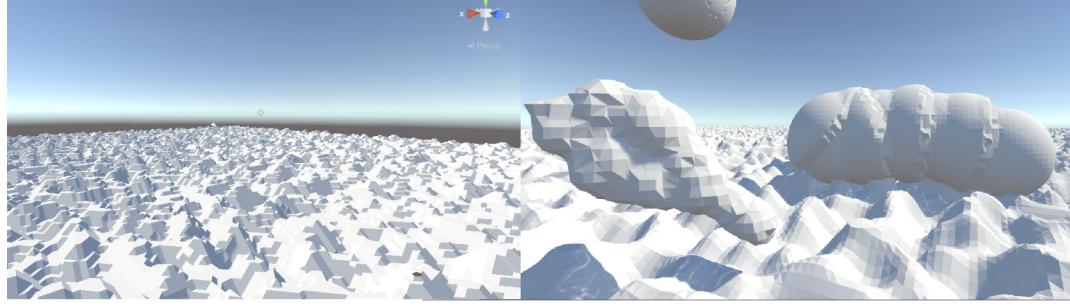


Figure 4: La malla generada mediante Marching Cubes según el uso de interpolación lineal

3. Por último, se crean las facetas dentro de los cubos formadas por los puntos de los ejes por las que la isosuperficie los corta. Se usa una segunda tabla, llamada tabla de triángulos, que contiene un array de números para cada posible combinación de facetas en un cubo. Cada array contiene, para cada triángulo de esa combinación los ejes en los que se encuentran los puntos que forman dicho triángulo. Cada array contiene a lo sumo 5 triángulos, por lo que puede tener una longitud máxima de 15 + 1 números (el último se usa para indicar cuándo acaba el array). Los primeros 3 números del array indican el primero triángulo, los segundos 3 el segundo triángulo, etc. El algoritmo sigue creando facetas de triángulo dado un array hasta que se encuentra con un valor -1. El índice ya obtenido en el primer paso del algoritmo se usa también en esta tabla.

Siguiendo el ejemplo anterior, el índice 8 en la tabla de triángulos apunta al array {3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}.

```

MallaTriangulos <- empty
Para cada cubo:
    índice <- obtenerIndice(cubo.vértices)
    ejes <- tablaEjes[índice]
    Para cada eje en ejes:
        P1 <- eje.P1; V1 <- cubo.valor(P1)
        P2 <- eje.P2; V2 <- cubo.valor(P2)
        eje.puntoCorte <- P1 + (isolevel - V1)(P2 - P1)/(V2 - V1)
    T <- tablaTriángulos[índice]
    Para i en [0..4]
        si T[i*3] == -1
            BREAK
        P1 <- eje[T[i*3]]
        P2 <- eje[T[i*3 + 1]]
        P3 <- eje[T[i*3 + 2]]
        mallaTriangulo.añadirTriangulo({P1, P2, P3})

```

CodeBlock 2: Pseudocódigo del algoritmo de creación del terreno

4.4 Implementación del algoritmo de Marching Cubes

Para la creación de la malla del terreno en el juego, hace falta guardar en una clase el campo escalar que lo define. Tanto el guardado de esta información como la generación de la malla se decidió hacer en una clase llamada *WorldGen*. En el editor de Unity, se añadió el script que contiene la clase al objeto cámara, para que de esta forma siempre hubiera un objeto de tipo *WorldGen* en el runtime de la escena. De esta forma, siempre que se cargue dicha escena, será llamada la función *Start()* del script *WorldGen*. En dicha función se realizan todas las acciones necesarias para la malla del terreno.

En Unity, para crear una malla de forma dinámica, hacen falta dos componentes principales:

1. Un array de todos los vértices entre los que se forman los triángulos de la malla.
2. Un array que define los triángulos de la malla, donde de tres en tres aparecen los índices del array de vértices que forman cada triángulo.

Teniendo estos dos componentes, se crea un objeto *Mesh*, se le asignan los arrays, se calculan las normales y se crea un renderizador para visualizar la malla y un colisionador para que las hormigas puedan interactuar con ella a partir del nuevo objeto *Mesh*.

La obtención de los vectores y triángulos se haría en la función *CreateMeshData()*, donde para cada cubo del campo escalar se obtendrían los triángulos representativos del terreno que pasan por ella. No hace falta que los triángulos del array vengan en ningún orden concreto, por lo que se permitió iterar sobre cada cubo del campo escalar en cualquier orden.

Se decidió identificar cada cubo del campo escalar mediante la coordenada de su esquina de menor valor posicional. En *CreateMeshData()*, se itera sobre todos los puntos del campo escalar menos los últimos en cada dimensión (de 0 a x-1, de 0 a y-1 y de 0 a z-1), ya que los cubos en las penúltimas coordenadas de cada dimensión contienen las últimas coordenadas. Para cada punto, es llamada la función *MarchCube()*. Dada una coordenada de un cubo, esta función realiza los siguientes pasos:

1. Observa los distintos vértices del cubo y, según si son menores o mayores que la isosuperficie, obtiene el índice que representa el tipo de cubo. Si el índice es 0 o 255, es decir, si el cubo está por completo debajo o encima del terreno y la superficie no lo corta, la función termina sin añadir vértices ni triángulos a las listas.
2. Usando el índice del cubo, obtiene el array de los índices de ejes cortados en la tabla *TriangleTable*.
3. Obtiene, para cada índice de eje cortado en dicho array, los índices de las dos esquinas del cubo que lo forman en la tabla *edgeIndexes*.
4. Para cada pareja de esquinas obtenida, se calcula mediante interpolación lineal el punto por el que pasa la superficie en el eje que forman. La interpolación usa los valores del terreno en cada uno de los dos puntos, obteniendo así el punto en el que se encontraría el valor isosuperficie entre ellos.
5. Ahora se tiene para cada triángulo del cubo los 3 puntos que lo forman. Se le añade la posición del cubo y se añaden a la lista de vértices. Después, en la lista de triángulos, se añaden los índices en la lista de vértices de los que se metieron.

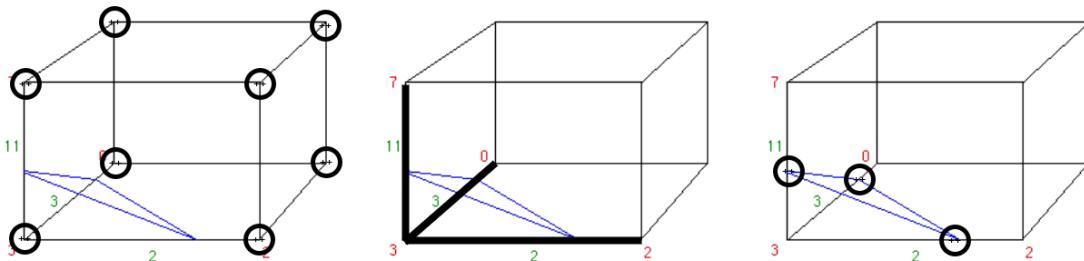


Figure 5: 1. obtención de índice. 2. obtención de ejes cortados. 3. obtención de vértices.

El proceso completo, por tanto, es: *WorldGen* genera o carga el campo escalar tridimensional que representa el mapa. Al iniciarse el juego, itera sobre todos los cubos del campo escalar y obtiene la lista de vértices y triángulos del algoritmo Marching Cubes. Crea la malla del terreno usando estas dos listas, y a partir de ella crea un colisionador y un renderizador. Para decidir el tamaño del mapa, *WorldGen* también cuenta con una serie de variables estáticas que definen las dimensiones del campo escalar.

4.5 Edición del mapa en tiempo real

Una de las grandes ventajas de usar Marching Cubes es la capacidad de editar la malla que genera simplemente cambiando los valores del campo escalar que se representa. Esto permitió implementar un modo de edición de mapas para el jugador, donde a partir de un mapa base sencillo podría crear un escenario interesante para el desarrollo de una colonia de hormigas. Una versión inicial de este modo fue de lo primero que se implementó, en gran parte para poder testar el funcionamiento del algoritmo de los Marching Cubes y situaciones específicas en todo tipo de terrenos.

Editar el campo escalar es tan simple como designar un área y disminuir o incrementar el valor de todos los puntos del campo escalar dentro de ese área. Se decidió usar una esfera como área, ya que requiere cálculos de área simples, y porque edita el terreno de forma más natural. Se creó un objeto esfera básica de Unity y se colocó delante de la cámara. Se implementaron controles para que el jugador pudiera alejar o acercar la esfera a la cámara, incrementar o disminuir su tamaño y aplicar un cambio sobre el campo escalar, ya sea incrementando o disminuyendo el valor de los puntos que se encontraran dentro de ella.

Se consideró usar colisiones de alguna forma para determinar si un punto estuviera dentro de la esfera, pero resultó ser más simple considerar el origen de la esfera y su radio para encontrar las coordenadas enteras.

La edición misma de los puntos del campo escalar se hizo mediante la función *TerrainEditSphere()*, que ejecuta los siguientes pasos:

1. Dado el origen de la esfera y su radio, itera sobre todos los puntos del campo escalar que se encuentran dentro del cubo que contiene la esfera.
2. Para cada punto que se encuentra dentro de la distancia radio del centro de la esfera, se añade a una lista de puntos junto a un valor asignado. Este valor es mayor cuanto más cerca del centro de la esfera. Si la función se invocó usando el clic derecho, estos valores serán negativos, es decir, se eliminará terreno. Si se invocó usando el clic izquierdo, los valores son positivos y se expandirá el terreno.
3. La lista de posiciones y sus valores se pasa a la función *EditTerrain()* de la clase *WorldGen*. Esta función añade los valores a las posiciones en el campo escalar, se asegura de que después de añadirlos ningún valor se encuentre por debajo del mínimo ni por encima del máximo valor permitido, y llama la función de generación de la malla para actualizar el terreno.

El jugador podría mantener pulsados los botones de clic izquierdo o derecho para editar el terreno, y en cada *FixedUpdate()* la función *TerrainEditSphere()* es llamada, aplicando gradualmente el cambio sobre el mapa.

4.6 Chunks

4.6.1 Problema de rendimiento del mapa completo

Después de la implementación de la edición del terreno en tiempo real, durante el testeo de edición de mapas, pruebas con cualquier mapa que no fuera pequeño mostraban problemas de rendimiento. Esto se debía a que en cada *FixedUpdate()* en la que se modifica el terreno, había que recargar la malla entera, iterando sobre la totalidad del campo escalar a pesar de modificar solo una pequeña porción de ella. Este proceso podía tardar hasta 4 o 5 segundos según el tamaño del mapa. Renderizar una malla enorme de por sí también causaba bajas de rendimiento, al no poder bajar la calidad o esconder las partes más lejanas de una misma malla.

La solución a estos problemas fue la implementación de chunks: secciones de mapa de tamaño equivalente en los que se dividiría el campo escalar, y que se encargarían de generar sus propias secciones de malla. De esta forma, al editar el mapa, se generarían de nuevo solo los chunks que contuvieran la zona modificada. También existiría la opción de implementar poder bajar la calidad o simplemente no mostrar chunks lejanos, en caso de querer mostrar mapas enormes.

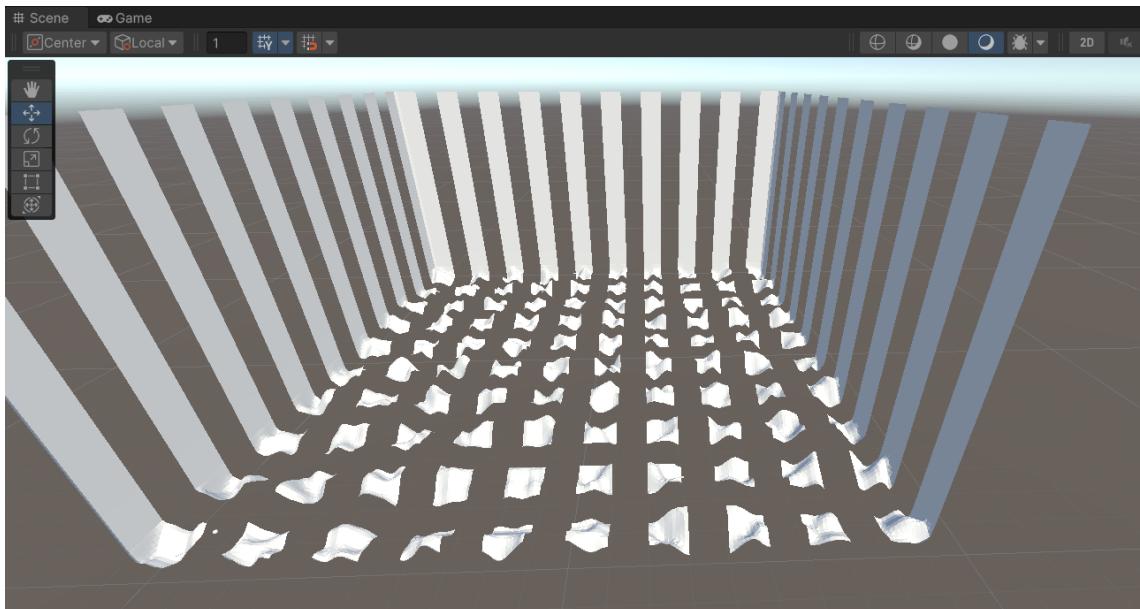


Figure 6: Los distintos chunks que componen un mapa separados para su visualización

El uso de chunks es una técnica de ahorro de memoria gráfica y computación popular en videojuegos con mapas extensos, pero el juego que más inspiró su uso en este proyecto fue el popular videojuego Minecraft. En él, el mundo se divide en una infinita expansión de chunks de tamaño 16x384x16, de los cuales solo se cargan los que se encuentren en un cierto radio alrededor del jugador. Su extensa altura se debe a que, mientras que horizontalmente el mundo se divide en chunks, cada chunk expande la altura completa del mundo. En el caso de este proyecto, el tamaño de chunks fue variable durante su desarrollo, hasta acabar estableciéndose como 10x50x10.

4.6.2 Implementación de chunks

Para implementar y poner en funcionamiento los chunks en el juego, se siguieron los siguientes pasos:

1. Se creó una nueva clase llamada *Chunk*. *WorldGen* generaría los objetos *Chunk* necesarios para representar el mapa según su tamaño la ejecución del juego. Se le implementaron los siguientes componentes:
 1. Se movieron todas las tablas de triángulos, ejes y vértices para el funcionamiento del algoritmo Marching Cubes de *WorldGen* a *Chunk*. Se modificaron para ser estáticos, y así tener tan solo una copia en memoria, independientemente de la cantidad de chunks activos.
 2. Las funciones de generación de malla e implementación de Marching Cube, previamente en *WorldGen*.
 3. Las dimensiones y posición del objeto *Chunk*, que le permiten identificar a qué sección del campo escalar corresponde.
 4. Los componentes necesarios para guardar y gestionar las colisiones y visibi-

lidad de una malla.

2. *WorldGen* obtendría un diccionario de *Chunks*, indexados por su posición como *Vector3Int*, para poder guardar y gestionar los objetos *Chunk* generados durante la ejecución del juego.
3. En la función *editTerrain()*, se implementó la habilidad de solo actualizar las mallas de los chunks que contuvieran puntos editados. Para ello, se usó un *HashSet* de *Vector3Int*, siendo las posiciones (e identificadores) de los chunks. Para cada punto del campo escalar obtenido en la lista de puntos a cambiar, se añadiría el *chunk* que lo contuviera al *HashSet*. Luego, para cada identificador en el *hashSet*, *WorldGen* llamaría la función de limpieza y generación de malla del *chunk* respectivo.

Había que tener en cuenta que puntos del campo escalar en el eje de un chunk también afectaban al siguiente chunk, ya que un punto del campo escalar afecta a los 8 cubos que lo contienen. Sin ese check, había inconsistencias en las interconexiones de *Chunks* (Figura 7).

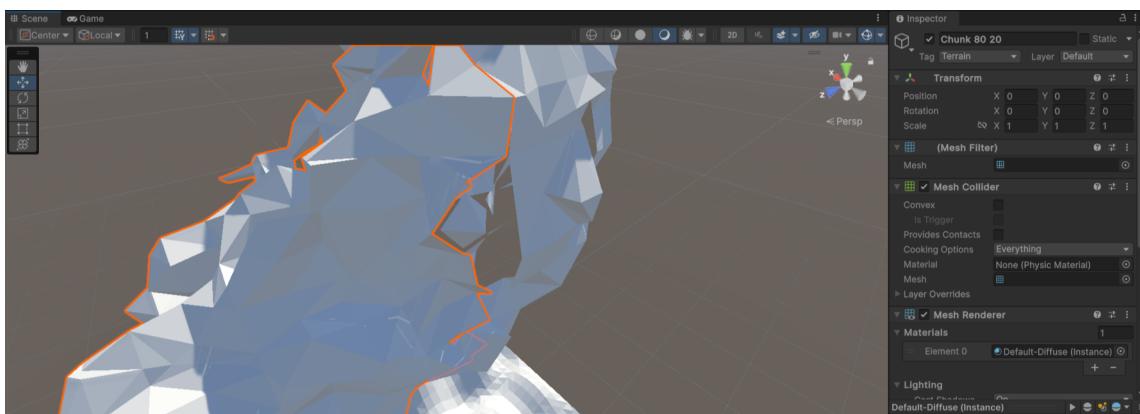


Figure 7: inconsistencia en la conexión entre chunks

El resultado de la implementación de chunks fue inmediato: se podía editar partes del terreno de mapas grandes sin ninguna repercusión en la fluidez del juego. Los chunks también estaban bien interconectados, y no se podía ver ninguna separación. Las hormigas podían navegar entre los chunks como si fueran una única malla.

5 Desarrollo gráfico del juego

5.1 Concepto estilístico general: low-poly

Una parte importante de la presentación visual de un videojuego es su cohesión artística. Elementos visuales que se difieren demasiado del aspecto general pueden ser disonantes y arruinar la estética del juego. Los videojuegos más famosos comparten esta cohesión visual de un estilo determinado.

Para este demo, se decidió usar una estética gráfica conocida como low-poly. Se define por el uso de modelos tridimensionales de pocos polígonos, simulando el estilo retro que se usaba en sistemas antiguos para mejorar rendimiento. Hoy en día se usa mucho en juegos indie como una decisión estética a posta. Adopta el concepto de ser único y diferente de la realidad, oponiéndose a juegos que se centran en tener los gráficos más fotorealistas posibles.

Se eligió este estilo por tres motivos. El primero fue que funciona bien con el algoritmo de Marching Cubes, ya que este crea superficies inherentemente poligonales. El segun-

do motivo fue la facilidad de trabajar en el estilo low-poly: no requiere de modelos, texturas y animaciones detalladas y realistas. Por último, el estilo low-poly es uno que siempre me ha gustado y, como desarrollador del proyecto, quise implementarlo.

Las dos formas principales mediante las cuales se consiguió mantener un estilo low poly entre varios modelos fueron gracias al uso de flat shading en vez de smooth shading (figura 8) y simples colores en vez de texturas complejas.

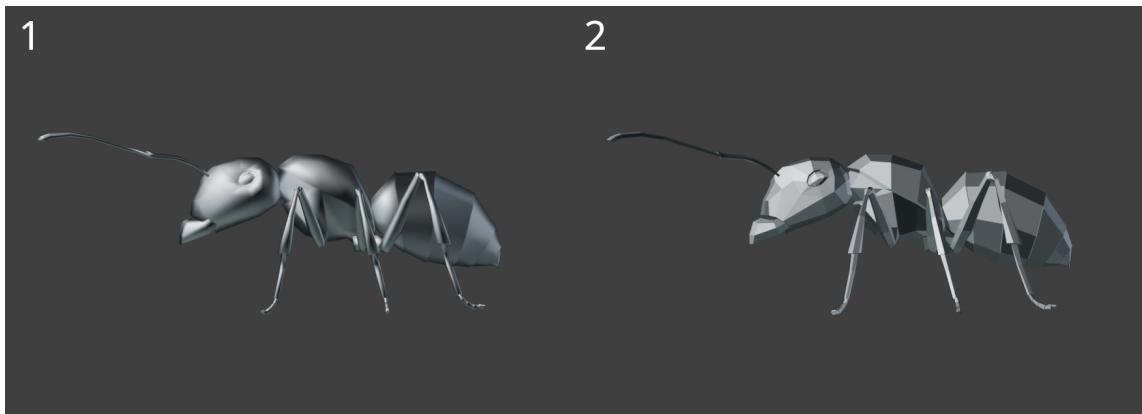


Figure 8: Vista de lado de la hormiga renderizada con Smooth Shading (1) y Flat Shading (2)

5.2 Obtención y creación de modelos

Hubo que obtener varios modelos tridimensionales para representar los elementos más importantes del juego: las hormigas y las fuentes de comida.

Los modelos tridimensionales más complejos usados en este proyecto fueron obtenidos en línea, ya que la creación de modelos en 3D es un arte que no se vio asequible aprender, encima de todos los demás aspectos del juego que hubo que desarrollar. Algunas de las excepciones a esta regla fueron la creación del modelo de la hormiga reina, aunque fue editando el de la hormiga trabajadora; la creación de la base de la maízca de maíz y la creación del modelo de huevo de hormiga.

5.2.1 Hormiga trabajadora

El modelo base de la hormiga se obtuvo gratis en la página web Sketchfab, una plataforma en línea en la que se publican, comparten y venden modelos 3d. Desde la obtención del modelo, el creador ha eliminado la página de esta, por lo que es imposible acreditar al creador original hasta que consiga encontrarlo en alguna otra página web.

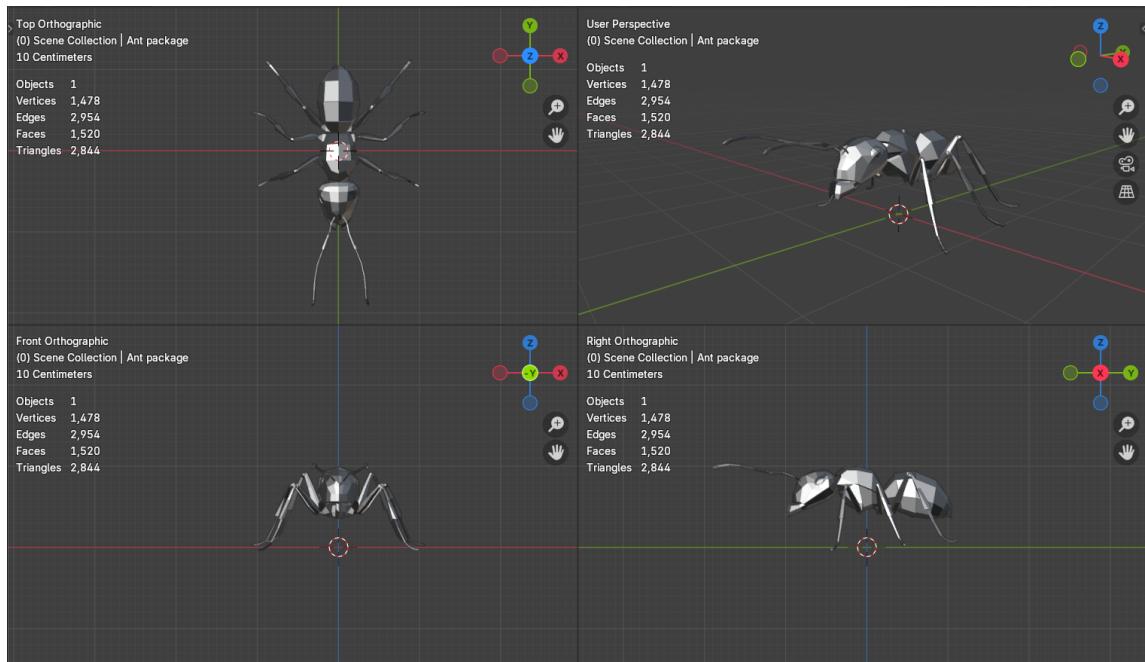


Figure 9: Quadview del modelo de la hormiga trabajadora

5.2.2 Hormiga reina

La hormiga reina se creó partiendo de la base de la hormiga trabajadora. Emulando la apariencia media de reinas de las distintas especies de hormigas, se modificó la forma de la malla del modelo. El tórax y abdomen se expandieron. Especialmente el abdomen, ya que la hormiga reina se encarga de dar luz a todas las hormigas del nido. Gracias al tórax más amplio, combinado con el abdomen, resaltó la corpulencia de la reina hormiga.

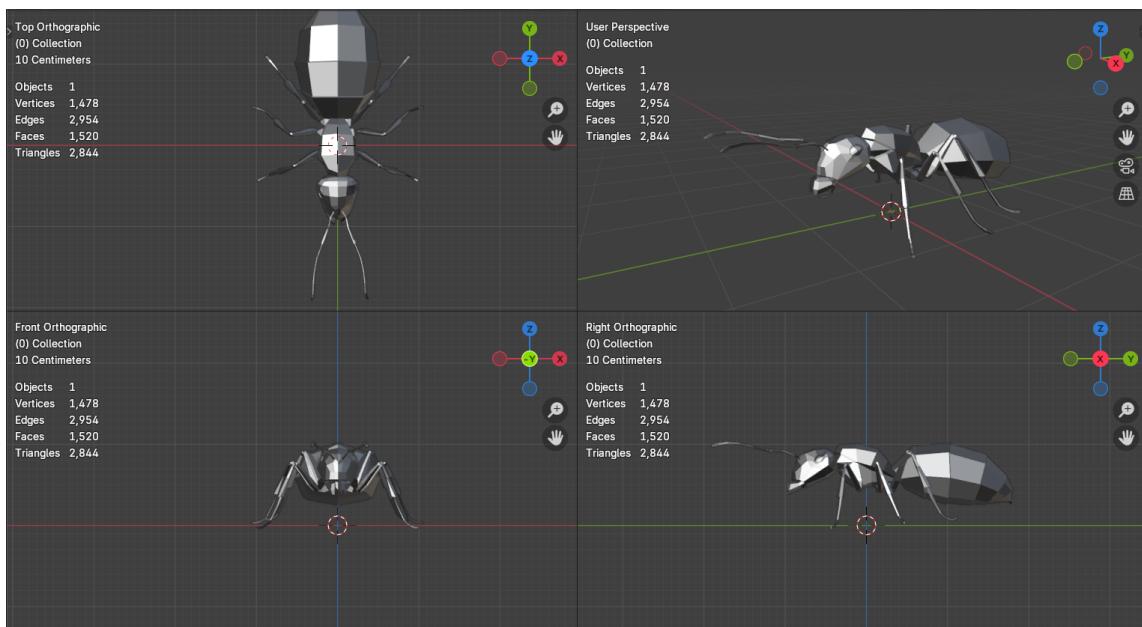


Figure 10: Quadview del modelo de la hormiga reina

5.2.3 Maíz

La pepita de maíz, al igual que la hormiga, fue obtenida en Sketchfab. La mazorca fue creada en Blender, con el objetivo de tener una fuente de comida semipermanente para poder mostrar el sistema de feromonas de las hormigas: en cuanto una hormiga encontrase la mazorca, marcaría un camino hacia ella, y el resto de las hormigas reconocerían y seguirían dicho camino para recolectar más pepitas de maíz de la mazorca has-

ta agotarla.

Hubo que poder representar la mazorca perdiendo pepitas de maíz una a una al ser recolectadas por las hormigas. Había dos opciones:

- Hacer las pepitas parte del modelo y crear animaciones individuales para hacerlas invisibles una a una. La ventaja sería el uso de un único objeto en el editor de Unity, simplificando su uso. Las desventajas de esto eran tener que crear una alta cantidad de animaciones para desaparecer cada una de las pepitas, y que siempre tendrían que desaparecer en el mismo orden.
- Hacer que cada pepita fuera un objeto independiente, con una posición y orientación estática relativa a la mazorca hasta que alguna hormiga lo recogiera. Se podría gestionar el orden en que las pepitas desconectaran de la mazorca.

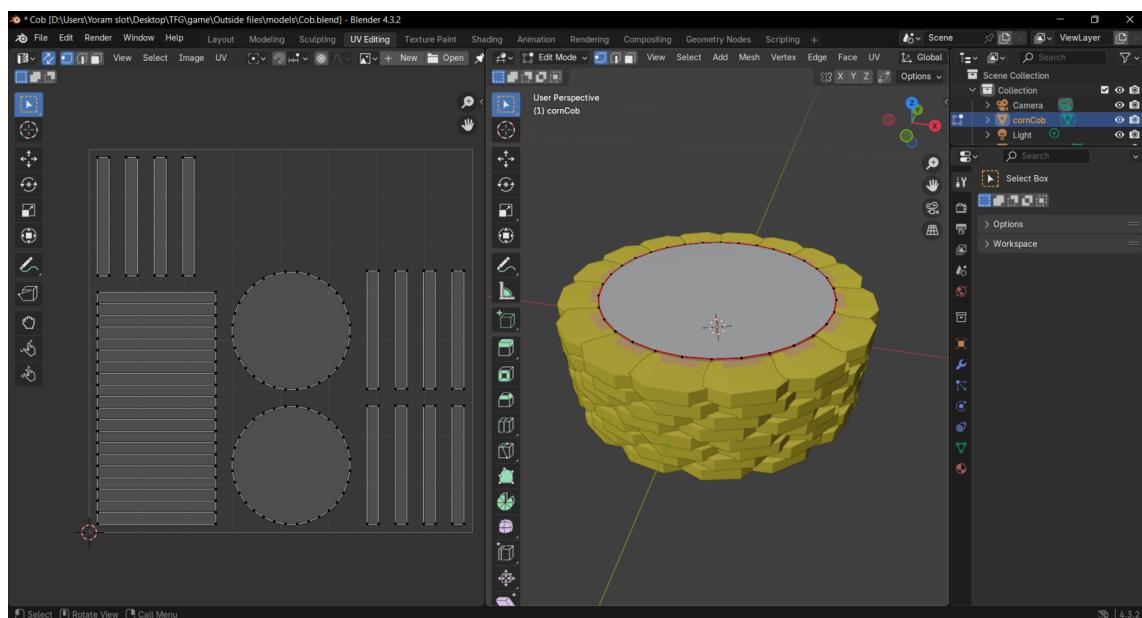


Figure 11: La mazorca en blender con las pepitas incrustadas y a su izquierda su mapa de texturas

Se decidió usar la segunda opción. Para ello, hubo que obtener las posiciones de cada una de las pepitas relativo al objeto mazorca, para poder colocarlas respecto a la mazorca dentro del juego. Para tanto la creación del modelo y la obtención de estas posiciones se siguieron los siguientes pasos:

1. Se creó el objeto en Blender a base de un cilindro simple. Se hizo más ancho que alto para representar una sección de una mazorca, no una entera.
2. Se añadió la pepita de maíz a la escena. A base de duplicar y rotarlo, se consiguió cubrir el exterior de la mazorca con 159 pepitas.
3. Se escribió un script de Python usable en Blender que iteraría sobre cada objeto seleccionado, notando su posición y orientación. Se usó teniendo seleccionado todas las pepitas de la escena, y se guardaron sus datos en un fichero aparte para uso posterior (figura 12).
4. Se obtuvo el mapa de texturas de la mazorca. Se exportó y fue usado para crear una textura para la mazorca: Un color general para los lados de la mazorca y una esfera en las bisecciones para representar el núcleo de esta.

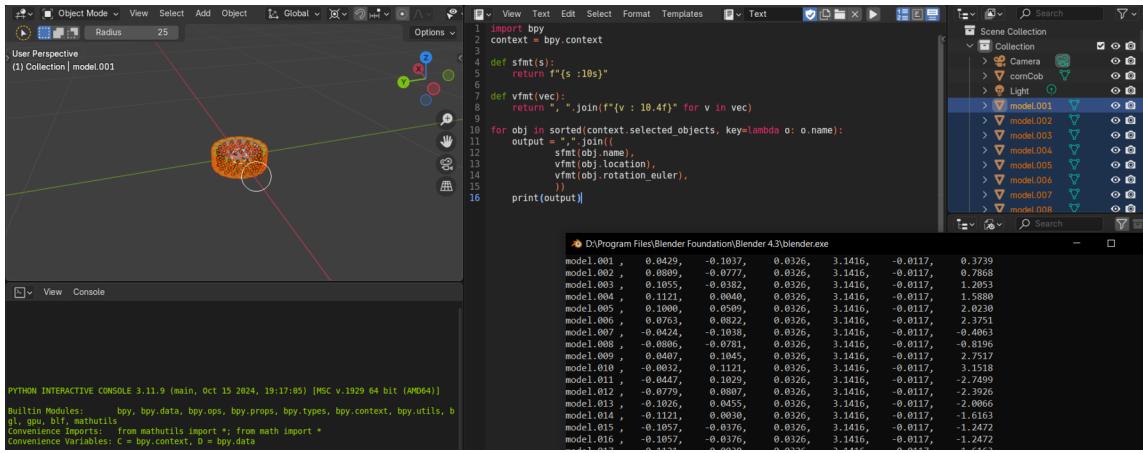


Figure 12: Obtención de las posiciones de las pepitas mediante un script de python

Se implementó el modelo en Unity asignándole el script nuevo *CornCob*. Al instanciar un objeto *CornCob*, este instanciaría los objetos pepita en las posiciones obtenidas en Blender. Sin embargo, al ser hijos del objeto mazorca, su transformación se aplicaba a las pepitas, modificando erróneamente su tamaño y posición. Se decidió, por tanto, asociar el script *CornCob* a un objeto vacío y tener al objeto mazorca y los objetos pepita debajo de esta para poder modificar sus escalas independientemente.

La clase *CornCob* contenía todas las posiciones y orientaciones de las pepitas en un array, además de los ratios de transformación de estos valores necesarios para ajustarlos al entorno de Unity. Se le añadió un diccionario, donde los identificadores de las pepitas en la mazorca se indexaron por sus posiciones en esta, para poder gestionar qué pepitas se encontraban en qué posición y recordar que se encontraban en la mazorca.

5.2.4 Huevo

No se pudo encontrar un modelo 3D usable como huevo de hormiga que cumpliese con la estética low-poly del juego; los únicos resultados eran modelos realistas. Por tanto, se decidió crear el modelo de hormiga desde 0.

Para ello se siguieron los siguientes pasos (figura 13):

1. Se partió de un objeto cubo primitivo en Blender.
2. Se alargó horizontalmente.
3. Usando el modificador Subdividir Superficie a nivel 3, se convirtió en un elipsoide. Este modificador divide las caras de una malla en caras más pequeñas para alisar los modelos. El nivel 3 es el máximo en Blender, y convierte incluso cubos en esferas.
4. Se movieron los vértices centrales del modelo hacia arriba, para que tomase una forma más orgánica y parecida a los huevos de algunas especies de hormigas.
5. Se movieron hacia arriba un poco los vértices inferiores del modelo.
6. Usando alisado laplaciano y ajustando un poco los ejes inferiores del huevo, se le dio una forma más natural.

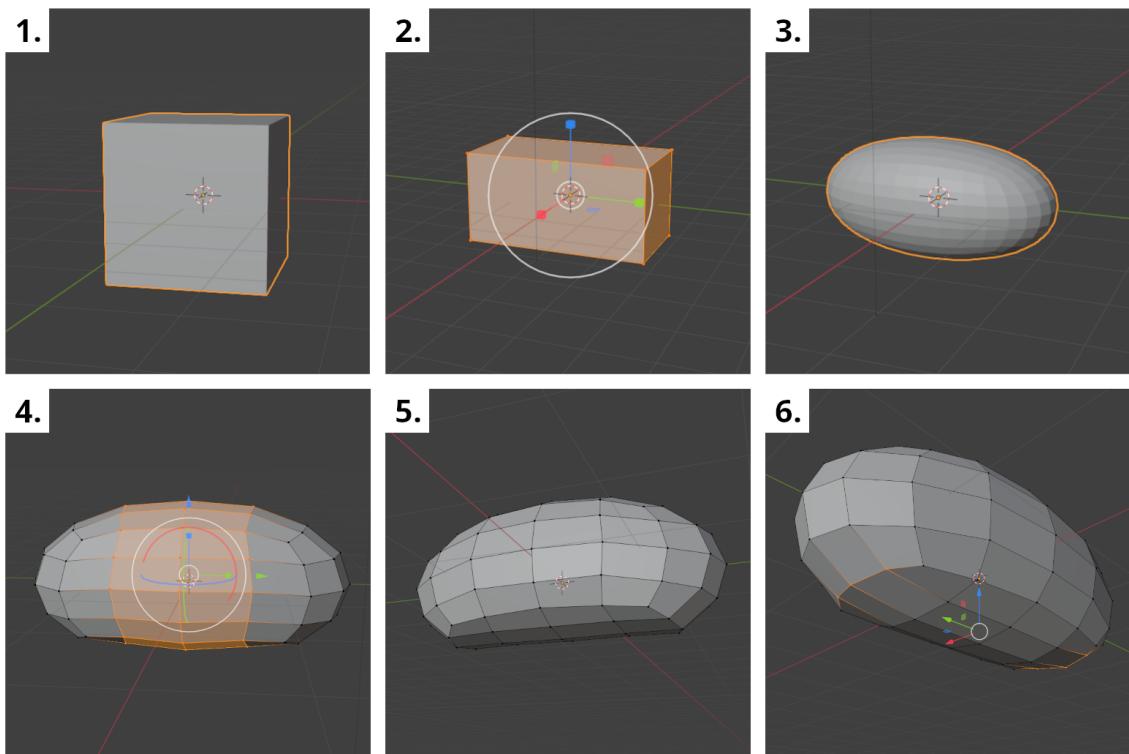


Figure 13: La creación del huevo dividido en pasos

5.3 Rigging

En animación 3D, para poder mover una malla, hace falta darle una estructura virtual que funcione como su esqueleto. A cada área de la malla se le asigna una parte del esqueleto, para que se muevan juntos, en un proceso llamado *skinning*. Estos dos pasos se denominan rigging, y permiten la creación de animaciones.

5.3.1 Creación del esqueleto

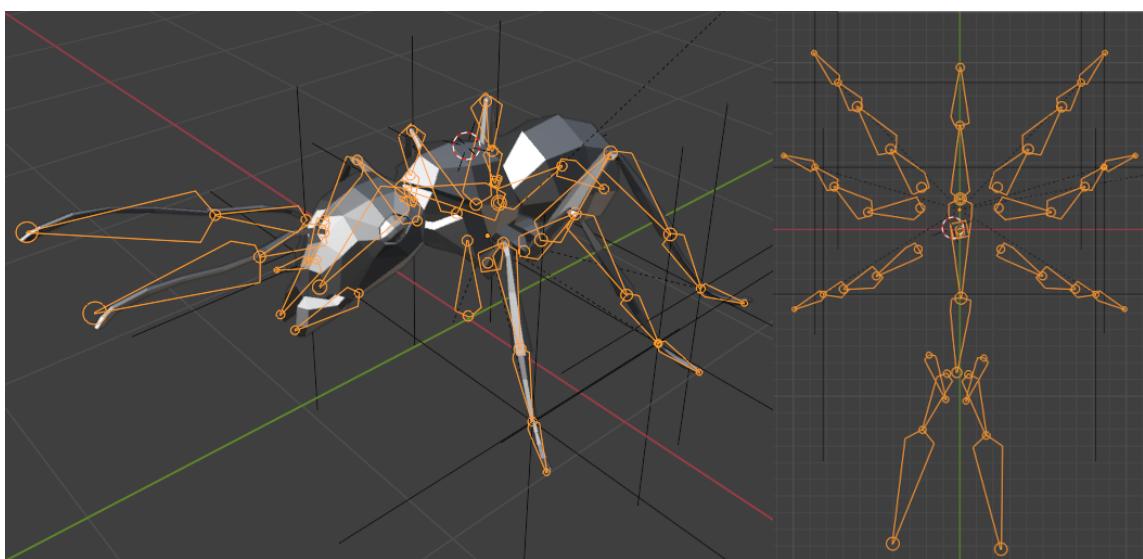


Figure 14: izq. el esqueleto superpuesto con el modelo. der. el esqueleto visto desde arriba.

El modelo de la hormiga obtenido en *SketchFab* no contenía un esqueleto, por lo que hubo que construir uno en *Blender*. La parte principal del cuerpo de la hormiga está formada por el tórax, la cabeza y el abdomen. La cabeza y el abdomen son rígidos, por lo que se correspondieron con un hueso cada uno. El abdomen requirió 2 huesos, ya que es una parte flexible. Luego un hueso por cada segmento de las patas, un hueso para cada mandíbula y 2 huesos para cada antena.

El esqueleto es simétrico, lo cual simplificó varios aspectos de la animación, ya que se podría animar solo una mitad del cuerpo y tenerlo reflejado en la otra mitad. Para habilitarlo, tuvieron que ser nombrados de forma específica los pares de huesos: los de la parte izquierda y derecha tienen el mismo nombre, solo diferenciados por una L en los de la izquierda y una R en los de la derecha. Por ejemplo: *Leg_1_2_L* y *Leg_1_2_R*.

5.3.2 Skinning

El proceso de ajustar las distintas partes del modelo a los huesos se realizó mediante "weighted painting". Consiste en usar un pincel para destacar aquellas superficies del modelo que quieras que se muevan con el hueso. Se seleccionan hueso a hueso, y para cada uno ilustras las partes a las que corresponden. Como la mayoría de las partes de la hormiga son exoesqueleto no elástico, tan solo hizo falta seleccionar los vértices relevantes al hueso, poner su valor de peso a 1 y hacer selección inversa para poner todos los demás vértices del modelo a 0. Las únicas excepciones a esta regla fueron los dos

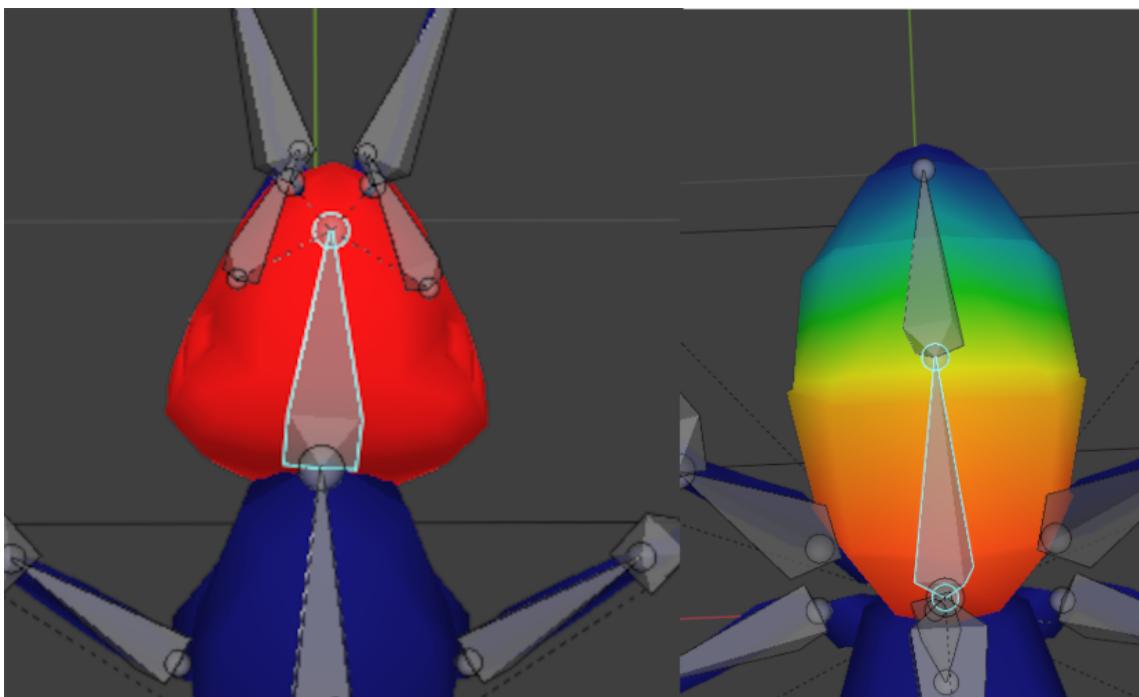


Figure 15: El modo weighted painting. A la izquierda, toda la cabeza se mueve con el hueso. A la derecha, la influencia del hueso sobre el abdomen degrada cuanto más lejos del esqueleto se encuentra el área.

huesos de abdomen y los cuatro huesos de las antenas, en los cuales se usó un degradado. Esto se hizo para simular su elasticidad.

5.4 Creación de animaciones

Blender proporciona un editor de animaciones útil para poner en movimiento inmediatamente los modelos que se obtuvieron y crearon. Con la ayuda de cursos en línea gratuitos, fue posible aprender a usarlo lo suficientemente bien como para crear las animaciones necesarias para el juego. La primera de esta, y la más importante, fue el ciclo de caminar de la hormiga.

5.4.1 Inverse kinematics

Animar cualquier movimiento involucrando las patas del modelo de la hormiga resultaba requerir una gran cantidad de trabajo, dado que hay 6 patas con 4 huesos cada. Para simplificar el proceso, se usaron inverse kinematics. Las kinemáticas inversas son una

forma de decidir cómo mover articulaciones de modelos sin tener que modificar individualmente las posiciones y orientaciones de cada uno de sus huesos. Usando un constraint al final de una articulación, mover este constraint hace que el sistema calcule las posiciones y orientaciones necesarias para que la articulación lo siga. Para el modelo de la hormiga se usaron 6 Constraints, uno para cada pata, en sus extremidades donde se encontrarían las patas con el suelo.

5.4.2 Walk cycle

Blender tiene 3 formas de animar mediante keyframes: Timeline, graph editor y dope sheet. Todos tienen sus ventajas e inconvenientes, pero para la animación de caminar vino mejor el graph editor. Este permite modificar el movimiento de huesos y objetos mediante grafos, facilitando la creación de movimientos más naturales. En ella simplemente hubo que mover los constraints de las patas de una forma repetitiva, arrastrándolos hacia atrás para simular el avance de la hormiga y moviéndolas hacia delante en un arco para simular un paso. Las hormigas mueven sus patas de 3 en 3, lo que simplificó también el proceso de animar la forma de caminar al poder mover los constraints en tandem.

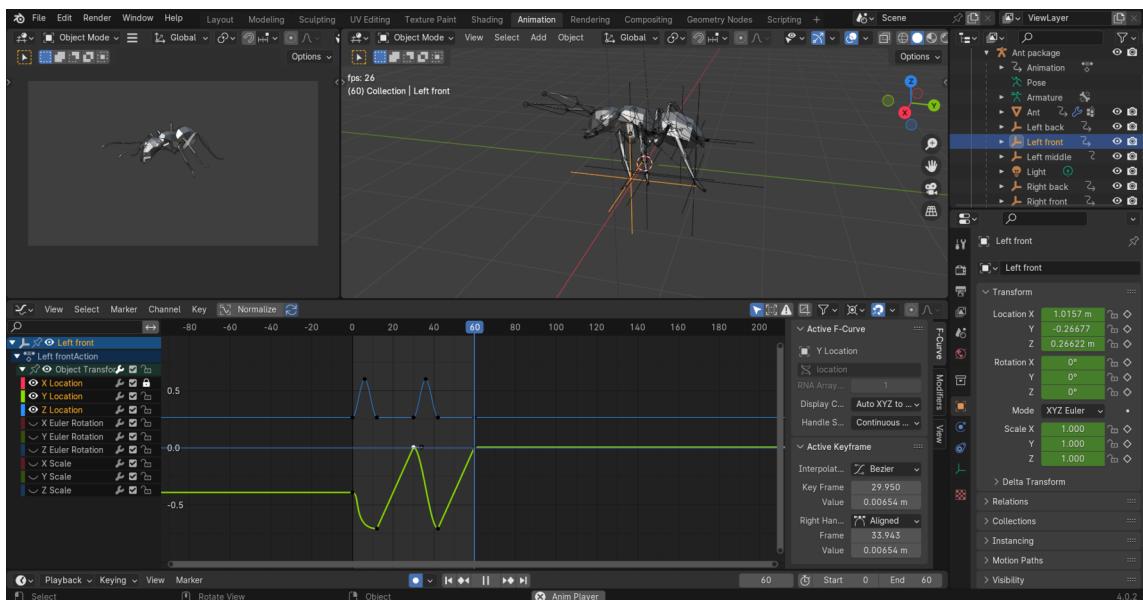


Figure 16: El timeline de animación de uno de los constraints.

Las patas de la hormiga se movieron bien, pero el tórax era demasiado influenciado por estos movimientos. Para evitar esto, se editó el "stiffness" del tórax para limitar el efecto que tienen los demás huesos sobre él. También se limitaron los constraints para que influyeran solo a 3 huesos de distancia, y así no afectar partes del esqueleto más allá de las patas.

Ahorró mucho trabajo el uso de los constraints, pero aun así faltó retocar cosas del modelo. Hubo que editar rotaciones heredadas y movimientos de otros huesos para evadir ángulos extraños y sobrenaturales. Por ejemplo, los últimos huesos de las patas siempre mantendrían el mismo ángulo global, y el tórax se movería demasiado con las patas.

Tras resolver los problemas, la animación funcionó bien, pero la hormiga no movería la cabeza. Las hormigas buscan con sus antenas los caminos de feromonas que ellas y otras hormigas marcan mientras caminan, y se quiso añadir esto a la animación. La forma más sencilla fue creando una animación aparte para el movimiento de la cabeza y añadiéndola a la de caminar. Para ello se usó el menú de animación no lineal, el cual

permite combinar acciones distintas en animaciones individuales. Se creó la acción de búsqueda con las antenas y luego se combinó con la de caminar para crear una única animación en la que la hormiga camina mientras busca con sus antenas.

5.4.3 Idle animation

Para que las hormigas no se queden inmóviles mientras no se mueven por el mapa, hace falta tener una animación "idle". Considerando que las hormigas siempre están sin-

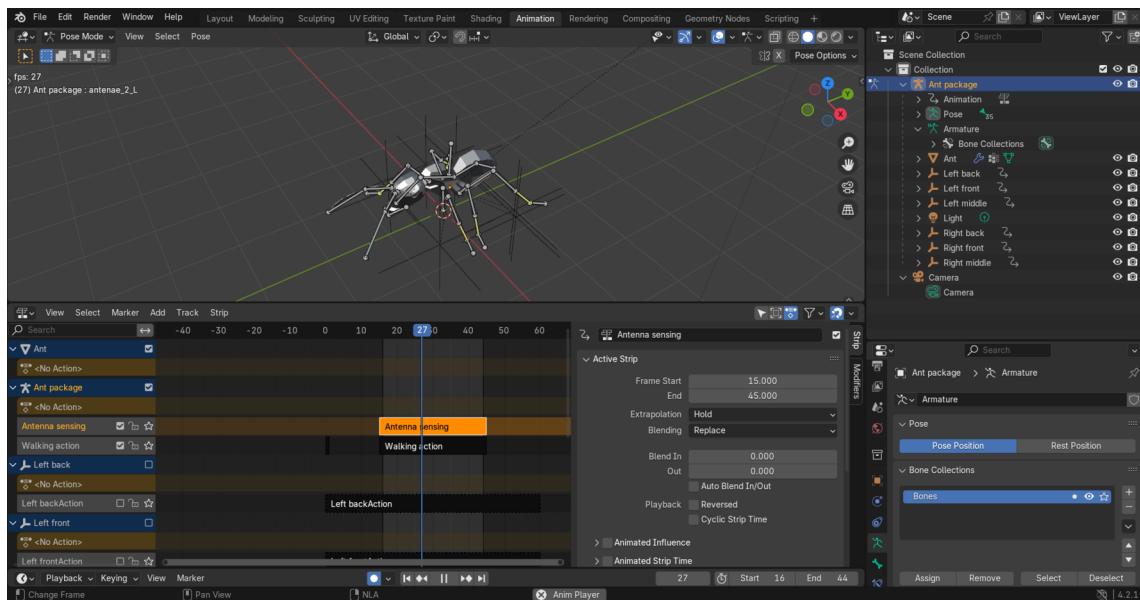


Figure 17: Las acciones Antenna sensing y Walking action se juntan en el menú de animación no lineal.

tiendo el terreno mediante sus antenas, se creó una animación de 70 frames en la que la hormiga mueve la cabeza de derecha a izquierda mientras siente el terreno con sus antenas. En Unity, esta animación se muestra mientras la hormiga está inmóvil.

5.4.4 Girar

Solo hizo falta la animación de girar mientras la hormiga está inmóvil, ya que para cuando camina hacia delante y se gira a la vez, la animación estándar de caminar hacia delante es suficiente.

La animación consistió en la hormiga girando 24 grados en 24 frames para ayudar a simplificar el proceso de animación. En Unity, el objeto hormiga gira dentro del motor 3D, por lo que en la animación misma el cuerpo principal de la hormiga siempre mirará hacia delante, mientras que las patas se mueven como si se girara la hormiga.

Para conseguir esto de forma simple, se siguieron los siguientes pasos:

Se registró la pose por defecto de la hormiga como el primer frame de animación.

Se giraron los 6 constraints de las patas 24 grados alrededor del centro de la hormiga, y se registró como el último frame de la animación. De esta forma, durante los 24 frames, los constraints se moverían alrededor de la hormiga en un arco.

Para cada pata se creó un paso de 24 grados en la rotación opuesta, de tres en tres. Los primeros tres dan el paso en la primera mitad, los otros durante la segunda mitad. Ahora las patas acabarán en el mismo sitio donde empezaban, pero se moverían como si el suelo debajo de la hormiga hubiese rotado.

Para vender más la sensación de movimiento, se inclinaron el tórax y la cabeza de la hormiga en la dirección de rotación. De esta forma, la hormiga miraría hacia donde girara.

Para conseguir la animación de giro en la otra dirección, se copió y pegó la original en modo espejo.

5.4.5 Interact

Esta animación originalmente se llamó *picking up*, ya que consistió en que la hormiga recogiera algo del suelo delante de ella. Sin embargo, durante la producción se decidió también usar la primera sección de la animación para la acción de excavar de la hormiga y comer de la reina.

La animación consiste en la hormiga agacharse para recoger un objeto después de arquearse hacia atrás para observarlo con sus antenas, y tiene una duración de 34 frames. El objeto que se recoge no forma parte de la animación, sino que en el juego se encontrará un objeto en esa posición que la hormiga podrá levantar.

La primera mitad de la animación, el inspeccionar el objeto con las antenas, le da una sensación tanto de curiosidad como de precaución a la hormiga. La animación acaba con la hormiga en la pose inicial por defecto, excepto que su cabeza está inclinada hacia arriba, sus antenas no apuntan directamente hacia delante y sus mandíbulas se encuentran abiertas. Esto se hace para simular que ahora la hormiga lleva en sus mandíbulas un objeto.

5.4.6 Carrying

La animación de llevar un objeto encima mientras la hormiga permanece inmóvil se creó a partir de la pose por defecto de la hormiga. La cabeza se levantó con las mandíbulas abiertas mientras que las antenas, dejando espacio para el objeto que la hormiga llevará en las mandíbulas, suben y bajan con un ritmo tranquilo. Estos cambios se combinan con otras animaciones usando NLA (Non Linear Animation) strips para crear varias versiones de la hormiga llevando un objeto mientras se mueve. Las creadas son las siguientes:

- Caminar con un objeto.
- Girar hacia la izquierda con un objeto.
- Girar hacia la derecha con un objeto.
- Idle con un objeto.

5.4.7 Putting down

La animación de dejar en el suelo un objeto simplemente consiste en la hormiga agachándose hasta llegar con la cabeza al suelo y abriendo las mandíbulas para soltar el objeto. Luego vuelve a retomar su pose por defecto, cerrando sus mandíbulas, poniendo rectas sus antenas e incorporándose.

5.4.8 Attack

Esta fue una de las animaciones que no se llegó a usar en la versión final de la demo del TFG, pero se guardó para uso futuro en el caso de seguir desarrollando el juego. Fue creado para peleas entre hormigas de distintos nidos, y muestra la hormiga mordiendo con fuerza hacia delante.

La animación del ataque hace uso de dos principios de animación: Buildup y overshoot. La hormiga se prepara para dar un mordisco moviendo primero su cuerpo hacia atrás, antes de disparar hacia delante (buildup). Luego, en la cúspide de la distancia que avanza el cuerpo de la hormiga, al final de su trayecto, tiene dos frames en los que llega más lejos de lo que debería parecer posible, antes de volver a la distancia esperada. Estos dos frames de overshoot son difíciles de ver, pero hacen que se sienta más la velocidad del ataque. La hormiga abre las mandíbulas al principio del movimiento hacia delante y las cierra rápidamente al llegar delante para dar más impacto.

5.4.9 Getting hit

Para la animación de ser golpeado fue importante crear un impacto. Para ello, la hormiga retrocede hacia atrás, mueve su cabeza hacia arriba, contorsiona sus antenas y desdobra su abdomen hasta conseguir una pose anormal. Esto ocurre de forma muy rápida, en 3 frames. La hormiga se mantiene en la pose durante 6 frames, para simular lo que en videojuegos se denomina hitstun: un periodo que ocurre tras ser golpeado, en el que el luchador muestra una postura dolorosa durante unos frames para aumentar el impacto del golpe. Después de este periodo, la hormiga lentamente se incorpora y retoma su pose por defecto.

Esta fue otra de las animaciones no incorporadas en la versión actual de la demo.

5.4.10 Dying

Para la animación de muerte se crearon dos versiones: una "estática" y otra "dinámica". La estática muestra a la hormiga, después de haber sido herida, doblar las patas y derrumbarse al suelo hasta quedarse boca arriba. Esta animación cuenta con el hecho de que se encuentra en el suelo, y por tanto la gravedad atrae la hormiga hacia abajo. Sin embargo, considerando que las hormigas en el juego podían escalar paredes y techos, se creó una segunda versión: la dinámica. La animación dinámica no mueve el tórax de la hormiga, sino que cuenta con el juego quitando el rigidbody de la hormiga en cuanto se muera para que el cuerpo pueda caer hacia abajo. Consiste en la hormiga contrayendo las patas y acabando en una pose compacta e inmóvil. Esta pose final es útil, ya que las hormigas muertas pueden ser llevadas por otras hormigas enemigas a sus nidos para ser consumidas.

5.4.11 Dead

Cuando la hormiga se encuentra muerta, en vez de no moverse, se creó una animación en la que las patas de la hormiga muestran contracciones repentinas. Esto simula el fenómeno real que ocurre en los insectos y da más personalidad al juego. Al igual que con la animación de morir, existen dos versiones. La inicial estática en la que la hormiga se encuentra bocabajo con las patas dando espasmos hacia arriba, y la segunda dinámica en la que el tórax del modelo no se ha movido de su posición inicial.

5.4.12 Hatch

Esta fue la animación más difícil de crear con diferencia. Se trata de una hormiga saliendo de un huevo. En la vida real, las hormigas salen de sus huevos en forma larval, y luego pasan por una fase de pupa y acaban como hormigas adultas. En el caso del juego, se sacrificó cierto grado de realidad para simplificar el proceso para la demo.

Primero se creó una pose inicial fetal de la hormiga para cuando se encontrara dentro del huevo. Desde esta, se creó una animación de "nacimiento" en la que la hormiga se

incorpora y acaba en su pose por defecto.

Después hubo que hacer que el huevo se rompiera alrededor de la hormiga. Se consideró editar el movimiento de cada sección del huevo manualmente, pero esto sería muchísimo trabajo. Se decidió usar físicas simuladas para crear las animaciones. Los pasos que se siguieron para conseguir esto, ignorando muchos fallos y pasos deshechos, fueron:

Se juntaron el modelo de la hormiga en su pose fetal y el modelo del huevo de tal forma que la hormiga se encontrara dentro del huevo.

Usando el modificador de superficie de Blender para convertirlo en un objeto hueco. El modificador toma la superficie de la forma y le añade grosor. En vez de un objeto relleno, el huevo pasó a ser una cáscara conteniendo la hormiga.

Usando la extensión “cell fracture” de Blender, se fracturó el modelo del huevo, descomponiéndolo en secciones pequeñas. Luego se quitaron todas las partes pequeñas irrelevantes separadas de la superficie del huevo, ya que eliminarlas no abría el huevo.

Se le añadió un RigidBody pasivo a la hormiga. Esto significa que interactuaría con otros RigidBody, pero al ser pasivo, no se movería él mismo por impactos de otros objetos.

Se registraron las posiciones iniciales de todas las secciones del huevo en la animación hatch. Según cómo se movía la hormiga durante su animación de nacer, se les asignaban RigidBodies activos a las secciones del huevo el frame antes de impactar con la hormiga. Esto se hizo en el frame de antes porque cuando se le asigna un RigidBody a una pieza, esta cae inmediatamente hacia abajo el siguiente frame, siendo afectada por la gravedad. Dándole el RigidBody justo antes del impacto, las hormigas son lanzadas por este en vez de caer.

Además de activar los RigidBodies de cada pieza en los frames correctos, hubo que testear qué tipo de colisionador darle a cada pieza para que su colisión con la hormiga funcionara bien con la animación. Activar el atributo de deformación de malla también mejoró las físicas.

Después de conseguir que todas las partes movidas por la hormiga siguieran sus trayectos, hubo que hacerlas desaparecer al alejarse del huevo. Se creó una esfera grande alrededor del huevo para ayudar a decidir cuándo hacer desaparecer las piezas. Al salir de la esfera, su escala se pondría a 0 para hacerlos invisibles. Inicialmente, se usó la desactivación del render, pero este cambio no se exportaría a Unity.

Tras algunos ajustes, fueron “baked” las físicas de las partes del huevo. Esto consiste en convertir los movimientos de los objetos durante el tiempo a keyframes para crear una animación que no requiere las físicas de Blender para reproducirse.

Con la animación de las partes del huevo terminado, se guardaron de forma separada para el juego las siguientes partes: La pose fetal de la hormiga, su animación de nacer, el modelo del huevo no fracturado y la animación del huevo fracturado abriendose. Este último tuvo que exportarse juntando las animaciones independientes de cada parte del huevo en uno general.

5.5 Implementación de las animaciones e acciones

5.5.1 Animator

El modelo 3D de la hormiga fue exportado de Blender a Unity como un objeto FBX: un formato de fichero muy usado para guardar modelos 3D, animaciones y otros recursos digitales asociados. Unity usa el formato FBX internamente para importar objetos 3D, y los convierte a prefabs. Los ficheros de las hormigas trabajadora y reina fueron importados, por tanto, con su malla, esqueleto y animaciones agrupados.

Para poder controlar la reproducción de las animaciones de un prefab en Unity, se suele usar un componente Animator, que se encarga de controlar las animaciones de un GameObject. Funciona como una máquina de estados, donde cada estado representa una de las animaciones del objeto. Entre estos estados se encuentran las transiciones: conversiones de un estado a otro que ocurren cuando se cumplen ciertos criterios o cuando la animación de un estado acaba.

El componente Animator también puede contener variables que se usan para decidir cuándo ejecutar una transición de un estado a otro o para cambiar aspectos de las animaciones. Hay cuatro tipos: *Float*, *Bool*, *Int* y *Trigger*. Los primeros funcionan como uno se esperaría. El último, *trigger*, se usa solo en transiciones. Cuando una transición que requiere una variable *trigger* ve que está activado dicho *trigger*, ejecuta la transición y desactiva la variable. Los valores de las variables del animator son accesibles desde los scripts mientras se tenga una referencia al componente animator.

5.5.2 Modelo inicial: animaciones básicas de movimiento

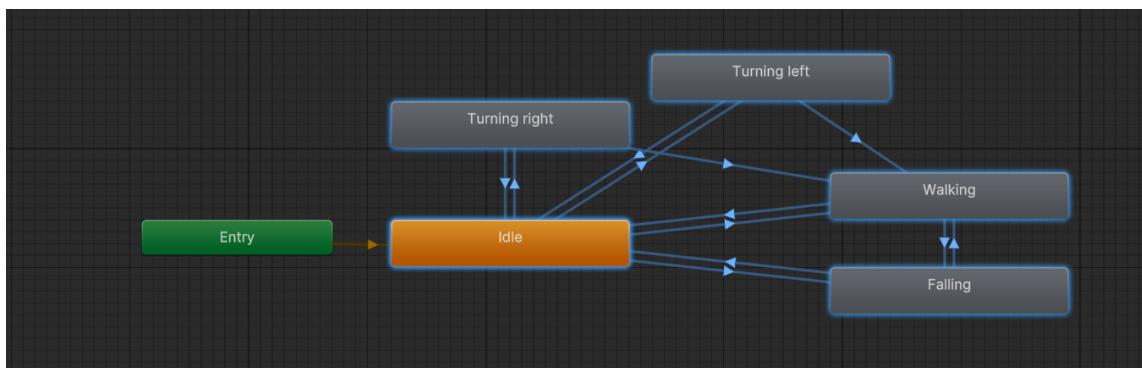


Figure 18: Modelo inicial del animator

La primera versión de la implementación de las animaciones en el animator cubrió todas las animaciones de movimiento por defecto de la hormiga y las transiciones entre estas. Se usaron las siguientes variables para gestionar las transiciones entre las animaciones:

- *Walking*: booleano activado por la hormiga cuando su velocidad no es cero. Se usa para transicionar entre las animaciones de quedarse en el sitio y caminar.
- *Grounded*: booleano activado por la hormiga cuando se encuentra sobre una superficie. Si está desactivado, la hormiga irá al estado *Falling*.
- *Turning*: entero usado para mostrar que la hormiga está girando. Si el valor es 0, la hormiga no está girando. Cuando es 1, la hormiga se está girando hacia la derecha. Cuando es -1, está girando hacia la izquierda. Se usa para moverse y hacia los estados de giro.

La hormiga podía moverse hacia delante y girar a la vez, en cuyo caso se reproduciría la animación de caminar. Por tanto, no se implementó una transición del estado de caminar a los estados de giro, pero el reverso sí.

Para hacer que las animaciones se repitieran indefinidamente mientras que el animator se mantuviera en un mismo estado, hubo que desactivar la propiedad *Has Exit Time* de este (figura 19). Esta propiedad señala si el estado busca transicionar después de acabar la animación o si las transiciones son independientes de la duración de la animación. También hubo que activar la propiedad *loop time* de las animaciones de las hormigas dentro de sus prefab.

5.5.3 Segundo modelo: hormiga llevando objeto

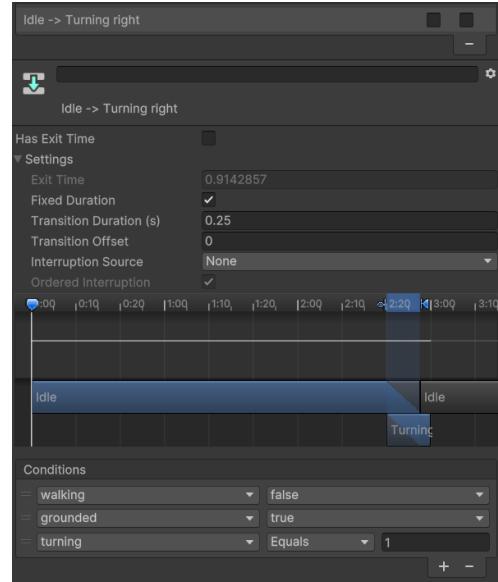


Figure 19: Detalles de la transición del estado Idle al estado Turning right

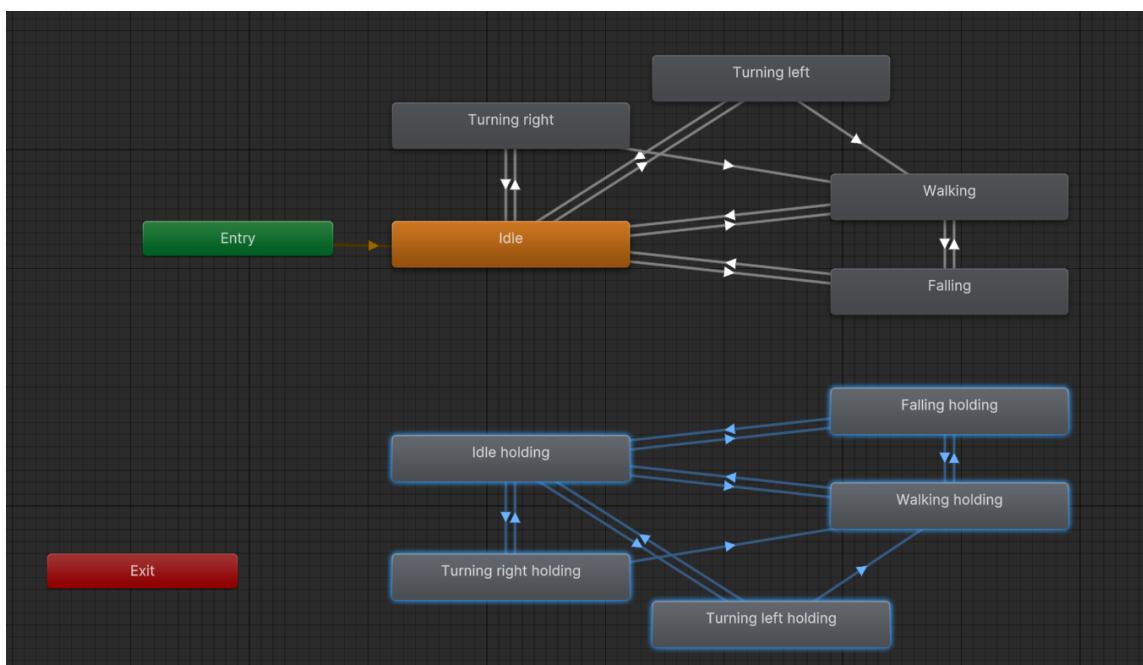


Figure 20: Versión con animaciones llevando objeto añadidos

Se añadieron los estados con las animaciones donde la hormiga se mueve por el mapa mientras tiene un objeto en la boca. El sistema de estados y transiciones es idéntico al previo, usando las mismas variables, ya que lo único que cambia es la animación en cada estado (figura 20).

Hubo que encontrar una forma de poner un objeto entre las mandíbulas de la hormiga, de tal forma que se moviese con su cabeza. Se decidió crear un *EmptyObject* (objeto vacío, es decir, sin modelo 3D) como hijo del hueso de la cabeza. Al ser su hijo, se movería con él, y solo hubo que encontrar la posición y orientación local respecto al hueso para que se encontrara entre las mandíbulas. Después, si se quisiera hacer que un objeto fuera llevado en las mandíbulas de la hormiga, solo haría falta

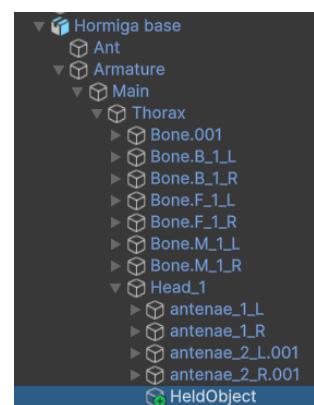


Figure 21: Posición de HeldObject en el esqueleto

colocarlo en el objeto vacío. Las clases de las hormigas trabajadora y reina recibieron una referencia al objeto vacío de su esqueleto para fácil acceso.

5.5.4 Tercer modelo: implementación de acciones de interactuar con el entorno

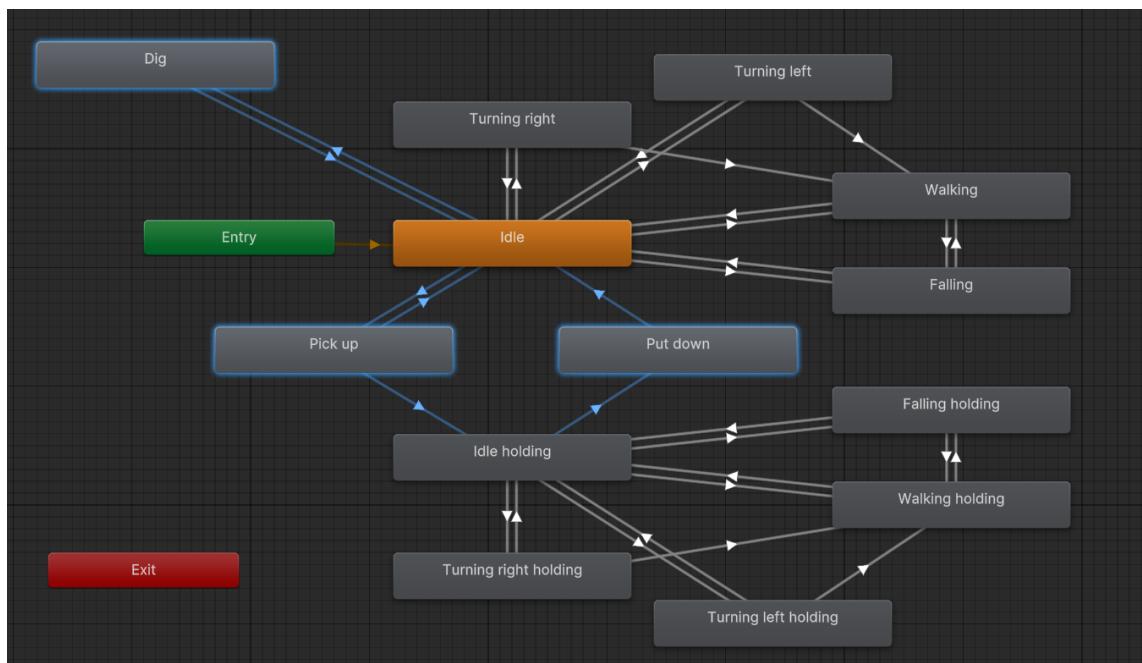


Figure 22: Versión tras la implementación de las acciones Dig, Pick up y Put down

Se implementaron las 3 acciones básicas de la hormiga en la siguiente versión:

- *Pick up*: La hormiga recoge un objeto. Este estado se creó para ser una transición desde el grupo de estados sin objeto al grupo con objeto. En este estado, no solo se reproduce la animación de recogida, sino que también se ejecuta una función, configurada en el *Prefab* mismo. Se implementó la ejecución de la función *PickupEvent()* de la clase de la hormiga en el momento de la animación en el que la hormiga cierra las mandíbulas para recoger el objeto.

La función *PickupEvent()* se encargaría de verificar que el objeto que la hormiga quisiera recoger no se hubiese movido o sido cogido por otra hormiga. En caso de que sí, la recogida del objeto fallaría, y la función activaría las variables que harían transicionar de vuelta al grupo de estados sin objeto antes de que acabara la animación de recoger. En caso de poder recoger el objeto, este se pondría en el objeto vacío *HandledObject* en las mandíbulas de la hormiga, y se transicionaría al otro grupo al acabar la animación.

- *Put down*: La hormiga deja el objeto en el suelo. Similar al estado *pick up*, pero para volver al grupo de estados sin objeto. Este también ejecutaría una función desde la animación: en el momento que la hormiga abriera las mandíbulas para soltar el objeto, llamaría la función *PutDownAction()*. Esta función se encargaría de soltar el objeto en el terreno delante de la hormiga. Después de la animación, el estado transicionaría al grupo de animaciones sin objeto.
- *Dig*: La hormiga excava un *DigPoint*. Para re-

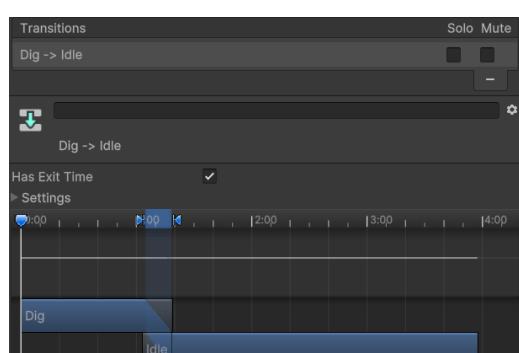


Figure 23: Ajuste de tiempo de transición para salir antes de completarse

presentarlo, se usó la animación de recoger objeto, pero se implementó de tal forma que la animación transicionara al estado *Idle* antes de ser completada, mediante un *exit time* más corto que la duración de la animación (figura 23). De esta forma, en vez de subir la cabeza como si llevara un objeto, la animación de excavar acabaría volviendo a la animación por defecto sin objeto cogido.

Para gestionar la excavación, la animación llamaría la función *DigEvent()*. Este se encargaría de comprobar que el *DigPoint* a excavar todavía existiera, y ejecutaría su función *Dig()* para editar el terreno. La hormiga solo debería poder mover entre las dos redes de estados al recoger o depositar un objeto. Por tanto, se implementaron dos estados mediante los cuales

5.5.5 Cuarto modelo: Implementación de guardado y cargado

Después de implementar la funcionalidad de guardar y cargar la partida, se podía cargar una hormiga llevando un objeto desde memoria. Sin embargo, ya que el animator asumía que la hormiga empezaría sin un objeto en la boca, no se podía transicionar directamente al grupo de animaciones en el que la hormiga lleva un objeto.

Para remediar esto, se añadieron dos transiciones entre los estados *Idle* del grupo sin objeto a *Idle holding* del grupo con objeto. Estas transiciones se ajustaron para que su duración fuera instantánea, y se añadió la variable *IsHolding* al animator para gestionarlo. Gracias a ser instantáneo, la hormiga ya se encontraría en la animación correcta tras ser cargada de memoria antes de ser visible por el jugador.

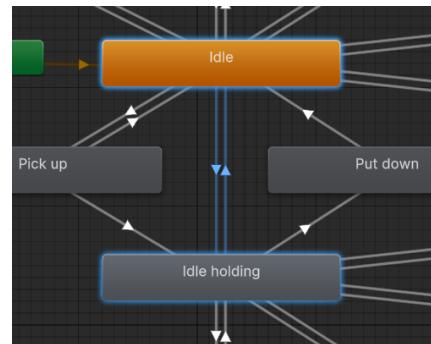


Figure 24: Transiciones entre *Idle* y *Idle holding*

5.5.6 Quinto modelo: Edad y nacimiento de la hormiga trabajadora

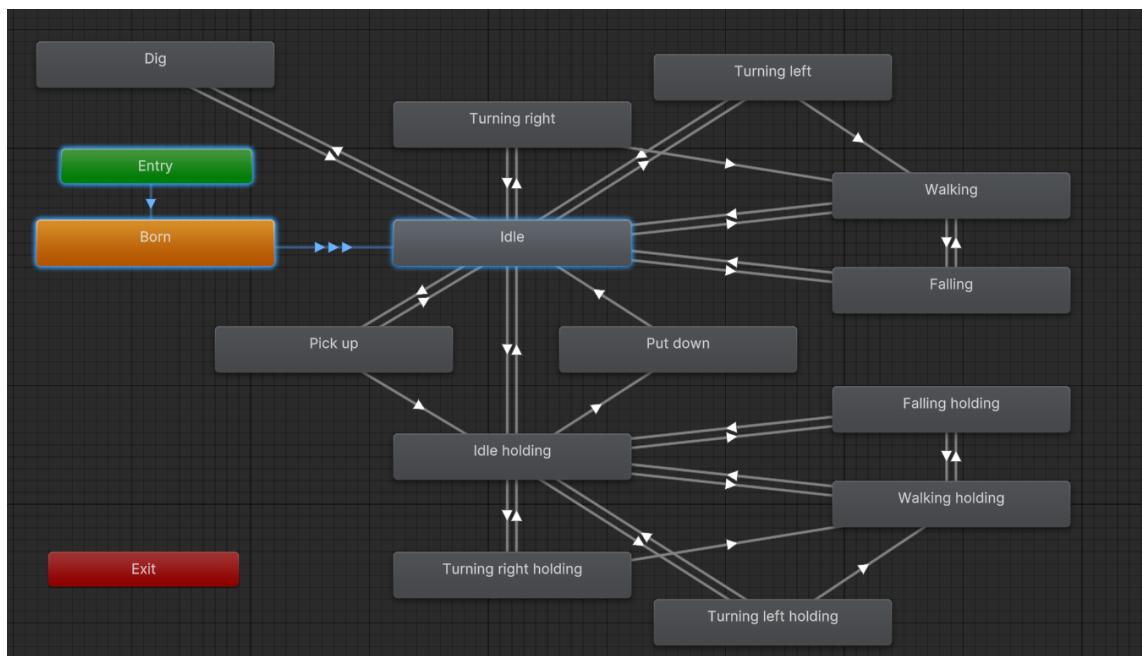


Figure 25: Versión final del animator

Se implementó el ciclo de nacimiento de la hormiga, usando las animaciones de la hor-

miga en posición fetal y su incorporación, el modelo estático del huevo completo y el modelo del huevo roto con su animación de rotura.

A la clase *Ant* del script de la hormiga trabajadora se le añadió la variable entera *age* para representar la edad de esta, que es incrementado cada segundo. Este se usó para tres cosas: gestionar cuando la hormiga sale del huevo, y el tamaño y velocidad de la hormiga.

Al objeto hormiga se le añadieron los modelos de huevo estático y huevo roto. Para sincronizar las animaciones de nacimiento y rotura de huevo, se modificarían los tres objetos a la vez según la edad de la hormiga:

1. Antes de llegar al valor 100, la hormiga no ha nacido aún y se encuentra dentro del huevo. Por tanto:
 1. El componente de animación de la hormiga se encuentra en su primer estado *Born*. La reproducción de la animación está pausada, para que la hormiga se encuentre congelada en la posición fetal del comienzo de la animación de nacimiento. El árbol de comportamiento de la hormiga no es llamado al no haber nacido aún, por lo que la hormiga no toma acciones.
 2. El huevo estático está visible. De esta forma, la hormiga no se ve y se mantiene dentro del huevo.
 3. El huevo roto está desactivado e invisible.
2. Al llegar al valor 100, se ponen en marcha las animaciones:
 1. El componente de animación de la hormiga es despausado para empezar la animación de nacimiento. Al acabar este, transiciona al estado *Idle* y su árbol de comportamiento empieza a funcionar.
 2. El huevo estático se desactiva y se vuelve invisible. No se vuelve a usar.
 3. El huevo roto es activado, volviéndolo visible y empezando su animación de rotura al mismo tiempo que la animación de la hormiga. De esta forma parece que la hormiga rompe el huevo para salir de él. Deja de ser un objeto hijo del objeto de la hormiga, para que no se mueva con este cuando empiece a caminar.
3. Al llegar a la edad 108, unos segundos después de nacer, se desactiva el objeto huevo roto.

Para tener en cuenta que se podría cargar de memoria una hormiga ya nacida, se creó una transición instantánea desde el estado *Born* a *Idle*, que la hormiga usa al empezar a existir con una edad mayor de 100. También se desactivan ambos huevos en hormigas cargadas de memoria ya nacidas .

El tamaño de la hormiga es modificado según su edad. A edad 0, el tamaño de la hormiga y los huevos es de 25% de su tamaño final. Incrementa en tamaño hasta llegar al 100% de su escala a edad 200. La velocidad de la hormiga también escala con su tamaño, para prevenir disonancias visuales entre la velocidad de su animación de caminar y su velocidad de movimiento.

5.5.7 Modelo de la reina hormiga

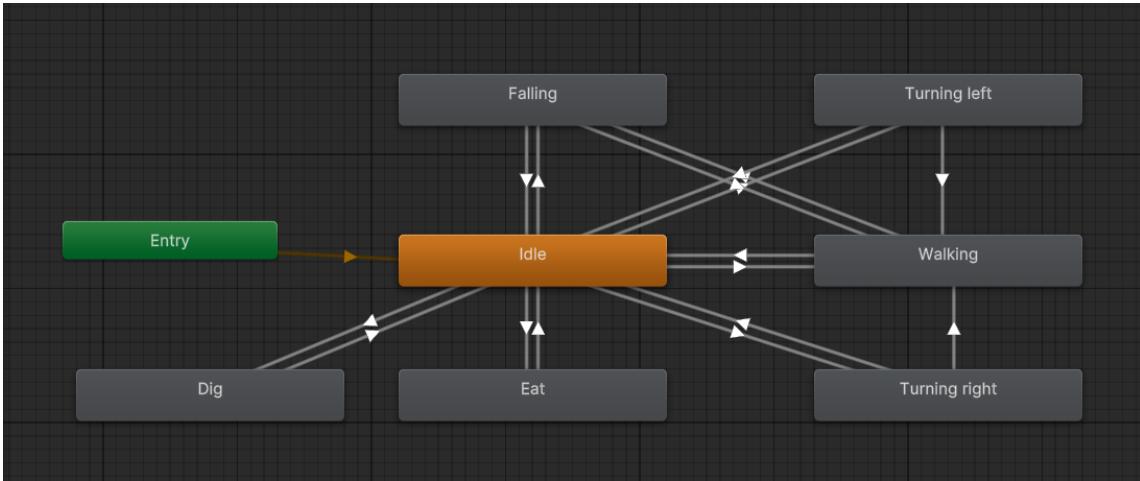


Figure 26: Animator de la hormiga reina

La reina hormiga nunca se encargaría de llevar y mover objetos. Por tanto, se creó su animator a partir de la de la hormiga trabajadora, descartando los estados en los que llevaría un objeto. El estado recoger objeto se sustituyó por el estado comer, en la que la hormiga reina consume pepitas de maíz para recobrar energía y poder dar a luz a más hormigas.

Se creó una animación modificada a partir de la animación de recoger, usando solo la primera mitad en la que la hormiga observa lo que va a recoger y lo muerde. Esta animación se usó en el estado Eat para mostrar la hormiga reina comiendo las pepitas de maíz. Estas serían eliminadas en cuanto la reina los muerde, después de lo cual la reina vuelve a su animación original.

Para el resto de las animaciones se usaron las mismas que las de la hormiga trabajadora. El esqueleto de la hormiga reina tiene distintas proporciones que el de la hormiga, pero el mismo número de huesos, por lo que las mismas animaciones se pudieron usar sobre ambos modelos sin tener un aspecto extraño.

5.6 Color del fondo: debajo o encima del terreno?

Por defecto, el fondo del juego es una skybox, pero usando la función `backgroundColor()` de la clase `Camera`, este puede ser cambiado a un color sólido. Para la estética del juego, se decidió cambiar el color de fondo a un azul pastel.

La cámara del jugador es capaz de atravesar el terreno y ver los túneles y cámaras del nido, mirando a través de los lados más cercanos a la cámara. También se puede ver aún el cielo, rompiendo la ilusión de encontrarse bajo suelo.

Se decidió hacer que el color de fondo fuera dinámico, azul sobre el suelo y color tierra debajo de este. Para ello, en cada `FixedUpdate()` habría que comprobar si la posición de la cámara se encuentra bajo el terreno o no, y ajustar el color de fondo según el resultado. Hasta este punto se podía mirar si un punto entero `Vector3Int()` del mapa se encontraba debajo o encima del suelo comprobando su valor en el campo escalar con `SampleTerrain()`, pero la cámara podía moverse entre estos puntos, en coordenadas `Vector3()`.

Se implementó una función de sobrecarga `SampleTerrain()` que tomara una coordenada `Vector3()`. Este obtendría el valor del campo escalar entre sus puntos usando interpolación lineal. Usando esta nueva función, se pudo implementar el fondo de mapa di-

námico.

5.7 Aplicando color al terreno

5.7.1 Aplicar multiples colores a una malla

Durante la mayoría de la creación del proyecto, las mallas de terreno han tenido un único color blanco. Cuando llegó el momento de desarrollar el estilo visual del juego, hubo que decidir cómo representarlo. Se consideró usar texturas, pero la naturaleza dinámica del algoritmo Marching Cubes dificultaría su implementación, por lo que se decidió usar colores simples.

Al simplemente asignarle un material con color a WorldGen y hacer que las mallas de los chunks usen ese material, es posible cambiar el color del terreno. Sin embargo, en vez de representar todo el mapa en el mismo color, se quiso poder diferenciar las zonas excavadas por las hormigas de la superficie. Para ello se necesitaron dos cosas: poder aplicar más de un color al terreno y poder diferenciar qué partes de la malla se encuentran debajo del suelo y cuáles se encuentran encima.

La clase malla, aparte de los arrays de vértices y triángulos requeridos para su funcionamiento, contiene un array assignable opcional de objetos Color llamado colors. Tiene el mismo tamaño que el array de vértices y representa los colores de estos. Cuando se renderizan los triángulos, estos tienen como color un gradiente entre los 3 colores de los vértices que lo forman. Se implementó, pues, la creación de una lista de colores y su asignación a la malla:

Se añadió a la clase *Chunk* una lista de colores: *List<Color> colores*

En la función *MarchCube()*, cada vez que se añaden 3 vértices y un triángulo a sus listas respectivas, se añadirían tres entradas a la lista colores con los colores relevantes.

Durante la creación de la malla, se asignaría dicha lista al array de colores de la malla.

La asignación de colores no tuvo efecto sobre la malla inicialmente, ya que el material base de la malla no permitía el uso de estos. Para poder representar los múltiples colores, se tuvo que cambiar a un uso; como material, un *ParticleSurface*. Este permite la visualización de una variedad de efectos visuales en la malla, el más importante de los cuales es el gradiente de colores en los triángulos usando el array de colores.

Esta implementación permitió el uso de múltiples colores, pero hubo que poder decidir qué partes de la malla se encontraban debajo del terreno. Un primer intento fue mirar si se encontrara el vértice respectivo dentro de alguna parte del nido. Aunque las funciones de la clase *Nest* para ver si un punto del mapa se encontraría dentro del túnel ya fueron implementadas durante el desarrollo de esta parte, el uso de estas fue inconsistente, ya que muchos de los vértices del terreno se encontrarían justo fuera de las partes de nido.

5.7.2 Guardar una copia del mapa original

La solución fue guardar la versión original del mapa, y cada vez que se asignaría un color a un vértice, mirar si se encontraba debajo del terreno en el mapa original. Si fuera el caso, el color asignado sería marrón tierra. Si no, verde hierba. Las implementaciones necesarias para llevar esto a cabo fueron:

Crear nuevas funciones de obtención de valor del campo escalar original, iguales a las preexistentes del campo escalar original. Estos incluían, aparte de la obtención de valor de un punto `Vector3Int`, el de un valor entre puntos de la malla con `Vector3`.

Añadir un nuevo array 3D a `WorldGen` llamado `memoryMap`, que contendría el campo escalar original.

Añadir la función de codificar, guardar y descodificar para cargar el nuevo array a la clase `GameData`.

En la función `MarchCubes()`, añadir el check de debajo de tierra: Si un punto se encuentra sobre la superficie en el mapa actual pero bajo la superficie en el mapa original, se encuentra dentro del nido, por lo que se le asigna el color marrón. En otro caso se le asigna el color verde.

Funcionó bien tanto durante la excavación del terreno por las hormigas como tras guardar y cargar el mapa. Sin embargo, en el modo edición del mapa, todos los cambios resultarían en terreno marrón. No tuvo sentido, ya que el editor del mapa crea el mapa “prehecho” en el que empiezan las hormigas y debería estar cubierto por hierba en su estado natural. Se decidió asignar a `memoryMap` una copia del mapa nuevo cada frame que el jugador editase el terreno, pero asignar arrays multidimensionales de ese tamaño cada `FixedUpdate` resultaba en una carga de procesamiento demasiado alta. Se añadió a la clase `Chunk`, por tanto, una función que actualizaría solo su porción del campo escalar al `memoryMap` si se editara su terreno en el modo edición de mapas.

Finalmente, se decidió asignar también el color marrón a las superficies del mapa que se encontraran bocabajo o laterales, ya que no tendría sentido que creciera hierba sobre paredes y techos. Para ello, en `MarchCubes`, antes de mirar si el vértice está dentro de un área excavada, se mira la normal de su triángulo. Si este no se encontrara dentro de 80 grados del vector `Vector3.up`, directamente se le asignaría el color.

5.8 Uso de partículas

5.8.1 Qué son partículas

En el ámbito de los videojuegos, las partículas son pequeños elementos visuales, como imágenes o mallas, que se utilizan para crear efectos como fuego, magia, explosiones o cualquier animación que requiera la representación de una gran cantidad de elementos visuales. Las animaciones que crean son dinámicas y pueden ser tridimensionales.

En Unity, las partículas se gestionan mediante sistemas de partículas: componentes que se pueden agregar a un objeto en la escena y controlan la creación, movimiento y comportamiento de las partículas. Designan un área dentro de la cual se emiten las partículas y permiten la gestión de las muchas propiedades distintas de su apariencia y comportamiento.

5.8.2 Obtención de partículas

La motivación inicial de la implementación de partículas en el juego fue para representar los caminos de feromonas sobre el terreno. Se quiso usar partículas que quedarían bien dada la estética low poly del juego, es decir, hechas de mallas poligonales simples.

Se buscaron recursos en la Asset Store de Unity, una plataforma en línea donde los usuarios de Unity pueden comprar o descargar recursos (assets) para sus proyectos de desarrollo de videojuegos. Se encontró un paquete de partículas low-poly gratuito llamado “Polygons’ Low-Poly Particle Pack” publicado por el usuario de Unity “Polygons Stuff”. Este contenía una variedad de materiales, mallas y emisores de partículas prefabs, de los cuales se acabó usando el emisor de partículas nube de radiación, al parecerse más a una nube de feromonas.

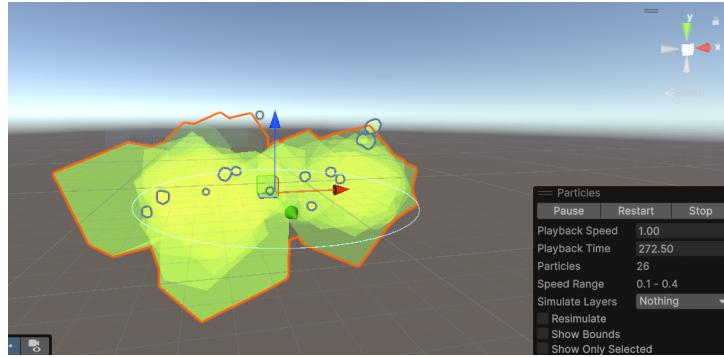


Figure 27: Emisor de partículas de nube de radiación.

Partiendo del emisor de nubes de radiación, se obtuvo el emisor de partículas adecuado para representar las feromonas aplicando las siguientes modificaciones:

1. Se eliminó el subsistema de partículas que soltaba las nubes más pequeñas, pasando a usar solo las nubes grandes.
2. El área de emisión se cambió de modo círculo 2D a esfera, para no necesitar tener en cuenta la orientación de la feromona sobre el terreno.
3. Se extendió el tiempo de vida de las partículas, se disminuyó el tamaño inicial, se cambió su color inicial a verde-amarillo, se modificó el gradiente para que el color cambiara a rojo, se eliminó la gravedad y se aumentó el número máximo de partículas y la frecuencia de emisión.
4. Posteriormente, se modificaron las partículas para volverse visibles e invisibles al nacer y morir más gradualmente, para que no apareciesen y desapareciesen tan repentinamente que llamaran la atención.

5.8.3 Representar Feromonas

La primera implementación de las partículas para representar las feromonas fue crear un objeto sistema de partículas sobre cada coordenada del mapa en la que se encontrara una feromona. Se usó un diccionario para guardar referencias a los objetos según sus coordenadas del mapa. Durante el testeo, se observó que grandes cantidades de sistemas de partículas afectaban notablemente el rendimiento visual del juego, por lo que se optó por buscar una forma distinta de emitir las partículas de las feromonas.

El segundo método implementado para emitir partículas se basó en usar un solo objeto *ParticleSystem* para emitir las partículas de todas las feromonas. Para ello se usó la interfaz *EmitParams* de la clase *ParticleSystem*. A partir de un objeto *ParticleSystem* se podía crear una interfaz *EmitParams* especificando ciertos parámetros de emisión. Este a su vez se podría usar en la función *Emit()* de la clase *ParticleSystem*. Se emitirían partículas siguiendo las propiedades cambiadas por la interfaz *EmitParams*, y para las propiedades no cambiadas por la interfaz se usarían las del mismo objeto *ParticleSystem*.

Se creó la clase *Emitter*, que se ocuparía de emitir todas las partículas, y se le metió una referencia al objeto sistema de partículas de las feromonas. En ella se creó la función *EmitPheromone()*. Dada una posición, la función crearía una interfaz *EmitParams* con el parámetro posición cambiada y pasaría a usarla para emitir una partícula en la posición dada usando *Emit()*. Cada *FixedUpdate()* la clase *Emitter* emitiría una partícula en cada

coordenada de feromona usando esta función.

```
Public void EmitPheromones(Vector3 pos, int age)
{
    ParticleSystem.EmitParams ep = new ParticleSystem.EmitParams;
    {
        Position = pos,
        ApplyShapeToPosition = true, //hace que no siempre se emitan igual
        StartSize = 10 + (50f * age / 100f) //tamaño de 10 a 60 dependiendo de edad
    };

    PheromoneParticleSystem.Emit(ep, 1); //emitir una partícula con las propiedades dadas
}
```

CodeBlock 3: Función EmitPheromones de la clase Emitter.

La implementación inicial de este método solo emitiría partículas en la coordenada de la primera feromona o en la última. Se descubrió que se debía a que, al emitir una partícula por intervalo de juego, se llegaría demasiado rápido al límite de partículas permitidas por el *ParticleSystem*: en cuanto se liberara un hueco, se emitiría uno en la primera coordenada de la lista. Para remediarlo, se aumentó el límite de partículas y se disminuyó la frecuencia de emisión.

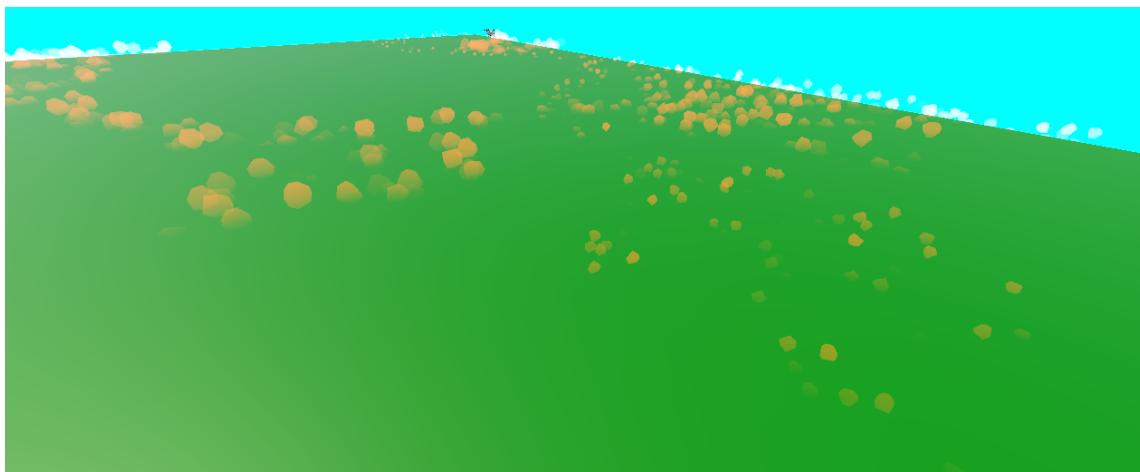


Figure 28: Caminos de feromonas sobre superficie plana. Los de la izquierda son más recientes, los de la derecha más viejas.

Para disminuir la frecuencia de emisión, se decidió implementar un sistema de emisión nuevo. Se clasificaron las coordenadas en 8 grupos distintos según la paridad de sus tres coordenadas (2^3). Cada 10 *FixedUpdate()* se emitiría una partícula en todas las coordenadas de feromonas en uno de los grupos. Así cada coordenada recibiría una partícula cada 80 actualizaciones (poco menos de un segundo).

También se implementó la desactivación de partículas de feromonas mientras el jugador se encontrara debajo del suelo. Debido al sistema de renderización, desde dentro del terreno el jugador no podía ver las partículas de feromonas fuera del terreno, pero sí las partículas que se encontraran dentro del terreno. Esto resultaba visualmente extraño y podría confundir al jugador, por lo que se decidió pausar la simulación del *ParticleSystem* y desactivar su renderizador mientras que el jugador se encontrara debajo del suelo.

Por último, también se implementó la visualización de la edad de las feromonas. La función *EmitPhermone* se editó para tener en cuenta la edad de cada feromona, y cambiaría su tamaño según este valor. Cuanto más cerca de desaparecer, más pequeñas se-

rían las partículas.

5.8.4 Representar DigPoints

En el caso de los puntos de excavación, no habría suficientes en el mapa a la vez para necesitar la implementación de un sistema especial. Por tanto, simplemente se usó el *ParticleSystem* anterior, editado para cambiar su color, como el objeto al que se asoció el script *DigPoint*. De esta forma no hubo que implementar nada en código para su funcionamiento.

Se modificaron el tamaño y movimiento del *ParticleSystem* para representar mejor los puntos de excavación. Se modificó también el color, para diferenciar mejor puntos de excavación de caminos de feromonas. En vez de empezar en verde y cambiar a un tono rojizo, mantendría un color rosa-dorado constante.



Figure 29: Una hormiga rodeada de puntos de excavación dentro del terreno

5.9 Representación gráfica del nido

Para representar las distintas partes del nido, se quiso usar un material transparente, permitiendo que el jugador pudiese diferenciar las partes del nido del resto del mapa y a la vez mirar a través de las paredes exteriores del nido. Sin embargo, el solapamiento de las secciones transparentes de los distintos objetos tridimensionales que componen las partes del nido creaba zonas de transparencia más opacas que otras (figura 30, izquierda superior). Hubo que encontrar una forma de eliminar dicho solapamiento.

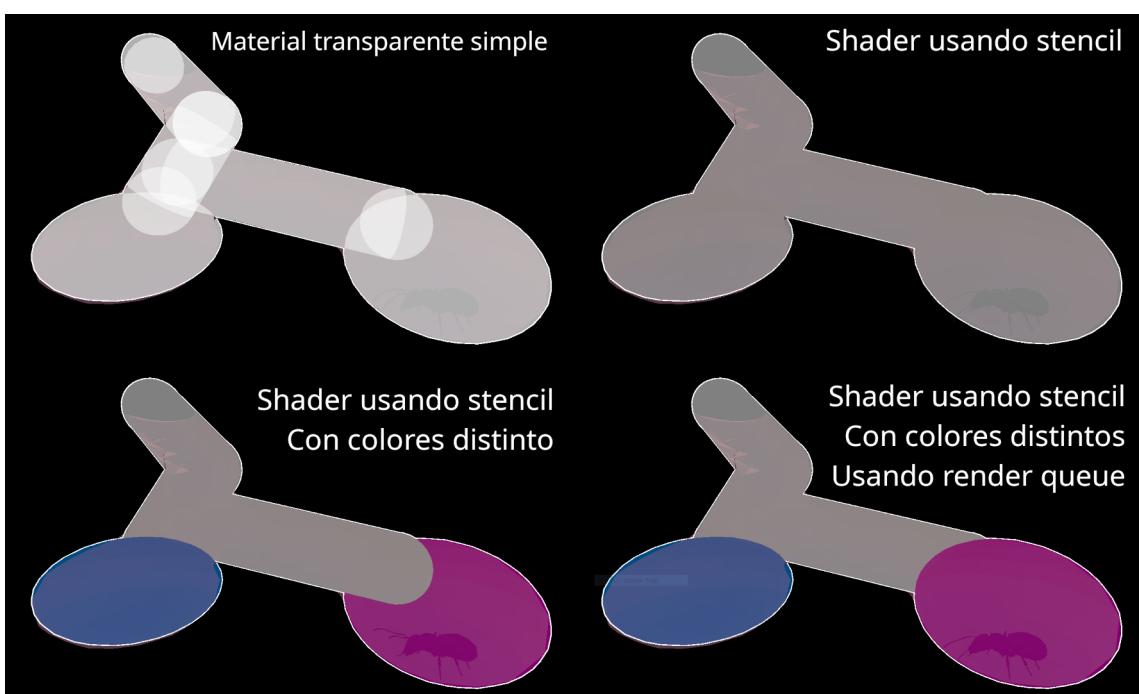


Figure 30: Las distintas fases de representación visual de las partes del nido

5.9.1 Incorporación de la Universal Rendering Pipeline

La Universal Rendering Pipeline (URP), o Canalización de Renderizado Universal, es una solución de renderizado en Unity para la creación de gráficas. Es la versión mejorada y más avanzada de la built-in Rendering Pipeline o canalización integrada que se usa por defecto en Unity.

Las canalizaciones de renderizado se usan para renderizar los elementos del juego, gestionando los canales de colores, sombras, albedos, etc. Por defecto, Unity usa una canalización integrada de propósito general con opciones de personalizaciones gráficas limitadas. Este sirvió para la mayoría de las gráficas de juego, pero limitó el aspecto gráfico del nido, ya que no existe ninguna forma de representar materiales transparentes que no se solapan.

URP se incorporó con el fin de utilizar su mayor capacidad de personalización de shaders para representar las piezas del nido de forma transparente, sin que se superpusieran sus materiales. Se pudo obtener de forma sencilla y gratuita en la *Asset Store* oficial de Unity.

5.9.2 Buffer de stencil

En URP, los shaders pueden incorporar el uso del buffer de stencil (plantilla). Se trata de una máscara de propósito general sobre cada píxel del output gráfico que permite guardar o descartar píxeles.

En el proceso de renderizado, para cada píxel de la pantalla, se miran los shaders de todos los objetos que podrían aparecer en dicho píxel, desde más lejos a más cercano, para decidir qué color mostrar en el píxel. Normalmente, se representa el objeto más cercano a la cámara si no es transparente.

En este proceso, los shaders pueden observar y modificar el valor del buffer de plantilla para decidir si descartar su píxel. Se quiso hacer que el shader del material de las partes del nido no se depicte si en el mismo pixel ya se encontrara otro pixel de las partes del nido.

La modificación por texto de shaders es un asunto complejo en el que no se adentró para la realización del proyecto, con como excepción el uso del buffer de plantilla. Esto requiere una composición de comandos específicos dentro de una de las secciones del archivo shader. Los comandados introducidos son los siguientes:

```
Stencil {
    Ref 20
    ReadMask 20
    Comp NotEqual
    Pass Replace
}
```

Los comandos se contienen dentro de un bloque Stencil para señalizar el uso de esto al lector del shader.

- El primer argumento, *Ref*, se usa para definir el valor numérico que se usará en las siguientes operaciones (referencia).
- El segundo argumento, *ReadMask*, denota la máscara que se usará para leer/escribir datos al búfer de plantilla.
- El tercer argumento, *Comp*, define la función de comparación que se usa para

decidir si descartar o no el píxel que el shader renderizaría. En este caso se usa *NotEqual*, significando que si en el buffer se lee un valor que no sea igual al valor de referencia 20, el píxel se guardará. Si el valor es igual, el píxel se descartará y no se mostrará.

- El cuarto argumento, *Pass*, define una acción a tomar además de guardar el píxel si la comparación devuelve el valor verdad. En este caso, si no se lee el valor de referencia 20 en el búfer, este es reemplazado (*Replace*) por el valor de referencia 20.

De esta forma, el shader del primer objeto *NestPart* que mostrará un píxel en uno de los píxeles de la pantalla ve que el valor del buffer no es igual a 20. Se guardará su píxel para ser renderizado y se pondrá el valor del búfer a 20, después de lo cual los demás shaders del nido en el mismo píxel verán que el valor es 20 y descartarán su píxel.

Se puede observar el resultado de este método en la parte superior derecha de la figura 30.

5.9.3 Uso de distintos colores en las partes del nido

Después de haber conseguido resolver el problema de solapamiento, se quiso poder diferenciar visualmente los tres tipos de cámara de nido y los túneles unos de los otros. Se decidió, por tanto, crear múltiples materiales de colores distintos usando el mismo shader. Ya que se usó el mismo shader, no se solaparon las partes del nido, ni siendo distintos colores transparentes.

Se decidió usar amarillo para la cámara de la reina, azul para las cámaras de comida y morado para las cámaras de huevo. Los túneles se mantuvieron de color blanco simple, para destacar más las cámaras. Se añadió un material de color verde para destacar las cámaras que aún no estuviesen excavadas del todo. Se puede ver el resultado del uso de los colores en la parte izquierda inferior de la figura 30.

También se añadieron dos materiales opacos usando un shader simple para uso en la colocación de partes nuevas del nido. Esto se hizo para que se pudiera ver dónde estuvieran respecto a las otras partes ya colocadas del nido, ya que usar el mismo shader que estos eliminaría las sombras y superposiciones visuales que aportan información espacial. Se usó un material opaco blanco para posiciones válidas y uno rojo para posiciones inválidas.

5.9.4 Uso de Render Queue

Uno de los problemas del uso de colores en las partes del nido fue que se podía mostrar las partes del túnel en vez de la cámara donde estos dos se interseccionan. Ya que la parte del túnel que conecta con la cámara se encuentra dentro de esta, y además se le atribuye más importancia a las cámaras que a los túneles que las conectan, se quiso priorizar la renderización de las cámaras.

Se decidió usar el render queue para resolver el problema. El render queue representa el orden en el que se renderizan los distintos materiales de la escena del juego. Cada material tiene asignada una etiqueta de valor numérico entero entre 0 y 5000 denominada Render Queue, que se usa para decidir el orden de renderizado respecto a los demás objetos.

Objetos con un render queue más bajo son renderizados antes, y por tanto detrás de los siguientes objetos. En el caso de los materiales que usan el buffer stencil, los objetos que son renderizados después tendrán sus píxeles descartados, por lo que interesa

darle un valor menor al material de las cámaras que al de los túneles. Unity usa 5 valores por defecto de etiqueta para sus objetos:

- *Background* (1000): Usado para los elementos del fondo del mapa.
- *Geometry* (2000): Usado para los objetos opacos de la escena.
- *Transparent* (3000): Los objetos transparentes de la escena. Los materiales del nido son transparentes, por lo que sus etiquetas de Render queue tendrán valores similares a este.
- *Overlay* (4000): Los componentes de la interfaz gráfica. Estos deben aparecer delante de cualquier otro objeto de la escena, por lo que son los últimos en ser renderizados.

Los valores de Render queue aplicados a los distintos materiales fueron los siguientes:

- 3001 para el material de parte no excavada aún. La excavación de estas es una prioridad del nido, por lo que deben ser fácilmente visibles y, por tanto, recibieron la prioridad más alta entre materiales del nido.
- 3002 para los materiales de las cámaras de huevo, comida y reina. Las cámaras no pueden solaparse, así que da igual que tengan la misma prioridad.
- 3003 para el material blando de los túneles. Estos son de menos importancia que las cámaras, y tenerlas como última prioridad mejora notablemente el aspecto del nido.

Se puede observar el efecto del uso del Render queue en la parte derecha inferior de la figura 30.

5.10 Consecuencias de uso de URP

5.10.1 Necesidad de nuevos shaders para cada material



Figure 31: Material de error sobre la hormiga tras pasar a URP

El cambio de canalizador de renderizado afectó a todos los objetos de la escena. En concreto, los materiales usados para el canalizador por defecto no usaron shaders válidos en la URP. Estos se mostrarían en la escena como material de error de color rosa claro (figura 31).

Se sustituyeron todos los materiales por otros que usaron los shaders compatibles con la URP. Esto se hizo guiándose por la tabla de conversión de shaders del canalizador de renderizado integrado a shaders de URP que se puede encontrar en la página web de

documentación de Unity, [Unity3d.com](https://unity3d.com).

5.10.2 Creación de shader en URP para el terreno

En URP no hay un shader por defecto que use los colores de vértices de la malla dada y produzca sombras a la misma vez. Por tanto, el terreno que en la canalización de renderizado integrado mostró estas dos cualidades de repente no pudo ser representado correctamente. Solo hubo acceso por defecto a shaders que usaban los colores de la

malla sin producir sombras y shaders que producían sombras pero no usaban los colores de la malla (figura 33).

Se consideró aprender a escribir shaders para crear uno personalizado que cumplía los requisitos gráficos del terreno, pero escribir shaders directamente con código es un proceso complicado si no se es familiar. Por suerte, URP proporciona al usuario de Unity acceso a la funcionalidad Shader Graph: una forma de crear shaders para Unity de forma altamente visual y high level. Gracias a esta herramienta, se pudo crear un shader muy sencillo para representar el terreno (figura 33). Se creó enchufando el valor Vertex Color de la malla en el color usado para un shader que proyecta sombras. El uso de este se puede observar en la imagen izquierda inferior de la figura 32.

Con este shader, la única imperfección de la renderización del terreno fueron las grietas de luz en sus sombras. Estos se debieron a que el canal de sombreado tuviera en cuenta las normales de la malla, de las cuales no se hace la media entre vértices en el algoritmo de Marching Cubes. Por suerte, en vez de tener que modificar la complejidad de dicho algoritmo, se bastó con disminuir la influencia de las normales de la malla en las sombras a 0. Esta funcionalidad fue posible gracias a la URP, y su efecto se puede observar en la parte inferior derecha de la figura 32.

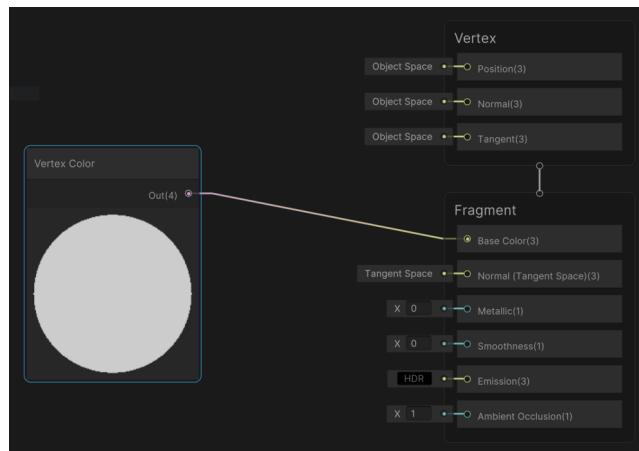


Figure 32: Shader personalizado creado en Shader Graph

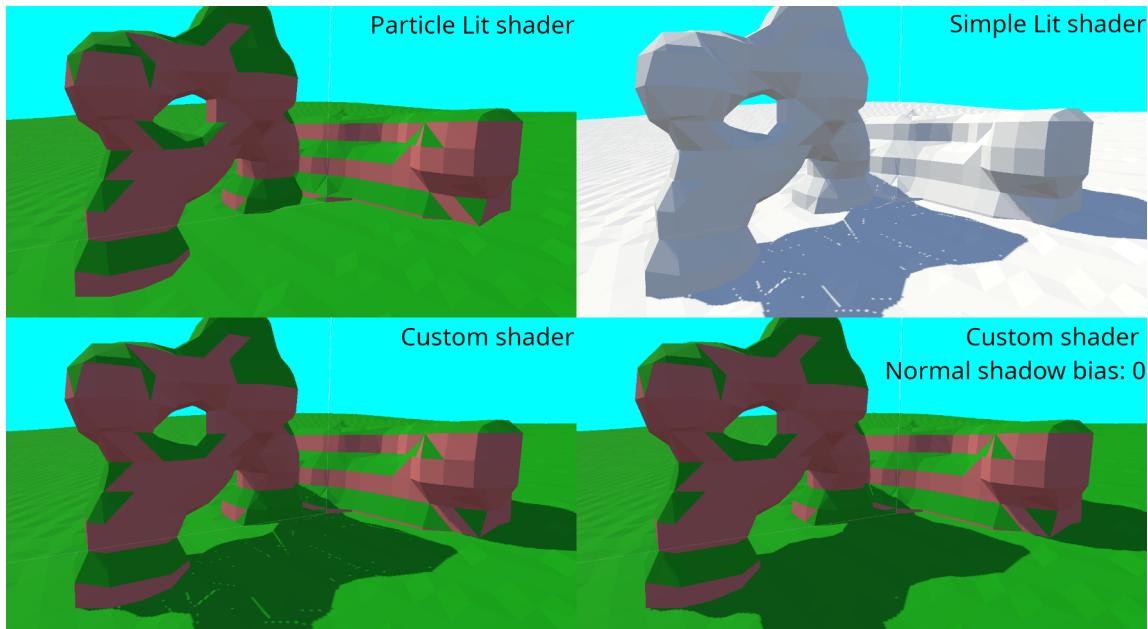


Figure 33: Los distintos shaders y ajustes probados para representar el terreno en URP

6 Físicas del juego

6.1 Interacción de la hormiga con el terreno

6.1.1 Los dos estados de la hormiga

El sistema de físicas del juego requirió que las hormigas pudiesen caminar sobre cualquier parte de la malla generada por el algoritmo de Marching Cubes. Esto significó que debieron poder caminar sobre superficies boca abajo, paredes y el mismo suelo. No se pudo, por tanto, aplicar la fuerza de gravedad estándar sobre las hormigas cuando éstas se encontraran sobre alguna superficie, ya que causaría que se cayeran. Sin embargo, la posibilidad de la hormiga de separarse de alguna pared o techo y caerse al suelo por la fuerza de gravedad también tuvo que tenerse en cuenta.

Por tanto, para gestionar el movimiento de la hormiga en el entorno, se le asignaron dos estados posibles: el de estar en una superficie y el de estar cayendo. El vector de fuerza de gravedad solo se aplicaría a la hormiga al estar en el estado cayendo.

Para decidir si se encontraban en el estado sobre la superficie o cayendo, la hormiga debió ser capaz de sentir la malla del terreno debajo de ella. Para conseguir esto, el agente usó la función *Raycast()*. Dado un punto de origen, una dirección, una longitud máxima y una máscara, esta función proyectaría una línea con dichas características y devolvería un objeto *RayCastHit* al colisionar con algún objeto en una de las capas especificadas en la máscara. En caso de no colisionar, devolvería false. El objeto *RayCastHit* contendría la información de posición de la colisión, la normal de la superficie con la que se colisionó, una referencia al *collider* con el que colisionó, etc.

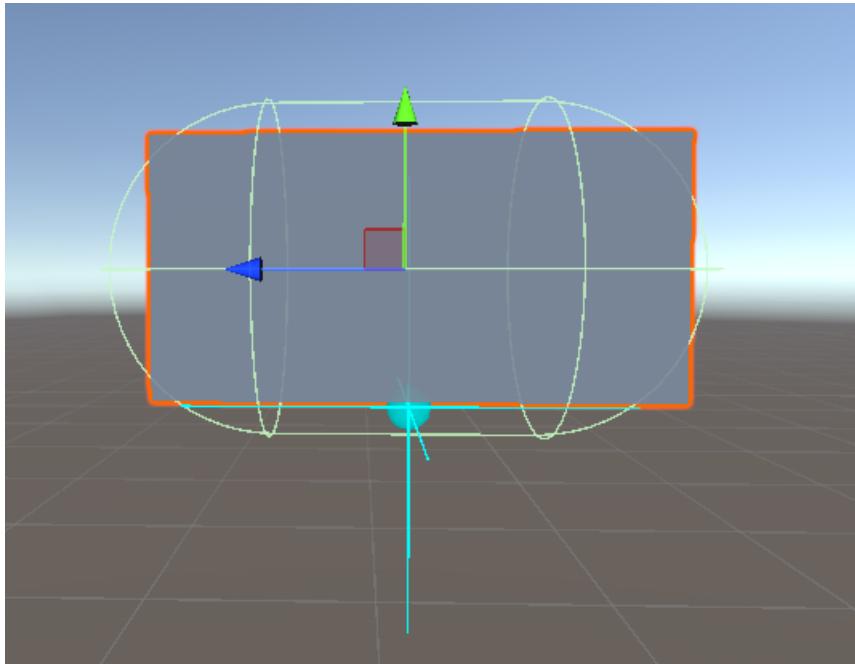


Figure 34: Cuboide rectangular usado para pruebas tempranas de las físicas de la hormiga

Cada *Update()* el script *Ant* llamaría la función *Raycast()*, proyectando un rayo corto desde su centro hacia abajo. Solo se quiso detectar si se encontrara sobre una de las mallas de la superficie, por lo que a estos se les asignó la capa “terreno” y *Raycast()* usó la máscara que solo colisiona con dicha capa. Si la función *Raycast()* devolvía true, significaría que detectó el terreno debajo de la hormiga, por lo que se activaría el estado “sobre superficie” y la gravedad se desactivaría. Si la función devolvía false, se consideraría que la hormiga estaba en el estado cayendo, por lo que se le activaría la gravedad normal.

6.1.2 Gestiónar movimiento sobre malla y más Raycasts

Después de programar que la hormiga no se cayera por la gravedad, hubo que implementar su movimiento en el estado “sobre superficie”. Debió moverse paralelo a la superficie para no separarse y caerse. Por lo tanto, en vez de que la hormiga se moviera hacia donde mirara, se decidió hacer que se desplazara acorde al vector de su dirección proyectada sobre la superficie del suelo. La función *Vector3.ProjectOnPlane()*, incluida por defecto en Unity, nos permite obtener este vector dada la dirección hacia delante de la hormiga y la normal de la superficie sobre la que se encuentra.

Se escribió un simple código para mover la hormiga en la dirección del vector proyectado si este se encuentra “sobre superficie” al pulsar la tecla de flecha hacia arriba para poder probar el sistema. Proporcionó movimiento aceptable en terreno plano, pero mostraba problemas al subir por elevaciones. Debido a su gran cápsula de colisión, al subir por elevaciones, el Raycast se alejaría demasiado de la superficie del terreno, poniéndolo en estado de caída y dejando inamovible el agente (Imagen 35 a). Para solucionar esto, se añadieron más raycasts a los extremos de la hormiga, con el objetivo de siempre poder sentir parte de la superficie sobre la que se encontraría el agente.

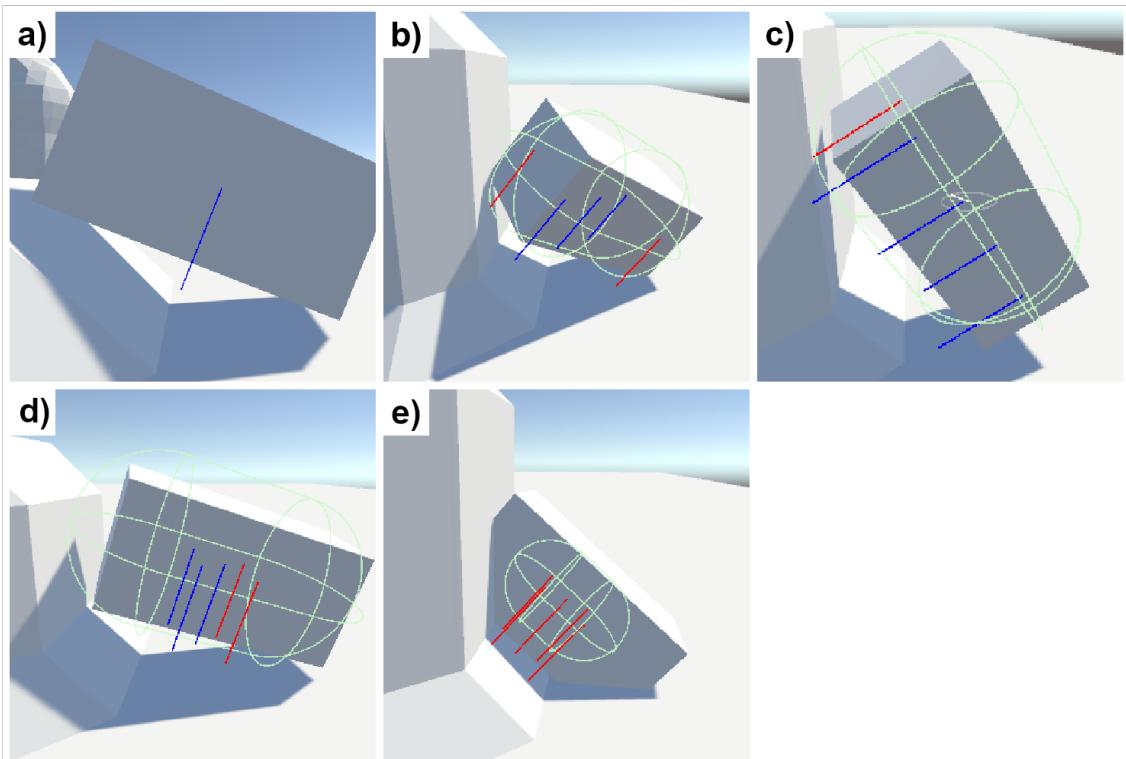


Figure 35: Casos obtenidos al testear

Al tener más raycasts, se necesitaba gestionar cuántos harían falta que sintieran el terreno para que el agente se considerara sobre la superficie y se pudiera mover. Se decidió inicialmente que este no estuviera cayendo si 3 o más de los 5 raycasts sintieran el terreno. Al usar múltiples raycasts, se tuvo que decidir también qué normal de las superficies detectadas usar para calcular la proyección del vector de movimiento sobre la superficie. Se decidió usar inicialmente la media de los normales de los planos del terreno que los raycasts vieran. Este sistema de usar múltiples raycast permitió al agente subir cuestas con más facilidad, pero no evitó que se quedara pillado del todo (figura 35 b). Se probó disminuir los Raycast activados necesarios a solo 2, lo cual dificultaba aún más que se quedara pillado, pero nunca llegó a evitarlo del todo. En particular, cuestas repentinas dificultaban la detección del terreno. En la figura 35 c podemos ver cómo solo uno de los raycast ve el terreno al intentar subir la cuesta.

6.1.3 La orientación de la hormiga sobre la malla

Otro obstáculo fue tener que encontrar una forma de hacer que la dirección en la que mirara el agente fuera perpendicular al terreno. El agente podía acabar mirando en una dirección diagonal con respecto al terreno, lo que podría causar que se separase de ella al moverse por superficies curvadas.

Esto no mostraba ser un problema demasiado grande en el caso de moverse por superficies cóncavas (figura 36.1), ya que la superficie misma empujaría la parte delantera de la hormiga, haciéndola seguir la dirección de la inclinación.

Destacaba más en superficies convexas (figura 36.2), donde hacía falta ajustar la dirección de la hormiga según el terreno sobre el que se encontraba; es decir, el vector hacia arriba del agente debió ser similar a la normal de la superficie sobre la que se encontraba. Esta normal se calcularía como la media de los normales obtenidos en los RayCastHits que proporcionarían los Raycasts, y permitió al agente mover a velocidades más rápidas sin desconectarse del terreno. Sin embargo, aún sufría problemas al subir cuestas y caminar por terrenos irregulares: su cambio de dirección repentina cau-

saba que los extremos del agente pudieran chocar con el terreno y separar a la hormiga de ella.

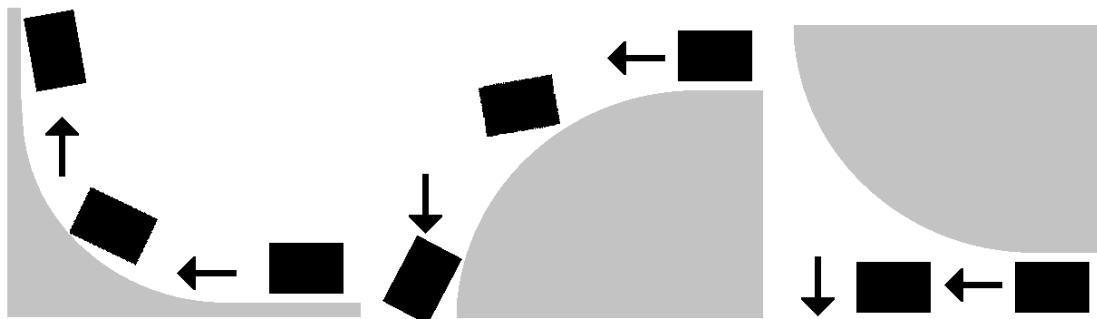


Figure 36: trayecto del agente sobre superficies inclinadas sin un corrector de dirección.

Aunque el método de ajustar la orientación de la hormiga sobre la superficie usando sus normales ayudaba con la no separación de la hormiga del terreno, había casos en los que los raycasts podían detectar terrenos con orientaciones muy distintas, lo que causaba cambios bruscos en su dirección. Su nueva orientación a su vez detectaría otros terrenos, por lo que cambiaría de nuevo bruscamente de dirección en el siguiente frame. El agente podía quedarse pillado ciclando entre dos orientaciones cada frame.

Para remediar este problema, se intentó acercar los raycasts. Si su distancia no fuera mayor a la longitud media de los ejes de los triángulos del terreno, no debería poder detectar más de 2 distintas en cada dirección. Esto funcionó y evitó que se quedara pillado cambiando de dirección, pero volvió a causar que el agente se quedara pillado en superficies cóncavas.

Se decidió en un compromiso final: reducir el tamaño de la cápsula de colisión del agente (figura 35 e). Con esto se sacrificó cierto realismo, ya que posteriormente los extremos de los modelos de las hormigas podrían atravesar el terreno más fácilmente. Sin embargo, en cuanto a funcionalidad, solucionó tanto el problema del agente quedándose pillado en superficies convexas como el problema del agente quedándose pillado en superficies irregulares al ajustar su dirección.

6.1.4 Centro de gravedad de la hormiga

Se quiso que la hormiga siempre se mantuviera boca arriba respecto a la superficie, y también considerar la posibilidad de que una hormiga cayera y aterrizará en su espalda. En otras palabras, la posición por defecto de la hormiga, fuera sometida a la fuerza de gravedad o a algún método para mantenerla sobre la superficie en la que se encontrara, debería ser con los pies hacia el suelo o dicha superficie. Para solucionar esto, se movió el centro de gravedad de la hormiga.

Aunque la posición del centro de gravedad es algo que se puede asignar dentro del programa de Unity por sí, se usó un script especial que permitiría ajustar de forma más visual la posición del centro de gravedad. La esfera azul clara en la figura 34 indica el nuevo centro de masa del cuerpo rígido: en vez del centro del objeto, ahora se encontraba en la parte inferior. Dada su nueva posición, ahora si la "hormiga" se encontrara bocabajo y se le aplicara una fuerza hacia una superficie, rotaría hasta alinearse con dicha superficie por su cuenta.

6.1.5 Sostener la hormiga sobre la superficie

Aunque con el cambio anterior la hormiga ya no se quedaría pillada en las superficies convexas, aún se podía desconectar de ellas al moverse por ellas a mayores velocidades. Como podemos ver en la figura 36, en el primer caso de subir una cuesta, la hormiga se mueve hacia la superficie y, por tanto, no llega a alejarse nunca. En el segundo caso, el agente eventualmente se separa del terreno si se mueve lo suficientemente rápido, pero en cuanto no detecta dicha superficie, cae hasta detectar de nuevo el terreno. En el último caso, donde un agente intenta subir una cuesta curvada desde debajo del terreno, al dejar de detectar el terreno, cae hacia abajo.

Hubo que encontrar una forma de hacer que el agente se acercara al terreno sobre el que se situaba. Se intentó hacer esto de tres modos:

1. Ajustando la posición del agente manualmente. Se puede modificar la posición del transform del objeto hormiga directamente. Esto causaba que la hormiga colisionara con el terreno, aun teniendo en cuenta su distancia a ella, ya que podía tener partes más cercanas a la hormiga que los raycast vieran. Los cambios repentinos de distancia también provocaron muchos movimientos rápidos y bruscos antinaturales.
2. Usando gravedad. Se puede modificar la gravedad para que empuje hacia la superficie debajo de la hormiga. Esta dirección se obtiene usando la dirección contraria a la media de normales detectadas por los raycast del agente. Este método conseguía que la hormiga se pudiera mover en superficies de cualquier dirección de forma natural, ya que, gracias al punto de gravedad nuevo, la hormiga siempre estaría boca arriba respecto a la gravedad. Sin embargo, este método no fue el que se llegó a usar, ya que tenía dos desventajas grandes:
 1. La fuerza que la gravedad aplica sobre la hormiga es demasiado lenta. Si el agente se movía a suficiente velocidad en una superficie convexa, llegaría a alejarse antes de que la gravedad pudiera alinearla con el suelo.
 2. La fuerza de gravedad es universal. Cambiar la dirección para un agente afecta a todos los demás.
3. Usar la función `AddForce()`. `AddForce()` es una función de Unity que permite aplicarle a un *GameObject* una fuerza en una dirección determinada. Como con la gravedad, gracias al punto de gravedad de la hormiga, la función `AddForce` causa que se alinee con la superficie de forma automática. Resuelve los dos problemas de usar gravedad:
 1. La fuerza de gravedad es una aceleración y, por tanto, afecta periódicamente más y más al *GameObject* hasta que este llegue a su velocidad de caída máxima. Esto afecta lentamente a la hormiga cuando se mueve, ya que cada vez que se aleja un poco de la superficie, tiene que volver a ir aumentando su velocidad. `AddForce()` en cambio afecta de forma uniforme a la hormiga, aplicando siempre el mismo grado de fuerza sobre ella. Esto lo hace consistente, y tras encontrar la fuerza necesaria que aplicar en cada `FixedUpdate`, funcionó sobre la gran mayoría de superficies.
 2. `AddForce()` se aplica individualmente sobre cada *GameObject*, mientras que la gravedad es universal. Cambios en la dirección de la fuerza aplicada sobre una hormiga no afectarían a los demás.

6.1.6 Rotar la hormiga para seguir la malla

Todos los cambios anteriores causaron que la hormiga pudiese seguir la gran mayoría de terrenos sin problema. Sin embargo, en los casos de picos en el terreno, la hormiga no es capaz de ajustarse lo suficientemente rápido para seguir el relieve. Para solucionar este problema, se decidió otra vez ajustar manualmente la rotación de la hormiga con la función *adjustAntToGround()*.

El primer método de ajustar la rotación de la hormiga probada fue ajustar la normal de la hormiga a la normal general de las superficies detectadas por sus raycast. Los cambios de orientación de la hormiga cambiarían la normal general, por lo que la hormiga no se quedaría quieta nunca.

Esta vez se decidió usar la distancia a la que los raycasts detectaran el terreno para decidir cuándo rotar. Se ajustaría la rotación de la hormiga solo si las diferencias de distancia entre raycasts adyacentes fueran notables. Por ejemplo, si los raycast del lado derecho de la hormiga detectaran lejos la superficie, mientras que los del lado izquierdo lo detectaran cerca, habría que rotar la hormiga hacia la derecha. La función *adjustAntToGround()* implementó este cambio manual usando los datos de las colisiones de la función *SenseGround()*.

Este ajuste manual ayudó a alinear la hormiga con el terreno en todos los casos, pero aún había picos de terreno en los que la hormiga no rotaba lo suficientemente rápido. Hubo que añadir una rotación manual más brusca, solo en los casos de estos picos. Para reconocer cuando la hormiga no rotaba lo suficientemente rápido para seguir el terreno, podría verificarse si su RayCast central viera terreno. En cuanto no lo viese, la hormiga rotaría hacia la misma dirección que los raycasts de sus esquinas que tampoco viesen el terreno. Se ilustra un ejemplo en la figura 37 (simplificado a dos dimensiones).

La aplicación de la rotación se implementó usando la función de Unity *MoveRotation()*. Este rota objetos en Unity de forma gradual y respetando las físicas del juego. Aplicar la rotación inmediatamente causaría colisiones abruptas con otras hormigas.

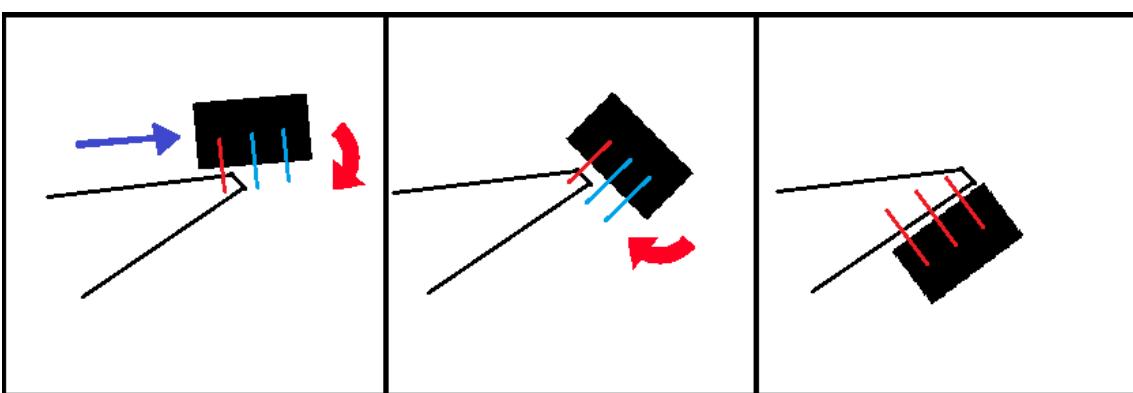


Figure 37: Esquema 2D de efecto de rotación manual sobre el agente. Lineas rojas y azules representan los raycast detectando y no detectando terreno respectivamente.

6.2 Interacción entre hormigas

6.2.1 Implementación inicial

A parte de las colisiones con el terreno, se tuvo que gestionar también las colisiones con los otros objetos dinámicos del entorno. Se optó por no incluir colisiones con pepitas de maíz y mazorcas, pero sí se quiso incluir colisiones con otras hormigas, para no tener casos en los que dos o más hormigas se encontrasen solapándose.

La implementación inicial de interacciones físicas con las otras hormigas fue simplemente usar el mismo colisionador esférico que se usaba para interactuar con el terreno, permitiéndolo colisionar con las capas del terreno y de las hormigas.

6.2.2 Hormigas caminando sobre otras hormigas

También se consideró la capacidad de hormigas caminando sobre otras hormigas. En la naturaleza esto pasa a menudo, existiendo incluso especies de hormigas que se sacrifican para formar puentes para el resto de la colonia. Se probó permitir a las hormigas detectarse con los RayCast que usaban para situarse sobre el terreno, pero causó un par de problemas difíciles de resolver:

1. Las hormigas podrían moverse con otras encima, alejándolas del camino o tarea que tuvieran que completar. Se podría evitar esto haciendo que las hormigas con otras hormigas encima no se movieran, pero si una hormiga se encontrase en una zona de mucho tráfico, podría quedarse pillada indefinidamente.
2. La forma esférica del colisionador de las hormigas dificultaba poder subirse el uno al otro.
3. Las hormigas podían acabar usándose entre sí como plataformas, empezando de esta forma a separarse del terreno y volar.

Se decidió, por tanto, no permitir subirse a otras hormigas. Se modificaron los *Raycast()* de la hormiga para que solo usaran la máscara de la capa del terreno, y se disminuyó la fuerza de rotación que las hormigas podían aplicarse entre sí para que no se pudiesen empujar de tal forma que una se quedara encima de otra.

6.2.3 Cambio de colisionador

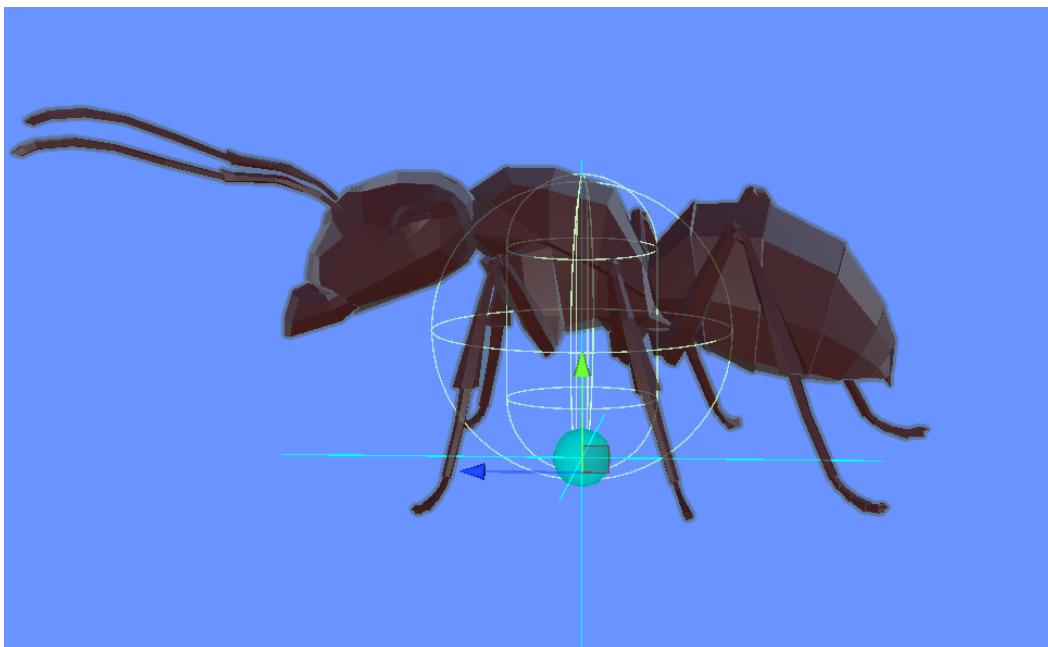


Figure 38: Los colisionadores de la hormiga. La esfera interactúa con el terreno, y la cápsula se usa para colisionar con otras hormigas. El círculo verde representa el centro de gravedad.

Debido al colisionador esférico de las hormigas, estos sufrían colisiones donde se empujaban hacia arriba y hacia abajo con la misma frecuencia que hacia los lados. Esto podía causar que las hormigas se separasen momentáneamente del terreno, interrumpiendo sus tareas y animaciones. Otro problema del colisionador esférico fue que las hormigas no se pudiesen acercar demasiado. En túneles estrechos, se podían quedar

pilladas dos o tres hormigas intentando salir del nido o entrar en una cámara.

Se decidió, por tanto, usar otro colisionador para las interacciones físicas con las otras hormigas. Se añadió un colisionador de cápsula a la hormiga, más pequeña que la esfera. Se modificó la cápsula esférica para solo colisionar con la capa de los objetos del terreno, y se asignó a la nueva cápsula la capa de los objetos hormiga. Esto resolvió ambos problemas adecuadamente.

6.2.4 Huevos de hormigas

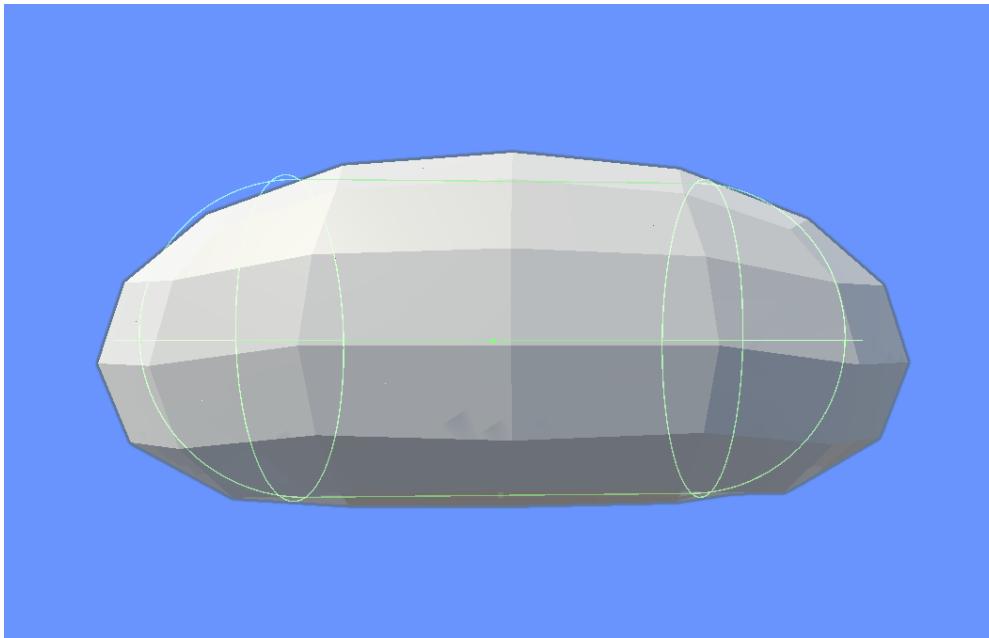


Figure 39: La cápsula de colisión con otros huevos del huevo

Mientras las hormigas se encontraran en el huevo, para que no se solaparan, se les añadió un colisionador de cápsula más grande para las colisiones con otras hormigas. Esto proporcionó un realismo adecuado entre huevos en las cámaras, pero causaba problemas de colisión con hormigas ya nacidas. Se decidió, por tanto, desactivar todas las colisiones entre hormigas y huevos. Esto se hizo asignándole una nueva capa a los objetos huevo del *prefab* de la hormiga, que solo colisionara con otros de su tipo y la capa del terreno. Los colisionadores de las hormigas con otras hormigas serían desactivados hasta nacer.

6.3 Interacción entre hormiga y otros objetos

Hubo una consideración de habilitar hasta cierto nivel las colisiones de las hormigas con los objetos pepitas de maíz y mazorcas del juego. La idea inicial era dejarlos empujarse, pero esto demostró causar una serie de complicaciones difíciles de resolver.

Por una parte, mazorcas empujables significaba que las hormigas podrían sin querer empujar una mazorca a un lugar difícil de llegar, o incluso lanzarla fuera del mapa, perdiendo permanentemente el recurso. Por otra parte, si se llegara a llenar una parte del nido de pepitas de maíz, estas podrían prevenir que las hormigas tocaran el suelo, separándolas del terreno y dejándolas en un estado de caída sin poder hacer nada más. Adicionalmente, cualquier intento de hacer que las hormigas tuvieran en cuenta los objetos del mapa para su pathfinding complicaría enormemente su funcionalidad y su capacidad de encontrar caminos de forma eficiente.

Se decidió por tanto desactivar las colisiones entre las hormigas y cualquier objeto que

no fuera ni el terreno ni otra hormiga.

6.4 Interacción de objetos llevados por las hormigas

6.4.1 Mientras son llevados por las hormigas

Después de la acción de recoger objetos, las hormigas eran capaces de mover pepitas de maíz y huevos de un lado del mapa al otro. Mientras llevasen estos objetos, podían quedarse atascados al colisionar el objeto con el terreno o empujar a otras hormigas con el objeto.

Se decidió, por tanto, desactivar todas las colisiones de los objetos en cuanto fueran recogidos por una hormiga y reactivarlos solo cuando fueran depositados.

En el caso de la pepita de maíz, esto se hacía destruyendo su componente Rigidbody en la función *SetToHoldCorn()*. En cuanto fuera depositado, se le asignaría un nuevo Rigidbody en la función *PlaceCorn()*.

En el caso del huevo de hormiga, no se quiso eliminar su Rigidbody, ya que este contenía ajustes específicos realizados en el editor, y el nuevo Rigidbody que se le asignaría en el código no tendría dichos cambios. En vez de esto, se le activaría el modo cinemático al Rigidbody en la función *SetToHoldAntEgg()*. Este modo se usa tradicionalmente durante animaciones dentro de los juegos, en los que las físicas y colisiones de dichos objetos son desactivadas. Al ser depositado, el modo sería desactivado en el Rigidbody del huevo en la función *PlaceAntEgg()*.

Este método es más sencillo que eliminar y asignar un nuevo Rigidbody, pero los colisionadores de los objetos en modo cinemático aún calculan colisiones (pero no las usan), por lo que se mantuvo el método previo en las pepitas de maíz para ahorrar carga de rendimiento.

6.4.2 Al ser depositados en el mapa

Aparte de reactivar las colisiones de los objetos de pepita y huevo al ser depositados por la hormiga en el nido, hubo que resolver un problema de colisión de dicho objeto con el terreno. La animación de depositación de la hormiga lo hace bajar la cabeza, y luego soltar el objeto donde se encontraba en sus mandíbulas. Debido a la forma dinámica del terreno y todas las posibles orientaciones de la hormiga, había muchos casos en los que la hormiga soltaría el objeto dentro del terreno, en vez de sobre él. Esto causaría que el objeto se cayera al vacío, perdiéndose el recurso de comida o el huevo de hormiga.

Para prevenir esto, en las funciones *PlaceCorn()* y *PlaceAntEgg()*, después de separar el objeto de la hormiga que lo lleva, se comprueba la posición del objeto. Si se encuentra debajo del terreno, es movido una fracción en la dirección del terreno repetidas veces hasta no encontrarse debajo de esta. La dificultad de este proceso es decidir en qué dirección se encuentra el terreno, para que el objeto no se mueva por dentro del terreno hasta salir de él en un lugar completamente distinto. Se decidió obtener la dirección hacia la superficie del terreno de la siguiente forma:

- Si la hormiga se encuentra dentro de una cámara del nido al depositar un objeto, la dirección hacia la superficie se considera el vector de dirección del objeto hasta el centro de dicha cámara.
- Si la hormiga no se encuentra dentro de una cámara, la dirección hacia la superficie se considera el vector de dirección del objeto hacia una unidad por encima de la hor-

miga, usando la orientación hacia arriba de la hormiga misma.

7 Sistema de movimiento de la hormiga

Uno de los aspectos más importantes del funcionamiento de la hormiga y de los primeros que se implementaron fue su capacidad de movimiento. Se definieron 3 acciones distintas de movimiento básico de la hormiga:

- Girar hacia la derecha o izquierda. Esto se haría a una velocidad de 67.5 grados por segundo, girando la hormiga sobre su eje y mediante la función *MoveRotation()*.
- Moverse hacia delante. Esto se hacía proyectando el vector hacia delante de la hormiga sobre el plano horizontal representativo de la superficie sobre la que se encontrara. El plano se crearía tomando la media de las normales obtenidas de sus 5 *raycast* sobre el terreno. Se movería hacia delante usando la función *MovePosition()*.
- Moverse hacia delante y girar a la vez. Esto se hacía aplicando ambas funciones *MoveRotation()* y *MovePosition()*.



Figure 40: La hormiga sobre el terreno y su objetivo en pequeño monte.

Después de implementar estos tipos de movimiento, hubo que definir cuándo aplicarlas. El gran obstáculo fue que había que traducir un objetivo en el entorno tridimensional a decisiones de movimiento bidimensionales para llegar a ella: la hormiga solo es capaz de moverse hacia delante, izquierda y derecha en su plano horizontal.

Se creó la función *FollowGoal()*. Dada la media de las normales de la superficie sobre la que se encontraba la hormiga y el objetivo en el espacio al que quiere llegar, esta función decidiría qué movimientos tomar proyectando el objetivo sobre el plano formado por la media de las normales. Sus pasos eran los siguientes:

1. Obtener el plano formado por la media de las normales obtenidas por los *raycast* de la hormiga que detectan la superficie debajo de ella. Se solía usar la orientación de la hormiga para decidir el plano, pero este no siempre se encontraba bien orientado sobre su superficie, y usar los datos de la superficie proporcionaba una decisión más informada sobre el terreno.

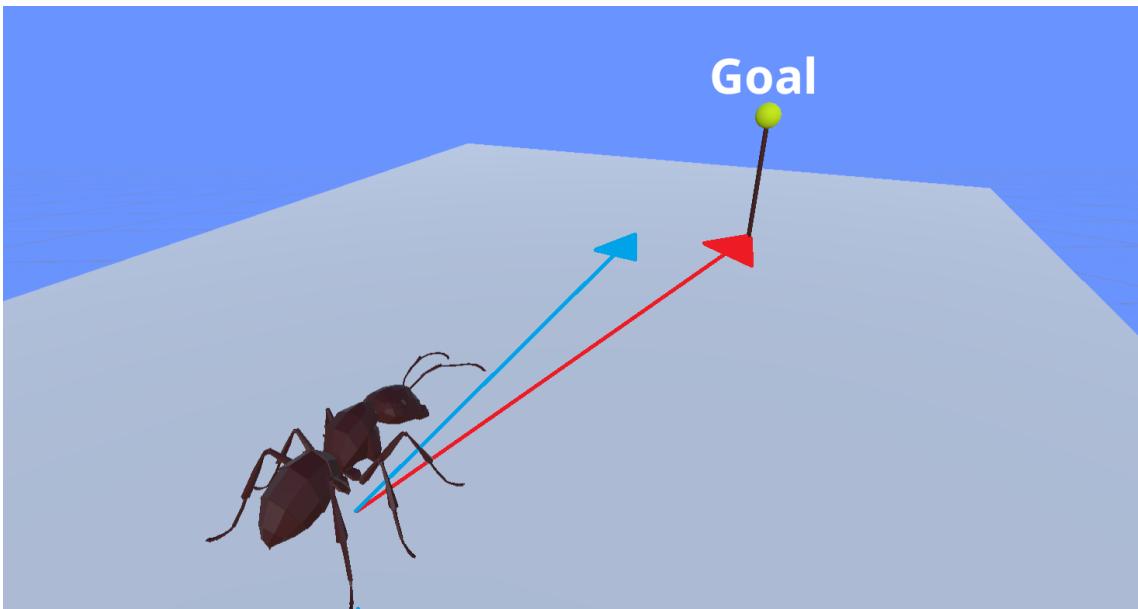


Figure 41: El plano de la superficie de la hormiga. Proyectadas sobre ella el objetivo, el vector hacia el objetivo (rojo) y el vector hacia delante de la hormiga (azul)

2. Obtener el vector de la hormiga al objetivo y proyectarla sobre el plano. También proyectar el vector hacia delante de la hormiga. Se usó la función *ProjectOnPlane()* de la clase *Vector3*.
3. Obtener el ángulo en grados entre los dos vectores proyectados mediante la función *Angle()* de la clase *Vector3*.
4. Dado el ángulo entre los dos vectores, decidir si girar o no y en qué dirección. Si el ángulo fuera mayor de 5, modificar la variable de giro del componente *Animator* de la hormiga según en qué dirección habría que girar. Esta dirección se obtenía mirando si el vector proyectado del objetivo se encontraba a la derecha o a la izquierda de la hormiga (línea azul en la figura 42). Dicha variable se usaría luego en la función de aplicar movimiento para girar en la dirección especificada.
5. Dado el ángulo entre los dos vectores, decidir si la hormiga puede avanzar. Si se encontrara dentro de cierto ángulo mínimo (entre líneas negras en figura 42) la variable de mover hacia delante del componente *Animator* sería activada. Si se encontraba fuera, se desactivaría. La función de aplicar movimiento posterior-

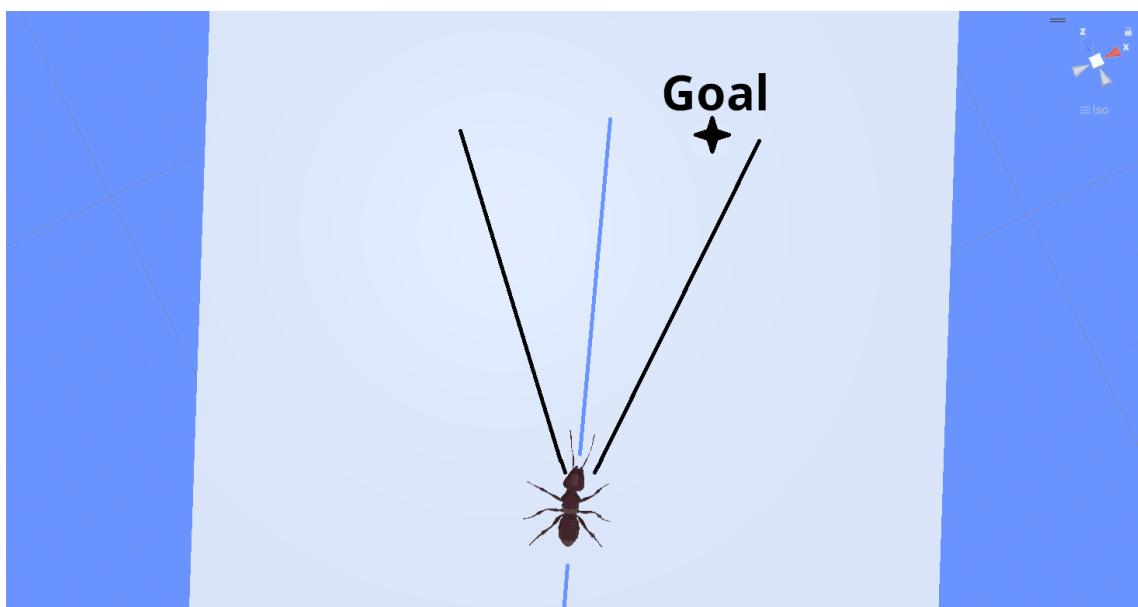


Figure 42: Vista desde arriba de objetivo proyectado sobre el plano de la hormiga.

mente miraría dicha variable para decidir si avanzar.

Este sistema funcionó bien para traducir el movimiento de la hormiga, pero fallaría rápido en cuanto se encontrara algún obstáculo entre la hormiga y su objetivo. La hormiga también era susceptible a quedarse pillada cuando el objetivo se encontrara justo por encima o debajo de ella. Fue necesario implementar un sistema de creación de caminos para la hormiga, la cual acabaría usando este sistema de movimiento para traducir sus pequeños pasos a movimientos.

8 Sistema de pathing de la hormiga

8.1 Qué es pathfinding

Pathfinding es un proceso mediante el cual una entidad en un mapa encuentra una ruta eficiente para ir de una posición inicial a un objetivo, evitando obstáculos y adaptándose a los elementos dinámicos del entorno del juego. Es necesario y usado en prácticamente cada juego que tiene entidades que se mueven por el mapa sin input directo del jugador. Juegos como Age of Empires son definidos en gran parte por su pathfinding, revolucionario para su tiempo.

El juego tendría hormigas moviéndose por el mapa, explorándolo, afectándolo constantemente. Hubo que encontrar una forma de implementar un sistema de pathfinding para poder habilitar este movimiento. Pero el gran obstáculo fue el terreno dinámico capaz de tomar cualquier forma.

8.2 Navmesh

En Unity se suele usar NavMesh para esto. NavMesh es una estructura de datos abstracta que ayuda a IA con pathfinding, y Unity contiene componentes que facilitan su uso. Funciona convirtiendo superficies del terreno en mallas especiales formadas por nodos y usando el algoritmo A* para encontrar caminos eficientes entre estos nodos. Además, tiene capacidades para evitar obstáculos móviles en sus trayectorias.

Aunque su implementación es simple, no es lo suficientemente versátil para gestionar el tipo de movimiento que se requerirá sobre una superficie como la del juego. Los dos problemas centrales son:

Se crean mallas especiales de navmesh usando mallas existentes, decidiendo qué partes son navegables. Las mallas del juego son múltiples y dinámicas, dificultando su implementación.

Navmesh asume que los agentes no pueden escalar paredes ni techos, que sus vectores up siempre apuntan hacia arriba. Ha habido implementaciones de navmesh sobre objetos grandes esféricos como planetas, pero requerían dividir el terreno en múltiples mallas interconectadas con vectores de arriba distintos. No era viable intentar encontrar una forma de distinguir qué partes del terreno se encontrarían bocabajo, laterales o boca arriba y dividir la malla según esas orientaciones.

8.3 Diseñar pathfinding y feromonas

Hubo que crear un sistema de pathfinding especializado para dejar las hormigas navegar por el terreno. Esto fue un proceso largo, en la que se crearon 2 versiones distintas de un sistema de pathfinding insuficientes, hasta culminar en una tercera versión final

apto y usable. La razón de la dificultad fue por causa de la naturaleza dinámica del terreno, y la capacidad de las hormigas de escalar virtualmente cualquier parte de este.

Hormigas, de por si, detectan solamente sus alrededores inmediatos. Como consiguen seguir caminos complejos en la vida real? Usando caminos de feromonas. Marcando por donde caminan al salir del nido, tienen una forma consistente de volver a este. Exploran, y cuando encuentran comida, vuelven al nido marcando con sus feromonas el camino de vuelta. Otras hormigas reconocen estas feromonas, y saben seguir las para recolectar la comida. Un camino pequeño se vuelve congestionado, cuando más y más hormigas se mueven por ella, reforzando aún más los caminos hacia fuentes de comida.

Se decidió implementar esta táctica de la naturaleza en el videojuego. No solo resolvía el problema de pathfinding, sino también daría másrealismo al comportamiento de las hormigas y la simulación de una inteligencia de colonia. Las tres versiones de pathfinding implementados se basaron en esto.

8.3.1 Primera versión: nodos de feromonas en puntos enteros

La idea era simple: las hormigas soltarían nodos de feromona donde caminaban, detectables por otras hormigas. Se colocarían sobre los puntos enteros del campo escalar más cercanos a por donde pasaban las hormigas. Cada nodo tendría un valor, indicando lo avanzado que estuvieran en su camino. Puntos cerca del nido tendrían valores menores, y los más lejanos, mayores. Las hormigas verían estas feromonas y se acercarían a ellas. Se implementó de la siguiente forma:

Se creó una clase nueva llamada *Pheromone*, unida a un objeto esfera sin colisiones para representar a los nodos.

La hormiga miraría la coordenada de enteros del campo escalar más cercano cada *FixedUpdate()*. Si no hubiese ya un nodo en ese punto, colocaría uno.

Cada hormiga contenía una referencia al objeto *Pheromone* que estuviera siguiendo.

Cada nodo usaría la función *Physics.OverlapSphere()* cada *FixedUpdate()*, que detecta colisiones con objetos dentro de un área designado. Se usaría con una máscara de capas para detectar solo hormigas, y si detectara una hormiga, se consideraría que la hormiga la oliera.

Al detectar una feromona, si la hormiga se estaba moviendo hacia valores mayores y el valor del nodo detectado fuera mayor que el del que la hormiga seguía, se reemplazaría con el detectado. El inverso, si la hormiga se movía hacia valores menores.

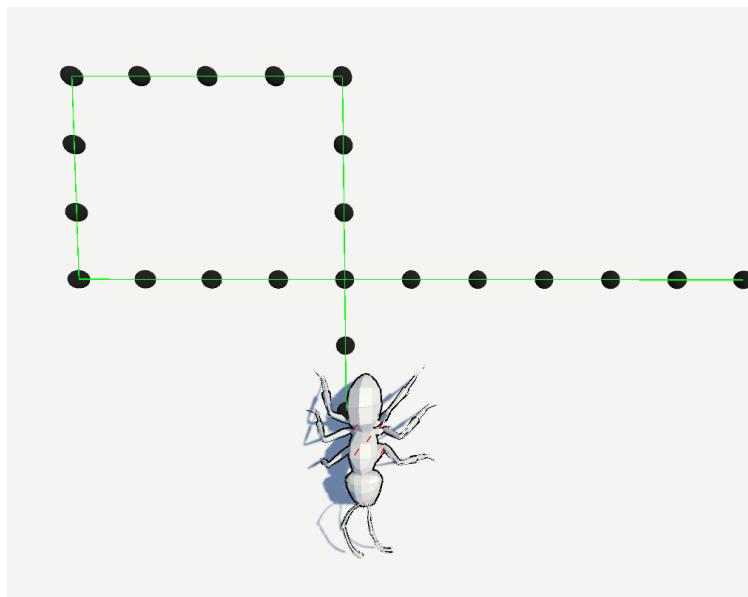


Figure 43: Una hormiga habiendo marcado un camino de nodos de feromonas

8.3.1.1 Cambiar el sistema de detección de feromonas

Tras ver que muy rápidamente habían más feromonas que hormigas, se decidió mover la función de detección de colisiones de la feromona a la hormiga, para que hubieran menos usos de este y minimizar impacto en el rendimiento. En vez de usar la función *OverlapSphere()*, se añadió un colisionador cuboide especial a la hormiga. Este colisionador fue modificado para solo colisionar con los objetos feromona y obtuvo el indicador *triggerCollider*. El flag causaría que no habrían colisiones, y que intersecciones entre el colisionador y una feromona llamaría las funciones *OnTriggerEnter()*, *OnTriggerExit()* y *OnTriggerStay()*. La gestión de detección y seguimiento de caminos se harían en usando estas, la hormiga teniendo constantemente una lista de feromonas dentro de su rango.

La forma del cubo de colisión fue modificada para mejor representar el área de detección de feromonas de la hormiga: se puso con el centro en la cabeza de la hormiga, ya que en la vida real las hormigas detectan las feromonas en parte con sus antenas. Se hizo también más ancho que alto, para prevenir detectar feromonas en superficies justo encima de la hormiga.

8.3.1.2 Representar caminos distinguibles

Para poder crear múltiples caminos, las feromonas obtendrían un número identificador correspondiente a un camino, junto con su posición en ese camino. De esta forma, si una hormiga quisiera seguir un camino determinado, solo tendría que notar su identificador e ignorar feromonas que no formaran parte del camino.

Las feromonas tendrían que poder formar parte de múltiples caminos, ya que sino los caminos se cortarían los unos a los otros. Se podría crear múltiples nodos de feromona en cada punto, uno para cada camino que pasara por él, pero aumentaría innecesariamente la cantidad de objetos en escena. Se decidió representar en un solo nodo múltiples feromonas distintas.

La versión inicial de esto fue añadir a cada objeto feromona una lista de tríos de integers. Uno representaría el id del camino. Otro la posición del nodo en dicho camino. El último representaría la edad de la feromona de dicho camino. Si una feromona llegara a su edad máxima, se eliminaría de la lista de tríos. Si la lista quedara vacío, el objeto

nodo de feromona mismo sería eliminado de la escena.

Después, se desarrolló más esta idea, creando una clase nueva llamada *PheromoneNode*. Esta sería la ligada a las esferas gameobject representativas de las feromonas, y contendrían una lista de objetos *Pheromone*.

8.3.1.3 Errores y soluciones

Esta versión, en su simpleza, demostraba varios problemas en testeо:

1. El camino de las hormigas era poco natural, moviéndose por los caminos cortos rectos entre feromonas adyacentes, girando 90 grados hacia feromonas laterales.

Para que hubiese más naturalidad en el movimiento, y para permitir que las hormigas tomaran pequeños atajos, se aumentó la distancia de detección de las feromonas. Ahora las hormigas, en vez de ver la siguiente feromona solo al llegar a la previa, la podrían ver de antes. Esto permitió que se giraran antes hacia feromonas laterales, y les hacía seguir el camino general en vez de las pequeñas secciones rectas que lo formaran.

2. El sistema de seguimiento de goles no funciona bien con goles justo encima o debajo de la hormiga. La caja de colisiones de la hormiga estaba diseñada para no ver feromonas encima o debajo de ella, pero si se encontrara una pared o una plataforma entre ella y la siguiente feromona, podía cambiar de orientación al treparla y quedar debajo o encima. Se quedaría pillado indefinidamente sin saber en qué dirección moverse (Figura 44).



Figure 44: La hormiga no sabiendo cómo moverse hacia la feromona marcada.

La solución a esto fue que la hormiga vigilara si su feromona objetivo acababa en cima o debajo de ella. En caso de que sí, elegiría otra feromona que siguiera entre las que se encontraran dentro de su rango.

3. Las hormigas podían llegar a una feromona y no ver la siguiente al encontrarse demasiado encima o debajo de la hormiga (figura 45).

Esto se remedio aumentando la altura de la caja de colisiones con feromonas de la hormiga, permitiendo que viera feromonas a un nivel por encima y por debajo de ella si se encontrara horizontal.

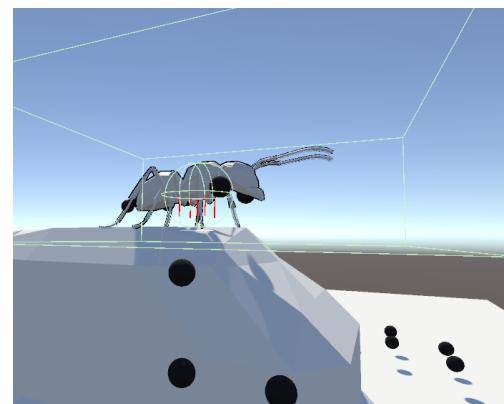


Figure 45: La hormiga incapaz de ver la siguiente feromona del camino

4. Con su caja de colisiones más vertical, la hormiga podía el siguiente nodo encima de ella, y no saber llegar. La hormiga seleccionaría otro nodo del camino que estuviese siguiendo dentro de su rango, pero las únicas otras alternativas serían nodos a los que ya llegó o detrás de ella, y se quedaría pillada ciclando eternamente entre nodos no aptos para seguir.

8.3.1.4 Veredicto final

No hubo solución real al último problema encontrado en testeо. Se podían seguir metiendo parches para intentar resolver escenarios específicos, pero considerando la naturaleza del terreno dinámico, no se podía saber con seguridad que se llegaría a cubrir todas. El problema fundamental fue que este sistema no le daba la información necesaria del terreno a la hormiga, dándole puntos a seguir, pero no cómo.

8.3.2 Segunda versión: feromonas sobre la superficie

La segunda versión del sistema de feromonas se creó con la intención de proporcionar más información a la hormiga sobre el terreno. Esta vez no se colocarían las feromonas en los puntos enteros del campo escalar, ignorando la variedad de formas que podía tomar el terreno. Se colocarían sobre la superficie misma, en las posiciones de las hormigas al soltar una feromona, con la misma dirección hacia arriba que la hormiga en el momento de colocarlo. De esta forma, las feromonas comunican información sobre el terreno que se cruza.

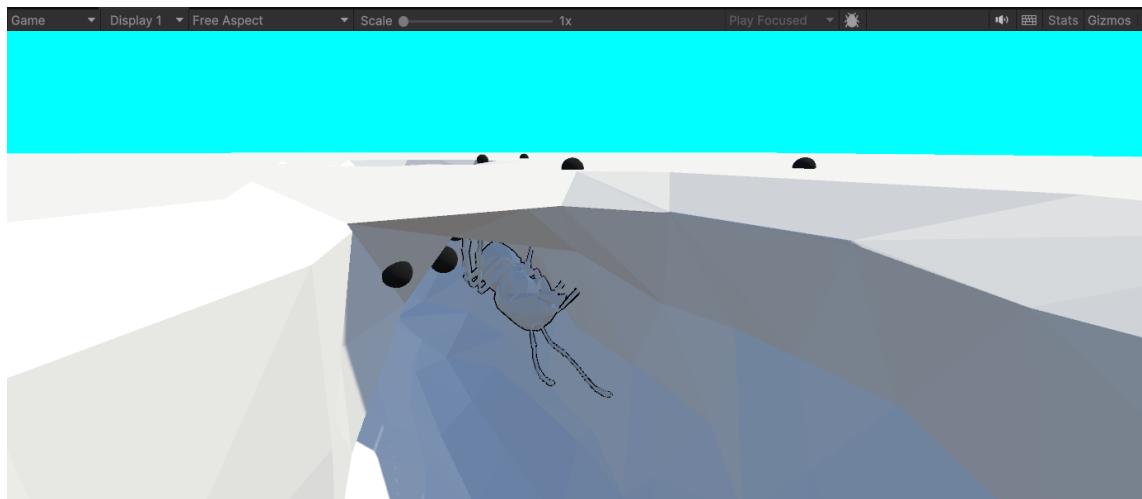


Figure 46: Una hormiga siguiendo feromonas en el sistema de feromonas sobre superficie

8.3.2.1 Cuando depositar feromonas

La idea principal es que las hormigas, al trepar por distintas ondulaciones del terreno, depositarían las feromonas sobre cada ángulo nuevo de este, marcando la forma del terreno. Para emular esto, se decidió que las hormigas soltaran las feromonas cada vez que el ángulo entre su vector arriba y el vector arriba de la última feromona que pusieran fuera mayor de 10.

Si la hormiga viaja por una sección del terreno plano, no cambiará de ángulo. Para evitar que no dejara feromonas en largas distancias, se implementó que la hormiga midiera su distancia a su previa feromona colocada. Si la distancia sobrepasara 3 ud de distancia, colocaría otra feromona. De esta forma, la hormiga coloca una feromona cada 3 ud de distancia o 10 ángulos de diferencia de orientación, el que venga primero.

8.3.2.2 Eliminación de PheromoneNode

Ya que las feromonas podrían encontrarse en cualquier parte del mapa, no tendría sentido asumir que dos o más se podrían encontrar en exactamente el mismo sitio. Se eliminó la clase PheromoneNode, y se usó la clase Feromona en los objetos esfera representativos de las feromonas.

8.3.2.3 Detección y seguimiento de las feromonas

Debido a la mayor potencial distancia entre feromonas, habría que aumentar bastante el área de detección de la hormiga para buscar la siguiente feromona. No se quiso aumentar tanto el área, ya que permitiría a la hormiga ver feromonas a distancias poco realistas. En vez de eso, se implementó que las feromonas de un camino tendrían referencias a la siguiente y previa feromona. Así, cuando una hormiga llegara a una feromona, obtendría inmediatamente el siguiente a seguir.

Ahora que las hormigas solo tendrían que usar su detección de feromonas cuando no tenían ya un camino a seguir, no tendría sentido tener un colisionador constante para encontrar las feromonas. Se decidió cambiar el sistema de detección para que la hormiga usara la función *OverlapSphere()* cada diez *FixedUpdates()* para encontrar las feromonas: no tan a menudo para bajar impacto en rendimiento, y no tan pocas como para perderse alguna feromona por su camino.

8.3.2.4 Llegar a una feromona

Inicialmente, se consideraba que una hormiga llegaba a una feromona al estar lo suficientemente cerca de ella. Esto causaba un problema: en terrenos de curvas repentinas, como la cima de una pared fina, la hormiga podía estar cerca de una feromona colocada por una hormiga previa sin estar en el mismo ángulo que la hormiga previa estaba en ese momento. Ese ángulo era necesario para que el sistema de seguimiento de goles funcionara bien y la siguiente feromona no se encontrara encima o debajo de la hormiga.

Hubo que asegurarse, pues, de que la hormiga siguiera acercándose lo suficiente a la feromona hasta tener un ángulo similar al de la hormiga previa. Para ello, se implementó que solo se considerara que la hormiga llega a una feromona si su ángulo de arriba es similar al de la hormiga previa: al colocar una feromona, giró para tener el mismo ángulo de arriba que la de la hormiga al colocarla. Otra hormiga compararía su ángulo de arriba con el de la feromona, respectivamente.

Para cuando se implementó este sistema de feromonas, la hormiga ya tenía las funciones de ajuste a la orientación del terreno. Aun así, al moverse hacia delante, tardaría un momento en tomar la orientación correcta. Las feromonas que colocaba, por tanto, tendrían ángulo hacia arriba no representativo del vector normal del terreno. No mostraba ser un problema al seguir caminos en la misma dirección en la que fueron colocadas, pero podía provocar que las hormigas se quedaran pilladas, nunca alineándose con las feromonas al seguir los caminos al revés.

Se decidió arreglar ese fallo haciendo que las hormigas usaran el vector normal de la superficie sobre la que colocaron cada feromona para ver si colocar una nueva feromona y como vector arriba de dicha feromona. Esto ayudó, pero aún había ciertos casos donde los caminos no se podían completar, sobre todo al salir de nidos.

8.3.2.5 Feromonas auxiliares

8.3.2.6 Veredicto final

A pesar de los problemas que este sistema soluciona, la naturaleza impredecible del terreno dinámico y las posiciones elegidas para poner feromonas seguían creando casos en los que las hormigas intentaban llegar a feromonas detrás de una pared, debajo o encima de ellas, o requiriendo un ángulo no reproducible al acercarse desde ciertos puntos o al haber sido empujadas por otras hormigas.

Hubo un largo periodo de parches y testeos, pequeños cambios para intentar resolver casos nicho de hormigas quedándose pilladas, pero no hubo una solución universal. La hormiga no tenía suficiente información sobre el terreno sobre el que viajaba. Es entonces que se decidió empezar de nuevo, abandonar este diseño y crear uno que tuviera en cuenta todas las formas posibles del terreno.

8.3.3 Versión final: adyacentCubes

Este sistema de pathfinding responde a todos los problemas con las pasadas versiones: La falta de percepción de la forma real del terreno. Se podría decir que las hormigas se habían estado moviendo por el mapa a ciegas, solo sabiendo que se encontraban sobre una superficie y que se tenían que mover hacia una coordenada. No hubo pathfinding real, ya que no se buscaban caminos, sino que se intentó reutilizar los que las hormigas realizaron aleatoriamente en su exploración.

El sistema de adjacentCubes fue creado para tomar ventaja del fundamento sobre el que se formaba el terreno y dividir este en “casillas”. Pequeñas secciones de superficie interconectadas en un grid tridimensional. Cada cubo de marcha, por así decirlo, contiene una de 256 posibles combinaciones de triángulos. Cada superficie de una combinación determinada siempre conectará con los mismos cubos adyacentes. El sistema adjacentCubes se centra en poder obtener casillas de superficie adyacentes, permitiendo de esta forma trabajar con algoritmos clásicos de búsqueda de camino sobre el terreno del juego.

8.3.3.1 CubeSurface

El primer paso de la implementación de este diseño es la representación de las partes más pequeñas del mapa: la superficie dentro de un cubo de marcha. Se creó inicialmente una estructura, pero problemas de duplicación de datos en vez de pasar por referencia hicieron que se cambiara a una clase.

Para poder representar una superficie, se necesitarían dos cosas:

- La posición del cubo que contiene la superficie
- Un array de 8 booleanos que representan los vértices del cubo. True si se encuentran debajo de la superficie, false si se encuentran fuera de la superficie.

Hay combinaciones de Marching Cubes en las que se encuentran múltiples superficies dentro de un mismo cubo. Es por esto que el segundo elemento es necesario, para especificar a cuál de las superficies se refiere.

Para crear un objeto cubeSurface se necesitan dos cosas: la posición del cubo y un vértice del cubo que se encuentre debajo de la superficie. Con estos, la nueva función GetGroup() obtendría el array de booleanos que define la superficie.

La función CornerFromNormal(), dado un cubo y la normal de una de sus superficies, podía obtener uno de los vértices del cubo debajo de dicha superficie. A partir de esta, se podría obtener el array de booleanos de la superficie con GetGroup(). Esto permitiría a las hormigas obtener la superficie sobre la que se encontraban usando el vector normal obtenido de sus raycasts.

8.3.3.2 Obtener superficies adyacentes

Aunque la primera versión de este sistema solo devolvía los cubos adyacentes, lo que definió su nombre, posteriores versiones obtendrían sus cubeSurface adyacentes. Esto es lo que permitió el funcionamiento de los pathfinders clásicos como el A*.

Para determinar con qué cubos adyacentes conecta un cubeSurface, basta con mirar la cara que conecta los dos cubos. Si la cara es cortada por la superficie, dicha superficie conecta los dos cubos. Que la cara sea cortada significa que dos de los lados que forman la cara son cortados por la superficie. Que alguno de los lados sea cortado significa que, de los vértices que lo forman, uno se encuentra sobre la superficie del cubo y el otro debajo. Por tanto, si una cara muestra tanto vértices debajo de la superficie como vértices por encima de la superficie, es cortada por la superficie.

Usando esta deducción, se creó la función FaceXor(). Dado el grupo de booleanos del cubeSurface y el vector de dirección del centro del cubo hasta la cara que conecta con el cubo del cual se quiere comprobar la adyacencia, realiza la función lógica XOR sobre los vértices de la cara. Si todos los vértices se encuentran debajo o encima de la superficie, devuelve false y la superficie no conecta con el otro cubo. En caso contrario, sí conecta.

A la clase Chunk se añadieron las tablas faceIndexes (dado el índice de la cara, devuelve los índices de los 4 vértices que lo forman) y faceDirections (dado el índice de la cara, devuelve la dirección de dicha cara). FaceIndexes ayudó a simplificar mucho la función TrueCorners, permitiendo acceso eficiente a los vértices. FaceDirections sirvió para obtener las posiciones de los cubos adyacentes, sumando la dirección obtenida a la posición del cubo inicial.

Para obtener también el objeto cubeSurface del cubo adyacente, se creó la función TrueCorner(). Esta, dado el grupo de booleanos de un cubeSurface y el índice de la cara del cubo que conecta con el adyacente, devuelve (si hay), del cubo nuevo, un vértice en la cara que conecta los dos cubos que se encuentre debajo de la superficie. Estos vértices se pueden usar para obtener el objeto cubeSurface del nuevo cubo.

8.3.4 CubePheromone: caminos de feromona

Con la implementación de *adyacentCubes*, se programó un nuevo sistema de caminos de feromonas tomando elementos de ambas versiones previas. Se crearon las clases *CubePaths* y *CubePheromone*:

- *CubePheromone* representaría una feromona. Contenía 3 variables para identificarlo:
 - La posición Vector3Int del cubo que contenía la feromona.
 - El id del camino del que la feromona formará parte.
 - Un array de 8 booleanos para identificar a la superficie del cubo sobre la que se encontraría la feromona. Más tarde, este y la posición del cubo fueron reemplazados con un objeto *cubeSurface* para simpleza.
- *CubePheromone* también contendría referencias al siguiente y al previo cubePheromone en su camino.
- *CubePaths* se encargaría de gestionar todos los caminos de feromonas. Contenía dos diccionarios estáticos:
 - Un diccionario llamado *cubePherDict*, indexado por las posiciones de los objetos cubePheromone. Cada entrada contenía la lista de cubePheromone dentro del cubo respectivo, ya que podría haber múltiples feromonas en la misma posición. Contendría todas las feromonas del mapa para fácil acceso.
 - Un diccionario llamado *pathDict*, que contendría todas las primeras fero-

monas de cada camino. Fue indexada por los identificadores de los caminos, conteniendo el primer objeto cubePheromone de este.

La idea era que las hormigas tendrían en su memoria un grupo de identificadores de caminos a comida, que las hormigas compartirían con los demás. Cualquier hormiga que entrase al nido también sería informada, y si un camino empezaba a carecer de comida, esta información también sería compartida con los demás. Esta idea no llegó a desarrollarse antes de la simplificación del sistema.

8.3.4.1 Creando caminos en CubePheromone

Se creó la función PlacePheromone() en la clase CubePaths. Este, dada una posición y un objeto CubePheromone, metería dicha feromonía en la lista de feromonas en esa posición en el diccionario cubePherDict. Si no había ya una entrada en el diccionario en esa posición, se crearía una nueva. Si ya había una entrada en el diccionario, y ya había una feromonía del mismo camino en ello, sería reemplazada por el nuevo.

Se añadieron dos funciones para que la hormiga pudiese marcar superficies con feromonas:

- *ContinuePheromoneTrail()*, que dada la posición del cubo y el objeto CubePheromone previo, obtendría la superficie del cubo nuevo que conectara con la superficie previa. Crearía un nuevo CubePheromone como extensión del camino (mismo id de camino y con feromonía previa igual al original), y lo colocarían en el mapa con *PlacePheromone()*.
- *StartPheromoneTrail()*, que dada la posición del cubo y la normal de la superficie sobre la que colocar la feromonía, crearía un nuevo objeto CubePheromone con un id de camino nuevo y lo colocaría tanto en *cubePherDict* con *PlacePheromone* como en *pathDict* como comienzo de un nuevo camino.

La hormiga misma, mientras soltara caminos, vigilaría que al cambiar de cubeSurface, este nuevo se encontraría adyacente a la feromonía colocada previamente mediante la nueva función DoesSurfaceConnect(). Si se conecta, usaría la función de continuar el camino de feromonas. Si no, empezaría una nueva.

Se notó que las hormigas a veces podían moverse diagonalmente entre cubos, por lo que empezarían un nuevo camino no conectado al previo. Para remediarlo, si una hormiga tenía una feromonía previa pero no se encontraba adyacente a ella, una función buscaría en un radio de dos cubeSurface alrededor de la hormiga. Si una de las superficies contenía la feromonía previa, se colocarían feromonas entre esa y la posición nueva de la hormiga sin empezar un camino nuevo.

8.3.5 La simplificación del sistema de feromonas y implementación de edad

El sistema de feromonas de cubePheromone requería mucha gestión de comparación de información entre las hormigas y diferenciación de los distintos caminos. También hay que marcar qué caminos aún tienen comida al final, y cuáles se conectan con otros caminos y si merece la pena separarse de uno para seguir otro.

Se decidió, por tanto, bajar la complejidad del sistema. Ahora las hormigas dejarán feromonas en las coordenadas de los cubos por los que pasan. Luego, cuando una hormiga quisiera moverse a un lugar, usaría un algoritmo de búsqueda de caminos para intentar llegar al lugar objetivo. El algoritmo se expandiría sobre casillas dentro del nido (la hormiga conoce el nido y sabe moverse por él) y casillas con feromonas.

Se eliminó la clase *CubePheromone*. Se añadió un diccionario de enteros indexados por *Vector3Int* y se eliminaron los diccionarios previos. El diccionario nuevo se llamó *cubePheromones* y representaría las casillas que tuvieran feromonas (si hay una entrada en un *Vector3Int*, su cubo contiene feromona) y sus edades (los enteros representan la edad de cada feromona).

La función de depositar feromonas se simplificó mucho. Ya no hizo falta diferenciar entre comenzar y continuar un camino. Simplemente se pondría el valor de la entrada posición de la hormiga en el diccionario a 100: *cubePheromones[pos] = 100*. Esto crearía una nueva entrada si no había ya una, y actualizaría el valor de la entrada si ya existía.

La edad de las feromonas se gestionó en la clase *CubePaths*. Ya que es una clase estática, su función *FixedUpdate()* es llamada sí o sí una sola vez cada *fixedTimeDelta*. Esta consistencia se usó para poder asegurarse de que las feromonas siempre tardarían la misma duración:

En cada *FixedUpdate()* se añade el valor *fixedDeltaTime* a un contador.

Si el contador es mayor o igual a 3 (segundos), se pone a 0 y se itera sobre cada feromona en el diccionario.

A cada feromona iterada se le resta 1 al valor.

Si el valor de la feromona llega a 0, se elimina del diccionario.

No se itera sobre las entradas del diccionario mediante *foreach*, ya que este no permite la edición de la estructura de datos iterada. En vez de eso, primero se obtiene la lista de claves del diccionario, y después se itera sobre esa lista, accediendo a las entradas del diccionario mediante las llaves.

Se eligieron 3 segundos para que la duración de una feromona nueva fuera 300 segundos, es decir, 5 minutos exactos. De esta forma se puede ver en tiempo real cómo caminos no reutilizados dejan de existir.

8.3.6 Usar seguimiento de goles con *cubeSurface*

La hormiga, para moverse hacia lugares, usaría el sistema de seguimiento de goles. Por tanto, hubo que decidir dónde situar los goles para moverse de una superficie a otra. Se escribió la función *GetMovementGoal()*, a la que se le pasa el *cubeSurface* sobre el que se encuentra la hormiga y la dirección hacia el siguiente *cubeSurface*. La función devuelve el objetivo a seguir para que la hormiga llegue al siguiente *cubeSurface*.

Los pasos que sigue son:

La función comprueba que la dirección es hacia un cubo adyacente. Si no, devuelve el punto en el centro del cubo actual más la dirección dada. Esto sirve para prevenir trabajar con cubos no adyacentes, pero aún crear una solución plausible en caso de un error.

Se pone el valor del gol al que ir al centro del cubo actual y se le suma la dirección hacia el siguiente cubo multiplicado por 40. Esta distancia extra servirá para alejar más el gol de la hormiga al proyectarlo sobre el plano en seguimiento del gol.

Mira la cara compartida entre los dos cubos. Tiene en cuenta los vértices de esta que se encuentren debajo de la superficie, y añade la distancia del centro del cubo hacia ellos al gol. Esto se hace para evitar casos donde el terreno corta casi horizontalmente los dos cubos y el gol se encuentra prácticamente encima de la

hormiga.

8.3.6.1 Error de la hormiga moviendo entre dos cubos

El seguimiento del gol permite que la hormiga empiece a moverse hacia él antes de estar totalmente alineada con él. Esto se hace para que el movimiento sea más natural y menos rígido, permitiendo a la hormiga acercarse usando una curva. Una consecuencia de esto es que la hormiga, intentando moverse al siguiente cubo del camino, se mueva al lateral del actual. Al llegar a este nuevo cubo, intenta volver al camino, moviéndose hacia el que justo ha dejado. A menudo, gracias a la forma del terreno, esto causa que la hormiga gire, vuelva al cubo, se mueva, vuelva al lateral, se gire, etc. Creando un bucle del que tarda en salir, si es que lo consigue.

Para prevenir esto, se decidió posicionar el gol teniendo en cuenta no solo el siguiente cubo, sino el que viene después (si es que hay un paso más en el camino). Así, la hormiga, al llegar al cubo lateral, puede seguir moviéndose hacia delante mientras vuelve al cubo. Para implementar esto, se creó una versión de `GetMovementGoal()` que tomara como entrada, además de la superficie actual y la dir al siguiente cubo, la dir del siguiente cubo al posterior.

Esto arregló el problema, pero causó otro. En ciertos casos, la dirección del segundo paso hace que la hormiga no siga el camino correctamente. Causa que la hormiga se aleje de la pared que debe subir (figura 47) o del eje que debe circular. Después de buscar una forma de identificar estos casos, se obtuvo lo siguiente:

1. Se toma la cara que conecta el cubo de la hormiga con el siguiente (cara1) y la cara entre el cubo siguiente y el cubo posterior (cara2).
2. Se seleccionan, si hay, los dos vértices de la cara 2 que se tienen en común con la cara 1.
3. Si los dos vértices están debajo del terreno y los otros dos de la segunda cara no, o están encima del terreno y los otros dos dentro, se da el caso donde la hormiga se quedaría pillada.
4. En dicho caso, no se toma en cuenta la segunda dirección para colocar el gol.

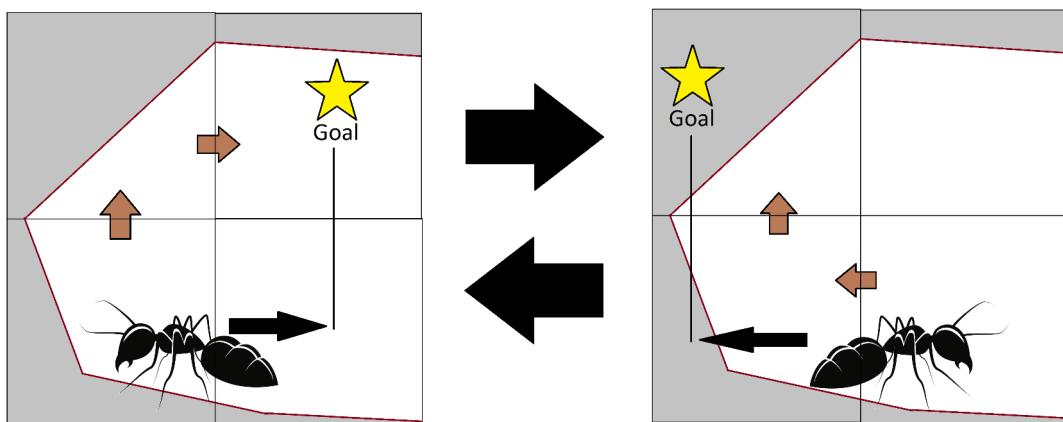


Figure 47: Bucle en el que se puede quedar pillado la hormiga

8.4 La implementación de algoritmos de búsqueda de caminos

Gracias al nuevo sistema de *AdyacentCubes* y la habilidad de la hormiga de seguir caminos de *CubeSurface*, fue posible implementar algoritmos de pathfinding en el terreno

dinámico de Marching Cubes. Los dos algoritmos implementados fueron la búsqueda en anchura y A*.

8.4.1 Búsqueda en anchura

La búsqueda en anchura, conocida como breadth-first search en inglés, es un algoritmo de búsqueda no informado para encontrar elementos o recorrer grafos. Comienza en el *CubeSurface* sobre la que se encuentra la hormiga y se exploran todos los vecinos de este. A continuación, para cada uno de los vecinos, se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el mapa o se llegue al límite de distancia de búsqueda. Se denomina no informado porque no usa ninguna estrategia heurística para su búsqueda.

Técnicamente, no es un algoritmo de búsqueda de caminos, ya que el diseño por defecto no obtiene el camino hacia el objetivo encontrado. El algoritmo fue implementado de forma que se pudiera encontrar el camino hacia el punto encontrado, usando un diccionario en el que cada *CubeSurface* alcanzado tiene asociado el *CubeSurface* previo por el que se llegó. De esta forma se puede obtener el camino más corto mediante el que se llega al objetivo encontrado.

El algoritmo se implementó en varias funciones usando la función *GetNextSurfaceRange()*. Se trata de una función que, dada una frontera de superficies y una colección de las superficies ya visitadas, devuelve la siguiente frontera y actualiza los valores de la colección de visitados para ser reutilizada en la obtención de la siguiente frontera.

GetNextSurfaceRange() devuelve una lista de *CubeSurface* que representa la siguiente frontera, y toma los siguientes argumentos:

- Un *CubeSurface* con la superficie inicial de la hormiga. Se usa para obtener el rango inicial en la primera llamada de la función.
- Un *Vector3* con la dirección en la que mira la hormiga. Se usa para expandir la frontera priorizando la misma dirección que la hormiga, para que se obtengan caminos en los que esta no gira innecesariamente.
- Una *List<CubeSurface>* llamada *CurrentRange*, lista de superficies que representa la frontera actual desde la que se expande.
- Un *Dictionary<CubeSurface, CubeSurface>* llamado *CheckedSurfaces*, diccionario usado tanto para ver qué superficies ya se han alcanzado y no deben ser añadidas a la frontera y por qué superficies se pasa para llegar a las siguientes.

Por cada superficie de la frontera dada, se obtienen sus superficies adyacentes. Los nuevos obtenidos que no se encuentren en *CheckedSurfaces* son añadidos a este como claves con los valores siendo la superficie padre. También son añadidos a la lista del siguiente rango que la función devolverá.

Esta función se usa en diversas otras, tanto para encontrar un camino hacia un objetivo cercano (cuando la hormiga se aleja del camino que está siguiendo, para volver a este) como para elegir a qué superficie, a un cierto rango de casillas, ir mientras se explora.

8.4.2 A*

A* es un algoritmo de búsqueda en grafos informado. Esto significa que sí usa una heurística para guiarse por el grafo. Algoritmos que usan heurísticas tienen la tendencia de no encontrar los caminos más óptimos. A* se diferencia de estos al usar una combinación del valor heurístico y el coste de desplazarse al nodo para decidir el siguiente nodo

a explorar, mediante la evaluación $f(nodo) = \text{coste}(nodo) + \text{heurística}(nodo)$.

A* mantiene dos estructuras de datos auxiliares, que podemos denominar *abiertas*, implementadas como una cola de prioridad (ordenada por el valor $f()$ de cada nodo), y *cerradas*, donde se guarda la información de los nodos que ya han sido visitados. En cada paso del algoritmo, se expande el nodo que esté primero en abiertos y, en caso de que no sea un nodo objetivo, calcula la $f()$ de todos sus hijos, los inserta en abiertos y pasa el nodo evaluado a cerrados.

Se implementó el algoritmo usando como nodos los *CubeSurface*. Se usó la estructura de datos importada *PriorityQueue* para representar la estructura de datos *abiertos*, que toma como entradas dos valores: la superficie a guardar y su valor $f()$. Se usó otra vez un diccionario llamado *previo* para guardar las superficies previas de cada superficie alcanzada y así obtener el camino óptimo encontrado a la superficie objetiva.

El A* se implementó en múltiples funciones de búsqueda:

- *GetPathToPoint()* para encontrar caminos de una superficie a un punto. Puede buscar por cualquier superficie, y se le pasa como argumento el límite de distancia a la que puede buscar. Se usa cuando la hormiga decide moverse hacia un punto cercano, y el límite de distancia representa el rango de visión de la hormiga.
- *GetKnownPathToPoint()* es similar a *GetPathToPoint*, pero se distingue en no tener límite de distancia. En vez de eso, el algoritmo A* solo puede expandirse sobre superficies dentro del nido y superficies con feromonas. Esto se hace para representar cómo las hormigas saben llegar a fuentes de comidas encontradas por otras hormigas dejando caminos de feromonas, y cómo las hormigas saben moverse por su nido.
- *GetPathToSurface()* es similar a *GetPathToPoint*. La única diferencia es que no busca una superficie cerca de un punto en el espacio, sino que busca llegar a una superficie específica.
- *GetKnownPathToMapPart()* también solo se expande por superficies dentro del nido o con feromonas, pero en vez de buscar un punto específico, busca una parte del nido determinado. Este puede ser una cámara, un túnel o la entrada del nido.

9 Sistema de excavación

9.1 División de excavación en pequeñas tareas

Una de las dificultades iniciales del proyecto fue encontrar una forma de hacer que las hormigas pudiesen modificar el terreno, excavándolo para formar los nidos. Hubo que encontrar un método de dividir la excavación de un túnel o área en pequeños pasos compatibles entre hormigas, además de hacer que dicha excavación representara adecuadamente el área designada a excavar por el jugador.

Una de las ideas iniciales fue usar la función de modificación del terreno que se usaba para editar mapas en el modo creación de mapas. Este podía añadir o quitar valores del campo escalar en un área esférico. Se consideró que las hormigas usarían la función repetidas veces dentro de las áreas de excavación con un tamaño de esfera pequeña en las coordenadas de sus mandíbulas. Sin embargo, era impreciso y la implementación sería sumamente difícil si se quería asegurar que la forma excavada se ajustara bien a

la designada.

La solución al problema se encontró al considerar la simpleza del campo escalar. Excavar un área implica modificar los valores del campo escalar contenidos en el área y recargar la malla del terreno. Por tanto, podemos dividir el trabajo de excavación en pequeñas partes siguiendo los siguientes pasos:

En cada coordenada entera del área a excavar se crea un objeto especial.

Cada uno de los objetos contiene el valor que debe tener el campo escalar en su coordenada para representar el área excavada.

Las hormigas pueden interactuar con los objetos. Si lo hacen, el punto es excavado: el valor de su posición en el campo escalar es modificado y la malla de terreno del chunk(s) que contiene el punto es recargada para reflejar el cambio. Luego, el objeto es eliminado.

9.2 Desarrollo de DigPoints

Se creó la clase *DigPoint*. Este representaría una tarea de excavación y contendría el valor con el que debería acabar el punto del campo escalar en el que se encontrara. Cuando una hormiga interactuase con él, editaría el campo escalar, mandaría a recargar el chunk en el que se encontrara y sería eliminado.

Durante el desarrollo del juego, antes de llegar a usar partículas, se usaba una esfera blanca como objeto de juego para representar cada *DigPoint*. Durante las pruebas iniciales, fue evidente que inicializar todos los *DigPoint* de un área fue excesivo (figura 48). No solo podía bajar bastante el rendimiento la enorme cantidad de objetos nuevos, sino que también causaba que las hormigas pudieran intentar excavar puntos más dentro de la superficie. Esto último podría causar excavaciones no realistas, e incluso hacer que la hormiga se quede pillada intentando llegar a un punto de excavación fuera de su alcance.

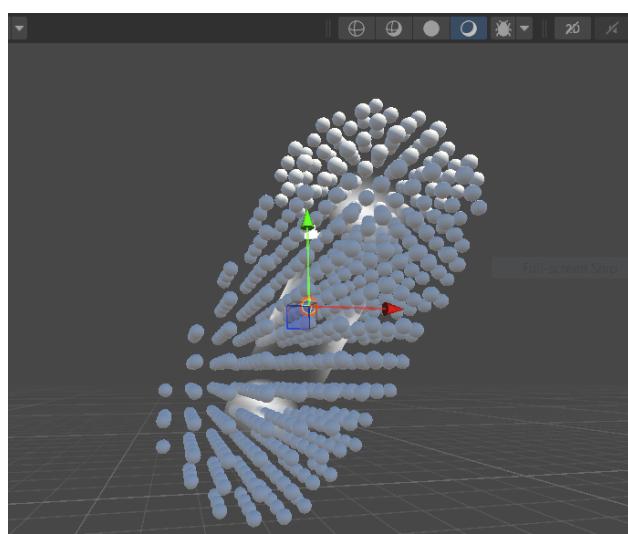


Figure 48: Primer implementación de *DigPoints* que inicializa objetos en todos los puntos del túnel.

Hubo, por tanto, que distinguir entre los puntos justo debajo de la superficie y los más profundos y solo inicializar los primeros. Se decidió solventar esto creando un nuevo sistema:

- Se añadió un diccionario estático a la clase *DigPoint* que usaría las posiciones del campo escalar como claves. El valor de cada posición sería una tupla que

contendría el valor a cambiar del terreno, y una referencia al *DigPoint* inicializado en dicha posición (null si no se inicializó aún).

- Cuando un área es convertida a datos de excavación, antes de añadirlos a los diccionarios, se mira cada punto. Si un punto que hay que excavar se encuentra adyacente a la superficie, se inicializa *DigPoint* en sus coordenadas y se guarda la referencia al nuevo objeto en el diccionario.
- Cuando una hormiga excava un *DigPoint*, se miran los puntos adyacentes. Los que se encuentran en el diccionario y no han sido inicializados, se inicializan.

De esta forma se mostraría el mínimo número requerido de puntos de excavación, a la vez que, conforme las hormigas excavaran, todos los demás irían apareciendo hasta completar la parte del nido indicada.

Más tarde, para simplificar el funcionamiento y mejorar la comprensión del código, se intercambió la tupla valor-referencia por una estructura llamada *DigPointData*. Este contenía los dos valores de la tupla: el valor nuevo del terreno y la referencia a su *DigPoint* si fuera inicializado. Más tarde aún se cambió la estructura por una clase, ya que las estructuras no se pasan por referencia en C# y causaban problemas de modificación de sus datos. Con esto, el diccionario de todos los puntos de excavación ahora contenía los objetos *DigPointData* indexados por sus posiciones.

9.3 Conversión de áreas del nido a DigPoint

Cada vez que el jugador designa un área nueva del nido, esta debe descomponerse en todos los puntos de excavación necesarios para formarlo. Los métodos de selección del área y los tipos de partes del nido se desarrollan en la sección del nido más adelante. Lo único que importa para los puntos de excavación son los dos tipos de formas que pueden tener las áreas:

- Un túnel en forma de tubo, formado por dos esferas y un cilindro que los conecta.
- Una cámara en forma de elipsoide, formada por una esfera con dimensiones modificadas.

Cuando el jugador coloca uno de estos, se ejecutan dos pasos:

Usando la función *pointsInDigObject()*, se obtiene un diccionario de *DigPointData* que representan el área.

En la función *toDigPoints()*, el diccionario general estático de *DigPointData* es actualizado con los valores del diccionario nuevo: las entradas existentes son actualizadas, las nuevas introducidas y los nuevos *DigPoints* que hacen falta ser inicializados se inicializan.

La primera función mira los puntos dentro y justo fuera del área y obtiene los valores que deben obtener dichos puntos para que la superficie resultante se conforme con el área. Hubo que encontrar, por tanto, una forma de calcular los valores y de encontrar los puntos relevantes.

9.3.1 Obtener los valores de los puntos: GetMarchingValue()

Se escribió una función que obtiene, dado un área túnel o cámara y un punto en el espacio, el valor que debe tener ese punto para que el algoritmo Marching Cubes pueda representar el mapa. Usa cálculos distintos para cada tipo de área:

- **Cámara:** Tiene forma de elipsoide. Por tanto, usando la ecuación cartesiana que define a los elipsoides, podemos obtener un valor relativo a la superficie de uno de estos.

Esta ecuación toma la forma $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$

Donde se asume que el centro del elipsoide se encuentra en la coordenada cero, los valores X, Y y Z son las coordenadas del punto en los ejes, y a , b y c son las longitudes de los semiejes en las direcciones de los ejes respectivos. La función es equivalente a 1 en la superficie del elipsoide. Es menor que 1 dentro del elipsoide, disminuyendo en valor al acercarse al centro, donde vale 0. Es mayor que 1 fuera del elipsoide, incrementando al alejarse.

El resultado de la ecuación usando la posición del punto relativo al centro de la cámara se divide por dos para que así valga 0.5 si se encuentra justo en la superficie el punto. Luego se multiplica por 255. De esta forma, la isosuperficie de marching cubes que equivale a 127.5 se encuentra en la superficie de la cámara. Por tanto, el resultado equivale al valor que tiene que tener el punto para que la isosuperficie esté en el mismo lugar que la superficie de la cámara.

- **Túnel:** Ya que este consiste en múltiples formas conectadas, no hay una ecuación cartesiana singular que lo defina.

Si fuera solo una esfera, se podría obtener fácilmente el valor mirando la distancia del punto al centro de la esfera. Si equivale al radio de la esfera, se encuentra en la superficie. Dado que la forma es dos esferas conectadas por un cilindro, se puede obtener el valor de marching cubes siguiendo los siguientes pasos:

- Se toma la recta que va desde el centro de una de las esferas al centro de la otra.
- Dado el punto, se busca la distancia entre este y la recta. Esto se hace proyectando el punto sobre la línea infinita definida por los dos centros de las esferas. Si la proyección no se encuentra en la recta, la distancia más corta es desde el punto hasta su extremo más cercano. Si se encuentra entre los dos extremos, la distancia mínima es desde el punto hasta la proyección.
- La distancia obtenida se multiplica por 127.5 (valor de isosuperficie) y se divide por el radio del túnel.

9.3.2 Iterar sobre los puntos que hay que modificar

Para obtener los *DigPoint*, hay que mirar todos los puntos que podrían necesitar ser excavados y obtener sus valores con *GetMarchingValue()*. Los que tengan un valor menor a la isosuperficie y los que se encuentren al lado de estos deben ser devueltos por la función *pointsInDigObject()* de la clase *Nest*.

Una forma simple de mirar todos los puntos relevantes sería iterar sobre todas las coordenadas enteras dentro del cubo que contiene el área y usar *GetMarchingValue()* sobre cada una. Sin embargo, dado que un túnel puede ser muy largo, el cubo que lo contiene puede ser de un volumen mucho mayor que este. Esto podría hacer que el juego se quede pillado más de lo debido calculando muchos valores inútiles.

La forma que se implementó finalmente fue muy similar al proceso que se implementa en los algoritmos A*: se selecciona el punto más cercano al centro del área, del que se sabe seguro que se encuentra sobre la superficie del terreno, y se mete en una pila llamada *pointsToCheck*. Luego, hasta que la pila se vacíe:

1. Se saca el siguiente valor de la pila *pointsToCheck*.
2. Se añade su valor de Marching Cubes y posición al diccionario de salida.
3. Se añade su posición al HashSet *checkedPoints*.
4. *Por cada coordenada adyacente, si no se encuentra en checkedPoints, se añade a pointsToCheck.*

Con este método, se incluyen también los puntos que se encontrarán aún debajo de la superficie adyacentes a los que se encontrarán sobre ella. Esto es importante porque la posición de la superficie del terreno es influenciada por ambos.

9.4 Restricciones de inicialización de *DigPoints*

Tanto en las funciones *Dig()* como *pointsInDigObject()* se tiene que decidir si inicializar o no puntos de excavación adyacentes al punto excavado. Inicialmente, solo se hacían dos verificaciones:

1. El valor que quiere tenerse tras excavar debe ser mayor que el valor actual. Así no se crean puntos de excavación que hacen lo opuesto a excavar: llenar áreas.
2. Debe tener un punto adyacente que no esté debajo de la superficie. De este modo se ignoran los que no se encuentren demasiado profundos bajo la superficie.

Sin embargo, durante las pruebas se notó la necesidad de más restricciones. Estos fueron:

3. No inicializar los que no se encuentrasen sobre la superficie. Al colocar un área a escavar dentro de otro ya excavado, puede cumplirse que el valor que se quiere tener en un punto ya excavado sea mayor, y que se inicialice un *DigPoint* fuera de la superficie (figura 49). Excavar este no cambia el terreno, por lo que se añadió una comprobación adicional para siempre ignorar los que se encuentran sobre la superficie.
4. No inicializar los puntos que, tras ser excavados, no se encontrarían sobre la superficie. Aunque pueden cambiar el terreno si se encuentran al lado de un punto sobre este, no tiene sentido que las hormigas lo excaven. Se añadió una comprobación para que nunca se inicializaran.

En ambos casos anteriores, el punto aún se añade al diccionario de *DigPointData*, ya que el valor cambiado podría aún afectar el terreno, a pesar de no pasar al punto al otro lado de la superficie. Esto ocurre si un punto adyacente suyo sí se encuentra o se encontrará tras ser excavado en el otro lado de la superficie. Entonces, ¿cómo se aplican estos cambios si nunca se inicializan sus puntos de excavación y, por tanto, no pueden ser excavados?

Esto se solucionó en la función *Dig()*. Cuando una hormiga excava un *DigPoint*, se miran los puntos adyacentes. De estos, los que deben ser inicializados y no lo han sido aún, son inicializados. Luego, si alguno tiene uno de los *DigPointData* que nunca debe tener un *DigPoint* inicializado, su valor es aplicado al campo escalar, además del del punto excavado, y se elimina del diccionario.

Esto causa un último problema: podían existir *DigPointData* que no debían ser inicializados, pero no se encontrasen alrededor de *DigPointData* que sí deberían ser inicializados. Por tanto, nunca sería aplicado su valor y se mantendrían indefinidamente en el diccionario. Estos se denominaron puntos sinsentido. Se creó una función llamada *isPointless()* para identificarlos en *pointsInDigObjects()* e ignorarlos, nunca añadiéndolos

al diccionario.

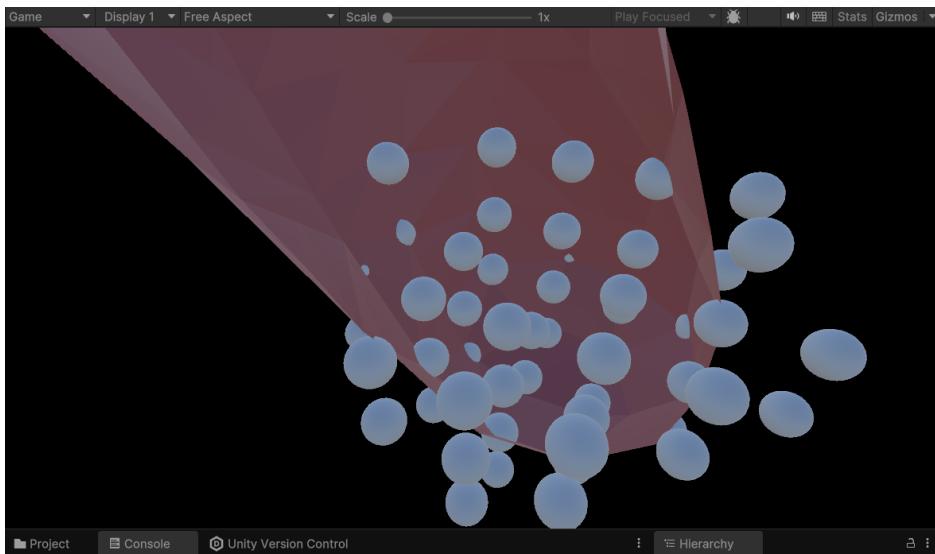


Figure 49: *DigPoints* creados dentro de un túnel ya excavado al colocar otro área.

9.5 El problema de la separación del terreno

Uno de los problemas más grandes que se observaron durante el testeo del sistema de excavación fue que las hormigas podían acabar excavando un área del terreno de tal forma que una sección de *DigPoints* acabara encontrándose separada del resto del terreno. Ya que las hormigas no son capaces de saltar, las islas voladoras de puntos de excavación resultantes serían inaccesibles por las hormigas, y nunca se podrían excavar.

9.5.1 Orden de excavación de *DigPoints*

Para intentar prevenir los casos de separación del terreno, se implementó un sistema de prioridad de excavación de los *DigPoint* para influenciar el orden en el que las hormigas excavasen el terreno.

Se creó la función *ReachableScore()* para obtener una puntuación para un *DigPoint* dado su posición, mirando sus puntos adyacentes. Para cada punto adyacente fuera de la superficie se le añadiría un punto a su valoración. De esta forma, los *DigPoints* de las partes del terreno más separadas del resto obtendrían una puntuación alta, mientras que los que se encontraran muy debajo de la superficie obtendrían una puntuación mínima.

En la función *SenseTask()* de la hormiga trabajadora y *SenseDigTask()* de la hormiga reina, se implementó el uso de la puntuación obtenida por *ReachableScore()* para priorizar los puntos de excavación detectados menos probables de causar una separación de terreno al ser excavados.

9.5.2 Excavación automática de *DigPoints* inaccesibles

Después de mejorar la Inteligencia Artificial de las hormigas lo suficiente para evitar que múltiples hormigas seleccionaran el mismo *DigPoint* a excavar, la eficiencia de excavación del nido aumentó notablemente. Sin embargo, esto tenía la desventaja de volver a replicar el problema de la separación del terreno. Con una alta cantidad de hormigas excavando, con que algunos tardaran más en llegar a la zona de excavación que los demás, en cuanto llegaran sus *DigPoint*, ya se podrían encontrar separados del terreno.

Viendo que no se podría evitar del todo el problema de forma simple, influenciando el comportamiento de las hormigas sin sacrificar velocidad de excavación, se decidió implementar un sistema de autoexcavación. Este sistema identificaría los *DigPoint* separados del resto del terreno, gradualmente los excavaría. Su implementación consistió en dos pasos:

1. Identificar los *DigPoint* separados del resto del terreno. Esto se hizo mediante la función *IsSeparated()*. Dado el punto del *DigPoint*, esta función realiza una búsqueda en anchura en el campo escalar del mapa. Empieza en el punto dado y se expande hacia puntos adyacentes debajo del terreno. Si encuentra un punto del campo escalar debajo del suelo sin *digPointData*, se verifica que el *DigPoint* aún está conectado con el resto del terreno. Si no encuentra uno, y solo hay más posiciones con *DigPoint* o nada conectado a él, se sabe que forma parte de una sección separada del resto del terreno.
2. En cada objeto *DigPoint*, periódicamente comprobar si está conectado con el resto del terreno mediante la función *IsSeparated()*. En cuanto la función confirma que está separado, se ejecuta la función *Dig()* sobre el *DigPoint* para excavarlo, y se autodestruye.

10 El nido de las hormigas

El nido de las hormigas se compone de los túneles y cámaras que estos excavan. El juego comienza con la hormiga reina recién llegada al mapa, que debe excavar la primera cámara donde comenzará a poner huevos. Mediante van naciendo las hormigas nuevas; estas saldrán en búsqueda de comida para guardar en otras cámaras que deben ser excavadas por estas. Cuantas más hormigas, más debe expandirse el nido, de tal forma que pueda contener los nuevos trabajadores, huevos y comida.

Un aspecto muy importante de este nido es la capacidad del jugador de colocarlo en cualquier parte del mapa. La cantidad y posiciones de las cámaras, el sistema de túneles que las interconectan y qué entradas hay con el mundo exterior son decisiones que causarían que cada partida fuera distinta.

Para representar a las distintas partes del nido, se creó la clase *NestPart*. Para gestionar la eternidad del nido, se creó la clase *Nest*.

10.1 Clase NestPart

La clase *NestPart* representaría una parte del nido de las hormigas colocada por el jugador. Podría ser de dos tipos según su forma: Túnel y cámara. Aunque las cámaras podrían variar según su función (guardar comida o huevos), todas se compondrían de la misma forma. Las funciones de *NestPart* que gestionarían las posiciones de las piezas que formarían la parte del nido cambiarían de comportamiento según el tipo de parte del nido del que se tratara.

La función *setMode()* gestionaría el modo en el que se encontrara la parte del nido. Desactivaría los objetos no usados y activaría los que sí se usarían. Solo se usaría en la creación de un *NestPart*, para determinar de qué tipo sería este.

10.1.1 Visualización

Primero se implementó el tipo túnel. Se decidió representarlo en la escena mediante dos esferas y un cilindro. A cada extremo del túnel se colocarían las esferas, y el cilin-

dro las conectaría, dando la ilusión de un cilindro con una semiesfera en cada extremo.

Inicialmente, el objeto cilindro contenía el script *NestPart.cs*, además de tener como hijos las dos esferas. Sin embargo, se producían complicaciones, ya que las transformaciones aplicadas al cilindro se aplicarían también a los dos cilindros, aparte de las transformaciones locales de estos para estar en cada extremo del cilindro (figura 50). Se decidió, por tanto, usar un *emptyObject* (objeto vacío en Unity) para contener el script, las dos esferas y el cilindro.

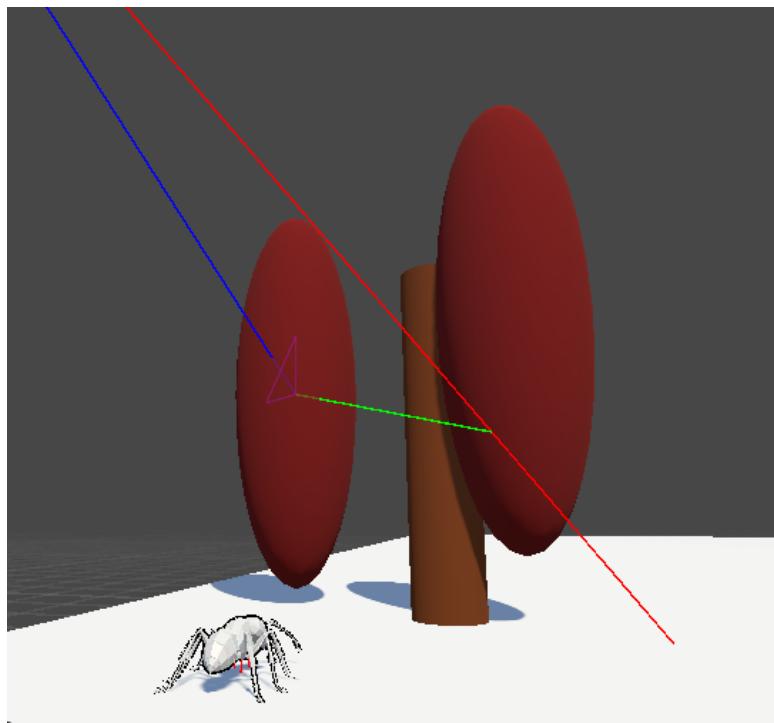


Figure 50: Deformaciones al colocar un túnel con el cilindro como objeto padre

La cámara se representó usando únicamente la primera de las dos esferas, desactivando los otros dos objetos. Este se podría deformar en las distintas dimensiones para tomar la forma elipsoidal de la cámara.

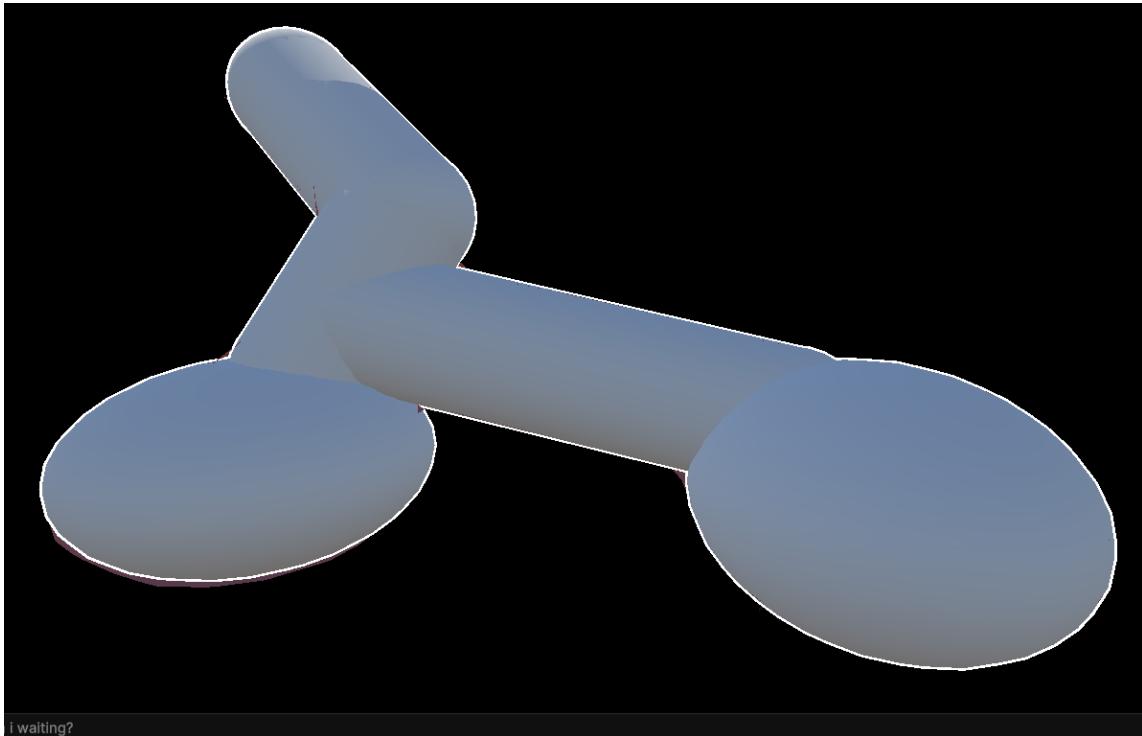


Figure 51: Dos NestPart de tipo cámara conectadas por varias NestPart de tipo túnel

Los materiales y eventualmente shaders usados para la representación gráfica de los NestPart se tratan en la sección del desarrollo gráfico de este documento.

10.1.2 Como se coloca un túnel

Hubo que encontrar una forma intuitiva para que el jugador pudiera colocar las partes del nido en el mapa y decidir sus dimensiones. Para el modo túnel, hubo que decidir dónde empezaba y hasta dónde llegaría, en línea recta. También se pudo decidir el radio del túnel hasta cierto grado.

La colocación de un túnel tiene dos fases:

1. Selección del comienzo del túnel. Cuando el jugador hace clic izquierdo en el modo colocación de túnel, la clase *FlyCamera*, encargada del punto de vista del jugador, lanza un raycast desde dicho punto de vista en la dirección del ratón. Este raycast usa una máscara para poder colisionar tanto con el terreno del juego como con otra parte del nido.

En caso de impacto, se crearía un objeto NestPart a partir de la original. Se pondría su modo a túnel mediante *SetMode()* y mediante *SetPos()* se colocarían sus posiciones de inicio y final en la coordenada de impacto.

2. Después de haber seleccionado el primer punto, el jugador seleccionaría el siguiente. Esto no se hace con un raycast, ya que debería poder acabar en cualquier parte del mapa sin necesidad de una superficie con la que colisionar el raycast. *FlyCamera* por tanto entra en un modo especial en el que esconde el ratón y usa la función *TakePlacingInputs()*. Este toma los movimientos del ratón del jugador y otros inputs:

1. Los movimientos del ratón mueven la segunda posición del túnel en el plano horizontal. Hacia arriba lo aleja del jugador, hacia abajo lo acerca e izquierda y derecha lo mueven en las respectivas direcciones relativas a la dirección en la que mira el jugador.

2. Mientras no se mantiene pulsado la tecla shift, la rueda del ratón mueve la segunda posición hacia arriba y abajo.
3. Mientras se mantiene pulsado la tecla shift, la rueda del ratón aumenta y disminuye el radio del túnel.
4. Al hacer clic izquierdo mientras se mantuviera pulsado la tecla shift, se cancela la creación del túnel.
5. Al hacer clic izquierdo sin mantener pulsado la tecla shift, se confirma la colocación del túnel

10.1.3 Restricciones de la colocación del túnel

El funcionamiento de las hormigas dentro del nido asume que las partes del nido están conectadas. Por tanto, para asegurarse de que el jugador no construya túneles separados del resto del nido, se limitaron las opciones de colocación de la siguiente forma: tan solo el primer túnel del nido puede ser colocado sobre el terreno. Cualquier intento de colocar un túnel tras el primero no detectará el terreno para ser elegido como punto de partida del túnel. Todos los túneles siguientes solo pueden ser colocados a partir de otros túneles y cámaras del nido.

Se quiso prevenir dos otras posibilidades de colocación de los túneles: que ambos extremos se encontraran fuera del nido y del terreno; y que el túnel interseccionara con el suelo de las cámaras, al ser esto donde se dejan los objetos de la cámara. Por tanto, se creó la función booleana *IsValidPosition()*, que se encargaría de señalar si la posición actual de una parte del nido fuese válida para ser colocada durante la colocación de partes de nido.

La función *IsValidPosition()* realizaría las siguientes comprobaciones mientras la parte del nido estuviera siendo colocada por el jugador:

1. Comprobar que no se encontraran ambos ambos extremos del túnel (los centros de las dos esferas) fuera del terreno y del nido a la vez. Esto se hizo usando la función *WasAboveSurface()*, que mira el mapa original sin las modificaciones de terreno del nido. De esta forma, las partes dentro del nido se considerarían debajo del terreno.
2. Para cada cámara en el nido, comprobar que su parte inferior no se encontrara dentro del túnel. Esto se hizo usando la función *DistancePointLine()* que mira la distancia entre un punto y una línea no infinita que también se usa para crear *DigPoints* a partir del túnel.

10.1.4 Como se coloca una cámara

Contrario al caso del túnel, la cámara usaba una coordenada para decidir su posición. Se coloca en dos fases, como con el túnel:

1. Seleccionar el centro de la cámara. Esto se hace igual que con el modo túnel, usando un raycast. Se crea un objeto *NestPart* en modo cámara con el centro en el lugar de colisión.
2. Despues de colocar el punto de origen, la clase *FlyCamera* esconde el ratón y usa la función *TakePlacingInputs()* para tomar los comandos del jugador, de forma distinta que para el modo túnel:
 1. Sin tener pulsada la tecla shift, los movimientos del ratón mueven la posición horizontal de la cámara y la rueda del ratón lo mueve verticalmente.

2. Pulsando la tecla shift, los movimientos del ratón afectan las dimensiones X y Z (horizontales en Unity) de la cámara y la rueda del ratón cambia su dimensión Y. Estas dimensiones están limitadas por un máximo y un mínimo distintos para los ejes X, Z e Y para que se mantenga una forma de elipsoide aplastado.
3. Pulsando shift y clic izquierdo, se cancela la creación de la cámara.
4. Pulsando clic izquierdo sin shift coloca la cámara.

10.1.5 Prevención de intersección de cámaras

Al contrario del túnel, hubo que añadir ciertas restricciones a la colocación de las cámaras. Precisamente, no debería ser posible que dos cámaras tuvieran una intersección, ya que esto podría causar comportamientos extraños en las hormigas. Para poder detectar estas intersecciones, se decidió usar el sistema de colisiones de Unity. Se siguieron los siguientes pasos:

1. Añadir un colisionador a la esfera que forma la cámara. Se quiso usar un colisionador esférico, pero este no se podía deformar en una elipsis como la cámara. Por tanto, se acabó usando un *meshCollider*: un colisionador con la forma de una malla. Este tomaría la forma de la esfera y sí se deformaría con él. Durante el modo túnel, este se desactivaría.
2. Se añadió un *RigidBody* al *EmptyObject* que contiene el script y los objetos de esfera y cilindro. Sin un componente *RigidBody*, un objeto no registra colisiones en Unity.
3. Se retocaron los parámetros del *meshCollider* y *RigidBody* para que pudieran funcionar: a Unity no le gusta gestionar el centro de masa y tensor automáticos al usar *meshColliders*, especialmente cuando no está seguro de si es convexa o no la malla. Hubo que poner predeterminados el centro de masa y el tensor, y activar la propiedad convexa para los tres objetos del *EmptyObject*

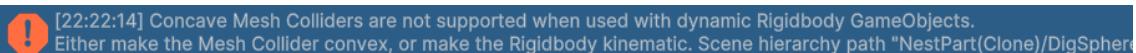


Figure 52: Mensaje de error de Unity al usar un *RigidBody* con un colisionador de malla no convexa

4. Para evitar que cualquier otro objeto pudiese colisionar con los objetos de tipo *NestPart*, se editaron los ajustes de qué capas de objetos podrían colisionar con él para solo habilitar su propia capa (figura 53).

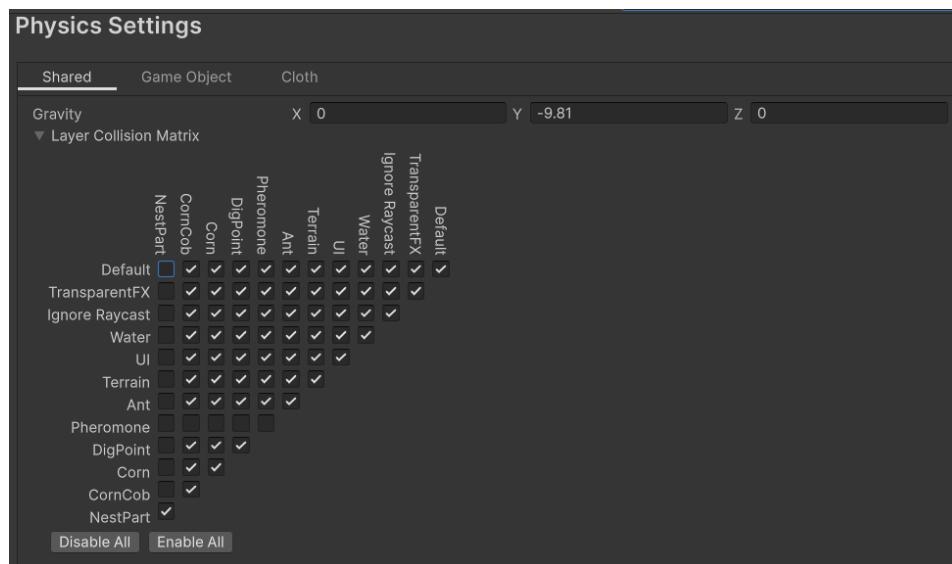


Figure 53: La ventana de ajustes de colisiones entre capas general de Unity

5. Se añadió un *HashSet* de *GameObjects* llamado *CollidingClones* a la clase para recopilar los objetos que se encuentren en colisión con la cámara. La funciones *OnCollisionEnter()* y *OnCollisionExit()* detectan cuando entra y sale un objeto, y gracias a que solo puede colisionar con otras cámaras, fue simple implementar añadir y sacar los objetos del juego del *HashSet* según si entraban o salían de la colisión con la cámara.
6. Usando la nueva función *IsNotCollidingWithOtherChamber()* se podía comprobar si la cámara a colocar se encontraba en un lugar válido, y se desactivaría la habilidad de colocarla si se encontraba en intersección con otra cámara existente. El color de la cámara cambiaría a rojo translúcido para señalizar al jugador que no se podía colocar.

La habilidad de colocar partes del nido a partir de otras partes del nido usando raycasts fue implementada después de la habilidad de detectar la intersección. Se añadieron colisionadores de malla a la otra esfera y el cilindro para que estas se pudiesen detectar, y no se desactivarían los colisionadores al cambiar de modo para que estos pudiesen ser detectados por los raycast. Esto provocaría que la parte cámara sintiera colisiones con los túneles, así que se especificó en *OnCollisionEnter* que solo se metieran en el *HashSet* objetos detectados que tuvieran un modo cámara.

10.1.6 Otras restricciones de colocación de una cámara

Aparte de no interseccionar la cámara con otras cámaras, hubo dos más situaciones que se quisieron prevenir: que parte de la cámara conectara con la superficie del terreno no dentro del nido, y que su suelo interseccionara con un túnel.

Para poder comprobar ambas restricciones a la vez, se decidió que se comprobarían ciertos puntos dentro y alrededor de la cámara mientras esta fuera colocada por el jugador. Las posiciones de estos puntos tendrían en cuenta las dimensiones distintas X, Y y Z que tuviera la elipsis de la cámara, y se dividieron en dos grupos principales:

1. Los puntos encima de la superficie del suelo de la cámara. En estos puntos solo se comprobaría que se encontraran dentro del terreno o dentro del nido. Si alguno de estos puntos se encontrara fuera del terreno, comprobado mediante la función *WasAboveSurface()*, que solo tiene en cuenta el terreno inicial del mapa sin considerar el terreno modificado por el nido, se consideraría que la cámara conectaba con la superficie y no podría ser colocada. Estos puntos son coloreados

en amarillo en la figura 54.

2. Los puntos en y debajo de la superficie de la cámara. Para cada uno de estos puntos se comprueba que no se encuentre ni fuera del terreno ni dentro de un túnel aún no excavado. Se hace esto usando la función *IsAboveSurface()* para ver si está fuera del terreno y mirando sus distancias a los túneles mediante la función *DistancePointLine()*. Estos puntos son coloreados en rojo en la figura 54.

Tanto esta comprobación de los puntos como la comprobación de colisiones con otras cámaras anterior se usan en la función *isValidPosition()* para determinar si una cámara siendo creada por el jugador puede ser colocada.

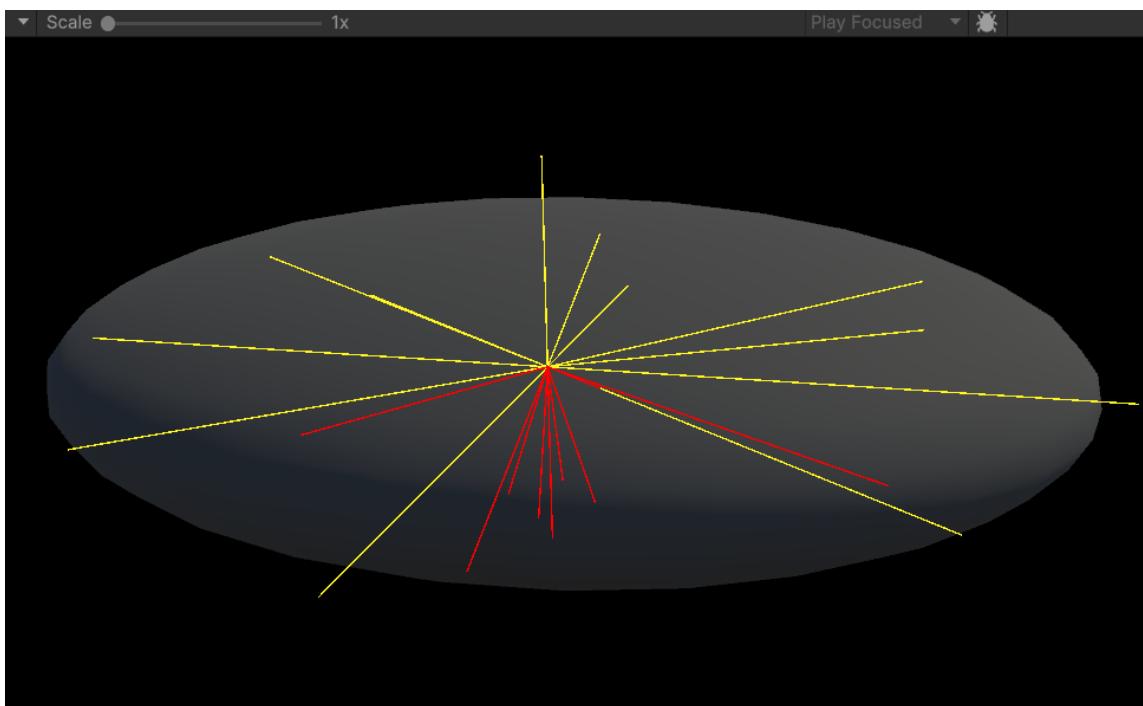


Figure 54: Los distintos puntos de comprobación conectadas con el centro de la cámara

10.1.7 Cómo comprobar si ha sido excavado un *NestPart*

Para prevenir que las hormigas trabajadoras pudiesen intentar llegar a partes del nido que no hayan sido excavadas aún, hubo que poder identificar qué partes han sido completamente excavadas y cuáles no. Para ello se implementó dos cosas:

1. Añadir un *HashSet* llamado *digPointsLeft* a la clase que contuviera todas las posiciones de los *DigPoints* necesarios para excavar el nido. Estos se obtendrían al colocar la parte del nido, cuando se generan todos los objetos *DigPointData* de *NestPart*.
2. Crear una función booleana llamada *HasBeenDug()* a la clase. En ella, se comprueba el *HashSet* *digPointsLeft*. Si está vacío, la función devuelve true, ya que no quedan puntos a excavar. Si no está vacío, itera sobre el *HashSet*. Para cada posición restante, si ya no existe un *DigPointData* en dicha posición del mapa, se elimina de *digPointsLeft*. Si aún existe un *DigPointData*, *HasBeenDug()* inmediatamente devuelve true.

De esta forma, *HasBeenDug()* se encarga tanto de gestionar el estado de excavación del *NestPart* como de informar sobre su progreso.

10.2 Clase Nest

La clase *Nest* se creó para gestionar todas las partes del nido. Contiene las referencias a todos los objetos *NestPart* que componen a este en su lista de *GameObject* llamada *NestParts*. El nido también gestiona la memoria general de qué fuentes de comida se han encontrado por hormigas. Finalmente, la clase *Nest* contiene las funciones necesarias para averiguar qué superficies del mapa se encuentran dentro de este y en qué tipo *NestPart* se encuentran.

10.2.1 Memoria del nido y mandar Tasks

Un aspecto fundamental del funcionamiento de las colonias de las hormigas es cómo exploran las afueras del nido en busca de comida. Hormigas que encuentran comida nueva informan al resto del nido de ello usando feromonas, después de lo cual el resto de las hormigas empieza a ir a recoger comida de la fuente hasta agotarla. Esto forma caminos semiconstantes repletos de hormigas moviéndose de y entre la fuente de comida. Para simular este comportamiento cooperativo, se decidió implementar la memoria del nido.

Las hormigas que exploran fuera del nido buscan encontrar las mazorcas de maíz repartidas por el mapa. Cuando encuentran uno, memorizan su localización y vuelven al nido. Cuando llegan al nido, comparten esta información con el resto de las hormigas. Esto se hace compartiendo la información con la clase del nido, que luego puede mandar a las hormigas que buscan trabajo a recolectar de las fuentes de comida conocidas. Las mazorcas son guardadas en las memorias de las hormigas y el nido mediante sus identificadores. El contenedor *KnownCornCobs* que guarda esta memoria en la clase *Nest* es, por tanto, un *HashSet* de enteros.

La función *GetNestTask()*, dada la superficie de la hormiga que pide un trabajo que hacer a la clase nido, puede devolver una tarea de recolección de comida de una de las mazorcas conocidas usando la subfunción *GetNestCollectTask()*. Para devolver un *Task* de recolectar comida, la subfunción tomaría los siguientes pasos:

1. Comprobar que hay entradas en la memoria del nido. Si no hay, el nido no sabe de lugares de comida y *GetNestCollectTask()* devuelve el valor falso.
2. Escoger un identificador de mazorca de la memoria al azar y comprobar que su mazorca correspondiente aún existe o tiene comida. Si no, se elimina de la memoria y se vuelve al paso 1.
3. Buscar un camino de la superficie de la hormiga a la fuente de comida usando la función *GetKnownPathToPoint()*. De esta forma, solo se puede encontrar un camino si los caminos de feromona llevan a la mazorca. En caso de no poder encontrar un camino a la mazorca, se asume que se ha vuelto inaccesible y se borra de la memoria, en cuyo caso se vuelve al paso 1. En caso de sí encontrar un camino, *GetNestCollectTask()* devuelve un *Task* de recolectar la comida usando dicho camino.

Después de haber llamado a la función *GetNestCollectTask()*, se actualiza el contador de mazorcas encontradas versus mazorcas restantes en el mapa en la interfaz gráfica, para reflejar los posibles cambios en la memoria del nido.

10.2.2 Saber si se está en el nido

Una parte importante del sistema de pathfinding del juego es que las hormigas tengan la capacidad de guiarse por el nido perfectamente, mientras que fuera del nido deben

guiarse por los caminos de feromonas o explorar a ciegas. Las funciones de pathfinding *GetKnownPathToPoint()* y *GetKnownPathToMapPart()* se limitan a expandirse por superficies en el nido y superficies con feromonas para simular esto. Por tanto, estas funciones necesitan una forma de comprobar que una superficie se encuentra dentro del nido de forma eficiente. Se decidió, por tanto, incluir en la clase *Nest* las funciones necesarias para determinar si una superficie se encuentra dentro de ella.

PointInNest()

Se creó la función booleana *PointInNest()*, que dada una coordenada, verificaría si se encuentra dentro del nido o no. Para cada *NestPart* del nido, se usaría la función *getMarchingValue()* para comprobar el valor de la coordenada respecto a este. Si alguno de los valores devueltos fuera menor que la IsoSuperficie, significaría que se encontraría dentro del *NestPart* y la función devolvería True. Después de iterar sobre cada *NestPart* sin obtener un valor menor a la isoSuperficie, devolvería False.

Esta función se optimizó guardando el índice del último *NestPart* que devolviera True en la clase *Nest*. Las funciones de búsqueda de caminos A* suelen mirar nodos adyacentes a las previas o cercanas a este. Por tanto, suelen mirarse grupos de superficies en el mismo *NestPart*, y si el *NestPart* en cuestión es el último en una lista de 20 partes, habría que hacer 20 usos de *getMarchingValue()* antes de devolver True. Para remediar este gasto de tiempo y tomar ventaja de este hecho, *PointInNest()* empezaría a iterar sobre la lista de partes del nido, empezando por el índice del último que devolvió true, usando una operación de módulo para volver a 0 cuando el iterador alcanzara el tamaño de la lista.

Las funciones de búsqueda usarían *PointInNest()* sobre la coordenada del centro del cubo que contiene la superficie a mirar. Esto resultaba ser poco preciso, ya que algunas superficies son pequeñas y se encuentran en las esquinas del cubo, y podían formar parte del nido sin que lo hiciera el centro del cubo, causando que algunas superficies dentro del nido se consideraran fuera del nido.

SurfaceInNest()

Se creó la función booleana *SurfaceInNest()* para poder comprobar con precisión si una superficie se encontraría dentro del nido. Dada una superficie *CubeSurface*, miraría el grupo de esquinas que lo definiera. Por cada esquina del cubo sobre su superficie, se comprobaría con *PointInNest()* si este se encontraba dentro del nido. Las esquinas de abajo serían ignoradas para ahorrar tiempo, ya que puntos debajo de la superficie no se podrían encontrar dentro de ninguna parte del nido. Con que una esquina sobre la superficie se encontrara dentro del nido, *SurfaceInNest()* devolvería true. Si ningún punto se encontrara dentro de una parte del nido, devolvería False.

PointInNestPart()

Las funciones *PointInNestPart()* y *SurfaceInNestPart()* se crearon para detectar si superficies se encontraran dentro de partes del nido de tipos específicos.

PointInNestPart() funcionaría similar a *PointInNest()*, excepto que tomaría como entrada, además de la posición, el tipo de nido del que se quisiera comprobar que contuviera el punto. Al iterar sobre los *NestPart*, se saltaría aquellos de otros tipos. Si el tipo de nido en el que se buscaría fuera “las afueras del nido”, *PointInNestPart()* devolvería el valor inverso de *PointInNest()*.

También empezaría a iterar sobre los *NestPart* por el índice del último *NestPart* que devolviese True en llamadas previas.

SurfaceInNestPart()

Muy similar a *SurfaceInNest()*, excepto que tomaría como entrada un *CubeSurface* y un tipo de parte de nido. Llama a *PointInNestPart()* para cada vértice del cubo fuera de la superficie dada. Si el tipo de parte de nido dado fuera “las afueras del nido”, devolvería el valor reverso de *SurfaceInNest()*.

10.2.3 Gestionar los objetos dentro del nido

Un aspecto fundamental del juego es la obtención de comida y huevos, y el almacenamiento de estos en sus cámaras respectivas. Para permitir que las hormigas sepan qué objetos se encuentran en partes del nido equivocadas para que puedan moverlos a las partes correctas, hubo que crear un sistema para gestionar la ubicación de cada objeto en el nido. Este se formó mediante los siguientes pasos:

1. Añadir un *HashSet* de identificadores de pepitas de maíz llamado *CollectedCornPips* y un *HashSet* de identificadores de huevos de hormiga llamado *AntEggs* a la clase *NestPart*. De esta forma se podría registrar en cada parte del nido los objetos que contuviera, introduciendo sus identificadores en el *HashSet* respectivo.
2. Añadir *HashSet* de identificadores de pepitas de maíz e identificadores de huevos de hormiga a la clase *Nest* llamados *lostEggs* y *lostCorn*. Estos dos *HashSet* se usarían para registrar los objetos depositados a los que no se les pudo atribuir una parte del nido. Todos los objetos en estos *HashSet*s se considerarían fuera de lugar.
3. Modificar las funciones *PlaceCorn()* y *PlaceAntEgg()* de la clase *Ant* de la hormiga para añadir los identificadores de los objetos soltados al *HashSet* respectivo del objeto *NestPart* en el que se encontrara la hormiga.

En el caso de encontrarse tanto dentro de una sección de túnel como dentro de una cámara donde estos dos se solapan, se priorizaría la cámara para decidir dónde se encontrase el objeto.

Si la hormiga no se encuentra dentro del nido o solo dentro de un túnel al soltar el objeto, se considera fuera de lugar y se añade al *HashSet* respectivo de objetos fuera de lugar de la clase *Nest*.

4. Crear las funciones *DisplacedEgg()* y *DisplacedCorn()* para poder obtener la lista de objetos fuera de lugar que se usaría para repartir tareas de relocalización de objetos a las hormigas en la función *GetNestTask()*. Estas funciones considerarían fuera de lugar:

Los objetos en *lostEggs* y *lostCorn*.

Pepitas de maíz dentro de cámaras de huevos.

Las pepitas de maíz dentro de la cámara de la reina, sin contar los primeros cinco. Estos son los que se llevan a la reina para comer.

Huevos en cámaras de comida y la cámara de reina.

5. En las funciones *OnDestroy()* de las clases *Corn* y *Ant*, que son invocadas al destruirse los objetos pepita de maíz y hormiga respectivamente, eliminar sus identificadores de los *HashSet* de la clase *Nest* y los objetos *NestPart*. Esto se hace para prevenir que las hormigas intenten ir a recoger un objeto inexistente.

11 Inteligencia artificial

11.1 Objetivos

Uno de los grandes objetivos de la creación de este juego de simulación fue que cada hormiga en el nido fuera un agente completamente independiente, pero capaz de comunicarse con las demás hormigas, reaccionando a sus alrededores y trabajando para el nido.

Hubo, por tanto, que escribir el código necesario para convertir a las hormigas en agentes inteligentes, capaces de percibir su entorno, procesarlo y actuar en respuesta.

De todos los aspectos del proyecto, este fue en el que más se trabajó, experimentando varios cambios y versiones fallidos, de los que no todo se puede recopilar, además de muchos ajustes al resto del desarrollo del juego.

Durante el inicio del desarrollo, la inteligencia artificial de la hormiga se basaba en un simple sistema de estados. Este demostraba demasiadas limitaciones, y fue reemplazado por el sistema de árboles de comportamiento. Antes de retirarlo, llegó a gestionar 6 estados distintos: Exploring para explorar, FollowingPher para seguir caminos de feromonas, FollowingPath para seguir caminos a objetos cercanos, ReachedObjective para cuando llegara a su objetivo, Controlled para cuando el jugador controlara la hormiga y Passive para cuando la hormiga se encontraba sin nada que hacer.

La dificultad de este sistema fue la complejidad de expansión de esta. Cada vez que se quisiese añadir un estado nuevo, había que gestionar las transiciones entre este y todos los demás. Y cuando se quiso avanzar en el área del comportamiento de la hormiga, hubo que buscar un sistema más modular y dinámico. Este fue el de los árboles de conocimiento.

11.2 Fluent behaviour tree

11.2.1 Descripción

Los árboles de comportamiento son una estructura jerárquica con forma de árbol que se puede usar para modelar las acciones y decisiones de un agente. Permiten descomponer tareas complejas en muchos pasos pequeños y crear inteligencias artificiales altamente modulares y reutilizables.

Uno de los aspectos clave de los árboles de comportamiento es que suele ser apátrida, es decir, no usa estados, sino que depende constantemente de la entidad o del entorno para decidir qué hacer. Cada actualización reevalúa el árbol de comportamiento contra el estado de la entidad y el entorno. En cada actualización, el árbol retoma desde donde lo dejó la última actualización. Debe averiguar en qué estado se encuentra y qué debe hacer ahora.

El sistema de árbol de comportamiento que se decidió usar se denomina Fluent Behaviour Tree. Se usa el término fluent para denotar que el árbol se construye en el código mismo, al contrario de los árboles de comportamiento tradicional que se cargan desde los datos. De este modo sacrifica un editor visual a cambio de poder escribirse de forma rápida y flexible. Para este proyecto, considerando que no se implementaría una inteligencia artificial demasiado compleja, fue considerado apto.

La biblioteca de la clase Fluent-Behaviour-Tree fue creada por Ashley Davis y adquirido de forma gratuita de su [repositorio oficial](#).

11.2.2 Funcionamiento

Estados de árbol de comportamiento

Cada nodo del árbol devuelve uno de los siguientes estados:

- *Success*: El nodo ha terminado su ejecución con éxito.
- *Running*: El nodo aún se está ejecutando.
- *Failure*: El nodo ha terminado su ejecución, pero ha fallado.

Tipos de nodos

Los distintos tipos de nodos que pueden contener el árbol de comportamiento son:

- *Action/leaf node*: Ejecuta una función o unas líneas de código. Estos deben devolver un estado de árbol de comportamiento.

Se añade al árbol de comportamiento con el código `.Do("nombre_acción", t => función)`

- *Sequence*: Ejecuta cada nodo hijo en secuencia. Si el nodo hijo devuelve true, pasa a ejecutar el siguiente hijo. Cuando su último hijo devuelve éxito, devuelve éxito. Cuando un nodo hijo devuelve *Failure* o *Running*, devuelve dicho valor.

Creado con `.Sequence("nombre_secuencia")` seguido de otros nodos y finalizado con `.End()`

- *Selector*: Ejecuta cada nodo hijo en secuencia hasta encontrar uno que devuelva *Success* o *Running*, y devuelva ese valor. Si un hijo devuelve *Failure*, pasa al siguiente hijo. Si su último hijo devuelve *Failure*, devuelve *Failure*.

Creado con `.Selector("nombre_selector")` seguido de nodos hijo y finalizado con `.End()`.

- *Parallel*: Ejecuta todos los nodos hijos en paralelo. Devuelve *Running* hasta que un número requerido de nodos hijo devuelvan *Failure* o *Success*.

Creado con `.Parallel("nombre_paralelo", num_fallos, num Éxitos)` seguido de nodos hijos y finalizado con `.End()`.

- *Condition*: Dada una función booleana, devuelve *Success* si esta devuelve True. Devuelve *Failure* si esta devuelve False. Se suele usar con el nodo selector.

Creado con `.Condition("nombre_condición", t => función_booleana)`.

- *Inverter*: Invierte el valor *Success* o *Failure* del nodo hijo. Devuelve *Running* si el nodo hijo lo devuelve.

Creado con `.Inverter("nombre_inversor")` seguido de algún nodo.



Figure 55: Forma que representa cada tipo de nodo

Ejecución

Se usa la función `Tick()` sobre el objeto árbol creado para ejecutarlo. Este ejecuta su primer nodo mediante la función `Tick()` sobrecargada. Cada nodo o ejecuta con `Tick()` sus nodos hijos según su tipo, o ejecuta su función en caso de ser un nodo hoja. La función

Tick() devuelve el estado del nodo ejecutado, y se le pasa el tiempo transcurrido desde la última llamada. Este valor no se usa por sí mismo, pero permite a las funciones llamadas por los nodos tener en cuenta el paso del tiempo. No se llegó a usar esta funcionalidad.

En las clases de la hormiga trabajadora y la hormiga reina, se implementó la función *Tick()* de su árbol de conocimiento en su función *FixedUpdate()*. Este se llamaría si y solo si la hormiga ya hubiera nacido y se encontrara sobre el terreno, no en estado de caída.

Se implementó una versión personalizada de la función *Tick()* que tomaría, además del tiempo transcurrido, también un string. Cada nodo añadiría su nombre al final del string, y los nodos de acción lo imprimirían en la pantalla. De esta forma se podía supervisar qué ramas del árbol se estaban ejecutando durante las pruebas del juego.

11.3 Implementación de Task

11.3.1 Motivo

El árbol de comportamiento se ejecuta desde el nodo base cada vez que es llamado, sin guardar la posición ni estado en la que acabó la inteligencia artificial en la última llamada. La clase de la hormiga tuvo, por tanto, que gestionar el estado en el que se encontrase para reanudar la toma de decisiones. Por ejemplo, si una hormiga llevase una pepita de maíz al nido, el árbol debería reconocer esto mirando si llevaba una pepita de maíz y viendo que aún no se encontrara en la cámara de comida. ¿Pero si una hormiga se encontrara fuera explorando, cómo podría el árbol de comportamiento verificar esto de forma eficiente?

En las primeras versiones de la implementación del árbol de comportamiento, se creó una versión del árbol que recordara en qué nodo se encontraba la última ejecución y reanudara desde allí. Sin embargo, demostró ser difícil gestionar que no se quedara pillado y necesitaba de más verificaciones para que la hormiga se comportara de forma dinámica y reaccionara a su entorno.

Se decidió, por tanto, crear una clase de información que podría tanto comunicar al árbol de comportamiento qué intentaba hacer la hormiga como aportar información vital para la ejecución de ciertos pasos: la clase *Task*.

11.3.2 Idea general

Cada hormiga guardaría un objeto *Task* que describiría la acción que la hormiga quisiera completar. Cada vez que el árbol es actualizado, puede mirar la tarea actual de la hormiga para decidir qué acciones tomar, y si no tiene una acción, tomará los pasos necesarios para decidir cuál debería ser su siguiente tarea. De esta forma, el árbol recibe el contexto necesario de la hormiga para completar acciones largas con múltiples pasos, en vez de solo ser capaz de reacciones inmediatas al entorno.

11.3.3 Tipos de acciones y tareas

Para cada trabajo y estado en el que pudiese encontrarse una hormiga, habría un *Task* asignado. Este describe la tarea o acción que la hormiga debe completar. Suelen contener una lista de *CubeSurface* que representa un camino que la hormiga debe seguir para llegar a su objetivo o completar la tarea. Los tipos de *Task* creados fueron los siguientes:

- *Explore*: La hormiga está explorando. Cada vez que se crea un *Task* de exploración con la función *GetExploreTask()*, esta usa la función *GetExplorePath()* si se encuentra fuera del nido para obtener un camino desde la superficie de la hormiga a una de las superficies cercanas fuera del nido más alejadas de feromonas visibles. Si se encuentra dentro del nido, la función usa *GetKnownPathToMapPart()* para encontrar un camino hacia fuera del nido. El objeto *Task* devuelto contiene el camino que la hormiga debe completar, además de un entero que representa cuántas veces más la hormiga debería tomar una acción de exploración antes de volver al nido.
- *Dig*: La hormiga debe excavar el objeto *DigPoint* señalado por la tarea. Los objetos *Task* de tipo *Dig* contienen el identificador del objeto *DigPoint* a excavar y un camino para llegar a este. Creado con la función de construcción general de *Task* pasándole su tipo, la id del objeto y el camino.
- *GetCorn*: La hormiga debe recoger una pepita de maíz. Los objetos *Task* de tipo *GetCorn* contienen el identificador de la pepita de maíz a recoger, y una camino para llegar a este. Creado con la función de construcción general de *Task* pasándole su tipo, la id del objeto y el camino.
- *GetEgg*: La hormiga debe recoger una pepita de maíz. Los objetos *Task* de tipo *GetCorn* contienen el identificador de la pepita de maíz a recoger y un camino para llegar a esta. Creado con la función de construcción general de *Task* pasándole su tipo, la id del objeto y el camino.
- *CollectFromCob*: Similar a *GetCorn*, excepto que la hormiga debe recoger la pepita de maíz de una mazorca. Contiene el identificador de la mazorca y un camino hacia este. La hormiga puede recoger las pepitas a más distancia cuando se encuentra en una mazorca para tener en cuenta su tamaño. Creado con la función de construcción general de *Task* pasándole su tipo, la id del objeto y el camino.
- *Eat*: *Task* reservada para la hormiga reina. Creado con *GetEatTask*, una función que, dada la id de una pepita de maíz, devuelve un task de tipo *Eat* con un camino hacia esta. Cuando la hormiga reina llegue a la pepita y se la coma, acabará la tarea.
- *GoOutside*: La hormiga debe salir del nido. Creado con la función *GoOutsideTask()*, contiene el camino a la salida del nido más cercano obtenido mediante A* en la función *GetKnownPathToMapPart()*.
- *GoInside*: La hormiga debe volver al nido. Creado con la función *GoInsideTask()*, contiene el camino al nido más corto encontrado mediante A* en la función *GetKnownPathToMapPart()*.
- *GoToChamber*: La hormiga debe ir a una parte del nido específica. Suele usarse para llevar pepitas de maíz a cámaras de comida. Se crea con la función *GoToNestPartTask()*, y obtiene el camino a una parte del nido del tipo buscado mediante *GetKnownPathToMapPart()*.
- *GoToTunnel*: Idéntico a *GoToChamber* excepto por el tipo de parte del nido al que debe ir la hormiga.
- *Lost*: La hormiga se encuentra perdida. Creado con la función *GetLostTask()*. Este mira las superficies *CubeSurface* cercanas. Si encuentra uno que se encuentra dentro del nido, el *Task* guarda el camino a este. Si no, guarda un camino a una superficie cercana aleatoria. La hormiga seguirá obteniendo un *Task* de tipo *Lost* hasta encontrar el nido o un camino de feromonas que conecte con este.

Cuando las funciones de construcción de los *Task* de tipo *GoOutside*, *GoInside* y *Go-*

ToNestPart no encuentran un camino a su objetivo, devuelven un objeto *Task* de tipo perdido a la hormiga.

- *Wait*: La hormiga permanece inactiva durante un periodo. Creado con la función *WaitTask()*, la hormiga espera la cantidad especificada. Usado para simular descansos de las hormigas.

11.3.4 Prevenir que múltiples hormigas reciban tareas para el mismo objeto

3 de las tareas que las hormigas trabajadoras pueden recibir incluye modificar un objeto que solo puede ser modificado por una hormiga a la vez: *GetCorn*, en la que se recoge una pepita de maíz; *GetEgg*, en la que se recoge un huevo; y *Dig*, en la que se excava un *DigPoint*. Para prevenir que múltiples hormigas pudiesen recibir una tarea con el mismo objeto, se implementó el uso de la variable *antId* en estos objetos.

Cuando una hormiga recibe un *Task* para recoger uno de los objetos o excavar, dicho objeto *Corn*, *Ant* o *DigPoint* tiene puesto su *antId* al identificador de la hormiga que recibe la tarea. Antes de poder recibir una tarea que incluye uno de estos tipos de objetos, la clase *Task* verifica si ese objeto ya está incluido en una de estas tareas. Si lo está, no puede asignarlo como tarea. Esta verificación se hace de forma eficiente siguiendo los siguientes pasos:

1. Se verifica el valor de *antId* del objeto en cuestión. Si su valor es -1, significa que ninguna hormiga va a recogerlo o excavarlo.
2. Si el valor es mayor que -1, se busca la hormiga con dicho identificador en el diccionario de hormigas, y se obtiene su tarea.
3. Si la tarea de la hormiga encontrada involucra el objeto, se previene asignar otra tarea involucrando dicho objeto.
4. Si la tarea de la hormiga no involucra el objeto, significa que ha sido interrumpido en algún momento y ha perdido su tarea. Por tanto, sí se puede asignar el objeto en un *Task* a la nueva hormiga.

11.4 Versión final de la implementación del árbol de comportamiento

La inclusión de la clase *Task* permitió simplificar mucho el diseño de los árboles de comportamiento de la hormiga trabajadora y la hormiga reina. El diseño principal de estos fue dividir el árbol en dos secciones: uno que se encarga de seleccionar el *Task* que la hormiga debe cumplir teniendo en cuenta su entorno, y uno que ejecuta las acciones que cada tipo de *Task* requiere para ser completado.

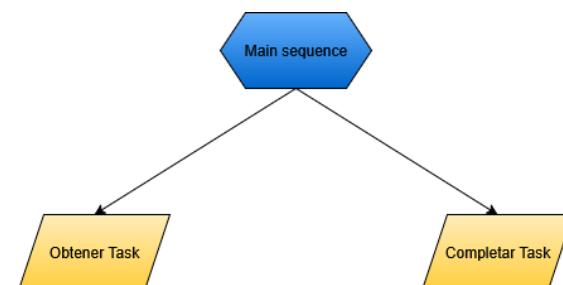


Figure 56: Primer nivel del árbol de comportamiento

11.4.1 Sección de obtención de tarea *Task de la hormiga trabajadora*

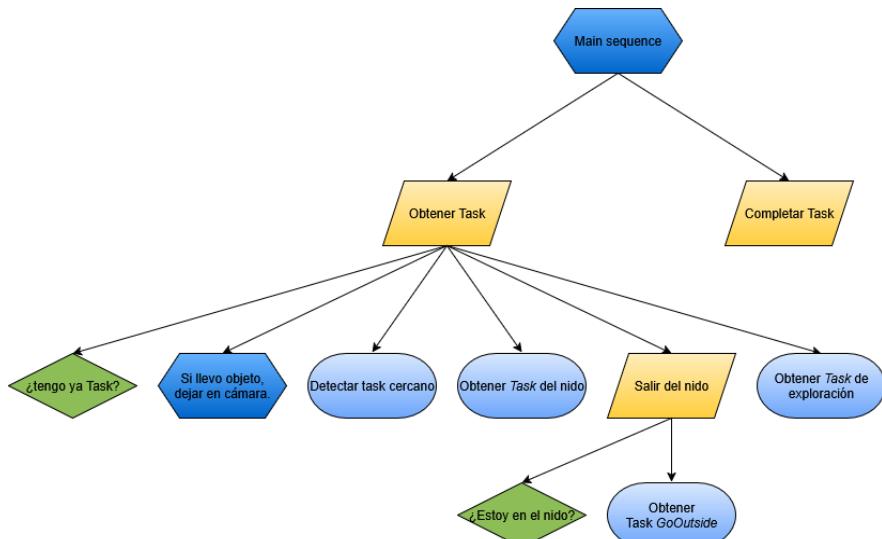


Figure 57: Árbol con el subárbol de selección de objetivo de la hormiga trabajadora expandida

Cada nodo hoja de la sección de obtener *Task* le asignaría una tarea a la hormiga, pero hubo que tener en cuenta el orden en el que se mirara qué tareas asignar. Cabe destacar que, pase lo que pase, si la hormiga no tiene una tarea asignada, siempre se le asignará una (o activará una animación durante la cual no se llama al árbol), y eventualmente se le devuelve un estado *Success* al nodo padre para que la secuencia principal ejecute el nodo “Completar Task”.

1. El primer nodo, condicional, comprueba si la hormiga ya tiene un *Task*. Si tiene, devuelve *Success* y el nodo de selección acaba.
2. El segundo nodo es de secuencia, y es cabecera del subárbol que se encarga de asignar una de las tareas necesarias para llevar un objeto a la cámara respectiva dentro del nido, dependiendo del estado de la hormiga (figura 58). Primero se comprueba que la hormiga lleva un objeto; si lleva comida, hay que llevarlo a una cámara de comida o a la reina. Si lleva un huevo de hormiga, hay que llevarlo a una cámara de huevos de hormiga.

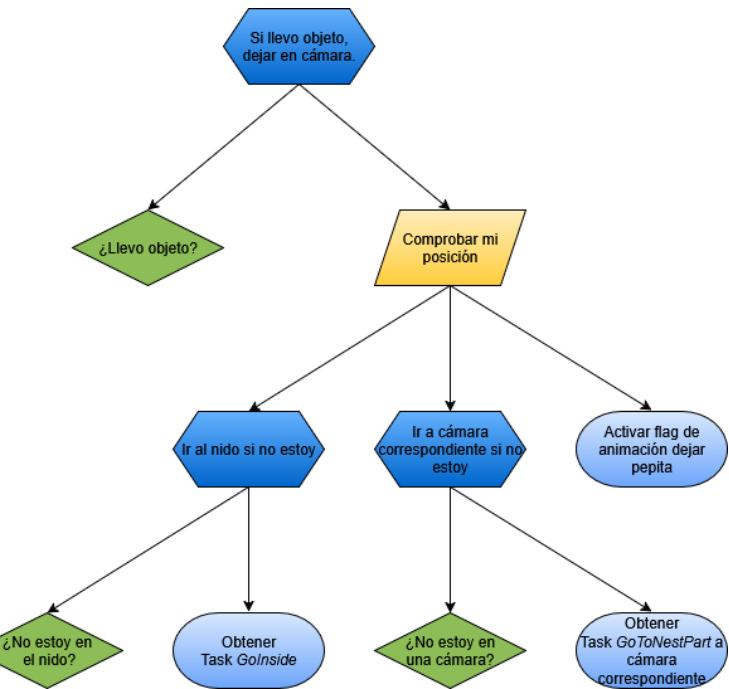


Figure 58: Subárbol de seleccionar Task para dejar comida en el nido

En el tercer nivel del subárbol se usan secuencias. En cada secuencia, se comprueba que la hormiga no se encuentra en el siguiente lugar. Si se encuentra, la secuencia devuelve *Failure* y el selector padre pasa al siguiente nodo.

1. La primera secuencia asigna la tarea de ir al nido si no se encuentra la hormiga en el nido.
2. La segunda secuencia asigna la tarea de ir a una cámara del nido para dejar comida si no se encuentra en uno. En dicho nodo, se selecciona un punto aleatorio en el suelo de una cámara correspondiente donde la hormiga dejará su objeto. Si lleva comida, la cámara será la cámara de comida más vacía del nido o la cámara de la reina si requiere más comida. Si lleva un huevo, será la cámara de huevos más vacía del nido. Si no hay cámaras de comida o huevos, se dejará en una cámara aleatoria.
3. Si la hormiga se encuentra dentro de la cámara correspondiente, se activará la animación de soltar en el suelo el objeto. Durante la animación, el árbol de comportamiento no es llamado. La animación llama a la función de soltar objeto, que deja a la hormiga sin ningún *Task*.
4. El tercer nodo, de acción, ejecuta la función *SenseTask()*. Esta función detecta el área alrededor de la hormiga buscando tareas que completar, usando la función *OverlapSphereNonAlloc()* para detectar colisiones con objetos de tipo pepita de maíz, mazorcas y puntos de excavación. Si detecta uno o más objetos válidos, selecciona uno de ellos y asigna una tarea a la hormiga según qué tipo de objeto. Ignora pepitas de maíz ya en cámaras de comida o para la reina, *DigPoints* seleccionados por otras hormigas y mazorcas que se han quedado sin pepitas. La prioridad de selección es:
 1. *DigPoint*. Si la hormiga ha detectado un punto de excavación, ignora todos los demás objetos. Selecciona el más accesible y más cercano, en ese orden de prioridad. Se comprueba lo accesible que es un *DigPoint* mediante la función *ReachableScore()*.
 2. Pepita o mazorca. Selecciona el más cercano, y si ha detectado una mazor-

ca, se añade a la memoria de la hormiga, para luego compartirlo con el nido. Si es la primera vez que la mazorca se detecta y, por tanto, aún está invisible, deja de estarlo.

Después de seleccionar el objeto *SenseTask()*, crea un objeto *Task* según el tipo de objeto y se lo asigna a la hormiga.

4. El cuarto nodo le pide a la clase *Nest* alguna tarea usando la función *GetNestTask()*. Esta función asigna la mayoría de los trabajos del nido, mandando a las hormigas a realizar una de cinco posibles *Tasks*. Usa una lista barajada de opciones para decidir en qué orden comprobar si hay alguno de los distintos tipos de tareas disponibles. La hormiga itera sobre la lista hasta encontrar una tarea del tipo indicado. La lista contiene 10 entradas, de las cuales algunas se repiten para aumentar la posibilidad de que se le asigne una tarea de su tipo a la hormiga. Los distintos tipos de tareas que pueden ser asignados son:

1. Un *Task* tipo *Dig* para excavar un *DigPoint* en el nido. Tiene un 2/10 de posibilidad de comprobar si hay puntos de excavación en el nido. Si no quedan, pasa a comprobar el siguiente en la lista.
2. Un *Task* de tipo *CollectFromCob* para recolectar comida de una de las mazorcas conocidas por el nido. Tiene un 2/10 de posibilidad de ser comprobado.
3. Un *Task* de tipo *GetEgg* o *GetCorn* para mover un objeto de un lugar del nido a otro. Comprueba si hay objetos fuera de lugar, y los mueve a sus lugares correspondientes. También se ocupa de asignar las tareas de llevarle comida a la hormiga reina.
Tiene un 4/10 de posibilidad de ser asignado, considerándose una prioridad en el nido. Esto se debe a que la hormiga reina siempre debe tener comida asignada, los huevos crecen más rápido en sus cámaras y deben siempre ser depositados allí, y que la comida fuera de cámaras de comida no cuenta para la puntuación del jugador y no es considerada para ser llevada a la reina. Si no hay objetos fuera de lugar y la hormiga reina tiene ya 5 piezas de comida a su disposición (o ya hay suficientes hormigas llevándole comida), se comprueba el siguiente tipo de tarea de la lista.

4. Un *Task* de tipo *Wait*. En la vida real, varias especies de hormigas no duermen como tal, sino que toman descansos cortos de aproximadamente un minuto o menos. Se quiso simular esto dándole un 1/10 de posibilidad de que una hormiga se mueva a un punto dentro de una cámara aleatoria y espere allí un periodo corto de tiempo. Esta tarea siempre está disponible, pero solo se realiza si menos del 20% de las hormigas tienen la misma tarea, para prevenir que todas las hormigas descansen si ninguna otra tarea está disponible.
5. Un *Task* de tipo *Explore*. Hay una 1/10 de posibilidad de que la hormiga considere salir del nido a explorar. Esto lo hace solo si menos de 20% de las hormigas se encuentran explorando. De esta forma, mientras haya tareas en el nido, solo una pequeña porción de hormigas sale a explorar.
5. El quinto nodo es selector, y le da a la hormiga la tarea de salir del nido si no está ya fuera para que pueda empezar a explorar. Como solo puede llegar a este nodo tras no recibir ninguna otra tarea del nido, se asegura de que, mientras no haya mazorcas conocidas y tareas en el nido, todas las hormigas se ocupen de buscar

la siguiente fuente de comida.

6. Si se llega al nodo sexto, la hormiga se encuentra fuera del nido. Se le asigna una tarea de explorar el mapa, incluido un contador para ver cuantas veces seguidas debe explorar su área alrededor inmediato.

11.4.2 Sección de ejecución de tarea *Task de la hormiga trabajadora*

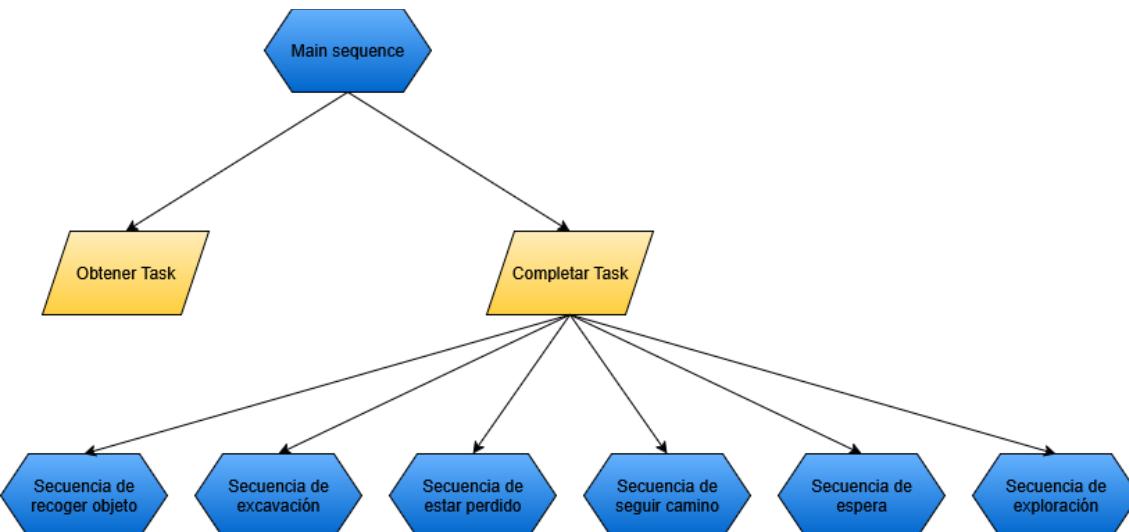


Figure 59: Árbol con el subárbol de ejecución de tareas de la hormiga trabajadora expandida

Cada nodo debajo del nodo selector de ejecución de tareas se encarga de tomar las acciones necesarias para completar una tarea específica. Son todos nodos de secuencia cuyo primer subnodo comprueba si la hormiga tiene el tipo de *Task* en cuestión. Si devuelve false, la secuencia falla y el selector pasa a la siguiente secuencia. Las secuencias se describen a continuación:

1. *Secuencia de recoger objeto*: Después de comprobar que la tarea de la hormiga es de tipo recoger objeto, se comprueba que el objeto a recoger aún es un objetivo válido mediante la función *IsValid()*. Según el tipo de tarea, esta función comprueba si la hormiga aún puede ejecutarla. En el caso de *GetCorn*, *GetEgg* y *CollectFromCob*, *IsValid* comprueba si el objeto aún existe, no ha sido ya recogido por otra hormiga y si aún se encuentra donde estaba al obtener la hormiga el camino. En cuanto alguna de estas condiciones no se cumple, la hormiga recibe un *Task* de tipo *NoTask* y sale de la secuencia.

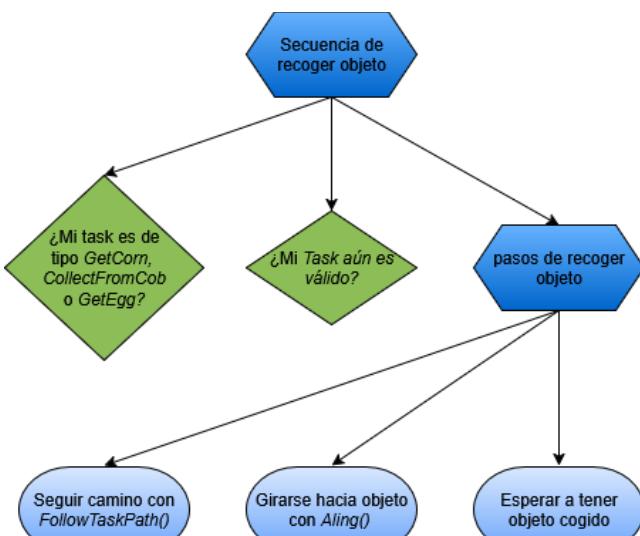


Figure 60: Subárbol de secuencia de recoger pepita

Si es válida la tarea, la hormiga sigue el camino con *FollowTaskPath()*. Esta función devuelve *Running* mientras aún quede camino y *Success* si no, para pasar al siguiente nodo. Luego, si no se ha alineado aún con el objeto, se alinea con *Align()*. En cuanto está alineado, se activa la animación de recoger objeto. La ani-

mación misma llama a la función *PickUpEvent()*, la cual se encarga de mover el objeto a la hormiga y borrar la tarea de recoger de la hormiga. El último nodo solo espera a que la hormiga deje de tener como tarea uno de tipo *GetCorn*, *GetEgg* o *CollectFromCob*.

2. *Secuencia de excavación*: Similar a la secuencia de recoger objeto, en esta, después de comprobar el tipo de tarea y si es válido o no, la hormiga se mueve hacia el punto de excavación, se alinea con él, activa la animación de excavar y se espera a que termine la acción. En este caso, la función *IsValid()* comprueba si el punto de excavación aún existe en el diccionario *digPointDict* de la clase *DigPoint*. Si no se encuentra dentro, significa que ya ha sido excavado, y la hormiga puede abandonar la tarea, siendo esta reemplazada por uno de tipo *NoTask*.

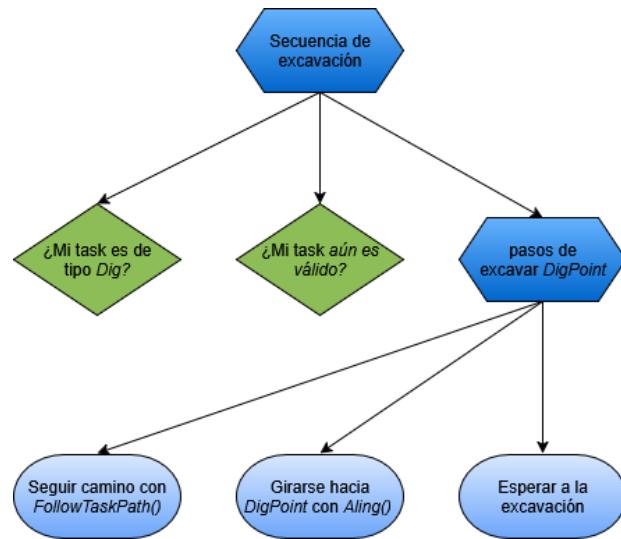


Figure 61: Subárbol de la secuencia de excavación

3. *Secuencia de estar perdido*: Después de comprobar que se trata de la tarea en cuestión, la secuencia ejecuta la función *FollowTaskPath()* para seguir el camino de la tarea. Si termina este camino, ejecuta la función *CheckLostStatus()* en el siguiente nodo. Esta función comprueba si la hormiga ha llegado al nido, en cuyo caso le pone una tarea de tipo *NoTask*. Si no, asigna una nueva tarea de tipo *Lost* a la hormiga, para que se mueva otra distancia y con suerte encuentre el nido. Además, mediante el contador de la hormiga, cada 10 veces que la hormiga completa una tarea de tipo *Lost*, comprueba si puede encontrar un camino al nido asignando a la hormiga una tarea mediante *GetInsideTask()*, por si se ha topado con un camino de feromonas que lleva al nido. Este llama a la función *GetKnownPathToMapPart()*, y si hay camino al nido, devuelve una tarea *GoInside* con el camino a seguir. Si no, devuelve una tarea de tipo *Lost*, y el contador se reinicia.

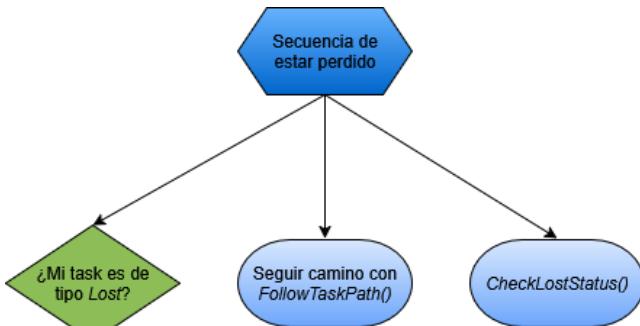


Figure 62: Subárbol de secuencia de estar perdido

4. *Secuencia de seguir camino*: Esta secuencia contiene los nodos necesarios para seguir un camino, y se usa para completar las tareas que consisten en ir de un punto a otro: *GoInside*, *GoOutside*, *GoToChamber* o *GoToTunnel*. Después

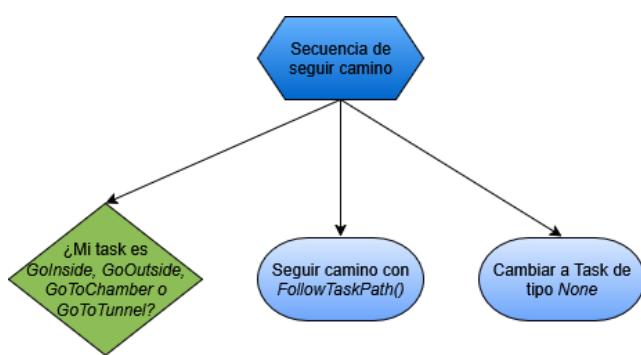


Figure 63: Subárbol de secuencia de seguimiento de camino

de comprobar el tipo con el primer nodo, el segundo nodo de la secuencia sigue el camino con *FollowTaskPath()*. Si el camino ha sido completado, el último nodo le asigna una tarea *None* a la hormiga.

5. *Secuencia de espera*: La secuencia que se usa para mantener inmóvil un rato a la hormiga usa su contador para gestionar cuánto tiempo de espera queda. Después de comprobar el tipo con el primer nodo, el segundo baja el contador en 1. El tercer nodo, de condición, comprueba si el contador ha llegado a 0. Si es verdad, llega al cuarto nodo que termina la secuencia, poniéndole una tarea *None* a la hormiga.

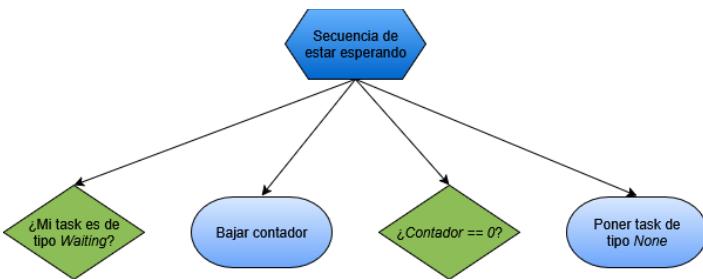
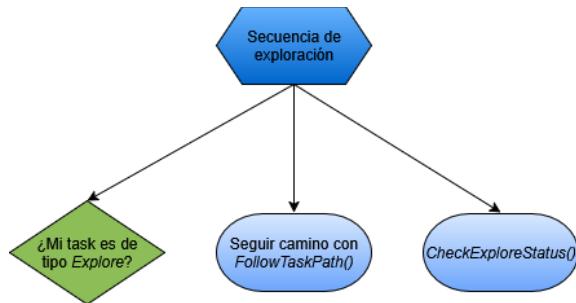


Figure 64: Subárbol de la secuencia de espera

6. *Secuencia de exploración*: La secuencia de exploración se parece a la de *Lost*. Cuando se completa el pequeño camino de exploración de los alrededores de la hormiga, se llama a la función *GetExploreStatus()*, la cual se encarga de 3 cosas:

1. Comprobar si hay una tarea cercana con *SenseTask()*. Si hay, la hormiga deja de explorar y se asigna la tarea.
2. Comprobar si no se ha llegado al límite de exploración. Cuando a una hormiga se le asigna la tarea *Explore* guarda un contador de veces que debería explorar su entorno directo. Si se ha llegado al límite, el contador se reinicia y la hormiga recibe una tarea de tipo *None*.
3. Si no se ha llegado al límite, a la hormiga se le asigna una nueva tarea *Explore* mediante *GetExploreTask()* sin obtener nuevo valor para el contador. Al contador se le resta uno.

Figure 65: Subárbol de la secuencia de exploración



El límite de exploración existe para que las hormigas no se queden indefinidamente fuera del nido explorando. El número de veces que debe repetir la tarea tiene también en cuenta cuánto tardan las feromonas en desaparecer, para que la hormiga no se quede sin un camino de feromonas que lo lleve al nido.

11.4.3 Sección de obtención de tarea *Task de la hormiga reina*

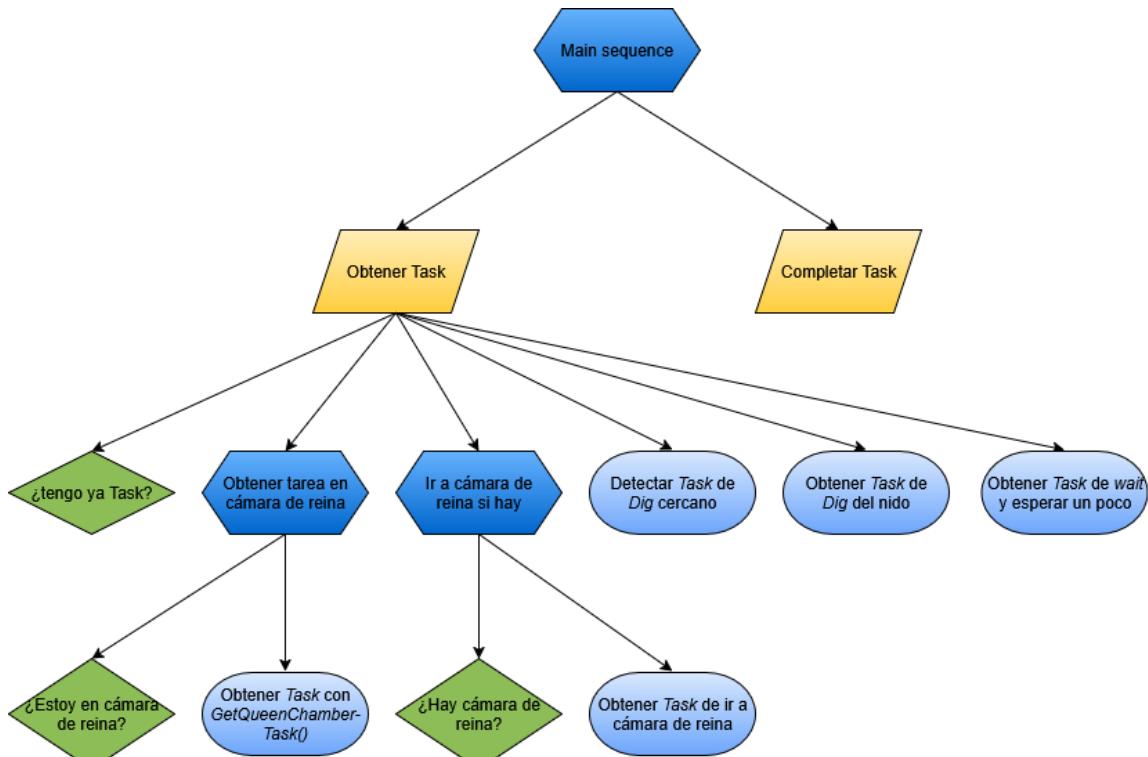


Figure 66: Subárbol de elección de tarea de la hormiga reina

La hormiga reina tiene menos diversidad de toma de decisiones que las hormigas trabajadoras. Al principio del juego, la reina se ocupa de excavar las partes iniciales del nido. En cuanto el nido ha sido establecido, se mantendrá dentro de una de las cámaras de reina y se ocupará tan solo de comer y poner huevos. En cuanto las primeras hormigas trabajadoras nacen, estas son las que se ocupan de buscar y recolectar comida, además de gestionar otras tareas del nido.

El nodo de selección de obtención de tareas de la hormiga reina decide en gran parte qué tarea realizar según la existencia de una cámara de reina en el nido y si la reina se encuentra en este. El funcionamiento de los subnodos es el siguiente:

1. El primer nodo es condicional, y comprueba si el *Task* de la reina es de tipo *none*. Si no lo es, se pasa al nodo de selección de compleción de tareas.
2. El segundo nodo es de secuencia, y se ocupa de asignar tareas a la hormiga reina si se encuentra dentro de una cámara de reina.
 1. El primer subnodo es condicional y sale de la secuencia si la reina no se encuentra dentro de una cámara de reina excavada.
 2. El segundo nodo llama a la función *GetQueenChamberTask()*, que siempre le asigna una tarea a la hormiga reina. Esta función comprueba si la hormiga reina tiene suficiente energía para dar a luz. Si tiene más de 5 energía y queda espacio en el nido, se le restan 5 y se inicializa un huevo de hormiga en el abdomen de la reina. Luego se le asigna una tarea *GoToChamber* con un camino a un punto aleatorio en la cámara de la reina. De esta forma se mueve a una nueva posición después de dar a luz para revelar el huevo si este se encuentra aún dentro del abdomen de la reina.

Si la hormiga no tiene la suficiente energía o no hay suficientes cámaras de huevo para almacenar la cantidad total de huevos de hormiga en el nido, la

función comprueba si hay comida en la cámara de la reina. Si hay, se le asigna un *Task* de tipo *Eat* para que coma una pepita cercana y así ganar más energía.

Si ni tiene suficiente energía para dar a luz ni hay comida en la cámara de la reina, obtiene una tarea de tipo *wait* para esperar un rato antes de comprobar otra vez si puede hacer algo.

3. El tercer nodo es de secuencia, y solo se puede llegar si la hormiga no se encuentra dentro de una cámara de reina. Manda a la hormiga reina a una cámara de reina para que pueda dar a luz. Su primer subnodo sale de la secuencia si no existe una cámara de reina excavada en el nido y su segundo asigna un *Task* de tipo *GoToChamber* a la primera cámara de reina.
4. El cuarto nodo usa la función *SenseDigPoint()* para detectar si hay un punto de excavación cercano. Si hay, se le asigna un *Task* de tipo *dig* con el más cercano a la reina.
5. El quinto nodo pide una tarea de excavación a la clase *Nest*. Si hay algún *DigPoint* en el nido no siendo excavado por otra hormiga, se le asigna a la reina como *Task* de tipo *Dig*. Solo se llega a este nodo si el nido aún no tiene excavada una cámara de reina, y se suele solo llegar a ella cuando se empieza el juego sin nido y solo la reina.
6. El sexto nodo pone a la reina en estado de espera unos segundos, asignándole un *Task* de tipo *wait*. Esto solo ocurre al principio del mapa, cuando el jugador no ha decidido aún dónde excavará el nido.

11.4.4 Sección de ejecución de tarea Task de la hormiga reina



Figure 67: Subárbol de compleción de tareas de la hormiga reina

El subárbol de compleción de tareas de la hormiga reina es similar al de la hormiga, dividiendo la realización de las tareas distintas en secuencias individuales. Comparte con el subárbol de la hormiga las secuencias de compleción de excavar, estar perdido, seguir caminos y esperar. La hormiga reina nunca se encargará de explorar y mover objetos, así que no tiene esas secuencias en común con la hormiga trabajadora. Sustituye la secuencia de recoger objetos por la secuencia de comer, que funciona de forma muy similar.

En la secuencia de comer pepita, los primeros dos subnodos se aseguran de que la hormiga reina tiene un *Task* de tipo *Eat* y que este sigue siendo válido (la pepita no ha dejado de existir, no ha sido movida y no lo lleva otra hormiga). Si es válido, procede a la secuencia de pasos necesarios para comer la pepita:

1. Moverse hacia la pepita mediante la función *FollowTaskPath()*.
2. Una vez al lado de la pepita, girar hasta estar mirándola con *Align()*.
3. Esperar a que termine la animación de comer el objeto.

12 Sistema de guardado y cargado

Uno de los aspectos más importantes de un videojuego con progreso es la capacidad de poder guardar la partida y cargarla para seguir jugando posteriormente. Desde el principio del proyecto ya se había implementado un sistema de guardado y cargado del mapa, pero fue simple e ineficiente en cuanto al espacio requerido. Luego se implementó la codificación para ahorrar espacio, y finalmente la serialización para guardar los otros elementos del juego.

12.1 Guardado y cargado del mapa

12.1.1 Texto plano

La primera versión del guardado fue a texto simple. Se implementaron las funciones LoadMap() y SaveMap:

- *SaveMap()* se encargó de escribir al archivo map.txt la información del mapa usando la clase nativa StreamWriter.
 1. Escribe las dimensiones X, Y, Z del mapa. Más tarde escribiría también las dimensiones de los chunks.
 2. Itera sobre todos los elementos del mapa, escribiéndolos al StreamWriter separados por espacios.
 3. Cierra y guarda el archivo map.txt.
- *LoadMap()* se encargó de cargar el archivo map.txt si hubiese y actualizar el mapa con él.
 1. Convierte todo el texto del archivo map.txt a un string mediante la función ReadAllText() de la clase File.
 2. Usando la función split() de la clase string, convierte el string a un array de string con todos los valores individuales.
 3. Usando Parse para convertir los strings a integers, asigna los primeros valores del archivo que describen el tamaño del mapa a variables. También el tamaño de los chunks cuando estos se implementaron.
 4. Asigna al mapa una nueva matriz con el tamaño del mapa obtenido.
 5. Itera sobre los valores restantes del archivo, usando Parse para guardarlos como valores en la matriz mapa.
 6. Elimina la malla del terreno existente, y genera uno nuevo usando el nuevo mapa.

Debido a no usar ninguna forma de compresión de datos, el archivo resultante solía ser grande. Usando un mapa de tamaño 50x100x50, el peso del archivo resultante fue de 1036 Kb, casi exactamente un megabyte.

12.1.2 SharpSerializer

Se empezó a usar SharpSerializer para guardar los datos, probando primero guardar los datos del terreno. Se probó con el mismo mapa que antes de dimensiones 50 por 100 por 50. El tamaño resultante fue de 43 Mb, un incremento de 43 veces. Esto se debía a que cada float de la matriz de terreno se guardaba en el JSON con el formato:

```
'<Item indexes="60,21,92">
`<Simple value="1" />
`</Item>'
```

SharpSerializer permite comprimir sus JSON a binario, y se usó para ver si pudiera bajar significativamente el espacio ocupado. Sin embargo, el tamaño de archivo seguía siendo bastante alto, ocupando 9 Mb.

Los valores del mapa y sus dimensiones en WorldGen son estáticos, pero no se pueden serializar datos estáticos. Por eso se creó una clase aparte llamada *GameData*, en la que se guardarían todos los datos de forma no estática para poder ser serializados, y que se deserializaría para que WorldGen pudiera absorber la información en sus variables estáticas en una nueva escena al cargar la partida.

12.1.3 Encodificación y decodificación

Para bajar el tamaño de la sección del mapa en el archivo serializado, se decidió crear un codificador y un decodificador. La idea era convertir los datos del mapa en el formato más pequeño posible, y se eligió que fuera un array de bytes.

Para que los valores de la matriz concordaran más con este formato, se editó el sistema de marching cubes para que, en vez de representar el campo escalar con valores entre 0 y 1, siendo 0.5 el centro, se usarán los valores enteros desde 0 a 255, situándose la isosuperficie en el valor 127.5. Esto se hizo porque con un byte se pueden representar los valores de 0 a 255.

El modelo de compresión de la matriz se centró en tomar ventaja de la repetición de datos: casi todo el cielo tenía valores 0, al igual que la gran mayoría del subsuelo contenía los valores 255. Se usarán bytes de etiqueta para informar sobre los siguientes 8 bloques de datos. Cada bloque de dato podría ser de dos tipos:

- Etiqueta valor 0: bloque formado por un byte singular, que contiene el valor del siguiente elemento.
- Etiqueta valor 1: bloque formado por dos bytes. El primero indica cuántos valores iguales hay seguidos. El segundo contiene el valor que hay repetido.

Ya que los bytes pueden contener valores de 0 hasta 255, se pueden guardar hasta 255 bytes seguidos repetidos en un bloque. El algoritmo se asegura además de iterar sobre los elementos de la matriz de tal forma que primero se mueve verticalmente, y luego en el plano horizontal. Esto proporciona la mayor cantidad de valores seguidos con valores iguales.

El tamaño del archivo serializado, en formato no comprimido sino XML, fue de 40 Kb. Una mejora en un factor de 1000 comparado con la versión serializada original.

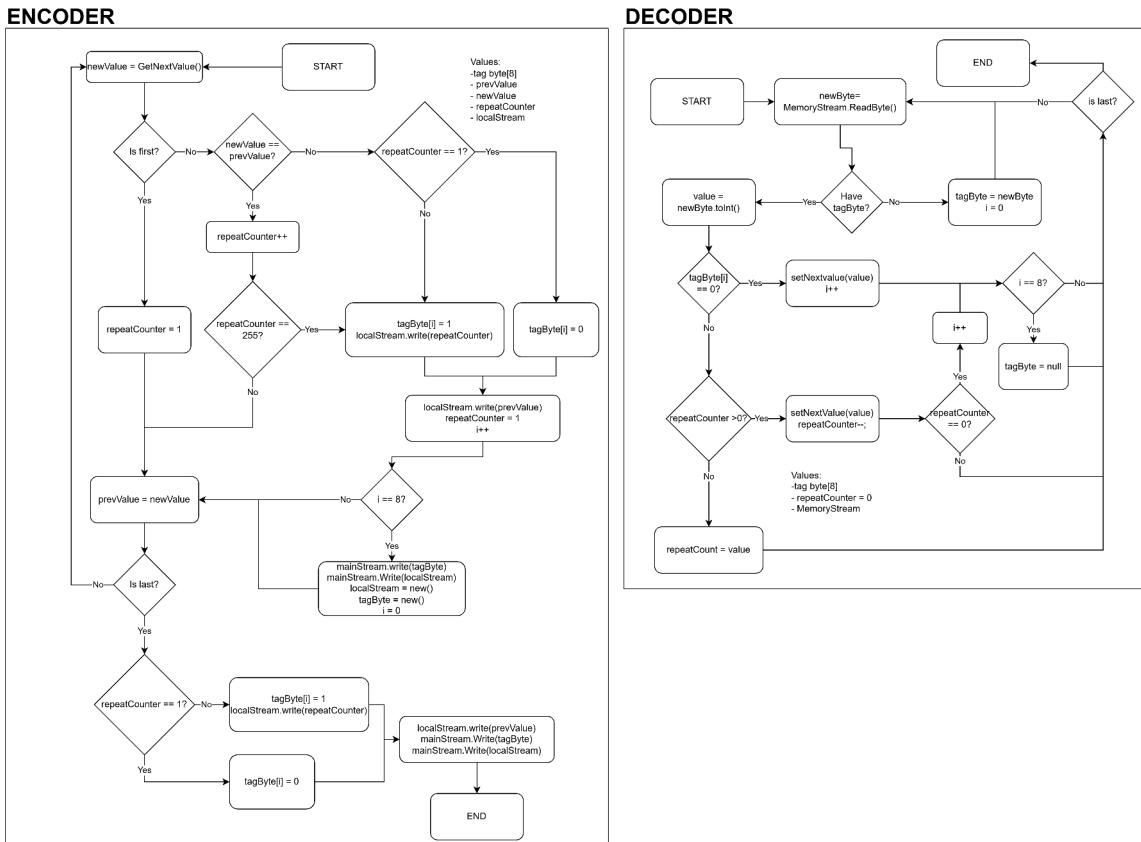


Figure 68: Visualización de los algoritmos de codificación y decodificación

12.1.4 Redundancia del mapa

La eficiencia de guardado de los datos del terreno mediante la codificación del campo escalar es máxima al guardar grandes cantidades de valores iguales consecutivos, pero sufre cuando se encuentran valores distintos. En un mapa muy editado por el jugador, suele haber grandes cantidades de valores consecutivos desiguales, lo cual puede incrementar substancialmente el tamaño del archivo de guardado.

Sin embargo, no todos los valores del campo escalar modificados por el jugador afectan la forma del terreno, ya que este solo se establece entre valores que se encuentren a lados opuestos de la superficie. Puntos del campo escalar debajo de la isosuperficie sin puntos adyacentes con valores sobre la isosuperficie no afectan al terreno, al igual que puntos con valores por encima de la isosuperficie sin puntos adyacentes con valores debajo de la isosuperficie.

Para ahorrar el máximo espacio posible, se creó la función *ApplyTerrainRedundancy()*. Este iteraría sobre los valores de los dos campos escalares, identificando los puntos redundantes. Aquellos que se encontraran sobre la isosuperficie serían puestos a 255, y aquellos debajo de la superficie a 0. Esta función se usaría cada vez que se guardara el mapa, antes de codificar sus campos escalares.

12.2 Qué es SharpSerializer

SharpSerializer es un serializador en binario y XML de código abierto para .NET. La serialización es el proceso de convertir el estado de un objeto (sus datos y propiedades) en un formato que puede ser almacenado o transmitido, normalmente en archivos JSON, XML o binario. Luego esos archivos pueden ser deserializados y convertidos en los mismos objetos en otra sesión de ejecución de código.

12.3 Por qué usar SharpSerializer

Para poder guardar y cargar los muchos objetos dinámicos del juego en archivos de juego, en vez de escribir un complejo y probablemente ineficiente convertidor de datos a archivo txt y viceversa, se decidió usar un serializador. La idea era guardar toda la información de los objetos activos en la escena al guardar, y al cargar, inicializar todos los objetos con los datos guardados.

Se consideró usar el serializador de Unity, pero aunque este permitía serializar las clases y estructuras de datos, no resultaba lo suficientemente flexible: no permitía integrar la inicialización de los distintos objetos. SharpSerializer, en cambio, sí proporcionaba esta flexibilidad: es un serializador escrito para C# en general, enfocado en proporcionar serialización rápida y simple.

12.4 Serialización de los objetos con SharpSerializer

12.4.1 ¿Qué hay que guardar?

Ya que no se guardarían los objetos del juego como tal, sino sus datos más importantes para luego recrear dichos objetos en una nueva escena cargada, hubo que crear contenedores de dichos datos que pudieran ser serializados.

Los objetos que tenían que ser guardados eran:

- Las hormigas
- Las pepitas de maíz
- Las mazorcas
- Las partes del nido
- Las tareas de las hormigas
- Los caminos que las hormigas están siguiendo dentro de la clase tarea, y por tanto también las superficies que los componen.

En cuanto a datos del juego, no objetos:

- Las feromonas (es un diccionario, no objetos instanciados individuales)
- Las relaciones entre los distintos objetos
- Los puntos de excavación no inicializados aún
- Cuáles de los puntos de excavación han sido inicializados
- El conocimiento del nido sobre las mazorcas encontradas
- El tiempo de juego y si el jugador ha ganado o no en el mapa

Para mantener un sistema simple, se decidió seguir usando la clase GameData que contenía los valores del terreno, añadiéndole una serie de clases de información para guardar los datos más importantes de los distintos objetos. De esta forma, todos los datos del juego se encontrarían en una singular clase serializable.

12.4.2 ¿Cómo se hace que una clase sea serializable?

Gracias a la documentación de SharpSerializer y prueba y error durante la implementación de la serialización, se descubrieron ciertos requerimientos necesarios para poder serializar todos los datos correctamente:

1. Usar propiedades en vez de variables: SharpSerializer ignora variables y solo guarda las propiedades de las clases al serializarlas. Hubo que convertir todas las variables a propiedades en las clases que se serializarían.
2. No usar valores estáticos. Estos también eran ignorados, y fue uno de los motivos por los cuales crear la clase *GameData* para copiar los valores estáticos de *WorldGen* a propiedades no estáticas en vez de serializar directamente *WorldGen*.
3. Todas las propiedades deben ser públicas para poder ser accedidas por el SharpSerializer.
4. Las clases por defecto son serializables a JSON y XML, pero para poder ser serializadas a binario, hace falta que tengan el atributo `SerializableAttribute`. Para que una clase tenga este atributo, hace falta marcarlo escribiendo “[serializable]” encima de su definición.
5. Los enums no son serializables. Para poder serializar los enums de la clase *Task* y *NestPart*, hubo que ordenar los posibles valores de cada enum y guardarlos según su índice.
6. Los *Vector3*, *Vector3Int* y *Quaternion* no se guardaban. Los contenedores que los contuvieran incluirían entradas para cada uno, con un número de referencia, pero ninguna línea a la que apuntaban estos. SharpSerializer no podía obtener y guardar los nombres de estas clases y, por tanto, ignoraba sus valores.

Se decidió crear las clases *serializableVector3*, *serializableVector3Int* y *serializableQuaternion*, que seguían todos los requerimientos anteriores para poder ser serializables y contenían funciones de conversión desde y a la clase original. Se usaron en las otras clases de información para guardar posiciones, orientaciones y algunos otros valores.

12.4.3 Relaciones entre objetos e indexación

Las relaciones entre objetos se distinguen de los demás valores a guardar, ya que no se refieren a una clase de objeto en sí. El objeto pepita de maíz puede encontrarse siendo sujetado por una hormiga, aún unido a una mazorca o suelto dentro del nido, y cada uno de estos estados requiere que se encuentre como objeto hijo de la hormiga o la mazorca o nada. Este tipo de relación y una referencia al objeto padre (si hay) deben ser guardados y cargados.

Normalmente, se guardaría el tipo de relación y una referencia al objeto con el que se relaciona, pero ya que todos los objetos serían inicializados de nuevo cada vez que se cargara el juego, las referencias antiguas no apuntarían a los mismos objetos. Hubo que encontrar una forma de identificar cada hormiga, mazorca y pepita de maíz individualmente y guardar esos identificadores.

Se decidió indexar todos los distintos objetos que podían formar parte de una relación. La idea es asignar un valor numérico único a cada objeto según su clase y guardar cada instancia de la clase dentro de un diccionario estático indexado por dicho valor. De esta forma, sabiendo el tipo de objeto y teniendo su identificador, sería fácil acceder a él.

En las clases de objetos con potencialmente múltiples copias en juego se realizaron los siguientes cambios:

1. Cada clase recibe una variable entera llamada `id` para identificarla. Asimismo, las clases de información respectivas guardarán una copia de dicha variable para ser serializada.

2. Cada clase obtiene un diccionario estático, donde se guardan referencias a todos los objetos de la clase usando sus identificadores como claves. De esta forma se podría acceder a cualquier objeto de una clase mediante su identificador desde cualquier otra clase.
3. Cuando se crea un objeto de una de las clases, se le asigna un identificador. Si es un objeto nuevo, se le asigna el primer valor numérico entero no en uso para su clase. Si es un objeto cargado a partir de su clase de información, se le asigna el identificador que tenía anteriormente.
4. Después de asignarle un identificador, las funciones creadoras añaden el objeto al diccionario respectivo.
5. En la función *OnDestroy()* de cada clase, la cual es llamada por Unity cuando se elimina un objeto con dicha clase, se elimina la entrada del objeto del diccionario que lo contenga.

12.4.4 Clases Información

Para cada objeto que se quiere guardar, se creó una clase que contendría la información necesaria para poder cargar el objeto en una escena nueva de tal forma que reanudase sus acciones (si tuviera) como si nunca se hubiese guardado ni cargado. Ya que se quiso ahorrar espacio y mantener simple el proceso, los objetos fueron modificados para poder funcionar con los mínimos datos guardados posibles. Algunas de las modificaciones fueron:

- Convertir datos generales en estáticas compartidas entre todas las instancias de la clase. Por ejemplo, la velocidad de las hormigas, el tamaño de una pepita, etc.
- Convertir funciones a estáticas, para evitar la necesidad de referencias a objetos con funciones necesarias. El mayor ejemplo de esto es *WorldGen*, al que los objetos *DigPoint* requerían referencias.
- Eliminar variables no estrictamente necesarias, y editar funciones para usar los mínimos variables posibles y así descartar unas más.
- En el caso de las hormigas, anteriormente su árbol de comportamiento guardaría la posición en la que se encontraba cada *FixedUpdate()*. Esto se eliminó editando su código y reestructurando el árbol de comportamiento para poder funcionar sin guardar su posición.

Las clases de información que se crearon fueron:

- **AntInfo:** Guarda los datos relevantes de un objeto *Ant*. Estos datos son:

Su identificador, el identificador de la hormiga que lo va a recoger si aún es un huevo, su edad, un objeto *TaskInfo* para guardar su tarea, un booleano que designa si lo controla el jugador, su posición, su orientación, un booleano que designa si tiene algo agarrado, el contador que usa para gestionar el árbol de comportamiento, y un *HashSet* que representa su memoria de las mazorcas que ha encontrado.

- **QueenInfo:** Guarda los datos relevantes de un objeto *AntQueen*, la reina hormiga. Estos datos son:

Un objeto *TaskInfo* para guardar su tarea, su posición y orientación, el contador que usa para gestionar su árbol de comportamiento y la cantidad de energía que tiene para poner huevos.

- **CornInfo.** Guarda los datos relevantes de un objeto *Corn*. Los datos son:

Su identificador, el identificador de la hormiga que lo va a recoger, su posición y su orientación.

- **CornCobInfo.** Guarda los datos relevantes de un objeto *CornCob*. Estos son:

Su identificador, su posición, su orientación y un diccionario que contiene los identificadores de los objetos **Corn** que se encuentran en la mazorca, indexados por el número de posición en la mazorca.

- **TaskInfo.** Guarda datos relevantes a un objeto *Task*. Estos son:

El identificador del *DigPoint* que la hormiga debe excavar (si el task no apunta a un punto de excavación, este valor es -1), el identificador del objeto *Corn* o objeto *hormiga* (huevo) que la hormiga debe coger (si no apunta a un objeto, su valor es -1), la posición a la que hay que ir si el task se trata de ir a un lugar, el índice del tipo de tarea y la lista de *SurfaceInfo* que representan el camino a seguir para llegar al objetivo de la tarea.

- **NestPartInfo.** Los datos de *NestPart* guardados son:

Los puntos de comienzo y final, su radio y el índice del enum del tipo de sección de nido del que se trata.

- **SurfaceInfo.** Guarda los datos importantes de un objeto *CubeSurface*:

La posición del cubo que contiene la superficie y un byte que representa el grupo de superficie. Se usa un byte en vez de un array de booleanos, ya que al serializar un array, cada entrada ocupa 4 líneas y se quiso ahorrar espacio.

Cada una de estas clases se definió dentro de la clase *GameData*, ya que es el único sitio donde se usarían y facilitaba su gestión. Para cada uno de ellos, se escribieron funciones de conversión del objeto original a su clase de información. Luego, *WorldGen* obtuvo referencias a todos los objetos originales de la escena y funciones que podrían hacer copias de estos a partir de las clases de información cargadas de *GameData*.

Las clases *DigPoint* y *DigPointData* eran tan simples que no hizo falta crear una clase auxiliar para su serialización. Lo único de la clase *DigPoint* que hizo falta guardar fue el diccionario de objetos *DigPointData* indexados por su posición. *DigPointData*, asimismo, solo contenía el valor que debía tener el terreno tras su excavación y una referencia al objeto *DigPoint* que lo representa si se ha inicializado en la escena. Se modificó para que se serializara el valor del terreno. Luego en *GameData* se guardaría el diccionario de *DigPointData* y una lista de posiciones de los *DigPoint* inicializados para inicializarlos al cargar la partida.

12.4.5 Variables en GameData

Las variables que *GameData* tuvo que copiar del juego y serializar para poder guardar y cargar una partida son:

- Los datos para representar el terreno: los arrays de bytes que representan el terreno y el terreno original codificados, las dimensiones del mapa y las dimensiones de los chunks.
- La posición y orientación de la cámara.
- Los HashSets que contienen todas las hormigas trabajadoras como *AntInfo*, la hormiga reina como *QueenInfo*, las pepitas de maíz como *CornInfo*, las mazorcas de maíz como *CornCobInfo*, las posiciones de los objetos *DigPoint* inicializados como *serializableVector3Int*, las partes del nido como *NestPartInfo* y las mazorcas conocidas por

el nido como enteros para representar sus identificadores.

- Que pepitas de maíz son cogidas por hormigas como un diccionario donde las llaves son los identificadores de las pepitas y los valores los identificadores de las hormigas.
- El diccionario de *DigPointData* indexadas por su posición (por algún motivo, cuando se usan como claves en un diccionario, los Vector3Int sí se serializan bien).
- El diccionario de las feromonas, sus valores de edad indexadas por su posición.
- La memoria de mazorcas encontradas por el nido como un *HashSet* de sus identificadores.
- El tiempo de juego como float y la variable booleana que guarda si el jugador ha ganado ya o no.

12.4.6 Guardado y cargado de mapas

Para poder guardar un mapa, se siguen los siguientes pasos:

1. Se crea un nuevo objeto *GameData* usando la función estática *SaveGame()* de su clase. Esta función toma todos los datos estáticos de *WorldGen* necesarios y los guarda como valores serializables. Luego crea y almacena las clases de información de todos los objetos del juego que hay que guardar. El objeto *GameData* devuelto contiene todos estos datos.
2. Se crea un objeto *SharpSerializer* con los ajustes de serialización adecuados.
3. Pasándole la dirección de archivo de guardado, se usa la función *Serialize()* del objeto *SharpSerializer* para serializar el objeto *GameData*.

Para poder cargar un mapa, se usa la función *LoadGame()* pasándole el nombre del archivo a deserializar, que toma los siguientes pasos:

1. Crea un objeto *SharpSerializer*, con el que se llama la función *Deserialize()* para deserializar la partida anterior guardada y devolver el objeto *GameData* cargado, llamándolo *loadedData*.
2. Se llama a la función *loadedData.LoadMap()*. Este decodifica el mapa y el mapa original, y asigna estos y los otros datos previamente guardados de *WorldGen* a sus correspondientes variables estáticas.
3. Usando la función *GenerateChunks()* de *WorldGen*, se generan los chunks.
4. Se usa la función *loadedData.LoadGameObjects()* para:
 1. Usar las funciones de inicialización de objetos de *WorldGen* para crear los objetos del juego a partir de las clases de información.
 2. Mirar el diccionario que indica qué pepitas son sujetadas por qué hormigas según sus identificadores. Usar esos identificadores para seleccionar las pepitas y hacer que las hormigas los sujeten.
 3. Para cada mazorca creada, mirar su diccionario de identificadores de pepitas conectadas. Para cada uno, poner la pepita inicializada con dicho identificador como hijo del objeto mazorca y ajustar su posición relativa a la mazorca para que parezca conectada.
 4. Asignar el diccionario de *DigPointData* a la estática de la clase *DigPoint*, y para cada identificador de *DigPoint* inicializado que se guardó, inicializar uno en la posición adecuada.

5. Asignar el diccionario de feromonas guardado a la estática de la clase *CubePaths*, donde se almacenan para usarse en el mapa.

12.5 Tipos de archivos guardados

Para gestionar los mapas guardados en memorias, hubo que separarlos por sus tipos. Por un lado estarían los mapas prehechos o creados por el jugador, que podría seleccionar para empezar partidas en ellos. Estos serán denominados simplemente mapas. Por otro lado estarían los mapas en los que el jugador ya empezó a jugar y decide guardar para reanudar posteriormente. Estos serán denominados partidas guardadas.

Ambos tipos de archivos se guardan y cargan de forma muy similar, pero usan funciones distintas para hacer los ajustes necesarios para diferenciar entre las partidas guardadas y los mapas. Las funciones de carga son:

- *StartMap()*: Carga un mapa prehecho o creado por el jugador. Requiere el nombre del archivo, y si este no existe, no ocurre nada. Después de comprobar la existencia del archivo, carga el mapa mediante la función *LoadMap()* y establece el valor del temporizador de tiempo jugado a 0. La función devuelve un valor booleano positivo si ha funcionado la carga de archivo, y falso si ha fallado o no existe.
- *LoadSaveFile()*: Carga una partida guardada, dado el número de partida guardada, un entero entre 1 y 3. Esta función solo se llama después de haber comprobado que la partida guardada existe, así que no devuelve un booleano para confirmar si ha funcionado o no. Usa la función *LoadMap()* para cargar el archivo con el nombre por defecto de la partida guardada.

Las funciones de guardado son:

- *SaveMap()*: Dado el nombre del mapa por parte del jugador, lo intenta guardar en memoria. Devuelve un booleano según si ha podido guardar la partida, para que se pueda notificar al jugador si el nombre elegido no es válido. Primero comprueba si existen ya mapas con el mismo nombre. Si existe un mapa prehecho con el mismo nombre, no guarda el mapa y devuelve False. Si no existe un mapa prehecho con el mismo nombre, lo guarda y devuelve true, aun si existiera un mapa hecho por el jugador con el mismo nombre. En ese caso, se sobreescribe. De esta forma, el jugador puede actualizar mapas previamente creados.
- *SaveGame()*: Dado el número de ranura de guardada en la que almacenar la partida guardada, sobreescribe el archivo previo si ya existía uno, y guarda el mapa. Después de implementar la clase *info*, crea y serializa un objeto de dicho tipo.

12.6 Guardando información sobre cada partida

Para que el jugador pueda distinguir entre las 3 partidas guardadas, fue necesario mostrar información relevante sobre cada una en la pantalla de selección de partida guardada. Una de las limitaciones de la deserialización de objetos es que no se puede hacer parcialmente. Por tanto, para poder obtener datos relevantes a una partida guardada, haría falta deserializar un archivo con un tamaño potencialmente alto, dependiendo de la extensión de su mapa y la cantidad de objetos que contiene.

Para prevenir potencialmente congelar el juego unos momentos al abrir la pantalla de partidas guardadas, se decidió crear una clase serializable aparte de *GameData* simple,

que guardaría solo los datos que se quisieron depictar en el menú. Esta nueva clase se denominó *Info*, y contendría los siguientes datos de una partida:

- El nombre del mapa que se eligió para la partida.
- La cantidad de comida recolectada en el nido.
- La cantidad de hormigas trabajadoras.
- El tiempo de juego.

Cuando un jugador decidiera guardar la partida, aparte de crear un archivo serializado para la clase *GameData*, se crearía también un objeto *info* que se serializaría y guardaría junto a este. Al entrar en la pantalla de carga de partidas, se deserializan los 3 objetos *info* de las partidas previamente guardadas para mostrar su información en la pantalla.

12.7 Estructura de archivos de guardado

Se decidió guardar todos los archivos serializados dentro de la carpeta *SaveData* en el archivo donde se encontrase el fichero de ejecución del juego. Dentro de esta carpeta se encontrarían las siguientes subcarpetas:

- Maps: Una carpeta con los escenarios prehechos por el desarrollador. Contendría algunos mapas simples con los que empezar a jugar por primera vez, además de algunos complicados para partidas posteriores cuando el jugador entendiera bien el juego.
- CustomMaps: Una carpeta con los escenarios creados por el jugador. Se separan de los prehechos, para poder diferenciarlos, ya que no se permite sobreescribir escenarios prehechos, pero sí escenarios hechos por el jugador.
- Save1, Save2 y Save3: Las tres carpetas que contendrían las partidas guardadas del jugador. Cada uno contiene, si hay una partida guardada, un archivo *data.xml* con la partida guardada serializada y un archivo *info.xml* con la clase *info* serializada.

Cada vez que se abre el juego, se comprueba la existencia de esta estructura de archivos en la carpeta del fichero de ejecución. Si falta alguna carpeta, esta es entonces inmediatamente creada.

13 Interfaz gráfica y modos de juego

La interfaz gráfica de usuario (GUI) es fundamental para la experiencia de juego dentro de un videojuego de simulación, ya que actúa como el principal canal de interacción entre el jugador y el sistema. Se crearon 3 interfaces gráficas generales: una para el menú principal, una para el modo de juego y una para el modo de edición de mapas.

En Unity, la creación de interfaz de usuario se hace mediante el uso de un objeto *Canvas* y su objeto *EventSystem* correspondiente. El primero contiene todos los objetos GUI que componen la interfaz, mientras que el segundo maneja todos los eventos que ocurren en el caso de

Mediante el uso de la escala automática del objeto *Canvas*, se pudo conseguir que todos los elementos de la interfaz gráfica escalaren bien con cualquier pantalla de ordenador. Los iconos, por ejemplo, se ajustaron para mantener siempre su relación de aspecto, y los paneles se ajustaron para moverse con los ejes de la pantalla.

En los modos de juego y edición de mapa, que usan raycasts para detectar objetos del terreno al presionar en alguna parte de la pantalla, se tomó en cuenta la interfaz gráfica, previniendo el uso de raycasts y lanzamiento de funciones de juego al pulsar el ratón mientras este se encontrara sobre la GUI.

Se desactivó también el modo de selección de botones de la GUI usando el teclado que está habilitado por defecto en el *EventSystem*, ya que esto podía provocar que el jugador sin querer pulsara botones mientras se intentaba mover por el mapa con las teclas WASD y espacio.

Los elementos del *canvas* que permiten al jugador introducir comandos pueden ser modificados para realizar funciones públicas de los scripts componentes de objetos en la escena al interactuar con ellos. No pueden ejecutar funciones estáticas ni privadas, por lo que cuando se quiso usar alguna de estas funciones, o había que cambiar su nivel de seguridad o hubo que crear una función pública aparte que las llamara.

13.1 Menú principal



Figure 69: Pantalla de menú principal del juego

El menú principal es la primera pantalla que el jugador ve cuando abre el juego. Muestra las 4 opciones posibles que el jugador puede tomar: empezar una partida nueva, cargar una partida previamente guardada, crear un mapa para posteriormente jugar en él, y salir del juego. El título del juego se muestra sobre las opciones y permanece allí en todos los submenús. Las cuatro opciones toman la forma de botones y realizan acciones al ser pulsados por el jugador.

En el *canvas* del menú, todos los objetos de la interfaz se agruparon según en qué parte del menú se encontraban. De esta forma, al pulsar uno de los primeros tres botones, estos solo tienen que desactivar el grupo de objetos actual y activar el grupo de objetos correspondiente al siguiente submenú. El cuarto botón sencillamente cierra el juego.

13.1.1 Clase GameSettings

El menú principal y los modos de juego y edición de mapa, ambos se encuentran en dos escenas diferentes. En la segunda escena, el modo de juego y el modo de edición de mapa funcionan de forma idéntica; lo único que cambia entre ellos es la interfaz gráfica, lo cual decide cómo el jugador puede interactuar con el mapa. Para poder diferenciar los dos, por tanto, hubo que señalizar a la segunda escena cuál de los dos interfa-

ces gráficos usar.

Al cambiar de una escena a otra, se pierde el acceso a las instancias de las clases de la escena previa. Esto dificultó la comunicación desde el menú a la segunda escena sobre qué interfaz gráfica usar. Por tanto, para guardar esta información de forma accesible para la segunda escena, se decidió usar una clase estática llamada *GameSettings*. En la escena del mapa, la clase *WorldGen* se encargaría de cargar un archivo de mapa o partida guardada o de crear un mapa nuevo plano según los valores de la clase *GameSettings*. Las variables de esta son las siguientes:

- *gameMode*: Un entero con valor 0 si el modo de juego seleccionado es editar el mapa, y 1 si el modo de juego es jugar en un mapa.
- *newMap*: Un booleano que señala si *WorldGen* debe cargar una partida guardada preexistente o cargar un mapa por primera vez.
- *fileName*: Un string con el nombre del archivo a cargar. Solo se considera si *newMap* es verdadero y *flatMap* es falso.
- *saveSlot*: La ranura de partida guardada a cargar. Solo se usa este valor si *newMap* es falso.
- *flatMap*: Un booleano que señala si crear un mapa plano o usar uno preexistente. Solo se considera si *newMap* es verdadero.
- *x_chunks*, *y_chunks*, *z_chunks* y *height*: Los cuatro valores que se usan en la creación de un mapa plano. Solo se consideran si tanto *newMap* como *flatMap* son verdaderos.

13.1.2 Submenú de selección de mapa

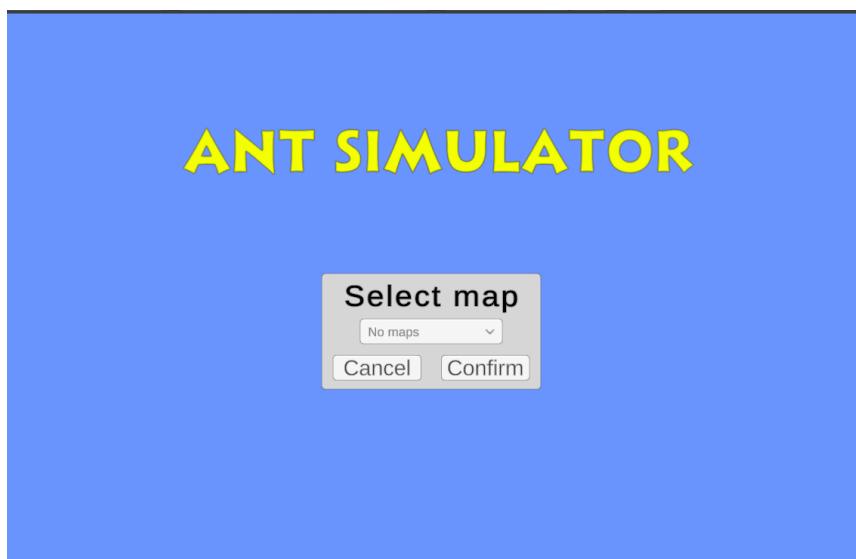


Figure 70: Submenú de nueva partida

En este menú, el jugador puede seleccionar uno de los mapas guardados para empezar una partida en él o editarlo. Al activar este menú, se observan todos los archivos en las carpetas *Maps* y *CustomMaps*, y se actualiza la lista de opciones del menú desplegable para contener todos los nombres de los mapas disponibles. Si se pulsa el botón *Confirm* habiendo seleccionado uno de los mapas, se cambia a la segunda escena después de modificar el valor *mapName* al nombre de mapa en *GameSettings* para cargar el mapa seleccionado. El botón *Cancel* esconde el submenú actual y vuelve a mostrar el menú principal.

Aunque solo puede ocurrir si el jugador manualmente elimina todos los archivos mapa del juego; si no se detectan archivos, la única opción que se muestra en el menú desplegable es “No maps”, y el botón *Confirm* no tiene ningún efecto.

Se puede llegar a este menú mediante el botón *New Game* del menú principal, en cuyo caso la variable *gameMode* de la clase *GameSettings* se cambia al valor 1 (modo de juego). También se puede llegar desde el submenú de creación de mapa, en cuyo caso la variable *gameMode* obtiene el valor 0 (modo edición de mapa).

13.1.3 Submenú de selección de archivo de guardado

En este submenú, el jugador puede elegir reanudar una de tres partidas guardadas previamente. En cuanto el submenú es activado, este mira las carpetas *Save1*, *Save2* y *Save3* dentro de la carpeta *SaveData* para comprobar si hay partidas guardadas. Si hay, deserializa los archivos *info.xml* y muestra los datos de las partidas en la UI del juego.



Figure 71: Submenú de selección de partida guardada sin partidas previas

Las tres partidas son representadas mediante botones. Cuando el jugador presiona uno de estos botones, si su partida guardada correspondiente existe, los valores de *GameSettings* son modificados para cargar dicha partida y se cambia a la segunda escena. Si no existe su partida guardada, no ocurre absolutamente nada. Si se presiona el botón *Exit*, el submenú se esconde y se muestra el menú principal.

13.1.4 Submenú de creación de mapa: tipo de mapa

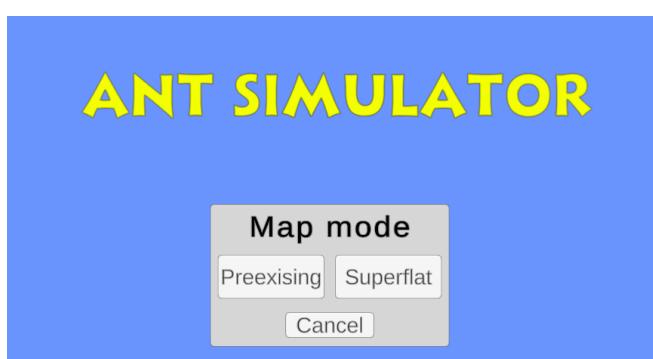


Figure 72: Submenú de selección de tipo de creación de mapa

Cuando el jugador decide diseñar un mapa en el editor de mapas, primero debe decidir si modificar un mapa preexistente o generar un mapa plano. El botón *Preexisting* muestra el menú de selección de mapa para que el jugador pueda seleccionar uno de los mapas preexistentes en los que basar el nuevo mapa. El botón *Superflat* muestra el menú de creación de

mapa plano. El botón *Cancel* muestra el menú principal.

13.1.5 Submenú de creación de mapa plano



Figure 73: Submenú de generación de mapa plano

En este submenú, el jugador puede introducir las dimensiones que quiere que tenga el mundo plano que será generado. Los controles deslizantes modifican los valores *x_chunks*, *y_chunks*, *z_chunks* y *height* de la clase *GameSettings* para ser usados por la función *generateFlatMap()* en la segunda escena. Los primeros tres valores deciden la cantidad de chunks a usar en cada dimensión. La variable *height* decide la altura del terreno, y el valor máximo del controlador deslizante es actualizado cuando se modifica el valor *y_chunks*.

Al pulsar *Confirm*, se modifican los demás valores de *GameSettings*, y se pasa a la segunda escena. En esta, *WorldGen* ve *GameSettings* y ejecuta la función *generateFlatMap()*. Este crea un mapa del tamaño de la cantidad de chunks por sus dimensiones y rellena sus valores. Todos los puntos de la matriz del mapa por debajo de la altura proporcionada son puestos a 255. Todos los puntos igual o por encima de la altura son puestos a 0.

El jugador puede poner la altura a 0, en cuyo caso el mapa generado estará vacío. Esto es útil cuando se quiere crear un mapa altamente personalizado, pero hay que tener en cuenta la necesidad de grandes masas de tierra para contener el nido.

13.1.6 Pantalla de carga

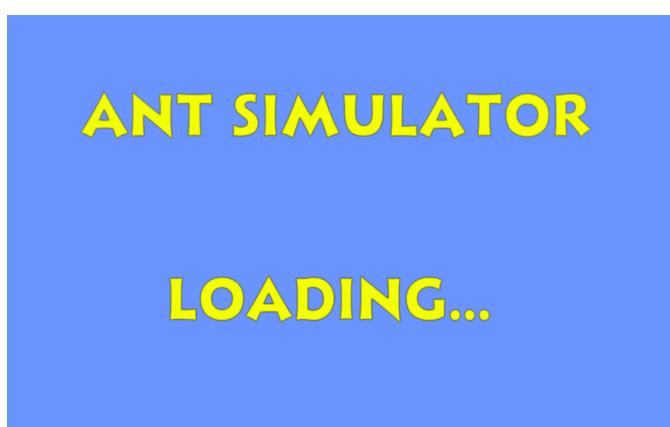


Figure 74: Pantalla de carga

Todas las funciones llamadas por los botones que cambian a la segunda escena primero desactivan el submenú actual y muestran la pantalla de carga. Este consiste simplemente en un texto con la misma fuente y color que el título, y sirve para informar al jugador que ha tomado una decisión y que el juego ha recibido su entrada. Antes de implementar esta funcionalidad, no se podía estar seguro de que pulsar el botón

surtió efecto, y podía dar la sensación de que el juego se quedara pillado.

13.2 Editor de mapas

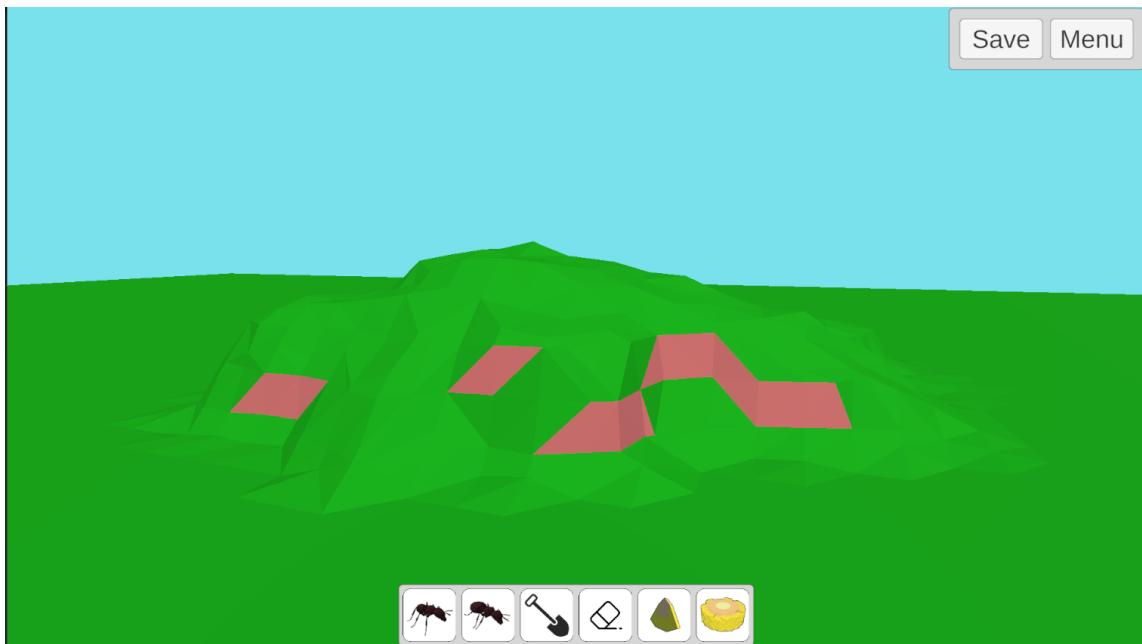


Figure 75: Modo edición de mapa

En el editor de mapa, al jugador se le proporcionan las herramientas necesarias para editar el terreno y crear objetos. En la parte inferior de la pantalla se encuentran 6 botones, cada uno correspondiendo a un modo de afectar el mapa (figura 75). En orden de izquierda a derecha, estos modos son:

1. Entrar en el modo colocación de hormiga. En ella, pulsar sobre el terreno crea una hormiga en dicha posición con la orientación correcta respecto a la superficie en la que se inicializa. Esto se hace usando un raycast con como máscara la capa del terreno.
2. Entrar en el modo colocación de hormiga reina. Funciona igual que la de la hormiga normal, pero coloca una hormiga reina. Solo coloca una reina si no hay ya una en el mapa, ya que cada partida puede tener tan solo una reina.

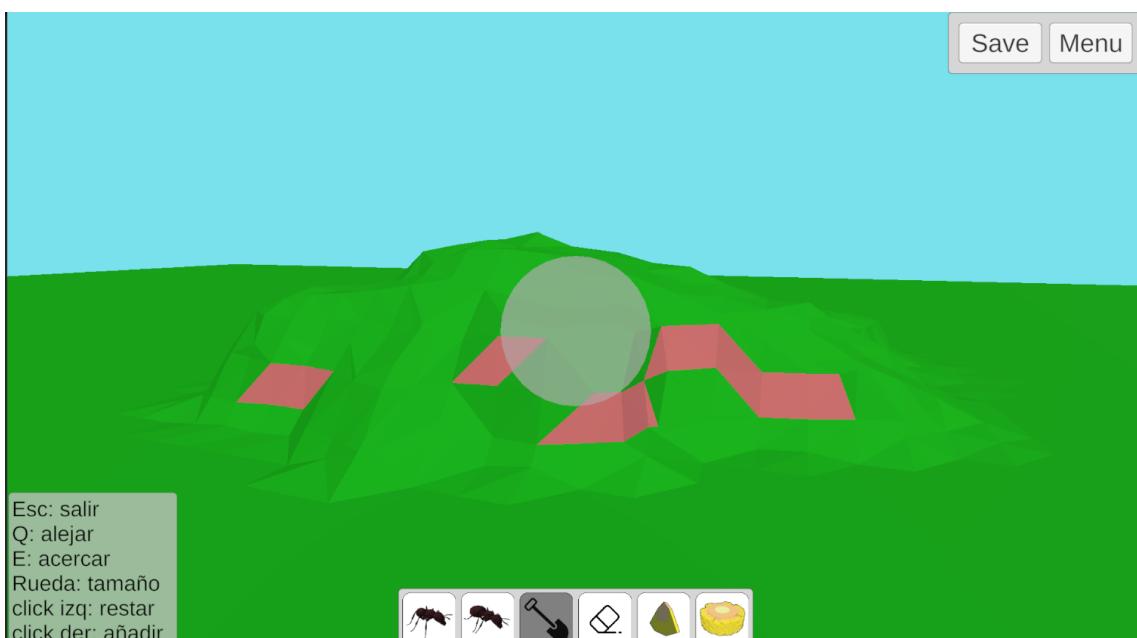


Figure 76: Modo edición de terreno

3. Entrar en el modo edición de terreno, donde el jugador puede modificar el campo escalar directamente. Esto lo hace mediante una esfera con tamaño variable (figura 76). Todos los puntos del campo dentro de la esfera tienen sus valores aumentados o disminuidos cuando el jugador presiona los botones del ratón, y los chunks que contienen dichos puntos son generados de nuevo para reflejar el cambio en el terreno.
4. Entrar en el modo borrado de objetos. En este modo, si el jugador presiona sobre un objeto en el mapa (hormiga, hormiga reina, pepitas de maíz o mazorcas de maíz), es eliminado.
5. Entrar en el modo creación de pepitas de maíz. Similar al modo hormiga, excepto que no se ajusta su orientación.



Figure 77: El control deslizante usado para decidir el número de pepitas en una mazorca

6. Entrar en el modo creación de mazorca. En este modo, se activa un control deslizante que se usa para decidir cuántas pepitas de maíz se encuentran en la mazorca. Cuando se inicializa la mazorca, esta obtiene dicha cantidad de pepitas repartidas en posiciones aleatorias de la mazorca. El mínimo número de pepitas es 1, y el máximo es 129.

El jugador puede salir de un modo pulsando de nuevo el botón o otro botón de modo. Para el modo de edición de terreno en el que se esconde el ratón, se implementó que también se puede salir de un modo pulsando la tecla escape.

Arriba a la derecha se encuentran los botones *Save* y *Menu*. Estos funcionan de la siguiente forma:

- Al pulsar el botón de guardar *Save*, aparece el panel *Save Map*. En ella el jugador puede insertar el nombre con el que quiere guardar el mapa. Al pulsar el botón *Confirm*, el juego comprueba que el mapa puede ser guardado. Si no puede ser guardado, se muestra un panel de aviso para informarle al jugador el motivo por el cual no se pudo guardar el mapa. Los motivos por los cuales un mapa no podría ser guardado son:

- El jugador no ha insertado un nombre.
- El nombre insertado por el jugador es igual al nombre de uno de los mapas preexistentes que no deben ser reemplazados.
- El mapa no contiene una hormiga reina, por lo que no se podría empezar a jugar.
- El mapa no contiene una o más mazorcas de maíz, por lo que no habría un objetivo de juego.

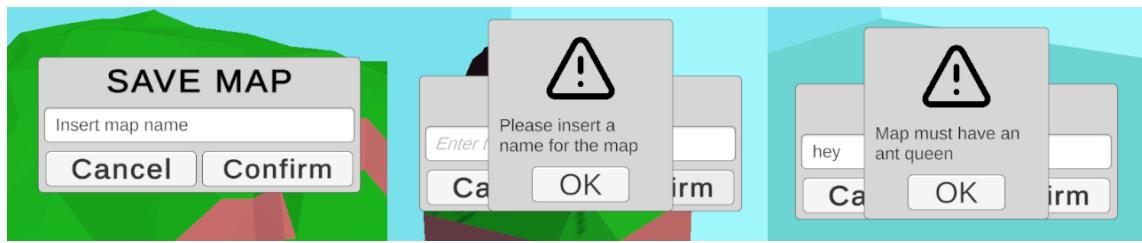


Figure 78: El panel de guardar mapa, y mensajes que pueden prevenir guardar

- Al pulsar el botón de menú, se muestra el panel de confirmación de volver al menú. Confirmar en este panel devolverá al jugador a la pantalla de menú principal de la primera escena. Cancelar cerrará el panel. Este panel es importante para que el jugador recuerde guardar el mapa, además de prevenir que pierda su progreso pulsando sin querer el botón.



Figure 79: Panel de confirmación de ir al menú

13.3 Modo de juego

Cuando el jugador decide empezar una partida nueva en un mapa o cargar una partida guardada anterior, la escena del mapa usa la interfaz gráfica de modo de juego. Esta interfaz debe comunicar al jugador la información relevante sobre el estado del nido y las fuentes de comida, para permitirle tomar decisiones informadas sobre la expansión del nido.



Figure 80: La interfaz gráfica del modo juego

La interfaz gráfica se puede dividir en cuatro componentes principales:

- El botón de gestión del nido
- El botón de visibilidad de feromonas

- El botón de control de hormigas
- La sección de información
- La sección de menú y guardado
- La sección de notificaciones

13.3.1 Botón de gestión del nido

Este botón se encuentra en la esquina izquierda del panel del modo juego y activa/desactiva el modo gestión del nido. Cuando se activa, se muestran visualmente las partes del nido, el botón pasa a ser coloreado para marcar el modo y aparece un panel interactivo en la esquina izquierda superior de la pantalla (figura 82). En este panel se encuentran cuatro líneas, cada una para un tipo de parte de nido. En cada una de las líneas, se encuentran 4 elementos, de izquierda a derecha:

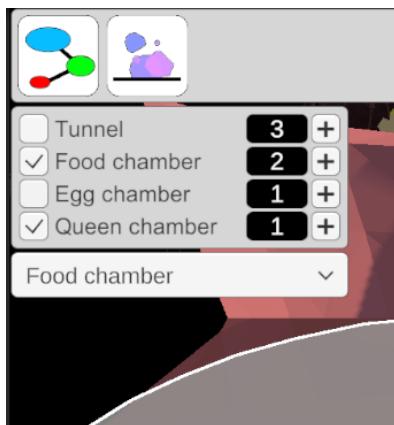


Figure 81: Panel de gestión del nido

1. Un toggle que activa/desactiva la visibilidad del tipo de parte de nido. De esta forma, el jugador puede decidir, por ejemplo, solo ver los túneles marcados o todo menos las cámaras de huevos.
2. El nombre del parte del nido del que se trata la línea.
3. La cantidad de partes del nido de ese tipo en el nido. Esto es útil para informar al jugador si faltan tipos de cámaras específicas.
4. Un botón que activa el modo “colocar parte del nido” para colocar una parte del tipo especificado. Este botón permite al jugador crear y modificar el nido añadiendo nuevas partes.

Adicionalmente, cuando el modo gestión de nido está activado, el jugador puede presionar sobre una parte de nido para seleccionarlo. Si selecciona una cámara, aparece un menú desplegable debajo del panel. Mediante este menú, que contiene los tipos de cámaras y se encuentra en el tipo de cámara de la parte seleccionada, el jugador puede cambiar su tipo. Cámaras seleccionadas cambian al color blanco, para que puedan ser diferenciadas de las cámaras no seleccionadas. Aparecen incluso si la visibilidad de su tipo de parte de nido ha sido desactivada.

13.3.2 Botón de visibilidad de las feromonas

Este botón es el segundo de la izquierda en el panel de modo juego. Le permite al jugador activar y desactivar la visibilidad de las partículas de las feromonas y *DigPoint*. Hace esto pausando/despausando la emisión de la clase Emitter y los objetos *ParticleSystem* que componen los puntos de excavación inicializados. El botón cambia entre un ícono que depicts feromonas coloreadas y uno que depicts el contorno de feromonas vacías para representar el estado activado/desactivado de la visibilidad de feromonas.

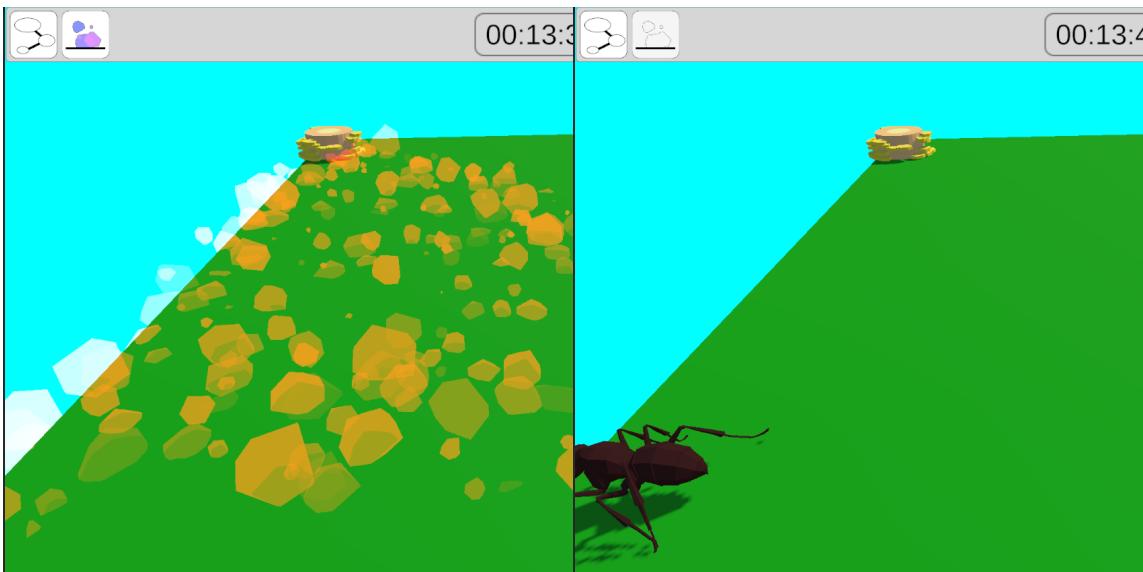


Figure 82: La activación y desactivación de las visibilidad de feromonas

13.3.3 Botón de control de hormigas

En el modo juego, el jugador puede seleccionar hormigas haciendo clic izquierdo sobre ellas. Cuando una hormiga es seleccionada de esta forma, obtiene un contorno blanco para diferenciarla de las demás hormigas y aparece el botón de control de hormigas en el lado izquierdo del panel de la interfaz gráfica.



Figure 83: El botón de control y una hormiga seleccionada por el jugador

Si se pulsa el botón, el juego entrará en el modo control de hormigas. En este modo, la cámara del jugador se posiciona detrás de la hormiga y procede a seguirla en tercera persona, pudiéndose rotar alrededor de la hormiga moviendo el ratón y alejarse y acercarse a la hormiga usando la rueda del ratón. Pase lo que pase, la cámara mirará hacia el ratón gracias a la función *lookAt()* de Unity, y se girará alrededor de este gracias a la función *RotateAround()*.

Mientras el jugador se encuentre en este modo, puede usar las teclas WASD para controlar el movimiento del ratón. Esto permite al jugador explorar el mapa, ayudando al nido a explorar los alrededores al poder dejar feromonas y encontrar mazorcas escondidas.

El botón de control y el ratón son escondidos durante el control de la hormiga, pero se puede salir de este presionando la tecla de escape.

Este modo fue añadido al juego principalmente para darle más capacidad de acción al jugador, ya que después de gestionar las necesidades del nido, el juego podía dejar al jugador sin acciones que tomar mientras se buscaban más mazorcas y la colonia crecía.

13.3.4 La sección de información

Esta es la sección del panel de modo de juego que le comunica al jugador las estadísti-

cas de su partida. Consta de 4 datos:

1. El tiempo de juego, en horas, minutos y segundos.
2. La cantidad de mazorcas encontradas por la colonia versus la cantidad total de mazorcas en el mapa. Este dato le informa al jugador su progreso en completar el mapa.
3. La cantidad de pepitas de maíz en cámaras de comida. Si el nido no contiene cámaras de comida, el texto muestra el símbolo “?” para demostrar que se desconoce la cantidad de comida recolectada por el nido, motivando su creación.
4. La cantidad de hormigas trabajadoras en el nido. Este valor muestra “?” cuando no se encuentra una cámara de huevos en el nido.



Figure 84: Sección de información

13.3.5 La sección de menú y guardado

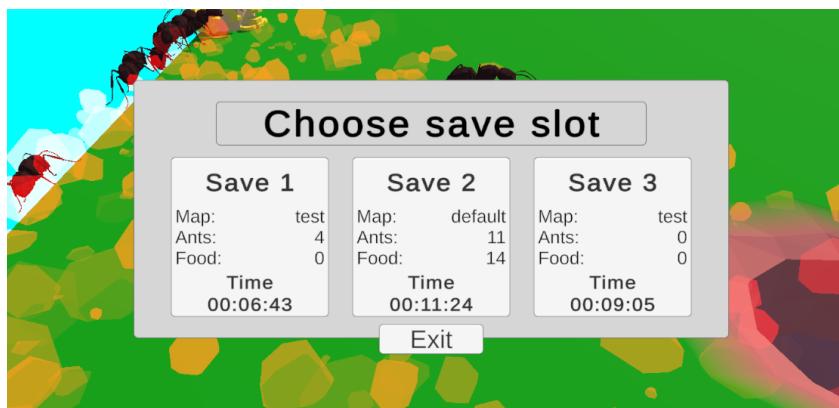


Figure 85: El panel de selección de ranura de guardado

En esta sección se encuentran los botones *Save* y *Menu*. Al pulsar el botón *Menu*, aparece el panel de confirmación de volver al menú principal, idéntico a como funciona en el modo edición de mapa. El botón *Save*, sin embargo, muestra el panel de las 3 ranuras de guardado (figura 85). Cuando el jugador presiona una de estas ranuras, la partida es guardada en esa ranura, creando los archivos *data.xml* e *info.xml* en la carpeta correspondiente y actualizándose la información de la partida guardada en el panel.

13.3.6 La sección de notificaciones

En la esquina inferior derecha de la pantalla pueden aparecer múltiples notificaciones para el jugador. Estas notificaciones informan al jugador de las acciones que debe tomar para el correcto funcionamiento del nido si falta algo, y sirven

como una pequeña guía para asegurarse de que el jugador pueda jugar correctamente al juego. Las posibles notificaciones son las siguientes:



Figure 86: 2 de las posibles notificaciones al jugador

- “*A queen chamber is required*” cuando el nido no contiene una cámara para la reina. Sin ella, la reina no podrá poner huevos y la colonia no puede crecer.
- “*More food chambers required!*” aparece cuando no hay suficientes cámaras de co-

mida en el nido para guardar la comida actual.

- “*More egg chambers are required*” aparece cuando no hay suficientes cámaras de huevos para sostener la población de las hormigas.

14 Balancear simulación con juego

El aspecto más importante de un videojuego es que sea divertido. Esto parece obvio, pero durante la creación de este videojuego basado en una simulación, fue fácil acabar centrándose más en el realismo de la simulación que en el aspecto lúdico del programa. Es importante que la simulación no tenga al jugador como un mero observador del desarrollo de la colonia, sino que debe permitir a este interactuar constantemente con la simulación y premiar su esfuerzo con el avance de la colonia.

Con el fin de mejorar la jugabilidad del juego de simulación, se buscaron posibles problemas y características que pudieran faltar mediante el análisis de la simulación y las pruebas de juego. A continuación, se idearon y aplicaron soluciones.

14.1 Requerir interacción y mantenimiento

14.1.1 La falta de necesidad de mantenimiento del nido

Uno de los aspectos principales de las simulaciones es que se pueden ejecutar y desarrollar sin la interacción de un agente exterior. En el caso de un videojuego, se quiere que el jugador interactúe con el juego, proporcionándole una variedad de posibles acciones, dándole impacto y consecuencias a sus decisiones.

En Ant Simulator, al jugador se le proporciona la responsabilidad y capacidad de asignar a la colonia donde escavar el nido. La naturaleza dinámica del terreno fue implementada precisamente para proporcionarle una infinidad de opciones en cuanto a la localización, forma y tamaño del nido. Esta gran variedad de opciones proporciona mucha libertad al jugador, pero si el juego no hace necesarias estas decisiones, no hay una motivación real para tomarlas.

Este problema se mostraba en las versiones iniciales del juego después de que el jugador hubiese colocado uno de cada tipo de cámara. La reina podía poner huevos en la cámara reina, toda la comida podía depositarse en el cuarto de comida y todos los huevos moverse a la misma cámara de huevos. Físicamente, las cámaras podían llenarse, pero nada detenía a las hormigas de aun así meter más objetos.

14.1.2 Solución: requerir la expansión constante del nido

La solución a este problema fue implementar la necesidad de expansión del nido. Esto se hizo creando dos variables que limitarían a las acciones de las hormigas según el estado del nido:

- *La capacidad de comida del nido*: Se definió una capacidad de comida máxima del nido según el número de cuartos de comida excavadas que este contuviera. Cada cámara de comida proporciona una extra capacidad de 30 piezas de comida al nido, y mientras la cantidad de comida en el nido no supera su capacidad total, las hormigas pueden seguir recolectando comida.

Al llenarse la capacidad de comida en el nido, las hormigas dejan de obtener tareas de recolectar comida y al jugador se le notifica mediante el sistema de notificaciones que faltan más cámaras.

- *La capacidad de huevos del nido:* Similar a la capacidad de comida, se definió una capacidad máxima de huevos en el nido según la cantidad de cámaras de huevo. Cada cámara de huevos sumaría 20 a la capacidad máxima, y cada cámara de reina sumaría 4. Esto último se hizo para prevenir que al principio del juego la reina tuviese que excavar por su cuenta, además de una cámara de reina, una cámara de huevos.

Mientras la capacidad de huevos se excediera, la reina no pone más huevos, dando igual su cantidad de energía.

Estos límites de capacidad se implementaron de forma general para el nido. Es decir, no prevendrían que las hormigas dejaran más de 30 pepitas de maíz en una cámara de comida o más de 20 huevos en una cámara de huevos. Esto se hizo para prevenir complicar más la inteligencia de las hormigas con acciones prohibidas. Para hacer que no se depositara más del límite permitido en las cámaras de forma más simple, las hormigas priorizarían dejar los objetos en las cámaras más vacías.

14.1.3 La falta de interacción directa con la simulación

Aunque el jugador podía dirigir las hormigas mediante las instrucciones de qué partes del nido excavar y dónde colocarlas, esto es una forma indirecta de controlar las hormigas. El jugador no tenía ninguna forma de interactuar de forma directa con las hormigas de la colonia y el terreno de fuera del nido. También se quiso buscar otra forma de interactuar con la colonia para ocupar el tiempo libre que tenía el jugador al terminar de satisfacer las exigencias de cámaras del nido antes de que surgieran más.

14.1.4 Solución: control directo de las hormigas

Se decidió implementar la capacidad de controlar directamente una hormiga. Esto permitiría al jugador escoger una de las hormigas y caminar con ella por el mapa. Esto le permitió al jugador mejorar los caminos de feromonas hacia mazorcas, además de conectar la colonia con nuevas fuentes de comida no alcanzadas aún, ya que la hormiga controlada soltaría un rastro de feromonas como las demás hormigas.

Esta opción nueva no solo añadió una nueva capa estratégica al juego, sino que de por sí resultó ser divertido poder tomar control de una hormiga y moverla por las distintas partes del nido y exterior.

14.2 El problema de la espera

Uno de los problemas más grandes en cuanto a la diversión del juego en sus versiones iniciales fue el comienzo. Normalmente, en un mapa se comienza el juego con tan solo la hormiga reina. Esto se hizo para reflejar la naturaleza real, en la que las reinas salen de su nido original, se aparean con hormigas príncipes y excavan el comienzo del nido por su cuenta antes de empezar a poner los primeros huevos.

En el videojuego, esto resultó ser altamente aburrido. Excavar una porción del nido con una sola hormiga puede tardar un rato, incluso creando solo un túnel y una cámara de reina pequeños. Adicionalmente, esperar a que las primeras cuatro hormigas que la reina puede poner al principio eclosionen creaba una cantidad dolorosa de tiempo muerto. Fue, por tanto, fundamental encontrar una solución a estos problemas.

14.2.1 Acelerar la velocidad de excavación de la reina

Para solucionar el problema de la cantidad de tiempo que hay que esperar para que la reina excavue las partes iniciales del nido, se decidió aumentar su velocidad de excava-

ción. En vez de acelerar la acción de excavar, que arruinaría su animación, se incrementó la cantidad de terreno excavado por la reina. Esto tenía sentido visual, ya que la hormiga reina es bastante más grande que las hormigas trabajadoras.

Las hormigas por defecto usan la función *Dig()* de un *DigPoint* para excavarlo. Para la hormiga reina se creó la función *BigDig()*. Este realizaría simplemente la función *Dig()* sobre el punto de excavación y los 4 *DigPoint* adyacentes si los hubiera. De esta forma, la velocidad de excavación de la hormiga reina se multiplicó por 5 (en la práctica, más bien por 3-4, considerando que no siempre hay *DigPoint* adyacentes).

14.2.2 Acelerar la velocidad de nacimiento de las primeras hormigas

La hormiga reina empieza siempre con 20 de energía, suficiente para dar a luz a 4 hormigas. Después de excavar la cámara de reina da a luz a estos 4 huevos, y el jugador debe esperar a que crezcan lo suficiente para salir de los huevos y ponerse a trabajar, poniendo en marcha la colonia.

Para que no tuviera que quedarse esperando unos minutos, se acortó enormemente el tiempo requerido para eclosionar de las 4 primeras hormigas del nido. De esta forma, esta bonificación de velocidad no sería usada en ningún otro momento de la colonia. La única desventaja de este método fue que ahora se podía ver en tiempo real el crecimiento de los primeros huevos, lo cual arruinó cierto realismo. La jugabilidad, sin embargo, es mucho más importante que el realismo cuando se trata de un videojuego.

14.3 La falta de un objetivo general

En juegos de simulación en los que se desarrolla una población, ciudad, colonia o cualquier sistema en general que puede crecer, gran parte de la diversión se obtiene de observar la expansión del sistema desde un comienzo humilde a una gran fuerza. En el caso de la Ant Simulator, ver la cantidad de hormigas crecer, la cantidad de trabajo y recolección aumentar debido a ello, y esto a su vez aumentar la velocidad de crecimiento del nido, se comprobó que era muy satisfactorio.

Aun así, cada juego debe tener un objetivo que realizar. Debe existir una meta que conseguir para motivar al jugador a probar el juego de nuevo y experimentar con estrategias distintas. Sin ello, tan solo el ciclo de crecimiento del juego puede resultar sin sentido.

14.3.1 Crear una meta para el juego

Se decidió crear una condición de victoria para las partidas del juego para motivar al jugador. Este terminó siendo la recolección de todas las pepitas de las mazorcas del mapa. En cuanto la última pepita fuera tomada de la última mazorca, el juego informaría al jugador de su victoria mediante un panel especial (figura 87). Este panel permite al jugador volver al menú o seguir jugando al juego.



Figure 87: Pantalla de victoria que se muestra al recolectar toda la comida de las mazorcas

14.3.2 Esconder información del jugador

La definición de un meta claro causó un problema específico: el juego necesitaba un grado de balance para que el jugador no pudiese conseguir el meta de forma demasiado fácil. En particular, gracias a la capacidad del jugador de controlar una hormiga y crear un camino de feromonas a cualquier mazorca del mapa, el aspecto de exploración del juego prácticamente fue anulado.

Para prevenir este método de conseguir una victoria demasiado acelerada, se decidió esconder las posiciones de las mazorcas del jugador hasta que las hormigas las encontraran. Se desactivarían los componentes visuales de las mazorcas y sus pepitas de maíz al comienzo del mapa, y se volverían a activar solo en cuanto una hormiga detectara la mazorca. De esta forma, el jugador no podría simplemente caminar hacia cada mazorca del mapa.

Este cambio no solo eliminó este exploit, sino que mejoró el aspecto de exploración del juego. El jugador ahora podría considerar las mazorcas ya encontradas y las áreas ya exploradas por hormigas mediante los caminos de feromonas recientes para deducir los puntos con mayor probabilidad de tener mazorcas no descubiertas. Mediante el control de la hormiga, podría caminar hacia dichos lugares en búsqueda de las mazorcas restantes.

Después de implementar la invisibilidad de las mazorcas, se descubrió rápidamente la importancia de informarle al jugador cuántas mazorcas no encontradas se encontraban en el mapa. Buscar mazorcas sin saber si te queda 1 o 10 puede ser muy frustrante. Fue el motivo por el cual en la interfaz gráfica se añadió el contador de mazorcas encontradas versus mazorcas totales del mapa.

15 Planificación del proyecto

15.1 Metodología usada

El proyecto consiste en la creación de una demo de videojuego para un público general. Por tanto, no existe un cliente determinado que exprese los requerimientos que debe tener el producto y con el que se pueda revisar el desarrollo de este. Aun así, se basó en la metodología ágil para el desarrollo del producto.

La metodología ágil es un enfoque para la gestión de proyectos que prioriza la flexibilidad, la colaboración y la entrega continua de valor, adaptándose a los cambios y las necesidades del cliente (en este caso, del mercado). Considerando la gran cantidad de imprevistos que inevitablemente aparecerían durante el desarrollo de un sistema complejo de simulación en 3D, la naturaleza flexible del método ágil fue atractiva para el proyecto.

Existen varias versiones de la metodología ágil, como Kanban, Scrum, Extreme Programming, Lean, etc. Todos muestran características distintas que los hacen mejores para ciertos proyectos que para otros, pero el que nos interesó fue la metodología ágil Scrum.

La metodología Scrum se basa en ciclos de trabajo denominados sprints en los que se completa alguna parte del proyecto. Estos pueden durar desde semanas hasta meses, y en cada uno se genera una versión del producto que supera a la anterior. El pilar fundamental de la metodología Scrum es que todas las decisiones se toman en función de la información existente y de la propia experiencia de los integrantes del grupo (en este caso, la experiencia del que realiza el TFG).

En la práctica, se dividió el TFG en varias secciones distintas necesarias para completar el juego de simulación de las hormigas. Se ordenaron para realizarlos en orden de importancia al funcionamiento del juego: en primer lugar, cosas como el terreno dinámico y las físicas de las hormigas, y la interfaz de usuario y modos de juego y mapas, al final. De esta forma, todos los pasos posteriores se realizarían con la información de cómo funcionan las partes más fundamentales de la simulación.

15.2 Preplanificación

Al principio del proyecto se creó una estimación de los sprints necesarios para acabar el videojuego simulador de la colonia de hormigas. Estos fueron, en orden de importancia y realización:

1. La creación del sistema de terreno dinámico.
2. La interacción de la hormiga con el terreno
3. La navegación de la hormiga por el terreno.
4. La creación y obtención de los recursos visuales 3D del juego.
5. La creación de animaciones. Este paso contiene también la creación de esqueletos para las animaciones de los modelos.
6. La implementación de asignar y excavar partes del nido.
7. La implementación de las animaciones como acciones del juego.
8. El ciclo de vida de las hormigas: desde su nacimiento hasta su forma adulta.

9. La inteligencia artificial de las hormigas.
10. Los controles del jugador para manejar la cámara y el nido de hormigas en la simulación.
11. La interfaz de usuario, incluida la creación y obtención de los iconos necesarios.
12. La creación de los mapas de juego, además de un modo de edición de mapa para el jugador.

15.3 Planificación y desarrollo

En el inicio de la realización de este proyecto, se creó una proyección estimada de la duración de los distintos sprints, teniendo en cuenta posibles dificultades. Esta planificación podría haberse considerado altamente optimista, considerando el rango de conceptos y habilidades que fueron necesarios aprender desde cero para realizar algunos de ellos, además de la multitud de dificultades y errores que tuvieron que ser resueltos.

La proyección inicial esperaba acabar el proyecto en un periodo de 3 meses. En la práctica, el proyecto se completó en 10 meses. Aunque hubo motivos extraacadémicos que contribuyeron a la extensión de la duración del desarrollo, el proyecto mismo mostró requerir una alta cantidad de revisiones, conocimientos nuevos y soluciones a problemas complejos.

Es difícil cuantificar una media de horas trabajadas cada semana, pero solían variar entre un mínimo de 10 horas y 20-30 cuando se podía permitir, y se pudo trabajar en la gran mayoría de semanas.

La creación del sistema de terreno dinámico.

Tiempo previsto: 3 semanas.

Tiempo final: 2 semanas. Media semana adicional para la división en chunks.

Este fue el primer sprint realizado. Gracias a la gran cantidad de documentación del algoritmo Marching Cubes, no hubo problema en implementarlo para formar el terreno dinámico.

La interacción de la hormiga con el terreno

Tiempo previsto: 2 semanas.

Tiempo final: 3 semanas.

El sprint inicial acabó en casi 1 semana, pero hubo que volver a dedicarle 2 semanas de trabajo dividido durante el resto del desarrollo, al encontrar problemas con su funcionamiento en casos particulares.

La navegación de la hormiga por el terreno.

Tiempo previsto: 4 semanas.

Tiempo final: 3-4 meses.

Sin duda, la sección del proyecto que más tiempo total requirió. Hubo varias iteraciones donde se intentó usar el concepto de feromonas para ayudar a guiar las hormigas por el terreno antes de encontrar una versión definitiva. Pathfinding en un terreno dinámico, en el que además los agentes pueden escalar las paredes y los techos, no es un problema muy tratado, así que gran parte del proceso fue inventar y probar soluciones nuevas.

La creación y obtención de los recursos visuales 3D del juego.

Tiempo previsto: 2 semanas.

Tiempo final: 3 semanas.

La obtención de los modelos tridimensionales fue rápida, pero hubo que crear desde cero algunos modelos y editar otros para formar nuevos. Se empezó el proyecto sin experiencia con modelaje 3D, pero se aprendió y aplicó relativamente rápido.

La creación de animaciones. Este paso contiene también la creación de esqueletos para las animaciones de los modelos.

Tiempo previsto: 2 semanas.

Tiempo final: 6 semanas.

Aprender a crear animaciones fue más difícil de lo previsto, y los procesos de creación de esqueletos de animación y su asignación a las partes de los modelos consumieron más tiempo de lo previsto.

La implementación de asignar y excavar partes del nido.

Tiempo previsto: 1 semana.

Tiempo final: 2-3 semanas.

La mayor dificultad de esta sección fue averiguar cómo dividir el proceso de modificar el terreno en pequeños pasos realizables por las hormigas y gestionar qué chunks modificar para reflejar los cambios. Una vez averiguado eso, la implementación fue rápida, pero se tardó más en llegar a ese punto de lo esperado.

La implementación de las animaciones como acciones del juego.

Tiempo previsto: 3 días.

Tiempo final: 3 semanas.

Hubo varios desconocidos necesarios para el funcionamiento correcto de las animaciones en juego. Entre ellos, el grafo de animaciones necesario y la dificultad de la implementación de acciones involucrando otros objetos o el terreno (recoger, llevar y depositar objetos, excavar el terreno).

El ciclo de vida de las hormigas: desde su nacimiento hasta su forma adulta.

Tiempo previsto: 3 días.

Tiempo final: 2 días.

Habiendo hecho las animaciones necesarias en los sprints de creación de animaciones, la implementación del ciclo de vida fue simple edición de scripts y resultó ser relativamente sencillo.

La inteligencia artificial de las hormigas.

Tiempo previsto: 2 semanas.

Tiempo final: 6 semanas.

El desarrollo de la inteligencia artificial de las hormigas y la reina hormiga tuvo dificultades hasta la implementación de un árbol de comportamiento. Este permitió que se adaptara fácilmente a algunos factores implementados posterior-

mente.

Los controles del jugador para manejar la cámara y el nido de hormigas en la simulación.

Tiempo previsto: 2 días.

Tiempo final: 1.5 semanas.

Se alargó principalmente para implementar la modificación de las partes del nido, los controles del modo de edición del mapa y el modo de control de hormiga en tercera persona.

La interfaz de usuario, incluida la creación y obtención de los iconos necesarios.

Tiempo previsto: 2 días.

Tiempo final: 2 semanas.

Se subestimó la cantidad de acciones que el jugador debería poder realizar mediante la interfaz, además de la obtención y creación de los iconos necesarios y la UI adicional para el sistema de guardado y cargado de mapas y partidas guardadas.

La creación de los mapas de juego, además de un modo de edición de mapa para el jugador.

Tiempo previsto: 2 días.

Tiempo final: 1 semana.

A pesar de ser considerado prioridad baja, el modo de edición de mapas fue de lo primero que se implementó en un sprint de 3 días, ya que fue necesario para poder probar las interacciones de las hormigas con el terreno y su pathfinding. Hubo un sprint adicional al final de la realización del TFG para dejarlo usable cómodamente por los jugadores.

Además del aumento de tiempo necesario para la realización de estos sprints, el trabajo fue ralentizado por componentes necesarios para el funcionamiento del juego que no se habían considerado inicialmente. Esta falta de conocimiento inicial se debió en gran parte a la falta de experiencia previa trabajando en videojuegos. Algunos de estos componentes necesarios fueron:

- La implementación del grafo de animaciones mediante el componente Animator para poder manejar las animaciones de las hormigas.
- La creación y edición de shaders especializados para la representación del terreno y el nido.
- La creación del sistema de guardado mediante la serialización y los cambios necesarios en los objetos y las relaciones entre estos para hacerlo funcionar.
- La estructura, funcionamiento y memoria del nido.
- El uso de sistemas de partículas para la representación visual de las feromonas y puntos de excavación.
- La limitación de acciones tomadas por la colonia cuando al nido le faltan cámaras de ciertos tipos para motivar la gestión del nido.
- El modo de control de hormiga y los controles especiales de la cámara durante dicho modo.

D. **Nombre Apellido1 Apellido2 (tutor1)**, Profesor del Área XXXX del Departamento YYYY de la Universidad de Granada.

D. **Nombre Apellido1 Apellido2 (tutor2)**, Profesor del Área XXXX del Departamento YYYY de la Universidad de Granada.

Informan:

Que el presente trabajo, titulado **Título del proyecto, Subtítulo del proyecto**, ha sido realizado bajo su supervisión por **Nombre Apellido1 Apellido2 (alumno)**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 201 .

Los directores:

**Nombre Apellido1 Apellido2 (tutor1)
Apellido2 (tutor2)**

**Nombre Apellido1 Apellido2 (tutor1)
Apellido2 (tutor2)**

Agradecimientos

Poner aquí agradecimientos...