
UNIVERSIDAD DE GRANADA

TRABAJO FIN DE GRADO
INGENIERÍA EN ...

# Titulo del Proyecto

**Subtitulo del Proyecto**

**Autor**
Yoram Joel Slot

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

Aqui 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

# Título del proyecto

**Subtítulo del proyecto.**

**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 T´ecnica Superior de IIngenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI XXXXXXXXX, 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 .

1. Introducci´on.
    Contexto/Antecedentes.
    Justificaci´on/Motivaci´on.
    Objetivos/Hip´otesis.
    Estructura de la memoria.
2. Estado del arte.
    Descripci´on de dominio del problema.
    Metodolog´ıas potenciales a aplicar.
    Tecnolog´ıas potenciales para usar.
    Trabajos relacionados.
3. Distintos cap´ıtulos sobre la propuesta, que cubrir´an:
    Descripci´on de la propuesta.
    Metodolog´ıa.
    Planificaci´on temporal.
    Presupuesto.
    Otros cap´ıtulos y secciones seg´un la metodolog´ıa y tipo de proyecto.
4. Conclusiones y trabajos futuros.
    Conclusiones.
    Trabajos Futuros.
5. Bibliograf´ıa.
6. Anexos

NOTES of things to add
- Used github with a special setting for unity projects

indice de palabras
- raycast
- Start()
- Update()
- FixedUpdate()
- isosuperficie

# 1. Introducción

## 1.1 Motivación

# 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

# Bibliografía

# Creación de la cámara y controles del jugador

# Creación del terreno dinámico: marching cubes

## Razonamiento

Una de las mecánicas que quiero que destaquen de la demo es la capacidad del jugador para poder elegir cualquier parte del mapa como la localización del nido. Esta libertad crearía mucho replay value, ya que habría una infinidad de opciones incluso en el mismo mapa. Sin embargo, cuanta más libertad le concedes a un jugador, más situaciones imprevistas, glitches y exploits pueden ocurrir. Es necesario entonces encontrar una forma muy simple y flexible de representar el mapa del mundo, que puede ser editado y interactuar con las otras partes del juego fácilmente.

Como comentado antes, quise emular la libertad de interacción con el terreno de Deep Rock Galactic, por lo que intenté descubrir qué sistema usaron para crear el terreno. Aunque nunca publicaron ninguna parte de su código (obviamente), en una entrevista con los desarrolladores del juego estos confirmaron que usaron una modificación del algoritmo marching cubes para generar su terreno, y para la destrucción de esta usaron CSG (Constructive Solid Geometry): a los meshes del terreno se le sustraen otros meshes para formar las excavaciones de los jugadores. Decidí entonces investigar el algoritmo de cubos de marcha.

Una de las otras opciones que investigué fue el sistema de terreno por defecto de Unity. El sistema de terreno de unity consiste en un mesh simple con valores de altura en cada coordenada del plano horizontal. Cada punto del mesh en dichas coordenadas entonces se encuentra en la altura definida. Esto significa que no puede formar túneles, ya que tienen un techo y un suelo. Además de que no es apto para la creación de un terreno altamente tres dimensional, no permite crear aperturas para cuevas en su superficie fácilmente. Por estas razones, rápidamente descarté la idea de usar los objetos terreno nativos de Unity y me centré en el algoritmo de cubos de marcha.

## Marching cubes, ¿Qué es?

Cubos de marcha 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, cada punto dentro del volumen teniendo un valor mayor o igual que la constante isolevel y cada punto fuera del volumen teniendo un valor de menos de isolevel, el algoritmo puede crear la superficie aproximada entre el volumen y las afueras colocando una malla de triángulos entre ellos. Este proceso es simplificado dividiendo el espacio en cubos que comparten caras, donde un lado de un cubo será cortado por la superficie si de los dos vertices 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 los lados.

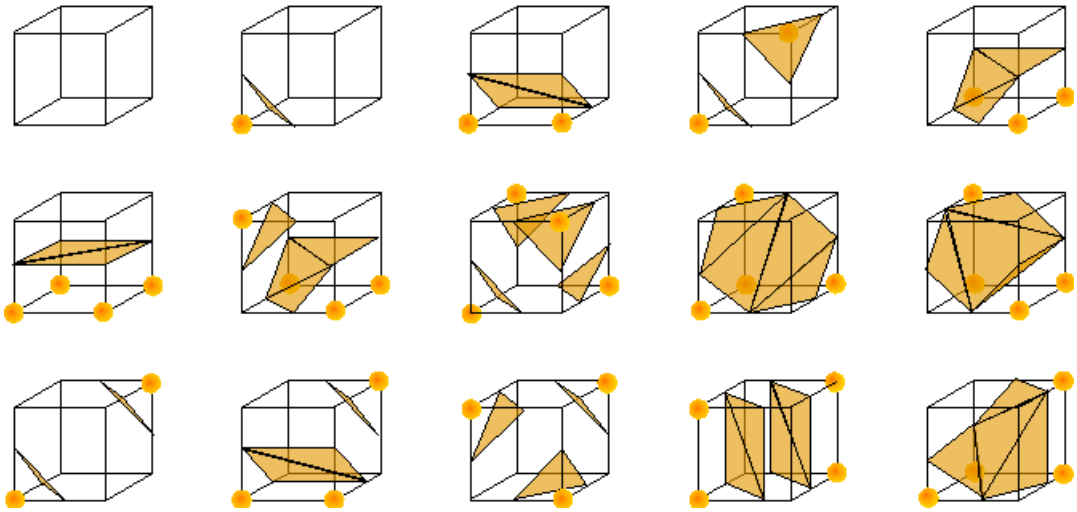

*Figure 1: Los distintas casos posibles en un cubo (sin tener en cuenta orientación)*

La dificultad a la que se enfrenta 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. Esto requeriría además bastante computación para generar los triángulos de cada cubo individual. Sin embargo, el algoritmo de cubos de marcha toma ventaja del numero limitado de combinaciones posibles en cada cubo, guardando todas estas posibilidades en lookup tables. Estos lookup tables guardan las 256 combinaciones posibles de triángulos y ahorran mucho tiempo de cómputo, acelerando la generación de las mallas.

# Funcionamiento del algoritmo de marching cubes

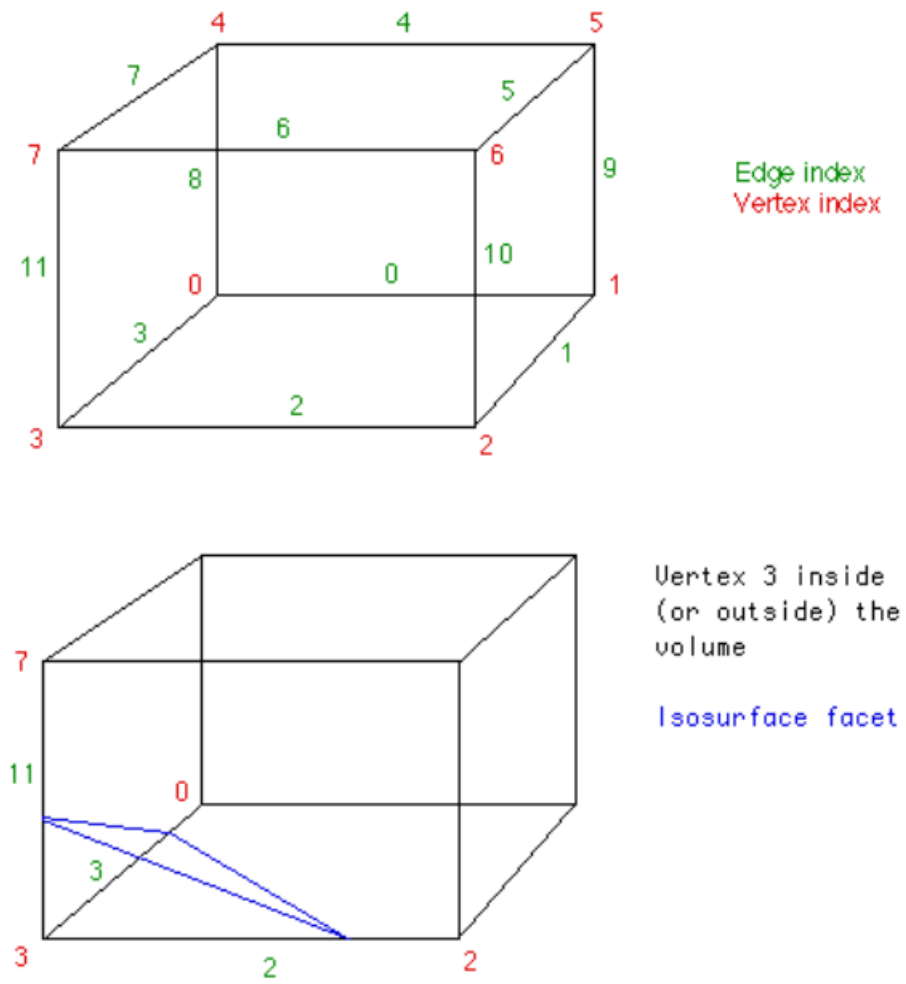


*Figure 2: Ejemplo de un cubo en el que la superficie corta los lados 2, 3 y 11*

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

- Para cada cubo, se obtiene 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, dado la figura 2, 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 representa los 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 imagen 1 [CHECK VALIDITY], 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).

- Se calculan los puntos de intersección en los ejes cortados. Esto se hace mediante interpolación linear 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 en la parte media del eje 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 vemos el la figura 3. 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 + (isolevel - V1)(P2 - P1)/(V2 – V1).

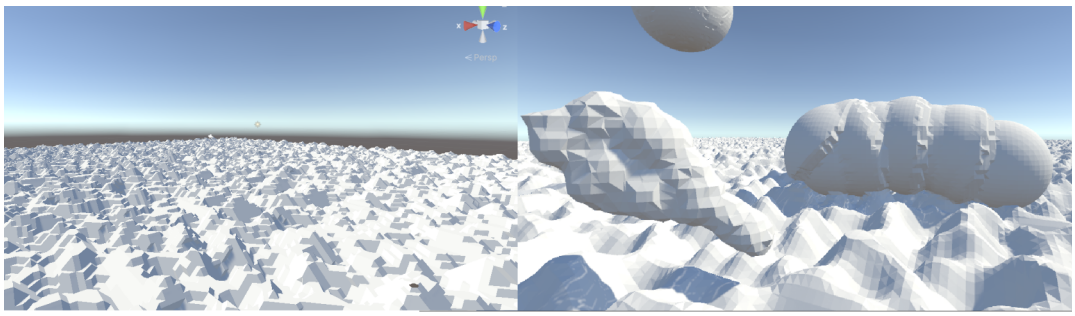


*Figure 3: La malla de marching cubes según el uso de interpolación lineal*

- Por último, se crean las facetas formadas por las posiciones por las que la isosuperficie corta a los cubos. 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 cuando acaba el array). Los primeros 3 números del array indican el primero triangulo, los segundo 3 el segundo triángulo, etc. El algoritmo sigue creando facetas de triangulo 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}.

```
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*

# Implementación del algoritmo de marching cubes

Para la creación del mesh de terreno hace falta guardar el campo escalar representado por este. Tanto el guardado de esta información como la generación del mesh se decidió hacer en una clase llamada WorldGen. Se añadió el script que contiene la clase al objeto cámara en unity, para que de esta forma siempre habría un objeto WorldGen en runtime y que siempre que se cargue la escena del juego se llamase la función Start() del script. Por tanto, en dicha función, se hacen todas las llamadas a funciones necesarias para la creación del mesh terreno.

En unity, para crear un mesh de forma dinámica, hacen falta dos componentes principales: un array de todos los vértices entre los que se forman los triángulos del mesh, y un array que representa los triángulos, 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 asigna los arrays, se calculan los normales y se crean un filtro y un colisionador a partir del nuevo objeto.

La obtención de los vectores y triángulos se haría en la función CreateMeshData, donde para cada “cubo” del campo escalar se obtendrían los triángulos que representan el terreno pasando por el cubo. 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 representar cada cubo del campo escalar mediante la coordenada de su esquina de menor valor posicional. Entonces, en CreateMeshData, se itera sobre todos los puntos del campo escalar menos los últimos en cada dimensión (de 0 a x-1, 0 a y-1 y 0 a z-1). Para cada punto, es llamada la función MarchCube(Vector3). Esta función realiza los siguientes pasos:

- En esta función se observan los distintos vértices del cubo y según si son menores o mayores que la isosuperficie se obtiene el índice que representa el tipo de cubo. Si el índice es 0 o 255, es decir el cubo está por completo debajo del terreno o encima del terreno y la superficie no lo corta, la función termina sin añadir vértices ni triángulos a las listas.
- Usando el índice del cubo, obtenemos el array de los índices de ejes cortados en la tabla TriangleTable.
- Se obtiene, para cada índice de eje cortado en dicho array, los índices de las dos esqui-

nas que lo forman en la tabla edgeIndexes.

- Para cada pareja de esquinas obtenida, se calcula mediante interpolación lineal el punto entre ellos por el que pasa la superficie. La interpolación usa los valores del terreno en cada de los dos puntos, obteniendo así el punto en el que se encontraría el valor isosuperficie. Este proceso es importante para po
- Ahora se tiene para cada tríangulo 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.

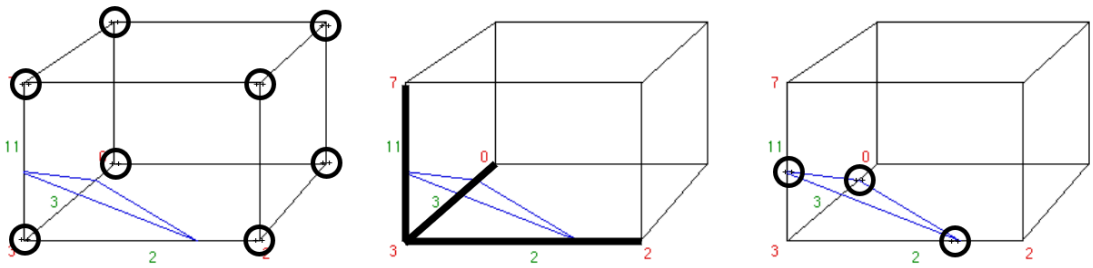


*Figure 4: 1. obtención de índice. 2. otención de ejes cortados. 3. obtiencion de vértices.*

El proceso completo por tanto era: WorldGen genera o carga el campo escalar tres dimensional que representa el mapa. Al iniciarse el juego, itera sobre todos los cubos y obtiene la lista de vértices y triángulos. Crea el mesh 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.

# Editación del mapa en real time

Una de las grandes ventajas de usar marching cubes es la capacidad de editar el mesh que genera simplemente cambiando los valores del campo escalar que se representa. Esto permitiría implementar un modo de edición de mapas sencillo para el jugador, donde de un mapa base se podría crear un escenario interesante para el desarrollo de una colonia de hormigas. De hecho, lo primero que se implementó en el juego fue este modo, en gran parte para poder testar el funcionamiento del algoritmo de los cubos de marcha.

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 entonces un objeto esfera que 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 suya.

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

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

- Dada 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.
- Para cada punto que se encuentra dentro de la distancia radio del centro, 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 click derecho, estos valores serán negativos, es decir, se eliminará terreno. Si se invocó usando el click izquierdo, los valores son positivos y se expandirá el terreno.

- 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 que después de añadirlas ningún valor se encuentra por debajo del mínimo ni por encima del máximo valor permitido, y llama la función de carga del mesh para que el terreno se actualice.

El jugador puede mantener pulsado los botones de click izquierda o derecha para editar el terreno, y cada update la función es llamado, aplicando gradualmente en cambio sobre el mapa.

# Chunks

Despues de la implementación de la edición del terreno en tiempo real, pruebas con cualquier mapa que no fuera pequeño mostraba problemas de rendimiento al tener que recargar la malla e iterar sobre todo el campo escalar cada FixedUpdate(). Este proceso podía tardar hasta 4 o 5 segundos según el tamaño del mapa. Renderizar una malla enorme también causaba bajadas de rendimiento, al no poder bajar la calidad o esconder las partes más lejanas de un mismo objeto.

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. También existiría la opción de implementar poder bajar la calidad o simplemente no mostrar chunks lejanos, en caso de querer mostrar mapas enormes.

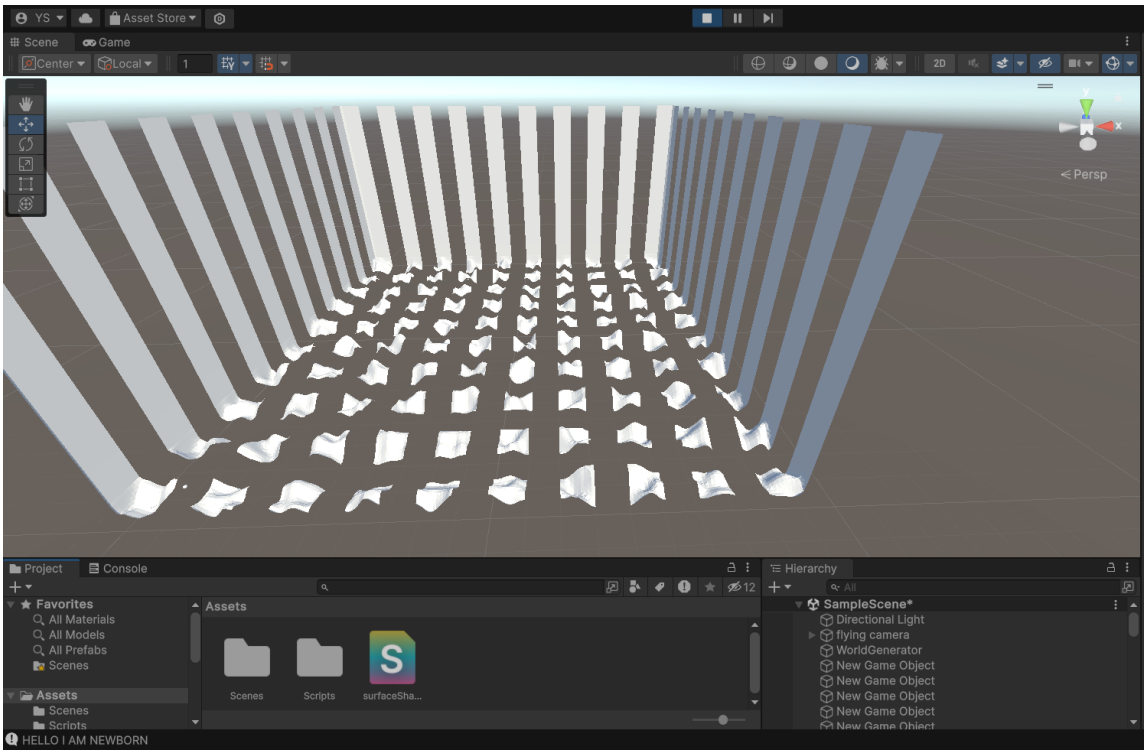


*Figure 5: Los chunks que componen un mapa separados*

El uso de chunks es una técnica de ahorro de memoria 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 será 10xYx10, donde y es la altura del mapa.

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

- Se creó una nueva clase llamado Chunk. WorldGen generaría la cantidad de chunks ne-

cesarios según el tamaño de mapa durante la ejecución el juego. Esta clase contenía los siguientes componentes:

Se movieron todas las tablas de tríangulos, ejes y vértices de marching cubes de WorldGen a Chunk, guardándose en memoria una copia de cada a pesar de instanciar múltiples objetos chunk poniéndolos en modo Static.

Las funciones de generación de malla y cubos de marcha previamente en WorldGen.

El tamaño y posición del chunk.

Los componentes necesarios para guardar y gestionar las colisiones y visibilidad de una malla.

- WorldGen obtendría un diccionario de chunks, ordenadas según un Vector3Int que designaría su posición para poder guardar y gestionar los objetos chunk generados durante la ejecución del juego. Además, WorldGen obtendría las variables de dimensiones x, y, z de los chunks.
- 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. Luego para cada punto del campo escalar obtenida 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, habían inconsistencias en las interconexiones de chunks (Figura 6)

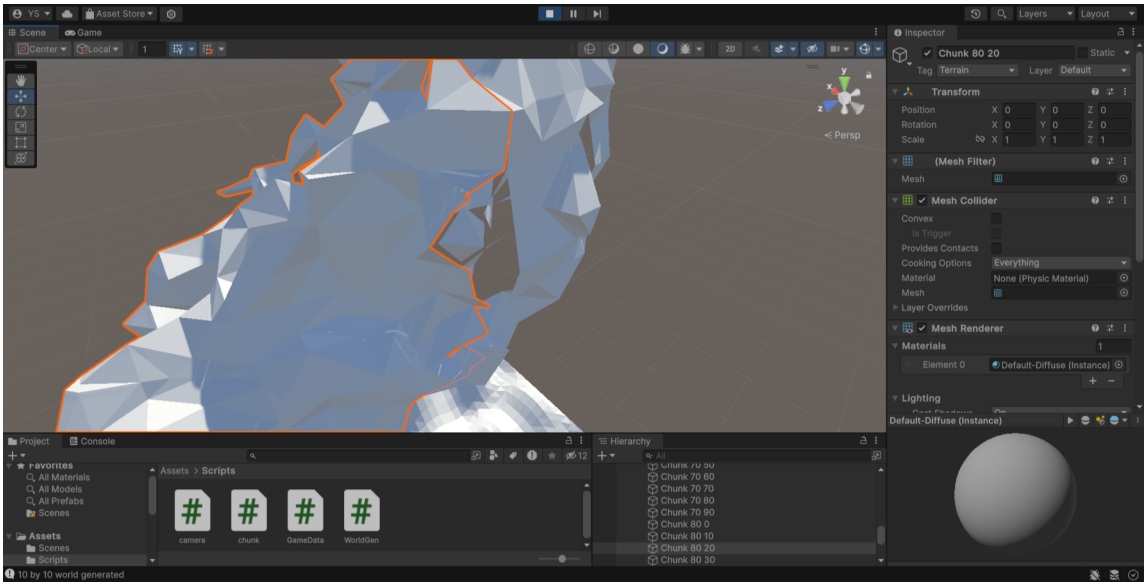


*Figure 6: inconsistencia en la conexión entre chunks*

La clase WorldGen ahora gestionaría la creación y guardado de los chunks, y mientras éstos ahora contenían toda la información necesaria para ejecutar el algorítmo de marching cubes y saber donde y cómo de grande eran, WorldGen aún conservaría el campo escalar en su enteridad. Este no se dividiría entre los objetos chunk, sino que estos podían acceder a sus secciones de mapa usando las funciónes de WorldGen de acceso al campo escalar.

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.

# Desarrollo gráfico del juego

## Concepto estilístico general: low-poly

- why choose low poly

- marching cubes works well with a generally low poly style due to its

- ngl i really like low poly

- makes creating models and workign with textures easier, which is important when you are making every single damn aspect of the game

- define low poly (before or after above?)

## Obtención y creación de modelos

Para el juego, hubo que obtener varios modelos tres dimensionales para representar los elementos más importantes: las hormigas y la comida.

La mayoría de los recursos tres dimensionales usados en este proyecto fueron obtenidos en línea, ya que la creación de modelos 3d es un arte que no vi asequible aprender encima de todos los demás aspectos del juego que debí desarrollar. Algunas de las excepciones a esta regla fueron la creación del modelo de la hormiga reina, editando el de la hormiga trabajadora, y la creación de la base de la mazorca de maíz.

### Hormiga trabajadora

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

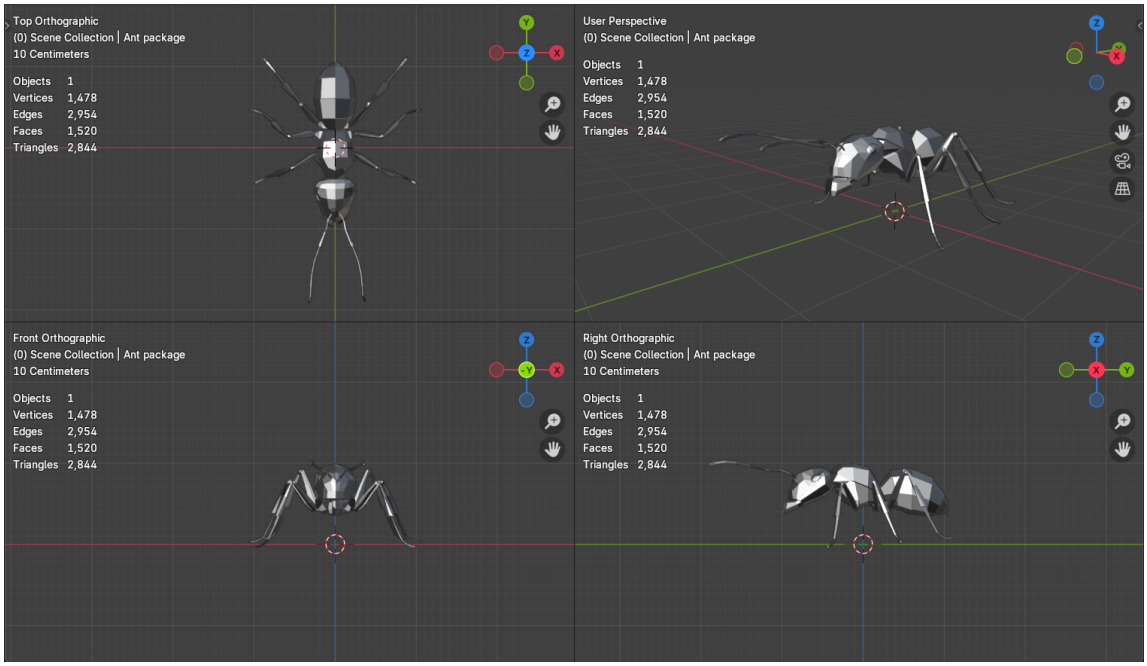


*Figure 7: Quadview del modelo de la hormiga trabajadora*

## Hormiga reina

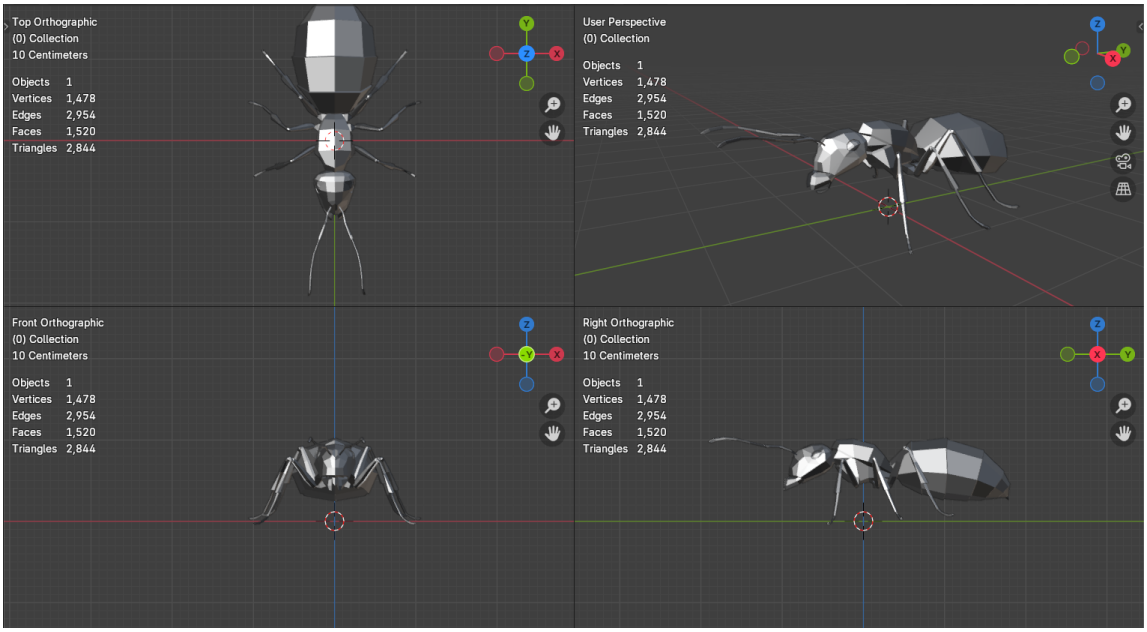


*Figure 8: Quadview del modelo de la hormiga reina*

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. El tórax más amplio, combinado con el abdomen resaltaron la corpulencia de la reina hormiga.

## Maíz

La pepita de maíz, al igual que la hormiga, fue obtenida en sketchfab. [INSERT LINK OR AUTHOR NAME?]. 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: como alguna encontraría el maíz, marcaría un camino hacia él, y el resto de las hormigas reconocerían y seguirían el camino. Podrían quitar pepitas de maíz de la mazorca uno a uno, hasta agotarla.

Hubo que poder representar la mazorca perdiendo pepitas de maíz uno a uno mientras las hormigas las agotarían. Habían dos opciones:

> Hacer que las pepitas fueran parte del modelo y crear animaciones para hacerlas invisibles. Las desventajas de esto era tener que crear una alta cantidad de animaciónes 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 posición y orientación estático relativo a la mazorca hasta que alguna hormiga lo recogiera. Se podría gestionar el orden en que pepitas desconectaran de la mazorca.

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 colocarlos en unity. Para tanto crear el modelo y obtener estas posiciones se siguieron los siguientes pasos:

- Se creó el objeto en blender a base de un cilindro simple. Se hizo más ancho que alto, representando una sección de una mazorca, no una entera.
- 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 de las pepitas.
- Se escribió un script de python usable en blender que iteraría sobre cada objeto seleccionado, notando su posición e orientación. Se usó teniendo seleccionado todas las pepitas, y se guardaron los datos obtenidos sobre cada pepita en un fichero aparte para

uso posterior.

- 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 unas esfera en las bisecciones para representar el núcleo de este.

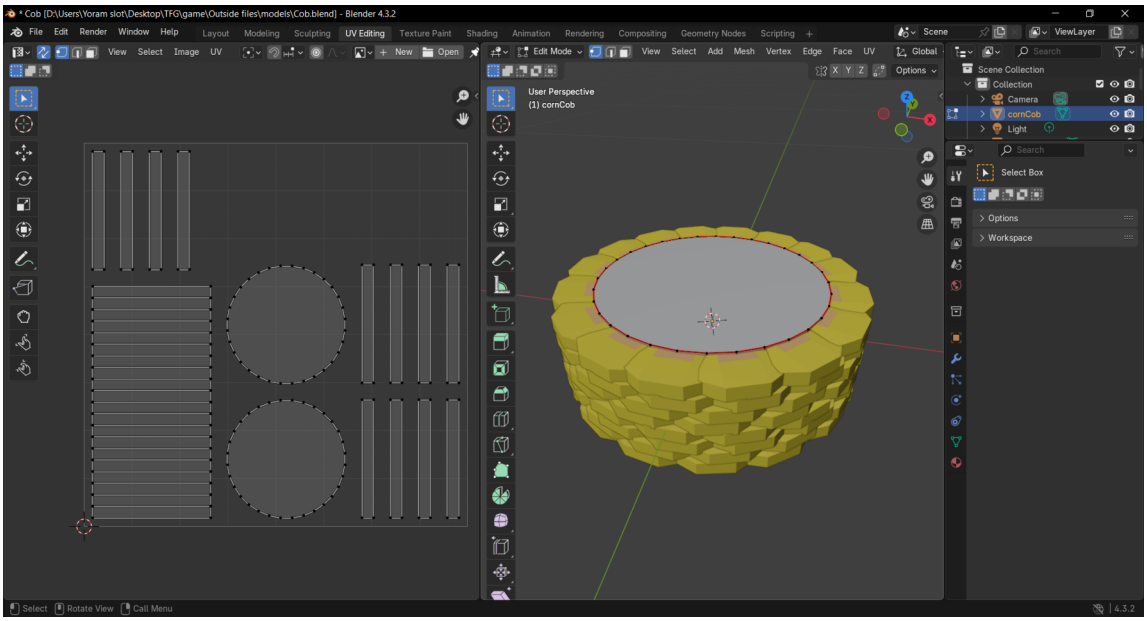

*Figure 9: La mazorca en blender con las pepitas incrustradas y a su izquiera su mapa de texturas*

IMPLEMENTACION (DIFFERENT AREA)

Luego era cuestión de al instanciar la mazorca, crear una pepita en cada uno de los sitios locales con la orientación. Hubieron algunas dificultades:

- La mazorca como padre causabe sus modificaciones sobre las pepitas, asi que hubo que ponerlo debajo de un emptyObject que recibió el script.

- El rigidbody cilindrico del emptyobject no se ajustaba a la de la mazorca, que por su cuenta tenia una transformación 1,1,0.5. Hubo que poner esa transformación en el emptyobject y reversarlo sobre todas las pepitas.

- Las magnitudes de las posiciones y orientaciones de las pepitas era errónea: hubo que buscar el valor que hiciera que se ajustara bien a la mazorca en dimensiones de unity.

- Aún hay algunas problemas de tamaño, las pepitas parecen ser demasiado grandes, lo que hay que gestionar. Añadi al CornCob la funcionalidad de guardar y cargarse, y le di un rigidbody y colision box de la mazorca. No contiene colisiones para todas las pepitas, pero parece inviable con el tiempo que tengo. Sacar las texturas creadas para la mazorca a unity en el FBX fue un misterio, por fin lo solucioné con settings del prefab en UNity mismo.

# Rigging

# Animaciones

- ant animations

- strips, combinations, exporting, etc...

- egg animation

# Color del fondo: debajo o encima del terreno?

How fondo works in unity and set to pastel blue

when in terrain can see tunnels but blue background doesnt make it feel underground

decide to make background blue overground, but black underground

Have functions to see if vector3Int point is below or above

add linear function to see if vector3 underground and use it to change background colors

# Aplicando color al terreno

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

[DESAROLLAR?]---------- Hubo un intento de usar una nueva URP, pero arruinó muchos aspectos previamente implementados, y no dio un resultado lo suficientemente bueno para justificar su uso.

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 que partes de la malla se encuentran debajo del suelo, y cuales se encuentran encima.

La clase mesh, aparte de los arrays de vértices y triángulos requeridos para su funcionamiento, contiene un array asignable opcional de objetos Color llamado colors. Tiene el mismo tamaño que el array de vértices, y representa los colores de estos. Cuando se representan 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 (formateado a un array) 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 usó como material un ParticleSurface. Este permite la visualización de una variedad de efectos visuales en la malla, la más importante de los cuales 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 encontraran debajo del terreno. Un primer intento fue mirar si se encontrara el vértice respectivo dentro de algún parte del nido. Aunque las funciones de la clase Nest para ver si un punto del mapa se encontrara dentro del túnel ya fueron implementados durante el desarrollo de esta parte, el uso de éstos fue inconsistente, ya que muchos de los vértices del terreno se encontrarían justo fuera de las partes de nido. [IS THIS ACCURATE???? CHECK AFTER MAKING TIMELINE]

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. Sino, 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 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 contuviera el mapa original.

- Añadir la función de codificar, guardar y descodificara para cargar el nuevo array a la clase GameData.
- En la función MarchCubes, añadir el check de debajo tierra: `WorldGen.SamplePastTerrain(v3) != WorldGen.SampleTerrain(v3)`. Si se cumple, el punto del mapa es resultado de ser excavado, por lo que se asigna el color marrón. En otro caso se 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 editador 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 demasiada 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 encontrarán 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 debajo del terreno, se mira la normal de su triángulo. Si este no se encontrara dentro de 80 grados del vector vector3.up, directamente se podría marrón y no se miraría si se encuentra debajo del terreno para ahorrar tener que mirar el campo escalar original.

## Visualización de las partes del nido

## Uso de partículas

### Feromonas

### Puntos de excavación

# Físicas e interacción de la hormiga con la malla

## Los dos estados de la hormiga

El sistema de físicas del juego requiere que las hormigas puedan caminar sobre cualquier parte de la malla generado por el algoritmo de marching cubes. Esto significa que deben poder caminar sobre superficies bocabajo, paredes y el mismo suelo. No se puede por tanto aplicar la fuerza de gravedad estándar sobre las hormigas cuando éstas se encuentran sobre alguna superficie, ya que causaría que se cayeran. Sin embargo, la posibilidad de la hormiga separarse de alguna pared o techo y caerse al suelo por la fuerza de gravedad también tiene que tenerse en cuenta.

Por tanto, para gestionar el movimiento de la hormiga en el entorno, se le asignan 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 encuentra en el estado sobre la superficie o cayendo, la hormiga debe ser capaz de sentir el mesh del terreno debajo suya. Para conseguir esto, el agente usa la funcion Raycast. Raycast, dado un punto de origen, una dirección, una longitud máxima y una máscara

proyecta una línea con dichas características y devuelve un objeto RayCastHit si colisiona con algún objeto en una de las capas especificadas en la máscara. Si no colisiona, devuelve false. El objeto RayCastHit contiene la información de posición de la colisión, la normal de la superficie con la que se colisiona, una referencia al collider con el que colisiona, etc.

Cada Update() el script Ant llama la función Raycast, proyectando un rayo corto desde su cen-

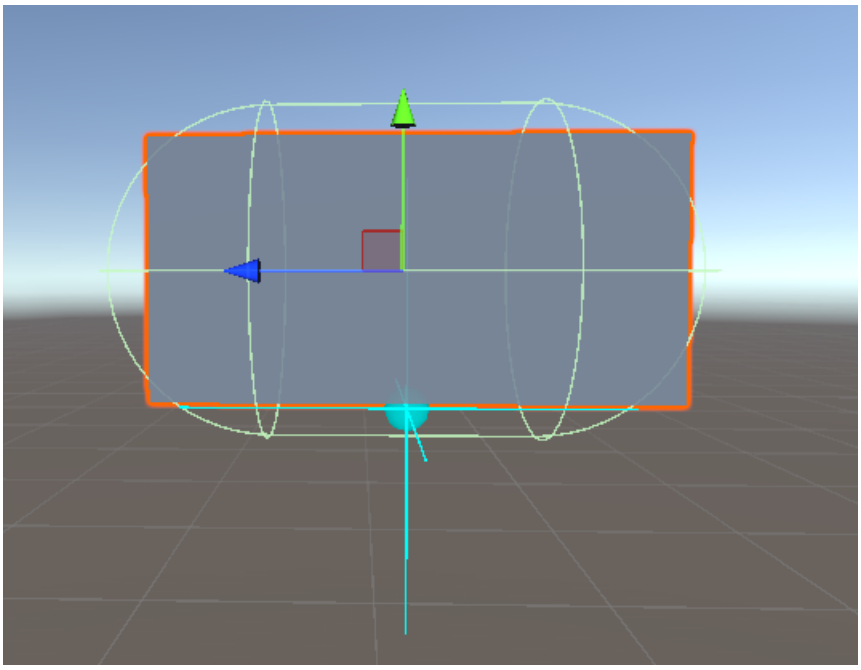

*Figure 10: Cuboide rectangular usado para pruebas tempranas de las físicas de la hormiga*

tro hacia abajo. Solo se quiere detectar si se encuentra sobre un mesh de la superficie, por lo que a estos se les asigna la capa “terreno” y Raycast usa la máscara que solo colisiona con dicha capa. Si la función Raycast devuelve true significa que ha detectado el terreno debajo de la hormiga, por lo que se activa el estado “sobre superficie” y la gravedad se desactiva. Si la función devuelve false, se considera que la hormiga está en el estado cayendo, por lo que se le activa la gravedad normal.

## Gestionar movimiento sobre malla y más Raycasts

Despues 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() incluido por defecto en Unity nos permite obtener este vector dado el vector 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 cayendo y dejando inamovible el agente (imagen 11 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.

Al tener más Raycasts, se necesitaba gestionar cuantas harían falta que sintieran el terreno pa-

ra 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 11 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 11c podemos ver cómo solo uno de los Raycast ve el terreno al intentar subir la cuesta.

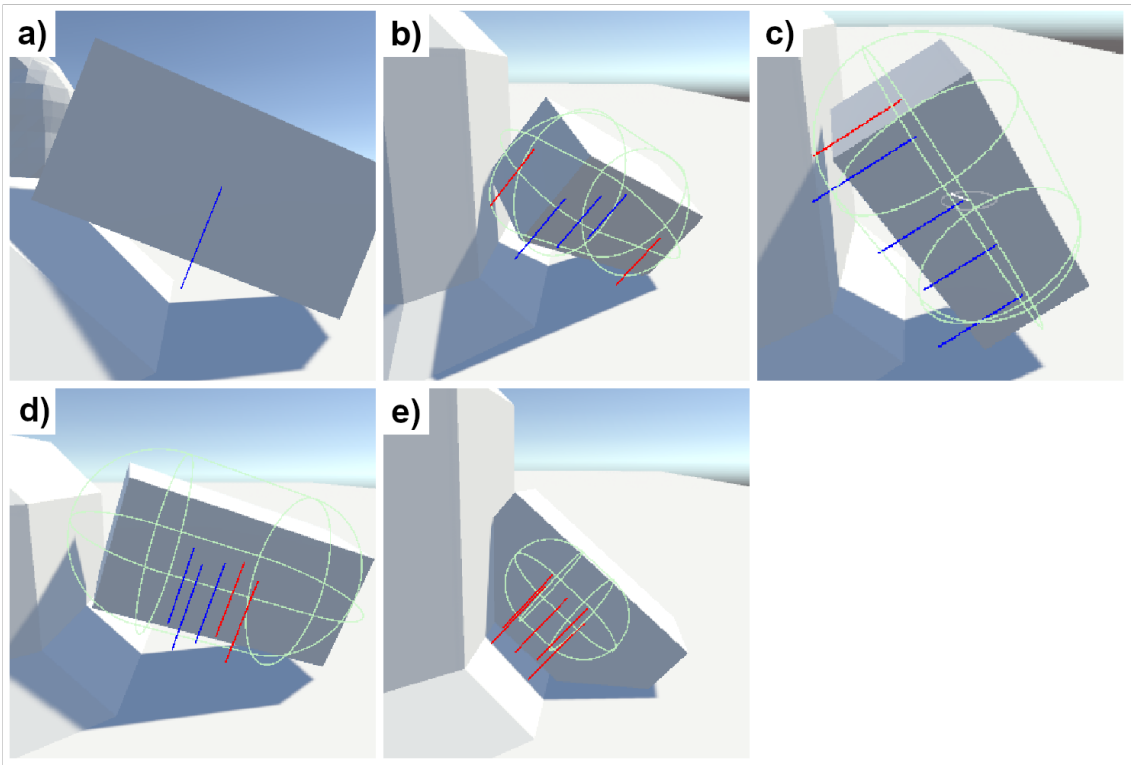


*Figure 11: Casos obtenidos al testear*

# La orientación de la hormiga sobre la malla

Otro obstáculo fue tener que encontrar una forma de hacer que la dirección en la que miraba 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 12.1), ya que la superficie misma empujaría la parte delantera de la hormiga haciéndolo seguir la dirección de la inclinación. Destacaba más en superficies convexas (figura 12.2), 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 encontrara. Esta normal se calcularía como la media de los normales obtenidos en los RayCastHits que proporcionaran 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 repentino causaba que los extremos del agente podrían chocar con el terreno y separar a la hormiga de ella.

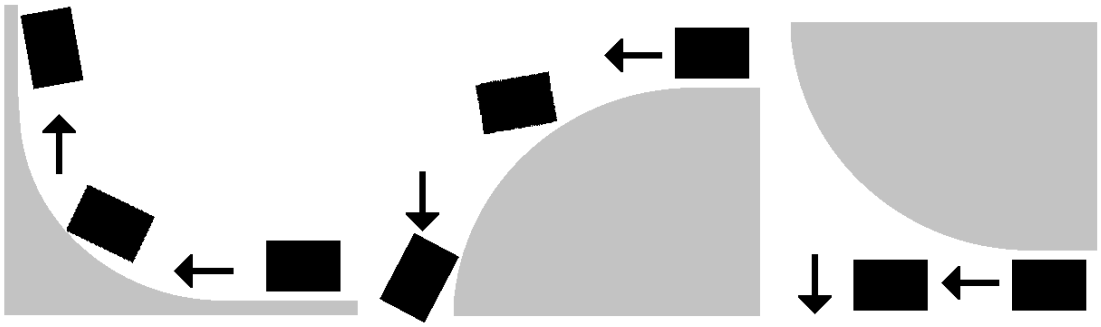

*Figure 12: trayecto del agente sobre superficies inclinadas sin un corrector de dirección.*

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, habían casos en los que los Raycasts podian 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 11 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.

AntTest rename and ant orientation correction
Ahora incluso cuando la hormiga se encuentra mirando el suelo, correcciona su angulo hacia arriba dada la distancia de los distintos puntos al suelo.

# Centro de gravedad de la hormiga

Se quiere que la hormiga siempre se mantenga boca arriba respecto a la superficie, y también se quiere considerar la posibilidad de que una hormiga caiga y aterriza en su espalda. En otras palabras, la posición por defecto de la hormiga, sea sometida a la fuerza de gravedad o algún método para mantenerlo sobre la superficie en la que se encuentra, debe 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 de 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 10 indica el nuevo centro de masa del cuerpo rígido: en vez del centro del objeto, ahora se encuentra en parte inferior. Dada su nueva posición, ahora si la "hormiga" se encuentra bocabajo y se le aplica una fuerza hacia una superficie, rotará hasta alinearse con dicha superficie por su cuenta.

# 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 12, 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:

- Ajustando la posición del agente manualmente. Se puede modificar la posición del transform del objeto hormiga directamente. Esto causaba que la hormiga clipearía con el terreno, aún teniendo en cuenta su distancia a ella, ya que podía tener partes más cercanas a la hormiga que los que los raycast vieran. Los cambios repentinos de distancia tambíen provocaron muchos movimientos rápidos y bruscos antinaturales.
- 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 bocarriba respecto a la gravedad. Sin embargo este método no fue el que se llegó a usar, ya que tenía dos desventajas grandes:

    La fuerza que la gravedad aplica sobre la hormiga es demasiada lenta. Si el agente se movía a suficiente velocidad en una superficie convexa llegaría a alejarse antes de que la gravedad pudiera alinearlo con el suelo.

    La fuerza de gravedad es universal. Cambiar la dirección para un agente afecta a todos los demás.
- 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 alínee con la superficie de forma automática. Resuelve los dos problemas de usar gravedad:

    La fuerza de gravedad es una aceleración y por tanto afecta periodicamente más y más al GameObject hasta que este llegue a su velocidád de caída máxima. Por 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. De esta forma es consistente, y cuando se encuentra la fuerza necesaria que aplicar cada FixedUpdate, funciona sobre cualquier superficie.

    AddForce se aplica individualmente sobre cada GameObject, mientras que la gravedad es universal.

# Sistema de movimiento de la hormiga

- Cuando la hormiga está sobre el suelo, puede moverse hacia delante y hacia la derecha/izquierda

- Implementación del sistema de seguimiento de gol

- proyectando dir sobre plano horizontal de la hormiga

- Hablar sobre versiones y cambios?

Problemas y soluciones - Al ver que las hormigas pudieron montarse entre si y empezerían a salir volando, incrementé un montón el drag y drag angular de estas para que no se pudieran empujar tan facilmente. Esto evitó que ocurriera tan a menudo, pero no lo arregló. Posibles soluciones a probar: -- Hacer que la hormiga solo puede estar grounded en la espalda de una hormiga, no en su barriga para evitar que se usen entre sí como suelo. - Poner muchas hormigas juntas crea un problema donde se escalan entre ellos sin llegar a donde quieren. -- Solución: limitar número de hormigas que puedan seguir un camino.

# Sistema de pathing de la hormiga

## 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 en 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.

## Navmesh

En unity se suele usar navmesh para esto. Navmesh es una estructura de datos abstracto que ayuda a IA con pathfinding, y unity contiene componentes que facilitan su uso. Funciona convirtiendo superficies del terreno en mallas especiales formados 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 que partes son navegables. Las mallas del juego son múltiples y dinámicos, dificultando su implementación.
- Navmesh asume que los agentes no pueden escalar paredes ni techos, que sus vectores up siempre apuntan hacia arriba. Han habido implementaciones de navmesh sobre objetos grandes esféricos como planetas, pero requirían dividir el terreno en múltiples mallas interconectadas con vectores de arriba distintas. 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.

## 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 seguirlas 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 resolvería el problema de pathfinding, sino también daría más realismo al comportamiento de las hormigas y la simulación de una inteligencia de colonia. Las tres versiones de pathfinding implementados se basaron en esto. [ACTUAL RESUMEN; THIS IS FALSE; I DID NOT REALLY CREATE PATHFINDING BEFORE ADYACENTCUBES]

## 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 llamado Pheromone unido a un objeto esfera sin colisiones para representar a los nodos.
- La hormigá miraría la coordinada 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 lo oliera.
- Al detectar una feromona, si la hormiga se movía 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.

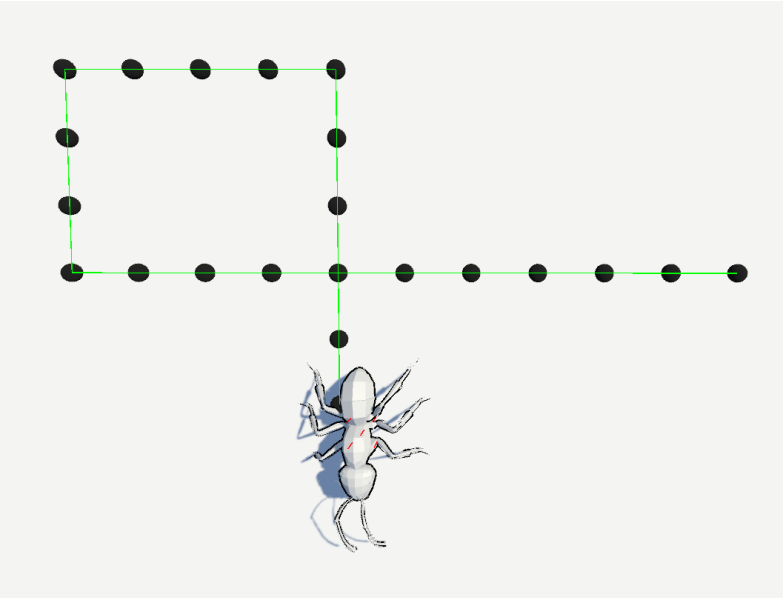

*Figure 13: Una hormiga habiendo marcado un camino de nodos de feromona*

## 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 haría provocaría 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.

## 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 incial 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.

## *Errores y soluciones*

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

- 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 también permitir que las hormigas tomaran pequeños atajos, se aumentó la distancia de detección de las feromonas. Ahora las hormigas, en vez de ver el siguiente feromona solo al llegar al previo, lo podrían ver de antes. Esto permitió que se giraran antes hacia feromonas laterales, y les hacia seguir el camino general en vez de las pequeñas secciones rectas que lo formaran.

- 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 suya, pero si se encontrara una pared o una plataforma entre ella y la siguiente feromona, podía cambiar de orientación al treparlo y quedar debajo o encima. Se quedaría pillado indefinidamente sin saber en qué dirección moverse (Figura 14).

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

- Las hormigas podían llegar a una feromona y no ver la siguiente al encontrarse demasiado encima o debajo de la hormiga (figura 15).

Esto se remedió 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 suyo si se encontrara horizontal.

- Con su caja de colisiones más vertical, la hormiga podía el siguiente nodo encima suya, 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 al que ya llegó o detrás suya, y se quedaría pillado ciclando eternamente entre nodos no aptos para seguir.

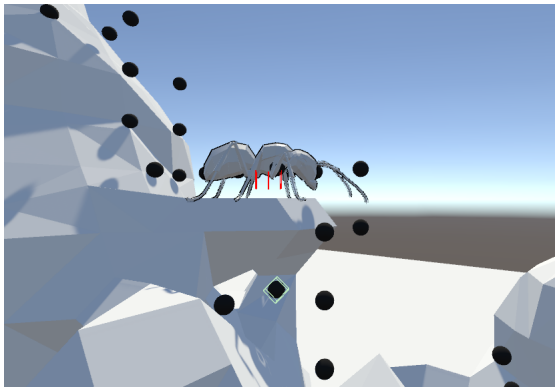

*Figure 14: La hormiga no sabiendo como moverse hacia la feromona marcada*

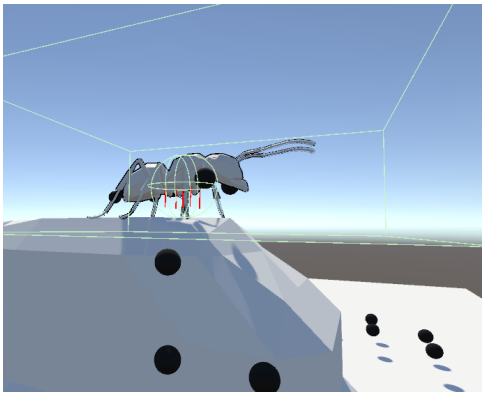

*Figure 15: La hormiga incapaz de ver la siguiente feromona del camino*

### *Veredicto final*

No hubo solución real al último problema encontrado en testeo. 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, dandole puntos a seguir pero no cómo.

## Segunda versión: feromonas sobre la superficie

La segunda versión del sistema de feromonas se creo con la idea de dar 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.

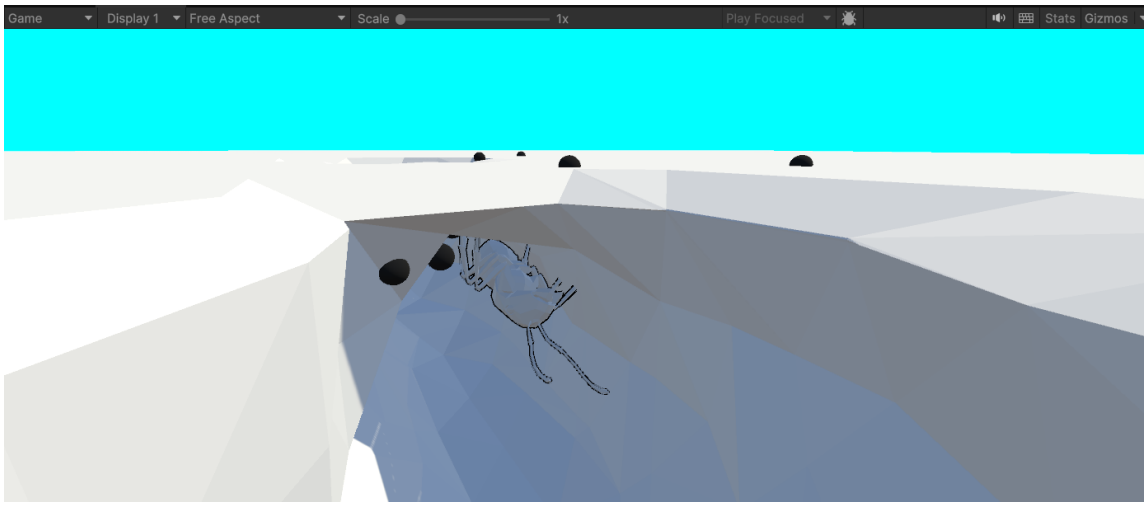


*Figure 16: Una hormiga siguiendo feromonas en el sistema de feromonas sobre superficie*

### *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 deje 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.

### *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.

### *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 a veces como para perderse alguna feromona por su camino.

### *Llegar a una feromona*

Inicialmente, se consideraba que una hormiga llegaba a una feromona al estar los suficientemente cerca a él. 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, que la hormiga siguiera acercándose lo suficiente a la feromona hasta tener un ángulo similar a la 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 a la 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 la 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 representativos del vector normal del terreno. No mostraba ser un problema al seguir caminos en la misma dirección en las que fueron colocadas, pero podía provocar que las hormigas se quedaran pilladas nunca alineándose con a 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 habían ciertos casos donde los caminos no se podían completar, sobretodo al salir de nidos.

### *Feromonas auxiliares*

### *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 suyos, o requiriendo un ángulo no reproducible al acercarse desde ciertos puntos o al haber sido empujados por otras hormigas.

Hubo un largo periodo de parches y testeos, pequeños cambios para intentar resolver casos nicho de hormigas quedándose pillados, 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.

## 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 las que las hormigas realizaron aleatoriamente en su exploración.

El sistema de adyacentCubes 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 adyacente. El sistema adyacentCubes 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.

## *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 hizo 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 cubos de marchas en la que se encuentran múltiples superficies dentro de un mismo cubo. Es por esto que el segundo elemento es necesario, para especificar a cual de las superficies se refiere.

Para crear un objeto cubeSurface se necesitan de 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 obtenida de sus raycasts.

## *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 cortadas por la superficie. Que alguno de los lados sean cortados 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 de 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 direc-

ción 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. Este vértices se puede usar para obtener el objeto cubeSurface del nuevo cubo.

## 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 formara 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, indexada por las posiciones de los objetos cubePheromone. Cada entrada contenía la lista de cubePheromone dentro del cubo respectivo, ya que podrían 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 feromonas 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 informado, y si un camino empezaba a carecer de comida, esta información también sería compartido con los demás. Esta idea no llegó a desarrollarse antes de la simplificación del sistema.

### *Creando caminos en CubePheromone*

Se creó la función PlacePheromone() en la clase CubePaths. Este, dado una posición y un objeto CubePheromone, metería dicha feromona 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 feromona del mismo camino en ello, sería reemplazado 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 feromona previa igual al original) , y lo colocarían en el mapa con PlacePhermomone().

StartPheromoneTrail(), que dada la la posición del cubo y la normal de la superficie sobre la

que colocar la feromona, crearía un nuevo objeto CubePheromone con un id de camino nuevo y lo colocaría tanto en cubePherDict con PlacePheromone y en pathDict como comienzo de un nuevo camino.

La hormiga misma, mientras soltara caminos, vigilaría que al cambiar de cubeSurface éste nuevo se encontraria adyacente a la feromona colocada previamente mediante la nueva función DoesSurfaceConnect(). Si se conecta, usaría la función de continuar el camino de feromonas. Sino, 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 feromona 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 feromona previa, se colocarían feromonas entre esa y la posición nueva de la hormiga sin empezar un camino nuevo.

### *La simplificación del sistema de feromonas*

- Como traía complicaciones el sistema complejo a la hora de diseñar las hormigas

- Cómo se simplificó: ahora posiciones y valores de tiempo

- Por qué funciona? Gracias a algoritmos de busqueda de caminos: podían expandir su búsqueda sobre superficies dentro del nido y superficies con feromonas.

### *Usar seguimiento de goles con cubeSurface*

- como se usa el sistema de followgoal en el nuevo sistema con la función GetMovementGoal()

- when on surface in the path that the ant is following, or with a pheromone of the pherPath the ant is following, move to the next in path

- When on surface next to part of path or pheromone, move to that surface.

- Since we don’t want to have the ant move in small straight lines, but be more natural, have goal be further,  allow greater turns.

- This can cause ant to exit path when moving towards next pos, then inmediately trying to get back onto path, then inmediately exiting path again, etc.

- Fix this by taking into account not only next pos to go, but pathstep after that too.

- New fix makes ant unable to scale certain walls. When at base and has to move forward to go up, but the step after that is above and behind the ant, the ant thinks it has to move backwards and away from the wall. When it does that and leaves the cube, it sees the that the next step is to move back into the cube, the step after going up, so it moves back into the cube. Loop ensues. Same thing can happen in a convex fashion.

- Fix was to recognize the two specific cases: get the vertices that form the face between the next cube and the cube after that. If two of those vertices (we will call them sharedCorners) coincide with vertices on the face between the current and next cube, then:

- if both sharedCorners are below the surface, and the other two aren’t, or both sharedCorners are above the surface, and the others arent, we have the case and should ignore the second direction.

# La implementación de algoritmos de búsqueda de caminos

- Created PathToDigPoint: Uses a* to find a path from the given surface to a point.

- tener en cuenta situaciones en la que hormiga intenta encontrar camino a nido sin estar conectado y quedarse pillado el buscador de caminos.

# El nido de las hormigas

## Composición y partes del nido

## Asignar zonas del nido

## Orientación del nido y funciones

El trabajo hecho para que la hormiga pueda moverse por el nido, aquí o en pathfinding? Las funciones aquí al menos? De saber si se está en el nido

# Sistema de excavación

- Hubo que encontrar una forma de dividir una excavación en tareas pequeñas que las hormigas podian compartir

- primer idea fue ir aplicando sphereDig sobre las partes del nido hasta excavarlo

    - implementación dificil

- eureka: excavar es cambiar valores del campo escalar → cada punto debe tener un nuevo valor para crear el túnel/cámara → crear objetos con dicho valor en cada punto, si la hormiga lo “excava” aplica el cambio de valor y deja de existir.

- implementaciones necesarias

  Obtener todos los puntos a cambiar y sus valores de forma medio eficiente

  Crear la clase DigPoint → varias iteraciones, desarrollar luego

  añadir funciones de excavación a la hormiga y tasks relevantes (Quizas guardar para IA hormiga?)

  añadido no instanciar objeto en puntos que permanecerán debajo del terreno pero cambia su valor, sino que ahora cuando se excavan los puntos adyacentes, estos se ponen en el valor que necesitan.

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

# Agradecimientos

Poner aquí agradecimientos...