eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Estudio de sistemas multiagentes en la simulación de un ecosistema

Autor/a

Aitor Domec Paz

2020

eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
INFORMATIKA FAKULTATEA
FACULTAD DE INFORMÁTICA

Grado en Ingeniería Informática

Computación

Trabajo de Fin de Grado

Estudio de sistemas multiagentes en la simulación de un ecosistema

Autor/a

Aitor Domec Paz

Directore/a(s)

Germán Rigau Claramunt

Resumen

Los sistemas multiagentes son un campo importante de la Inteligencia Artificial, puesto que nos permiten crear una serie de situaciones y sistemas con características únicas y especiales. Estos sistemas pueden resultar útiles en tareas como la el comercio online, la respuesta a desastres, el modelado de estructuras sociales e incluso la simulación de entornos, ecosistemas, comportamientos de mandas etc etc.
Sabiendo esto, este proyecto se centrará en el estudio de los sistemas multiagente a través de la simulación de un ecosistema. El objetivo final es conseguir una simulación de un ecosistema simple con animales que interactúen entre ellos simulando un entorno que se podría encontrar en la vida real. Los agentes de este entorno interactuarán entre ellos, se alimentarán, se reproducirán y tendrán un sistema de genética que transmitirán a través de la reproducción. También se hará un estudio estadístico de la población del sistema. Para crear el mundo sera necesario crear un generador de terreno procedural y para añadirle diversidad hemos incluido animales voladores.

Esta simulación puede ser analizada para, por ejemplo, entender mejor los caminos evolutivos de los animales en diferentes entornos o analizar los cambios poblacionales de los ecosistemas. El sistema será implementado en el lenguaje C# a través del motor de Unity.

Índice general

Resumen I

Índice general III

Índice de figuras VII

Índice de tablas XI

1. Introducción 1

1.1. Contenido del trabajo 5

2. Objetivos y Planificación inicial 7

2.1. Objetivos del proyecto 7

2.1.1. Generador de terreno 7

2.1.2. Boids 8

2.1.3. Ecosistema 8

2.2. Fases del proyecto 8

2.2.1. 1. Fase Preliminar 9

2.2.2. 2. Preparación / Introducción 9

2.2.3. 3. Desarrollo del Proyecto 9

2.2.4. 4. Conclusiones 9

2.3. Planificación 10
2.3.1. Planificación Inicial 10
2.3.2. Resultado Real 12
2.4. Herramientas utilizadas 14
2.4.1. Unity 14
2.4.2. Visual Studio Code 15
2.4.3. GitHub 15
2.4.4. Overleaf 16
2.4.5. GoogleDocs 16
2.4.6. meet.jitsi 16
2.5. Análisis de riesgos y viabilidad 16
2.5.1. Análisis de riesgos 16
2.5.2. Apartado especial COVID-19 19
2.5.3. Análisis de viabilidad 19

3. Estado del Arte **21**

3.1. Conceptos básicos 21
3.1.1. Mesh 21
3.1.2. Ruido Perlin 22
3.1.3. Boids 22
3.1.4. Sistema BDI 23
3.1.5. Pathfinding 23
3.2. Estado del arte 24

4. Desarrollo del proyecto **27**

4.1. Captura de requisitos 27

4.1.1. Generación procesal de terreno 27

4.1.2. Boids 28

4.1.3. Ecosistema 29

4.2. Análisis 29

4.3. Arquitectura del sistema 30

4.3.1. Elección de tecnología 30

4.3.2. Arquitectura 32

4.4. Implementación 38

4.4.1. Generación de terreno 38

4.4.2. Boids 51

4.4.3. Ecosistema 65

4.4.4. Cámara 90

4.5. Simulaciones 91

4.5.1. Conejos-Plantas 91

4.5.2. Zorros-Conejos-Plantas 93

5. Conclusiones **97**

5.1. Resultados 97

5.2. Trabajo futuro 99

5.2.1. Generación de terreno 100

5.2.2. Boids 100

5.2.3. Ecosistema 101

5.3. Valoración personal 102

Anexos

Bibliografía **107**

Índice de figuras

1.1. Diferentes propuestas de definiciones de la IA. 2
1.2. Ejemplo uso del programa de simulación multiagente MASSIVE. 3
1.3. Ejemplos de juegos de simulación. 4

2.1. Estimación de horas inicial por fases. 11
2.2. Resultado real de horas por fases. 13

3.1. Simulación Sugarscape de la biblioteca de modelos de NetLogo. 25
3.2. Comparación de Boids originales y nuestra implementación. 25
3.3. Captura del entorno JGOMAS. 26

4.1. Interfaz de Unity. 31
4.2. Comparación diagramas 'Comportamiento' original y nuestro. 36
4.3. Triángulos a partir de vértices. 39
4.4. Mesh de triángulos con altura nula. 40
4.5. Meshes de altura no nula. 41
4.6. Número de octavas con lagunaridad=2 y persistencia=0.5. 43
4.7. Generación de mapa de colores y ruido sin diferentes orígenes por octava. 43
4.8. Valores de persistencia con 6 octavas y lagunaridad=2. 44
4.9. Valores de lagunaridad con 6 octavas y persistencia=0.5. 44

4.10. Comparación de ruido y mapa de colores. 46
4.11. Ejemplos de uso del mapa falloff. 47
4.12. Comparación del mundo con y sin curva. 48
4.13. Curva utilizada. 48
4.14. Comparación del mundo con y sin casillas. 49
4.15. Comparación del mundo con y sin rellenar huecos. 50
4.16. Simulación original de una bandada de Boids. 51
4.17. Reglas básicas. Rojo: Centro manada. Verde: Dirección manada. Blanco: Separación. Negro: Nuestra dirección. 54
4.18. Puntos uniformemente generados en una esfera con espirales de Fibonacci. 57
4.19. Boid evadiendo obstáculo. 57
4.20. Cubo 3x3x3 desglosado. 59
4.21. Vistas mapa regiones 3x3x3 60
4.22. Persecución y huida de los agentes. 61
4.23. 1500 boids con uso de GPU. 65
4.24. Diagrama de búsqueda de casillas de agua. 67
4.25. Árboles en el mundo. 69
4.26. Heredado de genes. 70
4.27. Subdivisión de regiones en el mapa. 71
4.28. Regiones visibles para el conejo. 72
4.29. Línea recta en un sistema mapeado. 73
4.30. Recta dibujada en un sistema mapeado. 74
4.31. Ejemplo problema de las líneas rectas en el mapa. 75
4.32. Rectas generadas con el algoritmo de Bresenham. 76
4.33. Iteraciones algoritmo A*. 79
4.34. Ejemplos caminos con el algoritmo A*. 80

4.35. Camino generado con A*. 80
4.36. Jerarquía de las entidades. 81
4.37. Conejo consumiendo una planta. 82
4.38. Agentes teniendo crías. 85
4.39. Zorro persiguiendo a una presa. 86
4.40. Conejo huyendo de un depredador. 87
4.41. Menú de pausa. 88
4.42. Menús con información relevante. 88
4.43. Menú de inicio no finalizado. 89
4.44. Población plantas (verde) y conejos (blanco). 92
4.45. Comparación de velocidades y rangos de visión medios de conejos. . . . 93
4.46. Población plantas (verde), conejos (blanco) y zorros (rojo). 94
4.47. Poblaciones en un mundo 200x200. 95
4.48. Poblaciones con mayor valor nutricional en los conejos. 96

5.1. Capturas del ecosistema y los Boids. 98
5.2. Comparación resultados generador de terreno. 99
5.3. Ejemplo demo pathfinder y espirales de fibonacci. 99

Índice de tablas

2.1. Análisis de riesgos. 18

2.2. Análisis de riesgo especial COVID-19. 19

1. CAPÍTULO

Introducción

La inteligencia artificial es uno de los mayores campos de investigación en el ámbito de la informática incluso en la actualidad, también ha sido uno de los más relevantes por sus infinitas utilidades para casi cualquier campo de estudio. Hoy en día casi todo el mundo tiene una noción básica de lo que es la IA, ¿pero cual es la definición exacta, que es lo que compone la 'inteligencia' o la conciencia?

Curiosamente, aun no se ha llegado a un consenso para la definición exacta de este campo ya que se puede enfocar de varias formas diferentes. En el conocido libro de Russell y Norvig, *Inteligencia Artificial un enfoque moderno*[y Peter Norvig, 2008], se proponen cuatro enfoques diferentes para definirla, se pueden ver en la siguiente figura.

Sistemas que piensan como humanos	Sistemas que piensan racionalmente
«El nuevo y excitante esfuerzo de hacer que los computadores piensen… máquinas con mentes, en el más amplio sentido literal». (Haugeland, 1985) «[La automatización de] actividades que vinculamos con procesos de pensamiento humano, actividades como la toma de decisiones, resolución de problemas, aprendizaje…» (Bellman, 1978)	«El estudio de las facultades mentales mediante el uso de modelos computacionales». (Charniak y McDermott, 1985) «El estudio de los cálculos que hacen posible percibir, razonar y actuar». (Winston, 1992)
Sistemas que actúan como humanos	**Sistemas que actúan racionalmente**
«El arte de desarrollar máquinas con capacidad para realizar funciones que cuando son realizadas por personas requieren de inteligencia». (Kurzweil, 1990) «El estudio de cómo lograr que los computadores realicen tareas que, por el momento, los humanos hacen mejor». (Rich y Knight, 1991)	«La Inteligencia Computacional es el estudio del diseño de agentes inteligentes». (Poole *et al.*, 1998) «IA… está relacionada con conductas inteligentes en artefactos». (Nilsson, 1998)

Figura 1.1: Diferentes propuestas de definiciones de la IA.

Dentro de la **Inteligencia Artificial** existe el campo de los **sistemas multiagente** y es en este campo donde nos vamos a centrar principalmente en nuestro proyecto. Las utilidades de esta tecnología son mucho mas bastas de lo que la mayoría de las personas pueden llegar a pensar. Tiene utilidades en el mundo de los videojuegos, del cine o de la economía. También sirve para realizar toda clase de simulaciones como de sociedades, de pandemias, de animales, de ecosistemas, de insectos...

De hecho, el cine utiliza esta clase de herramientas desde hace mucho tiempo para crear escenas que de otra manera serían imposibles de rodar. Un gran ejemplo es el software MASSIVE[1] que son una serie de librerías para simular cantidades masivas de agentes. Este programa en concreto ha sido usado para crear famosas escenas como la batalla de la Última Alianza en *El Señor de los Anillos: la Comunidad del Anillo* o la batalla de los bastardos en la temporada 6 de *Juego de Tronos*. También ha sido usado en otras películas como Avatar, Eragon, 300, Harry Potter, Soy Leyenda etc etc.

[1] http://www.massivesoftware.com/index.html

(a) Batalla de los Bastardos, Juego de Tronos. **(b)** Batalla de la Última Alianza, SDLA.

Figura 1.2: Ejemplo uso del programa de simulación multiagente MASSIVE.

El mundo de los videojuegos también utiliza de manera frecuente los sistemas multiagente (MAS[2] a partir de ahora) para desarrollar los agentes que pueblen el juego como enemigos, aliados o NPCs[3]. Algunos juegos incluso centran toda su jugabilidad entorno a los sistemas MAS, de hecho, compone un género entero llamado "Videojuego de simulación"donde puedes simular desde personas, animales, países, ciudades... hasta hormigas si lo deseas.

[2]Multi Agent System
[3]'Non-Player-Character' o personaje controlado por el ordenador

(a) Rimworld. **(b)** Sim City.

(c) Banished. **(d)** Sims4.

Figura 1.3: Ejemplos de juegos de simulación.

Dentro de los sistemas multiagente nosotros queríamos realizar la **simulación de un ecosistema** donde las entidades pudieran interactuar entre ellas, comer, beber, huir, reproducirse, transmitirse genes etc. También queríamos explorar los sistemas de toma de decisiones existentes, más en concreto el sistema **BDI (Belief-Desire-Intention)** para los agentes. Para realizar la simulación necesitamos un terreno, este terreno podríamos crearlo nosotros a mano, pero esto llevaría tiempo y cada vez que quisiéramos cambiar de alguna manera el terreno tendríamos que volver a hacerlo. Esto nos lleva a un campo muy amplio y conocido que es el de la **generación procedural de terreno**. Nosotros nos pusimos como objetivo el crear nuestro propio generador de terreno con todo el trabajo y investigación que ello conlleva. Para finalizar, queríamos introducir un poco más de diversidad al ecosistema así que nos planteamos la tarea de introducir animales voladores en el mundo. Estos animales voladores serían una implementación de los conocidos **Boids**[Reynolds, 1987] de *Craig Reynolds*, unos agentes que simulan comportamientos de manada con una serie de reglas básicas.

Sabiendo esto, podemos dividir el trabajo en **tres apartados principales**: **Generación de terreno**, **Animales voladores - Boids** y **Ecosistema** cuyas capturas de requisitos explicaremos en el apartado *Desarrollo del proyecto: Captura de requisitos* , su arquitectura

y componentes principales en *Desarrollo del proyecto: Arquitectura del sistema* y su implementación más detallada en *Desarrollo del proyecto: Implementación*.

1.1. Contenido del trabajo

Nuestro trabajo está compuesto de otros **cuatro capítulos**:

- **Objetivos y Planificación inicial**: Hablaremos de la gestión del proyecto, las fases en las que lo dividimos, la planificación inicial que creamos, las herramientas que hemos utilizado y el análisis de riesgos y viabilidad del TFG. También describiremos los objetivos que inicialmente pensábamos completar en el trabajo.
- **Estado del Arte**: Introduciremos una serie de conceptos básicos importantes respecto al proyecto y hablaremos de trabajos relevantes que hemos encontrado sobre este mismo campo.
- **Desarrollo del proyecto**: El apartado principal donde explicaremos en detalle todo el trabajo realizado. Explicaremos la captura de requisitos y los analizaremos, describiremos la arquitectura general del sistema y explicaremos en detalle el funcionamiento de cada apartado relevante. También sacaremos conclusiones sobre el crecimiento de la población y de los genes más valorados a partir de las simulaciones que hemos realizado.
- **Conclusiones**: Sacaremos conclusiones acerca de los resultados obtenidos y analizaremos nuestro trabajo y valoraremos el resultado obtenido de este.

2. CAPÍTULO

Objetivos y Planificación inicial

En este capítulo explicaremos cuales eran los objetivos iniciales que planteamos al comienzo del proyecto, estimaremos su dificultad y problemas que acarrean y también hablaremos de la planificación que hemos creado para el trabajo.

2.1. Objetivos del proyecto

El objetivo inicial era la simulación de un ecosistema pero como hemos explicado brevemente antes (*Introducción*) también hemos tenido que crear un generador de terreno y una implementación de los Boids de Craig Reynolds. Cada una de estas tareas contiene sus propios objetivos y problemas.

2.1.1. Generador de terreno

No teníamos una serie de objetivos bien definidos para el generador de terreno, solo que queríamos que mantuviera una estética de *'casillas'* para mantener una estética simple y ahorrarnos problemas con la posibilidad de que los agentes atraviesen el objeto del mundo.

2.1.2. Boids

Para los animales voladores queríamos que cumplieran con las **reglas básicas** descritas por *Craig Reynolds*[Reynolds, 1987] que son **cohesión**, **separación** y **alineación**. También queríamos añadir una regla extra para que **evitarán las colisiones** con el entorno lo cual nos conduce a un conocido problema, *Como distribuir N puntos de manera uniforme en una esfera* que explicamos en detalle en el apartado *Implementación: Boids*. Adicionalmente también queríamos introducir unos **depredadores** para los boids y una forma de **optimizar el algoritmo** general de estos.

2.1.3. Ecosistema

Para el ecosistema teníamos una serie de objetivos bien definidos. Queríamos que los agentes pudieran **reproducirse**, **trasmitir genes a partir de la reproducción**, **huir de los depredadores**, **alimentarse y beber**, **tener edades** y **crías y embarazos**. También queríamos crear nosotros mismos nuestro propios modelos 3D de los animales aprovechando que en la asignatura de *Modelado 3D* aprendimos a usar *Blender*. Queríamos que hubiera **animales voladores** y también queríamos que hubiera unos **menús con la información relevante** del sistema para estudiarla. Así mismo, cuando los agentes tienen que encontrar el camino entre dos puntos queríamos implementar nosotros mismos como mínimo dos formas de pathfinding, siendo una el algoritmo de **Bresenham** (aunque no es un algoritmo de 'pathfinding' puede ser usado en nuestro mundo que estará formado por casillas) y otra el algoritmo **A* (*'A Star'*)**. También incluimos la tarea de estudiar y crear un sistema de toma de decisiones BDI para los agentes.

2.2. Fases del proyecto

El proyecto ha sido dividido en 4 fases principales, siendo estas *Fase Preliminar*, *Preparación / Introducción*, *Desarrollo del proyecto* y *Conclusiones*.
Estas cuatro fases dividen el proyecto en partes (con sus correspondientes tareas) que se deberían de completar en orden cronológico aunque no es necesario tener que completar una para poder pasar a la siguiente ya que pueden ser desarrolladas en paralelo.

Cabe destacar que la estimación de tiempo para cada fase se hará en el apartado de *Planificación*.

2.2.1. 1. Fase Preliminar

Esta fase trata sobre la preparación previa antes de comenzar a estudiar código, teoría etc etc. Aquí se creará el esqueleto del proyecto, se creará una primera estructura de las fases necesarias junto a un primer prototipo de la memoria con los diferentes apartados que vamos a querer explicar.
En resumen, en esta fase se plantearán (por decirlo de alguna manera) las estructuras a desarrollar del proyecto.

2.2.2. 2. Preparación / Introducción

En esta fase se adquirirán todos los conocimientos necesarios antes de comenzar el trabajo persé. Aquí se estudiará y explicará la teoría necesaria, se aprenderá a utilizar las herramientas y tecnologías escogidas y también se realizará el estudio del arte y del código base.
En resumen, es la preparación previa al desarrollo del proyecto.

2.2.3. 3. Desarrollo del Proyecto

Este es el apartado más importante del proyecto entero ya que es donde se completarán los objetivos del trabajo. Aquí se comentará el desarrollo de las tareas analizando los problemas encontrados y las soluciones implementadas. Esta será la parte del proyecto que más horas necesite y la que más carga de trabajo conllevará. Las tareas principales de este apartado son el desarrollo del código y la redacción de la memoria.

2.2.4. 4. Conclusiones

Sacaremos conclusiones a partir de los resultados obtenidos en las simulaciones y evaluaremos nuestro trabajo en el proyecto evaluando la cantidad de objetivos conseguidos, la calidad de estos, los problemas que han generado y las posibles mejoras a futuro.

2.3. Planificación

Todo proyecto que conlleve una considerable cantidad de horas necesita una fuerte planificación. En este apartado hablaremos de nuestra planificación para el proyecto. Las tareas originales, la estimación de tiempo de estas y cual ha sido el resultado final. Luego evaluaremos como de bien hemos seguido nuestra planificación original.

2.3.1. Planificación Inicial

Este trabajo tiene una estimación total de **300 horas** aproximadamente. Por tanto el número de horas que queríamos dedicar al proyecto era de 300 horas aproxidamadente. Disponiendo de 300 horas para repartir en cada fase, decidimos dividir cada fase en fases o tareas más pequeñas a las que asignamos un número de horas en función del máximo de horas de las que disponemos.

TAREAS					
Fase	Descripción	Horas estim.	Horas reales	Conjunto Horas	Conjunto Horas Reales
1. Fase preliminar	1.1 Preparación del calendario.	10		26	
	1.2 Asignación y proposición de tareas.	10			
	1.3 Preparación del diagrama Gantt	2			
	Analisis de riesgos y factibilidad	2			
	Analisis de herramientas a utilizar	2			
2. Preparación / Introducción	2.1 Introducción a la IA	10		60	
	2.2 Estudio de los diferentes modelos de organización de agentes	5			
	2.3 Estado del arte	10			
	2.4 Aprendizaje Unity	20			
	2.5 Estudio del código base	15			
3. Desarrollo del proyecto	3.1 Tareas objetivo a desarrollar	150		200	
	3.2 Redacción de la memoria.	50			
4. Conclusión	4.1 Resultados obtenidos	10		25	
	4.2 problemas / Objetivos opcionales o no logrados...	10			
	4.3 Posibles mejoras a futuro	5			
Reuniones		10		10	
		321		321	

Figura 2.1: Estimación de horas inicial por fases.

El número de horas estimadas por fase son:

- **1. Fase preliminar**: 26
- **2. Preparación / Introducción**: 60
- **3. Desarrollo del proyecto**: 200
- **4. Conclusión**: 25
- **Reuniones**: 10

Dando un total de **321 horas** estimadas para el trabajo. Este número es solo una estimación y el resultado real puede ser diferente. Toda la planificación se puede ver en el siguiente enlace[1].

El TFG se comenzó el **sábado 29 de febrero** con el objetivo original de presentarlo en la convocatoria de **junio**. Sin embargo, debido a las prácticas de trabajo que realicé entre **febrero** y **mayo** y al curso, esta opción fue descartada y la fecha de presentación que queríamos se cambió a la convocatoria de **septiembre**. De esta manera íbamos a disponer de un margen de tiempo mucho mayor para completar el TFG sin problema alguno. Este es el motivo por el que en los meses de **marzo**, **abril** y **mayo** el número de horas totales es bajo en comparación a los demás.

2.3.2. Resultado Real

Como puede verse en el calendario[2], estimábamos que tardaríamos unas **321 horas** en terminar el proyecto y asignamos el número de horas que creíamos necesarias para cada tarea. El resultado final ha sido ligeramente diferente, algunas de las estimaciones que hicimos originalmente no han resultado ser acertadas.

[1] https://docs.google.com/spreadsheets/d/1Yhi-wWllNt1QM72b2FXeZ8wgRlfOFkUiveZ7T3Fv6jc/edit?usp=sharing
[2] https://docs.google.com/spreadsheets/d/1Yhi-wWllNt1QM72b2FXeZ8wgRlfOFkUiveZ7T3Fv6jc/edit?usp=sharing

Fase		Descripción	Horas estim.	Horas reales	Conjunto Horas	Conjunto Horas Reales
1. Fase preliminar	1.1	Preparación del calendario.	10	5	26	13
	1.2	Asignación y proposición de tareas.	10	6		
	1.3	Preparación del diagrama Gantt	2	1		
		Analisis de riesgos y factibilidad	2	0,5		
		Analisis de herramientas a utilizar	2	0,5		
2. Preparación / Introducción	2.1	Introducción a la IA	10	5	60	47,5
	2.2	Estudio de los diferentes modelos de organización de agentes	5	3		
	2.3	Estado del arte	10	5,5		
	2.4	Aprendizaje Unity	20	14		
	2.5	Estudio del código base	15	20		
3. Desarrollo del proyecto	3.1	Tareas objetivo a desarrollar	150	153,5	200	224
	3.2	Redacción de la memoria.	50	70,5		
	4.1	Resultados obtenidos	10	6		
4. Conclusión	4.2	problemas / Objetivos opcionales o no logrados...	10	4	25	11
	4.3	Posibles mejoras a futuro	5	1		
Reuniones			10	7,5	10	7,5
			321	303	321	303

Figura 2.2: Resultado real de horas por fases.

Como vemos, el número final de horas que hemos dedicado son **303** sobre las **321** estimadas. Todas las tareas originales han sido cumplidas a excepción de una, el sistema de toma de decisiones BDI (*Belief.Desire-Intentions*). Del resultados real del calendario podemos destacar que la redacción de la memoria nos ha llevado más tiempo del estimado y que algunos de los pequeños apartados como *las posibles mejoras a futuro*, *los resultados obtenidos*, *el estado del arte* y *la introducción a la IA* han llevado menos tiempo del originalmente pensado. Por lo demás, el resultado real del calendario lo consideramos bastante cercano al original sin ningún desvió de gran importancia, el desvió más relevante ha sido el de la redacción de la memoria de unas **20 horas** que sobre las 300 del proyecto y teniendo en cuenta la gran importancia de este apartado consideramos pequeño y no importante. Las tareas que han acabado recibiendo menos horas de las estimadas, han recibido menos tiempo no por la falta de tiempo sino porque han acabado necesitando menos para ser completadas.

2.4. Herramientas utilizadas

Aquí me gustaría resaltar que al ser estudiante tuve acceso al paquete de estudiantes de GitHub[3] que proporciona una gran cantidad de programas a alumnos de forma gratuita, lo cual se agradece mucho ya que los estudiantes no siempre podemos permitirnos pagar las licencias de algunos programas.

2.4.1. Unity

Es un motor de videojuegos y es el entorno donde hemos elegido desarrollar el proyecto. El lenguaje de programación por defecto en Unity[4] es **C#**. Se ha elegido esto ya que la estructura y funcionamiento del motor hace que sea sencillo desarrollar sistemas multiagente y existe mucha documentación lo cual siempre es de gran ayuda.

[3] https://education.github.com/pack
[4] https://unity.com/es

2.4.2. Visual Studio Code

Para crear y editar el código se ha elegido Visual Studio Code[5] por comodidad y porque es compatible con Unity.

2.4.3. GitHub

Como sistema de almacenaje y seguimiento del proyecto se ha usado GitHub ya que es una herramienta sencilla, útil y que he usado con anterioridad. También permite crear proyectos para organizar sus tareas correspondientes con diferentes modelos de organización tales como *Bug Triage* o *Canvan*.

Cada vez que se ha trabajado en el código se ha subido al repositorio por medidas de seguridad para evitar la pérdida de información y para realizar un seguimiento del trabajo.

A lo largo del desarrollo del proyecto se han llegado a crear un total de cinco repositorios, todos de uso público:

- Ecosistema[6]
- Boids[7]
- Generación de terreno[8]
- Demo espirales de Fibonacci[9]
- Demo pathfinding [10]

[5] https://visualstudio.microsoft.com/es/
[6] https://github.com/DripyDev/PruebaEcosistem-boid
[7] https://github.com/DripyDev/Boids
[8] https://github.com/DripyDev/Generacion-de-terreno
[9] https://github.com/DripyDev/EspiralesFibonacci
[10] https://github.com/DripyDev/DemoPathfinding

2.4.4. Overleaf

Para la redacción de la memoria se utilizará la página web de Overleaf[11], un editor de LaTEX online. También se empleará como plantilla inicial la proporcionada por la UPV que se puede encontrar en el siguiente enlace[12]. Por supuesto, se han realizado modificaciones a la estructura del documento para que se amolde mejor a las necesidades del trabajo.

2.4.5. GoogleDocs

Para la creación y seguimiento de la planificación se ha usado GoogleDocs por su simpleza y utilidad. La planificación puede verse en el siguiente enlace: Planificación[13].

2.4.6. meet.jitsi

Debido a la especial situación causada por el **COVID-19** nos vimos obligados a realizar las reuniones entre el director del TFG y el alumno a través de la plataforma de videollamadas de Jitsi que es *meet.jitsi*[14].

2.5. Análisis de riesgos y viabilidad

No todos los trabajos son siempre viables o no siempre pueden ser terminados por las situaciones impredecibles que aguarda el futuro. Es por eso por lo que es importante analizar estos dos apartados, la viabilidad del proyecto para ver si realmente es viable y analizar los riesgos que pueden afectar a esta viabilidad y a ser posible preparar contramedidas. Este capitulo será usado para hablar de estos dos temas.

2.5.1. Análisis de riesgos

Todo proyecto puede llegar a sufrir contratiempos, algunos predecibles y contrarestables, otros aleatorios e impredecibles. Por ello es importante analizar estos posibles problemas

[11] https://es.overleaf.com/
[12] https://www.ehu.eus/es/web/informatika-fakultatea/ikasketak/gradu-amaierako-proiektua
[13] https://docs.google.com/spreadsheets/d/1Yhi-wWllNt1QM72b2FXeZ8wgRlfOFkUiveZ7T3Fv6jc/edit?usp=sharing
[14] https://meet.jit.si/

para tener contra-medidas que atenúen o nieguen los efectos negativos que puedan llegar a causar.
Estos son los principales problemas identificados que pueden afectar negativamente al trabajo.

- **Estimación de tiempo errónea:** Como siempre, las estimaciones de tiempo pueden estar muy equivocadas ya que no siempre es fácil saber cuanto tardarás en realizar algo, siempre pueden y suelen surgir contratiempos o complicaciones en las tareas. Al no haber una medida definitiva para este problema, la estimación para las tareas podrá ser cambiado de forma dinámica en función de las necesidades que surjan, en el caso de que surjan cambios, estos serán documentados.

- **Complicación de tareas:** Es posible que los objetivos del trabajo tengan una complicación significativamente mayor a la originalmente pensada, esto puede causar que ciertas tareas conlleven una carga de trabajo mucho mayor a la estimada, lo cual podría descolocar el calendario a seguir. Para evitar este problema, se analizarán todos los objetivos con el tutor del TFG para estimar la dificultad de los mismos. En caso de que aun así no se consiga evitar el contratiempo, se contempla la posibilidad de abandonar o dejar sin completar algún objetivo concreto y, en caso de que suceda, se justificará dicho abandono en el apartado *Conclusiones*.

- **Pérdida de información:** Siempre existe la posibilidad de perder el progreso del trabajo. Este riesgo es el que mayor impacto puede llegar a tener ya que puede cambiar la factibilidad entera del proyecto. Aunque es un riesgo con gran importancia es poco probable que suceda y como contra-medida, el proyecto estará guardado en un repositorio de GitHub donde se subirá el proyecto al terminar cada sesión de trabajo.

- **Problemas técnicos:** Puede que nos encontremos con un error o bug que no seamos capaz de solucionar y que afecte de manera negativa al proyecto. Este problema es difícil de cuantificar porque puede tener una importancia pequeña o extremadamente alta. La única solución que podemos tomar es intentar arreglar cualquier error que encontremos de manera inmediata y no acumularlos aunque esto implique un retraso sobre el horario.

- **Enfermedad o problemas de salud:** Este problema es difícil de abordar porque al final puede suceder cualquier cosa tanto al director como al alumno, por ello lo que

se propone es priorizar ciertas tareas que consideramos más relevantes aunque esto signifique tener que abandonar parte de los objetivos originales.

	Riesgo	**Impacto**
Estimación de tiempo errónea.	Medio	Medio/Alto
Complicación de tareas.	Bajo/Medio	Medio
Pérdida de información.	Bajo	Bajo/Alto
Problemas técnicos.	Bajo	Bajo/Alto
Enfermedad o problemas de salud..	Bajo	Bajo/Medio

Tabla 2.1: Análisis de riesgos.

2.5.2. Apartado especial COVID-19

Esta sección es especial por la extraordinaria situación en la que nos encontramos en el presente y que no predecimos en el apartado original de *Análisis de riesgos*. Esta extraña situación, altamente impredecible, ha afectado a todas las personas del mundo y también a los estudiantes que hemos estado trabajando en el TFG. Afortunadamente, por la naturaleza de nuestro campo de estudios, que es la informática, podemos permitirnos realizar gran parte, sino todo, el trabajo de forma telemática por ordenador. Por ello, dentro de lo que cabe, nuestro proyecto no ha sido afectado de manera excesivamente negativa. Las reuniones con el director del TFG *German Rigau Claramunt*, que inicialmente fueron hechas en persona, han podido ser realizadas por videollamada a través de *meet.jitsi*[15] sin grandes problemas excepto la pérdida de conexión eventual. Aunque si que es cierto que al no poder disponer de la sala de estudios especial para alumnos que están cursando master o TFG de la facultad de informática me he visto obligado a realizar por completo el trabajo desde mi hogar donde la concentración es algo más difícil.

Teniendo en cuenta esto, consideramos que el impacto de esta terrible pandemia para el desarrollo del proyecto ha sido, afortunadamente, bajo. El riesgo que representa para este también lo consideramos bajo porque nuestro trabajo puede ser realizado sin problemas de manera telemática.

	Riesgo	**Impacto**
COVID-19.	Medio	Bajo

Tabla 2.2: Análisis de riesgo especial COVID-19.

2.5.3. Análisis de viabilidad

Teniendo en cuenta los riesgos que acabamos de describir y el calendario y estimación de horas planteadas, consideramos el proyecto completamente viable. El TFG se comenzó con antelación de manera que las **300 horas** de trabajo estimadas se pudieran repartir en pequeñas dosis de trabajo de **3 o 4** horas diarias. El trabajo se comenzó con tiempo de sobra y las horas se han cumplido de manera satisfactoria.

[15] https://meet.jit.si/

3. CAPÍTULO

Estado del Arte

3.1. Conceptos básicos

Antes de comenzar con el proyecto, es necesario aclarar una serie de conceptos y objetivos del mismo.
El objetivo principal es el estudio y desarrollo de un sistema multiagente a través de una simulación de un ecosistema. Por ello, es necesario aclarar una serie de ideas sobre el campo de la **inteligencia artificial** como la toma de decisiones de los agentes en sistemas MAS. Teniendo en cuenta que también hemos creado un generador de procedural de terrenos, también explicaremos unos pocos conceptos necesarios para entenderlo al igual que haremos con los Boids.

3.1.1. Mesh

En Unity, un mesh contiene cierta información de un objeto, esta información es, los **vértices** que componen el objeto, los **índices de los triángulos** que van a formar las caras, los **vectores normales** que usará para los cálculos correspondientes a la iluminación y las **coordenadas uv** para mapear de manera correcta la textura del objeto en caso de que la tenga.

Esto es relevante porque el primer paso para generar un terreno es crear un mesh que podamos modificar ya sea con ruido o con un mapa de alturas.

3.1.2. Ruido Perlin

Es un tipo de **ruido gradiente** originalmente ideado por *Keneth Perlin*[Perlin, 1985] mientras participaba en la creación de la película *Tron* y su objetivo original era para la creación de texturas. Lo interesante de esta clase de ruido es que tiene **coherencia** entre sus valores, es un ruido semi-aleatorio donde los valores de un punto son relativamente similares a los de su alrededor. Esto es ideal para muchas clases de trabajos, entre ellos la generación de terreno que es la utilidad que le vamos a dar nosotros. Para generar nuestro terreno de manera procedural usaremos esta clase de ruido que explicaremos en mayor detalle en el apartado *Implementación: Generación de terreno*.

Existen múltiples métodos diferentes para generar ruido coherente como **Simplex** (o *'Perlin mejorado'* que fué creado por el mismo Keneth Perlin para tratar con algunos problemas del original), **OpenSimplex** (similar a Simplex pero de uso abierto) o **Wavelet**[DeRose, 2005] (creado y publicado por el estudio de animación de Pixar[1] para solucionar ciertos problemas de aliasing del ruido Perlin) entre otros. Nosotros hemos acabado eligiendo el **ruido Perlin** por comodidad ya que Unity contiene una implementación de esta clase de ruido dentro de la colección de funciones *Mathf*.

3.1.3. Boids

Los Boids son un programa de inteligencia artificial originalmente creado por *Craig Reynolds*[Reynolds, 1987] cuyo objetivo era simular el comportamiento de las manadas de pájaros. Los boids (*'bird-oid objects'*) son agentes que se mueven independientemente pero su comportamiento global acaba generando comportamientos de manada. Estos agentes se mueven en función de **tres reglas**. Estas reglas son, *cohesión* (el agente intenta colocarse en el centro de la manada), *separación* (mantienen una distancia mínima entre ellos para evitar colisionar) y *alineación* (el agente intenta seguir la misma dirección que

[1] Librería de artículos de Pixar: https://graphics.pixar.com/library/

la de la manada). Todo será explicado en mayor profundidad en el apartado *Implementación: Boids*.

3.1.4. Sistema BDI

Es un modelo de toma de decisiones para agentes inteligentes. BDI significa *Belief-Desire-Intentions* (Creencias-Deseos-Intenciones). Es un intento de acercar la toma de decisiones de agentes a la forma de tomar decisiones de los humanos. Está basado en la teoría desarrollada por *Michael E. Bratman* en su libro *Intention, Plans, and Practical Reason*[Bratman, 1987] sobre como funciona nuestra mente de manera fundamental a la hora de tomar decisiones para el futuro. Lo interesante de este modelo es su simplicidad al reducir la complejidad del comportamiento humano en una instancia más sencilla.

Los agentes tienen **tres componentes**, las **Creencias-Beliefs**, son un contingente de conocimientos acerca de otros agentes, objetivos, actividades o de su entorno. Estas creencias pueden ser modificadas a través de la comunicación con otros agentes, con su entorno o a partir de su propio razonamiento, es su conocimiento sobre el mundo. Los **Deseos-Desires**, representan el estado motivacional del agente, los objetivos o situaciones que el agente querría cumplir. Y las **Intenciones-Intentions** son la acción que el agente ha decidido tomar.

Esta clase de sistema de decisiones era una de las tareas originales del proyecto, queríamos que los animales tuvieran en cuenta estos tres componentes cuando tomaran decisiones. Aunque este modelo ha sido muy mejorado[Moncada,] e incluso criticado, decidimos incluirlo como tarea por su simpleza e interés.

3.1.5. Pathfinding

O *búsqueda de rutas*, es el trazado de camino más corto entre dos puntos. La ruta más corta entre dos puntos es un problema muy antiguo y estudiado, los algoritmos de pathfinding se especializan en esta clase de problemas. Existen multitud de diferentes algoritmo de pathfinding como **Dijkstra**, **A***, **D***, *Bellman Ford*, **Floyd Warshal**... Estos algoritmos nos interesan para que los agentes puedan encontrar el camino más corto entre ellos y su

objetivo. Hemos decidido usar **Bresenham** (aunque técnicamente no es un algoritmo de pathfinding) como algortimo principal y en caso de que no sirva, usaremos **A*** en concreto porque siempre que exista un camino válido, encontrará el óptimo.

3.2. Estado del arte

Los sistemas multiagente tienen una larga historia y infinitas utilidades. A través de la simulación de individuos (o agentes) que interactuan entre ellos y con su entorno, podemos sacar conclusiones acerca del comportamiento global, es decir, a través de la simulación de comportamientos *'micro'* podemos observar y sacar conclusiones del comportamiento *'macro'*. Pero, **¿cual es el interés del modelado basado en agentes?**. Según el artículo *Sistemas Multiagente y Simulación*[Caparrini, 2019], el intereses de estos modelos son cuatro:

- **La actividad autónoma del agente**: Los agentes toman las decisiones de manera propia e independiente con el fin de satisfacer sus objetivos propios.
- **La sociabilidad del agente**: Su capacidad de interactuar con otros agentes. Un ente MAS no es un ente completamente aislado sino un componente de una sociedad que a su vez surge del comportamiento global de los agentes.
- **La interacción es lo que conecta los dos conceptos anteriores**: A través de las interacciones surgidas en función de los dos conceptos anteriores es como pueden llegar a surgir comportamientos de competencia, cooperacion o conflicto que pueden ser estudiados.
- **La contextualización de los agentes**: Los agentes están en un entorno con el que interactúan para conseguir sus objetivos.

Con este paradigma, podemos modelar sistemas complejos donde podemos controlar todas las características y reproducir series de experimentos como si fuera un laboratorio. Uno de los primero campos que decidió experimentar con esta clase de sistemas fue la **ciencia social**. En el libro **Growing Artificial Societies**[Axtell, 1996] de *Joshua M. Epstein* y *Robert L. Axtell* responden a la pregunta, *¿como surgen las esctructuras sociales y comportamientos de grupo a partir de las interacciones de los individuos?* a través de

una simulación de un sistema que desarrollaron que llamaron **Sugarscape** sacando conclusiones de esta.

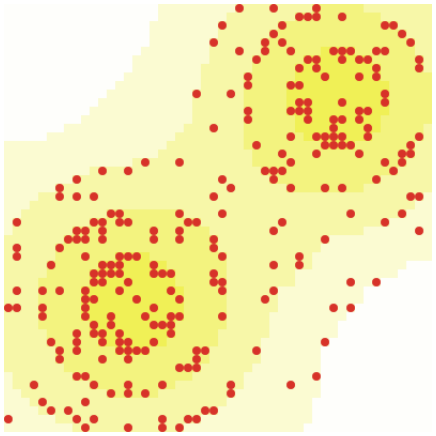
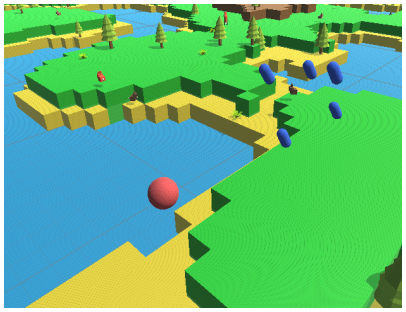


Figura 3.1: Simulación Sugarscape de la biblioteca de modelos de NetLogo.

Pero con los sistemas multiagente también podemos, por ejemplo, simular animales o más en concreto, comportamientos de manada. Para esta clase de comportamiento de grupo existe un modelo muy conocido que son los **Boids**[Reynolds, 1987] de *Craig Reynolds* donde con tres reglas básicas, **cohesión**, **alineación** y **separación**, consigue simular de manera bastante convincente manadas de pájaros o peces. Nosotros mismos hemos creado una implementación de este modelo para el proyecto.

(a) Implementación original de Boids.



(b) Nuestra implementación de los Boids.

Figura 3.2: Comparación de Boids originales y nuestra implementación.

Otro gran ejemplo de simulación de ecosistemas son las **colonias de hormigas**[Greco, 2012] o de otra cualquier clase de entes, como por ejemplo la simulación de unos entes llamados **Orphibs**[Nuno Barreto and Roque, 2014] o el conocido **juego de la vida** de *John Horton Conway* que fallecio recientemente (11 de abril) por complicaciones a causa del COVID-19. También podemos mencionar *JGOMAS*[2] con el que trabajamos en la asignatura de *ATAI ('Técnicas avanzadas de inteligencia artificial')* que es un entorno especial donde desarrollar modelos multiagente. Dentro de los modelos que lo componen encontramos el de un juego de *capturar la bandera* donde los agentes pueden tomar diferentes roles como *médico*, *soldado* y *apoyo* y tienen el objetivo de conseguir la bandera del equipo enemigo antes de que ellos lo hagan.

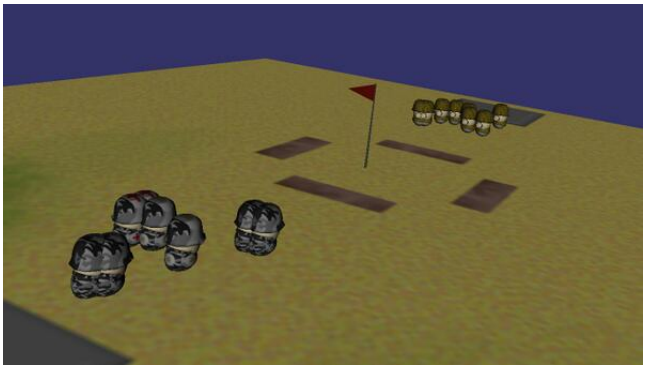


Figura 3.3: Captura del entorno JGOMAS.

Existen varias herramientas especialmente diseñadas para el desarrollo de esta clase de sistemas. Las más destacables son: *NetLogo*[3], con el que se puede trabajar de manera local o online, *JGOMAS*[4] que ya hemos mencionado o *GAMA*[5].

[2]http://gti-ia.dsic.upv.es/sma/tools/jgomas/index.php
[3]http://www.netlogoweb.org/
[4]http://gti-ia.dsic.upv.es/sma/tools/jgomas/index.php
[5]https://gama-platform.github.io/

4. CAPÍTULO

Desarrollo del proyecto

El proyecto que hemos desarrollado puede ser dividido en **tres** grandes apartados siendo estos: **Generación procedural de terreno**, **Boids** y **Ecosistema**. Estos tres apartados pueden considerarse como proyectos separados ya que cada uno puede funcionar independientemente sin el uso de los otros pero dentro del **Ecosistema** hemos unificado todos.

4.1. Captura de requisitos

En este apartado explicaremos cuales son los objetivos principales que queremos desarrollar durante el trabajo, cuales son los que hemos acabado cumpliendo y los problemas principales que tenemos que solucionar para una implementación correcta. Cada uno de los tres grandes apartados tienen sus propias tareas, objetivos y problemas propios.

4.1.1. Generación procesal de terreno

Partimos de un proyecto base[1] que contiene un generador de terreno propio que podemos usar para la simulación del ecosistema, pero uno de los objetivos originales era el crear nuestro propio generador de terrenos completamente independiente para no depender del código heredado. Y eso es lo que hemos hecho, el repositorio con nuestra implementación

[1] https://github.com/SebLague/Ecosystem-2/tree/master

de un generador procedural de terreno esta disponible en el siguiente repositorio[2]. Todo el código es de uso público y de momento no tiene licencia alguna de uso.

El objetivo inicial era el de conseguir un generador de terrenos que tuviera una temática de *'casillas'* similar al original del proyecto base para el ecosistema. Esto se ha conseguido ya que en nuestra implementación tienes la posibilidad de generar terreno tanto con temática de *'casillas'* como sin ella. Dentro de nuestro generador podemos jugar con los parámetros que usamos para generar el ruido, estos son **ancho y largo** (el mapa debe de ser cuadrado ya que de lo contrario las texturas se generarán de manera incorrecta), **número de octavas**, **persistencia**, **lagunaridad**, **escala**, **offset**, **multiplicador de altura** (solo útil cuando **no** tenemos un mundo formado por casillas), **semilla** y **usar mapa falloff o no**. Todos estos términos y sus utilidades serán explicadas en mayor detalle en el apartado *Diseño: Generación de terreno*. Una vez tenemos el ruido generado no hay ningún problema especialmente remarcable, únicamente el como acabar generando un mundo a partir de casillas.

4.1.2. Boids

Para darle un poco más de diversidad al ecosistema decidimos introducir **animales voladores** en el mundo. Para ello decidimos crear una implementación propia de los **Boids**, un programa de inteligencia artificial originalmente desarrollado por *Craig Reynolds* en 1986 publicado en su propio **artículo**[3]. Nuestra implementación está disponible en el siguiente repositorio[4] también de uso completamente público.

Dentro de los Boids podemos destacar **dos problemas** diferentes que tenemos que solucionar: **detección y evasión de obstáculos** y **optimización**. Para el **primer problema** tenemos que encontrar una manera de que los agentes sepan si van a colisionar o no y alguna manera de evitar esta colisión. El **segundo problema** tiene que ver con la eficiencia; los Boids siguen una serie de reglas básicas (que explicaremos en profundidad en el apartado *Diseño: Boids*) para las cuales tienen que detectar las posiciones de todos los boids del mundo, esto es de orden cuadrático pero lo hemos solucionado de dos maneras

[2]https://github.com/DripyDev/Generacion-de-terreno
[3]https://team.inria.fr/imagine/files/2014/10/flocks-hers-and-schools.pdf
[4]https://github.com/DripyDev/Boids

diferentes, a través de un **mapa** y a través del uso de **shaders** que explicaremos en mayor detalle más adelante.

4.1.3. Ecosistema

El ecosistema del proyecto está compuesto por **depredadores**, **presas**, **plantas** y **árboles**. Los objetivos iniciales eran que los agentes pudieran **alimentarse**, **beber**, **reproducirse y trasmitir genes** y **huir de los depredadores**. También queríamos tener alguna clase de sistema para analizar estadísticamente los datos relevantes como el número de agentes por tiempo o la velocidad media (la velocidad puede aumentar en función de los genes). Otra tarea original era diseñar un modelo BDI para la toma de decisiones de los agentes. El repositorio[5] correspondiente al ecosistema contiene la unificación de todos los proyectos. Los problemas que podemos resaltar sobre el ecosistema es el **pathfinding**, la **exploración de los agentes** y el **mapa de regiones**. Todo esto lo explicaremos en la sección *Diseño: Ecosistema*.

Es en este apartado donde hemos aprovechado parte del código que heredamos del proyecto base[6] aunque hemos creado de cero varias partes y hemos implementado otras tantas características nuevas. Todo esto lo especificaremos mejor en el capítulo *Arquitectura del sistema: Arquitectura*.

4.2. Análisis

Sobre las tareas originales, hemos completado la mayoría de manera satisfactoria. Dentro de la **Generación de terreno** hemos completado todos nuestros objetivos, lo cual no significa que no haya trabajo futuro.

Dentro de los **Boids** la única tarea que ha quedado pendiente (aunque le dedicamos algunas horas para intentar completarla) es el uso del mapa en el que subdividimos el mundo dentro de los shaders cosa que, aunque intentamos, decidimos abandonar porque la velo-

[5] https://github.com/DripyDev/PruebaEcosistem-boid
[6] https://github.com/SebLague/Ecosystem-2/tree/master

cidad del sistema era más que suficiente para el uso que iba a tener.

Y por último, dentro de **Ecosistema** quedó pendiente el sistema de toma de decisiones en función de una arquitectura **BDI (Belief-Desire-Intention)** que aunque preparamos teóricamente, nunca llegamos a implementar físicamente por falta de tiempo. Todo esto, por supuesto, no significa que no quede trabajo y mejoras futuras. Pero dentro de los objetivos originales, hemos conseguido cumplir la mayoría satisfactoriamente (teniendo en cuenta que probablemente haya varios apartados que puedan ser mejorados).

4.3. Arquitectura del sistema

Vamos a explicar las tecnologías que hemos utilizado para el desarrollo del proyecto, vamos a hablar de la arquitectura general del sistema (de los scripts que lo componen) y de las diferencias y elementos que hemos aprovechado del proyecto base[7].

4.3.1. Elección de tecnología

Todo nuestro proyecto ha sido implementado en **Unity**. Unity es un motor de videojuegos muy conocido y popular. Como lenguajes puede soportar **Javascript** o **C#** y para los shaders usa **HLSL** (*High-Level Shading Language*). En nuestro caso, todo el proyecto ha sido creado con **C#** y parte con **HLSL** para el apartado de optimización de los Boids.

Se ha escogido esta plataforma porque presenta muchas facilidades para el desarrollo visual de mundos y por la simpleza que aporta para sistemas multiagente como el nuestro. También al tener popularidad, contiene una gran comunidad que ayuda a resolver problemas de forma activa. Cuenta con una buena y extensa documentación dentro de su propia API[8] donde describen el funcionamiento de todas las clases y componentes que usa Unity lo que agiliza considerablemente el trabajo. Es una herramienta gratuita lo cual se agradece teniendo en cunta nuestra posición de estudiantes (aunque nosotros hemos tenido acceso a la versión de pago gracias a un paquete de Github para estudiantes que hemos mencionado en el apartado *Herramientas utilizadas*). Y es el lugar donde estaba

[7] https://github.com/SebLague/Ecosystem-2/tree/master
[8] https://docs.unity3d.com/2019.3/Documentation/ScriptReference/

desarrollado el código base que usamos para parte del **Ecosistema**.

A continuación explicaremos de forma general el funcionamiento del inspector de Unity y algunos conceptos básicos del mismo. La interfaz de Unity está compuesta de **cinco** apartados principales que son: la ***toolbar*** situada en la parte superior, la ***hierarchy window*** a la izquierda donde podemos ver la jerarquía de los objetos de la escena, la ***scene view*** en el centro donde vemos la escena que hayamos elegido, el ***inspector window*** a la derecha que nos muestra la información pública del objeto que hayamos elegido y el ***project window*** en la parte inferior donde están todos los archivos generados para el proyecto.

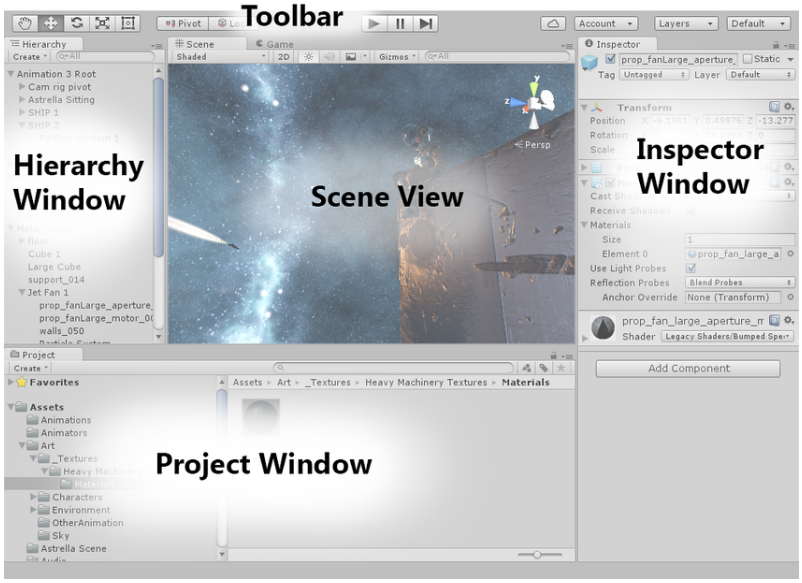


Figura 4.1: Interfaz de Unity.

Vamos a explicar **cuatro** conceptos diferentes para entender mejor la aplicación que hemos creado. Estos son: **Escena**, **GameObject**, **Script** y **Prefab**.

Una **Escena** es el entorno donde se genera el mundo virtual del videojuego. Está compuesta por GameObjects tales como objetos 3D, cámaras, fuentes de iluminación, objetos para interfaces etc etc.

Los **GameObjets** son objetos (vacíos o no) a los que añadimos propiedades como posiciones en el mundo, detectores de colisión, cámaras, físicas etc. A estos objetos se les puede asignar scripts para controlar su funcionamiento. Por ejemplo, podemos crear un

GameObject vacío, añadirle una forma de cubo y un script que lo mueva por el mundo en función de un input del teclado. Es el componente base de la escena.

Los **scripts** que pueden ser escritos en **C#** o en **javascript** pueden ser asignados a los GameObjets para controlar su funcionamiento. Aunque no tienen porque ser asignados.

Un **prefab** es, de manera resumida, una instancia de un GameObject con ciertos valores en sus propiedades que guardamos para poder clonar de manera rápida y cómoda. Un GameObject solo existe dentro de su Escena lo cual nos dificulta el poder usar dicho objeto (o similar) en otra diferente. Los prefabs nos facilitan esto ya que podemos crear una instancia de dicho objeto para poder usar en cualquier parte y tendrá los valores que hemos definido. Los conejos, zorros, plantas, árboles, boids y depredadores son todos prefabs.

La aplicación ha sido desarrollada con la versión **2019.3.0f6** de Unity y dos paquetes externos llamados Simplest Plot[9] utilizado para las gráficas que presenta la simulación y Text Mesh Pro[10] para los textos que aparecen en las interfaces de la pantalla. Ambos son paquetes gratuitos disponibles en la tienda de paquetes de Unity o *Asset store*[11]. Para la edición del código, como hemos explicado en el apartado *Herramientas utilizadas*, usamos **Visual Studio Code**.

4.3.2. Arquitectura

En este apartado desglosaremos el funcionamiento del programa desarrollado hablando de la arquitectura general del sistema y de los scripts principales que los componen y sus utilidades. El proyecto puede ser dividido en **tres partes**: **Generación de terreno**, **Boids** y **Ecosistema**.

4.3.2.1. Generación de terreno

Los entes necesitan un entorno donde moverse libremente y para eso debemos de generar un terreno. El generador de terrenos que hemos creado está compuesto por **cinco scripts**

[9] https://assetstore.unity.com/packages/tools/gui/simplest-plot-94876
[10] https://docs.unity3d.com/Manual/com.unity.textmeshpro.html
[11] https://assetstore.unity.com/

diferentes.

- **EditorGT.cs**: Un script especial para modificar el aspecto de la *inspector window* correspondiente a GeneradorTerreno.
- **FalloffGenerator.cs**: Una clase estática para generar un mapa Falloff.
- **Ruido.cs**: Una clase estática para generar el mapa de ruido que usamos para el terreno.
- **GeneradorTextura.cs**: Una clase estática que dado un mapa de alturas lo devuelve convertido en textura.
- **GeneradorTerreno.cs**: Script principal que con el ruido y las texturas, crea y altera los vértices del terreno y les asigna la correspondiente textura.

4.3.2.2. Boids

Para añadirle un poco más de diversidad al ecosistema, decidimos incluir animales voladores en el mundo. Estos animales voladores están divididos en dos clases, una llamada *Boids* que son presas y otra llamada *Depredadores* que, como el nombre indica, serán los depredadores de los Boids.

Su funcionamiento está compuesto por **seis scripts** principales diferentes. Estos son, **Boid.cs-Depredador.cs**, **RegionManager.cs**, **Spawner.cs**, **BoidCompute.compute-DepredadorCompute.compute** (los shaders) y **BoidSettings.cs-DepredadorSettings.cs**. También existe un script **CameraController.cs** para controlar el movimiento de la cámara y **MenuController.cs** para controlar los elementos de las interfaces (como los sliders para alterar los valores de los pesos de las reglas, acelerar el tiempo etc).

Cada uno de estos scripts tiene su cometido.

- **BoidSettings.cs - DepredadorSettings.cs:**

Estas son unas clases especiales que derivan de *ScriptableObject*. ScriptableObject es un contenedor de datos que se usa para almacenar grandes cantidades de información. Esto

es útil para evitar la copia innecesaria de datos en un proyecto ya que si todo el proyecto va a usar una serie de datos constantes y comunes, no hace falta tener varias copias de lo mismo.
Esto es perfecto para nosotros, ya que todos los boids comparten una serie de datos tales como los pesos de las reglas, el radio de visión, la distancia de separación, la velocidad máxima y mínima y la máscara de obstáculos a evitar. Lo mismo ocurre con los depredadores pero con menos reglas y sin distancia de separación.

La clase contiene la etiqueta **[CreateAssetMenu]**, gracias a la cual podemos crear un objeto dentro del inspector de Unity, una instancia de la clase que vamos a asignar al boid o al depredador desde *Spawner.cs*. Esta instancia será única para todos los agentes, así evitamos guardar datos repetidos.

- **RegionManager.cs:**

Es el encargado de dividir el mundo en regiones más pequeñas y administrarlas. Vamos a explicar en mayor profundidad su funcionamiento en el apartado *Subdivisión del mapa*.

En la escena, existe un objeto vacio con este script adjuntado. Al despertar el objeto (al comenzar la simulación) creará la subdivisión del mapa en función del número de regiones que queramos. Este número tiene que tener una raíz cúbica entera y la subdivisión se hará en cubos de mismo tamaño.

Esta clase contiene una lista estática de la estructura que hemos llamado **Region**. Esta estructura contiene su posición en el mundo, sus dimensiones, una lista de boids en ella y otra con los depredadores. Esta variable es la que representa la subdivisión del mapa per se. Todas las funciones para administrar el mapa, tales como eliminar/añadir un boid o depredador a una región, están implementadas en esta clase.

- **Spawner.cs:**

Su función es aparecer y inicializar los agentes iniciales y en el caso de que queramos usar la GPU, administrarla.

Tanto los boids como los depredadores tienen la función **Inicializar(settings, posicion)** donde *settings* es una instancia de la clase **BoidSettings.cs** o **DredadorSettings.cs** y *posicion* es la posición en el mundo donde va a aparecer el agente.
Como explicaremos en el apartado *Uso de la GPU*, en caso de que queramos usar un shader para procesar la información, **Spawner.cs** se encargará de ello.

- **Boid.cs - Depredador.cs:**

Son los scripts encargados de hacer funcionar a los agentes. Si usamos la CPU, los cálculos de cada agente se realizan en estos scripts, si usamos los shaders, solo se calculará la regla para evitar los choques.

Los cálculos de las reglas se detallarán en las secciones *Reglas básicas* y *Depredadores*.

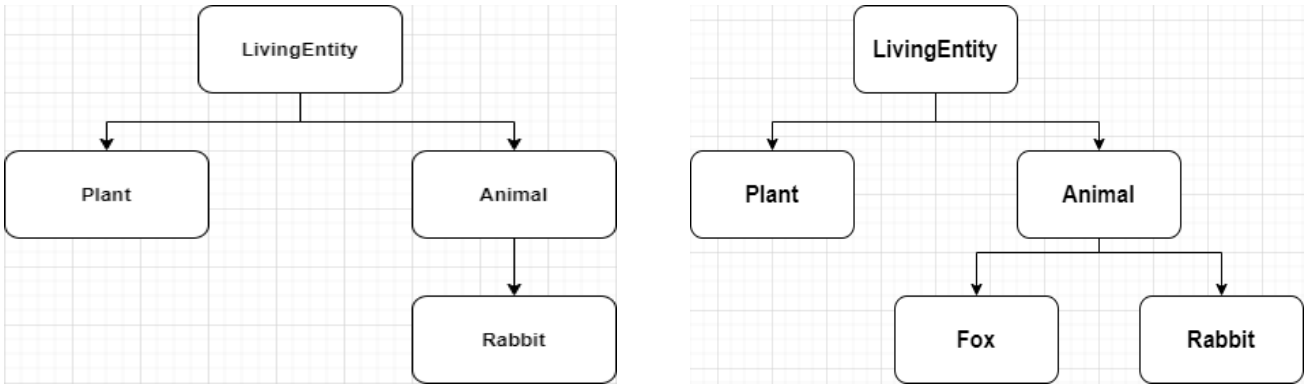
- **BoidCompute.compute - DepredadorCompute.compute:**

Los shaders para los cálculos de las reglas básicas de cada agente. Es una sola función que recorrerá todos los boids y depredadores existentes y calculará las reglas en función de ello. No usa la subdivisión del mapa en regiones así que la eficiencia es del orden **O(n*n)** pero al usar los recursos de la tarjeta gráfica es significativamente más rápido. Su funcionamiento está explicado en más detalle en el capítulo *Uso de la GPU*.

4.3.2.3. Ecosistema

En el proyecto base, los conejos podían comer hierba, no existía forma de reproducción y los zorros no podían comer a los conejos. Las entidades podían morir de hambre o sed pero no se detectaban entre ellos. Hemos dejado de utilizar varios scripts porque hemos rehecho desde cero gran parte del proyecto para amoldarlo a nuestro gusto. Por lo general, lo que más hemos aprovechado es el diseño general del sistema correspondiente al script **Environment.cs** perteneciente al grupo **Environment** y parte de la forma de trabajar del script **Animal.cs** del grupo **Comportamiento**. A continuación vamos a describir los scripts originales y cuales hemos usado, cuales no, cuales hemos modificado y cuales hemos reimplementado de cero. El ecosistema está principalmente compuesto de **tres grupos** de scripts diferentes:

- **Comportamiento**: Es donde se han implementado los scripts que controlan todas las acciones de los agentes y la jerarquía de estos.



(a) Diagrama original. **(b)** Diagrama nuestro.

Figura 4.2: Comparación diagramas 'Comportamiento' original y nuestro.

- *LivingEntity.cs*: Contiene la información que comparten los animales y las plantas. Esta información es sus coordenadas en el mundo, las coordenadas de la región en la que se encuentran y si están vivos o muertos. Lo hemos aprovechado y hemos modificado una función para proyectar la información de su muerte en pantalla y si queremos guardarla en un archivo que se crea al finalizar la simulación.
- *Plant.cs*: Script que controla como son devoradas las plantas. Lo utilizamos y hemos incluido una función para que la planta vaya recuperandose con el tiempo siempre y cuando no la hayan consumido por completo.
- *Animal.cs*: Script padre de **Fox.cs** y **Rabbit.cs**, controla como encuentran comida, pareja y agua, cuando crecen y se embarazan. Controla todas las acciones que pueden compartir las entidades. Lo hemos rehecho aprovechando parte del diseño general. También hemos añadido la posibilidad de que puedan **reproducirse**, **crecer**, **tener genes**, **los conejos huir de los depredadores** y **hemos cambiado la toma de decisiones**.
- *Rabbit.cs*: Originalmente este script está incompleto y no es utilizado. Nosotros hemos incluido aquí la toma de decisiones de los conejos cuya mayor diferencia con los zorros es que huyen de estos.
- *Fox.cs*: Script que hemos creado nosotros de cero. Al igual que con el script **Rabbit.cs**, controla el comportamiento del zorro.

- **Datatypes**: Son clases con tipos de datos personalizados para nuestro ecosistema. Estas clases contienen información como, por ejemplo, las diferentes causas de muerte posibles en el mundo que hemos simulado.

    - *CauseOfDeath.cs*: Una clase con las diferentes forman de morir de los entes que son: *hambre*, *sed*, *devorado* y *edad*. Lo utilizamos y hemos añadido la posibilidad de que mueran siendo devorados o por edad.
    - *Coord.cs*: Una clase de coordenadas de dos dimensiones. **No** lo utilizamos ya que en su lugar hemos usado el tipo de datos de Unity **Vector2**.
    - *CreatureAction.cs*: Clase con las diferentes posibles acciones de los agentes que son: *ninguna, desacansando, explorando, yendo a por comida, yendo a por agua, comiendo, bebiendo, buscando pareja, reproduciéndose y huyendo*. Lo utilizamos aunque las dos últimas las hemos añadido nosotros.
    - *Genes.cs*: Encargado de crear los genes aleatoriamente. Lo hemos **rehecho** de cero incluyendo la posibilidad de que los genes sean aleatorios o en función de los padres.
    - *Species.cs*: Clase con las posibles especies del mundo que son *indefinido, planta, zorro y conejo*. Lo utilizamos.
    - *Surroundings.cs*: Era una clase con datos que se utilizaban en Environment pero **no** lo usamos porque hemos rehecho dicho script y ya no era necesario.

- **Environment**: Dentro de este grupo están los scripts que se encargan de administrar el mundo en general y de gestionar el pathfinding y la división del mapa en regiones más pequeñas.

    - *Environment.cs*: Es el script principal, el encargado de **registrar eventos** como *movimientos* y *muertes*, dar la información que pueden sentir las entidades, hacer aparecer las poblaciones iniciales y los árboles y el movimiento del agente cuando está explorando. Lo hemos rehecho dentro del script **Mundo.cs** pero hemos empleado parte del diseño que usa para la información del mundo.
    - *EnvironmentUtility.cs*: Era la clase donde estaba implementado la forma de encontrar rutas entre dós puntos. Lo hemos rehecho de cero dentro del script **Pathfinder.cs** donde hemos implementado el algoritmo de **Bresenham** y **A*** (*'A-Star'*) en caso de que Bresenham no de un resultado válido.

- *Map.cs*: El encargado de dividir el mundo en regiones más pequeñas para optimizar la búsqueda de entidades cercanas. Lo hemos implementado de cero en el script llamado **Mapa.cs**.

- **Terrain**: Es el generador de terreno. El original no lo usamos ya que hemos implementado el nuestro completamente de cero y hemos explicado su arquitectura en el apartado *Arquitectura: Generación de terreno*.

4.4. Implementación

En este apartado vamos a explicar en mayor profundidad el funcionamiento de cada parte y como hemos solucionado los problemas que hemos ido encontrando.

El proyecto puede ser dividido en **tres apartados principales**: la *generación de terreno*, los *Boids* y el *ecosistema*. Cada uno de estos tres apartados va a ser explicado en detalle a continuación.

4.4.1. Generación de terreno

Para que los agentes puedan moverse libremente necesitan tener un terreno sobre el que andar. El código heredado tiene una implementación de un generador de terreno pero hemos decidido realizar nosotros mismos desde cero nuestro propio generador.

Este apartado puede ser dividido en tres, **generación del mesh**, **generación de ruido y mapa de colores** y la **generación del terreno final**. El primer apartado trata de la forma en la que tenemos que crear los vértices, las caras de estos y toda la información necesaria para la creación del terreno. El segundo es el ruido que vamos a utilizar para determinar la altura y color del mundo y la tercera trata sobre la implementación final que hemos realizado para su uso dentro del ecosistema.

4.4.1.1. Generación del mesh

Lo primero que necesitamos saber es la forma en la que Unity trata los vértices y la creación de los meshes. Un **mesh** es un objeto que contiene la información de un objeto, contiene las **posiciones de sus vértices**, los **índices de los triángulos** que componen las caras del mesh, las **coordenadas uv** de la textura y los **vectores normales** de las caras. Para formar correctamente un objeto necesitamos toda esa información, los vértices y los índices de los triángulos para darle la forma y el resto para su textura y iluminación. A la hora de formar triángulos a partir de los vértices, **Unity usa el sentido horario**. Si, por ejemplo, tuviéramos cuatro vértices (0,0), (1,0), (0,1) y (1,1), los dos triángulos que formarían la cara de esta casilla serían (0,0)-(1,0)-(0,1) y (0,0)-(1,0)-(1-1). En el siguiente diagrama se verá de forma más clara. Es importante tener en cuenta que existen combinaciones diferentes para acabar formando los mismos triángulos ya que lo único que importa es que se siga el sentido horario.

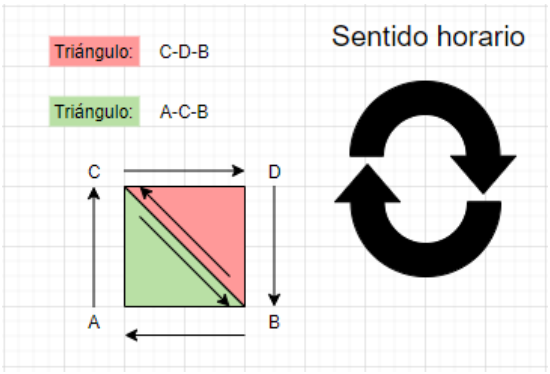


Figura 4.3: Triángulos a partir de vértices.

Sabiendo esto, podemos crear el mesh aunque de momento no tendremos en cuenta la altura de los vértices. Si formamos un cuadrado de altura y anchura **X**, vamos a necesitar **X*X vértices**, **X*X normales** (una por vértice), **X*X coordenadas uv** y **(X-1)*(X-1)*6 índices de triángulos** ya que cada cara está formada por dos triángulos de tres vértices cada uno y la última fila y columna no hacen falta porque forman parte del cuadrado formado una posición antes. El código para formar el mesh es simple:

```
1 void GenerarMesh(int ancho, int alto)
2     int indiceVertice=0;
3     int indiceTriangulo=0;
4     for(x=0, x<ancho, x++)
5         for(y=0, y<largo, y++)
6             vertices[indiceVertice] = (x,0,y);
```

```
7
8               //Triangulo1
9               triangulos[indiceTriangulo] = indiceVertice;
10              triangulos[indiceTriangulo+1] = indiceVertice + y + 1;
11              triangulos[indiceTriangulo+2] = indiceVertice + y;
12              //Triangulo2
13              triangulos[indiceTriangulo+3] = indiceVertice;
14              triangulos[indiceTriangulo+4] = indiceVertice + 1;
15              triangulos[indiceTriangulo+5] = indiceVertice + y +1;
16
17              indiceTriangulo+=6;
18              indiceVertice++;
```

Con esto, tenemos listo un objeto cuadrado con altura nula.

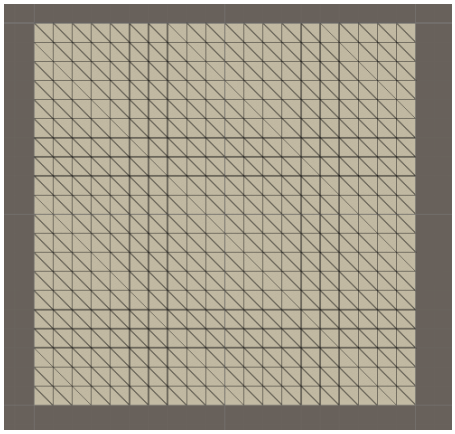
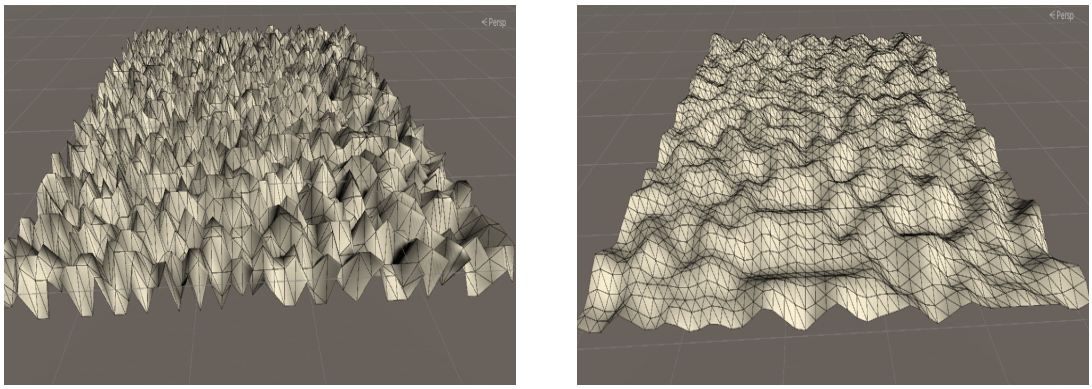


Figura 4.4: Mesh de triángulos con altura nula.

4.4.1.2. Generación de ruido y mapa de colores

Con el mesh ya creado y listo, vamos a modificar su altura, para ello podríamos usar valores aleatorios pero esto generaría un resultado poco deseable porque los puntos no tendrían ninguna clase de relación entre ellos. Queremos aleatoriedad pero no del todo, queremos que los valores de altura de cada punto tengan cierto grado de relación entre sus adyacentes ya que de lo contrario no conseguiremos efectos similares a los encontrados en la naturaleza. Para ello lo que vamos a usar es **ruido Perlin**. Este es un ruido gradiente creado por **Ken Perlin** y es ampliamente usado en el mundo de la generación procedural de terreno y en otros muchos campos. La libreria **Mathf** de Unity contiene una función llamada **PerlinNoise(x,y)** que dados dos números, devuelve la correspondiente altura re-

sultado del ruido Perlin. Si a cada vértice le damos como altura el valor correspondiente a (x,y) del ruido Perlin, conseguimos un resultado mucho mejor.



(a) Mesh con altura aleatoria. **(b)** Mesh con ruido Perlin.

Figura 4.5: Meshes de altura no nula.

Sobre este mesh podemos realizar una serie de mejoras. **PerlinNoise(x,y)** da valores entre uno y cero (aunque a veces pueden ser ligeramente superiores o inferiores a uno o cero) en función de los dos número que le pases, en esta primera implementación le pasamos los valores enteros de X e Y de los bucles que usamos para crear los vértices que componen el mesh. Pero el ruido Perlin funciona mejor cuando los valores que le pasamos tienen diferencias pequeñas entre ellos, es decir, es mejor pasarle valores con, por ejemplo, 0.1 de diferencia que 1 de diferencia. Esto es fácilmente solucionado dividiendo X e Y entre 10, por ejemplo.

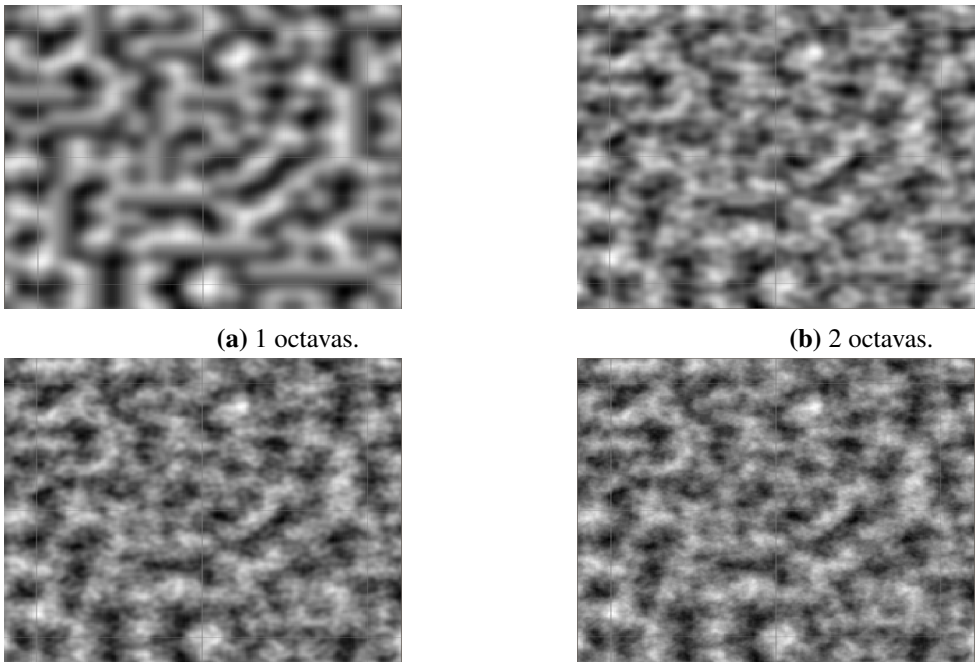
A la hora de generar el ruido, podemos añadirle múltiples mejoras, entre ellas las más importantes son el uso de **octaves**, una **semilla** para generar número aleatorios, **persistencia** y **lagunaridad** que a su vez afectan a la **amplitud** y **frecuencia**.

La **amplitud** de una onda es el rango de alturas que puede tomar mientras que la **frecuencia** es la inversa de la longitud de esta (Frecuencia=1/longitud). Las **octavas** son capas de ruido que vamos a acumular la una sobre la otra pero cada una tendrá un efecto menor y más específico. La **amplitud** afectará al rango de valores que podrá tomar cada capa y será reducido en función de la **persistencia** por lo que cada capa irá teniendo un rango de valores más pequeños lo cual causará como resultado una mayor granularidad en la capa final. Por otro lado, la **frecuencia** afecta a cuanto 'zoom' va a haber sobre el ruido en cada capa, cada capa tendrá una mayor frecuencia que la anterior para acumular detalles y será

multiplicada por la lagunaridad que suele valer 2 por defecto. Por otro lado, la **semilla** la usamos para generar números aleatorios. A la hora de acumular capas con las octavas, queremos que estas capas sean lugares diferentes del ruido por lo que las moveremos en función de números aleatorios generados gracias a la semilla.

- **Octaves**

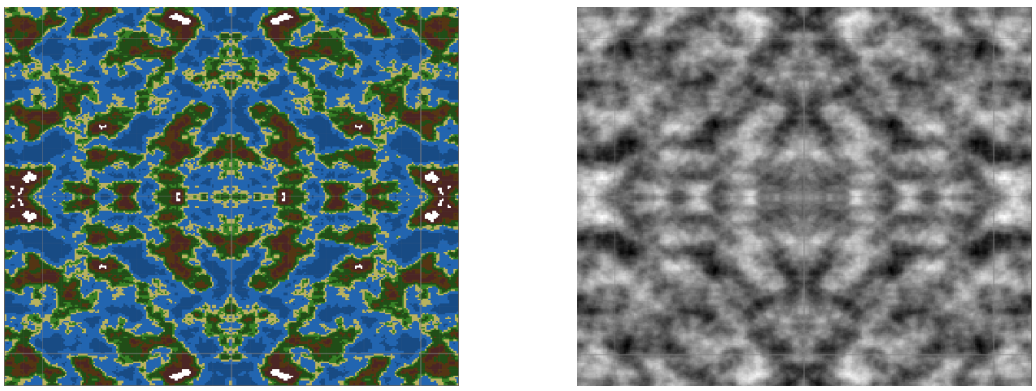
Para generar ruido con más detalles lo que podemos hacer es acumular varias capas de ruido unas sobre otras. Este número de capas son las **octavas** (o *'octaves'*), el número de capas que van a formar el ruido final, a mayor número, mayor detalle. Aunque es importante darse cuenta de que para que las octavas den buenos resultados es necesario tener en cuenta la **persistencia**, la **amplitud**, la **lagunaridad** y la **frecuencia** porque de lo contrario acabaremos con un ruido similar al blanco lo cual no queremos. También queremos que cada capa sea, a ser posible, de un lugar diferente del ruido por lo que sería interesante conseguir que cada octave sea de zonas diferentes. Esto es facil de conseguir ya que solo tenemos que generar una coordenada (x,y) aleatoria para cada octava con valores entre, por ejemplo, -10000 y 10000. Luego sumaremos esta coordenada (o *'offset'*) a los valores que usaremos para el ruido. De esta forma todas las capas vendrán de orígenes diferentes. Si no hiciéramos esto conseguiríamos un resultado interesante pero indeseable que mostramos a continuación.



(a) 1 octavas. **(b)** 2 octavas.

(c) 3 octavas. **(d)** 4 octavas.

Figura 4.6: Número de octavas con lagunaridad=2 y persistencia=0.5.



(a) Mapa de colores. **(b)** Ruido.

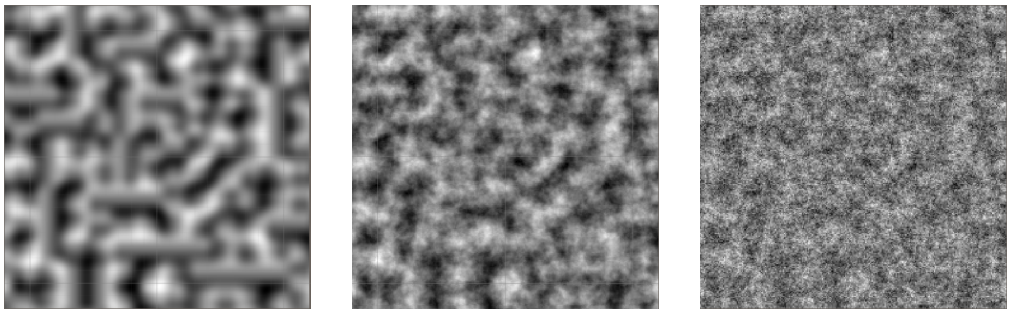
Figura 4.7: Generación de mapa de colores y ruido sin diferentes orígenes por octava.

A partir de unas cinco capas, apenas hay diferencia aunque esto es principalmente debido a la persistencia.

- **Persistencia**

Es el número por el que se multiplica la **amplitud** de cada octava, tiene valores entre cero y uno. Afecta a la amplitud de onda que vamos a usar, esto significa que al reducir los

posibles valores que pueden tomar las capas vamos a tener resultados más granulares. A mayor persistencia, mayor aspereza y granularidad en el resultado final. Si, por ejemplo, tenemos un valor de persistencia de 0.5, la amplitud de la primera capa será 1, la de la segunda 0.5, la tercera 0.25 etc etc. Cada capa añadida debería de tener menor importancia y menor rango de valores porque es esto lo que al ser acumulado con las otras capas genera más detalles pequeños.

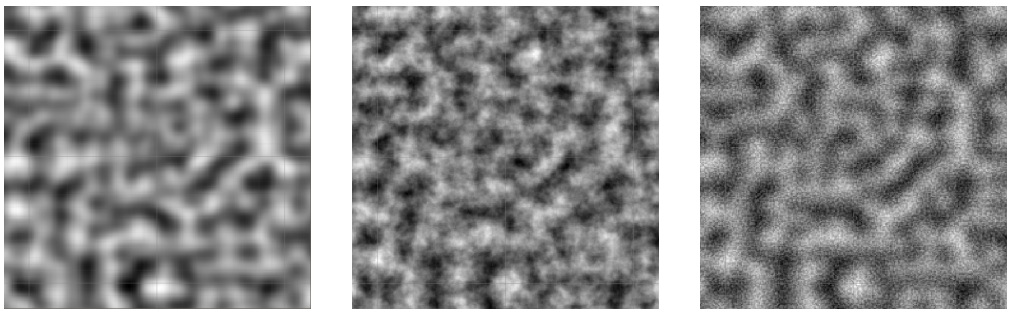


(a) Persistencia 0. **(b)** Persistencia 0.5. **(c)** Persistencia 1.

Figura 4.8: Valores de persistencia con 6 octavas y lagunaridad=2.

- **Lagunaridad**

La **lagunaridad** multiplica a la **frecuencia** en cada octava. Cuanto mayor sea la lagunaridad, mayor será la frecuencia de cada octava lo cual se traduce a un *'zoom'* sobre el ruido, esto causa como resultado que cada capa que añadimos esté más aumentada sobre el ruido por lo que tendrá más detalles. Su valor por defecto suele ser de 2. A mayor valor, mayor zoom sobre la textura lo cual se traduce a mayor detalle en zonas pequeñas, cuanto menor, menor detalle por lo que la frecuencia de cada capa será igual lo que causará que los valores acaben más difuminados.



(a) Laguranidad de 0. **(b)** Lagunaridad de 2. **(c)** Lagunaridad de 10.

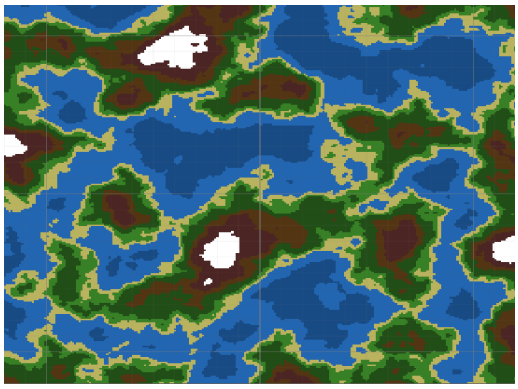
Figura 4.9: Valores de lagunaridad con 6 octavas y persistencia=0.5.

Algo que tenemos que tener en cuenta es que al acumular ruido con las octavas es que los valores ya no van a ser entre cero y uno, habrá varios valores superiores. Por ello deberíamos de almacenar la altura más baja y alta generada para después normalizar todos los valores entre estas dos alturas. Si **no hiciéramos esto**, el ruido tendría un predominante color blanco por la acumulación de capas (el color blanco es el correspondiente a un valor cercano al 1).

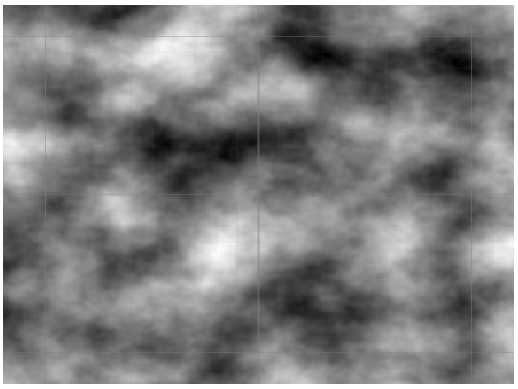
```
float[,] GenerarRuido(int ancho, int alto, float escalado, int octaves,
                    float persistencia, float lacunaridad)
    //Calculamos los offset de cada octava
    for(i=0;i<octaves;i++)
        offsetOctava[i]=DosNumerosAleatoriosEntre(-10000,10000);

    //Maximo y mininmo del mapa para normalizar despues
    maximaAltura = -infinito;
    minimaAltura = infinito;

    for(y=0;y<largo;y++)
        for(x=0; x<ancho;x++)
            amplitud=1;
            frecuencia=1;
            for(i=0;i<octaves;i++)
                X= (x+offsetOctava[i].x) / escalado * frecuencia;
                Y= (y+offsetOctava[i].y) / escalado * frecuencia;
                valorPerlin=PerlinNoise(X,Y);
                alturaRuido+=valorPerlin*amplitud;

                amplitud+=persistencia;
                frecuencia*=lagunaridad;

            maximaAltura = Mayor valor de ruido encontrado
            minimaAltura = Menor valor de ruido encontrado

            mapaRuido[x,y] = alturaRuido;

    Normalizamos los valores de mapaRuido entre maximaAltura y minimaAltura

    return mapaRuido;
```

Ahora que tenemos un mapa de ruido funcional, tenemos que traducirlo a un mapa de colores. Esto es sencillo, lo primero que necesitamos es una estructura para almacenar los *'terrenos'* que vamos a tener, dentro de cada uno debería de haber un color y una altura. Todo valor menor a dicha altura tendrá ese color. Tras esto, solo tenemos que recorrer el mapa de ruido que hemos generado y generar un mapa de colores que convertiremos a

una textura.



(a) Mapa de colores.



(b) Ruido.

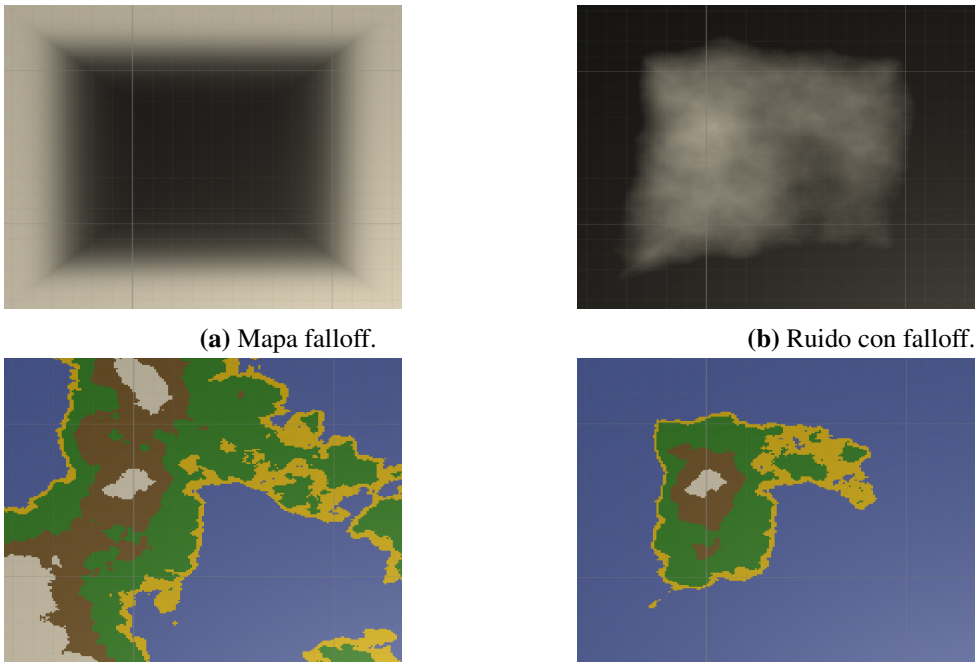
Figura 4.10: Comparación de ruido y mapa de colores.

Como punto final, podemos usar un **falloff map** para generar ruido. Este mapa es un tipo de mapa muy usado en el mundo 3D y en el mundo de los videojuegos. Suele ser usado para darle un gradiente a los colores en función de la posición de una cámara o de una fuente de luz etc. Generarlo es sencillo, queremos generar un mapa de X de alto y Y de largo. Recorremos las dos dimensiones y el valor del mapa falloff correspondiente de la matriz será la evaluación del mayor entre X e Y sobre una curva que en nuestro caso es:

Donde **a=3** y **b=2.2**. Aunque podríamos usar cualquier otra clase de curva.

$$f(x) = \frac{x^a}{x^a + (b - bx)^a}$$

Una vez generado el mapa, le extraeremos al valor del mapa de alturas el correspondiente del falloff. Esto resultaría en un ruido muy centralizado lo que se traduce a una *isla*. La utilidad de esto es para estudiar el ecosistema en mundos y situaciones diferentes. Se ha implementado principalmente para aprender a implementarlo y usarlo, no por su utilidad dentro de nuestro ecosistema.



(a) Mapa falloff.

(b) Ruido con falloff.

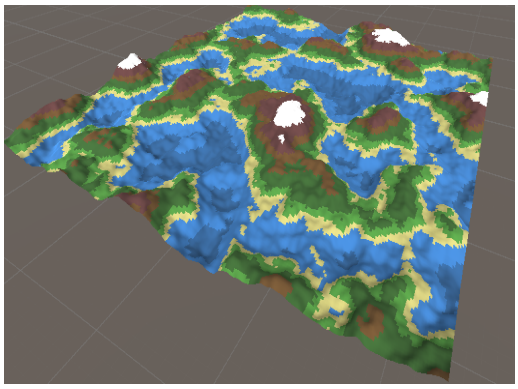
(c) Mapa colores sin falloff.

(d) Mapa colores con falloff.

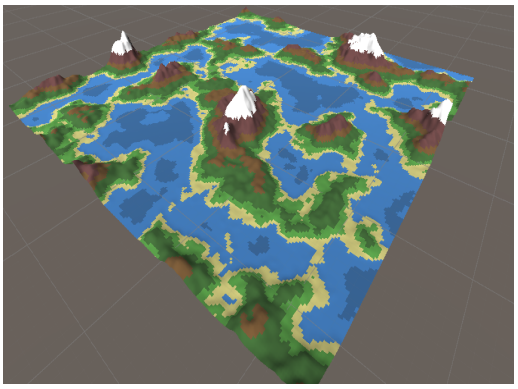
Figura 4.11: Ejemplos de uso del mapa falloff.

4.4.1.3. Generación del terreno final

Ahora que ya tenemos el mapa del ruido y la textura generada a través del mapa de colores, podemos generar un terreno con altura no nula y colores. Para la altura de cada vértice usamos el correspondiente valor del ruido multiplicado por un número alto arbitrario porque los valores del ruido son pequeños. Esto nos genera un problema y es que los vértices correspondientes al agua también tendrán cierta altura y esto no es deseable. Queremos que el agua tenga una altura nula o muy muy baja mientras el resto de puntos si que tengan altura. Para conseguir esto vamos a hacer uso de una **curva** para evaluar los valores en cada punto de la misma. Creando una curva, podemos alterar los valores de cierto umbral a nuestro gusto. Unity contiene un tipo de dato llamado **AnimationCurve** que tiene precisamente esta utilidad. Con esta nueva curva vamos a hacer que los puntos correspondientes al agua no tengan altura y el resto de los puntos si que la tengan.



(a) No usando la curva.



(b) Usando una curva.

Figura 4.12: Comparación del mundo con y sin curva.

Con esta función se pueden conseguir muchos efectos diferentes, podríamos, por ejemplo, hacer que las montañas estén más pronunciadas que el resto del terreno o darle más profundidad al agua pero solo en las zonas más profundas. En nuestro caso no hemos experimentado demasiado con esto porque el terreno final que usamos no tiene necesidad de esta curva ya que el mundo estará generado por casillas cuyos vértices compartirán altura. Estos resultados son parte del experimento realizado durante el aprendizaje sobre la generación procedural de terreno.

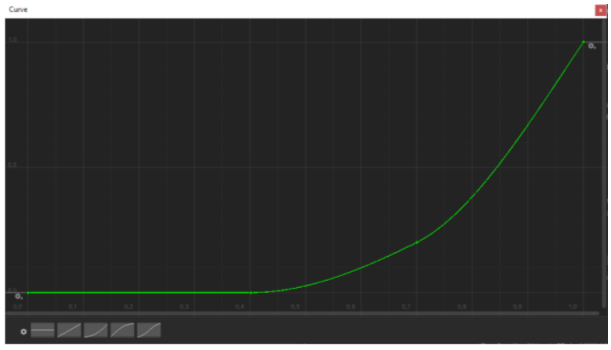
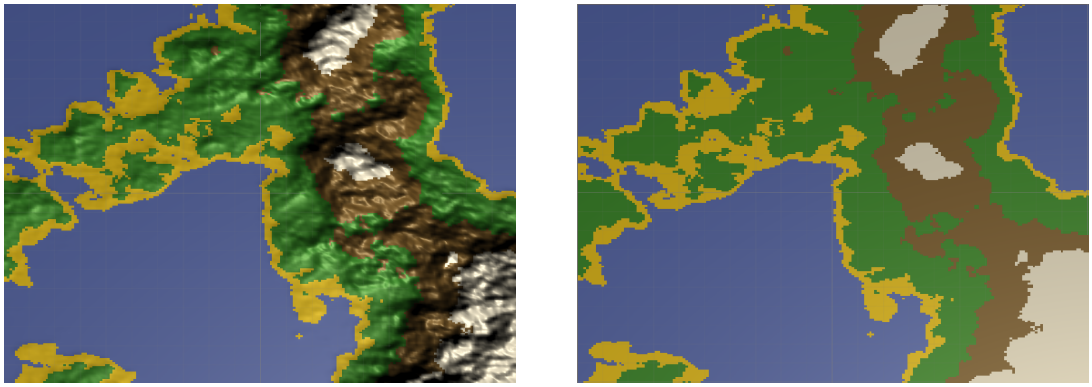


Figura 4.13: Curva utilizada.

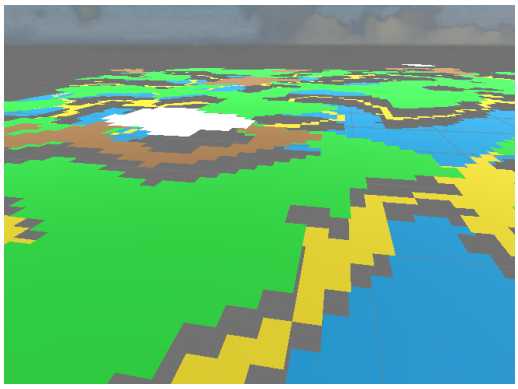
Ahora que ya sabemos generar de manera correcta el ruido, el mapa de colores y el terreno, vamos a alterar el terreno para que todo esté forma por casillas para darle una estética de 'cubos' al mundo. Hemos decidido esto porque si mantuviéramos la estética donde cada vértice pueda tener una altura diferente a los de sus adyacentes los modelos 3D de los animales atravesarían el mundo a la hora de moverse. Sin embargo con un mundo basado en casillas esto no pasaría. En este nuevo mundo, tenemos **cinco terrenos diferentes**. *Agua*, *arena*, *hierba*, *monte* y *nieve* así que las alturas de los vértices podrán ser entre cero y cuatro siempre con números enteros. Por cada fila y columna del mapa tendremos una casilla formada por cuatro vértices que serán los correspondientes al *noroeste*, *noreste*, *suroeste* y *sureste*. De esta manera creamos más vértices de los necesarios porque podríamos aprovechar los creados en una casilla para completar la siguiente, pero esta optimización queda relegada a trabajo futuro ya que con la implementación actual es suficiente.



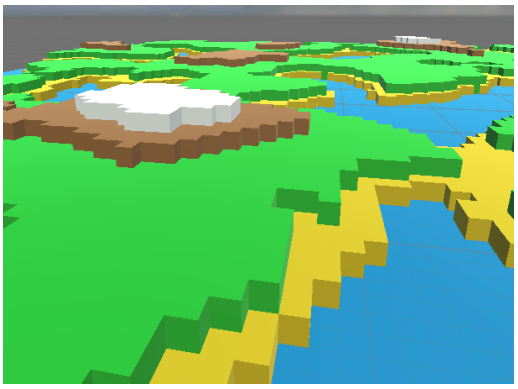
(a) Mundo sin casillas. **(b)** Mundo con casillas.

Figura 4.14: Comparación del mundo con y sin casillas.

Dentro de la implementación con casillas cabe destacar que tenemos que rellenar de alguna manera los huecos que se forman entre casillas adyacentes de diferentes alturas y las casillas en el límite del mapa. Para ello tenemos que detectar si nuestros vecinos (solo los que están arriba, abajo, a la izquierda o a la derecha de nosotros) están a alturas diferentes a la nuestra o si estamos al borde del mapa. Si lo estamos, recorreremos nuestros vecinos y crearemos una nueva casillas para rellenar el hueco en caso de que sea necesario. Esto de por sí es innecesario ya que no afectará al mundo per se pero mejora su aspecto visual significativamente.



(a) Terreno sin rellenar huecos.



(b) Terreno rellenando huecos.

Figura 4.15: Comparación del mundo con y sin rellenar huecos.

Lo último que tenemos que explicar sobre la generación de terreno es que información vamos a tener que almacenar para que el ecosistema pueda funcionar correctamente. Para esto necesitamos pasarle una matriz con los **centros de cada casilla**, otra de booleanos con las **casillas caminables** (aquellas que no sean agua), otra de booleanos indicando si las casillas son **vecinas a agua** y otra de enteros con los **índices del terreno correspondiente** a cada casilla que usaremos para las probabilidades de que aparezcan árboles o hierba en esa casilla.

4.4.2. Boids

Los boids o *"bird-oid objects"* es un programa de inteligencia artificial originalmente desarrollado por *Craig Reynolds* en 1986 y publicado oficialmente en 1987 bajo el título **Flocks, Herds, and Schools: A Distributed Behavioral Model**[12][Reynolds, 1987].

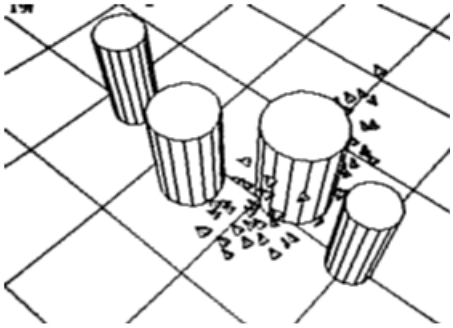


Figura 4.16: Simulación original de una bandada de Boids.

Tal y como comenta Reynolds en el primer apartado de su artículo, el comportamiento de las bandadas de pájaros, las escuelas de peces o las manadas de cualquier otro tipo de animal terrestre es hermoso y complejo. Esa clase de comportamientos puede tener muchas utilidades y existe una rama de la inteligencia artificial que se dedica al estudio exclusivo de esta clase de sistemas. La **inteligencia de enjambre** estudia los comportamientos colectivos de sistemas descentralizados y auto-organizados, naturales o artificiales. Esta clase de comportamientos puede ser visto en bandadas de pájaros, escuelas de peces, colonias de hormigas, crecimiento bacteriológico etc.

Al ser sistemas descentralizados, los componentes del sistema son independientes y actúan de forma individual. Sin embargo a partir de las interacciones entre dichos individuos, puede surgir un comportamiento global complejo o inteligente. Esto es interesante ya que significa que la complejidad del movimiento del conjunto surge como consecuencia de las interacciones de los individuos que la componen sin que estos sean conscientes de ello. Es decir, si cada componente sigue una serie de reglas simples, acaba surgiendo un sistema de movimiento complejo.

Esa es la premisa de este campo de estudio, a partir de un conjunto de reglas simples para cada agente, acaba surgiendo un comportamiento global intrincado a través de las interacciones de los mismos.

Reynolds describe una forma de simular bandadas de pájaros a través de tres reglas sim-

[12]https://team.inria.fr/imagine/files/2014/10/flocks-hers-and-schools.pdf

ples, **alineación, cohesión y separación**. El concepto es simple. Cada agente (boid a partir de ahora) va a moverse en función a este conjunto de normas.

- **Alineación:** El boid intentará dirigirse en la misma dirección general que la manada para no acabar separada del grupo.
- **Cohesión:** El boid intentará moverse hacia el centro de la manada para conseguir mayor seguridad.
- **Separación:** Los boids mantendrán una distancia mínima entre ellos para no chocarse.

Con estas tres normas, Reynolds consigue simulaciones con comportamientos bastante cercanos a su objetivo. Se puede ver una explicación más detallada en su página web[13]. Por supuesto, se pueden añadir más reglas para conseguir comportamientos aún mas completos. A continuación explicaremos cual ha sido nuestra implementación de los Boids y los resultados obtenidos.

Nuestros boids siguen las tres reglas básicas y otras dos, una para **evitar chocarse** contra los límites o obstáculos del mapa y otra para **huir de los depredadores**. El mapa se ha subdividido en regiones cúbicas para optimizar la manera en la que perciben su manada y a los depredadores.
Cabe destacar que los boids tienen tres parámetros básicos. La **velocidad**, **dirección** y **rapidez**. La dirección es el vector que representa hacia donde se dirige, la rapidez como de rápido va y la velocidad es la multiplicación de dirección y rapidez.

$$Velocidad = Direccion * Rapidez$$

4.4.2.1. Reglas básicas

Como hemos explicado antes, existen tres reglas básicas que siguen los boids, alineación, cohesión y separación. Cada boid tiene un radio de percepción y todos los boids que estén dentro de este radio serán considerados su manada. Aplicará las reglas básicas en función de dicha manada.
Cabe destacar que en la manada de un boid **nunca** se incluirá a él mismo.

[13] http://www.red3d.com/cwr/boids/

- **Alineación.** Dada la manada a la que pertenece el boid, calculamos la velocidad general del grupo. El pseudocódigo es simple, recorriendo la manada, sumamos sus velocidades, las dividimos por el número de boids de la manada (sin incluirnos a nosotros), le restamos nuestra posición y multiplicamos por el peso de la regla. Esta regla se puede aplicar tanto con la velocidad como con la dirección, darían resultados similares.

```
Vector3 Alineación(manadaBoids)
    Vector3 pV; //Sumatorio velocidades
    foreach(boid in manadaBoids)
        pV+=boid.velocidad;
    pV/=manadaBoids.Length-1;
    return (pV-me.position)*pesoAlineacion
```

- **Cohesión.** Similar a la alineación pero en lugar de la velocidad, lo hacemos con las posiciones de los boids de la manada para hallar el centro y luego calculamos el vector entre nosotros y el centro.

```
Vector3 Cohesión(manadaBoids)
    Vector3 pC; //Sumatorio posiciones
    foreach(boid in manadaBoids)
        pC+=boid.posición;
    pC/=manadaBoids.Length-1;//Centro de la manada
    return (pC-me.position)*pesoCohesion
```

- **Separación.** Con el fin de que el agente no se choque con compañeros, si un boid vecino está dentro de nuestro *rango de espacio personal*, X, calculamos el vector entre nosotros y él y lo guardamos en negativo para alejarnos de él.

```
Vector3 Separación(manadaBoids)
    Vector3 c; //Sumatorio de vectores de separación
    foreach(boid in manadaBoids)
        if(Distancia(boid.position, me.position) > X) //Si esta demasiado cerca
            c-=boid.position - me.position; //Vector para alejarnos
    return c*pesoSeparacion
```

Habiendo calculado las tres reglas, cada boid tendrá que calcular su nueva velocidad. La suma de las tres normas lo podemos considerar como la aceleración del mismo. El siguiente pseudocódigo sería la función principal de cada agente que estaría ejecutándose en bucle constantemente.

```
void Main()
    manada = PercibirBoidsRegion(region);
    Vector3 r1 = Cohesión(manada);
    Vector3 r2 = Separación(manada);
    Vector3 r3 = Alineación(manada);

    Vector3 aceleracion = r1+r2+r3;
    velocidad+=aceleracion;
    rapidez = Magnitud(velocidad);
    direccion = velocidad/rapidez;
    velocidad = dirección*rapidez;

    //Actualizamos los ejes del boid
    ActualizarEjesBoid();
    me.position += velocidad;
```

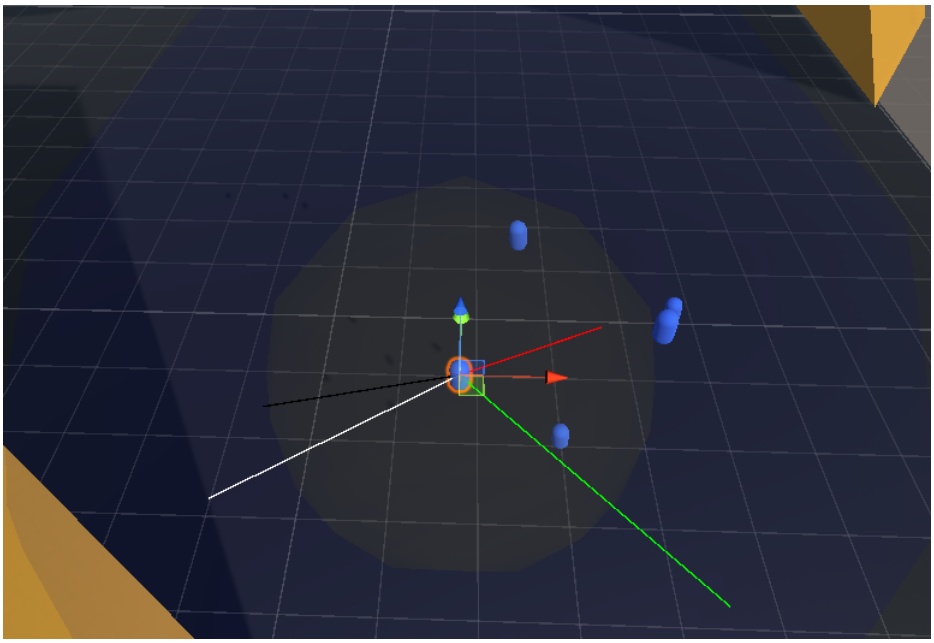


Figura 4.17: Reglas básicas. Rojo: Centro manada. Verde: Dirección manada. Blanco: Separación. Negro: Nuestra dirección.

El encargado de administrar el movimiento y las reglas del boid es el script **Boid.cs**.

4.4.2.2. **Evitar obstáculos**

Los boids tienen que evitar chocarse con los obstáculo del mapa, pero, **¿Como sabemos cuando se van a chocar?** y **¿En que dirección debemos ir para evitarlo?**. En este apar-

tado responderemos a estas dos preguntas.

- **Detectar choque**

Para detectar la posible colisión usamos el paquete de físicas de Unity incluido en la biblioteca llamada *UnityEngine* con el que podemos castear un rayo que nos devolverá la información del mismo.

Para ser más específicos, vamos a proyectar una pequeña esfera desde un origen en una dirección concreta, con una distancia de recorrido máxima y que solo podrá colisionar con objetos que tengan una máscara concreta. La función se llama ***SphereCast*** y pertenece al paquete ***Physics*** de Unity. Esta función tiene hasta 6 argumentos: *origen del rayo*, *radio de la esfera a proyectar*, *dirección, hit* (donde nos van a devolver la información del rayo), *distancia máxima del rayo* y la *máscara de obstáculos*. El origen del rayo sera la posición del boid, el radio de la esfera cualquier valor pequeño, (en nuestro caso hemos usado 0.2 pero cualquier valor bajo sería válido) la dirección es la misma que la del boid, el hit es una variable RaycastHit donde nos devuelven la información, la máxima distancia de la proyección es el radio de visión del agente y la máscara de obstáculos es una máscara llamada *Paredes* que se ha asignado a los objetos del mundo que queremos evitar.

Una vez obtenemos la información, si vamos a colisionar, buscaremos una nueva dirección para evitar el obstáculo. Sino seguiremos nuestro camino. El boid realiza esta comprobación de choques constantemente.

- **Evitar choque**

Una vez sabemos que vamos a colisionar, vamos a tener que encontrar una nueva dirección en la que no choquemos. Esto se puede hacer proyectando rayos a partir de nuestra posición y cambiando nuestra dirección en función del mismo. Esta es la segunda regla adicional de nuestros boids.
Vamos a tener que proyectar rayos en una esfera cuyo origen es nuestra posición. Se podría hacer de varias formas pero queremos una forma eficiente y a ser posible que estos

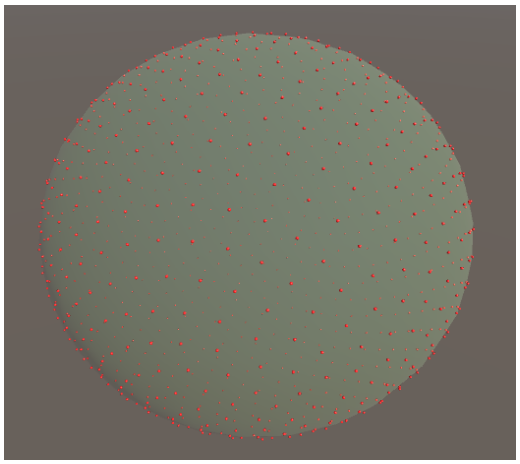
rayos estén distribuidos de manera uniforme. Esto nos conduce a un problema muy conocido y estudiado, **Como distribuir puntos en una esfera de manera uniforme**.

Uno de los métodos más eficientes para este problema es usar espirales generadas con el números de Fibonacci que da un resultado bastante bueno. Dado el numero de puntos que queremos distribuir de manera uniforme en una esfera **N**, vamos a devolver las posiciones de los puntos uniformemente distribuidos. Con estos puntos podremos sacar direcciones a partir de las cuales proyectar rayos. Cuando uno de estos rayos no colisione, seguiremos esa dirección.

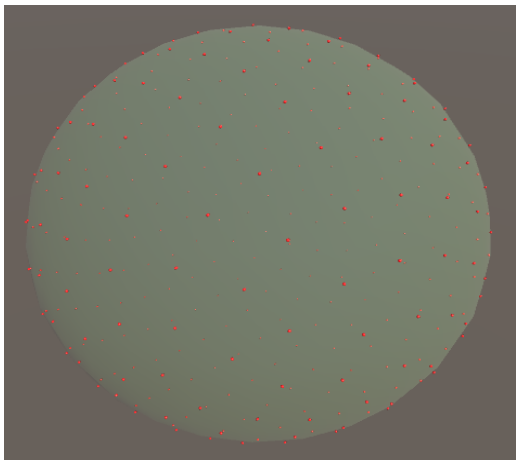
Lo primero que hacemos es calcular los **N** puntos generados en una esfera con nosotros como origen a través de las espirales de Fibonacci. El pseudocódigo sería el siguiente:

```
1 Vector3[] EsferaFibonacci(int n)
2     Vector3[n] puntos;
3     numeroAureo = (1+Raiz(5))/2;
4     for(i=0;i<=n;i++)
5         theta = 2*PI*(i/numeroAureo);
6         phi = Arcos(1-2*(i+0.5)/n);
7         puntos[i] = (Cos(theta)*Sen(phi), Sen(theta)*Sen(phi), Cos(phi));
8     return puntos;
```

Con esto, obtenemos **N** puntos distribuidos de manera uniforme en una esfera. Ahora tenemos que sacar las posibles direcciones que simplemente serán el vector entre nuestra posición y la del punto en la esfera y la primera que no colisione con un obstáculo será nuestra nueva dirección. Cabe destacar que estos puntos tendremos que pasarlos al espacio local del boid ya que de lo contrario los vectores apuntarán al origen porque estos puntos uniformemente distribuidos son desde el origen (0,0).



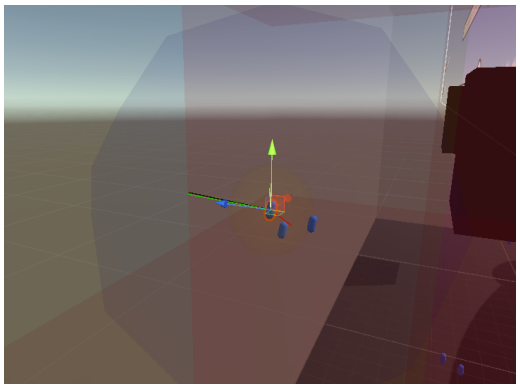
(a) 1000 puntos uniformemente generados en una esfera.



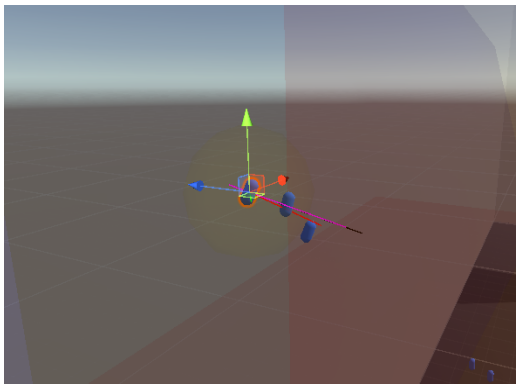
(b) 300 puntos uniformemente generados en una esfera.

Figura 4.18: Puntos uniformemente generados en una esfera con espirales de Fibonacci.

Las imágenes superiores han sido tomadas a partir de una demo creada por nosotros en Unity cuyo código es público en el siguiente enlace: Espirales Fibonacci[14].
En el siguiente ejemplo vamos a mostrar como los boid al detectar una colisión escogerán una nueva dirección para evitar dicho choque.



(a) Un boid a punto de colisionar.



(b) Nueva dirección (magenta) para evitar la colisión.

Figura 4.19: Boid evadiendo obstáculo.

[14]https://github.com/DripyDev/EspiralesFibonacci

4.4.2.3. **Subdivisión del mapa**

A la hora de calcular la manada (boids dentro del radio de percepción) a la que pertenece cada boid, tenemos que recorrer todos los existentes y ver cuales de ellos están lo suficientemente cerca como para formar una manada. Esa forma de calcular las manadas es de orden **O(n*n)** lo cual es un poco ineficiente y se puede mejorar en cierta medida. No hace falta recorrer todos los agentes porque algunos estarán claramente demasiado lejos, pero no sabemos como de lejos hasta calcular la distancia.
Esto puede ser evitado si subdividimos el mapa en regiones más pequeñas. De esta forma solo hará falta recorrer los boids en las regiones adyacentes a la nuestra para ver si forman parte de nuestra manada. En el peor de los casos (si todos los boids están en la misma región o adyacentes) la eficiencia será la misma, pero en el resto de los casos será más rápido.
Lo que vamos a hacer, es tener un script, *RegionManager.cs*, que será el encargado de dividir el mundo en regiones y controlar en que región se encuentra cada agente. Una vez dividido el mundo, los boids sabiendo el índice de la región en la que se encuentran solo tendrán que buscar vecinos en las regiones adyacentes. Cuando comprueban en que región se encuentran solo hará falta comprobar las regiones adyacentes a la última región en la que estaba.
Vamos a subdividir el mapa en regiones. Para ello necesitamos saber dos de los tres siguientes parámetros: *Nº regiones*, *dimensiones mapa* y *dimensiones región*. En nuestro caso vamos a suponer que el mapa y las regiones son cubos. A partir de la siguiente fórmula se pueden sacar cualquiera de los tres parámetros.

$$NRegiones = \frac{DimensionMapa^3}{DimensionCubo^3}$$

Una vez tenemos estos datos, vamos a tener que generar los cubos en orden para luego poder encontrar regiones adyacentes a través de índices en lugar de buscándolos. En nuestro caso, las regiones se generan en Z-X-Y. Es decir, tres bucles anidados uno para recorrer cada eje. Supongamos que vamos a generar un mapa 3x3x3, 27 regiones y *Z=Profundidad*, *X=Laterales* e *Y=Altura*. El pseudocódigo sería el siguiente:

```
1 void InicializarMapa()
2     float dRegion; //Dimension de las regiones
3     float rCubo; //Raiz cubica del ºN de regiones
4     Vector3 posX,posY,posZ; //Posiciones del cubo
```

```
5
6       for(y=0;y<rCubo; y++)
7           posX=posY;
8           posY.y += dRegion;
9           for(x=0;x<rCubo; x++)
10              posZ=posX;
11              posX.x += dRegion;
12              for(z=0;z<rCubo; z++)
13                  CrearRegion(posZ, dCubo);
14                  posZ.z+=dRegion;
```

Que generaría como resultado un mapa de regiones que desglosado sería así:

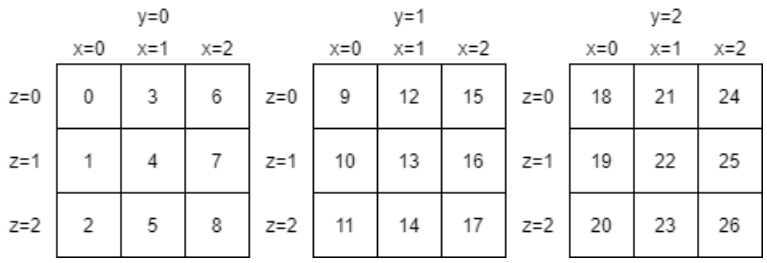


Figura 4.20: Cubo 3x3x3 desglosado.

El mapa de regiones es una lista de una estructura llamada *Region* que contiene la posición del **centro del cubo**, las **dimensiones**, una **lista con los boids dentro de la misma** y otra con los **depredadores**.

Una vez tenemos esto, al inicializar los boids tendremos que encontrar la región en la que se encuentran inicialmente y sus adyacentes. Tenemos que tener en cuenta que no todos los cubos tienen regiones adyacentes, el último piso del mapa no va a tener regiones superiores, por ejemplo. Entonces tenemos que encontrar una manera de saber cuando y que regiones vecinas vamos a tener.
Por fortuna, al haber formado las regiones de forma ordenada, podemos saberlo a partir de operaciones simples. Por ejemplo, para saber si tenemos una región superior, basta con sumar al índice de la región actual 9 (3x3, numero de cubos por piso) y si el resultado es mayor a 27 (número total de cubos) sabemos que no tenemos cubos por encima. Hallamos si tenemos cubos encima, debajo y a los laterales y en función a ello podemos saber si

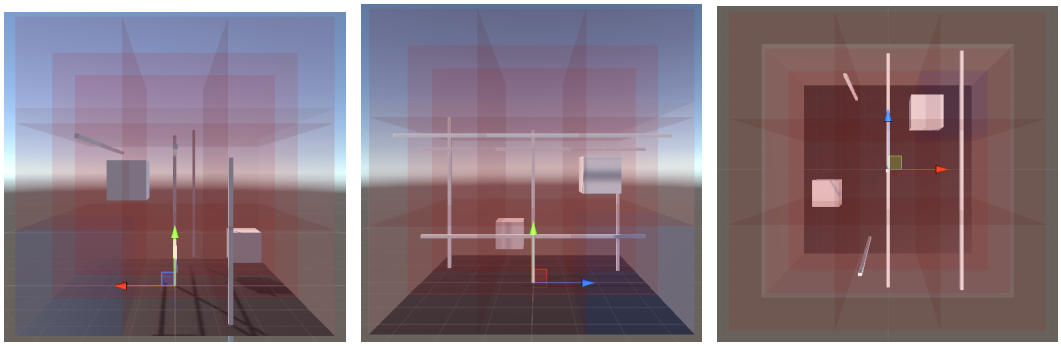
tenemos o no el resto de cubos adyacentes.

$$Arriba : X+9 < 27 -> X+9 | Abajo : X-9 >= 27 -> X-9$$

$$Delante : Xmod3 == 0 -> X+1 | Atras : (X-2)mod3 == 0 -> X-1$$

$$Derecha : X-(int)(X/9)*9 <= 9-3-1 -> X+3 | Izquierda : X-(int)(X/9)*9 >= 3 -> X-3$$

Una vez obtenido estos índices, el resto de regiones adyacentes no son más que combinaciones de estas. Cabe destacar que cuantas más regiones tengamos, más rápido puede llegar a ser ya que con pocas regiones hay más posibilidades de que los boids se encuentren en zonas adyacentes por lo que la eficiencia no mejoraría mucho.



(a) Vista frontal **(b)** Vista lateral **(c)** Vista superior

Figura 4.21: Vistas mapa regiones 3x3x3

4.4.2.4. **Depredadores**

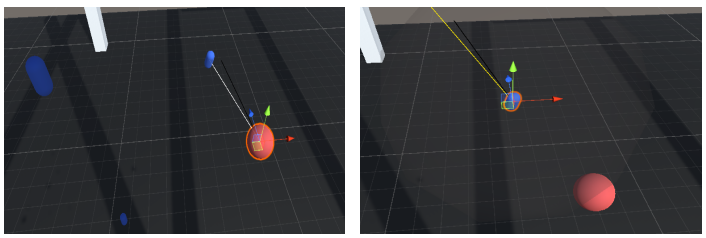
Una vez implementados los boids con las reglas básicas, podemos experimentar con ellos y añadirles más reglas. El objetivo del proyecto es simular un ecosistema así que sería interesante analizar el comportamiento de los boids si incluimos depredadores de los que tengan que huir.

Los depredadores serán controlados por su propio script *Depredadores.cs* y tendrán sus propios settings ya que no van a seguir las mismas normas que el resto de agentes del sistema. Los depredadores solo siguen dos reglas, una para evitar chocarse con obstáculos y otra para perseguir a las presas. A su vez, a las presas se les ha añadido una regla extra para huir. Para ello el **Region Manager** va a tener que controlar también en que zonas se encuentran los depredadores. Esto supone un pequeño cambio en la estructura de la región en la que vamos a añadir una lista de depredadores que indica el número de estos por zona. Controlar y administrarlo es igual a como lo hacemos con los boids.

La regla de ataque es simple, al igual que los boids, los depredadores tendrán un radio de percepción (mayor que el de los boids, al igual que la velocidad que será un poco mayor) y perseguirán a la presa más cercana a ellos.

```
1  Vector3 ReglaAtaque(listaPresas)
2      presaMasCercana = listaPresas[0];
3      distanciaAux = Distancia(me, presaMasCercana);
4      foreach(boid in listaPresas)
5          if(Distancia(me, boid) < distanciaAux)
6              presaMasCercana = boid;
7              distanciaAux = Distancia(me, boid);
8      return (presaMasCercana - me.position)*pesoAtaque;
```

Si el depredador consigue acercarse lo suficiente a la presa, esta morirá.



(a) Depredador persiguiendo al boid más cercano.

(b) Boid huyendo del depredador.

Figura 4.22: Persecución y huida de los agentes.

4.4.2.5. **Uso de la GPU**

Unity usa HLSL como lenguaje para los shaders. Nuestra implementación de los Boids es eficiente pero a la hora de simular varios cientos de agentes el rendimiento baja considerablemente ya que la CPU es el único encargado de procesar toda la información. Sin embargo, si pudiéramos usar la GPU y su capacidad de usar hilos, podríamos simular una mayor cantidad de agentes y eso es lo que hemos hecho.

Usando shaders hemos dado un hilo a cada agente para que realice los cálculos. El primer paso que hemos realizado para esto es cambiar ligeramente la disposición y estructura general del sistema. La reserva y uso de los threads (hilos) tiene que ser realizada en un script propio. Por eso hemos modificado el script llamado **Spawner.cs** que era el encargado de hacer aparecer a los agentes. Ahora su cometido es crear a los agentes y preparar y administrar el uso de la GPU en caso de que se quiera usar.

El primer paso para usar la GPU es pensar que datos vamos a tener que guardar en un buffer para que el shader pueda leerlo y procesar la información necesaria. Habrá **dos shaders**, uno para los **boids** y otro para los **depredadores**. La información a procesar es simple. Para cada agente necesitamos su *posición*, *dirección*, *velocidad*, *reglas* y *numero de boids en la manada*. Esos son todos los datos necesarios, la posición, dirección y velocidad para calcular las reglas y el numero de boids en la manada. Por supuesto, también necesitamos saber el número de boids y depredadores totales, el radio de visión y la distancia de separación. Pero esto es información común a todos los agentes.

Sabiendo la información necesaria para cada boid, creamos dentro de **Spawner.cs** una estructura (llamada **DatosBoid**) que albergue todos estos datos. Cabe destacar que, como vamos a guardar una estructura por boid en un buffer para que dicha información se pueda leer en un shader, vamos a tener que especificar el tamaño exacto de la estructura. Esto es una norma para HLSL ya que necesita saber el tamaño de la estructura para poder almacenarlo para su uso. En nuestro caso, tenemos dos números enteros y siete vectores cada uno compuesto por tres número de punto flotante por cada *DatosBoid*.

Para preparar los datos, tenemos dentro de **Spawner.cs** una función llamada **Establecer-**

DatosShaderBoid() que se encargará de recorrer todos los agentes (tanto boids como depredadores) para preparar sus correspondientes estructuras. Luego creará un buffer señalando el número de estructuras y su tamaño.

Es importante no olvidar que dentro del propio shader hace falta crear una estructura paralela a **DatosBoid** (que hemos llamado *'Boid'*) y que habrá que crear una variable del tipo **XXStructuredBuffer<Boid>** ('XX' representa si queremos escribir, leer o ambas. W=Write, R=Read. En nuestro caso necesitamos lectura y escritura) donde se guardará la lista con la información de cada agente.

Una vez hemos establecido toda la información que el shader necesita para funcionar, vamos a tener que asignarle un número de hilos a usar. A la hora de asignar hilos, se puede realizar de varias maneras diferentes (usando las entradas de los kernels etc). Para ejecutar el shader, tenemos que usar la función **Dispatch(kernel, num grupX, num grupY, num grupZ)**. Como nuestro objetivo no era profundizar en el uso de los ComputeShaders, hemos asignado el kernel cero con un grupo para cada dimensión. Dentro del shader, vamos a usar 1024 hilos.

El ciclo de vida es el siguiente:

- Preparar estructuras DatosBoid.
- Establecer los buffers.
- Guardar la información necesaria para el shader (rango visión, número de boids y depredadores y distancia de separación).
- Ejecutar el shader.
- Recoger la información procesada.
- Liberar los buffers.

Una vez ha terminado el ciclo, con la información actualizada de nuestra lista de **DatosBoid**, recorremos todas las presas y depredadores del mundo, actualizamos sus datos y procesamos sus próximos movimientos. Hay que resaltar que **Spawner.cs** calculará las

reglas básicas y la manada pero tanto el boid como el depredador calcularán la regla correspondiente a evitar la colisión por su cuenta. Las reglas básicas las calculamos al igual que antes.

Una vez hecho esto, solo falta crear un booleano para saber cuando queremos usar la GPU y cuando no.

4.4.2.6. **Resultados**

Al tener acceso a los recursos de una tarjeta gráfica, aunque la eficiencia sea menor, la diferencia es sustancial y el rendimiento general mucho mayor. Los shaders no usan la subdivisión de regiones que usamos de manera normal ya que no se pueden usar listas dinámicas en estos. Es posible que se pueda implementar de alguna manera pero tras intentarlo y no conseguirlo, hemos decidido no profundiza tanto en el uso de shaders porque no era nuestro objetivo.

Sin embargo, los resultados obtenidos han sido más que satisfactorios. Por supuesto, los componentes de cada ordenador afectan de manera significativa al rendimiento general. En nuestro caso, con una CPU *Intel Core i5-4440* y una tarjeta gráfica *Radeon RX 580 Series* el sistema soporta fácilmente unos 250 agentes con CPU y subdivisión del mapa (siempre y cuando no se encuentren todos los agentes en la misma región o adyacentes) y unos 1500 si usamos shaders para los cálculos.

Es posible que pudiéramos conseguir una mayor eficiencia (aunque no es nuestro objetivo) si depuráramos el código eliminando cálculos innecesarios y optimizando los necesarios, pero el resultado obtenido es satisfactorio y el rendimiento con el uso de shaders es más que suficiente para el uso que se le va a dar.

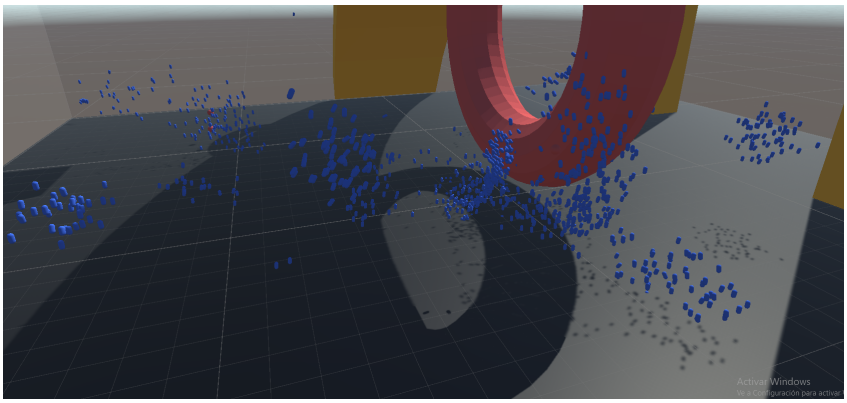


Figura 4.23: 1500 boids con uso de GPU.

4.4.3. Ecosistema

La simulación del ecosistema puede ser principalmente dividido en dos grandes apartados, **el mundo** y **los agentes** y en un tercer apartado más pequeño que serían los **Menús**. El *Mundo* a su vez está dividido en **tres** apartados principales, **Mundo**, **Mapa** y **Pathfinder** y los *Agentes* en **LivingEntity**, **Animal**, **Plant** y **Rabbit - Fox**.

El *Mundo*, es el encargado de **hacer aparecer la población inicial**, **registrar los eventos** como los movimientos o las muertes, **administrar lo que perciben los agentes** y sus movimientos. Por otro lado, los *Agentes* se encargarán de definir y controlar las acciones y interacciones entre los entes. A continuación explicaremos en mayor detalle los apartados que acabamos de describir. Los menús muestran información relevante en pantalla y al terminar la aplicación crean archivos con dicha información para su posterior análisis.

4.4.3.1. **Mundo**

- **Mundo**

El script **Mundo.cs** es el principal que **crea los datos necesarios** para el correcto funcionamiento del sistema, **registra los eventos**, crea las funciones necesarias para que los agentes puedan **percibir su entorno**, el **movimiento aleatorio de la exploración** de los entes, hacer **aparecer la población inicial, árboles y nuevos agentes creados por reproducción** y **crear archivos con la información registrada** como las causas de muerte

de los animales o la velocidad o rango de visión medios etc.

La percepción de los agentes se divide en cuatro partes: **alimento, agua, parejas y depredadores**. Para **percibir el agua**, podríamos recorrer el mapa entero en busca de la casilla con agua más cercana a nuestra posición actual, pero entonces tendríamos que estar buscando constantemente y aunque podríamos optimizar la búsqueda, por ejemplo, buscando solo en las casillas adyacentes dentro de nuestro rango de visión, esto realentizaría el sistema. En lugar de buscar constantemente, podemos realizar una única búsqueda para encontrar el agua más cercano a cada casilla del mundo antes de inicializar el sistema, de esta manera nos ahorraremos la búsqueda constante. Vamos a preprocesar los datos del agua, para ello vamos a recorrer el mapa entero y encontrar la casilla de agua más cercana, será menos eficiente que una búsqueda puntual en el mundo pero a la larga será más rentable. Por ello tenemos una matriz de coordenadas **aguaMasCercana** que representa la posición del agua más cercano dentro de esa casilla en el mundo. Obtener esta variable es simple:

```
Vector3[,] AguasCercanasMundo(Vector3[,] mundo)
    aguaMasCercana = Vector3[tamaño mundo, tamaño mundo];

    for(x=0; x<tamaño mundo; x++)//Recorremos el mundo
        for(y=0; y<tamaño mundo; y++)
            if( mundo[x,y] es caminable)
                for(i=0; i<casillasVisibles.Count; i++)//Casillas dentro de rango de vision
                    int nuevaX = x + casillasVisibles[i].x;
                    int nuevaY = y + casillasVisibles[i].y;
                    if(nuevaX && nuevaY dentro de rango del mundo)
                        aguaMasCercana[x,y] = Coordenada(nuevaX, 0, nuevaY);
                        break;//Agua encontrada pasamos a la siguiente iteracion
            else
                aguaMasCercana[x,y] = Invalido;

    return aguaMasCercana;
```

Vamos a resaltar que *CasillasVisibles* es un array con rango entre **-RangoVision** y **RangoVision** (sin incluir el 0) con las posiciones de las casillas adyacentes a nosotros dentro de nuestro rango de visión a partir de la posición en la que estemos y ordenada por distancia para aumentar la rapidez ya que de esta manera podrá encontrar antes el agua y pasar a la siguiente iteración.

Ahora que tenemos esta información, cuando un agente quiera encontrar agua bastará con

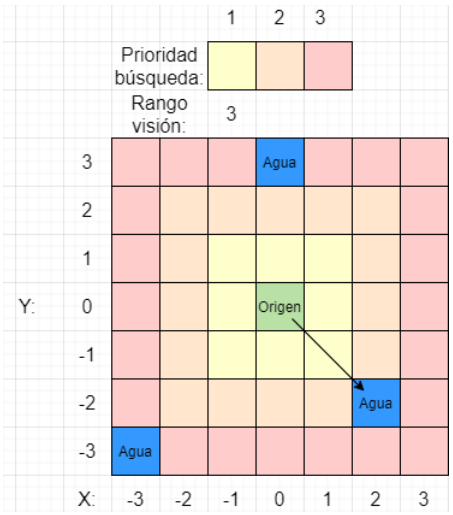


Figura 4.24: Diagrama de búsqueda de casillas de agua.

que nos pase su posición en el mundo y su rango de visión. Si la casilla en la que se encuentra tiene agua cerca y la distancia es menor o igual al rango de percepción del agente, se dirigirá hacia ella, sino no se le devolverá una coordenada inválida y tendrá que seguir explorando.

Para **percibir alimento**, primero tenemos que tener claro que clase de alimento busca el agente. Para ello tenemos una estructura de datos bastante útil que son los **diccionarios**. Un diccionario es una estructura de datos que puede contener llaves y valores de datos y nos deja almacenar cualquier clase de información y de la manera que queramos. Tenemos dos diccionarios diferentes, uno que almacena las presas en función del depredador (**presasDepredador**) y su contra-parte que almacena los depredadores en función de las presas (**depredadoresPresa**). De momento la cadena alimenticia es simple, los zorros son depredadores de los conejos y los conejos de las plantas. Teniendo en cuenta estos diccionarios y sabiendo que tenemos un mapa por especie con las posiciones de estos en el mundo, basta con obtener las entidades que sean presas nuestras en el mundo dentro de nuestro rango de visión y devolver la más cercana a nosotros.

La **percepción de parejas y depredadores** es similar a la de alimentos. Las parejas solo incluirán a todos los agentes de nuestra misma especie pero de sexo opuesto (que también estén buscando pareja).

La **percepción de depredadores** será igual que con los alimentos pero al revés ya que tenemos que huir de ellos.

La exploración de los agentes se hace de manera pseudoaleatoria. Los animales siempre tenderán avanzar al frente pero en caso de que no puedan, simplemente se dirigirán a una casilla aleatoria adyacente a la suya (siempre y cuando sea una casilla caminable). Para conocer la dirección hacia la que nos dirigimos necesitamos como mínimo dos coordenadas, por ello los agentes tienen una referencia a su posición actual y otra a la última en la que han estado. Con esta información (sabiendo que las dos coordenadas siempre serán adyacentes) tenemos una probabilidad de, en nuestro caso, un 40 % de simplemente avanzar hacia adelante, sino avanzaremos en una de las tres direcciones adyacentes a nuestra dirección actual. En caso de que ninguna de las tres nuevas direcciones sean válidas o que nuestra dirección sea nula porque hemos estado durante más de una iteración en la misma casilla, simplemente devolveremos una casilla caminable adyacente aleatoria sin tener en cuenta en la que ya nos encontramos.

Todos los **spawn de entes** se hacen en **Mundo.cs**. Dentro de estos podemos destacar cuatro funciones especiales para ello, una para la **población inicial**, otra para **agentes nacidos a través de la reproducción sexual**, otro para **las plantas** y otro para **los árboles**. Para **las plantas** simplemente buscamos una posición aleatoria de entre las que se encuentren libres en el mundo y cada X tiempo (0.5 segundos) haremos aparecer una planta en el mundo. De esta función cabe destacar que a la hora de generar un número aleatorio para buscar la coordenada, necesitamos una semilla diferente cada vez para el generador de números por lo que usamos los milisegundos de la hora actual que será diferente en cada iteración de esta función. En caso de que usáramos la misma semilla para todas las iteraciones siempre se generaría el mismo número aleatorio.

Los árboles son meramente decorativos de momento. Tenemos un parámetro público modificable dentro de **Mundo.cs**, **probabilidadArboles** (un número entre 0 y 1), que representa la probabilidad de que aparezcan los árboles. Para esta función basta con recorrer el mundo y en las casillas caminables (únicamente aquellas sin agua o donde ya haya un árbol) si un número aleatorio supera el la probabilidad de que aparezcan árboles, instanciaremos un árbol en esa posición y haremos dicha casilla incaminable. Para añadir

un poco de variabilidad a los árboles, su tamaño también cambia ligeramente de manera aleatoria.

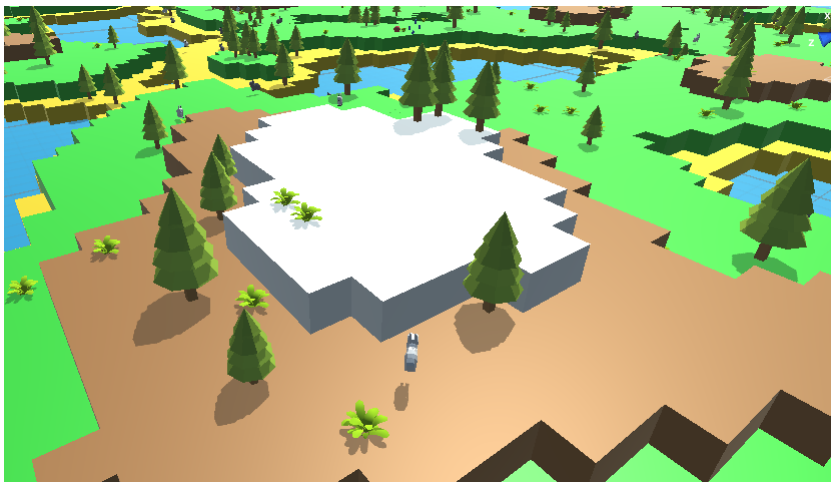


Figura 4.25: Árboles en el mundo.

La población inicial está almacenada en una estructura llamada **Poblacion** que contiene el *LivingEntity* y el número de entidades. Con la etiqueta **[System.Serializable]** podemos hacer que la estructura sea serializable por lo que podremos editarla dentro del inspector de Unity. Nuestra población inicial está compuesta de plantas, conejos y zorros. Simplemente recorremos la lista de *Poblacion* que tenemos en una variable llamada **poblacionInicial** y hacemos aparecer los agentes. Esta primera generación nacen todos en plena adultez, con valores aleatorios entre 0 y 0.4 en hambre, sed y instinto reproductivo y todos sus genes son aleatorios.

La reproducción sexual de los agentes tiene como resultado crías que heredarán de sus padres los genes. Representamos los genes como un array de bits para ahorrar memoria y si ambos padres comparten un gen, el hijo lo heredará teniendo en cuenta siempre la pequeña probabilidad de mutación de que el gen se active o desactive. En caso de que únicamente uno de los padres tenga el gen activo, el hijo tendrá un 50 % de probabilidades de heredarlo sin tener en cuenta el factor de mutación que es del 1 %. Los efectos de cada gen son controlados cuando el agente es inicializado y tienen diferentes efectos. Hasta ahora, contemplamos **seis genes** diferentes que son **mayor velocidad, menor tiempo de embarazo (solo para hembras), mayor rango de visión, mayor instinto reproductivo, crecimiento acelerado y deseabilidad (solo para machos)**. También tenemos el gen

masculino que simplemente indica si nacerá macho o hembra, hay un 50 % de posibilidades de que nazca macho sin probabilidad de mutación.

Padre:	1	0	1	0	0	1
Madre:	0	1	1	0	1	1
Hijo:	50%	50%	1	1%	50%	1

Figura 4.26: Heredado de genes.

El registro de eventos registra los movimientos y muertes de las entidades vivas. Cuando un agente termina de moverse de una casilla a otra el correspondiente mapa de entidades de su especie es actualizado como hemos explicado en el apartado *Mapa*. El registro de las muertes también actualizará el correspondiente mapa de la especie pero también actualizará un registro que tenemos para controlar los agentes existentes en el mundo. El encargado de este registro es un script llamado **AdministradorCausasMuerte.cs** que hemos explicado en el apartado *Menus*. Este script forma parte de la interfaz de usuario de la aplicación y es donde se almacena la información del número de entidades de cada especie vivas en cada momento y sus causas de muerte. Dentro de **Mundo.cs** tenemos una función que al terminar la simulación generará un texto con todas las causas de muertes de los entes a lo largo de dicha simulación.

La creación de archivos dentro de **Mundo.cs** únicamente es un archivo con las causas de muerte de los animales (no tenemos en cuenta las plantas). El control de las muertes y sus causas se hace cuando un agente es eliminado del mundo. El formato del archivo TXT generado es el siguiente:

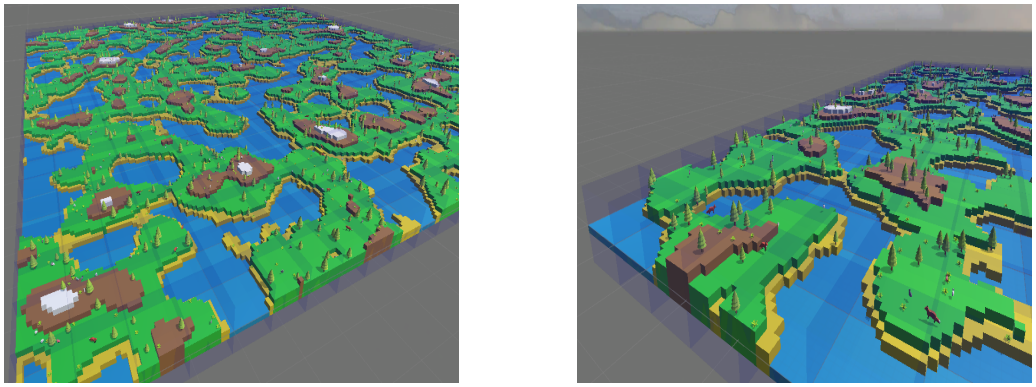
```
Causas muerte
Conejos     Zorros
Eaten: 212  Eaten: 0
Hunger: 243 Hunger: 27
Thirst: 22  Thirst: 0
Age: 0  Age: 0
```

Todos los archivos son almacenado en la carpeta llamada **Logs**.

- **Mapa**

El mundo está subdividido en regiones más pequeñas para optimizar la búsqueda de entes, como hemos hecho con los boids. El script **Mapa.cs** es el encargado de esto. Este script contiene una clase con el mismo nombre y cinco variables principales: **dos matrices**, una con los *centros de las regiones* y la otra con *listas de animales* que representan el número de entidades en cada región. Las otras tres variables son enteros, uno para definir el *tamaño de las regiones*, otro para el *número total de regiones* en las que vamos a subdividir el mapa y otro para especificar el *número total de entidades en el mundo*. Existirá una instancia de la clase *Mapa* para cada especie de animal en el mundo.

Dentro de la clase, tenemos una serie de funciones auxiliares que, como hemos mencionado antes, se encargarán de registrar los movimientos de entes entre diferentes regiones y de devolver los agentes dentro del rango de visión de uno para que puedan percibirse entre ellos.



(a) Imagen 1 **(b)** Imagen 2

Figura 4.27: Subdivisión de regiones en el mapa.

Para registrar los movimientos simplemente necesitamos **tres funciones**. Una para **añadir una entidad** en una región, otra para **eliminarla** y otra para **moverla** que solo usará las dos funciones anteriores. **Añadir(entidad, posicion)** necesita encontrar la región del mapa en la que se encuentra que sería la posición del cuadrante en el que está, dividido entre el tamaño de las regiones y luego actualiza el número de entidades de esa región. En **Eliminar(entidad, posicion)** solo cabe destacar que al tener los entes una referencia al índice en el que se encuentran dentro de la lista de entidades de la región, podemos eliminarlo directamente de la lista sin tener que buscarnos dentro de la misma, de esta manera ganamos un poco de eficiencia. **Mover(entidad, origen, destino)** es sencillo, *eliminamos*

la entidad de la región en la que se encuentra y la *añadimos* en la nueva. Resaltamos que como se va a llamar a esta función cada vez que un agente termina de moverse, vamos a llamar a **Eliminar** y **Añadir** solo cuando realmente vayamos a cambiar de región.

El resto de funciones auxiliares son **EntidadMasCercana(origen, distanciaVision)**, **ObtenerEntidades(origen, distanciaVision)** y **RegionesVisibles(origen, distanciaVision)**. La primera devolverá el agente más cercano a nosotros, usará la función **ObtenerEntidades(origen, distanciaVision)**, ordenará la lista en función de la distancia a nosotros y devolverá el primer elemento. **ObtenerEntidades(origen, distanciaVision)** devolverá una lista con todos los agentes a la vista, usará **RegionesVisibles(origen, distanciaVision)** para obtener las regiones visibles, las recorrerá y guardará aquellas entidades que estén dentro del rango de visión (no hay que olvidar que cada región contiene una lista con las entidades presentes en ella). Por último, **RegionesVisibles(origen, distanciaVision)** devolverá los índices de las regiones que podemos llegar a ver.

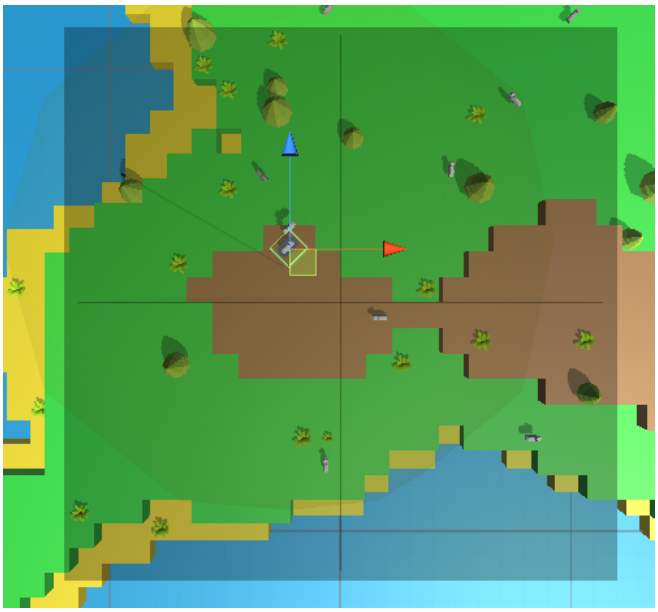


Figura 4.28: Regiones visibles para el conejo.

- **Pathfinder**

El **Pathfinding** o **Búsqueda de ruta** son una serie de algoritmos cuyo objetivo es conseguir *el camino más corto entre dos puntos*. Este es un problema muy popular y muy desarrollado en la ciencia computacional con muchas variantes y soluciones diferentes entre las cuales se encuentran los algoritmos de **Dijkstra**, **Bellman Ford**, **Floyd Warshal** etc etc.

Para controlar el camino que debe de tomar un agentes desde un punto A hasta otro punto B, vamos a necesitar un algoritmo que se encargue de devolvernos la ruta óptima entre esos dos puntos. En un mundo bidimensional como el nuestro, el camino más corto entre dos puntos es la **línea recta**. Pero esto nos conduce a otro problema conocido, el ***trazado de líneas rectas*** sobre un sistema mapeado. Es fácil trazar una línea recta en el mundo real, basta con usar una regla y un lápiz pero este problema se complica más cuando nuestro sistema está mapeado. Un claro ejemplo son las pantallas o monitores de ordenador. Estas pantallas están formadas por cientos o miles de píxeles, es un mapa bidimensional donde cada píxel tiene una posición (x,y) en la pantalla. En este sistema no es tan sencillo dibujar una linea recta porque la menor unidad de color para formar la imagen es cuadrada.

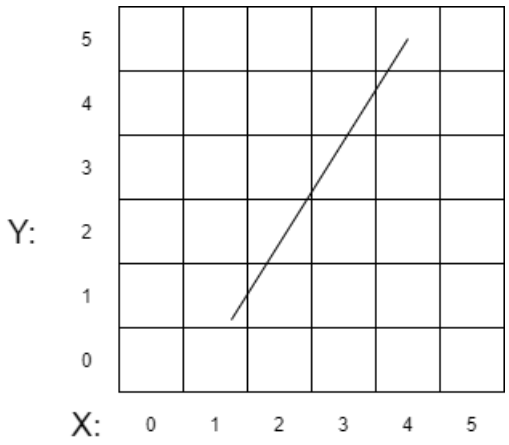


Figura 4.29: Línea recta en un sistema mapeado.

Teniendo en cuenta la imagen que acabamos de exponer, ¿Como dibujamos esa línea recta?. Podríamos intentar aproximar la recta quizás coloreando los cuadrantes diagonales pero esto sería una mala solución ya que la recta pasa por varios cuadrantes que quedarían fuera si aplicáramos esa solución.

Lo correcto sería dibujar todos los cuadrantes sobre los que pasa la recta aunque esto pueda dar como resultado una línea extraña y relativamente poco recta.

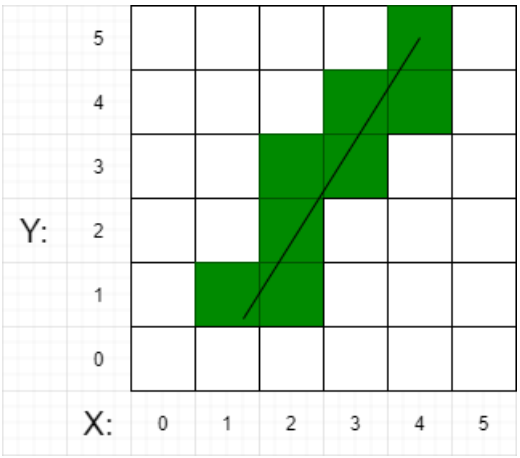
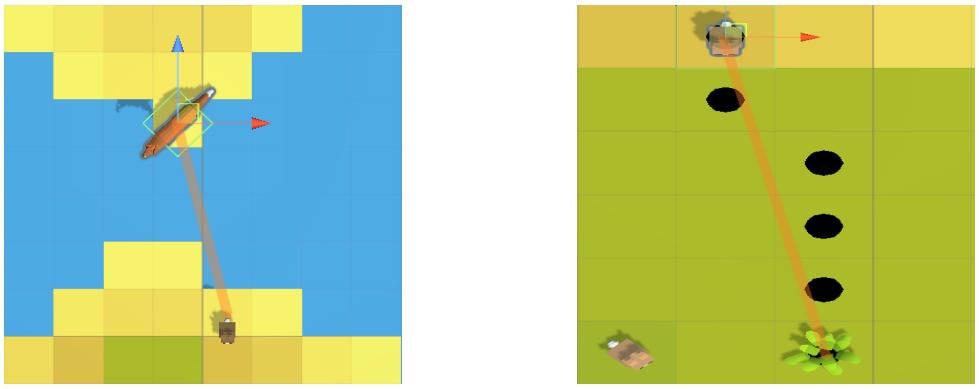


Figura 4.30: Recta dibujada en un sistema mapeado.

Esto es simplemente un ejemplo ilustrativo para explicar brevemente el problema de representar líneas rectas en un ordenador. En realidad lo que nos darán serán dos puntos y los algoritmos de dibujo de líneas rectas tratan de minimizar el numero de cuadrantes que tenemos que dibujar. Uno de los algoritmos más utilizados (y el que hemos implementado) es el de **Bresenham**. Este algoritmo usa cálculos de número enteros, lo cual consume menos memoria que si usáramos números de coma flotante.

Cuando nuestros agentes, por ejemplo, tienen que dirigirse a beber agua, conocen su posición y la del agua así que el camino más corto será la línea recta. Esto crea una serie de problemas ya que el algoritmo de Bresenham tiene limitaciones significativas en nuestro caso. Ciertamente, la recta es la ruta óptima, pero no siempre es posible. En nuestro proyecto, es posible que dicho camino sea imposible porque se encuentre alguna clase de obstáculo en medio tales como los árboles o el propio agua.



(a) Recta imposible. **(b)** Recta óptima generada con Bresenham.

Figura 4.31: Ejemplo problema de las líneas rectas en el mapa.

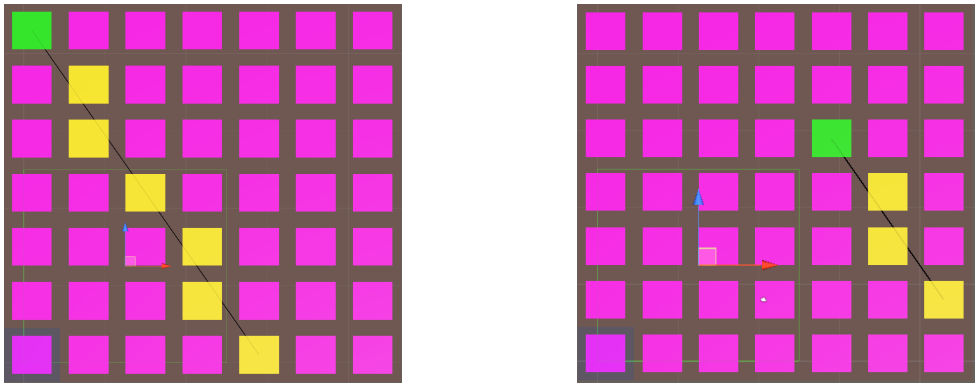
Este problema es extremadamente común y por ello existen algoritmos de pathfinding que lo solucionan. Técnicamente, el algoritmo de Bresenham no está diseñado para encontrar rutas óptimas porque suele ser muy común el encontrar obstáculos o pesos en las rutas en esta clase de problemas pero al ser un algoritmo rápido y teniendo en cuenta que en nuestro proyecto las rectas imposibles no sucederán muchas veces, hemos decidido implementarlo como primer intento para encontrar la ruta. Pero este problema no lo hemos dejado sin solución, ya que también hemos implementado de forma alternativa el algoritmo de pathfinding **A*** que detallaremos más adelante.

Dentro del script **Pathfinder.cs** están implementados los algoritmos de **Bresenham** y **A***. El primero se usará de manera principal por su velocidad y el segundo se usará de manera complementaria al primero en caso de que no pueda devolver un camino válido.

Existen varias formas de implementar el algoritmo de Bresenham, ya sea a través del cálculo de error o a través de la pendiente de la curva. Nosotros hemos implementado con cálculo de error. Dados el **origen (x1,y1)** y el **destino (x2,y2)**, lo primero que tenemos que hacer es saber si vamos a tener que aumentar o reducir en los ejes x e y que podemos averiguarlo fácilmente a través de la diferencia entre origen y destino. A continuación, si alguna de las diferencias es nula, usaremos una función auxiliar especializada ya que la recta será o bien horizontal o vertical. De lo contrario, entraremos en un bucle hasta encontrar la ruta o devolver un error.

```
Vector3[] BresenhamError(origen (x1,y1), destino (x2,y2))
    int dx = |x2-x1|;//Diferencias
    int dy = |y2-y1|;
    int sx = x1<x2? 1:-1;//Aumentar o reducir en los ejes
    int sy = y1<y2? 1:-1;

    if(dx==0 || dy==0)//Linea recta vertical u horizontal
        return BresenhamRecto((x1,y1), (x2,y2));//Funcion auxiliar especializada

    int err = dx+dy;//Error de la curva
    while(true)
        camino.Add(x1,y1);//Camino solucion
        if(x1==x2 && y1==y2)//Hemos llegado al final
            break;

        int e2 = 2*err;
        if(e2>=dy)//Error pasa el umbral
            err += dy;
            x+=sx;//Aumentamos X
        if(e2 <= dx)//Error pasa el umbral
            err += dx;
            y+=sy;//Aumentamos Y
    return camino;
```

La función auxiliar **BresenhamRecto(origen(x1,y1), destino(x2,y2))** se encarga de devolver la ruta siendo esta completamente vertical u horizontal. Así es más directo y rápido que si el algoritmo normal lo hiciera.



(a) Recta generada con Bresenham. **(b)** Recta generada con Bresenham.

Figura 4.32: Rectas generadas con el algoritmo de Bresenham.

Las imagenes que acabamos de mostrar han sido creadas a partir de una demo que hemos creado en Unity cuyo repositorio es público en el siguiente enlace[15]. Dentro de la demo

[15]https://github.com/DripyDev/DemoPathfinding

se pueden ver los resultados que devuelven el algoritmo de **Bresenham** y el **A*** que explicaremos a continuación. Los controles de la demo están detallados en su correspondiente archivo README.

Con este algoritmo ya implementado, los agentes ya pueden dirigirse a sus objetivos, pero como hemos comentado antes, hay casos en los que Bresenham no será suficiente para encontrar un camino. Necesitamos una función que sea capaz de encontrar la ruta óptima teniendo en cuenta posibles obstáculos. Es cierto que podríamos modificar Bresenham para que se amolde a nuestras necesidades, pero teniendo en cuenta que su objetivo no es el de encontrar caminos óptimos sino dibujar rectas y que existen multitudes de algoritmos que si son capaces de hacer esto como: **Dijkstra**, **A***, **Bellman Ford**, **Floyd Warshall**, **Johnson**, **D***. Hemos decidido implementar uno de estos, el **A*** (*'A star'*) en concreto, un algoritmo de búsqueda de grafos informados que se guía por un heurístico.

A* es fácil de entender. Cada casilla del espacio (o nodo, el algoritmo es aplicable a grafos) contiene tres valores, **G**, **F** y **H**. **G** es el coste desde el nodo de inicio al nodo actual. **H** es el valor heurístico del nodo y **F** es la suma de **G** y **H**. El coste entre nodos es **10** en caso de que sea un vecino vertical u horizontal y **14** en caso de que sea un vecino diagonal. La diagonal vale 14 (raíz de 2) porque matemáticamente la distancia entre el centro de una casilla de tamaño uno y el centro de su casilla vecina diagonal es 14, aunque podría usarse **1.4** pero con números enteros usamos menos memoria.

Lo primero a tener en cuenta es que vamos a tener dos listas de nodos, una para los ya analizados a la cual llamaremos **closedList** y otra para los nodos que aún tenemos que analizar, llamada **openList**. Partiendo desde el nodo origen, vamos a calcular sus parámetros, meterlo en la lista cerrada y a continuación analizaremos los de sus vecinos que añadiremos a la lista abierta. Calculamos los valores de los vecinos y cuando terminemos cogeremos aquel que tenga una menor **F** de la lista abierta y volveremos a analizar a sus vecinos (que no estén en la lista cerrada), así continuamente hasta llegar al destino o hasta que no tengamos más nodos en la lista abierta. Para rastrear el camino que seguimos, **cada nodo tendrá una referencia sobre el nodo del que viene**. El valor de **F** en nuestro caso es **H+G** pero realmente podríamos usar cualquier fórmula que queramos en función de que parámetro valoremos más. La eficiencia del algoritmo está fuertemente ligada al

heurístico que usemos y en el peor de los casos (en el que tenga que recorrer el mapa entero) tendrá una complejidad exponencial pero si existe una solución y el heurístico es admisible, A* encontrará la solución óptima.

Habiendo entendido ya el funcionamiento del algoritmo, solo queda realizar un pequeño cambio para que se amolde a nuestro caso. Tenemos que tener en cuenta que puede que algún nodo no sea válido/caminable. El cambio que esto requiere es mínimo, basta con comprobar que el nodo sea válido y en caso de que no lo sea, meterlo en la lista cerrada de forma inmediata.

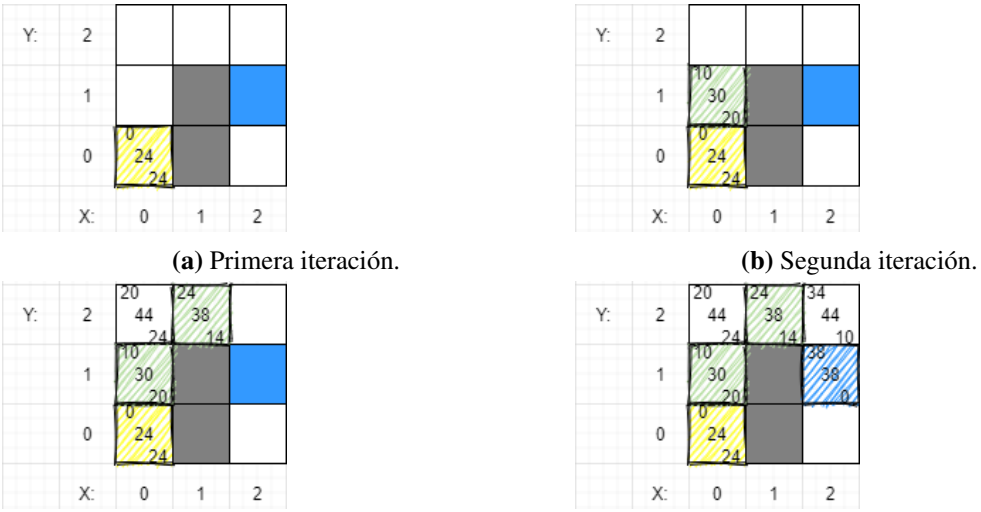
```
1  Vector3[] AStar(nodoOrigen, nodoDestino)
2      nodoOrigen.CalcularValores();
3      openList.Add(nodoOrigen);
4
5      while(openList no vacia)
6          nodoActual = NodoMenorF(openList);
7
8          if(nodoActual==nodoDestino)//Hemos encontrado el camino
9              return ReconstruirCamino(nodoActual);
10
11         openList.Remove(nodoActual);
12         closedList.Add(nodoActual);
13
14         foreach(Nodo v in Vecinos(nodoActual))//Nodos vecinos
15             if(closedList.Contains(nodoActual))//Saltamos al vecino
16                 continue;
17
18             if(!v.caminable)//Vecino no caminable
19                 closedList.Add(v);
20                 continue;
21
22             int gTentativa = nodoActual.g + CalcularH(nodoActual, v);
23             if(gTentativa < v.g)
24                 v.VieneDe = nodoActual;
25                 v.g = gTentativa;
26                 v.h = CalcularH(v, nodoDestino);
27                 v.f = v.g+v.h;
28
29                 if(!openList.contains(v))
30                     openList.Add(v)
31
32     return null;
```

La función **ReconstruirCamino(Nodo a)** simplemente recorrerá los nodos de los que procede el nodo final y devolverá la lista dada la vuelta. **CalcularH(Nodo a, Nodo b)**

devuelve la distancia entre ambos nodos teniendo en cuenta que el movimiento diagonal vale 14 y el resto 10 (como hemos explicado anteriormente).

```
int CalcularH(Nodo (x,y), Nodo (x1,y1))
    int xD = |x-x1|;
    int yD = |y-y1|;
    int resto = |xD-yD|;

    return 14*Min(xD,yD) + 10*resto
```

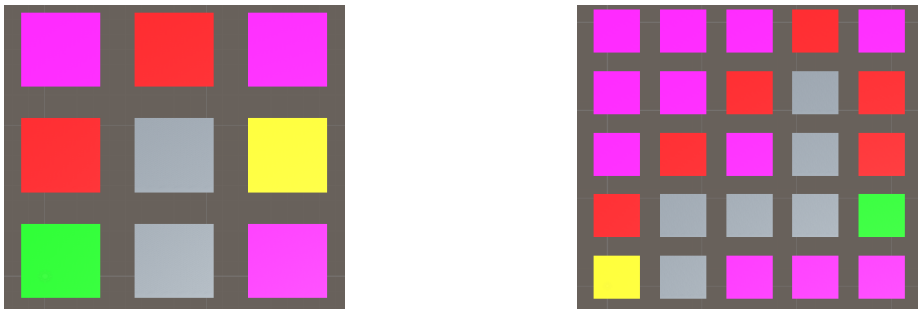
Tenemos que tener en cuenta que cada vez que usemos la función **AStar(origen, destino)** deberemos de resetear los valores del mapa de nodos que hemos usado, de lo contrario quedará comprometida la eficiencia del algoritmo en las siguientes iteraciones. En nuestro caso, cada animal tendrá una copia del mapa de nodos para que opere con ella como quiera.



(a) Primera iteración. **(b)** Segunda iteración.

(c) Tercera iteración. **(d)** Resultado.

Figura 4.33: Iteraciones algoritmo A*.



(a) Ejemplo 1 A*. **(b)** Ejemplo 2 A*.

Figura 4.34: Ejemplos caminos con el algoritmo A*.

Las imagenes que acabamos de exponer han sido tomadas a partir de una demo creada por nosotros en Unity. Es la misma demo que hemos usado para exponer los resultados del algoritmo de Bresenham. El siguiente enlace[16] contiene el repositorio.

Figura 4.35: Camino generado con A*.

[16]https://github.com/DripyDev/DemoPathfinding

4.4.3.2. Agentes

El control y administración de las decisiones de los agentes está implementada en **cinco scripts** dentro de la carpeta **Behaviour**. Todos los agentes son considerados *entidades vivas* y dentro de estas entidades tenemos **plantas** y **animales**. Los zorros y los conejos son clases separadas que heredan de los animales.

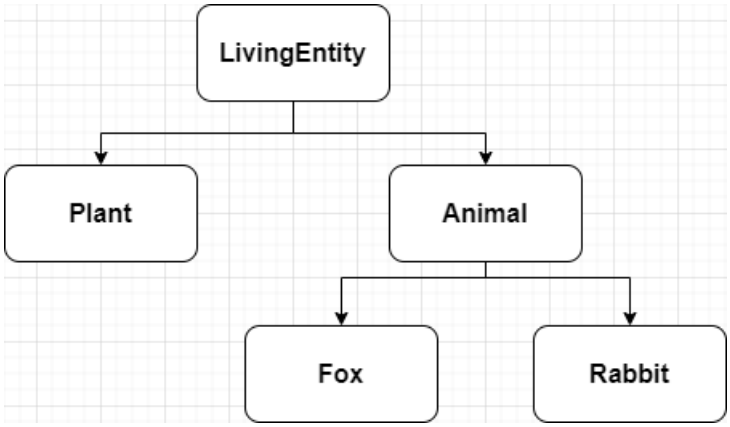


Figura 4.36: Jerarquía de las entidades.

- **LivingEntity**

Es la clase padre de todas con lo cual debe de contener lo que todos los entes tengan en común. Todos los agentes tienen una **posición en el mundo**, **posición en el mapa** (por la subdivisión del mundo) y si están **vivos o muertos**. La responsabilidad de esta clase únicamente es inicializar la entidad en el mundo (junto a sus materiales) y registrar la muerte (el registro de muertes es almacenado para sus consiguientes estadísticas). No hay nada más que destacar en este archivo.

- **Plant**

Las plantas son entidades vivas pero no van a moverse ni a tomar decisiones así que su correspondiente script también es sencillo. Las plantas tendrán un número que representa su 'valor nutricional' y otro para ver la velocidad a la que van a ser comidas. Dentro de la clase solo hay dos funciones, una llamada **Consume(float cantidad)** para cuando un agente las comience a consumir. La otra es **Update()** que es una función propia de Unity y se ejecuta una vez por cada fotograma. Vamos a usarla para que la planta vuelva a crecer

hasta su tamaño original siempre y cuando nos hayan comido un poco. Como se puede deducir, tanto cuando nos comen como cuando crecemos, el modelo de la planta se reducirá o aumentará en tamaño en función de la cantidad de 'valor nutricional' restante que tengamos.

Las plantas las haremos aparecer en el mundo de manera completamente aleatoria (no aparecerán en casillas no caminables, de agua o con árboles). Haremos aparecer una población inicial de plantas en el mundo y luego haremos aparecer otra en una posición aleatoria cada 0.5 segundos.

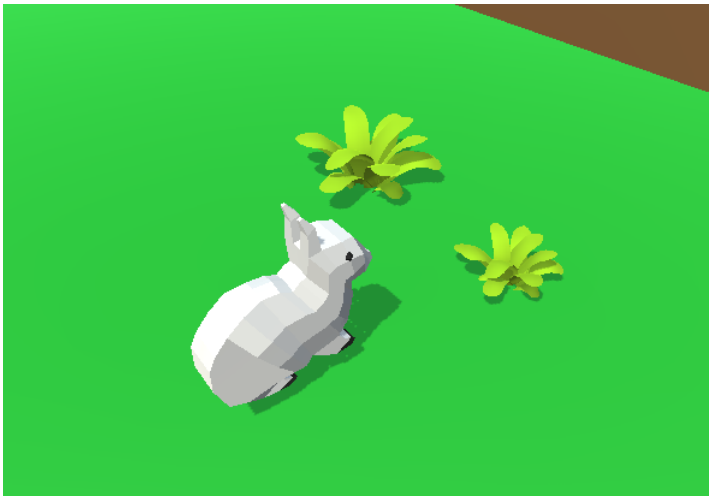


Figura 4.37: Conejo consumiendo una planta.

- **Animal**

El script de mayor tamaño de los presentes en **Behaviour**, administra los genes de los animales, como crean caminos, como actúan tras tomar una decisión, como encuentran agua, comida y parejas, el embarazo y crecimiento. Todas estas son las características que comparten tanto conejos como zorros.

Los genes tienen su propio script (**Genes.cs**) dentro de la carpeta **Datatypes**. Esta clase tiene **tres** parámetros principales, probabilidad de mutación que representa cuantas probabilidades tienen de que uno de sus genes mute, cuanto más alto más probabilidades habrá. Un booleano para determinar si es macho o no y un array de bits para representar

que genes tiene activados. Dentro de la clase existen funciones para generar los genes de manera aleatoria o a través de la reproducción sexual por lo que los heredará de sus padres en caso de que ambos padres tengan dicho gen activo (teniendo siempre en cuenta la probabilidad de mutación de que el gen se active o desactive).

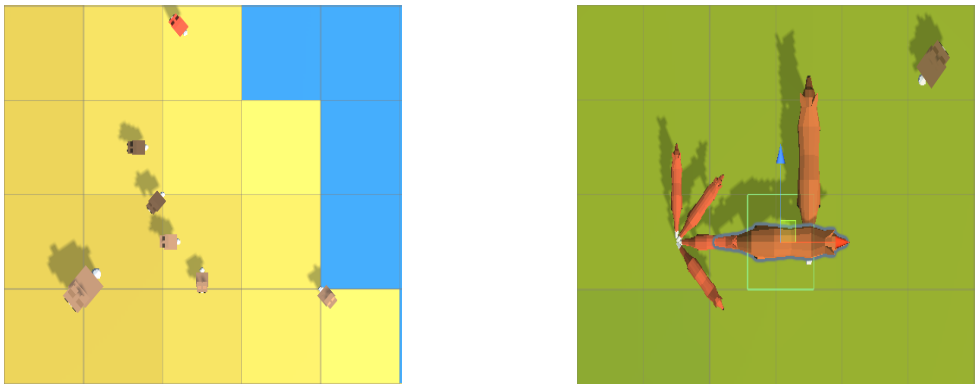
A la hora de **crear caminos** (usamos Bresenham o A* como hemos explicado en el apartado *Pathfinder*) lo primero que haremos será comprobar que la casilla a la que queremos llegar se encuentre dentro de los límites de nuestro mundo, en caso de que esté fuera de rango, alteraremos el destino por uno dentro de los límites. Una vez sea un destino válido, intentaremos generar una ruta con el algoritmo de Bresenham y en caso de que no se pueda porque haya un obstáculo en medio, usaremos A*.

La función **Act()** se llama después de que decidan que acción quieren tomar. Se encarga de preparar la acción. Por ejemplo, si el agente está explorando, busca una casilla a la que avanzar y cuando la encuentre comienza a moverse físicamente hacia ella (cuando un agente está explorando coge una casilla caminable de forma aleatoria pero con tendencia hacia adelante como se ha explicado en el apartado *Mundo* cuando hablamos de la exploración de los agentes). Si estamos moviéndonos hacia una fuente de alimento (ya sea una planta o otro animal) comprobaremos si somos vecinos del destino, en caso de que lo seamos comenzaremos a comer (o beber) pero en caso de que no lo seamos, crearemos una ruta hacia ahí. Si estamos buscando pareja, detectaremos los entes de nuestra misma especie y si también están buscando pareja solicitaremos reproducirnos con ellos, esta solicitud será aceptada por la hembra de manera aleatoria pero en función de la deseabilidad del macho que podrá aumentar en función de sus genes (cuanto más rojo sea su pelaje mayor probabilidades tendrá de que acepten su petición), si aceptan la solicitud procederemos a reproducirnos, si nos rechazan, guardaremos ese agente en una lista de *hembras no impresionadas* o *machos rechazados* para no solicitar constantemente apareamiento con la misma entidad.

Para **encontrar el sustento, agua o comida, o posibles parejas** vamos a usar funciones auxiliares implementadas dentro del script **Mundo.cs**. Para el agua, cuando hemos generado el mundo, hemos almacenado la posición del agua más cercana de cada casilla, de esa manera podemos acceder a ella de manera más rápida y no hará falta que estemos

buscando la fuente de agua más cercana de manera constante. Por supuesto, es posible que la casilla de agua más cercana esté fuera de nuestro rango de visión en cuyo caso el agente no percibirá la casilla y pasará a explorar con la esperanza de acabar encontrando casillas con líquido. En caso de que si la encuentre, generará una ruta óptima hasta ella y cambiará su objetivo actual a *dirigirse a por agua*. Para encontrar alimento, vamos a tener que percibir a todas las posibles presas en nuestro rango de visión, en caso de que haya alguna nos dirigiremos a por la presa más cercana si existe, de lo contrario, exploraremos, es similar a buscar agua. Para encontrar parejas, como hemos mencionado, sentiremos a los entes de nuestra propia especie que también busquen pareja y una vez estemos a su lado solicitaremos apareamiento, en caso de que acepten, la hembra quedará embarazada vaciando el instinto reproductivo de ambos, en caso de que no acepten lo agentes no volverán a solicitar apareamiento entre ellos nunca más. En el caso en el que existan múltiples posibles parejas válidas, nos quedaremos con la más cercana.

La **comprobación del embarazo** se hará en cada fotograma dentro de la función **Update()** que como ya hemos mencionado, se ejecuta una vez por fotograma. Simplemente comprobaremos si estamos embarazados o no, en caso de que lo estemos nos moveremos más lentos haciéndonos más vulnerables, nuestro instinto reproductivo dejará de aumentar y el tiempo de embarazo aumentará. Cuando dicho tiempo supere un umbral arbitrario, uno en nuestro caso, daremos a luz un número aleatorio de entidades (entre 4 y 13 en los conejos y entre 4 y 6 en zorros) que heredarán los genes de los dos padres. Las crías tendrán un menor rango de visión y velocidad hasta que lleguen a la adultez.



(a) Crías de conejo. **(b)** Crías de zorro.

Figura 4.38: Agentes teniendo crías.

La función de crecimiento es muy simple, vamos aumentando la edad de manera constante y dejaremos de ser crías cuando la edad sea mayor a **0.15**. Nuestra velocidad y rango de visión volverán a ser normales.

- **Fox**

Como hemos mencionado antes, **Fox** y **Rabbit** son los encargados de administrar su toma de decisiones ya que eso es lo único que los diferencia entre ellos. El script **Fox.cs** se encarga de actualizar el estado del zorro constantemente (para ello usará la función **Update()**) y de la toma de decisiones del mismo.

Dentro de **Update()**, que se ejecuta una vez por fotograma, vamos a hacer **tres clases de comprobaciones diferentes**. La **primera** es aumentar el hambre, sed, edad, instinto reproductivo, hacerlo crecer (serán crías hasta que tengan una edad superior a 0.15) y comprobar y administrar el embarazo.

La **segunda** comprobación es la toma de decisiones. Aquí tenemos que tener en cuenta tres estados diferentes en los que se puede encontrar el agente, puede estar en **movimiento**, **alimentándose** o **pensando su siguiente acción**. Si estamos en movimiento (tiempo físico que tarda en agente en moverse de una casilla a otra adyacente) no vamos a tomar ninguna decisión ni nada parecido hasta que terminemos el movimiento. Si estamos alimentándonos de una presa (ya sea una planta o un conejo) o bebiendo agua, no tomaremos decisiones hasta terminar y solo tomaremos una nueva decisión una vez haya pasado

cierto tiempo desde la última vez que lo hemos hecho (un segundo en nuestro caso).

La **tercera** y última comprobación es la correspondiente a la muerte del ente ya sea por hambre, sed o edad. Cuando cualquiera de estos tres parámetros supere un umbral (uno en nuestro caso) moriremos y nuestra causa de muerte será registrada para el estudio de dicha información.

Para la toma de decisiones teníamos como objetivo usar un sistema **BDI** (Belief-Desire-Intention) pero de momento usa un sistema reactivo para la toma de decisiones. El ente buscará pareja siempre que esté bien alimentado y su instinto reproductivo supere un umbral. Se reproducirá si la posible pareja también está buscando pareja y es del sexo opuesto. La hembra quedará embarazada reduciendo su velocidad en un 30 % y tras un tiempo dará a luz entre **4 y 6 crías** que hasta que sean adultos también se moverán mas lento y tendrán un rango de visión reducido. En caso de que el agente no esté bien alimentado o tenga sed, buscará agua y comida siempre dando algo de prioridad al agua. Una vez ha decidido que acción desea tomar, generará el camino óptimo para cumplir su objetivo. Las acciones que puede tomar son: **nada, descansar, comiendo, bebiendo, reproduciéndose, explorar, ir a por comida, ir a por agua, huir** (solo los conejos huyen) **y buscar pareja**. A la hora de necesitar crear un camino (no nos hará falta cuando estemos explorando, comiendo, bebiendo o descansando) solo lo generarémos siempre y cuando no seamos vecinos de la casilla objetivo pero en caso de que sea necesario lo haremos a través del algoritmo de Bresenham y A* en caso de que Bresenham no devuelva un resultado válido (como hemos explicado en el apartado *Pathfinder*).

Figura 4.39: Zorro persiguiendo a una presa.

- **Rabbit**

La diferencia entre los conejos y los zorros es únicamente la toma de decisiones (el conejo huirá de forma prioritaria) y el número de crías que dan a luz. El resto es igual y ha sido explicado en el apartado *Fox*.

Los consejos tendrán entre cuatro y trece crías. Cuando el conejo detecta un depredador dentro de su rango de visión, su prioridad absoluta será huir de este en dirección opuesta. Huirá a la casilla resultado de sumar a su posición el vector entre él y el depredador.

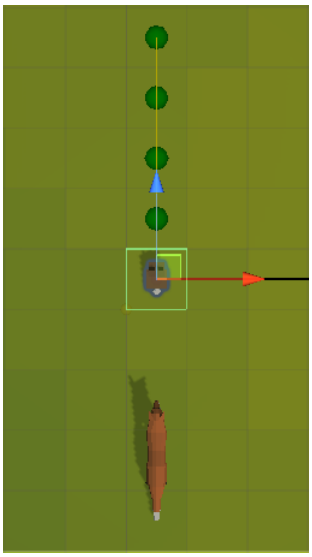


Figura 4.40: Conejo huyendo de un depredador.

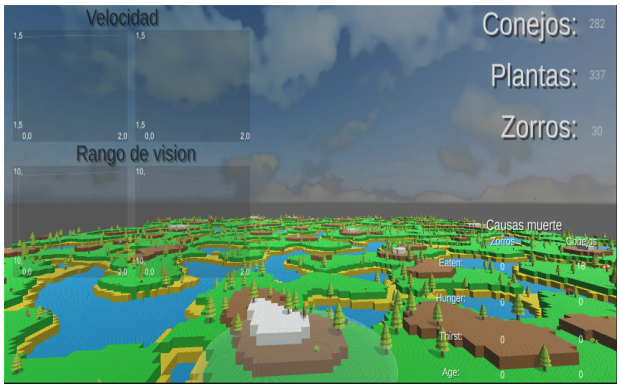
4.4.3.3. **Menús**

La simulación contiene una serie de menús para que podamos visualizar de manera cómoda algunos datos relevantes. Tenemos **tres menús** diferentes. Uno es la **pantalla de pausa** que se activa al darle al botón ***'escape'*** y simplemente pausa el juego reduciendo el parámetro *Time.timeScale* a cero que es la escala a la que pasa el tiempo. Si volvemos a pulsar el botón ***'escape'*** o pulsamos el botón **'Resume'** del menú emergente volveremos a la normalidad. La opción **'Menu'** que aparece no hace nada porque forma parte de un menú que intentamos diseñar originalmente pero que al final abandonamos porque no le veíamos gran utilidad.

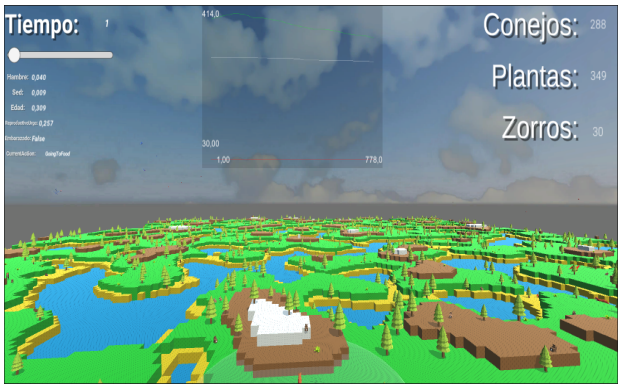
Los **otros dos menús** son para mostrar información relevante como el número de entes vivos en ese momento, las causas de muerte, las medias de velocidad y rango de visión de

Figura 4.41: Menú de pausa.

los agentes o una estadística sobre la población actual. Las estadísticas han sido realizadas con un paquete de Unity gratuito llamado Simplest Plot[17] como ya hemos mencionado en el apartado *Elección de tecnología*.



(a) Menú con la velocidad y rango de visiones medios.



(b) Menú con el número de seres vivos.

Figura 4.42: Menús con información relevante.

Para pasar de un menú a otro, basta con pulsar la tecla correspondiente al **tabulador**. Si clickamos en un animal, aparecerá a su alrededor una circunferencia que representa su rango de visión y también aparecerá su información como hambre, sed, edad, instinto reproductivo, si está embarazado o no y su objetivo actual en la parte superior izquierda del menú con la gráfica de población actual. Justo encima de dicha información tenemos una barra que podemos deslizar para alterar la velocidad a la que funciona el mundo. Al igual que con el menú de pausa, esa barra controla el parámetro **Time.timeScale** lo que nos permite acelerar o reducir la velocidad de la simulación. Este valor permitimos que vaya entre 0 y 100 pero no aseguramos que funcione correctamente si se le da valores

[17] https://assetstore.unity.com/packages/tools/gui/simplest-plot-94876

muy altos.

Vamos a aprovechar para resaltar un **bug** que nos hemos encontrado pero no hemos sabido solucionar. Al abrir la simulación por primera vez, si intentamos ejecutarla al instante salta un error en el GameObject que contiene la gráfica de velocidades medias de los conejos. Si eliminamos el script ***'SimplestPlot.cs'*** de dicho objeto y lo volvemos a añadir el error desaparece. **SimplestPlot.cs** es un script perteneciente a la biblioteca SimplestPlot que usamos asi que desconocemos el motivo de este bug y desgraciadamente no hemos sido capaces de solucionarlo de forma permanente. Pero este bug, aunque molesto, no es de grán importancia.

Como nota adicional, vamos a señalar que también creamos un menú de inicio para la simulación que acabamos apartando para priorizar otras tareas así que quedó incompleto. Este menú iba a ser el inicial y iba a servir para modificar los parámetros del ecosistema desde pantalla en lugar de desde el inspector de Unity.

Figura 4.43: Menú de inicio no finalizado.

4.4.4. Cámara

Como último apartado queremos explicar brevemente el funcionamiento de la **cámara** de la simulación.

Dentro de la escena existen un GameObject llamado *MainCamera* que tiene el componente *Camera* que le da las propiedades necesarias para que sea la cámara principal de la escena. También tiene un script que hemos creado llamado **CameraMovement.cs** para controlar su movimiento en función de las entradas leídas del teclado y ratón. Este script tiene **tres** parámetros, un float que es la **sensibilidad del ratón**, un tipo *transforma* que será la **posición de la cámara** y un booleano para el **movimiento recto**.

Los **controles** de la cámara son sencillos:

- **Cambio de modo de movimiento**: El booleano *'movimiento recto'* que hemos mencionado antes se activa o desactiva al pulsar la tecla **'shift'**. Estando activo, la cámara se moverá en direcciones globales sin tener en cuenta a donde mira. Si está desactivado, se moverá en su espacio local teniendo en cuenta a donde está mirando.
- **Movimiento**: Para controlar el movimiento general. Las teclas son: **adelante**: w **/** flecha arriba, **atrás**: s **/** flecha abajo, **izquierda**: a **/** flecha izquierda y **derecha**: d **/** flecha derecha. Estos movimientos son diferentes en función del booleano *movimiento recto*.
- **Arriba-Abajo**: No tienen en cuenta el *movimiento recto*, la tecla **espacio** hace que la cámara se mueva hacia arriba en el espacio global y **control izquierdo** que se mueva hacia abajo.
- **Dirección cámara**: Para controlar a donde mira la cámara tenemos que mantener pulsado el **click derecho** y mover el ratón hacia donde queremos mirar. El parámetro **sensibilidad del ratón** controlará como de sensible será este movimiento, a mayor valor, más rápido se moverá la dirección de la cámara.

El script tiene una función **movimientoCamara()** que lee el input del teclado y ratón y actua en función. Esta función es llamada en otra llamada **Update()** (que ha sido men-

cionada anteriormente en la memoria) que es una función predefinida de Unity que es llamada una vez por fotograma. Si aumentamos el factor de escalado de tiempo de Unity (*Time.timescale*) la cámara será afectada y se moverá más rápido de lo normal.

4.5. Simulaciones

Vamos a aprovechar este capítulo para analizar y estudiar los resultados de nuestras simulaciones. Lo primero que tenemos que tener en cuenta es la relación depredadores-presas existentes. La simulación *'normal'* suele incluir conejos, zorros y plantas donde los conejos son depredadores de las plantas y los zorros de los conejos. Es interesante ver y analizar como aumentan y disminuyen las poblaciones con respecto al tiempo. Las plantas aparecen de manera aleatoria en el mapa (en casillas caminables) cada X segundos, este valor debería de ser bajo para intentar sostener la población de conejos, por ejemplo, 0,5. Con los conejos y zorros, aparece una población inicial de adultos y tras eso tendrán que reproducirse para aumentar sus números.

Vamos a analizar **dos** situaciones diferentes. Una donde solo existan **conejos y plantas** y otra donde los **conejos tengan depredador**.

4.5.1. Conejos-Plantas

En esta simulación solo existirán conejos y plantas, los zorros no participarán. La población inicial será de **250 conejos** y **400 plantas**. Este sistema es un sistema de depredador-presa simple, las plantas se 'reproducen' infinitamente y los conejos las depredan por lo que las poblaciones de ambos entes irán aumentando y reduciendo en función de la población contraria. Cuando haya abundancia de alimentos, la población de roedores sufrirá un gran aumento por este superávit de comida pero cuando acaben con todos los recursos de su zona comenzarán a morir de hambre lo que causará una muerte masiva de conejos que a su vez hará que la población de plantas vuelva a aumentar. Si los parámetros iniciales son suficientes, es decir, que haya una cantidad de alimento inicial suficiente para la población inicial de conejos, este ciclo se repetirá infinitamente.

Los conejos tienen genes que pueden transmitirse entre ellos, de estos genes, los más interesantes para su supervivencia son el aumento de su **rango de visión** y aumento de la **velocidad** (que a su vez aumenta su ratio de hambruna). Como los recursos son limitados, los depredadores tendrán que empezar a competir entre ellos si quieren sobrevivir y con suficiente tiempo, los genes más útiles para la supervivencia prevalecerán y podremos ver cuales son más importantes. Tras bastante tiempo de simulación acelerada, podemos ver claramente los aumentos y disminuciones de la población tal y como hemos predicho antes.

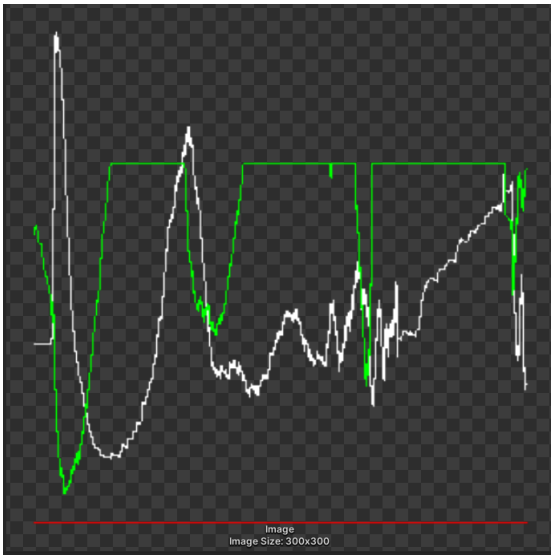
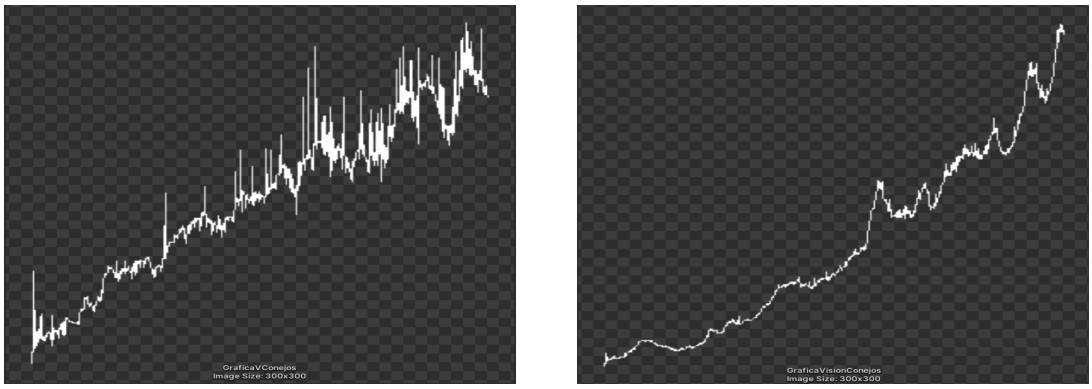


Figura 4.44: Población plantas (verde) y conejos (blanco).

En la figura que acabamos de exponer podemos ver el número de conejos (blanco) y plantas (verde) por tiempo. Resaltamos que el número representado de plantas está limitado a 500 para que no disminuya el tamaño de la gráfica y no deje distinguir los cambios más sutiles en la población de conejos.

Podemos observar las claras caídas y aumentos poblacionales. Mientras los recursos sean abundantes, los conejos se reproducirán sin control hasta que su entorno no sea capaz de sostenerlos y entonces comenzarán a morir de hambre y a competir por los pocos recursos restantes. El primer y mayor pico es porque la población inicial son todos adultos por lo que se reproducirán todos, el resto de picos son menores porque esta primera generación ira siendo sustituida por crías cuyos parámetros son un 30 % peores a los de los adultos así que les cuesta más sobrevivir. Esto explicaría las pequeñas caídas poblacionales. Vamos a mirar cuales son los resultados de los genes trasmitidos.



(a) Velocidad media. **(b)** Rango de visión medio.

Figura 4.45: Comparación de velocidades y rangos de visión medios de conejos.

Como vemos, tanto la velocidad de movimiento como el rango de visión han aumentado respecto al tiempo. La velocidad inicial es de **1.5** y el rango de visión de **10**. En las gráficas observamos que ambos han aumentado pero no lo han hecho en el mismo grado. El rango de visión medio ha subido hasta aproximadamente **30** mientras que la velocidad solo hasta **1.7**. Cuando un animal tiene el gen de la velocidad activo, su velocidad (que es la media de sus padres) aumenta, lo mismo pasa con el rango de visión. Observamos que el rango de visión ha sido más relevante que la velocidad (que también aumenta como de rápido pasamos hambre). Esto tiene sentido ya que como la velocidad también aumenta como de rápido pasas hambre, los animales valoran más el poder encontrar la comida desde más lejos que sus competidores.

Las causas de las muertes son **4311 por hambre**, **424 por sed** y **94 de vejez** en el momento en el que hemos parado la simulación porque podría seguir infinitamente. Vemos que el factor más importante, con una basta diferencia, es la comida. La escasez de comida explicaría porque no se premia la velocidad sobre el rango de visión.

4.5.2. Zorros-Conejos-Plantas

Este sistema es un poco más complejo que el anterior porque la relación entre presas y depredadores es doble, los conejos con las plantas y los zorros con los conejos. Al igual que en la simulación anterior, podemos aventurarnos a predecir que si la población de conejos aumenta causará un pico en la población de depredadores que a su vez acabarán con los

conejos causando un declive en los roedores que hará que la población de zorros acabe siendo controlada. Cuando los conejos sean muy cazados por depredadores, el número de plantas aumentará considerablemente lo que causará una gran abundancia de comida para las presas y si los zorros no son suficientes para controlar a los conejos, estos volverán a crecer de manera exponencial. Una pregunta interesante es ver si los zorros son capaces de controlar lo suficiente la población de conejos como para que el sistema sea eternamente sostenible.

Vamos a comenzar una simulación en un **mapa 300x300** con **400 conejos**, **700 plantas** y **100 zorros** con plantas apareciendo de manera aleatoria en el mapa cada **0.2 segundos**. El valor nutricional de un conejo es un valor entre 0.3 y 0.6 y las plantas tienen un valor de 1. Los zorros van a tener recursos más limitados porque los conejos necesitan reproducirse para multiplicarse, a diferencia de las plantas.

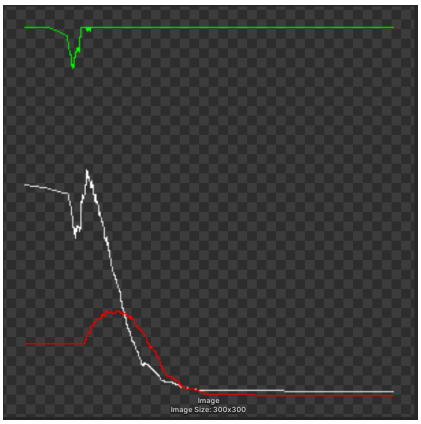


Figura 4.46: Población plantas (verde), conejos (blanco) y zorros (rojo).

Como observamos en la figura expuesta, el ecosistema no ha logrado el equilibrio. Los zorros han devorado a todas sus presas aumentando su número y acabando por exterminar a los conejos causando así su propia extinción. El número de zorros parece haber sido demasiado alto.

Vamos a realizar otra simulación con **400 conejos** y **50 zorros** en un mundo más pequeño de **200x200 casillas**.

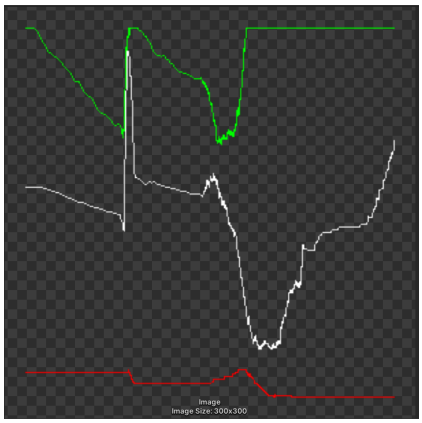


Figura 4.47: Poblaciones en un mundo 200x200.

Aún siendo el mundo de menor tamaño lo cual facilita el encuentro y reproducción de los animales, no parece que los zorros hayan conseguido mantener una población estable. Han acabado con todas sus presas de las zonas donde habitan y han acabado muriendo de hambre como consecuencia. Los conejos, como disponen de recursos con reproducción ilimitada seguirán un ciclo infinito como el que hemos descrito en la simulación sin depredadores. Los depredadores no se han reproducido lo suficiente como para que destaquen los genes de ciertos individuos pero parece que los conejos han valorado más tener un mayor rango de visión sobre una mayor velocidad.

¿Cuales son las causas de la extinción de los zorros?. Existen varias posibilidades, para empezar, puede que los zorros sean **demasiado eficientes cazando**. Los depredadores perseguirán a su presa hasta alcanzarla a no ser que encuentren una más cercana o decidan ir a por agua. La única manera de que el conejo consiga huir es si tiene mayor velocidad que la de su depredador ya que si este comienza a perseguirla y se mueven al mismo ritmo, acabará siendo devorada cuando no tenga escapatoria ya sea por llegar a la esquina de un mapa o por llegar a una casilla de agua. Otra posibilidad es que la **nutrición** que proporcionan los conejos **no sea suficiente** para mantener una población estable de depredadores. Para averiguar cual de las dos posibilidades es la respuesta a la pregunta, vamos a aumentar el valor nutricional de los conejos. Ahora tendrán un valor aleatorio entre 0.5 y 0.7. Las poblaciones iniciales y mapa serán los mismos.

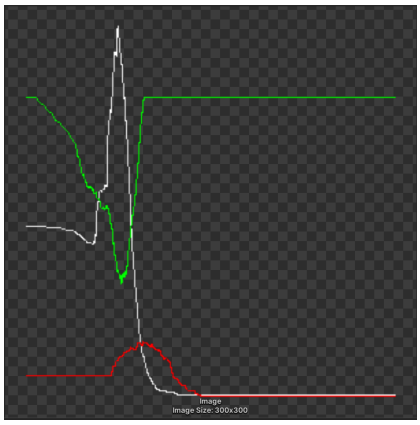


Figura 4.48: Poblaciones con mayor valor nutricional en los conejos.

La población de zorros ha aumentado más que antes pero al final el ecosistema no ha resultado estable y los zorros han acabado por extinguirse. Como no ha habido suficientes generaciones, no podemos sacar conclusiones sobre los genes mejor valorados.

Tras múltiples simulaciones más, no hemos conseguido crear un ecosistema que mantuviera un equilibrio. De esto podemos deducir que nuestras simulaciones de ecosistemas se aproximan a las reales siempre y cuando sea un sistema simple con un único depredador y una única presa pero cuando introducimos más factores (los zorros) el ecosistema deja de ser estable y acaban extinguiéndose. Desde el comienzo del proyecto sabíamos que los ecosistemas del mundo real son extremadamente complejos y con infinitas variables impredecibles e incluso aleatorias. Los resultados obtenidos del sistema zorros-conejos-plantas nos demuestra que esto es cierto y que si quisiéramos conseguir estabilidad en la simulación deberíamos de retocar y añadir nuevas características al sistema cosa que correspondería al *Trabajo futuro*.

5. CAPÍTULO

Conclusiones

En este apartado final hablaremos sobre las conclusiones que hemos sacado del proyecto, de lo que hemos aprendido durante el mismo, una valoración personal sobre el resultado general y sobre la gestión del trabajo, hablaremos de los objetivos conseguidos, los no conseguidos y explicaremos posibles mejoras para trabajo futuro.

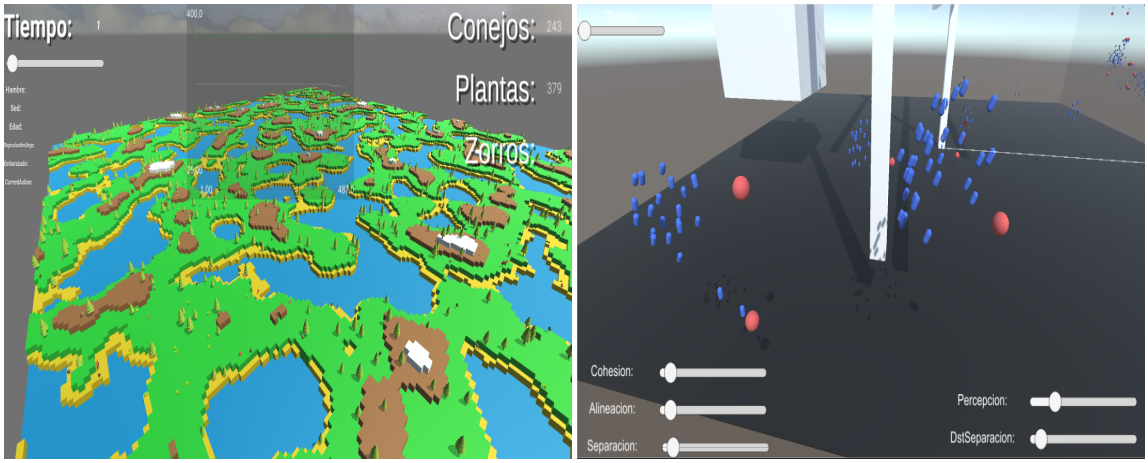
Queremos aprovechar este apartado para resaltar rápidamente que el contenido de esta memoria no ha podido ser repasado por el director *German Rigau Claramunt* por motivos personales suyos. La estructura de la misma si que ha podido ser repasada por el director, pero no el contenido.

5.1. Resultados

Tal y como planteamos al inicio del TFG, hemos cumplido la mayoría de los objetivos siendo el único que no hemos logrado completar por falta de tiempo el sistema de toma de decisiones BDI. Por lo demás, hemos creado un **generador de terreno** bastante completo, una implementación de los **Boids** de *Craig Reynolds* a la que hemos añadido varias reglas extras y un **ecosistema** donde unificamos todo y que incluye animales con la capacidad de **interactuar entre ellos**, **reproducirse** y **trasmitir genes**, **perseguir presas** o **huir de depredadores**, **morir** por diferentes causas y **alimentarse y beber**. También hemos creado una forma de **analizar estadísticamente** las características y datos del eco-

sistema como hemos explicado en la sección *Desarrollo del proyecto: Simulaciones* de donde sacamos conclusiones sobre el crecimiento de la población y las características más valoradas por los animales.

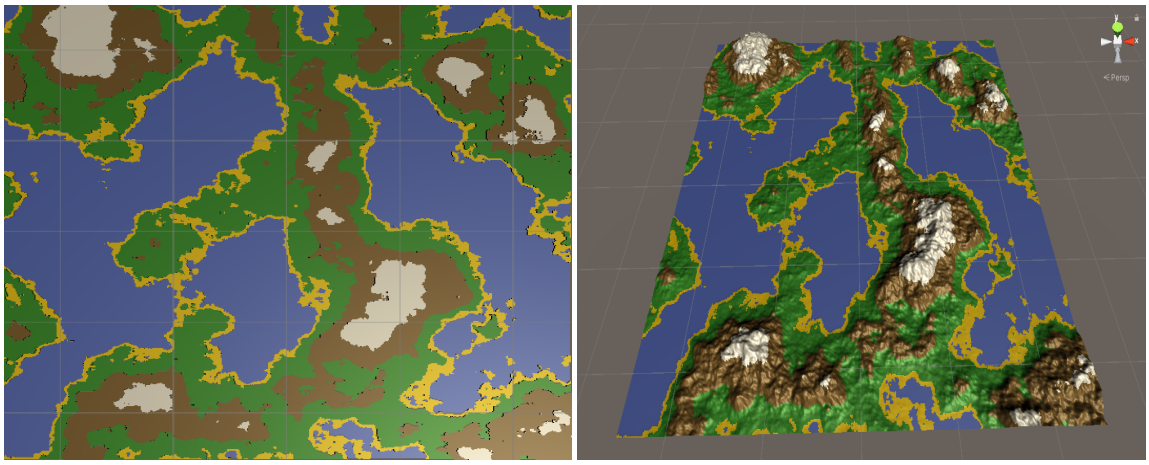
Visualmente el ecosistema es agradable y hemos acabado aprendiendo sobre muchos campos diferentes de los cuales no pensábamos que íbamos a acabar aprendiendo al comienzo del proyecto. Hemos acabado creando un total de **cinco repositorios** que hemos mencionada el el capitulo *Objetivos y Planificación inicial: Herramientas utilizadas: GitHub*. Uno para las **espirales de Fibonacci**[1] que usamos en la evasión de obstáculos de los boids, otro para los **Boids**[2], uno para experimentar con el **pathfinding** [3], uno para el **generador de terreno**[4] y el principal con el **ecosistema**[5] dentro del cual se unifican todos los demás.



(a) Captura ecosistema. **(b)** Captura Boids.

Figura 5.1: Capturas del ecosistema y los Boids.

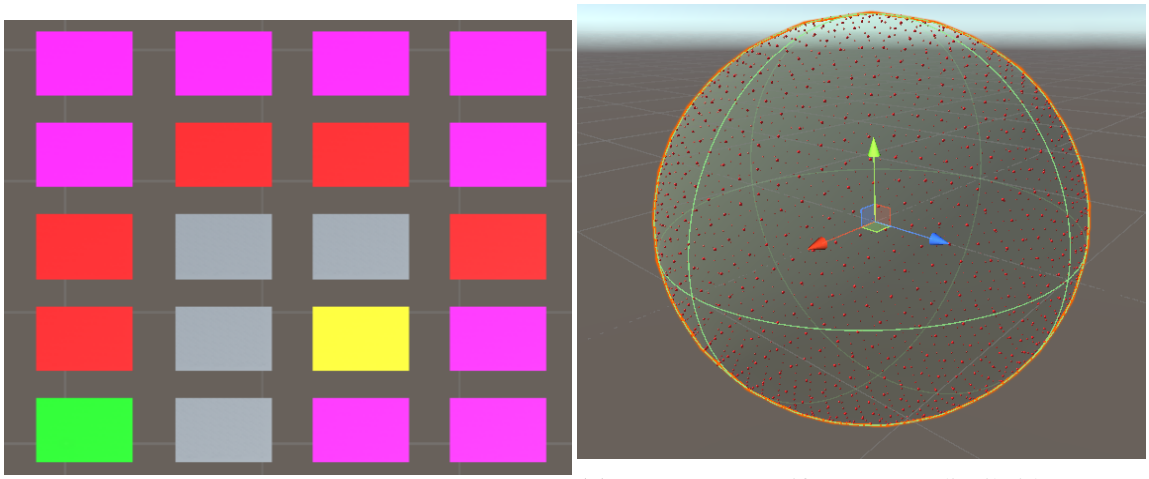
[1] https://github.com/DripyDev/EspiralesFibonacci
[2] https://github.com/DripyDev/Boids
[3] https://github.com/DripyDev/DemoPathfinding
[4] https://github.com/DripyDev/Generacion-de-terreno
[5] https://github.com/DripyDev/PruebaEcosistem-boid



(a) Mundo 500x500 con estilo de *casillas*.

(b) Mundo 500x500 generado.

Figura 5.2: Comparación resultados generador de terreno.



(a) Ejemplo pathfinder algoritmo A*.

(b) 2000 puntos uniformemente distribuidos en una esfera.

Figura 5.3: Ejemplo demo pathfinder y espirales de fibonacci.

5.2. Trabajo futuro

El resultado obtenido ha sido **satisfactorio**, aun teniendo en cuenta las tareas que no hemos conseguido completar. El generador de terreno cumple todos los requisitos que queríamos, los Boids cumplen todas las características que estaban planteadas para ellos desde el principio y el ecosistema también ha cumplido la gran mayoría de las características que deseábamos que tuviera.

Sin embargo, esto no significa que no puedan ser mejorados en el futuro. Todo trabajo siempre puede ser mejorado con algo más de trabajo y eso es lo que vamos a exponer ahora. Vamos a plantear posibles mejoras para cada uno de los tres apartados del proyecto.

5.2.1. Generación de terreno

La generación de terreno creemos que es el apartado que menos mejoras necesita si tenemos en cuenta la temática de *'casillas'* que hemos decidido seguir. Pero si no tenemos en cuenta esta temática, existen múltiples posibles mejoras ya que el mundo de la generación procedural de terreno es muy extenso y puede llegar a ser tremendamente complejo.

- **Optimizar el número de vértices del terreno**: A la hora de crear el terreno con temática de *'casillas'*, creamos vértices de más ya que creamos cuatro por cada casilla del mundo, incluidas las casillas necesarias para rellenar los huecos entre casillas adyacentes de diferentes alturas. Esto puede ser optimizado si a la hora de crear las casillas aprovechamos los vértices de la anterior para formar el nuevo cuadrado (siempre que estén a la misma altura). Por ejemplo, en un mundo **250x250** si usamos **cuatro** vértices por casilla, tendríamos un total de **250x250x4=250000** vértices pero si aprovechamos los vértices de las casillas adyacentes que tengan la misma altura podríamos reducir este número. A la hora de crear las casillas para rellenar huecos entre cuadrados de diferentes alturas, técnicamente, no debería de hacer falta ningún vértice más, solo harían falta más índices de triángulos para que se dibuje una cara ahí pero nosotros no tenemos esto en cuenta.
- **Climas**: Algunos generadores de terreno muy avanzados tienen en cuenta varios parámetros como humedad, calor, frió, porcentaje de vegetación etc etc para generar diferentes climas en su terreno. Nosotros podríamos adaptar algunos de estos parámetros para que tengan algún efecto en el terreno ya que lo único que tenemos es el porcentaje de que aparezcan árboles en el mundo.

5.2.2. Boids

La implementación de los Boids que hemos realizado es satisfactoria de manera independiente al ecosistema pero la interacción con el resto de animales es muy mejorable.

- **Hambre y sed**: Los boids y los depredadores no tienen en cuenta el hambre o la sed. Si que tienen en cuenta el cansancio y cuando llega a un umbral se dirigen al suelo y descansan pero esto es mejorable ya que se quedan quietos en el suelo hasta descansar a no ser que encuentren un depredador cerca.

- **Aleatoriedad al comportamiento**: Tanto los boids como los depredadores se rigen únicamente por sus reglas lo cual hace que su comportamiento sea hasta cierto punto predecible. Si un depredador persigue a un boid, si se mueven a la misma velocidad la perseguirá infinitamente a no ser que encuentre otro boid más cercano a él. Esto podría ser evitado añadiendo un factor de aleatoriedad a sus movimientos, direcciones o velocidades.

- **Interacción con el ecosistema**: Los depredadores y boids podrían interactuar con los animales del ecosistema de manera que los depredadores voladores pudieran cazar a las presas terrestres y que los depredadores terrestres pudieran cazar a las presas aéreas cuando estas se encuentren en tierra.

- **Reproducción**: Al igual que los animales del ecosistema se reproducen y se trasmiten genes, los boids y los depredadores podrían hacer lo mismo añadiendo así variabilidad a sus velocidades, rangos de visión o incluso los pesos correspondientes a cada norma. Sería interesante ver que parámetros son más interesantes para su supervivencia.

- **Uso de mapa con la GPU**: Como hemos explicado en el apartado *Boids*, cuando usamos los shaders para acceder a los recursos de la GPU, no usamos la subdivisión del mundo en regiones para optimizar la búsqueda porque no hemos sido capaces de conseguirlo aunque le dedicamos un tiempo considerable. En este sentido, podríamos seguir profundizando en el uso de los shaders para conseguirlo.

5.2.3. Ecosistema

El ecosistema es donde residen y interactúan los animales que hemos creado para simular el ecosistema. La única tarea que ha quedado pendiente es el sistema de toma de decisiones BDI, el resto de objetivos los hemos cumplido de manera positiva.

- **Sistema de toma de decisiones BDI**: Una de las tareas originales y que no hemos conseguido completar es la implementación de un sistema de toma de decisiones

BDI (Belif-Desire-Intentions) para los animales. Esto podría ser implementado sin problemas en el futuro ya que el único impedimento para no haberlo conseguido es la falta de horas.

- **Optimización de A***: Como hemos explicado en el capítulo *Ecosistema: Pathfinder*, hemos implementado el algoritmo A* para que las entidades puedan encontrar caminos a su objetivo siempre que Bresenham falle. Pero A* no es especialmente rápido lo cual hace que el sistema se ralentice tremendamente y soporte un número significativamente menor de agentes. Por ello decidimos no usarlo aunque esté implementado. Se podría trabajar en optimizar su uso con, por ejemplo, shaders para que si sea viable usarlo de manera natural.

- **Animaciones para los entes**: Dedicamos cierta cantidad de tiempo a aprender a usar el sistema de animaciones de Unity pero lo abandonamos para abordar tareas de mayor relevancia. Pero en el futuro podrían incluirse animaciones para los movimientos o las interacciones de los animales entre ellos para hacer el proyecto visualmente más agradable.

- **Estabilidad del ecosistema**: Como hemos explicado en el episodio *Simulaciones: Zorros-Conejos-Plantas*, no hemos conseguido simular un ecosistema con una población estable cuando participan zorros, conejos y plantas. Esto podría ser trabajado incluyendo nuevas características o modificando las existentes.

5.3. Valoración personal

En este proyecto hemos acabado trabajando en muchos apartados diferentes entre ellos pero con utilidades interrelacionadas. Hemos aprendido sobre **sistemas multiagente**, **inteligencia artificial**, **generación de ruido y de terreno** para crear el mundo, sobre **meshes**, sobre **Unity**, **shaders** para acceder a la GPU, **pathfinding** para que los agentes pudieran moverse, **Boids** para los animales voladores, **diseño de sistemas** para diseñar la arquitectura de los diferentes apartados del trabajo, **subdivisión de mapas** para mejorar la eficiencia y sobre **espirales de fibonacci** para la evasión de obstáculos de los boids entre otras cosas.

Al ser un trabajo de fin de grado propuesto por mi (el alumno) personalmente, el alcance y objetivos de este han ido cambiando ligeramente conforme avanzaba el proyecto. Pero hemos logrado la mayoría de las tareas que nos propusimos originalmente. Es el primer trabajo de tal magnitud que he realizado, 300 horas son bastantes pero el proyecto se empezó con tiempo más que de sobra para poder realizarlo con calma y aprovechando todas las sesiones al máximo ya que la capacidad de concentración suele bajar drásticamente tras unas tres o cuatro horas de trabajo seguidas. Teniendo esto en cuenta, el número de horas totales empleadas creo que han sido bien aprovechadas y han dado fruto a un resultado que, en mi opinión, considero bastante bueno. Aunque algunas partes del trabajo me han gustado y me han resultado significativamente más entretenidas que otras (el generador de terreno y los boids han resultado tremendamente entretenidos e interesantes de implementar), el trabajo en general ha sido una experiencia agradable y me ha proporcionado una nueva perspectiva sobre la cantidad de trabajo que se requiere para obtener esta clase de resultados. **En resumen**, ha sido un proyecto en el que he aprendido mucho y no me ha sido pesado.

Anexos

Bibliografía

[Axtell, 1996] Axtell, J. M. E. . R. L. (1996). *Growing Artificial Societies*.

[Bratman, 1987] Bratman, M. E. (1987). *Intention, Plans, and Practical Reason*. The Hume Series. CSLI.

[Caparrini, 2019] Caparrini, F. S. (2019). Sistemas multiagente y simulación. url: http://www.cs.us.es/ fsancho/?e=57.

[DeRose, 2005] DeRose, R. L. C. . T. (2005). Wavelet noise. url: https://graphics.pixar.com/library/waveletnoise/paper.pdf.

[Greco, 2012] Greco, G. (2012). Simulating an ant colony. url: https://terna.to.it/tesine/simulating$_a n_a nt_c olony.pdf$.

[Moncada,] Moncada, D. E. M. Modelo de toma de decisiones en agentes inteligentes, mejorando el esquema bdi. url: http://bdigital.unal.edu.co/51207/1/1053781414.2015.pdf.

[Nuno Barreto and Roque, 2014] Nuno Barreto, L. M. and Roque, L. (2014). Multiagent system architecture in orphibs ii. url: https://www.mitpressjournals.org/doi/pdf/10.1162/978-0-262-32621-6-ch095.

[Perlin, 1985] Perlin, K. (1985). An image synthesizer. url: https://www.cs.drexel.edu/ david/classes/papers/p287-perlin.pdf.

[Reynolds, 1987] Reynolds, C. W. (1987). *Flocks, Herds, and Schools: A Distributed Behavioral Model*.

[y Peter Norvig, 2008] y Peter Norvig, S. J. R. (2008). *INTELIGENCIA ARTIFICIAL UN ENFOQUE MODERNO. Segunda edición*.