Generación procedimental de ecosistemas y biomas y evolución inteligente de seres vivos en entornos naturales

Trabajo de fin de grado

Autores:

Georgi Mednikov

Eloy Moreno Cortijo

Daniel Cortijo Gamboa

Daniel González Cerdeiras

Andrés de la Cuesta López

Pablo Rodríguez-Bobada García-Muñoz

Supervisado por:

Carlos León Aznar

Universidad Complutense de Madrid

Facultad de Informática

Trabajo de Fin de Grado en Desarrollo de Videojuegos

Procedural generation of ecosystems and biomes, as well as intelligent evolution of living beings in natural environments

BCS Thesis

Authors:

Georgi Mednikov

Eloy Moreno Cortijo

Daniel Cortijo Gamboa

Daniel González Cerdeiras

Andrés de la Cuesta López

Pablo Rodríguez-Bobada García-Muñoz

Supervised by:

Carlos León Aznar

Universidad Complutense de Madrid

Computer Science Faculty

BSC in Game Development

Agradecimientos

Nos gustaría empezar agradeciendo el apoyo incondicional de nuestras familias y amigos, sin el cual este proyecto no hubiera sido posible.

Dar las gracias también a nuestro supervisor Carlos León Aznar por ayudarnos a realizarlo y aguantar nuestras estupideces. Gracias a él también por participar ocasionalmente en ellas.

Agradecer a Carlos Cervigón Ruckauer por proporcionar de forma desinteresada su opinión sobre el desarrollo.

Agradecer a aquellos que dedicaron su tiempo para realizar las pruebas de usuario.

Gracias a los gatos de Georgi por su capacidad de liderazgo y su apoyo en las largas sesiones invernales.

Resumen

Este es un estudio sobre la creación de un generador procedimental de personajes, más concretamente animales, que se pueden importar e interpretar como uno quiera, ya que genera resultados basados en datos. Los generadores procedimentales son uno de los mecanismos más convenientes a la hora de crear contenido en un proyecto informático, siempre que este lo permita, ya que reduce el esfuerzo humano a través de la automatización de tareas. Hacen referencia al conjunto de técnicas que se emplean principalmente en la industria del videojuego, y se utilizan para una pluralidad de objetivos, desde generar mapas o mundos virtuales hasta música con motivos dinámicos o incluso automatizar la generación de personajes.

Un animal es la consecuencia de un proceso de adaptación en su entorno, y por lo cual sus propiedades representan hasta cierto punto las características de este. Esto es un factor que se puede analizar y explotar. Dado un terreno con unas características, se pretende crear una serie de criaturas que interactúen entre sí y con el mundo que habitan, simulando la evolución para crear un ecosistema creíble, realista y sostenible. Además, para mostrar la veracidad de la simulación, se exporta el programa a un motor externo para demostrar el funcionamiento de la simulación y la utilidad de los datos resultantes.

El proyecto tiene una gran relación con la inteligencia artificial, y se basa en cuatro ramas del conocimiento principales: generación procedimental, para crear un mundo habitado; ingeniería de comportamientos, usando una máquina de estados y búsqueda de caminos para simular el comportamiento de las criaturas; programación evolutiva, para codificar la información de una criatura y realizar la reproducción entre estas y, finalmente, un cierto conocimiento del motor externo a utilizar como ejemplo para adaptar y mostrar los datos en dicho entorno.

Para evaluar el correcto funcionamiento de este programa se van a usar dos métodos: una prueba con usuarios en la que se pide a personas ajenas al proyecto que evalúan los datos resultantes en base a su experiencia y usar un sistema de telemetría basado en eventos que permite la recolección de datos para su posterior análisis, y de esta forma llegar a conclusiones que permitan la mejora y depurado del proyecto.

Abstract

This is a study about the creation of a procedural generator of characters, more specifically animals, which can be imported and interpreted however one wants, since it generates data based results. Procedural generators are one of the most convenient mechanisms available when it comes to creating content in a computer program , as long as it is reasonable within its context. This is due to this procedure reducing human labour through the automatization of tasks. The concept of procedural generators references the set of techniques used mainly in the videogame industry, and are used to accomplish a variety of goals, from generating maps or virtual worlds to music with dynamic motifs or the aforementioned generation of characters.

An animal is the consequence of a process of adaptation to its environment, and therefore its properties represent to a certain degree the characteristics of the latter. This is a factor that can be analyzed and exploited. Given a terrain with a certain set of characteristics, this study pretends to create a series of creatures that interact among themselves and with the world they inhabit, simulating the concept of evolution to create a believable, realistic, sustainable environment. Furthermore, in order to prove the veracity of the simulation, it is exported to an external engine to demonstrate the inner workings of the simulation and the utility of the resulting data.

This project is closely related to artificial intelligence and based around for main branches of knowledge: procedural generation, in order to create an inhabited world; behavioral engineering, using a state machine and pathfinding to simulate the behaviors of the creatures; evolutionary computing, to represent the creatures' genetic information and allow for realistic reproduction and, lastly, a certain amount of expertise related to the external engine chosen in which the resulting data will be displayed as an example.

In an effort to evaluate the correct performance of this program two methods will be used: a user based test in which people alien to the project evaluate the resulting data based on their experience with it and using a event telemetry system that allows for the collection of data for a posterior analysis, so that conclusions can be reached on how to improve the effectiveness of the project.

Índice

1. Introducción

La intención de este proyecto es crear y documentar el desarrollo de un generador procedimental de criaturas, que además simula un entorno dado un terreno, estableciendo de forma lógica un mundo virtual dotado de flora en la que se basa la existencia de la fauna. De esta manera, tras un cierto tiempo de simulación, se habrán creado criaturas adaptadas para sobrevivir, resultando una fauna capaz de vivir en armonía no solo con el entorno, sino también entre ellas, habiendo creado un ecosistema en equilibrio. Posteriormente, estas criaturas, exportadas como datos, pueden ser importadas y utilizadas para poblar entornos digitales, principalmente de videojuegos.

La generación procedimental es el proceso de creación de contenido basado en un algoritmo en lugar de imaginación humana directa. Es una herramienta frecuentemente usada en proyectos informáticos con unas ciertas características que permiten su uso, debido a las ventajas que presenta, en las que se indaga más adelante. Es particularmente destacable en la industria del videojuego, debido a que el contenido es uno de los factores claves que constituyen un videojuego. Además, suele apreciarse la rejugabilidad, es decir, que el juego sea capaz de entretener de manera distinta y prolongada a lo largo de distintas partidas. Esto es especialmente cierto con géneros concretos, en especial roguelike (juegos que cada partida generan un nivel único de manera aleatoria), porque se basan precisamente en la premisa de que cada partida es autocontenida. Como puede suponerse, una generación automatizada es superior a una manual en cuanto a velocidad y cantidad de recursos generados se refiere, volviendo la generación procedimental un proceso muy valorado entre los desarrolladores.

Esta herramienta tiene una gran cantidad de ventajas y a continuación se va a hablar de las más relevantes. En primer lugar, supone un gran ahorro de tiempo de desarrollo debido a la generación automática de contenido. Este a su vez es reemplazable, precisamente debido a su eficiencia. Si un resultado no es del todo deseado, se pueden producir nuevos hasta que uno sea satisfactorio. La variedad que genera este proceso a su vez genera rejugabilidad, debido a que el contenido se puede crear ipso facto entre una gran cantidad de posibilidades, este permanece impredecible más tiempo. Además, un generador procedimental es reutilizable entre proyectos o entre campos de contenido y a su vez consistente. Esto se debe a que al aplicar un algoritmo nunca se rompen las reglas establecidas en este. Los resultados son notablemente realistas cuando se usa para simular elementos reales, valga la redundancia, debido precisamente al punto anterior y puede generar contenido a escalas realmente grandes, como galaxias, si se desea. Por último, puede ayudar a superar limitaciones técnicas, por ejemplo, evitando el guardado del contenido y dejando que se genere durante ejecución.

Sin embargo, presenta una serie de limitaciones e inconvenientes, que de forma general tienden a relegarla al ámbito de los videojuegos o simulaciones: el contenido que genera está automatizado, al fin y al cabo, y debido a la falta de pulido o intencionalidad puede acabar no superando el aseguramiento de seguridad (*Quality Assurance*) establecido. Irónicamente, teniendo en cuenta que una de sus ventajas es el ahorro en tiempo, puede ser muy costoso si se intenta crear un algoritmo que demuestra ser elusivo, requiriendo una gran cantidad de modificaciones, y aún de esta forma puede no llegar nunca a alcanzar el nivel de efectividad deseado. Por otro lado, dedicar demasiados recursos al generador procedimental puede causar un déficit en otras áreas, creando una experiencia incompleta independientemente de lo eficaz que sea este. De la misma manera, un generador burdo puede crear experiencias puramente aleatorias, que resultan confusas y/o aburridas. Finalmente, no encaja bien con proyectos con una orientación muy específica, como novelas gráficas con un hincapié notable en la historia o dirección artística, o juegos competitivos, que necesitan estar cuidadosamente creados y equilibrados.

En resumen, los generadores procedimentales son una herramienta muy valiosa bajo las circunstancias apropiadas. Se puede generar desde música cuyo motivo varíe dependiendo de una o varias variables, hasta, por supuesto, mundos, niveles o personajes. Manteniéndonos en la industria de los videojuegos, uno de los que mejor ejemplifica este concepto es No Man's Sky, cuya premisa consiste en recorrer una galaxia prácticamente infinita, con sus incontables planetas y cada uno con su flora, fauna y características propias, todas creadas haciendo uso de un generador procedimental. De esta manera, se puede asegurar que no hay dos personas que hayan experimentado exactamente el mismo juego.

1.1. Motivación

Este proyecto pretende abarcar la generación procedural de un tipo de contenido específico, PNJs o "Personajes No Jugables", es decir, entidades sobre las que el jugador no tiene control de forma específica.

Cada proyecto tiene su propia dirección artística, y sus personajes sus propios rasgos característicos. Como consecuencia, en términos generales, tiende a no ser útil hacer un generador de personajes genérico porque cada desarrollador tiene sus propias necesidades. Es precisamente por este motivo que no se apunta a generar personajes como tal, sino animales. La diferencia fundamental es que unos requieren plasmar una personalidad a través de su diseño, mientras que las criaturas que habitan un mundo solo tienen la obligación de enriquecerlo con detalles. Por este motivo, precisamente, se hace que las criaturas evolucionen en un terreno con unas propiedades dadas; para que le den realismo y credibilidad a este, y en el acto, a sí mismas. El resultado del generador planteado está, además, basado en datos, lo que significa que se puede interpretar, por ejemplo físicamente, la criatura como uno quiera.

En resumen, se pretende programar un generador de un tipo que no existe de forma genérica que pueda servir para una variedad de finalidades, creando un ejemplo de un generador viable en una pluralidad de proyectos, capitalizando en el papel que tienen los

animales en un mundo, creando criaturas realistas pero dándole además la capacidad al usuario de hacer como plazca con los datos resultantes. El uso predominante de estos sería en el desarrollo de videojuegos, pudiendo usarse en juegos con una temática similar a la de No Man 's Sky (Hello Games, 2016), un juego de exploración espacial por planetas con ecosistemas y criaturas habitantes generados aleatoriamente.

1.2. Hipótesis del estudio

La premisa básica que se plantea en este proyecto es que, dado un terreno inicial, sea dado por el usuario o generado por el propio programa, se puede generar un mundo habitado por unas criaturas iniciales con un cromosoma propio para posteriormente realizar una simulación de la evolución sobre estas. Más concretamente, se quiere demostrar y ejemplificar el diseño y creación de un generador procedimental que crea y simula los elementos previamente mencionados para la exportación de la información a estructuras de datos útiles en proyectos externos. A su vez, se quiere que esta generación se vea como una herramienta viable a la hora de desarrollar contenido, y que los usuarios y desarrolladores vean potencial en un proceso como el previamente descrito. Se hace un especial hincapié en su uso en videojuegos, como método de llenar un mundo virtual con las criaturas desarrolladas de forma que resulte una simulación realista y coherente, a pesar de ser ajenas a este.

Como consecuencia, se establece como hipótesis que el sistema resultante va a permitir generar criaturas evolucionadas y adaptadas a un entorno. Estas podrán ser importadas en proyectos para su interpretación, como simulaciones y videojuegos, y poseerán características creíbles. De esta manera, los usuarios que interactúen con dicho proyecto percibirán animales verosímiles que proporcionen realismo y unidad a su entorno.

Dicha hipótesis implica la traducción de factores como comportamientos animales o la selección natural a un entorno simulado, a la vez que las funciones genéticas de cruce y mutación necesarias para la reproducción de las criaturas, dando lugar a nuevas generaciones con modificaciones respecto a las anteriores, permitiendo así una adaptación.

1.3. Objetivos

Los objetivos de este proyecto son los siguientes:

- Generación de un mundo físico con mapas de altura, humedad y temperatura, junto con una flora acorde con las propiedades del terreno.
- Sintetización y reproducción de los rasgos y características que definen un animal.
- Representación de dichas características en una estructura de datos, un cromosoma, basada en la programación evolutiva, junto a sus funciones de cruce y mutación, para simular la reproducción.

- Implementación de un sistema de navegación lógica de las criaturas por el terreno usando el algoritmo *A** con una heurística personalizada.
- Implementación de un sistema de interacciones entre una criatura y el terreno, flora y otras entidades del mundo.
- Simulación de comportamientos lógicos de las criaturas basados en suplir sus necesidades básicas, simulando sus ciclos vitales y aplicando así la selección natural.
- Implementación de un sistema que simule el paso del tiempo en el mundo, creando un ciclo día/noche y un sistema de turnos para las acciones de las criaturas.
- Exportación de criaturas a un formato genérico basado en datos para facilitar su uso. Exportación de la simulación a una biblioteca.
- Demostración ejemplar de una posible interpretación de los datos resultantes a un motor externo, al igual que la exportación de la simulación al susodicho motor.

1.4. Metodología

El primer punto a discutir es el entorno en el que se va a llevar a cabo el programa. Esto incluye tanto el lenguaje de programación a emplear como el entorno en el que se va a utilizar. Se barajaron cuatro lenguajes para este estudio: C++, C#, Java y Python. La primera opción es la más eficiente en cuanto a tiempo de ejecución. Sin embargo, representa una mayor inversión de tiempo de desarrollo, debido a ser el lenguaje de más bajo nivel entre las opciones barajadas. Python podría potencialmente reducir el tiempo de ejecución optimizando los cálculos matemáticos haciendo uso de Numpy, una librería extremadamente eficiente a la hora de realizar operaciones, pero es lento por naturaleza y algo ajeno a los integrantes del grupo. Por último, C# y Java son ambos lenguajes orientados a objetos en rasgos generales bastante similares, con la notable diferencia de que el grupo que desarrolla el proyecto tiene más experiencia y un mayor entendimiento de C#. Este tiene recolector de basura y pluralidad de librerías potencialmente útiles, lo que reduce en gran medida el coste de desarrollo. Además, es el lenguaje en el que se basa Unity, el motor que usaremos para el prototipo de demostración.

Teniendo esto en cuenta, se decidió desarrollar en C# por ser el punto intermedio entre tiempo de ejecución y desarrollo, además de ser el lenguaje que interpreta Unity, el motor sobre el que se va a representar gráficamente la simulación, por lo que así se ahorrarían posibles pasos de interpretación. Conociendo el lenguaje, el entorno de desarrollo se elige con facilidad, en este caso Visual Studio, porque todos los integrantes del grupo tienen acceso a él y están familiarizados con su funcionamiento.

C# podría presentar, sin embargo, una limitación en el desarrollo relacionada con las criaturas. Debido al funcionamiento de las referencias en este lenguaje, cuando una criatura tuviera que ser eliminada de la simulación, no sería borrada automáticamente

de la memoria mientras hubiera alguna otra referencia, como puede ser el caso de otra criatura que la recordara. Sin embargo, una criatura muerta debería a todos los efectos dejar de existir. C# proporciona formas de lidiar con problemas de esta naturaleza, como borrar datos manualmente sin depender del recolector de basura, pero esto parece una solución engorrosa, en un punto intermedio entre C++ y C#. Esto implica la implementación de un sistema de identificadores para las entidades del mundo, de manera que cuando una entidad quiera conseguir información sobre otra debiera primero intentar acceder con el ID de esta última, y en caso de que pereciera no se le otorgaran dichos datos.

El primer elemento necesario para comenzar el cumplimiento de los objetivos previamente descritos es la generación del mundo. Esta comenzará con una serie de datos básicos, que describen en la forma de una matriz bidimensional un mapa de alturas y masas de agua. La intención es que pueda darlo el usuario si así lo ve conveniente, pero que por defecto se cree uno aleatorio usando ruido de Perlin y una cierta cantidad de parámetros configurables para personalizar el resultado final de esta generación que, como será recurrente debido a la motivación del proyecto, podrán ser modificados por el usuario. Una vez se obtenga el mapa de alturas y masas de agua, este se utilizará para calcular de forma semialeatoria mapas de humedad y temperatura, que como último paso en la generación del terreno se empleará en la creación de una flora básica, el objetivo de varias interacciones básicas de las criaturas.

Una vez ya exista un terreno con sus propiedades y una flora con la que puedan interactuar las criaturas, el siguiente paso a dar será la generación de una genética propia para cada especie. Esto se representará a través de un cromosoma, una estructura que contiene toda la información que definirá cómo es una criatura y qué características tiene. En el caso de la primera generación de cada especie, simplemente se crearán cromosomas aleatorios, pues no habrá progenitores en los que se pueda basar la genética para crear a estos individuos. Estos serán los primigenios que evolucionarán para dar lugar a nuevas especies que se adaptarán a su entorno progresivamente mejor.

A continuación simplemente se crearán tantos de estos individuos pertenecientes a tantas especies como el usuario haya establecido y se distribuirán por el mapa. Todos los individuos de la misma especie aparecerán juntos en un primer momento para que tengan un punto de partida en común y sea más probable que puedan encontrarse en caso de necesitarlo, y las especies aparecerán medianamente distribuidas por el mundo para intentar abarcar todo el terreno generado y para evitar concentraciones de especies en puntos concretos, que podría dar lugar a extinciones casi inmediatas. La primera generación de cada especie compartirá un mismo cromosoma, ya que representa un punto común en el árbol filogenético de la supuesta evolución previa que las ha llevado hasta el punto presente. Como es lógico, en la población inicial de la simulación, para evitar que todas las criaturas de una misma especie tengan el mismo género, impidiendo la posibilidad de reproducirse, la mitad de las criaturas serán macho y la otra mitad hembra.

Mientras se ejecute la simulación, a cada especie se le asignará un nombre para poder identificarla y conservar de forma legible un historial, llevando la cuenta de las distintas

especies existentes y extintas y creando progresivamente sus respectivos árboles filogenéticos. De esta manera, se le dará al usuario la posibilidad de conocer la historia evolutiva de las criaturas, en caso de que le sea de utilidad.

Cuando la simulación termine, se exportarán una serie de datos: la información del mundo generado, por si se quisiera tener la información del entorno usado en la simulación; la información de las criaturas habiendo pasado por el proceso evolutivo, que luego se puede interpretar como uno vea conveniente para el proyecto en cuestión, y finalmente, los árboles filogenéticos previamente descritos.

Finalmente, para ejemplificar un posible uso de los datos resultantes y mostrar en directo la simulación en cuestión, se importará el código del programa y los datos a un motor externo, Unity por comodidad, para recrear tanto el mundo como las criaturas, representando varios de sus atributos en su aspecto físico y ejecutando todo esto en tiempo real. De esta forma se podrá ver tanto la simulación como que realmente se pueden interpretar los datos de las criaturas con libertad. Se va a usar la versión de Unity 2020.3.29f1. Para poder suplir de modelos tridimensionales a dicha recreación de la simulación se utilizará el programa Blender, en el cual se diseñarán todas las partes por las que puede estar compuesta una criatura.

Para depurar errores en el código, ya sean de ejecución o lógicos, se ha considerado pertinente un sistema de telemetría que genere eventos con información útil en partes clave del código. Posteriormente, estos datos se pueden procesar usando Python por lo accesible que es más una librería de renderizado gráfico, de las cuales hay bastantes, véase Plotly, para poder interpretar cómodamente la información y encontrar de esta manera fallos en la implementación del programa. Para facilitar la exposición dinámica de los datos, se ha planteado usar *Jupyter Notebook*, que permite ejecutar código por celdas y da de esta manera la posibilidad de sacar figuras actualizadas rápidamente.

Se utilizará Pivotal Tracker para la creación y organización de las tareas que han compuesto el proyecto, a la par que permite ser consciente de la velocidad de desarrollo. Se llevarán a cabo iteraciones semanales, tras las cuales se juzgará la validez de las historias y se escogerán las que van a ser realizadas la siguiente iteración. La duración de las historias se asignará siguiendo un sistema de puntos, que representaban una estimación del coste en tiempo de dicha tarea.

Para el control de versiones se utilizará un repositorio público en GitHub conteniendo tanto el proyecto de simulación como la adaptación a Unity. Se puede acceder al repositorio del estudio en el enlace: https://github.com/georgimednikov/TFG2122.

El desarrollo se ha planificado basado en el cronograma de la Tabla 1.1, teniendo en cuenta ciertos periodos de tiempo donde el equipo tendría menos tiempo por motivos académicos.

Fecha	Trabajo realizado
Mayo 2021	Concepción inicial de la idea, elaboración del grupo de trabajo y búsqueda de un director para el proyecto.
Noviembre 2021	Estudio sobre el estado del arte y las ramas del conocimiento relacionadas con el proyecto.
Diciembre 2021	Definición del funcionamiento y los detalles del programa y la organización posterior del desarrollo en Pivotal Tracker.
Enero 2022	Preparación para el desarrollo: Creación de proyectos de Visual Studio y Unity en un repositorio de GitHub.
Febrero 2022	Inicio del desarrollo: Implementación de la programación evolutiva relacionada con el proyecto y la creación de la generación del mundo. Combinar estas dos para generar criaturas primitivas en el mundo.
Marzo 2022	Creación de una máquina de estados que otorgue comportamientos a las criaturas y perfeccionamiento del contenido creado previamente según surjan errores o necesidades. Estructuras de importación y exportación de datos para utilizarlos en Unity, más una interpretación básica de estos.
Abril 2022	Perfeccionamiento de la simulación de criaturas en el proyecto base y de la representación de estas y sus acciones en Unity. Corrección de errores y comienzo de la memoria.
Mayo 2022	Pruebas de usuarios usando la representación en Unity. Finalización de la memoria y la corrección de errores restantes en el código.
Junio 2022	Publicación del proyecto y defensa del estudio.

Tabla 1.1. Cronograma del proyecto

1.5. Estructura del documento

El capítulo 1, como ya se ha visto, hace un resumen general del proyecto respondiendo las fundamentales preguntas de qué, cómo y por qué, además de añadir contexto a la idea y explicar resumidamente no solo el programa a realizar, sino también este documento.

El capítulo 2 sigue la línea de introducción, hablando sobre la situación actual de este tipo de proyectos en el mundo de la informática y ocasionalmente con un foco particular sobre la industria de los videojuegos, con el objetivo de dar más contexto todavía a la motivación y a la hipótesis previamente planteadas. A esto se le suma una discusión previa sobre el concepto actual de un ser vivo más algunas cuestiones sobre sus características para fijar unos cimientos en el estudio.

El capítulo 3 contiene el grueso del proyecto, describiendo todo el proceso de generación y simulación de un mundo habitado, y por lo cual se va a describir en mayor profundidad. Se ha intentado ordenar este capítulo de forma similar a la ejecución del programa. Como consecuencia, el subcapítulo 3.1 habla sobre la estructura del proyecto. A continuación, se comienza a describir la simulación en el subcapítulo 3.2, que detalla la creación del mundo y flora, describiendo las variables que influyen en este, cómo las puede configurar el usuario y cómo se emplean.

El subcapítulo 3.3 trata sobre las criaturas desde un punto genético. Al principio del mismo se hace hincapié en el aspecto evolutivo del proyecto, hablando sobre la implementación de la información genética, junto a su estructura y representación. Posteriormente, en el mismo, se trata la interpretación de estos datos y su traducción a cualidades e información que puede usar la máquina de estados de cada criatura, junto a un razonamiento explicando por qué seguimos el sistema que adoptamos.

El subcapítulo 3.4 trata la lógica de las criaturas durante su ciclo de vida, que incluye descripciones sobré cómo pierden y retienen información, navegan por el mundo e intentan cubrir sus necesidades a través de una máquina de estados que define su conducta.

El capítulo 4 hace referencia a la interacción del usuario con el programa: qué datos puede introducir y configurar para personalizar la simulación; cómo funciona y se ha perfeccionado la ejecución y finalmente se habla los datos resultantes y cómo exportarlos para emplearlos como el usuario vea conveniente. Este último concepto se ejemplifica en la adaptación a Unity, donde se describen los pasos dados para recrear visualmente las criaturas y el mundo, y la reproducción de la simulación del código en directo.

Una vez descrita toda la información relacionada con la creación y ejecución del programa se cierra el estudio con los siguientes capítulos: capítulo 5, que describe las pruebas realizadas para evaluar el funcionamiento del programa; el capítulo 6, que describe los resultados obtenidos en estas y su utilidad, y por último el capítulo 7, en el que se discute sobre la validez y la contribución del estudio al ámbito académico.

Finalmente, los últimos capítulos, 8 y 9, concluyen la memoria resumiendo las conclusiones del proyecto, posibles expansiones y mejoras, y la contribución individual de cada miembro integrante del grupo durante el desarrollo del estudio. Asimismo, una traducción de la información fundamental del estudio se encuentra al final del documento, en la forma de los capítulos 10 y 11, paralelos al 1 y 8 respectivamente, más la bibliografía y un anexo con documentos adicionales.

2. Estado del arte

Se tratan a grandes rasgos dos conceptos: uno es el proceso de evolución, y cómo se simula este proceso para la creación de las criaturas, y otro es la generación procedimental. A continuación vamos a hablar del estado del arte de cada uno.

2.1. Qué es un ser vivo

Dada la intención de este proyecto de representar, simular y evolucionar criaturas "vivas", el primer paso para el entendimiento sobre su funcionamiento y características es estudiar las distintas definiciones existentes sobre lo que es o no es un ser vivo u organismo.

2.1.1. Definiciones científicas de "organismo"

El mundo científico no ha sido capaz de trazar de forma unánime una línea que diferencie de forma precisa las entidades físicas carentes de vida de aquellas consideradas, por lo general, organismos. Siendo un concepto intrínseco de nuestra existencia y comprendido por todos, es sorprendente lo difícil que es de definir. Ya en el año 1970, Sagan catalogó, entre muchos intentos anteriores y sin duda posteriores, definiciones fisiológicas, metabólicas, bioquímicas, genéticas y termodinámicas (Sagan, 1970), todas con sus propios desafíos a la hora de establecerse como válidas.

Como primer ejemplo se va a hablar de la definición metabólica. El metabolismo se define como el conjunto de reacciones químicas y físicas que componen los procesos de la vida. Esta definición dicta que está vivo todo objeto que tiene límites definidos e intercambia constantemente con su medio parte de su materia, pero sin alterar sus propiedades generales, al menos durante algún periodo de tiempo. A estas reglas los retractores de la definición metabólica presentan, entre otros contraejemplos, el siguiente: una semilla, capaz como tienden a ser estas de permanecer décadas o incluso siglos sin intercambiar materia con su entorno, puede revivir cuando sus condiciones externas se vuelven más clementes para sus necesidades.

Por establecer un último ejemplo, la definición bioquímica se encuentra entre las más satisfactorias de las anteriormente mencionadas. Esta ve los organismos como sistemas que contienen información heredable y reproducible codificada en moléculas de ácido nucleico, y que metabolizan controlando la frecuencia de reacciones químicas usando catalizadores conocidos como enzimas. Existe, sin embargo, una enfermedad llamada

Tembladera, comparable a la de las vacas locas, causada por un prión, un agente con características virales, entre otras cosas, por no tener ácido nucleico propio. Estos agentes son capaces de usar el de las células del animal que habitan para reproducirse, cumpliendo así los requisitos sin llegar a considerarse un ser vivo.

Debido a lo amplia y variada que es la vida, y a que sólo conocemos y por lo cual únicamente evaluamos formas de vida terrestres, sufrimos de una restricción empírica a la hora de establecer reglas concretas que definan a una entidad como "viva". Sin embargo, se puede llegar a ciertas conclusiones aceptadas por regla general si se sintetiza lo máximo posible la expresión de la vida. De esta manera, Daniel E. Koshland define siete pilares fundamentales en los que se sustenta, según él, la vida (Koshland, 2002):

- Es un programa, un plan organizado que describe sus ingredientes necesarios y las interacciones necesarias entre ellos mientras el organismo persiste en el tiempo. Este programa está codificado en el ADN de los seres vivos, y es la información que los progenitores de una criatura le otorgan a su descendencia. Combinando la genética de ambos padres, que al pertenecer a la misma especie (u otra extremadamente similar) comparten un "programa" extremadamente parecido. Además, se producen pequeños cambios llamados mutaciones que permiten que los seres vivos puedan "improvisar", como se explica en el siguiente pilar.

- Es capaz de la improvisación, o en otras palabras, es capaz de modificar su programa cuando su entorno le presenta problemas. Debido a la pequeña fracción que representa un organismo en relación al mundo que habita, no tiene el poder o la capacidad de controlar su entorno, y por lo cual debe adaptarse. Si por ejemplo, un periodo cálido da lugar a una edad de hielo, modificando el entorno que habita la criatura y alterando sus propiedades, el sistema va a tener que sufrir cambios acordes para sobrevivir. Este pilar representa la evolución genética, y se explica en detalle en el apartado 2.1.2.

- Está compartimentalizado. Tiene un volumen concreto y está envuelto en una membrana que mantiene los ingredientes dentro y los químicos perjudiciales fuera. Esto se aplica a todos los niveles de complejidad, desde la unidad más pequeña de vida, la célula, con la membrana plasmática, hasta seres multicelulares, con piel, exoesqueletos, etc. Es a través de estas capas externas que se intercambian materiales con el exterior, introduciendo recursos en el sistema. Además, los seres complejos y de gran tamaño están especialmente compartimentalizados, creando una pluralidad de tipos de células especializadas en cumplir ciertas funciones, para posteriormente combinar varios tipos de células para formar órganos.

- Necesita energía, ya que es un sistema metabolizante. Según la física y la premisa fundamental de la definición termodinámica de la vida, los seres vivos son sistemas abiertos, y por lo cual intercambian energía y materia con el exterior. Necesitan hacer esto de forma constante para mantener la organización de su sistema, y lo logran a través de procesos biológicos basados en reacciones

químicas que se llevan a cabo haciendo uso de proteínas, enzimas, etc. estipuladas por el ADN de la criatura.

- Tiene la capacidad de regenerarse, debido a la pérdida de materiales a causa del metabolismo. Precisamente debido a que un organismo es, según la termodinámica, un sistema abierto, y la segunda ley dicta que "el calor va a fluir naturalmente de una reserva de energía a otra con una temperatura menor, pero no en el sentido opuesto sin ayuda externa" (Rajput, 1996) aumentando así la entropía. A causa de esto, las reacciones químicas que componen el metabolismo transforman recursos, y en el acto liberan energía inutilizable, calor, que aumenta la entropía general. Esta pérdida de energía se subsana reemplazando las substancias perdidas, y regenerando así el sistema. De la misma manera, las células de un ser vivo, por ejemplo, se gastan hasta ser inservibles, y la solución consiste en dejarlas morir para sustituirlas por nuevos individuos, reparando así el desgaste del sistema o regenerándose.

- Puede adaptarse sin necesidad de cambiar su programa, o lo que es lo mismo, adquirir y modificar comportamientos. Para esto, los seres vivos más complejos, por lo menos, cuentan con sistemas de recompensa y castigo. Cuando se da un evento perjudicial, una criatura sufre dolor y puede desarrollar miedo hacia la causa de su malestar, evitándose en el futuro. De la misma manera, comportamientos y eventos beneficiosos se recompensan con sensaciones positivas. Aunque es un comportamiento común de los seres vivos, se ejemplifica mejor a través de especies más desarrolladas, véanse los perros de Pavlov (Pavlov, 1929). En este experimento se tenía un grupo de perros a los que se les hacía sonar una campana antes de ser alimentados. Tras suficientes repeticiones, los perros comenzaban a salivar sin necesidad de ver u oler la comida. Esto demuestra que los animales pueden ser condicionados para reaccionar a un evento específico, creando un nuevo comportamiento en el proceso.

- Tiene formas de reclusión. Es esencial en un cuerpo metabolizante con una pluralidad de procesos diferentes tomando lugar en él que estos estén separados. De otra manera, los catalizadores de un proceso A podrían interactuar con los recursos B, generando compuestos no deseados y alterando gravemente ambos procesos. Para evitar acontecimientos como el anterior, las enzimas, un tipo de proteínas que aceleran las reacciones químicas, sólo interactúan con aquellas moléculas para las que se han diseñado. De esta manera, el "programa" tiene "programación defensiva", o un sistema que evita accidentes o descontroles debido a las interacciones de moléculas.

Una vez establecidos unos requerimientos fundamentales para que una entidad se considere viva, cualidades de las cuales carecen en mayor o menor medida las entidades inertes, la siguiente cuestión que surge tiene que ver con las diferencias en composición. Ambos casos están formados por materia inanimada: minerales, metales… Lo que los diferencia es la presencia de macromoléculas como ácidos nucleicos, péptidos, enzimas y hormonas entre otros. Todas estas están a su vez formadas por moléculas inorgánicas, pero en su conjunto cumplen funciones específicas, llevando a cabo procesos que

permiten las características descritas en los siete pilares. Los ácidos nucleicos contienen la información genética de la criatura, y hacen de "plano" a la hora de crearse la criatura, los péptidos transmiten información a las células, las enzimas tienen un papel en las reacciones químicas que componen los procesos biológicos, acelerándolos, etc. La presencia y relación de estas moléculas junto con aquellas inertes son lo que componen la forma de vida más básica, una célula, que luego permite la construcción de seres más complejos.

No todas las características previamente descritas ni todos los pilares pueden ser representados a través de la informática de forma viable, ya que hay restricciones tecnológicas en cuanto a lo que un ordenador es capaz de simular. Concretamente, el tercer y el séptimo pilares no podemos representarlos en este proyecto ya que hacen especial referencia a las células que componen una criatura, o a los orgánulos de estas células, elementos realmente complejos aunque diminutos que elevarían el coste de una simulación a niveles insostenibles. Haciendo estas dos excepciones, se propuso para este proyecto cumplir con todos los demás requisitos, con la restricción de diseño de representar únicamente organismos pluricelulares con unos comportamientos y tendencias lógicos.

Finalmente, todas las criaturas conocidas por el ser humano se encuentran categorizadas en base a sus características y datos genéticos en conjuntos. Este es el concepto de especie, "la segregación de la variabilidad genética total de la naturaleza en paquetes discretos, [...] que son separados unos de otros por barreras reproductivas, que previene la producción de un número demasiado elevado de combinaciones de genes incompatibles y discordantes. Este es el significado biológico básico de las especies y esta es la razón por la que hay discontinuidades entre especies simpátricas. Sabemos que los genotipos son sistemas epigenéticos extremadamente complejos. Hay límites severos en el número de variabilidad genética que puede darse en una solo población genética sin producir demasiadas combinaciones genéticas incompatibles" (Mayr, 1996). Son, en pocas palabras, grupos que engloban todos los individuos con unas características genéticas que los vuelven similares y capaces de reproducirse entre sí sin efectos perjudiciales o negativos en la descendencia, de ser viable la concepción. Según las especies se adaptan y evolucionan, proceso que se explica a continuación, estas se ramifican o extinguen. La documentación de la aparición y desaparición de las especies, junto con sus orígenes y descendencia suele representarse de forma visual a través de un esquema arborescente denominado árbol filogenético.

2.1.2. Principios neodarwinianos y la evolución genética

Es pertinente describir con detalle antes de continuar qué es la genética. La genética es la parte de la biología que trata de la herencia y de lo relacionado con ella, o dicho de otra manera, representa la información que un organismo le otorga a su descendencia.

Una vez definido qué compone un organismo y qué es la genética, falta explicar el paso posterior a la existencia de un ser vivo; la perpetuación de su información genética. Esto no quiere decir que su "programa", según las palabras de Koshland (Koshland, 2002), sea siempre el mismo; de hecho es imperativo que cambie para sobrevivir a cambios en

su entorno, véase el segundo pilar. Un elemento común en los seres vivos existentes es la voluntad de generar nuevas instancias de sí mismo para mantener el "programa" en marcha, ya que aquel que no muestra interés en este propósito muere y se extingue.

Es fundamental en este contexto comprender cómo se adapta la información de un organismo a su entorno, ya que para conseguir avances en la genética de un ser vivo, mejoras que le ayuden a proliferar; los cambios tienen que superar alguna especie de criterio, y eso es precisamente lo que lograron los principios de Darwin. Estos fueron en gran medida correctos, y dieron lugar, en combinación con el trabajo del científico contemporáneo Gregor Mendel, a las teorías de la evolución biológica y al concepto de la selección natural. Tras cierta actualización, se convirtieron en una rama del conocimiento de la biología fundamental en el entendimiento de los seres vivos, bajo una serie de nombres entre los que se encuentran "neodarwinismo" o "síntesis evolutiva moderna".

2.1.2.1. La selección natural

"… Las variaciones, por ligeras que sean y cualquiera que sea la causa de que procedan, si son en algún grado provechosas a los individuos de una especie en sus relaciones infinitamente complejas con otros seres orgánicos y con sus condiciones físicas de vida, tenderán a la conservación de estos individuos y serán, en general, heredadas por la descendencia. La descendencia también tendrá así mayor probabilidad de sobrevivir; pues de los muchos individuos que nacen periódicamente, sólo un pequeño número puede sobrevivir. Este principio, por el cual toda ligera variación, si es útil, se conserva, lo he denominado yo con el término de *selección natural*, a fin de señalar su relación con la facultad de selección del hombre" (Darwin, 1988).

La anterior cita es la definición del propio Darwin sobre el concepto que él nombró, y cuya denominación permanece a día de hoy, selección natural. Expresado de otro modo, Darwin está describiendo el criterio que sigue la vida para "seleccionar" a los individuos más aptos, o lo que es lo mismo, aquellos más capaces de sobrevivir y como consecuencia reproducirse, engendrando así más descendencia con las mismas características que hicieron aptos a sus progenitores. Para entender la anterior definición, Darwin fija en su obra cuatro principios fundamentales de la selección natural:

- Las características de la especie se pasan de generación a generación, es decir, la genética de los descendientes depende de sus progenitores, pues es en mayor parte la combinación de la genética de estos. De esta manera, el modelo de una especie se mantiene relativamente constante, a la vez que las características que volvieron aptos a los padres se pasa a sus hijos, volviéndolos aptos de la misma manera.

- Los individuos de una especie no son idénticos, a pesar de compartir gran parte de la información genética. A pesar de recibir la mayor parte de su información genética de sus progenitores, cada individuo sufre una combinación única de cambios que no depende de sus padres, mutaciones, otorgándole propiedades únicas. Cabe destacar que las mutaciones no son un proceso inteligente per se; ni la criatura ni ninguna otra entidad decide cómo mutar en base a sus necesidades.

Es un desarrollo completamente aleatorio, de ahí que haga falta un criterio o, como quien dice, un "filtro" que elimine las mutaciones negativas y esparza las positivas, como se explica a continuación.

- Cada generación nacen más individuos de los que son capaces de sobrevivir. Esto se debe a la competición entre organismos; el entorno que habita un ser vivo tiene recursos finitos, y por lo cual un individuo debe intentar cumplir con sus necesidades y hacerse con dichos recursos para asegurar su supervivencia, aunque esto sea perjudicial para la supervivencia de otros individuos. Hay varios tipos de selección, y se describen en mayor profundidad más adelante en este mismo capítulo.

- Solo se reproducen, y por lo cual ceden su genética, los supervivientes de dicha competición. Aquellos que no han conseguido los recursos necesarios para la supervivencia y reproducción no pueden generar descendencia, y por lo cual su genética se extingue con ellos. De esta manera, la población de una especie se vuelve más apta y capaz cada generación, tras la eliminación de los menos aptos dadas las características del entorno que los rodean. Además, los seres vivos tienden a generar más descendientes de los que pueden sobrevivir cuando logran reproducirse, asegurando normalmente tanto la competencia entre la nueva generación como la propagación de la información genética apta.

En resumen, cada organismo es único, y el propio medio en el que se encuentra determina si las mutaciones que le vuelven singular es beneficiosa o perjudicial, volviéndolo así apto o no apto. Por ejemplo, desarrollar un pelaje denso en un ambiente cálido suele ser contraproducente, y por lo cual una mutación negativa. Sin embargo, tener menos pelo que sus progenitores puede resultar útil, pues lidia mejor con el calor.

La selección natural viene dada por el entorno, tanto de forma directa debido a cambios en este, como a la interacción entre organismos en este debido a sus recursos finitos. Dada la investigación de Darwin, se categorizan tres tipos de selección natural, que varían en la forma en la que impactan la población de una especie.

- Selección estabilizadora: Hace referencia a aquellos criterios de selección que castigan valores extremos y recompensan los medios. Un ejemplo común de este caso es el tamaño de los bebés humanos. Un tamaño promedio no suele ser problemático de por sí, sin embargo un tamaño excesivo puede causar la muerte del bebé, la madre o ambos, sobre todo suponiendo un entorno natural previo a la existencia de la medicina moderna. En caso contrario, los bebés demasiado pequeños tienden a ser débiles y pueden estar subdesarrollados, aumentando la probabilidad de una muerte prematura.

- Selección direccional: El entorno recompensa los valores de un gen localizado en uno de los extremos, y por lo cual, cuanto más alejada se encuentre una criatura, menor será la probabilidad de que sobreviva y reproduzca. Ejemplos de este tipo de selección serían las jirafas, que con cuellos más largos consiguen más comida, o los animales con un color concreto que les camufla en su entorno, ya que cuanto

más te alejes de esta tonalidad más visible eres y por lo cual más probable que te vuelvas la presa de un depredador.

- Selección disruptiva: El criterio menos común, basado en que cualquier valor extremo es beneficioso, pero los valores medios perjudiciales. Un ejemplo del funcionamiento de este tipo de selección son las especies de pájaros que Darwin estudió en las Islas Galápagos. Todas provenían de una especie común, y desarrollaron características distintas para evitar competir por recursos: unos tenían picos grandes, para poder consumir semillas grandes, y otros picos pequeños para semillas pequeñas, es decir, se habían especializado. Sin embargo, los picos medianos resultaban poco útiles, y por lo cual esta característica era propensa a la extinción.

Cabe destacar que cuando se habla de "extremos" como variaciones beneficiosas se hace referencia a aquellas razonables. Un pico tan grande y pesado que dificulta el vuelo probaría ser contraproducente más veces de lo que es útil. Además, Darwin también habla de otro tipo de selección, la sexual. Esta se diferencia de las anteriores por no ser lo que Andersson categoriza de una selección ecológica, es decir, dictada por el entorno, sino que viene dada por los individuos de una especie (Andersson & Iwasa, 1995). Existen dos tipos de selección sexual: la intrasexual y la intersexual. En la primera, los miembros de un mismo sexo compiten entre sí por la posibilidad de reproducción. Un ejemplo de esto son los leones marinos, que luchan por este mismo motivo al entrar en celo, y el ganador consigue el privilegio de reproducirse con la mayor parte de hembras. En la intersexual, sin embargo, un sexo elige a su pareja, y es el otro sexo el que debe ser "persuasivo" para poder reproducirse, normalmente en galantes exhibiciones, como el plumaje de los pavos reales o los bailes de las Maratus volans, también llamadas arañas pavo real. Normalmente las hembras son las que limitan el acceso a la reproducción, debido a la carga que conlleva el embarazo, y debido a la selección intersexual se pueden dar diformismos sexuales, o diferencias físicas entre los individuos de ambos sexos de una especie.

2.1.2.2. La evolución biológica

El nombre de "evolución" no es uno que creara Darwin, ya que él hablaba en su célebre trabajo de "descendientes con modificaciones", pero ambos hacen referencia a las consecuentes modificaciones que sufren las especies debido a la selección natural. La generalización de una serie de mutaciones entre los integrantes de una especie se considera una evolución de la especie, y en casos donde una población o directamente todos los individuos de una especie evolucionan hasta diferenciarse lo suficiente de su especie original, dan lugar a lo que se categoriza como una nueva especie en sí, como ya se ha explicado. La evolución se da a varios niveles, dependiendo de la aptitud de la mutación y la cantidad de tiempo que ha transcurrido.

Primero ocurren lo que se llaman micro evoluciones, cambios en las poblaciones que modifican notablemente las características de los individuos cuando se comparan a las generaciones directamente anteriores de la especie, pero que no los vuelven irreconocibles, por así decirlo. Un ejemplo de este tipo de evolución sería el Pinzón de

Darwin, un grupo de especies de pájaros que estudió Darwin. Básicamente, no mucho tiempo atrás se introdujo una especie de pájaro a una isla en las Galápagos, y en no muchos años de esta habían surgido trece especies nuevas, todas con características distintas, sea en tamaño, forma de los picos… Con el objetivo de no competir entre sí por los limitados recursos, se especializaron, y aunque parecidas, todas son diferentes en comportamiento y rasgos en mayor o menor medida.

A una mayor escala se da lo que se conoce como macro evoluciones, evoluciones masivas a lo largo de grandes periodos de tiempo que modifican drásticamente los individuos, pasando por muchas especies intermedias. El ejemplo más evidente de esto sería la macro evolución que se dio cuando los primeros seres vivos, que eran acuáticos, se adaptaron a un entorno terrestre, desarrollando una variedad de rasgos desde pelo hasta alas. La macroevolución, que se entiende de forma más visual haciendo uso de un árbol filogenético, tiende a seguir unos patrones generales recurrentes (White et al., 2021):

- Estasis: Describe a los linajes o "ramas del árbol filogenético" que se mantienen relativamente constantes, y no sufren prácticamente cambios con el paso del tiempo. Los celacantos son un linaje de peces que hasta 1938 se consideraban extintos hace 80 millones de años, y sin embargo se redescubrieron individuos extremadamente similares en el océano índico. Por lo tanto, este tipo de peces lleva existiendo millones de años sin cambios significativos.

- Cambio en los caracteres: Hace referencia a cambios constantes en las características de una especie en el tiempo. Estos pueden darse más o menos rápido, o de forma más o menos dirigida,. Esto hace referencia a si la evolución tiende de forma constante a aumentar, por ejemplo, el número de costillas de un animal, o si lo hace de forma progresiva. Esto significaría hacer que con el paso del tiempo la especie tenga más costillas pero apareciendo en el proceso individuos con menos sin ser inmediatamente descartados.

- Formación de los linajes o especialización: Trata la velocidad a la que una rama del árbol filogenético da lugar a nuevas especies o linajes. Puede hacerlo de forma muy infrecuente, dando lugar a pocas especies a lo largo de un gran periodo de tiempo, de forma frecuente, generando nuevos linajes de forma constante, o de forma explosiva, formando muchas especies nuevas en muy poco tiempo.

- Extinción: Hace referencia a la muerte de todos los individuos de una especie, y por lo cual, la misma. Pueden darse extinciones masivas, como la edad de hielo, pero realmente es relativamente común la extinción de especies, ya que una ha podido dar lugar a otras especies más aptas, y por lo cual solo están desapareciendo los miembros no aptos, mientras que los evolucionados sobreviven. Por otro lado, las extinciones masivas acaban con gran parte de las especies vivas, pero esto a su vez genera cambios extremos en el ecosistema que pueden dar lugar a una mayor biodiversidad en el futuro. Véase la supremacía de los reptiles antes de la edad de hielo en contraste con la variedad de seres vivos que han existido desde entonces.

Un último concepto importante en la macroevolución es el que Goldschmidt acuñó con el nombre de "monstruo prometedor" (Goldschmidt, 2009). Hace referencia a individuos profundamente mutantes, que en vez de alterar levemente la genética heredada de sus progenitores hacen cambios muy notables. Estos cambios, si demuestran ser beneficiosos, pueden otorgarle una variedad genética a estos individuos que implica un gran potencial para establecer un nuevo linaje evolutivo. Esto se debe a la especialización saltacional, un concepto opuesto pero no excluyente del gradualismo darwinista. Significa que, con mutaciones lo suficientemente drásticas, una nueva especie puede saltarse un número de especies intermedias entre ésta y su padre. Esta tesis es controversial entre los científicos, pero hay ciertas líneas de evidencia que la dotan de credibilidad. Los registros fósiles proporcionan muy pocas pruebas de una macro evolución gradualista (Gould & Eldredge, 1993), y hay ciertos eventos que al gradualismo le cuesta explicar, por ejemplo, estructuras complejas como las alas que sólo proporcionan un beneficio cuando son funcionales, pues "de qué sirve media ala" (Gould, 1977). En este mismo documento argumentó que las alas, antes de permitir el vuelo, podían haber servido para regular la temperatura corporal, pero no todos los casos se puede explicar a través de "preadaptaciones". Presenta el caso de dos tipos de serpientes en la isla de Mauritius, Madagascar, que tienen el hueso maxilar de la mandíbula superior dividido en la parte frontal, y conectado con una articulación en la parte trasera, característica nunca vista en ningún otro tipo de vertebrado. "¿Cómo puede estar una mandíbula medio rota?" preguntaba Gould. El estudio de Günter Theiβen que revisa la viabilidad de la existencia de los "monstruos prometedores" contiene más información sobre el tema (Theiβen, 2005).

En resumen, la evolución es un proceso aleatorio, fundamentado en la selección natural suponiendo un entorno natural, que describe la propagación de los genes beneficiosos que aparecen en la población de una especie, logrando así una mejora de la aptitud de esta. Así, la vida se basa en un sistema que le ayuda a sobrevivir y proliferar en un entorno en constante movimiento, dando lugar a un ciclo de prueba, error y perfeccionamiento.

2.2. Simulación de seres vivos

La simulación de animales en el ámbito de la informática ha sido de gran interés para la ciencia desde que los avances tecnológicos han permitido tales ideas florecer. Desde aproximadamente la década de los 80, no han dejado de hacerse estudios sobre una pluralidad de aspectos de los seres vivos. Una de las principales motivaciones para este desarrollo ha sido la predicción de crecimientos o reducciones de poblaciones de animales, véase, por ejemplo, Whittemore (1983) para cerdos; Graham (1976) para ovejas o Oltjen (1986) para la producción de ternera y leche (Black, 2014). Estos modelos integran numerosas partes del sistema de un animal, con suficiente detalle en ocasiones para representar una estimación del potencial genético de la criatura para depositar proteínas o grasas o un entorno social además de climático. Estos programas siguieron evolucionando con el objetivo de lograr predicciones cada vez más acertadas, representando con una creciente precisión el funcionamiento de un organismo, añadiendo representaciones de enfermedades (Sandberg, 2006) y equilibrios

macrominerales y emisiones de gases invernaderos (Rigolot, 2010). Todos estos sistemas son teóricos, en el sentido de que los animales, aunque influenciados por una serie de factores presentes en los programas, carecen de comportamientos reales y se tratan como "máquinas"; sistemas que requieren unos ingredientes para generar unos resultados.

"La computación evolutiva es una rama de la inteligencia artificial en la cual se simula la evolución natural (basada en el principio de la 'supervivencia del más apto' de Charles Darwin) para resolver problemas complejos de búsqueda, optimización y aprendizaje" (Coello Coello, 2019). Simulando el funcionamiento de la selección natural, se pueden encontrar los valores más aptos dado un periodo de tiempo. Esta rama del conocimiento es fundamental para un proyecto con una motivación como la de este proyecto, ya que las herramientas que engloba este campo son las más adecuadas para representar la evolución de la vida artificial.

Un proyecto estrictamente relacionado con la programación evolutiva es ADAM (Pedersen et al., 2009), un programa que simula esquemas de cría selectiva en animales. Permite una gran cantidad de personalización por el usuario, a parte de extensiones, y es muy minucioso con la representación genética que emplea. El funcionamiento de ADAM consiste en la generación de una población base que da lugar a las descendencias posteriores. Estas comparten genética con sus progenitores y se generan cada paso temporal del programa. Los animales tienen un ciclo de vida dictado por el usuario, y la población se modifica según nacen y mueren nuevos individuos. A la hora de combinar la genética de los progenitores, implementan tres tipos de mezclas entre las que elegir: aleatoria, en la que se genera el cromosoma del hijo escogiendo un valor genético del padre o de la madre de forma arbitraria; trucando, es decir, generando el cromosoma del hijo combinando una parte del cromosoma del padre y otra de la madre, o eligiendo los genes óptimos en base a sus valores dados por la función de selección, la cual se explica más adelante. Para modelar y simular la información de la cría, usaron una simulación estocástica. Esto significa que los valores aleatorios que emplea la simulación dependen de uno o varios parámetros no aleatorios. De esta manera se consiguen valores impredecibles pero con una cierta relación entre ellos, lo cual vuelve el resultado más preciso. Finalmente, la información resultante es una memoria sobre las generaciones de animales y sus características y evoluciones, describiendo también los valores de endogamia estimados para cada indiviuo, entre otros datos.

2.2.1 Algoritmos genéticos

Cuando dos individuos de una especie generan un descendiente, este es un espécimen con una genética extremadamente similar a la de sus padres. Esto se debe a que es el resultado de la combinación de la información genética de sus progenitores. Este proceso se llama recombinación genética, y en la informática se representa a través de una función de cruce; un algoritmo que dicta de qué manera se mezclan los genes de los progenitores. Hay varias maneras convencionales de llevar a cabo este proceso. Suponiendo una representación binaria de un cromosoma, las funciones más comunes son (Gwiazda, 2006):

- Recombinación en un punto: Se crea un nuevo cromosoma y se va copiando en él el cromosoma de uno de los progenitores, y cuando se llega a un punto establecido de forma aleatoria, se empieza a copiar el cromosoma del otro progenitor hasta haber rellenado todo el cromosoma del hijo.

- Recombinación en x puntos: Sigue el mismo principio que la recombinación en un punto, pero en vez de copiar el segundo cromosoma hasta haber rellenado el cromosoma del hijo, lo copia hasta llegar a un segundo punto, momento en el cual se vuelve al primer cromosoma, repitiendo x veces.

- Corte y empalme: Al igual que en la recombinación en un punto, se escoge un punto aleatorio, pero esta vez el cromosoma no tiene un tamaño preestablecido. El cromosoma del hijo es el resultado de juntar la parte del cromosoma de un progenitor desde el comienzo hasta el punto establecido más la parte del segundo cromosoma del punto hasta el final, sin importar la longitud final.

- Recombinación uniforme: Dada una probabilidad de escoger el cromosoma de un progenitor u otro, se crea un nuevo cromosoma y se rellena cada uno de sus valores copiándolo del cromosoma del progenitor que salga al azar.

- Recombinación uniforme media: Mismo procedimiento que la recombinación uniforme estándar pero mirando cuántos valores son diferentes entre los cromosomas de los dos progenitores. El cromosoma del hijo es la copia de los valores coincidentes entre los padres, y la mitad de los que no coinciden se cogen de un progenitor y la otra mitad del otro, de forma aleatoria.

Una función de mutación, por otro lado, hace referencia al algoritmo genético que simula la mutación de los individuos, es decir, la desviación de la genética de sus padres, por así decirlo. De forma análoga a la función de cruce, se puede enfocar este proceso de maneras distintas, pero hay fundamentalmente dos maneras en un cromosoma binario de tamaño fijo (Obitko, 1998):

- Bit flip: Se recorre el cromosoma y de forma aleatoria, dada una probabilidad, se cambia un valor del cromosoma por su opuesto, es decir, un 1 por un 0 y viceversa.

- Uniforme: Se recorre el cromosoma y de forma aleatoria, dada una probabilidad, se cambia un valor del cromosoma por uno aleatorio, al ser binario, un 1 o un 0.

- Permutación: Se recorre el cromosoma cambiando el valor de una posición por el de otra aleatoria, dada una probabilidad.

Estos dos tipos de función se pueden implementar con cualquier algoritmo que uno desee, pero estas son las maneras más comunes y sencillas en el campo de la programación evolutiva, que además resultan ser relativamente efectivas, tanto en funcionamiento como en representación de sus análogas reales.

2.2.2 Vida Artificial

De este interés por la representación virtual de los seres vivos nace la vida artificial, un campo de estudio tecnológico centrado en la programación de entidades que imitan formas naturales de vida en un entorno dado, junto a sus procesos y evolución. C. G. Langton describió el concepto de la vida artificial como el estudio no sólo de "la vida tal y como la conocemos", sino también de "la vida tal y como podría ser" (Langton, 1995).

Esta biología sintética tiene un gran valor potencial en el ámbito científico. Existe una disciplina análoga llamada química sintética, cuyos principios son iguales pero se especializa en la creación de químicos y el estudio de sus reacciones en lugar de seres vivos. Gracias a esta rama de estudio, se pueden crear compuestos químicos que no se encuentran en la naturaleza siguiendo un procedimiento específico, pudiendo analizar sus propiedades y posibles usos. De esta manera se han creado una variedad de materiales cuyo descubrimiento hubiera sido problemático o imposible de otra manera, como electrodos de nanoalambres de plata, que desde su concepción en 2002 ha sido un tema de estudio importante en ciertos ámbitos científicos (Toro & Buriak, 2018). De la misma manera, la simulación de organismos puede dar lugar a descubrimientos de calibres similares, extendiendo los horizontes del estudio empírico de la biología más allá de la vida tal y como se conoce en la actualidad.

Uno de los primeros y más destacables ejemplos de vida artificial es *El Juego de la Vida* de John Horton Conway publicado en 1970 (Romero Dopico, s. f.). Este es un autómata celular, un modelo matemático para un sistema dinámico que evoluciona en pasos discretos. El espacio en el que se desarrolla consiste en una rejilla o matriz bidimensional, en el que se dan lugar las interacciones. Las celdas de la rejilla representan células, y pueden estar muertas o vivas. El programa empieza con un estado inicial llamado "semilla" y a continuación comienza la simulación, que consiste en la aplicación de las siguientes reglas de forma simultánea en el mundo: una célula muerta rodeada de tres células vivas pasa a estar viva; una célula viva rodeada de dos o tres células vivas se mantiene viva; si una célula viva tiene menos de dos o más de tres células vivas en su proximidad, muere. Este programa resulta realmente interesante debido a la representación que hace de la vida; la ejecución es caótica, casi impredecible, y sin embargo, dadas las simples reglas del juego, pueden darse infinidad de patrones con distintas características:

- Inmortales: Son configuraciones estables de células, que mientras no sean perturbadas por otro patrón, se mantienen sin modificación durante el paso del tiempo. Hay distinciones dependiendo de la forma que tomen, y subgrupos basados en si es un único grupo de células, varios grupos manteniendo su estado de forma conjunta o varios grupos que dependen entre ellos para mantenerse estables.

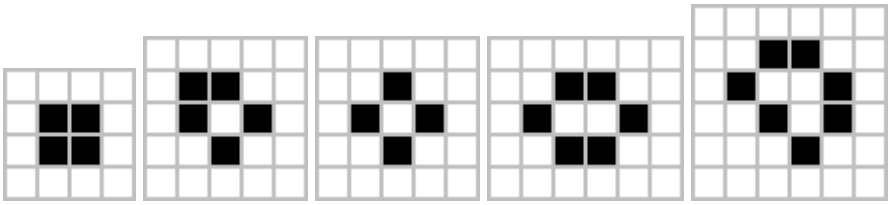


Figura 2.1. Patrones inmortales (El Juego de la Vida de Conway, 2022)
(https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)

- Astronaves: Son configuraciones que están en constante cambio y movimiento por el mundo. Mantienen un ciclo regular de muerte y nacimiento de células que las desplaza en una dirección dada. Pueden moverse a distintas velocidades en base al número de generaciones necesarias para avanzar un número de celdas.

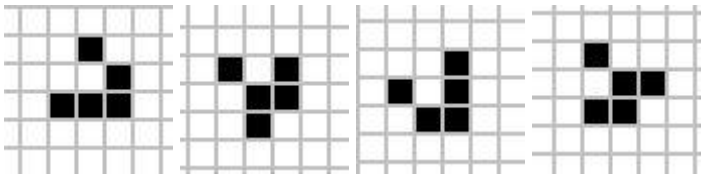


Figura 2.2. Astronaves (El Juego de la Vida de Conway, 2022)
(https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)

- Osciladores: Son fundamentalmente parecidos a los inmortales: se mantienen estáticos en su lugar de nacimiento mientras no sean perturbados. Sin embargo, no son estables, y ciclan entre una serie de estados generación a generación.

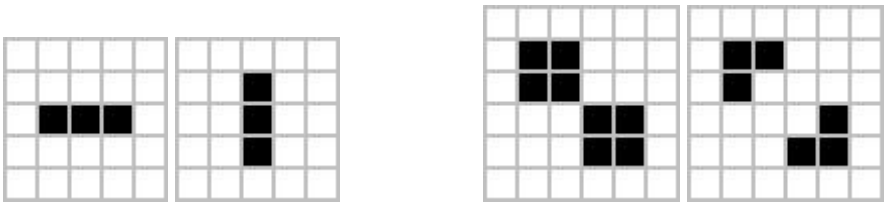


Figura 2.3. Osciladores (El Juego de la Vida de Conway, 2022)
(https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life)

- Otros: Existen infinidad de patrones, todos con características distintas, desde devolver una nave en la dirección contraria cuando esta colisiona con el patrón, hasta patrones de crecimiento indefinido, que están constantemente generando entidades como naves y por lo cual generando un crecimiento constante del número de células vivas.

En conclusión, el *Juego de la Vida* demuestra comportamientos emergentes y auto organizados en un entorno que podría denominarse como "caótico", aplicando un conjunto limitado de reglas sencillas. Esto puede verse como un análogo a la vida, haciendo una demostración simple pero efectiva de cómo, dado un entorno que lo permite, las entidades que lo habitan pueden dar lugar a patrones de forma autónoma con el paso del tiempo.

La vida artificial se enfrenta a una serie de desafíos, consecuentes de dos fuentes principales de limitaciones en el presente: la abrumadora complejidad que implica un organismo relativamente complejo y las limitaciones técnicas de la tecnología.

La primera restricción hace referencia a lo imposible que resulta en la actualidad representar con fidelidad un ser vivo multicelular. Una simulación realista de tal ser vivo implicaría la simulación simultánea de cada célula que lo componen y las interacciones que estas tienen entre sí. Esto ya es una tarea titánica, debido a la cantidad de tipos de células presentes en un cuerpo de un animal medio, pero hay que tener en cuenta que la biología puede llegar a alcanzar un nivel de complejidad que a día de hoy no se comprende plenamente, véase el cerebro, cuyo funcionamiento elude nuestro entendimiento. Recientemente, se ha representado por primera vez un genoma humano completo (Dunhill, 2022), lo cual deja ver lo lejos que estamos de poder realizar simulaciones a este nivel de detalle. También resultan problemáticos ciertos comportamientos que no son representables, ya sea porque no se sabe qué secuencia de pasos los componen o porque no se pueden describir con una especificación formal para que una máquina los ejecute (Langton, 1995).

El otro gran desafío es la limitación de la tecnología. Esta ha crecido en gran medida durante las últimas décadas, pero estos avances se están ralentizando o directamente deteniendo. Este crecimiento se debe en gran medida a la reducción de tamaño de los componentes de un ordenador, permitiendo la optimización de espacio, y con esta, la de potencia y rendimiento. Sin embargo, se está alcanzando el mínimo tamaño posible, y como consecuencia, se está estancando el crecimiento. Ahora mismo, el tamaño de un transistor ronda los 15 nanómetros, y se espera que lleguen a los 7 nanómetros aproximadamente, pero más allá de eso deja de considerarse un tamaño viable. Científicos del Laboratorio Nacional de Lawrence Berkeley han llegado a crear un transistor de 1 nanómetro de longitud, pero con dimensiones tan extremas, los efectos cuánticos pueden intervenir y hacer que estos dispositivos dejen de funcionar adecuadamente (Thompson, 2016). El siguiente paso sería construir un ordenador cuántico, la cual es una tecnología que no está desarrollada y no lo va a estar pronto.

Se va a discutir a continuación sobre tres ejemplos recientes de programas de vida artificial que aún siguen en desarrollo hasta la fecha de redacción de este documento.

El primero de estos programas es Aevol[1], una plataforma de genética digital con un intenso interés en la simulación realista de la información genética de individuos en una población y su evolución en base a un criterio de aptitud. Los organismos del programa compiten, se reproducen y mutan bajo las dinámicas darwinianas. Han usado como

[1] http://www.aevol.fr/

fundamento genómica de bacterias e investigación experimental de evolución microbiana, y como consecuencia su proyecto simula organismos a bajo nivel, sin inteligencia ni comportamientos autónomos. Sin embargo, han hecho una implementación realmente minuciosa y detallada de los cromosomas de las criaturas basándose en la programación evolutiva. La estructura genética de las criaturas no tiene un tamaño fijo, y se puede modificar a través de mutaciones no sólo los valores de los genes, sino también su longitud. Las funciones que un organismo es capaz de llevar a cabo, denominadas fenotipo, dependen de la interacción de las proteínas que lo componen, que a su vez depende de su genética. Comparando las habilidades fenotípicas de cada individuo con un conjunto de funciones arbitrario que componen la función de aptitud se consigue un valor de adaptabilidad, que define la probabilidad de que dicha criatura se reproduzca. Hay un proceso de "traducción" de genotipo, o el conjunto de genes del cromosoma, a fenotipo, pero debido a su complejidad y estricta relación con las especificaciones de Aevol, no se va a explicar en este documento. Se puede encontrar la información sobre esta transformación y muchos otros aspectos del proyecto en la página web Aevol. También detallan las fuerzas del programa, las cuales giran en torno a la flexibilidad de los cromosomas que crean y simulan, teniendo un tamaño dinámico, y la minuciosa representación y traducción de la información genética que contienen. Esto incluye los anteriormente mencionados fenotipos y proteínas hasta la pérdida de aptitud de un gen por tener material genético inútil, ya que ralentiza la evolución del organismo al darse con mayor probabilidad la posibilidad de que mute partes inservibles del cromosoma. Por otro lado, citan como debilidades de su programa las funciones biológicas que emplea el programa, ya que no representan con veracidad los entornos naturales de evolución de una criatura, y la simplicidad de la distribución de proteínas, ya que en los seres vivos reales se da una variedad de funciones entre estas que no es realista representar en un programa.

Aevol simula criaturas fundamentalmente sencillas, y sólo su genética. Sin embargo, el segundo ejemplo de vida artificial se centra en criaturas a más alto nivel, concretamente, en una especie de gusanos microscópicos llamada Caenorhabditis elegans. OpenWorm hace una simulación de un organismo multicelular incluyendo su sistema nervioso, con el objetivo de sentar unas bases que ayuden a entender sistemas complejos como el cerebro humano. El gusano en cuestión está compuesto por menos de mil células, y realiza operaciones vitales como alimentarse, reproducirse y evitar depredadores. Para lograrlo, han recopilado una gran cantidad de información sobre la especie a simular, sus características y comportamientos, y las han incorporado a sus modelos de software. Para el renderizado y la simulación "física" han usado sus propias plataformas llamadas Geppetto y Sibernetic. El funcionamiento del código no se explica como tal, aunque el proyecto es de código abierto, así que si uno quiere puede estudiarlo, pero hacen un gran énfasis en el sistema nervioso de la criatura. Se simula con músculos, que son estimulados por este, y se representan células con funciones concretas, como neuronas, que se basan en el modelo realista de Hodgkin Huxley. Más detalles en la página de OpenWorm[2].

[2] https://openworm.org/

Finalmente, a modo de último ejemplo y simulación de más alto nivel, se va a hablar de EcoSim, un simulador de ecosistemas. Pretende avanzar la comprensión de las dinámicas de los ecosistemas, para de esta manera proporcionar una vista global de sus comportamientos y ayudar a predecirlos. De todos los programas descritos hasta el momento, EcoSim es el primero que no solo simula poblaciones, sino también entidades únicas y discretas con características variables. De hecho, una gran parte de la atención de los desarrolladores se ha concentrado en representar las interacciones entre criaturas, con especial énfasis en la dinámica de cazador-presa. Para conseguir simular criaturas, crearon un mapa cognitivo basado en lógica difusa o fuzzy logic. Este tipo de lógica es una rama de la inteligencia artificial que "ayuda a los ordenadores a crear imágenes grises y con sentido común de un mundo incierto" (Kosko & Isaka, 1993). Esto significa que son capaces de responder a una pregunta con valores intermedios, no polarizados, de manera que en vez de decir "sí" o "no" pueden dar un valor en un espectro, normalmente entre 0 o 1, indicando la situación del estado en relación a dos extremos. Esto es razonable debido a los procesos biológicos de los seres vivos, que no pasan de un estado a otro de forma súbita, sino que es gradual, como el hambre o la sed. También se genera un árbol filogenético según aparecen y se extinguen las especies, manteniendo las relaciones entre estas.

La información resultante se divide en tres tipos de datos: estadísticas globales, como el número de miembros de una especie o los árboles filogenéticos; información relativa a las especies como tal, como su aptitud media; e información relativa a individuos del mundo, como su edad o nivel de energía. Más información sobre EcoSim[3] y las numerosas conclusiones a las que han llegado sobre los ecosistemas con el uso de este simulador en su página web.

2.3. Generación procedimental

Hasta donde se tiene constancia, entre los primeros generadores procedimentales se encuentra el célebre juego de rol de mesa Dungeons and Dragons, sobre el cual, en 1976, TSR Hobbies lanzó una expansión que permitía la creación de mazmorras aleatorias en base a una serie de reglas, siempre que el resultado fuera válido para el DM (el jugador que controla y crea los encuentros en las mazmorras). Funcionaba fundamentalmente de la misma manera que Rogue, uno de los videojuegos más influyentes de la industria, de manera que se creaban y juntaban habitaciones de una manera concreta para generar un nivel completo, distinto cada vez. Dicho juego fue tan influyente que dio lugar a un género, Roguelike, basado en la generación dinámica de los niveles.

RoboBlitz (2006) llevó el concepto a dos ámbitos simultáneamente: los gráficos y la optimización. Hizo uso de la generación procedimental para crear dinámicamente muchas de las texturas de los modelos, evitando así volverlas parte del juego y reduciendo drásticamente las dimensiones del juego, teniendo un tamaño final de 50 megabytes. Esta idea ya se había explorado previamente, cuando la memoria era

[3] https://sites.google.com/site/ecosimgroup/research/ecosystem-simulation

realmente limitada, en entregas como The Sentinel (1986), pero este solo lo aprovechaba en la generación de niveles.

Dwarf Fortress (2006) es destacable en este contexto debido a que cada partida genera de forma procedimental la totalidad del contenido externo a las mecánicas: desde el mundo con su terreno y edificaciones hasta las historias secundarias y la principal, junto con todos los personajes independientemente de la importancia que tengan. Dwarf Fortress genera una personalidad para cada personaje basándose en una serie de facetas psicológicas que el juego contempla, que se utilizan para calcular los principios morales, intereses, etc. Genera cientos de personajes notables en la geografía del mapa y les asigna acciones que realizan que pueden o no interactuar y afectar a otros personajes notables o incluso secundarios del mundo. En base a estos eventos, que cabe destacar que siguen sucediendo durante la partida, genera una historia que se le presenta al jugador como el hilo argumental principal del juego. Además, el juego guarda las interacciones del jugador con todas las entidades del juego, para poder adaptar los diálogos en base a sus acciones.

Cabe mencionar también Left 4 Dead (2008), un juego que consiste en derrotar oleadas de enemigos mientras se intenta cumplir con una serie de objetivos. Los desarrolladores crearon dos inteligencias artificiales llamadas “directores”. Estas no generaban recursos como tal, pero comparten la intencionalidad de crear de forma fluida y procedimental. Tenían, entre otras, las funciones de reproducir efectos de sonido o líneas de diálogo y modificar la música, los puntos de spawn y la cantidad de enemigos y recursos dependiendo de las acciones de los jugadores. También contaba con un sistema procedimental de animaciones, que dependiendo de los lugares de impacto en los enemigos al morir estos calculaban una animación realista, generando un “impacto” con repercusiones sobre el resto del cuerpo.

Finalmente, el ejemplo más notable y reciente es No Man's Sky (2016), un juego de temática espacial que genera el universo y todo lo contenido en este de forma dinámica según el jugador llega a nuevas zonas inexploradas. Desde las masas de agua y las criaturas hasta los planetas que habitan y las galaxias que estos forman están creados de esta manera, con un total estimado de casi 18 trillones y medio de posibles resultados. Afirman simular de la forma más fiel posible las fuerzas físicas que dictan el funcionamiento del universo hasta los comportamientos animales, que incluso tienen personalidades y preferencias por unos objetos sobre otros. No es de conocimiento público el funcionamiento de sus algoritmos, pero una estimación de los terrenos, por lo menos, podría ser la generación de estos en Minecraft (2011), que usa un generador de ruido de Perlin para generar alturas, y cualquiera de estas por debajo de un nivel está llena de agua. En base a las propiedades del terreno se fijan biomas, con unas características físicas y con una fauna y flora concretas. Con estos biomas, más algoritmos de generación de minerales en la tierra, por ejemplo, se van sustituyendo unos materiales del subsuelo por otros. Este proceso se hace chunk a chunk, fragmentos del terreno que compone el mundo. Cuando se llega a una zona que no se ha generado porque no se ha visitado, se repite el proceso con los chunks nuevos. De esta forma se genera el mundo progresivamente y siguiendo una serie de reglas concretas. Las

criaturas en ambos juegos, sin embargo, fueran o no creadas de antemano, tienen unas características arbitrarias en relación al mundo que habitan.

Existen también ejemplos fuera del ámbito de los videojuegos, que es el predominante, como en el cine, en el que ocasionalmente se usa este procedimiento para crear espacios visualmente interesantes haciendo uso de lo que se denomina en la industria como una "fábrica imperfecta", que hace pequeñas modificaciones a ciertos modelos originales para crear copias similares pero con distinciones, lo cual demuestra ser útil a la hora de generar realismo. Un ejemplo de este tipo de software es MASSIVE (Multiple Agent Simulation System in Virtual Environment), que se utiliza para generar el efecto de multitudes en el cine y la televisión.

Como se puede observar, la generación procedimental se fundamenta en el uso y creación de algoritmos basados en valores aleatorios. Esto se estudia en las siguientes secciones de este apartado.

2.3.1 Números aleatorios

Las técnicas usadas para esta generación dependen de la generación aleatoria de números (Barriga, 2019). Esto se debe a que, aunque es posible obtener resultados variados mediante un algoritmo determinista, la aleatoriedad otorgada por estos números garantiza que los resultados varíen en cada ejecución de manera más sencilla y consistente. Es por esta razón que la mayoría de procedimientos empleados en esta generación implementan independientemente generación aleatoria de números en su interior, o requieren que se les proporcione un número de manera externa, a partir del cual se realiza la generación.

La generación aleatoria de estos números es en realidad pseudo-aleatoria. Esto se debe a la propia naturaleza de los ordenadores, ya que son incapaces de generar valores verdaderamente aleatorios sin partir de un término sobre el cual operar. La mayoría de lenguajes de programación implementan métodos que permiten generar estos números, pero dependen de un elemento clave: la *seed*, o semilla: un número que procesan de tal manera que les permite obtener resultados aparentemente aleatorios. Por ejemplo, en el lenguaje C# (dotnet-bot, s. f.), esta *seed* es obtenida a partir de la fecha y hora del instante en el que se ejecuta el código, siendo pues distinta en cada generación posterior por la naturaleza lineal del tiempo. El hecho de que las mismas *seeds* produzcan siempre los mismos números puede ser y es normalmente aprovechado, sobre todo, por parte de los desarrolladores de videojuegos. Para poder replicar una misma generación basta con hacer que varios jugadores utilicen la misma *seed* en el algoritmo generador, proporcionándoles la misma experiencia a todos ellos.

Este sistema es usado en juegos basados enteramente en la generación procedimental, como los Roguelikes mencionados anteriormente, que permiten la introducción de la *seed* por parte del jugador para personalizar su propia experiencia. Sin embargo, los números usados en este caso, debido a la necesidad de ser manejados por usuarios, son obtenidos mediante el proceso de hashing. Este proceso consiste en una función que es capaz de transformar datos introducidos en otros de manera determinista. De esta

manera, el usuario puede fácilmente introducir frases simples que se transforman en el tipo de dato necesario para el proceso de generación.

La generación de números aleatorios se puede matizar de distintas formas dependiendo del resultado esperado. Los números más comúnmente generados son números de coma flotante entre 0 y 1; o números enteros entre 2 valores especificados. Además, a veces se hace uso de distintas fórmulas sobre estos números para simular una distribución de resultados Gaussiana frente a una lineal, lo que la aproxima a una variedad más natural. Si se pretende dar una mayor posibilidad a ciertos resultados se utiliza una distribución ponderada, en la cual los resultados posibles tiene un "peso" propio, que indica su posibilidad de ser elegidos frente al resto (por ejemplo, un número de peso 1 tendría la mitad de posibilidades que uno con peso 2). Estos pesos pueden venir dados por datos establecidos previamente, o ser obtenidos a partir del valor del resultado potencial en cuestión (Short & Adams, 2017a).

Los número aleatorios tienen numerosos usos, siendo uno de los más notables en el contexto de este proyecto la generación de mapas de altura, que se usan en la generación del terreno.

2.3.2 Mapas de alturas

Los mapas de alturas son un tipo de dato complejo que, haciendo uso de las técnicas discutidas anteriormente, genera una matriz bidimensional. Este procedimiento es crucial a la hora de generar terrenos en videojuegos o simulaciones, especialmente en aquellos tridimensionales. Cada celda de la matriz que compone un mapa contiene datos que varían gradualmente respecto a sus celdas contiguas, creando cierto suavizado entre celdas.

Para obtener esta variación gradual, uno de los algoritmos posibles es el box blur, normalmente usado para el procesado de imágenes. Consiste en un filtro formado por una matriz bidimensional. Esta matriz contiene una serie de números, que, junto a sus dimensiones, indican cuales de sus vecinos influyen en su valor, y con qué peso: el resultado final de la celda será la media de todos los valores de las celdas contiguas y la suya propia, teniendo cada valor el peso relativo especificado en la matriz del filtro. De esta manera se puede partir de un mapa de alturas con valores altos en los lugares que queramos que sean los picos de este mapa, y, mediante la aplicación de este filtro de manera continuada, obtener finalmente el resto del mapa por el cual se van extendiendo la influencia de estos picos (Short & Adams, 2017a).

La Figura 2.4 muestra un filtro simple, el cual asigna a una celda el valor de la media de los valores de todas las celdas a 1 de distancia y el suyo propio, con el mismo peso.

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}.$$

Figura 2.4. (*Box Blur*, 2021)
(https://en.wikipedia.org/w/index.php?title=Box_blur&oldid=1058880555)

Otro algoritmo para la generación de mapas de altura es el algoritmo de diamante cuadrado. Para el funcionamiento de este algoritmo es necesario tener una matriz cuadrada de un número impar de puntos o celdas, lo que puede limitar su uso en ciertas especificaciones. Este algoritmo, tras la asignación de valores en las esquinas de esta matriz, va alternando pasos de cuadrado y diamante: el paso del cuadrado encuentra cuadrados formados por celdas que hayan sido asignadas esquinas para el inicio del algoritmo (normalmente las esquinas de la matriz como tal), y asigna el valor de la celda que se encuentra en su centro a la media de los valores de estas esquinas más una variación aleatoria que se dicte. El paso de diamante realiza la misma acción, pero buscando formas de diamantes o rombos en lugar de cuadrados. Siguiendo estos pasos alternándose, el mapa termina siendo completado con valores derivados directamente de los valores de las esquinas iniciales.

La Figura 2.5 muestra el procedimiento del algoritmo en una matriz de tamaño 5x5.

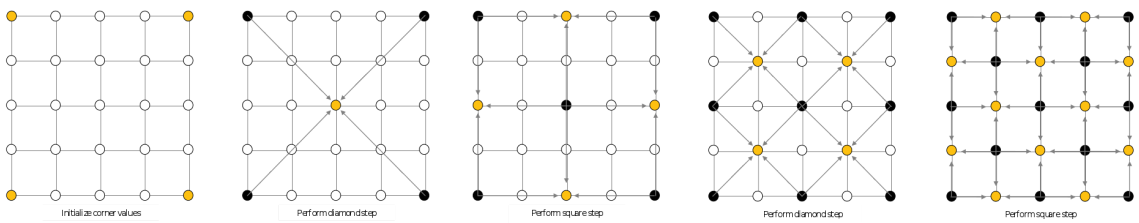


Figura 2.5. (*Diamond-Square Algorithm*, 2021)
(https://en.wikipedia.org/w/index.php?title=Box_blur&oldid=1058880555)

Algunos de los algoritmos más comúnmente utilizados para la generación de mapas de altura son el de ruido de Perlin, o el de ruido Simplex (siendo el último una versión revisada y mejorada del primero, creada por su mismo autor). El proceso de creación comienza con la asignación de vectores escalares de valor aleatorio a cada intersección de una cuadrícula de *n* cuadrados. Una vez asignados, para cada una de las intersecciones que forman parte de la misma celda que el vector candidato se crea un vector *offset*, que es el vector de desplazamiento entre el vector candidato y el punto evaluado (que puede ser normalizado para evitar que la influencia del candidato aumente con la distancia). Tras esto, se realiza el producto escalar del vector de cada intersección con su offset; y, por último, se realiza una interpolación entre estos puntos.

Esto resulta en una cuadrícula en la cual los valores cercanos a las intersecciones se aproximaba al de su gradiente y el del offset.

Al usar el ruido de Perlin, es común realizar varias pasadas del algoritmo sobre una misma matriz, dando lugar a resultados más interesantes. La Figura 2.6 muestra el resultado de varias pasadas de ruido de Perlin.

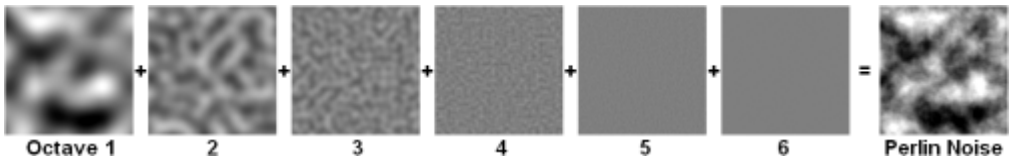


Figura 2.6. (*libnoise: Tutorial 4: Modifying the parameters of the noise module*, s. f.) (http://libnoise.sourceforge.net/tutorials/tutorial4.html)

2.3.3 Secuencias

Las secuencias son, como su nombre indica, sucesiones de datos de un tipo concreto. En lo que a la generación procedimental se refiere, los sistemas que generan secuencias producen, partiendo de una base aleatoria, una secuencia determinada mediante una serie de reglas de transformación previamente establecidas. Esta serie de elementos es una derivada de esta aleatoriedad inicial, y puede ser posteriormente interpretada de múltiples formas dependiendo de la necesidad; por ejemplo, interpretando cada elemento de la secuencia como un bloque específico de un conjunto disponible a la hora de crear un nivel lineal en un videojuego.

Uno de los sistemas principales de creación de secuencias es el Sistema-L. El sistema parte de unos elementos iniciales: las variables, que es el dominio de los valores que pueden existir en la secuencia; las constantes, que son aquellos elementos que no se ven afectados por ninguna regla; el valor inicial; y las reglas. Estas últimas son reglas "gramaticales" que se aplican a la cadena base, haciendo que los elementos de la cadena vayan transformándose hasta dar lugar a una cadena final. Este sistema es de naturaleza recursiva, pues los resultados obtenidos por las reglas son evaluados de nuevo por el sistema, produciendo una cadena nueva que es a su vez evaluada. La interpretación de la secuencia final depende del propósito de su creación, siendo una posibilidad la traducción de cada elemento del dominio de la misma a una acción o paso de un proceso, produciendo "pseudo programas" o funciones a partir de una entrada base.

Por ejemplo, para un sistema con variables A y B, ninguna constante, el inicio en A y las reglas (A→B) y (B→AB), las primeras iteraciones del sistema producirían las secuencias mostradas en la Figura 2.7.

n=0 : A
n=1 : B
n=2 : AB
n=3 : BAB
n=4 : ABBAB
n=5 : BABABBAB
n=6 : ABBABBABABBAB
n=7 : BABABBABABBABBABABBAB

Figura 2.7. (*Sistema-L*, 2022)
(https://es.wikipedia.org/w/index.php?title=Sistema-L&oldid=141801563)

La cadena de Márkov (Short & Adams, 2017a) es otro sistema de producción de secuencias utilizado en muchos ámbitos. Consiste en un grafo de nodos conectados unidireccionalmente entre ellos. Cada una de estas conexiones tiene asignada posibilidad de ser seleccionada. De esta manera, comenzando en un nodo específico (marcado como el primer término de la secuencia), se escoge mediante un número aleatorio cuál será la siguiente conexión recorrida. El nodo al final de esta conexión es añadido a la secuencia, y se repite el proceso hasta alcanzar una longitud de secuencia deseada o hasta llegar a un nodo designado como final de secuencia. Este modelo implica que la posibilidad de que se pase por un nodo depende sólamente del nodo anterior.

Es también posible generar una cadena de Márkov a partir de una secuencia ya existente, para facilitar la construcción de secuencias que sigan una lógica aproximadamente similar. Para hacer esto, se debe observar pares de elementos, anotando cuán comúnmente un elemento sigue a otro en la secuencia, lo que nos da los porcentajes de las conexiones entre elementos. Así, de una secuencia ABACAB podríamos deducir que del nodo A hay un 33% de posibilidades de ir al nodo C y un 66% de ir al nodo B, y que de ambos hay un 100% de posibilidades de ir al nodo A.

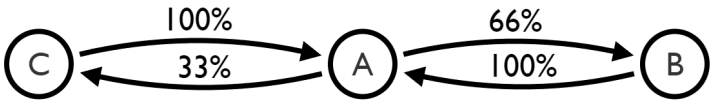


Figura 2.8. Cadena de Márkov

Las cadenas de Márkov pueden tener distintos números de orden (orden 1, orden 2), que indican la cantidad de elementos por nodo. En la figura anterior, esto cambiaría el dominio de los nodos a todas las posibles combinaciones de 2 letras. El hecho de que los nodos tan sólo dependan de su nodo anterior hace a las cadenas de Márkov ideales para un gran número de simulaciones. Una de sus aplicaciones es la meteorología: asumiendo

que el tiempo meteorológico en una región depende tan sólo de lo acontecido el día anterior, se pueden crear predicciones y modelos climatológicos sobre la misma.

2.3.4 Generación de espacio

La generación procedimental es usada frecuentemente para generar entornos o espacios usados posteriormente en simulaciones o videojuegos. Estos entornos son representados normalmente como matrices cuyas casillas contienen grupos de datos, lo que los hace estructuras complejas y costosas de producir de manera manual. Para poder facilitar su creación, se utilizan numerosos métodos que son capaces de generar conjuntos de datos usados en la misma.

Uno de estos métodos de generación es el de caminos aleatorios. Los caminos aleatorios son técnicas que comienzan en un punto concreto y van haciendo pasos con los que definen una forma o resultado final. En el contexto de la generación procedimental, podemos ver los caminos aleatorios como una secuencia de puntos en un espacio. Estos caminos pueden ser de varias dimensiones (Short & Adams, 2017a).

Los unidimensionales parten de un punto, e inicialmente y cada cierto intervalo de distancia avanzada, deciden si deben ascender o descender en el siguiente intervalo. De esta manera se produce una línea que baja o sube de manera aleatoria. Una posible interpretación de estos datos es la definición de relieve del terreno de un entorno bidimensional. Los bidimensionales parten igualmente de un punto, pero los pasos que pueden dar son hacia el norte, este, sur u oeste. De esta manera se genera una línea que gira en distintas direcciones a lo largo de su recorrido. Normalmente este tipo de caminos es generado repetidamente hasta dar con uno que cumpla los requisitos necesarios para ser usado en su contexto (por ejemplo, la creación de carreteras entre puntos de interés).

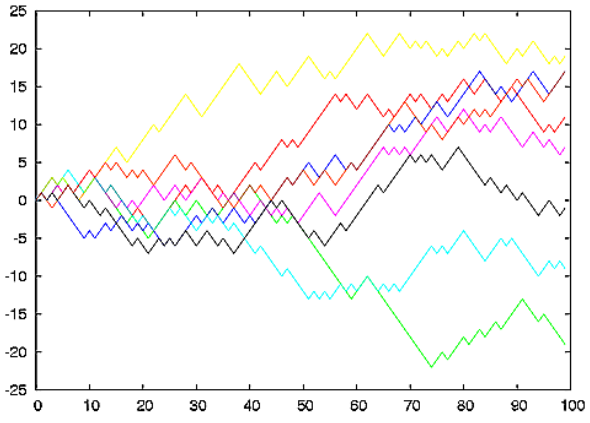


Figura 2.9. (*Camino aleatorio*, 2021)
(https://es.wikipedia.org/w/index.php?title=Camino_aleatorio&oldid=135029342)

Los autómatas celulares son otro de estos métodos auxiliares de generación: son un amplio grupo de sistemas que están formados por celdas discretas con estados definidos

y reglas que especifican cómo el estado de una celda es influido por el estado de las adyacentes como el *Juego de la Vida*, mencionado en el apartado 2.2.2. Partiendo pues de un estado inicial, se puede simular la evolución de los estados de las células a lo largo del tiempo. Estos autómatas no sólo pueden ser usados tanto como para simular la evolución y vida de poblaciones; también pueden generar estructuras como cuevas y mazmorras, o simular la propagación aleatoria de fuego sobre un terreno (van der Linden et al., 2014).

Otra manera de generar un espacio es el *asentamiento,* una técnica que utiliza la física para distribuir elementos de distinto tamaño en un espacio. A cada uno de los elementos se le da una forma y tamaño aleatorios, se posicionan todos los elementos en un espacio reducido, haciendo que haya solapamiento entre los mismos, y se deja que una simulación física simple haga que las formas se "empujen" entre sí hasta separarse por completo del resto y establecerse en un lugar del espacio. De esta manera, todos los elementos se distribuyen en este espacio de manera aleatoria. Esta técnica puede ser utilizada para formar estructuras como cuevas, interpretando las salas que las componen como dichos elementos, que se reparten y conectan mediante pasillos donde sea necesario (Short & Adams, 2017b).

Por otro lado, los azulejos Wang es un sistema usado para definir un conjunto de celdas y cómo se conectan sus lados en un plano. El concepto se basa en casillas con lados de un color específico, que debe coincidir con los de los lados de las casillas adyacentes. De esta manera se generan casillas de tal manera se puedan conectar de manera correcta, y se asigna un valor o elemento específico a cada color: por ejemplo, si se está generando un plano de un nivel, un color podría ser una conexión mediante una puerta y otro un pasillo amplio. Las casillas usadas por este algoritmo pueden haber sido creadas previamente, o generarse mientras el sistema rellena el plano.

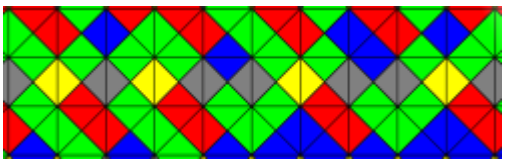


Figura 2.10. (*Azulejos Wang*, 2021)
(https://es.wikipedia.org/w/index.php?title=Azulejos_Wang&oldid=139368967)

Uno de los desafíos de este sistema es evitar la repetición de patrones en el plano, para lo cual se deben recurrir a sets de casillas específicamente generados para este propósito.

2.3.5 Fraccionamiento del espacio

En los espacios generados como entorno, es común necesitar dividir un espacio en regiones pequeñas, por ejemplo, para dividir un terreno en secciones para producir una malla de navegación en un videojuego. Para esto, existen numerosos procedimientos:

Uno de estos es la partición de espacio binaria, un método recursivo que divide las regiones de un espacio en 2 a partir de un punto aleatorio. Tras la primera división, se divide uno de los espacios resultantes en dos de nuevo, y se repite el proceso hasta llegar a una condición de parada. Esta técnica produce también una estructura en forma árbol que relaciona las distintas regiones y sus particiones: cada región creada es un nuevo nodo de un árbol, que, al ser dividida produce otros 2 nodos hijos, y así sucesivamente.

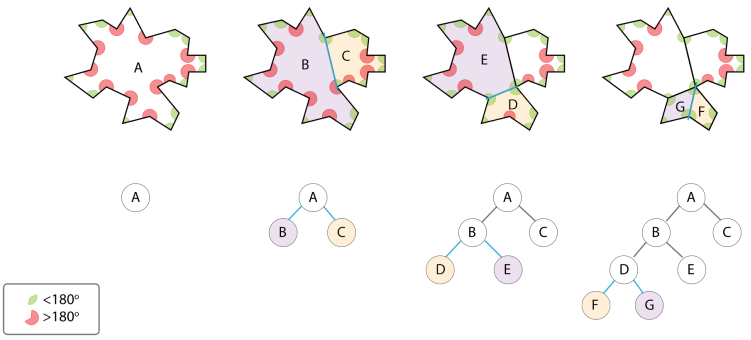


Figura 2.11. (*Binary Space Partitioning*, 2022) (https://en.wikipedia.org/w/index.php?title=Binary_space_partitioning&oldid=1080819490)

De esta manera se puede compartimentar un espacio ya generado, poniendo un punto de conexión simple entre cada una de las particiones producidas por el algoritmo. El videojuego *Caves of Qud* produce de esta manera la estructura interna de algunos lugares como ruinas (Short & Adams, 2017a).

Los diagramas de Voronoi son otra técnica de partición del espacio, que genera regiones a partir de puntos colocados aleatoriamente en un plano. Estas regiones tienen la propiedad de que los bordes entre dos puntos diferentes se encuentran en el punto medio entre ellas, por lo que cada punto dentro de un región se encuentra más cerca de su centro de región que de otros. Esta propiedad permite su uso en algoritmos de búsqueda de caminos permitiendo simplificar varios órdenes de magnitud la búsqueda. Para generar estos diagramas se usa el algoritmo de Fortune (Marbate & Gupta, 2013), pero el algoritmo no tiene en cuenta obstáculos en las condiciones iniciales.

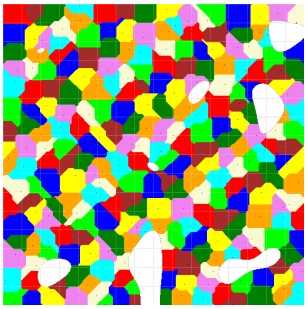


Figura 2.12. Zonas de diagrama de Voronoi

También existen los diagramas de Dijkstra, que funcionan de manera similar a los de Voronoi pero en un mapa de celdas discretas, donde el punto a la cual pertenece cada celda se comprueba haciendo la distancia de la misma hasta cada punto hasta encontrar el más corto.

Por último, otro procedimiento a destacar es el de los mapas de árbol, que son técnicas que dividen una zona rectangular en elementos rectangulares de distinto tamaño. Este tamaño es relativo a los datos de una lista de elementos que se quiere representar: a mayor sea la magnitud de un elemento respecto a otro, mayor será el tamaño de su zona en el resultado. Aunque normalmente suele ser utilizado a la hora de visualizar datos, esta técnica puede usarse en tándem con generación de datos aleatoria para crear la disposición de zonas en un espacio.

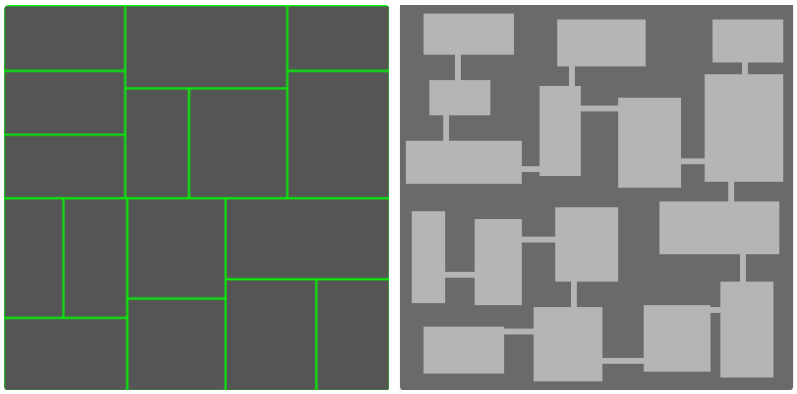


Figura 2.13. (Pierre, 2019)
(https://pvigier.github.io/2019/06/23/vagabond-dungeon-cave-generation.html)

Estas regiones creadas por los métodos anteriores pueden utilizarse en el contexto de crear escenarios basados en regiones o habitaciones, que resulta útil, como se puede suponer, a la hora de generar espacios de forma procedimental. Esto plantea la siguiente cuestión: dado un espacio cuya forma no es fija, las entidades que se sitúen en este tienen que ser capaces de navegar por ellos de forma coherente. Para esto existe una rama de la inteligencia artificial que engloba los algoritmos y métodos especializados en el cálculo de rutas óptimas, llamada búsqueda de caminos o *pathfinding* en inglés.

2.4. Búsquedas de caminos

La búsqueda de caminos es el proceso a través del cual se encuentran rutas desde un punto de inicio a uno destino en un entorno. Estas rutas son las óptimas, esquivando los obstáculos que se encuentran en el mismo. Estos obstáculos no tienen por qué ser entidades "físicas" en el mundo y pueden representar una dificultad abstracta, de forma que la ruta óptima en cuanto a distancia no es necesariamente la mejor en cuanto a deseabilidad. Por ejemplo, un sistema de navegación tiende a escoger el camino teniendo en cuenta el tráfico en cada carretera, y evitando así posiblemente el camino más corto si este implica un mayor tiempo de viaje. Esta búsqueda de caminos es relevante en una gran cantidad de casos, desde el planteado en el ejemplo anterior hasta el entorno de las simulaciones y videojuegos, en los que es fundamental que las entidades naveguen de forma coherente y dirigida por el mundo que habitan. De esta misma manera, dado el contexto y objetivo de este estudio, es necesario tener un conocimiento profundo sobre este concepto para poder recrear comportamientos lógicos en las criaturas que se generan.

El coste de calcular un camino es alto, por lo tanto es preferible minimizar la longitud del mismo o hacer búsquedas menos costosas. Si no se tiene en cuenta los obstáculos que pueden haber entre dos puntos, o no hay obstáculos, se puede minimizar el coste si la búsqueda del camino consiste en generar una línea recta hasta el objetivo. El coste de generar una línea recta tiene un coste $O(N)$, siendo N el número de nodos que hay entre el inicio y el fin del camino; mientras que una búsqueda de un camino tiene un coste $O(E * log(V))$ o superior, siendo V el número de vértices y E el número de aristas entre estos vértices. Sin embargo, generar una línea recta solamente es posible si el entorno consiste en una matriz de posiciones discretas, mientras que la búsqueda de caminos sirve para cualquier entorno que se pueda representar como un grafo, cosa que es posible con una matriz. El algoritmo más común de generación de líneas rectas en una matriz es el algoritmo de Bresenham (Bresenham, 1965). Es un algoritmo de error incremental, que consiste en avanzar desde el punto de inicio hasta el final de forma discreta en un eje mientras que en los otros se avanza una fracción del avance en el primer eje. De esta manera, solamente se cambia de unidad en los segundos ejes cuando el error acumulado supere la unidad, y así se genera una línea sin "huecos". Este algoritmo funciona solamente en el octante primero, por lo que antes de realizarlo, hay que transformar los incrementos al primer octante. La línea generada se puede utilizar finalmente como camino.

Cuando se tienen que tener en cuenta los obstáculos del entorno o el entorno es más complejo que una matriz de posiciones discretas, es necesario realizar una búsqueda de caminos. Estos caminos se basan en grafos de vértices conectados por aristas y su recorrido. El recorrido de este grafo es lo que diferencia los distintos algoritmos y sus costes. Un algoritmo ejemplar es el algoritmo de Dijkstra (Brezzi et al., 1959), el cual consiste en, partiendo de un nodo de inicio, calcular el camino más corto al resto de nodos del grafo. El coste de cada vértice es un número positivo. Al principio pone a todos los vértices una distancia de infinito, excepto al vértice inicial, al que se pone una distancia de 0. Después, hasta encontrar el vértice objetivo, se selecciona el vértice no seleccionado con la menor distancia y se consideran sus adyacentes, poniendo a cada

uno de distancia la que tiene el vértice considerado más el coste de la arista que los conecta. Se marca como visitado el vértice y se repite el proceso. Este algoritmo utiliza la programación dinámica para eliminar caminos que nunca serán los óptimos, y es un algoritmo general para cualquier grafo. Sin embargo, el coste del cálculo se puede reducir si se conocen características del entorno a priori.

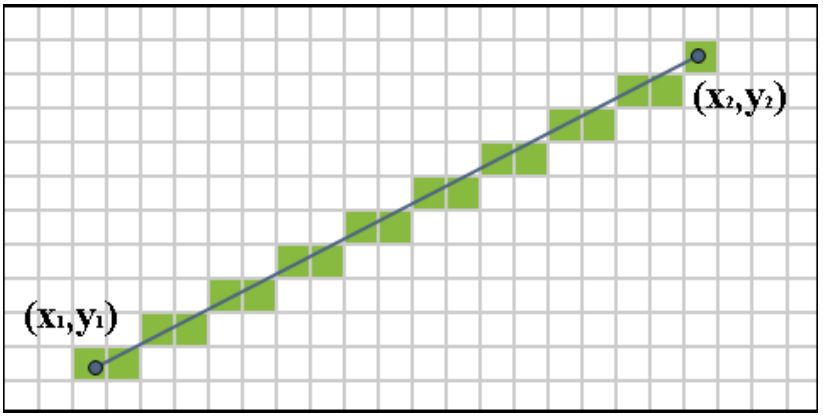


Figura 2.14. (*Algoritmo de gráfico de segmento de Bresenham*, 2005) (https://es.frwiki.wiki/wiki/Algorithme_de_trac%C3%A9_de_segment_de_Bresenham)

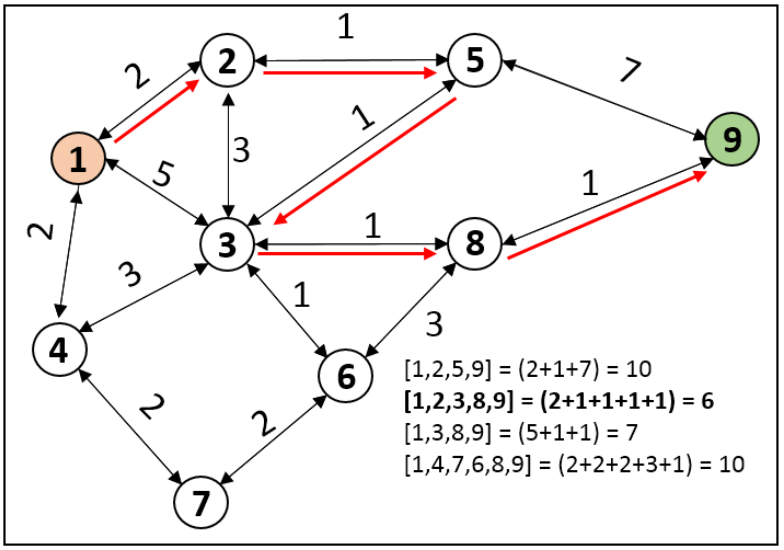


Figura 2.15. (*Ilustración del algoritmo de Dijkstra*, 2019) (https://www.researchgate.net/figure/Illustration-of-Dijkstras-algorithm_fig1_331484960)

Utilizando heurísticas se puede reducir el número de vértices que se exploran a la hora de calcular un camino. Una heurística es una aproximación a la solución utilizada cuando el coste de averiguar la solución completa es alto. Dependiendo de lo precisa que

sea la heurística se pueden descartar más vértices, y la precisión depende de la naturaleza del problema. Un algoritmo que utiliza heurísticas para calcular caminos es *A** (Russell et al., 2020). Este algoritmo asigna a cada vértice el coste mínimo para llegar al mismo desde un vértice inicial y el coste aproximado que devuelve la heurística. Este coste debe ser optimista: no debe devolver un coste mayor que el que devolvería el camino óptimo. Al principio, asigna a todos los vértices un coste infinito y al inicial un coste 0. El algoritmo consiste en seleccionar el vértice con menor coste asignado, asignar a los vértices adyacentes al vértice actual el coste descrito anteriormente, y marcar el mismo como visitado; repitiendo hasta que se llegue al vértice destino. Dos heurísticas utilizadas comúnmente con *A** son la heurística de Manhattan y la Euclídea, ambas se calculan en base a la distancia, utilizando las Ecuaciones 2.1 y 2.2, siendo el punto (x_1, y_1) el punto de inicio y (x_2, y_2) el punto final.

$$d_{Manhattan} = |x_2 - x_1| + |y_2 - y_1|$$

Ecuación 2.1. Distancia Manhattan

$$d_{Euclídea} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Ecuación 2.2. Distancia Euclídea

Estos métodos y algoritmos son extremadamente eficientes, pero cuentan con la limitación de estar diseñados para su aplicación en entornos bidimensionales con valores discretos, como la matriz de valores de la Figura 2.16. Para solventar este problema, se han ideado una serie de soluciones en espacios 3D con valores continuos, o lo que es lo mismo, no basado en casillas. El más popular se podría decir que son las *NavMeshes* (Cui & Shi, 2010), del inglés "malla de navegación", que consiste en la división del terreno por el cual las entidades pueden moverse en secciones poligonales. Luego, aplicando *A**, se conecta la sección en la que comienza la entidad que quiere moverse con la sección en la que se encuentra su destino, como se haría sobre una matriz pero inventando las celdas, por así decirlo, generando así un grafo con aristas. Una vez se ha encontrado la secuencia óptima de secciones que conectan el punto de origen con el de destino, se navega dentro de estas buscando la ruta más corta, que es en línea recta siempre que sea posible, y hacia la esquina más lejana visible en caso de haber obstáculos. Un resultado ejemplar de este método se muestra en la Figura 2.17.

7	6	5	6	7	8	9	10	11		19	20	21	22
6	5	4	5	6	7	8	9	10		18	19	20	21
5	4	3	4	5	6	7	8	9		17	18	19	20
4	3	2	3	4	5	6	7	8		16	17	18	19
3	2	1	2	3	4	5	6	7		15	16	17	18
2	1	0	1	2	3	4	5	6		14	15	16	17
3	2	1	2	3	4	5	6	7		13	14	15	16
4	3	2	3	4	5	6	7	8		12	13	14	15
5	4	3	4	5	6	7	8	9	10	11	12	13	14
6	5	4	5	6	7	8	9	10	11	12	13	14	15

Figura 2.16. (Algoritmo de búsqueda A*, 2022)
(https://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*)

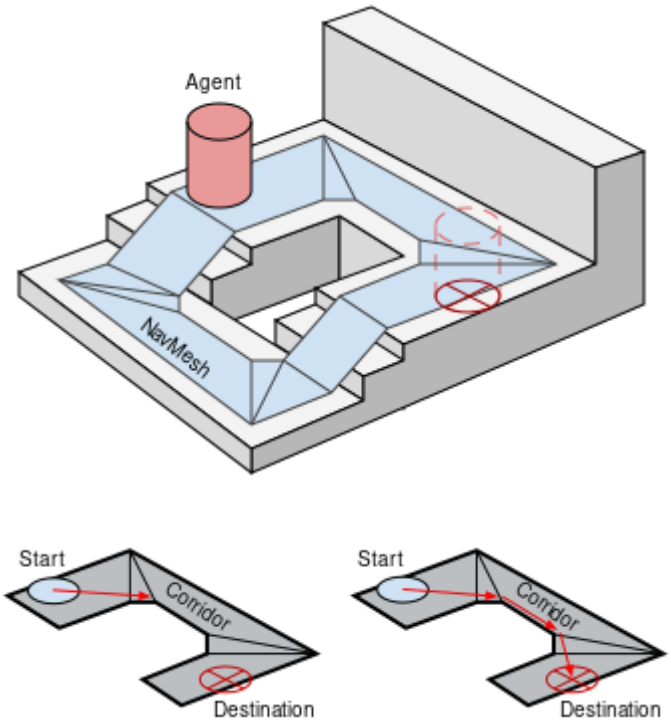


Figura 2.17. Demostración gráfica de una *NavMesh* (Unity, 2022)
(https://docs.unity3d.com/Manual/nav-InnerWorkings.html)

3. Creación y simulación de un mundo virtual habitado

Este capítulo se centra en describir cada uno de los elementos que participan en la creación y subsequente simulación del mundo sobre la que se basa el estudio: La generación de un terreno básico con características físicas, su población por parte de elementos estáticos, y por último, la creación de las criaturas. En estas se profundizará, comenzando con la descripción de la implementación interna desarrollada en base a la programación evolutiva, junto a la interpretación de datos genéticos en información útil para la simulación. Posteriormente se discute los comportamientos deseados en las criaturas, y cómo se ha creado un sistema que intenta recrearlas en el mundo virtual. También se habla sobre la interacción de estas con el mundo y lo que esto implica para su supervivencia y reproducción.

3.1. Estructura y detalles relacionados con los proyectos de la simulación

Para la implementación del proyecto hubo que tomar una serie de decisiones relacionadas con el desarrollo. El lenguaje de programación junto al entorno de desarrollo son dos factores que se han discutido con anterioridad en el apartado 1.5.

En el repositorio hay dos proyectos principales: el prototipo de demostración de la simulación en tiempo real en Unity, concepto en el que se profundiza en el apartado 4.2, y la solución de Visual Studio, que contiene todos los proyectos relacionados con la simulación base, desde el código fuente hasta el sistema de eventos usado para su depuración (véase el apartado 5):

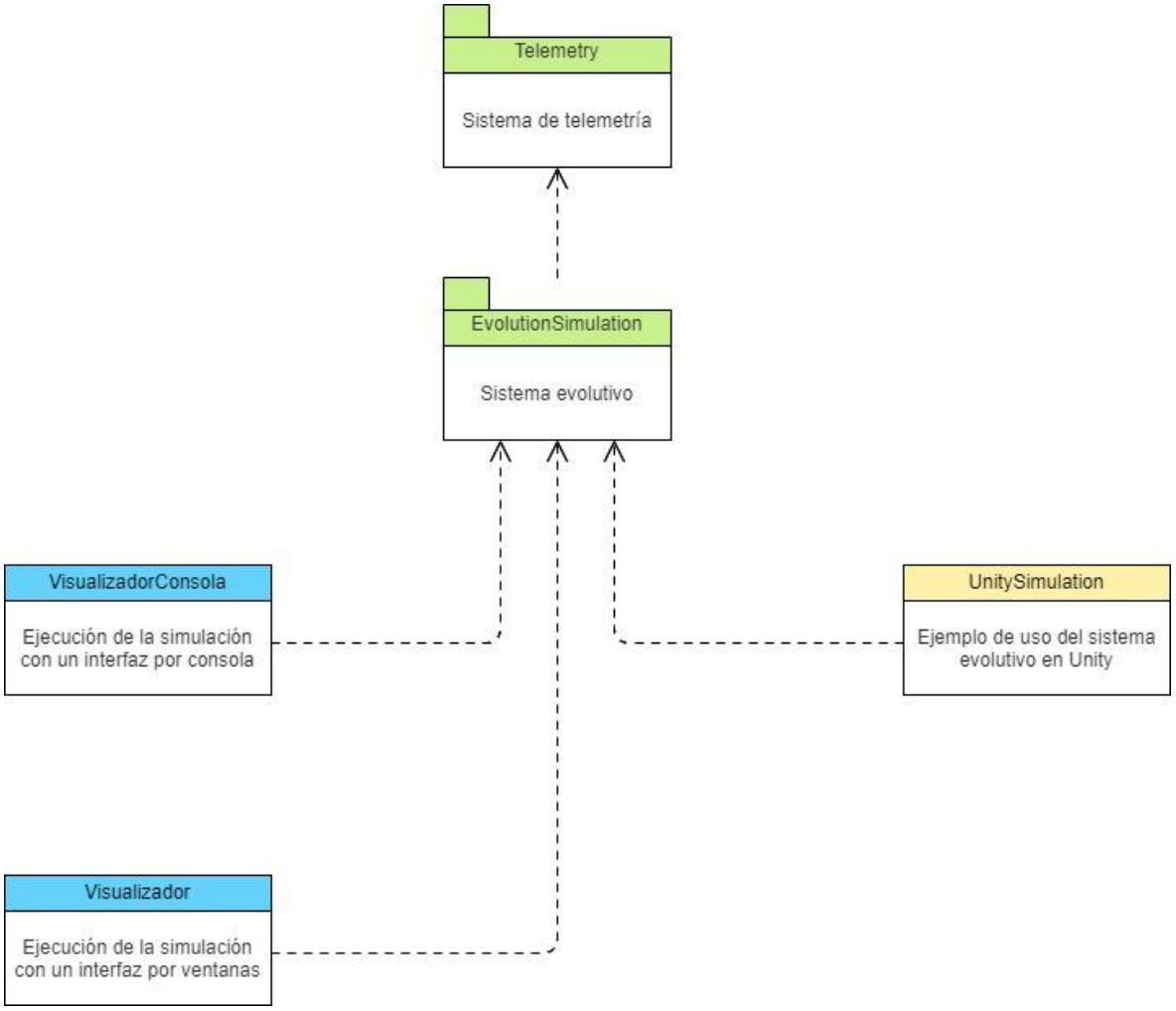


Figura 3.1. Diagrama de proyectos (obtenido a través de *Visual Paradigm Online Free Edition*)

- En primer lugar, *EvolutionSimulation* genera una librería dinámica que contiene todo el código relativo a la generación y simulación del mundo y las criaturas. El punto de entrada al código de la simulación está en la estructura llamada *Simulation*, que se encarga de inicializar, ejecutar y exportar la simulación. Ésta contiene el mundo de la simulación, que a su vez contiene las entidades que lo conforman y el mapa en el que se mueven. Para facilitar el acceso a la información de la simulación, mientras se está ejecutando la misma; existe una estructura *EventSimulation*, que permite la suscripción a los eventos que se producen en la misma.

- Los proyectos *VisualizadorConsola*, *Visualizador* y *UnitySimulation* dependen de *EvolutionSimulation*, ya que utilizan la librería dinámica generada por éste proyecto. Como el proyecto de Unity es externo, tras la compilación de *EvolutionSimulation*, se copia la DLL generada en el proyecto de Unity. Ésto se

hizo para agilizar el desarrollo, ya que ambos proyectos se realizaron en paralelo. Además, éste proyecto utiliza el sistema de telemetría con el fin de evaluar su funcionamiento. Hace uso de una constante de pre-compilación (*TRACKER_ENABLED*) para habilitar la recogida de eventos en las ejecuciones.

- El visualizador por consola (*VisualizadorConsola)*, un proyecto que se comunica con el usuario haciendo uso de la consola de un programa. Pide de esta manera la información inicial al usuario y lo mantiene informado del proceso de ejecución a través de una barra de progreso escrita con símbolos. Más allá de esto, simplemente usa la información que consigue del usuario para invocar el código del proyecto anterior, actualizando los datos que muestra durante la ejecución acorde.

- El visualizador de Windows (*Visualizador)*, análogo al de consola pero que utiliza Windows Forms para pedir y mostrar información. Es una implementación rudimentaria, ya que el interés de este estudio no se encuentra en las interfaces, pero se creó con el objetivo de poder mostrarle al usuario una interfaz más amigable que la consola. De la misma manera que el proyecto anterior, cuenta con una barra de progreso que se actualiza acorde con los avances del programa.

- El último proyecto (*Telemetry*) es código relacionado con el tracker y los eventos que se usan para evaluar el funcionamiento del programa y para ayudar a la depuración de errores y valores defectuosos. Se explica en profundidad en el apartado 5.

- Adicionalmente, existe el proyecto de *UnitySimulation*, que se encuentra en una solución diferente, pero que también depende de la librería de *EvolutionSimulation*. Este proyecto se justifica como un ejemplo de uso de la librería, su funcionamiento se explica en el apartado 4.2.

3.2. Generación y especificaciones del terreno y flora

El terreno donde se realiza la simulación está representado mediante un array bidimensional cuyas casillas representan cada una de las posiciones que puede ocupar una criatura. Estas casillas son cuadradas, aunque otra implementación potencialmente más equilibrada podría haber sido representar el mundo con casillas hexagonales en lugar de cuadradas. De esta manera, las esquinas estarían “más lejos” y representarían un mayor coste en distancias. En un futuro, se podría rehacer la estructura del mundo para implementar este sistema o simplemente dar opción a elegir la forma de las casillas.

Para la realización de esta tarea, decidimos inspirarnos en el sistema de clasificación de zonas de vida de Holdridge (*Holdridge Life Zones*, 2022). Este es un esquema de clasificación de biomas que se basa en tres ejes: Biotemperatura, precipitación y evapotranspiración potencial. Para la realización de este proyecto usamos la temperatura y la precipitación, pero debido a limitaciones temporales, se decidió usar la altura (la cual es un eje auxiliar del sistema de Holdridge) para representar el clima.

Cada casilla del array es una instancia de la clase *MapData*, una clase que contiene números que representan información de la altura, temperatura, humedad, y una referencia a una planta si esta casilla contiene una. Los números de temperatura, humedad y altura son del dominio [0, 1], representando el 1 el máximo valor posible de una característica de la simulación y el 0 la menor. También se describen las regiones, que están relacionadas con la búsqueda de caminos explicada en el apartado 3.4.4.

3.2.1. Información inicial y su interpretación

Para poder generarse un terreno, es necesario tener mapas que proporcionan valores a los números de altura, humedad y temperatura de *MapData*. Se necesitan, pues, tres mapas distintos para poder comenzar la generación del terreno.

En caso de que no se proporcionen mapas generados previamente, estos se generan como parte de la ejecución. Los mapas comienzan su generación como mapas de ruido de Perlin con varias octavas superpuestas (siendo dichos procesos los explicados en la sección 2.3.2). Todos los mapas son matrices bidimensionales cuyas casillas contienen un número real de coma flotante cuyo valor es asignado en base al algoritmo descrito.

El primer mapa que se genera es el mapa de alturas. Este representa la altura del terreno en un punto, representando el valor 1 la máxima altura posible y 0 la mínima. Partiendo de un mapa de perlin previamente generado (como se especifica en el apartado anterior), al cual se le aplica una función de suavizado, configurable por el usuario. Por defecto, la fórmula es la representada por la Ecuación 3.1.

$$f(x) = \begin{cases} 0 & : x \leq 0 \\ \frac{0.5}{2}\, sen(\frac{\pi}{0.3}(x - \frac{0.3}{2})) + \frac{0.5}{2} & : 0 < x < 0.3 \\ 0.5 & : 0.3 \leq x < 0.6 \\ \frac{0.5}{2}\, sen(\frac{\pi}{0.4}(x - \frac{0.4}{2} - 0.6)) + \frac{0.5}{2} + 0.5 & : 0.6 \leq x < 1 \\ 1 & : x \geq 1 \end{cases}$$

Ecuación 3.1. Suavizado por defecto de la altura

Donde x es el valor del ruido de Perlin en ese punto. La representación gráfica de los valores correspondientes a x se muestra en la Figura 3.2.

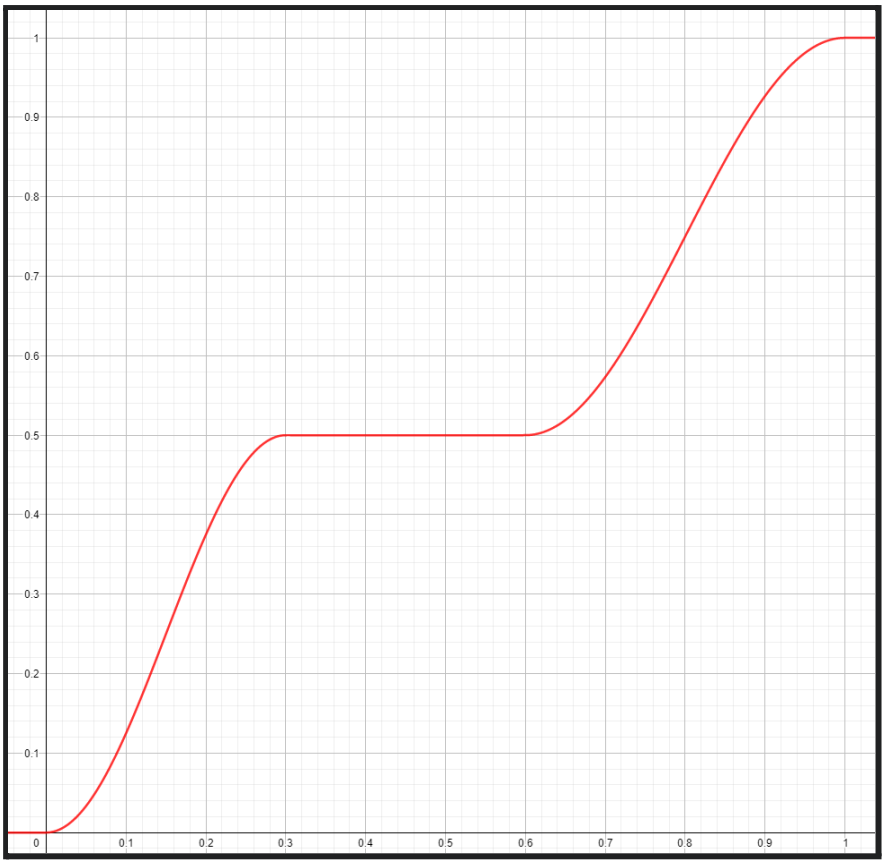


Figura 3.2. Valores de la ecuación por defecto.

Se usó la fórmula representada por la Ecuación 3.1 debido a que de esta manera se restringe la cantidad de montañas y agua a un rango menor en el mapa de ruido, produciendo terrenos relativamente planos, con algunas montañas y lagos en menor medida.

El siguiente mapa que se genera es el de las temperaturas, cuyas casillas indican la temperatura de la zona (siendo 1 lo más cálido posible y 0 lo más frío). A este mapa se le aplica una función de influencia según la altura usando el mapa de alturas anterior, de tal manera que cuanto más alto sea el punto, más frío es, siguiendo la fórmula de la Ecuación 3.2.

$$f(x) = \begin{cases} -1 & : x < 0 \\ (2x)^2 - 1 & : 0 \leq x < 0.5 \\ 2(x - 0.5) & : 0.5 \leq x < 1 \\ 1 & : x \geq 1 \end{cases}$$

Ecuación 3.2. Función de influencia de la altura sobre el resto de mapas

Cuya representación gráfica da lugar al gráfico de la Figura 3.3.

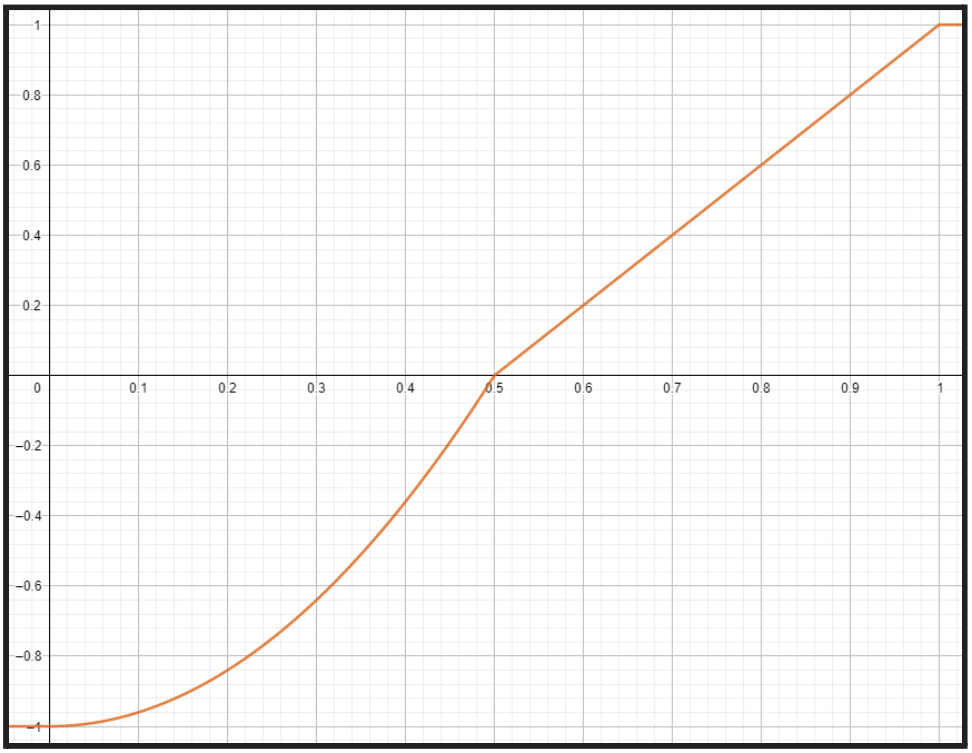


Figura 3.3. Valores de la influencia por defecto.

Esta modulación mediante el mapa anterior se hizo para que el terreno tuviera coherencia con los terrenos reales. En estos, las zonas de mayor altura se encuentran a menor temperatura debido a que cuanto menor es la presión más energía se necesita para modificar la temperatura, y la presión disminuye conforme aumenta la altura. Además, el efecto invernadero tiene un efecto menor a mayores alturas porque la presión baja implica una menor concentración de gases que asisten en este efecto.

El último mapa en ser generado es el de humedad, en el cual el valor 1 representa agua, y el 0 completa ausencia de la misma. A este mapa, al igual que al otro, se le aplica una función de influencia según la altura. De tal manera que cuanto más bajo sea el punto, más húmedo, siguiendo la lógica previamente establecida. La función usada para este propósito es la misma que en la temperatura (la representada en la Ecuación 3.2). La forma en la que se aplica esta fórmula es que, para valores positivos de la función, se resta el valor obtenido al valor de temperatura de la casilla. Al mismo tiempo, se le suma al de la humedad de esa misma casilla. En el caso de resultados negativos, se le resta (sumándose, ya que es negativo) a la humedad y se aplica un suavizado a la temperatura. Esto se debe a que esta función también sirve para determinar a partir de qué altura se considera agua, ya que si la función devuelve valores negativos marca esa casilla como acuática y ésta regula la temperatura de sus alrededores.

Todas las masas de agua modifican el mapa de temperaturas, haciendo que tiendan a la temperatura media del mapa. Esta decisión fue basada en una de las propiedades del agua: el agua tiene una capacidad calorífica tan alta, lo que es una propiedad muy importante, puesto que contribuye de forma muy notable a la regulación meteorológica y del clima. Esto se debe a que, al ser muy elevado el calor específico del agua, las grandes masas acuáticas regulan las fluctuaciones extremas, en particular las temperaturas (A & W, 2003).

Una vez generados los mapas, se procesan todos juntos en uno. Este proceso comienza por suavizar las alturas según una fórmula explicada más adelante. Posteriormente, se aplica el mapa de alturas para modificar los mapas de temperatura y humedad: en las zonas del mapa de temperaturas que se superponen a valores altos en el mapa de alturas los valores de las temperaturas son rebajados. Además, las zonas consideradas como agua del mapa de alturas modifican los otros mapas de manera que el mapa de humedad es actualizado incrementando la humedad en las zonas cercanas al agua, y, en el mapa de temperatura, las temperaturas cercanas a estas zonas son suavizadas.

Una vez procesados los valores, se produce un mapa de flora, con este mapa se generan las plantas que existirán durante la simulación, descritas en el siguiente apartado.

3.2.2. Tipos de flora y su generación

Las plantas que pueblan el terreno de la simulación parten de la clase básica *StaticEntity*. Esta clase es usada para definir entidades ajenas a las criaturas que no tienen capacidad de moverse. Además del identificador propio de todas las entidades, esta clase cuenta con los valores *X* e *Y*, que representan la posición de la casilla en la que se encuentra la entidad, y los número *currHP* y *maxHP*, que representan la salud actual y máxima de la entidad.

De esta clase hereda la clase *Plant*, que tan sólo expande la funcionalidad de su clase padre mediante la adición de un identificador del tipo enum que indica el tipo específico de planta que es (*Grass*, *Bush*, *EdibleTree* o *Tree*). Se hace una última distinción adicional entre los distintos tipos de plantas: las no comestibles, que heredan de *Plant*; y las comestibles, que heredan de una clase hija de *Plant* llamada *EdiblePlant*.

EdiblePlant expande la funcionalidad base de *Plant* mediante la implementación de métodos que permiten a la planta ser comida por una criatura que interactúe con ella, y regenerarse constantemente para poder crecer de nuevo una vez consumida. Cuando una criatura intenta comer una planta, resta su daño a la misma, y ésta le otorga un porcentaje de su nutrición asociada proporcional a la vida quitada. Una vez una planta es reducida a 0 puntos de vida, debe regenerarse por completo antes de poder ser intentar ser consumida de nuevo.

Los parámetros especificados en la Tabla 3.1 representan los valores por defecto del programa, que pueden ser cambiados por el usuario (véase el apartado 4.1.1).

Tipo de planta	Características
Grass	Hereda de la clase *EdiblePlant*, teniendo como tipo *Grass*. Tiene, por defecto, una cantidad máxima de vida de 10 puntos, y otorga una cantidad de nutrición, por defecto, de 2.8 puntos a la criatura que se la come. Además, se regenera por completo, por defecto, en 24 horas de simulación.
Bush	Hereda de la clase *EdiblePlant*, teniendo como tipo *Bush*. Tiene, por defecto, una cantidad máxima de vida de 20 puntos, y otorga una cantidad de nutrición, por defecto, de 6.4 puntos a la criatura que se la come. Además, se regenera por completo, por defecto, en 120 horas de simulación.
EdibleTree	Hereda de la clase *EdiblePlant*, teniendo como tipo *EdibleTree*. Tiene, por defecto, una cantidad máxima de vida de 50 puntos, y otorga una cantidad de nutrición, por defecto, de 13.2 puntos a la criatura que se la come. Además, se regenera por completo, por defecto, en 360 horas de simulación.
Tree	Hereda de la clase *Plant*, teniendo como tipo *Tree*. Tiene, por defecto, una cantidad máxima de vida de 0 puntos, pues, al no ser una planta comestible, su vida nunca es manipulada por ninguna criatura y no hay necesidad de asignarle un número válido. De la misma manera, no tiene ni puntos de nutrición otorgados, ni período de regeneración.

Tabla 3.1. Tipos de plantas generadas

Para generar estas plantas se combinan los mapas de humedad y temperatura, usando la fórmula de la Ecuación 3.3.

$$f(h,t) = min(max(h^{1.75} - (2t - 1)^2, 0), 1)$$

Ecuación 3.3. Función de densidad de flora

Donde “h” es la humedad y “t” es la temperatura en ese punto. Dando lugar a un mapa de flora que contiene valores en el intervalo [0, 1], mostrados en la Figura 3.4.

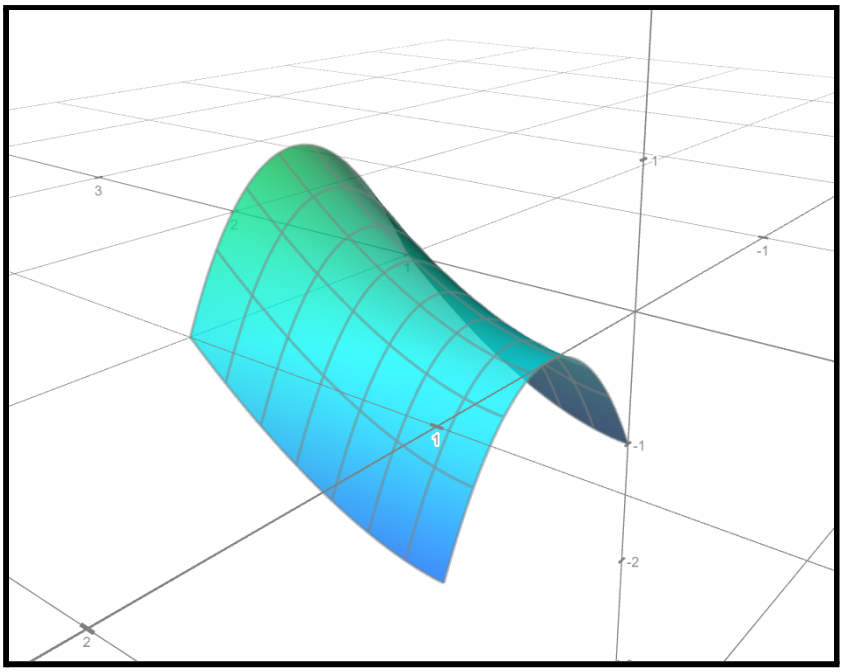


Figura 3.4. Representación visual de la Ecuación 3.3.

Los valores de este mapa indican la probabilidad de que en esa casilla haya una planta. El tipo de planta que se genera depende también de este número: las casillas con valores altos tienen altas probabilidades de generar árboles, las de valores bajos tienden a generar hierba, y los arbustos son favorecidos en valores medios. De esta manera se obtienen zonas de alta densidad de vegetación formadas mayoritariamente por árboles, y yermos con vegetación escasa, principalmente hierba.

3.2.3 Configuración de la generación del terreno

En el proyecto, la clase World recibe un *WorldGenConfig*, que es una clase auxiliar que contiene toda la información necesaria para configurar la generación de terreno y flora. En la Tabla 3.2 se explican sus parámetros:

Atributo	Características
mapSize	Indica las dimensiones del mapa. Este parámetro es ignorado si se provee algún tipo de mapa de alturas, humedades o temperaturas, tomando como valor las dimensiones del mapa proporcionado.
heightWaves	Indica las octavas a usar a la hora de generar un mapa de alturas. Por defecto recibe una octava de frecuencia 0.5, amplitud 1 y desplazamiento aleatorio.

humidityWaves	Indica las octavas a usar a la hora de generar un mapa de humedad. Por defecto recibe una octava de frecuencia 0.5, amplitud 1 y desplazamiento aleatorio.
temperatureWaves	Indica las octavas a usar a la hora de generar un mapa de temperaturas. Por defecto recibe una octava de frecuencia 0.25, amplitud 1 y desplazamiento aleatorio.
heightMap	Indica si se ha proporcionado un mapa de alturas. Al proporcionarse no se generará este tipo de mapa. Por defecto es null.
humidityMap	Indica si se ha proporcionado un mapa de humedad. Al proporcionarse no se generará este tipo de mapa.Por defecto es null.
temperatureMap	Indica si se ha proporcionado un mapa de temperaturas. Al proporcionarse no se generará este tipo de mapa.Por defecto es null.
regionMap	Indica si se tiene que tiene que generar un mapa de regiones de Voronoi. Por defecto es null.
heightModifiedByFunction	Indica si se quiere que se aplique la fórmula de *evaluateHeight* al mapa de alturas generado o proporcionado. Por defecto es true.
evaluateHeight	Función a usar para modificar el mapa de alturas. Debe ser una función definida para todo número real y con una imagen de [0, 1]. Por defecto es la función representada en la Ecuación 3.1.
evaluateInfluence	Función a usar para que el mapa de alturas influencie al resto de mapas. Debe ser una función definida para todo número real y con una imagen de [-1, 1]. Por defecto es la función representada en la ecuación Ecuación 3.2.
evaluateFlora	Función a usar para determinar la cantidad de flora de un punto. Debe ser una función definida para toda pareja de reales en el rango [0,1] y con una imagen de [0, 1]. Por defecto es la función representada en la Ecuación 3.3.
temperatureSoftener	Función a usar para determinar cuánto suaviza el agua a la temperatura. Debe ser una función definida para toda pareja de reales en el rango [0,1], un valor de temperatura media y tiene que tener una imagen de [0, 1]. Por defecto es la función representada en la Ecuación B.1.
floraSelector	Función usada para determinar el tipo de planta que se va a seleccionar. Debe ser una función definida para todo número real en el rango (0, 1] y con una imagen de 0, 1, 2,

	3. Por defecto es la función representada en la Figura B.1.
type	Indica si se quiere usar una de las plantillas previamente hechas. Se debe poner en Custom para que no se use solo mapSize. No tiene un valor por defecto ya que es necesario indicar uno en el constructor.
numPasses	Cantidad de masas de tierra completamente aisladas en una región de 32x32. Es usada para la generación del mapa de Voronoi. Por defecto es 2.

Tabla 3.2. Atributos de *WorldGenConfig.json*

La configuración cuenta también con una serie de plantillas pre hechas, que se explicarán más adelante.

- Default: Es la plantilla por defecto, que usa los valores indicados anteriormente.

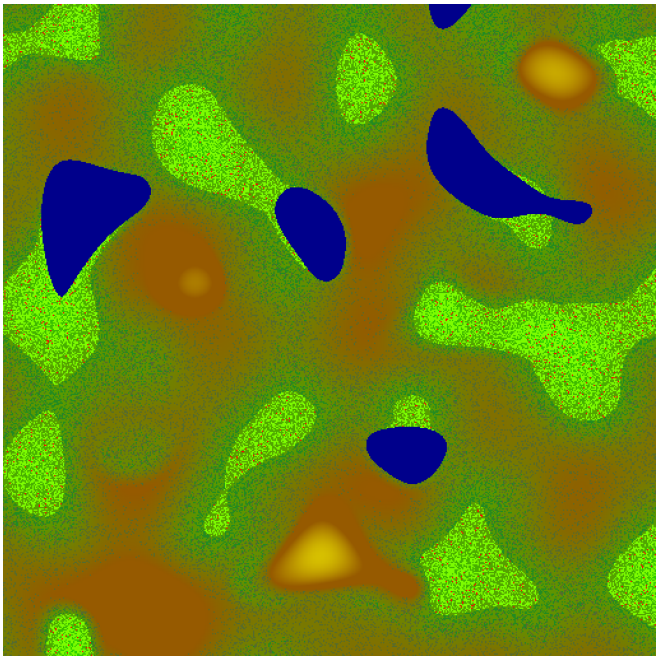


Figura 3.5. Ejemplo de mapa por defecto o "*default*".

- Atolón: Usa un mayor número de octavas y la fórmula de suavizado de alturas está modificada de tal manera que aumenta las alturas entre dos radios concéntricos y disminuye en el resto.

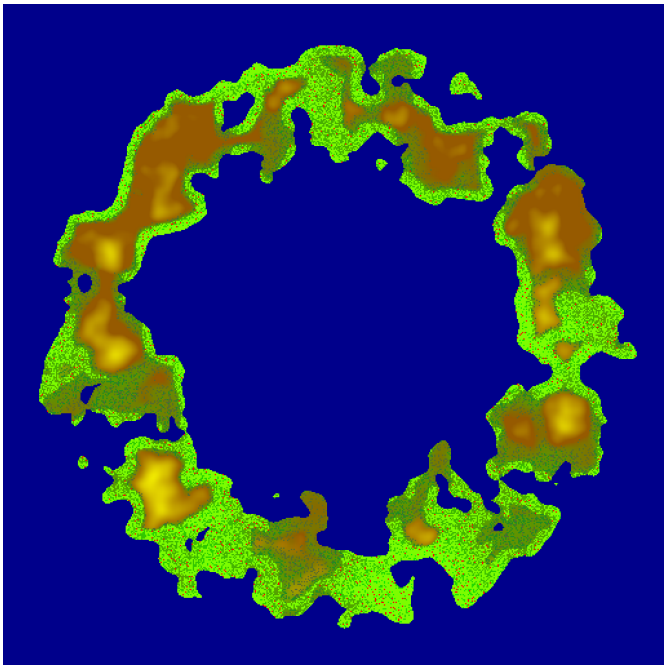


Figura 3.6. Ejemplo de mapa “atolón”.

- Pantano: Usa las mismas octavas que el atolón, pero con la función por defecto de suavizado de alturas. También modifica el *numPasses* para que se genere correctamente el mapa de Voronoi.

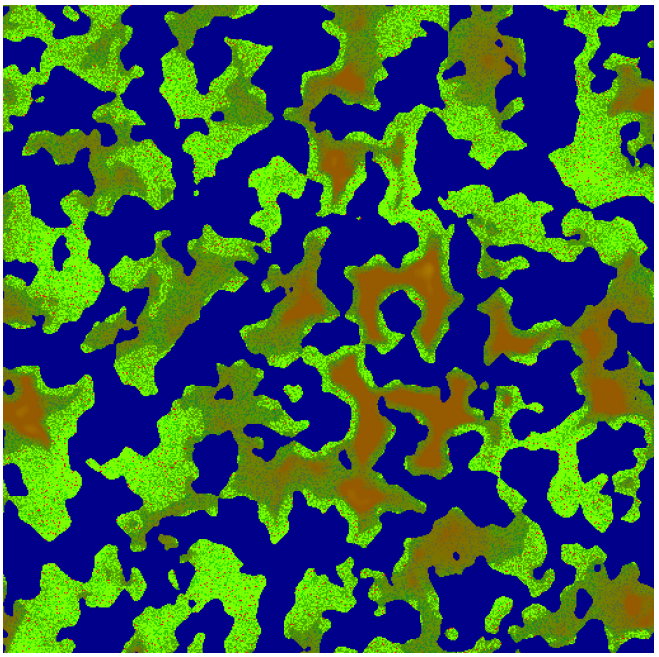


Figura 3.7. Ejemplo de mapa “pantano”.

- Isla: Al igual que el atolón, usa más octavas y la fórmula de suavizado de altura está modificada para que la altura disminuya a mayor distancia del centro del mapa.

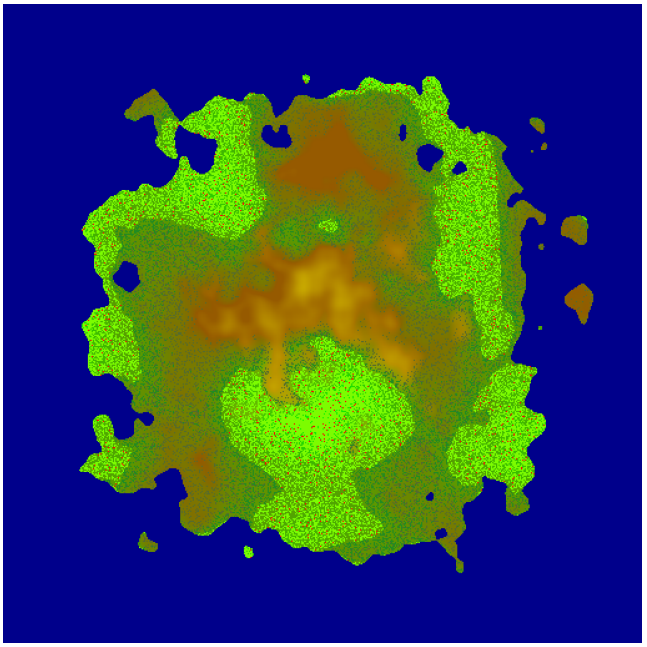


Figura 3.8. Ejemplo de mapa “isla”.

3.3. Representación genética de las criaturas

Dada la naturaleza e intención del proyecto, es fundamental que las criaturas tengan una genética que dicte las propiedades del animal y que pueda pasarle a su descendencia. Para lograr esto se ha empleado programación evolutiva, como se ha mencionado previamente, y se han definido una serie de características y propiedades que se representan en un cromosoma, para hacer posteriormente uso de los conceptos que se dan en la programación evolutiva para transmitirlos, heredarlos e interpretarlos.

3.3.1. Sintetización de un animal en características definitorias

Este proyecto no pretende hacer una representación completamente fiel de un ser vivo; no se pretende simular sus células o replicar las proteínas que intervienen en los procesos biológicos de una criatura, al igual que no se pretende crear un cromosoma comparable al de un animal real. El motivo es que dado el propósito del proyecto, generar criaturas adaptadas a su entorno, principalmente para su interpretación y uso en videojuegos, no es necesario tal realismo interno, sólo una lógica funcional y coherente con respecto a los sistemas naturales de los seres vivos. Debido a esto, se ha tenido que crear un sistema propio que represente las características de la criatura. Este es un

concepto que se lleva explorando desde hace décadas por juegos de rol, siendo el mayor representante Dungeons & Dragons, que usa un sistema de atributos y habilidades que ha servido de inspiración. El principal motivo por el que se decidió usar este sistema parte de la base del objetivo del proyecto. Ya que queremos tener un ecosistema que pueda ser usado en videojuegos, parece lo más razonable usar un sistema que esté afianzado en este grupo. Por ello, creemos que el grado de abstracción que ofrece Dungeons & Dragons es suficientemente preciso para la tarea encomendada; sin embargo, el motor y el tipo de solución aceptaría también, con los cambios oportunos, otro sistema de características.

En dicho juego, hay seis atributos que representan de forma fundamental la aptitud de un personaje en cualquier campo físico, mental o social con un valor numérico. Después, existen una variedad de habilidades, representadas también con valores numéricos calculados en base a los atributos en combinación con otros factores, que representan la actitud del personaje a la hora de realizar acciones concretas. Un ejemplo de atributo sería la destreza del personaje, que impacta en gran medida en la capacidad del personaje de realizar acrobacias, acciones que requieran destreza manual, etc. De manera similar, las criaturas constan de los atributos especificados en la Tabla 3.3, que vienen dados por el cromosoma y pueden tener una cierta dependencia entre ellas si esta es razonable (el valor de un atributo se calcula parcialmente en base al valor de otro, ver 3.3.3):

Atributo	Definición
Fuerza	Define el daño de los ataques.
Constitución	Define la vida máxima.
Fortaleza	Bloquea una cantidad del daño que fuera a recibir. Representa una capa exterior robusta, como exoesqueletos o pieles gruesas.
Perforación	Cantidad de fortaleza que ignora los ataques de la criatura. Representa colmillos o mecanismos como el del taladro de ostra.
Movilidad	Define la velocidad y la eficiencia energética del movimiento de la criatura. Una mayor movilidad implica que la criatura puede moverse más a menudo y más rápido gastando menos energía en ello.
Resistencia	Define la frecuencia con la que la criatura necesita descansar.
Percepción	Permite ver a más distancia y detectar la presencia de criaturas camufladas con mayor precisión.
Camuflaje	Reduce la probabilidad de que las demás criaturas detecten su presencia hasta que esta se hace evidente a través de una interacción.
Agresividad	Representa la tendencia a combatir con otras criaturas. Una criatura con mayor agresividad se atreverá a atacar a otras criaturas más amenazadoras.

Conocimiento	Tiene una mayor memoria para recordar la localización de recursos en el mundo.
Tamaño	Define las dimensiones de las criaturas y tiene menor influencia en la velocidad de movimiento. También especifica el valor nutricional del cadáver.
Metabolismo	Se hizo una generalización del metabolismo en todas las criaturas, ya que tiene consecuencias distintas dependiendo de si el sistema es endotérmico o exotérmico, así que se utiliza el esquema del primer mecanismo. De esta manera, y dado que el metabolismo hace referencia al conjunto de procesos biológicos que requieren energía, como se ha establecido previamente, se llega a la siguiente conclusión: cuanto más activo el metabolismo mayor la energía y los recursos perdidos, en el caso de la simulación, hidratación y energía consumida. Sin embargo, más activa es la criatura, pues es el caso contrario a la hibernación. También hay una relación entre la temperatura corporal de una criatura y la actividad de su metabolismo, pues "un organismo viviendo a una mayor temperatura no tiene otra opción que sintetizar más ATP (fuente principal de energía en las células) y consumir más oxígeno" (Clarke & Fraser, 2004).
Temperatura Ideal	La temperatura idónea sobre la que se calcula el rango de temperaturas que soporta.
Rango de temperaturas	Una cantidad de unidades de temperatura por encima y por debajo de la temperatura ideal que la criatura puede soportar sin sufrir consecuencias. Ver 3.4.1.
Longevidad	Cuántos años vive la criatura y cuánto tardan en llegar a la adultez. Las criaturas recién nacidas que no pertenecen a la generación primigenia empiezan siendo crías, con sus cualidades reducidas, que van aumentando hasta llegar a su valor real según se acercan a alcanzar la adultez.
Dieta	Define el tipo de materia que puede consumir. Puede ser herbívoro, carnívoro y omnívoro.
Miembros	Número de extremidades que tiene el cuerpo de la criatura. Aumenta el consumo de energía.

Tabla 3.3. Atributos de las criaturas

Aparte, con la intención de representar características que pueden tener los animales pero no dependen directamente de sus propiedades físicas, hay otro tipo de información que se representa junto a los atributos que se ha denominado como habilidades. Se han ideado una cierta cantidad que pretende ejemplificar el funcionamiento de este tipo de variables, añadiendo factores adicionales a las criaturas. Los seres vivos son demasiado complejos y variables para representar todas sus características en este proyecto, así que

se han escogido algunas y se han simplificado para poder reproducirse. Estas habilidades se listan en la Tabla 3.4.

Habilidad	Descripción
Arbóreo	Dicta si se puede mover por los árboles y a qué velocidad. Se puede mover por el suelo con una penalización.
Volador	Dicta si se puede mover por el aire y a qué velocidad. Se puede mover por los árboles con una penalización y por el suelo con otra aún mayor.
Venenoso	Dicta si al atacar infringe veneno al oponente, que hace daño con el tiempo, y en caso afirmativo cuánto.
Visión nocturna	Dicta si suaviza la pérdida de visión de noche, y en caso afirmativo cuánto.
Cuernos	Dicta si tiene daño añadido en sus ataques.
Mimetismo	Dicta si tiene un valor añadido en la intimidación. Basado en criaturas que imitan a otras más amenazantes, como las serpientes reales, que aunque inofensivas imitan a otras serpientes venenosas con los patrones de sus escamas.
Erguido	Dicta si puede alcanzar recursos en árboles independientemente de ser o no arbóreo o aéreo, y aumenta cómo de lejos puede ver.
Púas	Dicta si hace daño al ser atacado. Basado en animales como el puercoespín.
Carroñero	Dicta si puede consumir cadáveres en mal estado sin sufrir consecuencias por comer materia putrefacta. Además, obtienen más beneficios al consumir los cadáveres.
Pelaje	Dicta si soporta mejor temperaturas bajas sin necesidad de alterar su metabolismo.
Paternidad	Dicta si la especie cuida de sus recién nacidos. Se explica en el apartado 3.3.1.

Tabla 3.4. Habilidades de las criaturas

Cabe destacar que además de depender de los atributos, también dependen de las habilidades, que siguen un criterio concreto. Las habilidades se “desbloquean”, queriendo esto decir que tienen que tener un valor lo suficientemente alto como para que sean útiles y no inicios o vestigios, de la misma manera que las alas solo sirven para volar cuando están desarrolladas. Esto se calcula en base a un porcentaje dado por el usuario, de manera que si una habilidad tiene un valor igual o superior al porcentaje dado del valor máximo de la susodicha, se considera que se posee, y en caso contrario no. Una vez se poseé, la efectividad se calcula en base a cuánto se acerca el valor que tiene la criatura

al máximo, de forma que una criatura con alas muy desarrolladas tiende a volar más rápido que otra que ha dedicado menos recursos al propósito de volar. En la siguiente gráfica se muestra un ejemplo de la efectividad de una habilidad dependiendo de su valor. En este caso, la habilidad se considera desbloqueada a partir de los 20 puntos y tiene un máximo de 50. En la Figura 3.9, el eje de abscisas representa los puntos en la habilidad, el de ordenadas el porcentaje de efectividad.

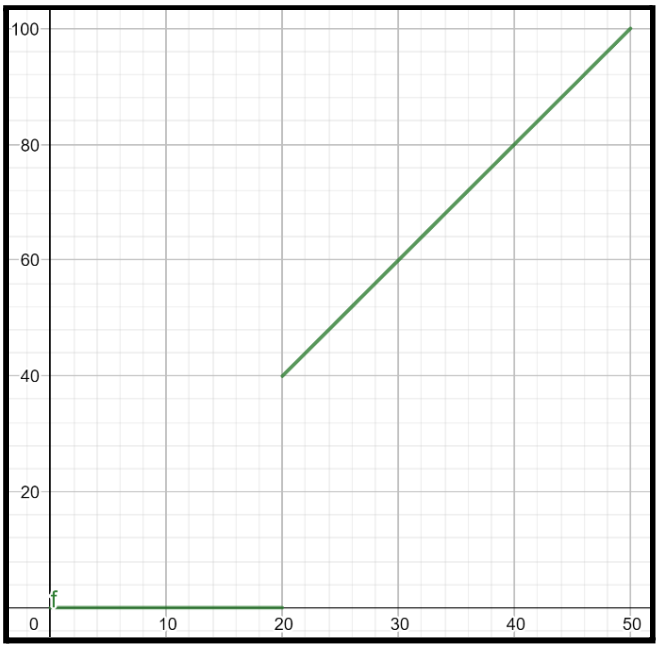


Figura 3.9. Puntos de habilidad frente a porcentaje de efectividad

La intención tras esta implementación también es crear datos flexibles y versátiles que una vez el usuario haya importado las criaturas pueda usar para añadir comportamientos o características adicionales en su propio proyecto.

Los valores por defecto de todos estos datos, al igual que cualquier valor configurable por el usuario, se han establecido siguiendo una lógica fundamental basada en el funcionamiento de los seres vivos en la vida real, y modificados en base a los resultados proporcionados en comparación de los deseados. Por ejemplo, se ha intentado describir la lógica del mundo real relacionando de forma directamente proporcional la movilidad y el número de miembros, la fuerza y el tamaño, etc. Estas relaciones vienen dadas por defecto, y pueden ser modificadas por el usuario, como se explica más adelante en este mismo capítulo.

3.3.2. Cualidades internas de las criaturas simuladas en el entorno virtual

Definimos las cualidades como variables que se usan en la lógica interna del programa para comprobar el estado y características de una criatura. Son la interpretación de los

atributos y habilidades, y tienen especial peso en el proceso de calcular el comportamiento de esta en la máquina de estados que dicta su conducta, véase 3.4.

Se dan los tipos de cualidades, con explicaciones de los cálculos más complejos por defecto en la Tabla 3.5.

Cualidad	Descripción
Estado	Llevan la cuenta de la situación de la criatura en base a su ciclo de vida y necesidades. Monitorizan la vida, hidratación (se consigue bebiendo), energía (se consigue comiendo), descanso (se consigue durmiendo) y edad actual. Se forman fundamentalmente por tres cualidades cada una: el estado actual de la criatura, un nivel de consumo, en base al cual se van consumiendo los recursos actuales y finalmente, tienen máximo. Un ejemplo del funcionamiento de estas cualidades sería el caso de una criatura que come, llenando su energía actual hasta el máximo, que se va reduciendo cada turno en base al consumo que hace de esta, hasta que necesita conseguir más. Estas cualidades tienen fundamentalmente el mismo funcionamiento, pero en la implementación por defecto no se calculan de la misma manera. Algunas dependen de atributos, como la vida máxima de la constitución, mientras que algunos son valores arbitrarios como la energía, hidratación o descanso máximos. En este caso, por ejemplo, para simplificar los cálculos, se estableció un valor máximo para estos recursos y los atributos modifican el consumo, de forma que dependen de la criatura igualmente.
Internas	Hacen referencia a las características de una criatura que están relacionadas con los procesos internos. Se incluyen las cualidades análogas a los atributos de tamaño, metabolismo y número de miembros, más la temperatura máxima y mínima que soporta teniendo en cuenta si tiene o no la habilidad de pelaje. También se guarda el tipo de dieta que sigue la criatura, y si es carroñera en base a la habilidad con el mismo nombre.
Movimiento	Representan la velocidad de movimiento en casillas del mundo a la que se mueve la criatura en cada medio. Todas las criaturas tienen movimiento terrestre, que por defecto es igual al valor del atributo movilidad. Si son arbóreos, el valor de movilidad modificado por esta habilidad se le asigna a la velocidad arbórea, y la terrestre es igual a la arbórea con una penalización. De igual manera, si puede volar, a esta se le asigna movilidad modificada por esta habilidad, a la velocidad arbórea la velocidad aérea con una penalización y a la terrestre la arbórea con la misma penalización aplicada de nuevo. De esta manera las criaturas se especializan en un medio. La velocidad, a su vez, se reduce de forma proporcional a su tamaño cuando pueden moverse por medios no terrestres, ya que entorpece el movimiento.
Combate	Incluyen el daño de los ataques en base a la fuerza y modificados por la habilidad de cuernos, el daño que se bloquea en base a su fortaleza,

	el daño que se infringe cuando se ataca a la criatura en base a las púas, la fortaleza que se ignora cuando se ataca a otra criatura en base a la perforación y finalmente el veneno, que se explica a continuación. También se incluiría en esta clasificación las cualidades de agresividad, que depende del atributo con el mismo nombre, y la intimidación, que depende del tamaño y es análoga a la agresividad. Estas cualidades sirven para decidir si una criatura ataca a otra, al comparar los dos valores. Si una criatura es venenosa, cada vez que daña a un enemigo (es decir, que su ataque perfore la fortaleza del oponente) aplica un veneno, que hace que la acción de ataque tenga un coste energético mayor. Este veneno tiene comportamiento de efecto de estado: realiza funciones específicas al ser aplicado, cada vez que se actualiza y al acabar su duración o ser cancelado prematuramente. El veneno es creado con una duración y daños crecientes según lo venenosa que sea la criatura que lo aplica, y cada vez que la criatura se actualiza, le resta la cantidad de vida establecida como su daño. Tras estar activo la cantidad de ciclos establecidos en su duración, el veneno se retira de los estados activos de la criatura y deja de tener efecto. Varias instancias de veneno con distintas características pueden estar activas en una misma criatura a la vez, lo que la hace muy potente en combate, y justifica su elevado coste energético adicional en la acción de ataque.
Interacción	En este conjunto se incluyen aquellas relacionadas con cómo la criatura interactúa con el mundo. Por ejemplo, la distancia a la que ve, que depende de la percepción y si es de noche se penaliza, dependiendo de la visión nocturna que tenga. A su vez, existe el camuflaje, que dificulta que otras criaturas la identifiquen aún estando en su rango de visión. También se representa si puede alcanzar recursos en un árbol, si es arbóreo, aéreo o si tiene la habilidad de erguido, o en el aire, si es aéreo. Finalmente, se guarda el valor del conocimiento de la criatura, que influye en la mente de esta, influenciando cuántos elementos puede recordar del mundo y durante cuánto tiempo.
Reproducción	Se engloba el sexo de una criatura, más cuánto tiempo debe pasar entre procesos de reproducción y si en base a su estado actual se encuentra en celo o no. La habilidad de paternidad también se conserva como una cualidad, para darle el comportamiento a la criatura si se ha desbloqueado. Funcionan de tal manera que una criatura hembra está en celo cuando tiene las necesidades de estado suplidas, está en celo y ha pasado una cierta cantidad de tiempo desde la última vez que se reprodujo. En este caso, si encuentra un compañero, pierde el celo, empieza a gestar y se reinicia el tiempo que lleva sin reproducirse.

Tabla 3.5. Cualidades de las criaturas

Por lo general, son la interpretación de los valores genéticos a unidades internas de simulación, por ejemplo, el daño que hace una criatura al atacar a otra depende de su fuerza, pero no tienen porqué coincidir ambos números; se puede dar que los valores de los genes están muy simplificados y en la simulación se quieran valores más grandes o que una cualidad depende de dos atributos a la vez, de forma que no se puede simplemente copiar el valor de una de estas. Al mismo tiempo, no hay motivo por el que estos cálculos sean una parte intrínseca de la simulación, en el sentido de que no hay que utilizar unas fórmulas concretas y por lo cual se puede dejar al usuario configurarlas como quiera. Por defecto, hay una serie de cálculos que se han probado y pulido en el desarrollo en base a experiencias y el funcionamiento real de los aspectos a reproducir en la simulación. Si se desea, se pueden modificar usando herencia en el código de la simulación. El cálculo y representación del cromosoma se explica en el siguiente apartado, pero conceptualmente es importante conocer la relación entre los genes, los atributos y las cualidades de las criaturas, así que se presenta un ejemplo en la Figura 3.10.

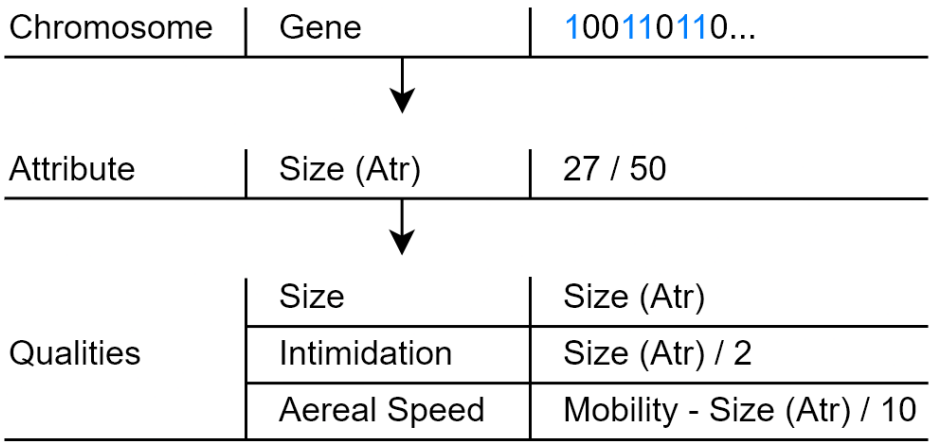


Figura 3.10. Proceso de conversión de cromosoma a cualidades

Cada criatura tiene un tiempo de vida máximo esperado, y un umbral a partir del cual se le considera adulta. Cuando una criatura debe utilizar una de sus características, se realizan unos cálculos que devuelven esta característica modificada por la edad: partiendo de un multiplicador base de cría, se comprueba qué porcentaje de la edad hasta el umbral de adultez ha vivido, que se le suma al multiplicador final. De este modo, una cría tendrá una cantidad mayor de cualidades conforme vaya creciendo hasta llegar a ser adulta. Una vez una criatura es adulta, el multiplicador de la criatura se mantiene estable, devolviendo siempre la cantidad completa de las cualidades.

3.3.3. Programación evolutiva, cromosomas y genes

Al igual que la genética real, la programación evolutiva tiene como punto de partida una representación de la información de cada individuo, un cromosoma, formado por fragmentaciones de los datos con funciones concretas, los genes. La información que contiene un cromosoma se puede representar de una pluralidad de maneras, dependiendo de las necesidades de la implementación.

En este caso, se decidió usar un sistema binario, en el que el cromosoma es una secuencia de unos y ceros. Uno de los motivos es que esta implementación facilita la aplicación de funciones de cruce y mutación. Estas se explican más adelante en el capítulo, pero en resumen usar valores binarios es una forma cómoda y eficiente de tratar la genética, aparte de flexible porque se puede interpretar como uno quiera. Usar valores decimales pone a los algoritmos genéticos en una situación incómoda, en la que habría que diseñar funciones matemáticas para que las funciones de cruce y mutación no sean demasiado exageradas. Un ejemplo de este problema sería encontrar el valor del gen de un hijo cuando el del padre vale 2 y el de la madre 10. ¿Se hace la media de la diferencia y el resultado es 6, se coge un valor aleatorio entre cero y la diferencia y se suma al mínimo, tiende a un extremo…?

En un primer momento, el primer acercamiento a la representación del cromosoma se planteó en base a las dependencias de los atributos, pues como se mencionó en los apartados anteriores, estos pueden estar relacionados entre sí. Al mismo tiempo, dos relaciones no tienen por qué ser iguales ni afectar de la misma manera a un mismo atributo. Surgió así la idea de que cada atributo fuera la combinación de una serie de posiciones de bits, que puestos en continuación uno de otro y transformados a decimal desde binario daría un valor que se asignaría a los atributos y habilidades. Usando este procedimiento, un mismo bit sería más significativo cuanto más tarde se incorpore al valor binario. Por ejemplo, supongamos que el atributo de fuerza se representa con los bits ABCDE del cromosoma. El bit A se multiplicaría por 2 ^ 4 debido a su posición, es más significativo que el bit E en la posición 0. Ahora supongamos que el atributo de tamaño depende levemente de fuerza, mientras que el atributo de intimidación depende en gran medida. Entonces se podrían repetir bits de fuerza con mayor o menor magnitud en la representación de estos atributos, de manera que tamaño podría ser XXXBX e intimidación XBXXX, donde X representa un bit cualquiera. Ambos dependen del mismo atributo pero en medidas distintas, a la vez que se reduce el tamaño del cromosoma al reutilizar bits.

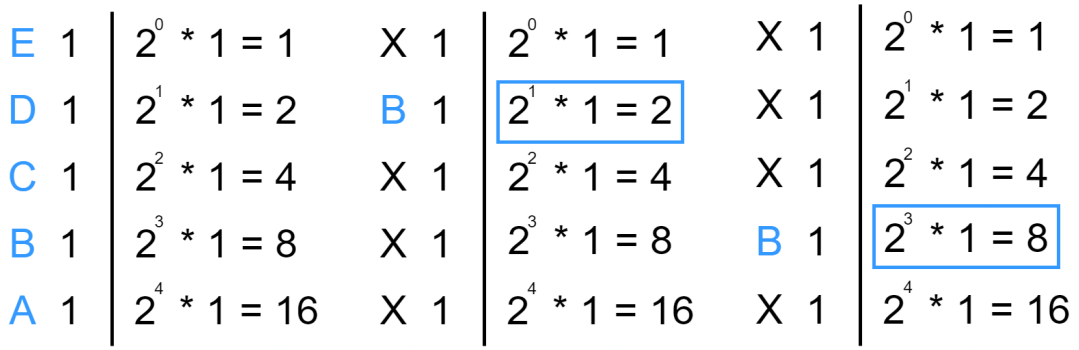


Figura 3.11. Ejemplo de un mismo bit de una secuencia con distinto peso en otras

Sin embargo, a parte de ser una implementación confusa e inconveniente tiene dos grandes problemas: el primero es que el usuario no puede configurar los valores de los atributos, pues las posiciones se asignan en el código, y de no ser así no tiene sentido que el usuario tenga que escoger posiciones de bits de un cromosoma cuya implementación no conoce. El segundo problema, es que los algoritmos genéticos, los cuales se explican más adelante, podrían generar cambios muy drásticos en la genética de una nueva criatura, si se modifica el valor de un bit con gran influencia (véase el bit A, que además puede formar parte de varios genes a la vez).

La segunda y última implementación que se probó consiste en la asignación de dominios o rangos para cada gen, de forma que tienen una posición inicial y final en el cromosoma, y el valor de un gen es el número de bits activos en su dominio. De esta forma, todos los bits tienen la misma importancia en el cromosoma, y por lo cual no hay ni mutaciones inútiles ni "monstruos" como diría Goldschmidt. Además, esta implementación permite al usuario no solo asignar sus propios valores máximos y dependencias como se explica a continuación, sino que tampoco está limitado a valores múltiplos de 2, como implicaba la representación binaria. Se presentó este sistema a Carlos Cervigón Ruckauer, profesor de programación evolutiva en la facultad de informática de la Universidad Complutense de Madrid, quien dio una opinión positiva al respecto, mencionando la posibilidad que se explicó previamente de usar valores enteros.

Los genes en este caso representan los atributos en base a los que se calculan las cualidades más las habilidades previamente descritas. Dichos atributos, cualidades y habilidades, no son configurables por el usuario de forma externa, ya que la máquina de estados y cálculos de la lógica se han hecho en torno a estos. Sin embargo, el usuario puede establecer a través de un archivo JSON los valores máximos que cada gen puede tomar. Por defecto el programa tiene un JSON con una serie de valores establecidos. También se configuran en este archivo las dependencias de cada gen a través de porcentajes. Por ejemplo, si el tamaño tiene un valor máximo de 50 y depende un 20% de la fuerza, cuyo valor máximo es 20, esto significa que el cromosoma reserva espacio para representar 50 - (0.2 x 20) y a la hora de calcular el valor del tamaño le suma a este

una regla de 3, donde el 100% del valor de fuerza representa un 20% de 50 (valor máximo del tamaño), que se obtiene del cromosoma. De esta manera, cuando se ejecuta el programa, la estructura del cromosoma se calcula en base a los datos dados por el usuario, dando una longitud a cada gen en base a su valor máximo posible teniendo en cuenta el valor máximo que ha establecido el usuario y la aportación máxima o mínima posible de cada dependencia. Para asegurar que los valores que recibe el programa en el JSON son válidos, se pasan primero por el Validator (ver 4.1.1). Así se asegura que no puede llegar, por ejemplo, una combinación de dependencias cuya suma de valores puede superar el valor máximo establecido por el usuario para un gen dado, un valor máximo negativo para un gen, etc.

Un último elemento importante en la programación evolutiva es el concepto del apocalipsis, que consiste en la muerte de toda la población que constituye la simulación. Esto se hace cuando se estanca la genética de los seres vivos, y no sufren cambios que mejoren su aptitud en un número de generaciones. Se empieza una simulación nueva desde el principio con nuevos cromosomas, para ver si con el paso del tiempo surgen individuos más aptos, y en caso contrario, se conserva como el mejor individuo el más apto de la simulación anterior. Aunque efectivo, este sistema no es coherente con el objetivo de este estudio; no se buscan los individuos más eficientes posibles, pues la evolución no lo hace tampoco, sino aquellos que son capaces de sobrevivir y proliferar en su entorno, aún siendo imperfectos. Además, un apocalipsis causaría una incongruencia en caso de que la generación original se estancara: si se guardan las especies que existen tras x años de evolución debido a este estancamiento y se van a generar nuevas, ¿estos nuevos individuos se crean en el año 0 o el año x? Si se crean en el 0, se podría dar un bucle infinito de regeneraciones de la simulación, pero si se mantiene el paso temporal ni la generación original ni ninguna de las anteriores ha evolucionado durante el tiempo que ha dictado el usuario, volviendo los resultados deshonestos.

3.3.3.1. Funciones de cruce y mutación

Ya se explicaron los principales algoritmos que se encuentran de forma general en la programación evolutiva para representar este tipo de procesos, al igual que el significado de estos en el apartado 2.2.1. Debido a esto, no se van a explicar de nuevo, pero cabe mencionar los tipos de funciones de cruce que se describieron: recombinación en uno y dos puntos, corte y empalme y recombinaciones uniforme y uniforme media.

Aparte, hay implementaciones más elaboradas, pues las anteriores eran las más tradicionales y dependiendo de las necesidades de uno pueden ser suficiente o no. En este caso se decidió que la función de cruce que se emplearía sería una recombinación uniforme. Menos corte y empalme, que no es válida ya que el cromosoma de este proyecto tiene una longitud fija, cualquiera de las anteriores implementaciones es funcional, pero se decidió usar una recombinación uniforme. Para llegar a esta conclusión se fueron descartando las otras opciones. La recombinación en un punto puede dar como resultado cromosomas muy poco recombinados, y se manipula el algoritmo para que corte por la mitad aproximadamente, tiene el mismo inconveniente que la recombinación en dos puntos: tiende a mantener los valores de muchos genes iguales que los de la madre o el padre, cuando la recombinación uniforme tiende a hacer

la media por así decirlo. Por último, la recombinación uniforme media no parecía aportar una diferencia relevante respecto a la uniforme estándar en este caso, pues no hay un gran interés en mezclar con precisión la mitad de cada cromosoma.

De la misma manera, en el estado del arte también se explicaron los dos tipos básicos de función de mutación: bit flip y permutación. Existen, de nuevo, incontables maneras de implementar una función de mutación, pero dada la representación binaria de este cromosoma, sumado a la falta de necesidad de algoritmos más complejos, estas son las principales opciones para este proyecto. Se escogió realizar una mutación uniforme, ya que realizar permutaciones no proporcionaba ningún beneficio tangible, y el método bit flip es similar al uniforme pero cambiando el bit siempre que se fuese a dar la mutación, cosa que tampoco interesaba, porque aumentaría significativamente la diferencia entre los cromosomas. Cabe destacar que por norma general en las mutaciones la probabilidad suele ser baja, entre un 2 y un 10 por ciento, dejando en nuestro caso al usuario poder modificar ésta probabilidad, teniendo como valor por defecto un 10%.

3.3.4. Especies, taxonomía y árboles filogenéticos

El concepto de las especies es fundamental dada la temática del estudio. En la vida real, la diferenciación entre dos poblaciones de individuos en especies diferentes es un proceso subjetivo, que establecen los científicos con el objetivo de categorizar a los seres vivos. Debido a esto hubo que idear una manera consistente de crear estas divisiones en la simulación.

La conclusión a la que se llegó es que hay que medir la diferencia entre un individuo y su especie comparando los cromosomas. Esto es tan sencillo como contar la cantidad de valores en los genes que coinciden y comparar este resultado a un porcentaje dado por el usuario; si se iguala o supera este umbral, los cromosomas se suponen lo suficientemente similares para pertenecer a la misma especie. Cuando se crea un nuevo individuo, este proceso se repite hasta que se incluye dentro de una especie existente. En caso contrario, esta criatura se considera la fundadora de una nueva especie.

Se presentan tres grandes problemas con esta metodología: el primero se da en la comparación de cromosomas previamente descritos; uno de los dos requeridos es el del individuo que acaba de aparecer, ¿pero a quién pertenece el restante? La idea original era hacer una media de los atributos y habilidades de los individuos de la especie, pero eso significa que estos se van modificando con cada microevolución de la especie, aumentando la "vida" de la especie en cuestión más allá de lo real. En otras palabras, tardaría demasiado en haber suficiente variación genética para que un individuo no se clasifique en una especie, si es que esto llega a pasar. La solución es guardar una copia del fundador de la especie, para usarlo como referencia estática a la hora de hacer la comparación genética.

El segundo problema se da en la reproducción. Puede surgir un individuo que crea una nueva especie, pero esta está destinada casi con completa seguridad a la inmediata extinción, pues no hay más individuos y por lo cual la reproducción no es posible. Para

solucionar esto, se ha hecho que los miembros de una especie puedan relacionarse con la especie progenitora y aquellas descendientes.

Por último, se encuentra el problema de la convergencia genética. Describe el caso de que un grupo de especies progresivamente den lugar a nuevas especies cada vez más parecidas, hasta converger en una misma. En este caso, dos especies con distintos orígenes pero una genética parecida, ¿son miembros de la misma? Esto se conoce como el problema del diamante, y es una pregunta que no tiene una respuesta concreta, pues es un caso prácticamente imposible en la vida real. Debido a la complejidad de los seres vivos reales es muy difícil que se dé la situación descrita, por lo que se decidió tomar una decisión subjetiva y suponer que no; cuando nace una criatura se evalúa la similitud del cromosoma con el de sus progenitores o las especies que hayan entre medias, es decir, "hermanos" en el árbol filogenético, si coincide pertenece a esa especie, si es un antecesor ya extinto, se considera una nueva especie.

La aparición y extinción de las especies se documenta progresivamente durante la ejecución del programa. La aparición de especies se acaba de describir, y la extinción se da simplemente cuando muere el último individuo de una. Para cada especie inicial se genera un "historial" en el que se guarda esta información, que se utiliza para generar un árbol filogenético que se le proporciona al usuario. Más detalles al respecto en el apartado 4.1.3.

Para presentar la información de forma legible, a cada especie se le asigna un nombre en base a su cromosoma más cierta aleatoriedad, para que dos especies representadas en el problema del diamante no se nombren de la misma manera. Cada consonante y vocal tienen una probabilidad asignada, con el propósito de volver los nombres más legibles usando más las letras más legibles. Luego se analiza cada gen para, en base a las probabilidades de las consonantes, elegir una, y posteriormente se asigna aleatoriamente una o dos vocales con una probabilidad. Dichas vocales son aleatorias en base a sus probabilidades. De forma aleatoria, una vez se ha procesado un porcentaje del número de genes aparece una probabilidad de haber un espacio, el cual se pone de forma automática cuando se ha procesado un porcentaje mayor de los genes, tras el cual la siguiente letra es mayúscula.

En conclusión y para terminar el apartado, en la Tabla 3.6. consta un pequeño resumen de las decisiones tomadas en relación con la programación evolutiva y las especies.

Representación genética	Procedimiento
Representación del cromosoma	Dividido en regiones que representan genes. Cada gen tiene un valor máximo y puede depender en un porcentaje del valor de otro/s gen/es. La parte independiente del valor de un gen se representa con bits. La suma de los bits activos del gen más los porcentajes declarados de otros genes (calculados de la misma manera) define el valor final del gen.
Función de cruce	Asignación de los bits del cromosoma copiando el valor del bit de uno de los progenitores, hasta completar el cromosoma. 50% de probabilidad entre progenitores.
Función de mutación	Mutación uniforme, se recorre todo el cromosoma bit a bit, y tras cumplirse una probabilidad, hay un 50% de probabilidad de poner un 1 o un 0.
Definición de especie	Una criatura pertenece a una especie si se da suficiente similitud genética en base a un porcentaje. En caso contrario, es una nueva especie

Tabla 3.6. Resumen de la representación genética utilizada

3.4. Conducta y pautas de comportamiento

Una de las premisas fundamentales de este proyecto es la implementación de comportamientos inteligentes que hacen de selección natural. Todas las criaturas tienen las mismas acciones y comportamientos, pero actúan de manera distinta dependiendo de sus características. A continuación se describe qué pueden hacer las criaturas y en base a qué toman la decisión de si realizar una acción o no.

3.4.1. Conducta e interacciones deseadas

El objetivo de las criaturas de la simulación es la supervivencia prolongada y la procreación para la propagación y evolución de su especie. Por ende, el comportamiento de la criatura debe complementar estos objetivos mediante la búsqueda y adquisición de recursos necesarios para su supervivencia como agua, sustento, o un lugar seguro en el que dormir; y la reproducción cuando sea posible.

En primera instancia, para implementar este comportamiento, se planteó utilizar redes neuronales, pero enseguida se descartó debido a la gran cantidad de datos que necesitan estas, lo que haría necesarios demasiados años de simulación para entrenar las redes y tener resultados medianamente convincentes. Una segunda aproximación fueron los sistemas de razonamiento basados en casos, ya que supone una implementación natural para simular un comportamiento adaptativo, y sería muy notorio de cara al usuario observar cómo los animales aprenden con el avance de los años. Pero también se

descartó esta aproximación, principalmente por la misma razón que la anterior, un número muy alto de casos para conseguir un comportamiento lógico. Además, otro inconveniente de esta implementación sería la obtención de casos "buenos" o "útiles". Se puede dar la situación de que las criaturas se desarrollen en un entorno muy beneficioso para ellas, donde no le falten los recursos que necesite, y no obtengan casos útiles para su supervivencia que le hagan comportarse de forma lógica.

Finalmente, la implementación que se eligió para el comportamiento de las criaturas fue la máquina de estados finitos. Con este sistema, las criaturas presentarían un comportamiento lógico desde un principio con el que pueden responder a los problemas que se presenten en su entorno con una configuración de estados inicial igual para todas las criaturas. La diferencia entre comportamientos entre diferentes criaturas sería dada por una "mente", que decidirá cómo responder a sus problemas según el entorno en el que se encuentre y sus propias características. El funcionamiento y la relación de estos dos sistemas se explica en los siguientes apartados.

3.4.2. Máquina de estados como motor de comportamientos de las criaturas en el mundo virtual

La máquina de estados que describe el comportamiento de las criaturas contiene una serie de estados análogos a los diferentes comportamientos implementados de las criaturas. Uno de estos estados es asignado como el activo, y, cuando es el turno de actuar de la criatura, lo hace en acorde con la acción designada en este estado activo. El estado activo cambia mediante la evaluación de una serie de transiciones, que son condiciones que unen un estado con otro y se evalúan antes de que la criatura tome una decisión: si la condición se cumple, el estado al otro lado de la transición se vuelve el activo. De esta manera se crea una red de posibles acciones a tomar por la criatura, cuya acción resultado es afectada por las variables que percibe en su entorno.

Para controlar la ejecución de acciones, se usa un sistema de recursos en las criaturas llamado *puntos de acción*. Una de las finalidades de este sistema es simular el trabajo necesario para llevar a cabo las acciones de los estados: las criaturas podrán reservar su energía para realizar acciones costosas y/o podrán realizar acciones con bajo coste consecutivamente. Por ejemplo, para una criatura con una movilidad muy alta, moverse es mucho menos costoso que para otra que es menos móvil y podrá hacerlo más a menudo. El otro objetivo es dar un mayor nivel de actividad a los seres con un metabolismo rápido. Para ello, cada ciclo de la simulación, las criaturas generan una cantidad de *puntos de acción* equivalente a el coste estándar de una acción al que se le suma una cantidad que depende del metabolismo de la criatura, siendo esta cantidad mayor cuanto más rápido sea el metabolismo. A la hora de realizar una acción, una criatura debe gastar la cantidad de *puntos de acción* asignada como coste de la acción, que puede ser modificado por las cualidades de la criatura. La criatura continúa haciendo acciones mientras tenga suficientes *puntos de acción* para realizarlas. Una vez el coste de una acción excede sus *puntos de acción* actuales, la criatura termina su ciclo, conservando los *puntos de acción* restantes para añadirlos a los obtenidos nuevamente en su siguiente turno.

Para poder implementar este sistema decidimos usar la librería Stateless[4], que nos permitirá crear máquinas de estados con estados simples y complejos. Sin embargo, en ejecución nos dimos cuenta de que esta librería no valdría para nuestro proyecto: las transiciones dentro de los macro estados no funcionaban correctamente, y al entrar de nuevo en un macro estado este no volvía a su estado interno inicial. Además, la librería no se adaptaba adecuadamente al sistema de *puntos de acción*, por lo que se decidió abandonar el uso de ésta y hacer una implementación propia. Las máquinas de estados que se han implementado siguen el esquema habitual de éste modelo, como se ha comentado anteriormente; pero, para adaptarlas al sistema de *puntos de acción*, separamos la evaluación de las transiciones y los cambios de estados de la propia acción de los estados. Así conseguimos que la máquina de estados cambie, pero sólo ejecute su acción cuando posea los *puntos de acción* necesarios para la misma. Para ello, los estados tienen un coste de *puntos de acción* necesario para ser ejecutados y su acción sólo será ejecutada cuando se disponga de los *puntos de acción*, pero se puede transicionar a otros estados sin haber ejecutado la acción, o esperar a tener los *puntos de acción* suficientes para ejecutar la acción.

3.4.2.1. Estados y macro estados

Entre los estados incluídos en la máquina se diferencian dos tipos: macroestados y estados simples. Los macro estados son aquellos que engloban situaciones generales en las que se puede encontrar la criatura, y contiene en su interior una red con la misma estructura que la máquina de estados que los contiene a ellos; de esta manera, son máquinas de menor escala que dictan el comportamiento de una criatura dentro de una misma circunstancia global. La mayoría de los macroestados o estados compuestos son una instancia de la clase *CompoundState*. Los estados simples, por otro lado, son aquellos que contienen sólo una acción que realizar cuando están activos y heredan de la clase *CreatureState*.

La máquina de estados de la criatura contiene la siguiente jerarquía de estados:

- Estado *Dead*: Un estado simple cuya acción hace que la criatura se retire de la lista de criaturas vivas del mundo. Cuando esto sucede, surge en el lugar donde se encontraba la criatura un cadáver, una entidad que hace de fuente de alimento para otras criaturas con dietas que permitan su consumo. Este tiene un valor nutricional dependiente del tamaño de la criatura original, y pasa por un proceso de putrefacción que reduce su valor nutricional y aumenta el daño que le hace al consumidor si este no es carroñero. Una vez termina su putrefacción, desaparece.

[4] https://github.com/dotnet-state-machine/stateless

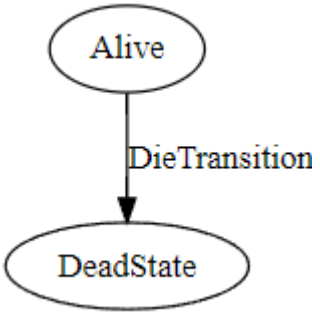


Figura 3.12. Estado Dead

- Macroestado *Alive*: Un macroestado que contiene todas las acciones que puede realizar una criatura mientras esté viva. Dentro de él se encuentran los estados y transiciones detallados en la Figura 3.13.

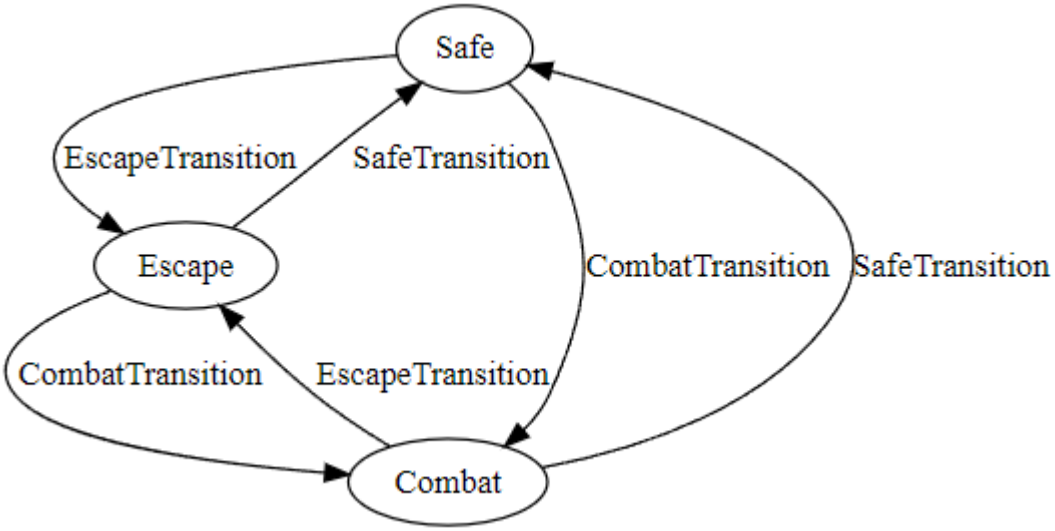


Figura 3.13. Estado Alive

  - Macroestado *Escape*: Un macroestado que contiene las acciones que toma la criatura cuando se encuentra en desventaja en una situación de combate y debe deshacerse de su atacante. Contiene, a su vez:
    - Estado *Fleeing*: Un estado simple que hace que la criatura se desplace a la posición más alejada de su atacante que pueda percibir.
  - Macroestado *Combat*: Un macroestado que encapsula todas las acciones tomadas por la criatura cuando está luchando con otra considerando que tiene posibilidades de salir victoriosa según su criterio. Contiene los estados y transiciones detallados en la Figura 3.14.

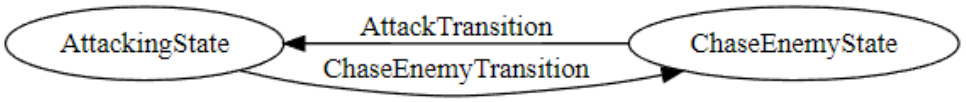


Figura 3.14. Estado Combat

- Estado *ChaseEnemy*: Un estado simple que hace que la criatura se dirija a la última posición conocida de la criatura que considera su oponente.
- Estado *Attacking*: Un estado simple que hace que la criatura intente atacar a la que considera su oponente hasta que este muera. Las fuentes de daño y vida vienen de las cualidades y habilidades de ambas criaturas.

- Macroestado *Safe*: Un macroestado en el cual se incluyen todas las acciones fuera de los macro estados anteriores; es decir, aquellas en las que la criatura no está en peligro inminente. Al entrar en este estado, la criatura calma su pánico si es que había entrado en él, pues hereda del estado compuesto *CalmState*. Dentro de este estado se encuentran los estados y transiciones detallados en la Figura 3.15.

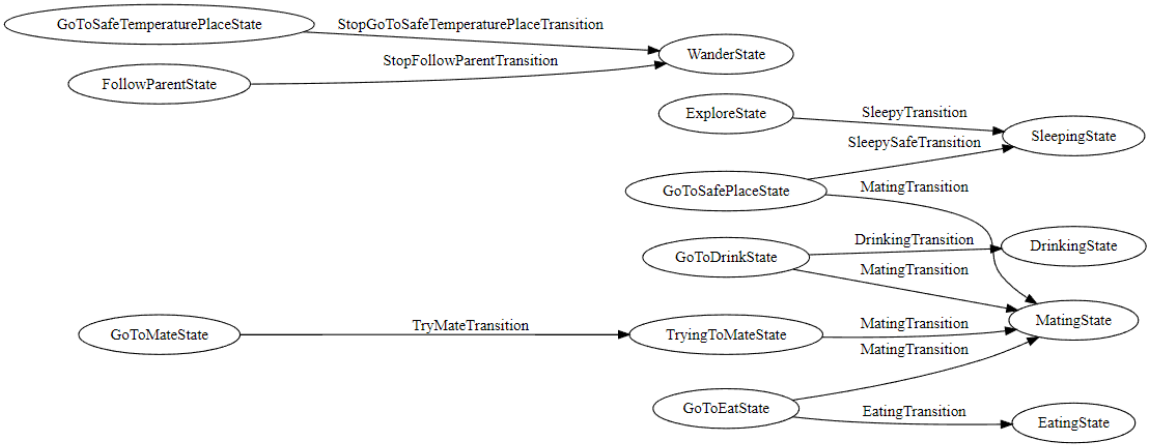


Figura 3.15. Estado Safe.
Por legibilidad, se han omitido la mayoría de transiciones del estado Wander, detalladas más adelante.

- Estado *Drinking*: Estado simple que aumenta el nivel de hidratación de una criatura mediante beber de una fuente de agua del terreno.

- Estado *Eating*: Estado simple que aumenta el nivel de energía de la criatura mediante la consumición de sustento apropiado para la misma. Según la dieta de la criatura se hace un cálculo del recurso que le es más eficiente consumir, y, una vez elegido, se interactúa con él de tal manera que parte del mismo es consumido y otorga energía a la criatura.

- Estado *Explore*: Estado simple que hace que la criatura se desplace por zonas cercanas no exploradas en busca de recursos de los cuales no tenga constancia alguna, como fuentes de agua o de alimento.

- Estado *FollowParent*: Estado simple que hace que la criatura se desplace a la posición del progenitor que ha decidido seguir. Este estado es accedido por aquellas criaturas con un alto nivel de paternidad. Si una criatura está siguiendo a su padre y éste muere o le pierde de vista, si sabe dónde está su madre la empieza a seguir. Si no sabe donde hay ninguno de sus progenitores actúa solo, hasta que deja de ser un crío y ya no entra a este estado o hasta que se cruza con uno de sus progenitores, que vuelve a seguirlo.

- Estado *GoToDrink*: Estado simple que hace que la criatura se desplace a la posición con agua que la criatura conoce a la que puede llegar de manera más eficiente.

- Estado *GoToEat*: Estado simple que hace que la criatura se desplace a la posición con sustento más cercana. Estas posiciones tienen en cuenta la dieta de la criatura, haciendo que los que sólo se alimenten del tipo de comida acorde a la misma.

- Estado *GoToMate*: Estado simple que hace que la criatura se desplace a la posición en la que se encuentra la criatura con la que se va a aparear.

- Estado *GoToSafePlace*: Estado simple que hace que la criatura vaya a una posición que recuerda como segura, sin criaturas hostiles cerca, donde podrá descansar.

- Estado *GoToSafeTemepraturePlace*: Estado simple que hace que la criatura se dirija a una zona con una temperatura apta para la misma, para que no reciba daño por estar en un entorno no apto.

- Estado *Mating*: Estado que representa el período en el cual dos criaturas están apareándose para producir descendencia. Tiene una duración que debe completarse (estando activo el estado) para que el apareamiento sea un éxito.

- Estado *Sleeping*: Estado simple que reduce el gasto energético y de agua de la criatura, poniéndola en un estado de reposo, y hace que

su nivel de descanso vaya aumentando lentamente. Además, mientras este estado esté activo, la percepción de la criatura se reduce a un 75% (configurable por el usuario) de su valor normal.

- Estado *TryMate*: Estado simple que contiene una acción realizada sólo por machos de una especie. Consiste en la solicitud de procreación del macho a la hembra, que responderá acorde con su estado de celo y sus necesidades vitales (comer, beber, dormir, huir).

- Estado *Wander*: Estado simple que hace que la criatura se mueva en direcciones aleatorias. Es el estado por defecto de la criatura, al que va cuando no tiene ninguna necesidad urgente, o ningún recurso que obtener.

3.4.2.2. Transiciones

Las transiciones son las condiciones relacionadas con un estado que se comprueban a la hora de actuar estando este estado como activo. Si una condición se cumple, se ejecuta un cambio de estado al especificado por la transición, que se vuelve el activo y se intenta ejecutar. Estas transiciones son añadidas a cada estado tras su construcción cuando se crea la máquina de estados de la criatura.

Debido a la gran cantidad de transiciones presentes en la máquina de estados, su presencia en este apartado resultaba engorrosa, sus definiciones se han movido al Apéndice, en la Tabla B.1.

3.4.3. Relación de las criaturas con el entorno físico, flora y fauna

Para poder llevar a cabo las acciones y conductas previamente descritas, las criaturas necesitan tener un cierto conocimiento sobre el mundo que habitan. Así pues, sus relaciones con el entorno dependen de lo que captan a través de sus sentidos y de lo que recuerdan haber captado.

Los seres vivos complejos, que son en este caso el tipo de organismo que se pretende describir, poseen cinco sentidos básicos conocidos como el gusto, tacto, vista, oído y olfato. En este apartado, el de la percepción, se ha hecho otra simplificación del funcionamiento de los animales, pues representar todos estos elementos de forma fiel a la realidad sería costoso tanto en rendimiento como en tiempo de desarrollo y no proporcionaría una gran diferencia en el comportamiento final de las criaturas se estima. Como consecuencia, la implementación de la percepción de las criaturas se basa en percibir los elementos que se encuentren a una distancia a su alrededor, percatándose de su mundo en un círculo en base a su posición. El radio de este círculo depende de su percepción y se ve modificado por su visión nocturna cuando sea de noche, y el si detecta a no a otras criaturas en su radio tiene en cuenta el nivel de camuflaje de estas.

Las criaturas cuentan con “tácticas de manada”, de manera que intentan defender a sus aliados a la vez que en caso de sentirse amenazadas huyen hacia estos. Si una criatura

recibe daño proveniente de otra hostil, todos los aliados de la primera capaces de luchar, es decir, con un nivel de vida aceptable, deciden si entrar en el combate. Esto lo hacen comparando la agresividad colectiva de todos los aliados capaces de luchar que han visto la agresión; si no supera la intimidación del enemigo, huyen. En caso contrario, se unen al combate atacando a la criatura hostil, que a su vez puede dar lugar a "tácticas de manada" por su parte. Sin embargo, se considera teóricamente un sistema equilibrado ya que los cazadores cazan de forma independiente, mientras que las criaturas amenazadas buscan la seguridad de sus aliados, por lo que no deberían formarse "guerras".

Una vez una criatura localiza recursos, sean estáticos, como agua o plantas o dinámicos, como otras criaturas, un factor que sí es extremadamente relevante es que esta tenga lo que se denomina "object permanence" o "permanencia de objetos". Esto hace referencia a la comprensión de que los elementos del mundo perduran existiendo aunque uno no pueda percibirlos. Para lograr esto, se diseñó una estructura que almacena temporalmente la información que adquiere la criatura, una *memoria*, junto a un sistema que establece sus prioridades en base a esta, llamada *mente*.

3.4.3.1. Sistema de memoria de las criaturas

La memoria es un sistema que se actualiza previamente al estado de la criatura, y se basa en un radio de visión, que se calcula en base a la percepción y se ve modificado de noche por la visión nocturna, y un tiempo de permanencia, que indica cuánto tiempo dura un elemento en la memoria de la criatura, calculado en base al conocimiento. Con cada paso de la simulación, la memoria se actualiza con lo percibido a su alrededor, y esta información es la que utiliza la criatura a través de la mente, la cual se explica a continuación, para tomar las decisiones en la máquina de estados.

La implementación de la memoria es problemática, pues si se intenta recordar una gran cantidad de elementos, con suficientes criaturas y cierto tamaño de mundo esto se vuelve muy costoso en espacio. Esta es la conclusión a la que se llegó tras la primera implementación que se realizó, en la que cada criatura tenía una copia vacía del mundo formada por una matriz bidimensional compuesta por un tipo específico de información que describía el entorno en una posición. Este "mapa mental" se iba rellenando según la criatura exploraba, a la vez que olvidaba aquellas partes que no percibía en una cierta cantidad de tiempo. Conceptualmente, esta es la forma más realista de enfocar una memoria, pero a parte de ser costosa en tiempo, debido a que había que recorrer una matriz del tamaño del mundo para cada criatura, actualizando sus valores, como ya se ha dicho era increíblemente costosa en espacio. En un equipo con Windows 10, procesador AMD Ryzen 9 3900X 3.8 GHz, tarjeta gráfica NVIDIA GeForce RTX 3070 Ti, RAM 2x16 GB 2400 DDR4, con un tamaño de mundo de 512 x 512 y número de individuos de 200, para un entorno de pruebas rápido, el programa ocupaba en torno a 4 o 4,5GB de memoria en ejecución, por lo que no era viable.

La solución a este problema de rendimiento es desechar la copia individual del mundo, a favor de conservar información de manera independiente, es decir, en vez de una matriz que represente en mundo donde cada casilla lleva la cuenta de aquello que se encuentre en ella, haya algo o no, guardar solo aquellas posiciones con información relevante, en

forma de listas con un límite para restringir el tamaño que puedan llegar a tener. Con esta implementación ya no solo no se guardan potencialmente grandes cantidades de datos inútiles, sino que también se simplifica la búsqueda de recursos. Con los mapas mentales había que buscar de forma progresiva, con un radio cada vez mayor hasta alcanzar en radio de percepción, la primera instancia de cada tipo de recurso que le interesa a una criatura, mientras que con listas ya tienes los recursos localizados. Además, las listas se pueden ordenar en base a un criterio, lo cual se aprovecha a la hora de calcular las prioridades de la criatura, véase el siguiente apartado, el cual explica la mente.

La memoria consta de dos tipos de datos fundamentales para representar los datos anteriormente descritos.

El primero son posiciones, que a parte de unas coordenadas y el número de ciclos (unidades de tiempo) restantes hasta ser olvidada, cuentan con información sobre el peligro que la criatura siente en dicha posición. Un peligro positivo implica que a la criatura no le interesa moverse por esa posición, pues implica un riesgo, mientras que un peligro negativo representa experiencias positivas. El peligro positivo se asigna ante la presencia de criaturas intimidantes o haber recibido daño en esa zona, sea por ataques de criaturas o debido a temperaturas extremas. Por otro lado, experiencias positivas serían haber bebido, comido o dormido sin haber sido atacado o interrumpido. Estas experiencias influyen en las posiciones cercanas, y propagan su valor de forma decreciente en base a la distancia. Por ejemplo, en una posición tiene un peligro D = 8, entonces la influencia presentada de forma gráfica sería la de la Figura 3.16.

2	2	2	2	2
2	4	4	4	2
2	4	8	4	2
2	4	4	4	2
2	2	2	2	2

$$d_x = \frac{D}{2^x}$$

Figura 3.16. Ejemplo de la propagación del peligro desde una posición

Donde x representa la distancia desde la posición original cuyo peligro es D y d representa el conjunto de posiciones influenciadas por D a una misma distancia x.

Este cálculo se aplica cuando una criatura quiere conocer el peligro de una posición; se recorren las posiciones guardadas, dividiendo el peligro de la posición entre la distancia de esta a aquella cuyo peligro se quiere conocer y haciendo el sumatorio de todos los valores. Los peligros se usan a la hora de decidir cuáles son los recursos óptimos, como

se ve en la mente en el siguiente apartado, o el camino a seguir, como se explica en 3.4.4. Cuando una posición es olvidada, se olvida con ella las experiencias que contenía.

El otro tipo de dato son recursos, que a su vez se componen de varios tipos. El más sencillo de estos recursos hace de base para la herencia de los demás, ya que para ciertos cálculos se emplea polimorfismo, y contiene una posición y unos ciclos restantes para ser olvidado. Se usa este tipo de recurso para representar el agua, ya que no es una entidad per sé en el mundo, sino una propiedad de una posición. Para guardar información sobre las entidades del mundo, se expande el concepto de recurso básico para que contenga también un identificador. Con este identificador, se le puede pedir al mundo información sobre la entidad en cuestión para poder interactuar con ella. Se utiliza para guardar información sobre plantas, cadáveres y otras criaturas. Por último, hay un último tipo de recurso que se utiliza para las presas, en caso de que la criatura no sea herbívora, que añade una variable que representa el valor relativo de esa criatura. Este valor se calcula en base a su tamaño, y por lo cual a su valor nutricional, y su distancia, de manera que cuánto más grande y cerca más aumenta dicho valor. Esto se utiliza a la hora de calcular, de ser pertinente, a qué criatura cazar. Todo lo relacionado con prioridades, sin embargo, se explica más en detalle en la mente.

Una vez establecida la estructura interna de la memoria, el siguiente factor a tratar son los tipos de datos que se guardan. Cabe reiterar que, como se explicó previamente, el peligro en una posición se ve influenciado por el resto de peligros cercanos. Esto implica, por ejemplo, que si dos criaturas potencialmente peligrosas existen cerca la una de la otra, sus peligros se superponen, generando un peligro en las posiciones cercanas mayor al que genera una sola criatura. La memoria guarda datos sobre los elementos de la Tabla 3.7.

Elemento	Descripción
Aliados	Lista con la información de las criaturas aliadas recordadas. Una criatura es aliada cuando pertenece a la misma especie que esta, a la especie progenitora o a una de las descendientes. Se ordena en base a la distancia, yendo los más cercanos primero.
Padre y madre	Los progenitores de esta criatura. Si esta criatura tiene la habilidad de paternidad, sigue a uno de sus padres hasta que se alcanza la madurez o estos mueren. También se puede dar que siguiera a un padre, este muera y no recuerde dónde se encuentra el otro, en cuyo caso actúa de forma independiente hasta alcanzar la madurez o reencontrarse con el otro progenitor. Los progenitores realmente no tienen el comportamiento de cuidar a los hijos, la habilidad de paternidad sólo implica que los hijos sigan a uno de sus padres.
Pareja	El aliado del sexo opuesto en celo más cercano que recuerda esta criatura. Es el objetivo en caso de que esta criatura busque la reproducción.
Enemigo	Representa, si hay, la criatura con la que esta se encuentra en combate. Se asigna cuando esta criatura le ha atacado o el enemigo ha atacado a esta o a uno de sus aliados. Se pierde la referencia

	cuando el enemigo muere o esta criatura huye o deja de verle. El caso de asignar un enemigo porque este atacara a un aliado solo se da cuando la agresividad conjunta de los aliados capaces de luchar (con una cantidad de vida razonable) que han visto el ataque supera el peligro en la posición del enemigo, en caso contrario no entran en combate.
Amenaza	Representa a la criatura peligrosa más cercana a esta, si hay. Una criatura considera a otra peligrosa cuando la agresividad de la primera es inferior a la intimidación de la segunda. Se utiliza para huir de forma preventiva de una criatura amenazante que se acerca demasiado.
Presas	Lista con la información de las criaturas no aliadas que ha visto esta cuyas intimidaciones no superan la agresividad de esta. Solo se actualiza si esta criatura no es herbívora. Se ordenan en base a su relación tamaño (valor nutricional), distancia, aumentando su valor cuanto más grandes son y más cerca están.
Cadáveres frescos	Lista con la información de los cadáveres en buen estado que recuerda haber visto esta criatura. Solo se actualiza si esta no es herbívora, y en caso de ser carroñera todos los cadáveres que encuentra se añaden a esta lista. Se ordena en base a la distancia, yendo los más cercanos primero. Esto se debe a la naturaleza efímera de los cadáveres, puede que percibiera uno más grande en comparación al más cercano, pero si se tiene hambre no vale la pena recorrer la mayor distancia sin la seguridad de que haya realmente una fuente de alimento.
Cadáveres putrefactos	Lista con la información de los cadáveres en mal estado que recuerda haber visto esta criatura. Funciona de la misma manera que la lista anterior y la complementa. Representa las fuentes de energía menos deseables.
Fuentes de agua	Lista con las posiciones de agua recordadas. Se ordena en base a la distancia, yendo las más cercanas primero.
Fuentes de agua seguras	Lista con las posiciones de agua recordadas en las que se ha bebido previamente de forma exitosa. Funciona de la misma manera que la lista normal de fuentes de agua. Su uso se explica en profundidad en el siguiente apartado.
Plantas	Lista con la información de las plantas comestibles recordadas. Se ordena en base a la distancia, yendo las más cercanas primero. Cuando se percibe una planta recordada y esta no tiene fruto, se borra de la memoria.
Plantas seguras	Lista con la información de las plantas comestibles recordadas que se han comido previamente de forma exitosa. Funciona de la misma manera que las la lista normal de plantas. Su uso se explica en profundidad en el siguiente apartado.
Posiciones	Lista con las posiciones recordadas con un peligro negativo. Se

seguras	ordena en base a la distancia, yendo los más cercanos primero.
Posiciones con temperaturas seguras	Lista con las posiciones recordadas con una temperatura aceptable para la criatura. Si una criatura está en una posición con una temperatura que no puede soportar, esto genera peligro positivo que se tiene en cuenta a la hora de calcular caminos. Sin embargo, una lista con las posiciones ordenadas por distancia es conveniente cuando se está muriendo a causa del clima.
Regiones exploradas	Diccionario con las regiones del mapa que recuerda haber recorrido. Éstas regiones se corresponden con las usadas para la navegación. No siguen un orden específico.

Tabla 3.7. Elementos a recordar para una criatura

Por último, falta explicar el funcionamiento de la mente: Recordar consiste sencillamente en procesar cada entidad percibida. Esto consiste en actualizar su información en la lista pertinente, fijando su posición, los ciclos restantes hasta el olvido al máximo, etc. si ya se encontraba en ella el recurso, o añadirla a esta en caso contrario. Este proceso se repite para cada entidad, y una vez no quedan elementos que percibir las listas se ordenan en base a un criterio, que depende del tipo de recurso, y se retiran los últimos elementos de las listas que superen el tamaño máximo establecido, ya que estos son los menos valiosos.

Durante un ciclo, la criatura tiene que hacer comprobaciones con el propósito de tomar decisiones, y para eso accede a la información almacenada en la memoria. Cuando quiere visitar un recurso, busca en su memoria la posición recordada y va a este. Es importante aclarar que toda la información que se guarda tiene una posición asociada, ya que es incoherente, por ejemplo, pedirle al mundo la posición de otra criatura, pues no son omniscientes. Mientras una criatura ve un recurso actualiza cada ciclo su información, incluyendo su posición, y cuando deja de percibirlo, deja de actualizarla. Posteriormente, la memoria devuelve las posiciones recordadas cuando se le pide información, y es solo cuando se está cerca de una posición (porque si no se olvida como se explica a continuación) que se utiliza el identificador de una entidad para interactuar con ella.

En último lugar, olvidar consiste en retirar de la memoria información que lleva una cierta cantidad de tiempo sin ser actualizada, lo cual se hace con una sencilla comprobación. Si se da el caso, el recurso olvidado se elimina de las listas, y por lo cual ya no se contempla en la toma de decisiones. Sin embargo, también se olvidan los recursos cuya información guardada en la memoria resulta errónea. Un ejemplo evidente de esto sería el recuerdo de que una criatura se encuentra en una posición, pero al volver a visitarla tras un cierto tiempo la criatura ya no está allí, sea porque se ha movido o porque ha muerto. Esto puede resultar problemático ya que solo se debe borrar aquella información que se demuestre errónea. En caso anterior, si la criatura recordada muere sin ser vista no tiene sentido que esa información se elimine de forma inmediata, pues las criaturas no son omniscientes, por lo que solo debe pasar cuando se detecte una

incongruencia (esta criatura debería estar aquí pero estoy viendo que no está). En el contexto de la memoria, este caso se da cuando la posición recordada de una entidad se encuentra dentro del círculo de percepción de una criatura, pero los ciclos restantes de dicha entidad hasta ser olvidada no se corresponden con el máximo posible. En otras palabras, se está viendo la posición en la que se recuerda la entidad, y sin embargo su información no se actualiza porque no se ha percibido su presencia.

3.4.3.2. Sistema de prioridades razonadas o "mente"

La mente es el siguiente paso conceptual de la memoria; si la memoria guarda la información que la criatura recoge al navegar por el mundo y la ordena para facilitar su acceso, la mente utiliza dichos datos para establecer las prioridades de la criatura. Cuando se habla de prioridades se hace referencia al recurso prioritario relacionado con una decisión. La mente analiza y establece los siguientes:

- Progenitor a quien seguir: Si tiene la habilidad de paternidad, una criatura sigue a uno de sus padres hasta alcanzar la madurez, ambos mueran o muera el que estaba siguiendo si se olvidara de la posición del otro. Se asigna uno aleatoriamente entre el padre o la madre al nacer la criatura, pero se asigna nuevamente cuando este muere y encuentra al otro progenitor.

- El cadáver prioritario: Dadas las condiciones de la criatura, representa cuál es el cadáver que quiere comerse. Sabiendo que las listas de cadáveres están ordenadas como se describió previamente, el criterio es el siguiente: por defecto, se escoge el primer cadáver fresco en memoria; si no hay y la criatura está cerca de la inanición, se escoge el primer cadáver putrefacto que haya en memoria; si no se recuerda ningún cadáver o solo se recuerdan putrefactos pero no se está cerca de la inanición, el cadáver prioritario se considera nulo.

- La planta prioritaria: Dadas las condiciones de la criatura, representa cuál es la planta que quiere comerse. De nuevo, con las listas ordenadas, el criterio es el siguiente: si no se recuerda ninguna planta, la prioritaria se considera nula; si la criatura está cerca de la inanición, se escoge la más cercana entre las plantas normales recordadas y las seguras; si no se está cerca de la inanición, se escoge la planta más valiosa entre las dos listas siguiendo un criterio. El criterio se basa en el cálculo de una relación entre el peligro de la planta y su distancia. Ambos factores se "normalizan", haciendo que sus valores vayan de 0 a 1. Las distancias se dividen entre la máxima de las dos, de forma que si la distancia A es mayor que la distancia B, las relaciones son $A / A = 1$ y $B / A < 1$.

  Los peligros funcionan de forma similar, pero al poder ser negativos se necesita eliminar la restricción inferior, dejando como única condición que sean mayores de 1, para que puedan tener un influencia positiva en caso de ser una posición segura. Para lograr esto, se añade otro paso al cálculo: sea C un peligro mayor que D, antes de calcular las relaciones se calcula el máximo entre C y un valor pequeño en relación a los posibles valores de peligro, como 0,1. De esta forma, si C y D son positivos, la relación funciona de la misma forma que la de la distancia. Sin embargo, si C es positivo pero D negativo, C es el peor caso en peligro,

representado con 1, mientras que D es mejor que un valor positivo o neutral y por lo cual menor que 0; si C y D son negativos, mantienen su relación de tamaño al ser divididos por el mismo número, aunque se deforma la magnitud. Se considera que esta deformación en este caso no es tan relevante ya que al ser ambas opciones seguras ambas son aptas en gran medida, y por lo cual una cierta cantidad de error es justificable. Se muestran ejemplos en la Figura 3.17.

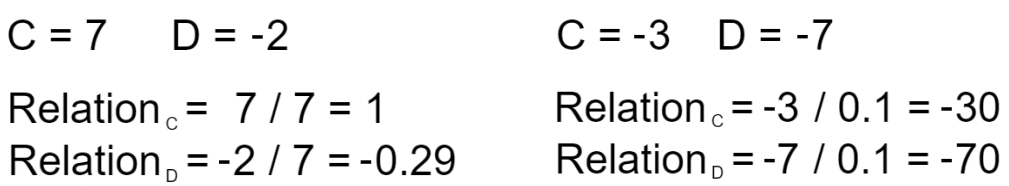


Figura 3.17. Ejemplo de las fórmulas para las relaciones

Así pues, con los valores "normalizados", se suman los de la distancia y el peligro, siendo el menor el más apto. Este proceso se repite con cada elemento en cada lista hasta encontrar los dos óptimos, y luego con estos, para encontrar el recurso más apto.

- El cadáver y la planta prioritarios se mantienen separados como forma preventiva en caso de que se quiera utilizar uno u el otro, y no el óptimo en base a un criterio preestablecido.
- La fuente de agua prioritaria: Mismo funcionamiento y criterio que las plantas, representa la fuente de agua más atractiva de forma lógica para una criatura.

La mente hace de wrapper para la memoria, añadiendo funcionalidad y haciendo de intermediario entre la criatura y su memoria, limitando también el acceso a la última de forma externa. Como consecuencia, la mente es la fuente de información sobre el estado interno de la criatura a la hora de seleccionar objetivos, y se usa principalmente en la máquina de estados para crear los comportamientos de cada criatura.

3.4.4. Navegación por el mundo simulado

Las criaturas deben ser capaces de moverse por el mundo, esquivando obstáculos y moviéndose según su capacidad de movimiento. En la simulación hay tres posibles formas de movimiento: terrestre, arbóreo y aéreo. La capacidad de moverse por estas "alturas" depende de si la criatura puede escalar o volar, y todas las criaturas se pueden mover por el suelo. Los caminos se calculan de dos maneras diferentes, utilizando *A** de dos niveles o el algoritmo de Bresenham. Ambos se han explicado previamente en el apartado 2.4. perteneciente al estado del arte.

El primer método se utiliza cuando la criatura hace su movimiento por tierra o sobre árboles y el segundo cuando se mueve volando. La toma de decisiones que dicta cuándo se elige un cálculo u otro depende de las características de las criaturas, de la distancia a

la que se encuentra su objetivo y de la densidad de árboles que hay hasta el objetivo, que se calcula utilizando una heurística o lógica propia en el cálculo del camino. Para ello se busca la intersección entre las Ecuaciones 3.4 y 3.5, encontrando así el umbral de distancia a partir del cual es mejor ir volando que por tierra. En estas ecuaciones, $dist$ es la distancia del camino, $\rho_{arborea}$ es la densidad arbórea del camino, $h_{arborea}$ es la altura de la capa arbórea, $v_{arborea}$ es la velocidad arbórea de la criatura, v_{suelo} es la velocidad por suelo de la criatura, h_{aerea} es la altura de la capa aérea, v_{aerea} es la velocidad aérea de la criatura.

$$Coste_{Aereo}(dist) = \left(\frac{dist + 2 \cdot h_{aerea}}{v_{aerea}}\right)$$

Ecuación 3.4. Función del coste de moverse de forma aérea

$$Coste_{Terrestre}(dist, \rho_{arborea}) = \left(\frac{dist + 2 \cdot \rho_{arborea} \cdot h_{arborea}}{v_{arborea} \cdot \rho_{arborea} + (1 - \rho_{arborea}) \cdot v_{suelo}}\right)$$

Ecuación 3.5. Función del coste de moverse de forma terrestre y arbórea

Para la navegación por tierra se utiliza el algoritmo de A^* en dos niveles: uno a alto nivel utilizando un mapa de regiones creado utilizando un diagrama de Voronoi explicado en el apartado 2.3.5. y uno a bajo nivel dentro de cada región.

El cálculo del mapa de regiones se planteó de dos maneras distintas: en un primer momento se consideró utilizar el algoritmo de Fortune ya que tiene un coste de tiempo de $O(n\,log(n))$, que es bastante eficiente. Sin embargo, debido a que el mapa tiene obstáculos que las criaturas no pueden atravesar, principalmente masas de agua, y estos no deben considerarse parte de ninguna región, se optó por el segundo algoritmo planteado. Este consiste en dividir el mapa en trozos cuadrados y en cada trozo generar un punto colocado aleatoriamente, mientras que la posición se corresponda en el terreno a suelo. Una vez los puntos se colocan, se expanden de forma iterativa recogiéndose como una cola, y colocando los puntos nuevos al final. La expansión de los puntos es recorrer los puntos ortogonales y diagonales del punto actual y comprobar si estos son agua o pertenecen a otra región. Si no es así, se añaden los nuevos puntos a la cola y se marcan como pertenecientes a la región. Si pertenecen a otra región, se establece un enlace entre las dos regiones. Este método tiene un coste mayor, $O(n^2)$, pero solventa el problema que planteaba el algoritmo de Fortune.

El algoritmo a alto nivel utiliza estas regiones como nodos y las conexiones entre regiones como aristas de un grafo que ocupa todo el mapa del mundo. La heurística

utilizada para *A** es la de Manhattan ya que el movimiento es similar al movimiento en una cuadrícula y esta heurística es la que mejor aproxima el coste.

Por otro lado, el algoritmo a bajo nivel utiliza las posiciones de la matriz del mundo como nodos, incluyendo las alturas; es decir, cada posición de la matriz del mundo puede atravesarse por tierra o sobre los árboles. Las aristas del grafo representan como el salto de una posición a otra, en diagonal u ortogonal, o como el cambio de medio en una misma posición, pasando del suelo al aire, por ejemplo. Además, tiene en cuenta el peligro del vértice, el cual es la presencia e intimidación de otras criaturas cerca del vértice, aumentando el coste de ese nodo si hay peligro. La heurística utilizada es una creada específicamente para esta simulación, que tiene en cuenta la densidad aproximada de árboles hacia la posición objetivo y si la criatura se mueve más rápido por encima de los árboles que por tierra. El coste que devuelve esta heurística es la distancia mayor de los dos ejes de movimiento, ya que el coste de movimiento en diagonal y ortogonal es el mismo, multiplicado por la velocidad en tierra (GS) y dividido por la velocidad sobre árboles (AS) si decide ir por encima de estos, o al revés si va por tierra.

$$f(x,y) = \begin{cases} max(abs(x),\ abs(y)) \cdot GS/AS & : GS > AS \\ max(abs(x),\ abs(y)) \cdot AS/GS & : GS \leq AS \end{cases}$$
$$GS, AS \in \aleph$$

Ecuación 3.6. Coste de la heurística de *A** de bajo nivel

El movimiento aéreo funciona de forma distinta, sin embargo, ya que su movimiento no se ve restringido por obstáculos como masas de agua o flora. Se considera que las criaturas vuelan por encima de cualquier obstáculo, por lo que no necesitan esquivar posiciones de forma inteligente y moverse de forma lineal resulta no solo lógica suficiente, sino también óptima. Para generar esa línea recta, dado que las posiciones del mundo son casillas en una matriz cuadrada, se utiliza el algoritmo de Bresenham modificado para que devuelva las posiciones de la línea ordenadas de forma creciente en distancia a la criatura. Si la criatura no comienza su navegación en el aire, se le añade al comienzo de la línea el movimiento para empezar a volar, que consiste en moverse en la misma posición cambiando la altura a aire y al final el movimiento de detenerse en la altura objetivo.

4. Datos del programa y demostración de usabilidad

En esta parte del documento se va a hablar concretamente sobre aquello relacionado con el inicio, ejecución y resultados del programa, describiendo cómo estos pasos son configurables y cómo se han optimizado. También, para demostrar la usabilidad de este proceso, se describe en detalle cómo se ha realizado el prototipo de demostración en Unity.

4.1. Información de entrada, ejecución y datos resultantes

4.1.1. Configuración de la simulación

Uno de los objetivos durante el desarrollo ha sido que el programa que refleje el estudio realizado sea configurable dentro de lo posible. Esto se debe a la motivación del proyecto: crear un generador genérico que se pueda usar en una pluralidad de contextos. Para lograrlo, este ha de ser personalizable, y por lo cual una gran cantidad de datos los puede retocar el usuario como deseé. Sin embargo, el programa tiene una serie de valores que se utilizan por defecto hasta que el usuario los modifique. Se pueden configurar varios apartados del proyecto que ya se han descrito anteriormente en el documento a través de archivos, los cuales son:

- El mundo en el que se realiza la simulación. Por defecto, el programa tiene una configuración por defecto compuesta por una serie de valores y funciones que definen cómo se genera el mundo. Si el usuario no proporciona un archivo que represente el mundo, se genera uno nuevo aleatorio con esta configuración. Sin embargo, el usuario puede proporcionar un mapa de alturas configurado como una matriz bidimensional en un archivo JSON que determina el tamaño, elevaciones y masas de agua del mundo, a partir del cual haciendo uso de la configuración previamente mencionada se generan el resto de mapas necesarios para la simulación del mundo. Por otro lado, si ya se ha realizado una simulación con anterioridad y se desea, el usuario puede reutilizar un mundo ya creado, siempre y cuando proporcione los dos mapas que genera la simulación al terminar su ejecución (el mundo como tal y el mapa de regiones en el que este se divide).

Además, se puede proveer una configuración diferente a la que hay por defecto para la generación del mundo. Para ellos, se utiliza la clase *WorldGenConfig* que contiene los siguientes parámetros que se pueden modificar para la generación del mundo: el tamaño del mapa, el tipo de mapa, las funciones para el ruido de Perlin, los mapas de altura, humedad y temperatura, y las funciones de evaluación y suavizado. Ésta clase requiere de programación y no se puede configurar mediante un archivo, ya que requiere de la definición de funciones. En el caso de las simulaciones, se utiliza una configuración diferente si se provee de un mapa de alturas, como se ha comentado anteriormente.

El programa primero comprobará que si hay archivos de configuración del mundo y de sus regiones, si no lo hay comprueba si hay mapa de alturas para la generación del terreno; si tampoco está presente este archivo se generará un mundo con la configuración por defecto.

El archivo que representa el mundo se debe de llamar *World.json* para que sea procesado por el programa. El archivo debe de estar estructurado como una matriz cuadrada de objetos donde cada objeto tiene la información presente en la Tabla 4.1.

Atributo	Descripción
height	Altura en ese punto.
humidity	Humedad en ese punto.
temperature	Temperatura en ese punto.
flora	Probabilidad de flora en ese punto.
plant	Objeto que representa la información de una planta con los siguientes apartados: ● *type:* Tipo de la planta. ● *eaten:* Si ha sido comida por completo. ● *x:* Posición en x. ● *y:* Posición en y. ● *id:* Identificador de la entidad. ● *maxHp:* Salud máxima. ● *currHp:* Salud actual.

isWater	Si la casilla es agua.
regionId	Región de Voronoi a la que pertenece.

Tabla 4.1. Atributos de cada objeto de *World.json*

Existen ejemplos sobre el archivo del mundo en el apartado A del apéndice.

El archivo que representa el mundo se debe de llamar *RegionMap.json* para que sea procesado por el programa. El archivo debe de estar estructurado como un array de objetos donde cada objeto tiene la información de la Tabla 4.2.

Atributo	**Descripción**
spawnPoint	Objeto que representa el punto original de una región. Tiene los siguientes apartados: ● *X:* La posición en el eje X del punto. ● *Y:* La posición en el eje X del punto.
links	Diccionario que guarda las regiones colindantes en forma de su identificador, y las casillas que forman la frontera en forma de su posición.

Tabla 4.2. Atributos de cada objeto de *RegionMap.json*

Existen ejemplos sobre el archivo de regiones del mundo en el apartado A del apéndice.

Estos dos archivos se pueden usar cuando se quiere repetir una simulación en un mapa ya exportado en una simulación anterior. También se podría crear desde cero, pero significaría un esfuerzo notable para el usuario.

El archivo que representa el mundo se debe de llamar *HeightMap.json* para que sea procesado por el programa. El archivo debe de estar estructurado como una matriz de números en coma flotante en la que cada número indica la altura de la casilla que le corresponde.

Existen ejemplos sobre el archivo de mapa de alturas en el apartado A del apéndice.

- La genética, concretamente el cromosoma, cuyo tamaño se calcula dinámicamente en el código como se explica en el apartado 3.3.3. El cromosoma siempre consta de una serie de atributos y habilidades, pero no de una estructura fija. Esto quiere decir que no necesariamente el valor máximo de un atributo es el que existe por defecto, y de la misma manera el mismo atributo no tiene porqué depender concretamente de otro atributo en una cierta cantidad. Esta información, valores y dependencias, se describen en un archivo JSON que existe por defecto y es necesario para la ejecución del programa. Se ha intentado hacer lo más legible posible, para facilitar que el usuario lo retoque como quiera sin complicaciones innecesarias. Un ejemplo de esta accesibilidad es que los atributos, que se representan internamente a través de una enumeración, se representan en el archivo con un nombre comprensible fuera del contexto del código.

  El archivo que representa el cromosoma se debe de llamar *Chromosome.json* para que sea procesado por el programa. El archivo debe de estar estructurado como un array de objetos donde cada objeto tiene la información presente en la Tabla 4.3.

Atributo	Descripción
feature	El nombre de la cualidad que se está modificando, ya sea un atributo o una característica de la criatura.
maxValue	El valor numérico máximo correspondiente a esa cualidad.
relations	Un array de objetos donde cada objeto representa la relación entre otras cualidades y ésta, representado con la siguiente información: ● "feature": El nombre de la cualidad con la que está relacionada. ● "percentage": Cómo influye esta cualidad a la actual. Es un valor numérico entre -1 y 1

Tabla 4.3. Atributos de cada objeto dentro de *Chromsome.json*

Existen ejemplos sobre el archivo del cromosoma en el apartado A del apéndice.

También referente a la genética, se puede proveer también de un archivo en el que se configuran los valores de habilidad necesarios para poder utilizar cada habilidad. Estas configuraciones toman valores entre 0 y 1, representando la cantidad relativa de habilidad necesaria para poder desbloquearla. Por ejemplo, un valor de 0,3 en la habilidad de "Alas", teniendo ésta un máximo de 10, significa que si la criatura tiene un valor mayor o igual que 3, podrá usar las alas y volar.

El archivo que representa el los desbloqueos de las habilidades se debe de llamar *AbilityUnlock.json* para que sea procesado por el programa. El archivo debe de estar estructurado como un array de objetos donde cada objeto tiene la información que aparece en la Tabla 4.4.

Atributo	Descripción
Item1	El nombre de la habilidad.
Item2	La cantidad relativa de habilidad para ser desbloqueada.

Tabla 4.4. Atributos de cada objeto dentro de *AbilityUnlock.json*

Existen ejemplos sobre el archivo de habilidades en el apartado A del apéndice.

- Los "parámetros universales". Este es un término que se ha acuñado internamente para hacer referencia a las muchas variables que existen en la simulación que necesitan un valor pero este es subjetivo y por lo cual no depende del mundo, de los cromosomas, etc. Contiene los parámetros relacionados con toda la simulación y aspectos más generales de las criaturas y del entorno. Ejemplos son el tiempo mínimo que dura el ciclo vital de una criatura, los valores mínimos y máximos de varias cualidades ajenas al cromosoma, como el tiempo entre el celo de las criaturas, el porcentaje de la percepción original que se mantiene de noche por defecto, el tamaño máximo de las camadas… Todas estas variables existen por defecto en el programa con valores que se creen convenientes o por lo menos funcionales, pero se pueden declarar en un archivo JSON y si el programa lo encuentra sobreescribirá estos valores con aquellos descritos en este.

  El archivo que representa estos parámetros se debe de llamar *UniverseParameters.json* para que sea procesado por el programa. El archivo debe de estar estructurado como un array de único objeto donde cada entrada representa el nombre del parámetro y su valor numérico. Debido a la gran cantidad de atributos de éste archivo, su descripción se ha movido a la tabla B.2 del apartado B del apéndice.

El contenido de los archivos de modificación es validado al ser introducido en el programa. Esto quiere decir que se comprueba que los valores que se introducen cumplen los requisitos que se esperan de los mismos. Para ello se utiliza la clase *Validator* que contiene las funciones necesarias para validar cada tipo de archivo de entrada. Si algún archivo no se ajusta a lo esperado, el *Validor* informa del tipo de fallo.

Los archivos de configuración se han de proveer en la misma carpeta al programa, siendo únicamente necesario para la simulación el archivo de cromosoma . Si no se

proporcionan el resto de archivos, los valores de la simulación que se modifican con los archivos tendrán una configuración por defecto, como se ha comentado anteriormente.

4.1.2. Ejecución del programa

Un factor muy importante a tener en cuenta es que este programa no está pensado para ser ejecutado de forma dinámica durante otro proceso, en otras palabras, si se quiere utilizar la información resultante en otro proyecto, este tiene que ejecutarse con anterioridad. Esto se debe al tiempo de ejecución: 100 criaturas en un mundo 512 x 512 de un mapa *Default* durante 10 años tarda estimadamente 33 horas. Como consecuencia, se debe ejecutar el programa con anterioridad, y exportar la información al proyecto en cuestión. La interpretación de los datos depende del usuario, pero lo normal es que el usuario pueda ver el progreso de la simulación de alguna forma. Como se vió en el apartado 3.1, hay dos ejecutables que se encargan ofrecer una mínima interfaz al usuario y de dar retroalimentación sobre el progreso de la simulación. Su comportamiento prácticamente es equivalente, pero cabe destacar que el ejecutable por consola provee de una mayor información durante la ejecución de la simulación. Además, en el ejecutable de consola, el avance de la barra de progreso es relativo al tiempo total de simulación, mientras que en el ejecutable de ventana, el avance de la barra se produce con cada año de simulación. A continuación se explica el funcionamiento de ambos programas.

Una vez comienza la ejecución del programa pero antes de la ejecución de la simulación, se solicita al usuario que configure la ruta del directorio dónde se encuentran los archivos de entrada vistos en el apartado 4.1.1.

```
Input a valid directory containing the necessary data files
for the program:
```

Figura 4.1. Solicitud de la ruta de los archivos de entrada por consola

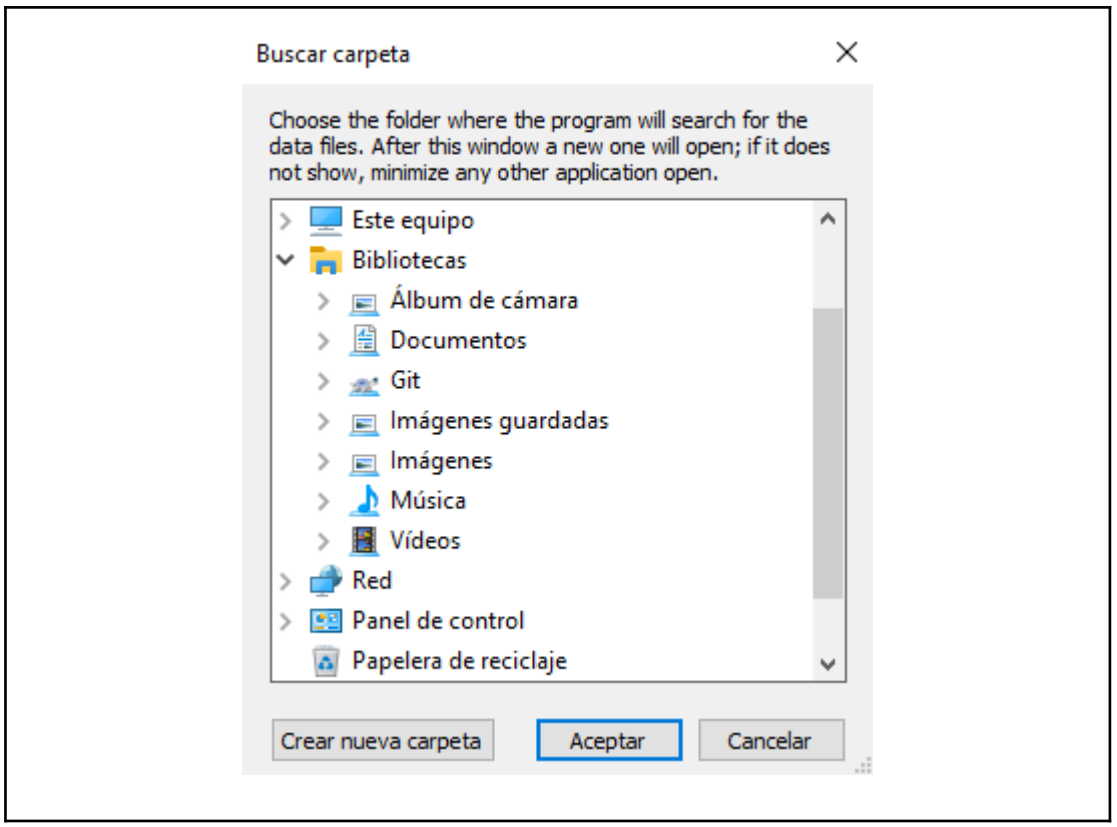


Figura 4.2. Solicitud de la ruta de los archivos de entrada por ventana

Justo después se requiere otra ruta a un directorio, que será donde se guarden los archivos de salida tras la simulación, que se verán en el apartado 4.1.3.

```
Input a valid directory in which the resulting data will be
saved:
```

Figura 4.3. Solicitud de la ruta de los archivos de salida por consola

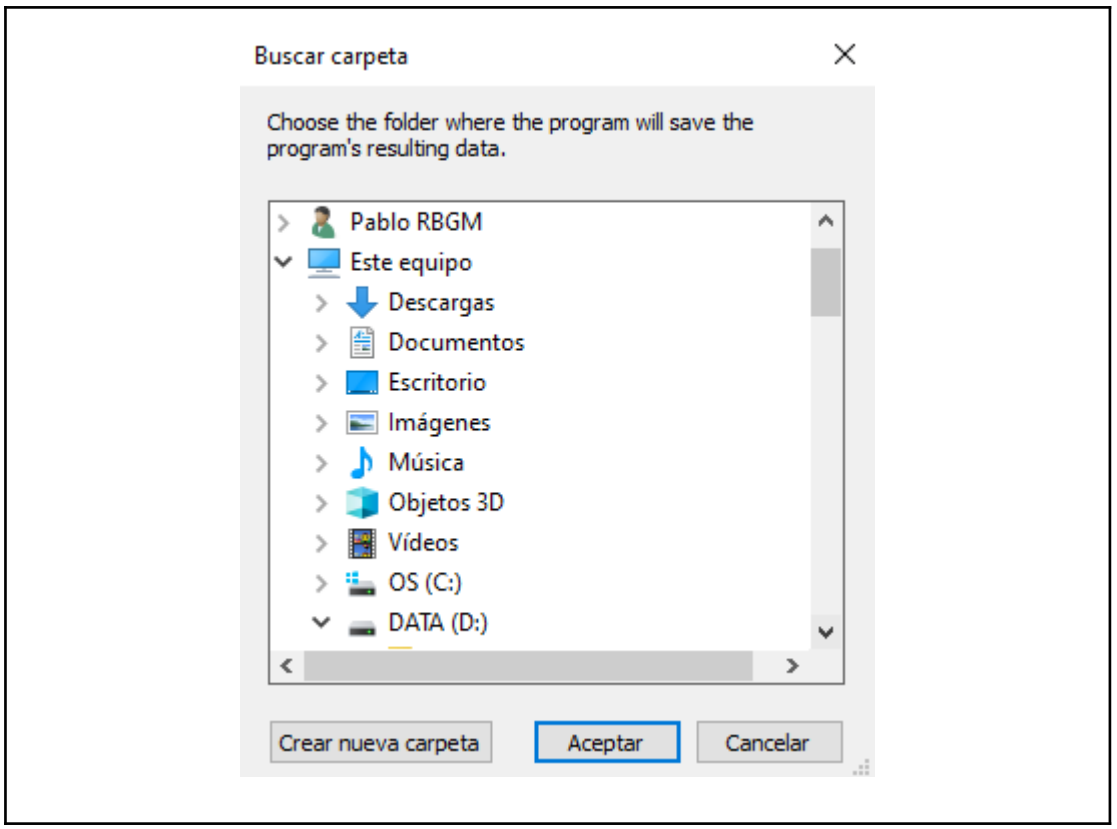


Figura 4.4. Solicitud de la ruta de los archivos de salida por ventana

Los siguientes dos pasos sólo ocurren si en el directorio con los archivos de entrada no se encuentra ningún archivo de configuración del mundo, por lo que es necesario preguntar al usuario sobre la configuración que considere para la generación del mundo. Se requiere el tamaño y el tipo de mundo a generar.

```
Input how big in squares the world is going to be. Must be
a number larger or equal to: 100
```

Figura 4.5. Solicitud del tamaño del mapa por consola

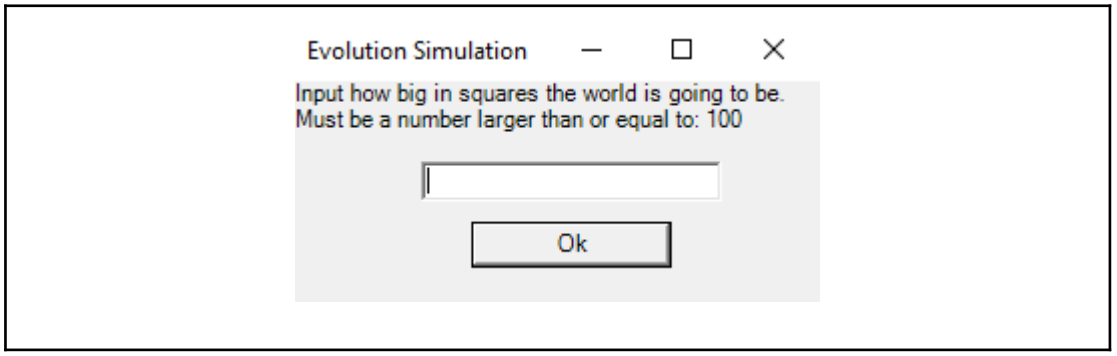


Figura 4.6. Solicitud del tamaño del mapa por ventana

```
Input the number of the type of map that should be
generated:
<0> Default | <1> Island | <2> Atoll | <3> Swamp | <4>
Custom
```

Figura 4.7. Solicitud del tipo de mapa por consola

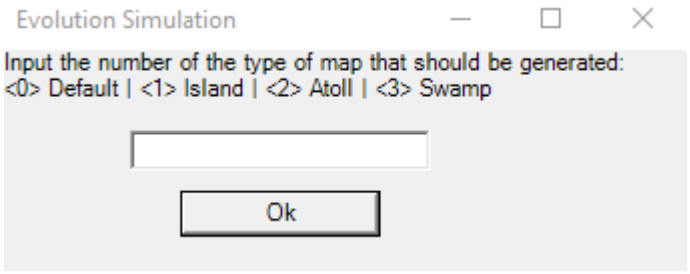


Figura 4.8. Solicitud del tipo de mapa por ventana

A continuación, se solicita información referente a la duración de la simulación y su estado inicial. En primer lugar se requiere el número de años que se quiere simular.

```
Input how many years of evolution are going to be
simulated:
```

Figura 4.9. Solicitud de la duración de la simulación por consola

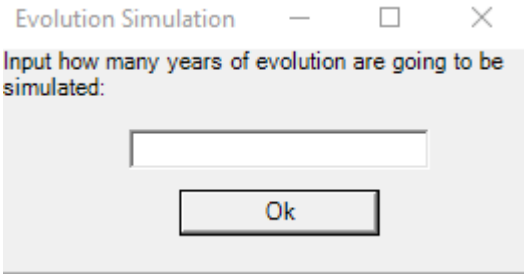


Figura 4.10. Solicitud de la duración de la simulación por ventana

Después, el número de especies originales que van a poblar inicialmente el mundo.

```
Input how many species are going to be created initially.
Must be a number larger than or equal to: 5
```

Figura 4.11. Solicitud del número de especies iniciales por consola

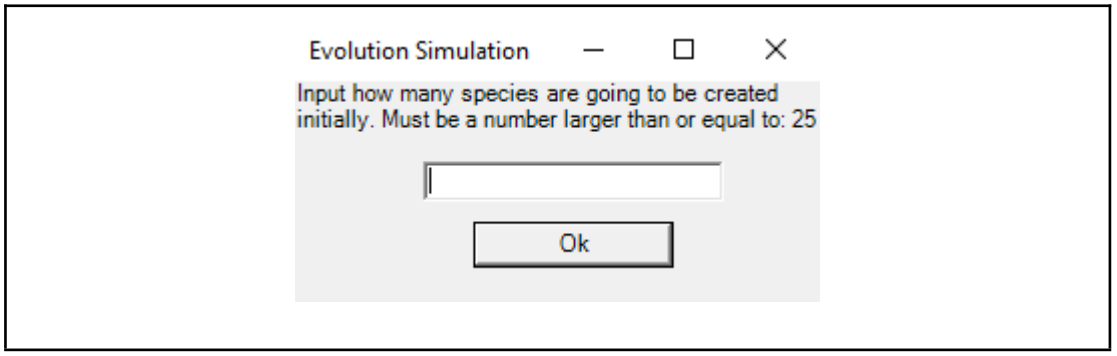


Figura 4.12. Solicitud del número de especies iniciales por ventana

Luego, el número de individuos que van a constituir inicialmente a cada especie. Éstos individuos empezarán siendo adultos para que el inicio de la simulación sea más dinámico.

```
Input how many individuals per species are going to be
created. Must be a number larger or equal to: 2
```

Figura 4.13. Solicitud del número de individuos por especie por consola

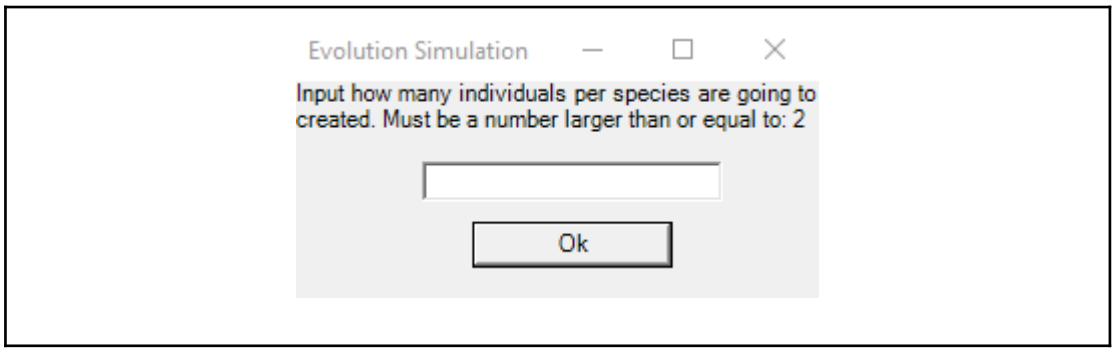


Figura 4.14. Solicitud del número de individuos por especie por ventana

Por último, si toda la información introducida por el usuario es correcta, la simulación comienza. Se muestra en ambos casos una barra de progreso para informar al usuario del avance de la simulación.

```
[......                                                    ]
6%
Ticks: 4818/87600 | NumCreatures: 1 | Births: 0 |
Entities to Update 4 | 00:00:00.2123291
```

Figura 4.15. Progreso de la simulación en consola

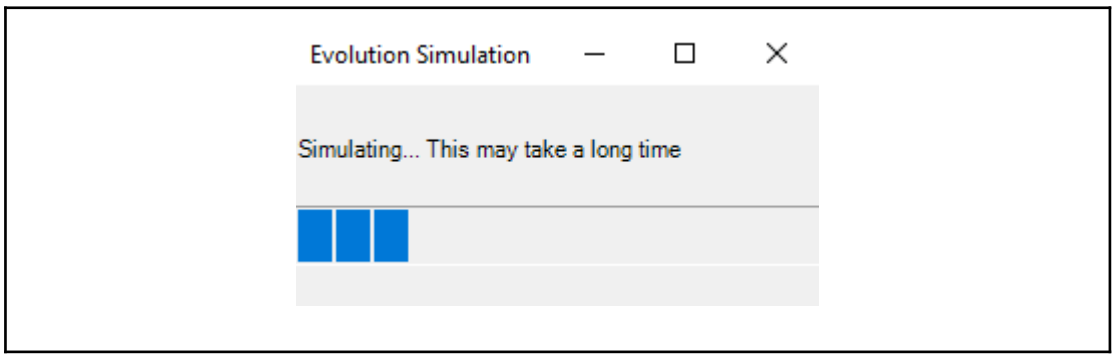


Figura 4.16. Progreso de la simulación en ventana

Tras comenzar la simulación no se requiere ningún input del usuario; es un proceso automático que se realiza durante una cantidad de tiempo relativamente grande, y simplemente se concluye el proceso una vez se termina la simulación. En caso de que, dadas las condiciones para esta, todas las criaturas se mueran de forma prematura, es decir, antes de que se pudieran simular la cantidad de años estipulados, se actúa como si acabase la simulación, exportando el mismo número de especies que con las que se ha iniciado la simulación, exportando las que más tiempo hayan estado vivas.

Una ejecución completa del programa en su estado inicial, una vez se corrigieron los errores que impedían el correcto funcionamiento del programa, estimaba una duración de 31 días. La configuración para esta estimación era la misma que se comenta al principio de éste apartado: un mapa tipo *Default* de 512x512 casillas y 200 criaturas iniciales. Ésto supone una duración notablemente más alta que la actual. Se logró agilizar la ejecución haciendo uso de un *Profiler*, que permitió el análisis de rendimiento durante la ejecución del programa, permitiendo ver dónde se perdía la mayor cantidad de tiempo. Una vez se tuvo conocimiento de estos fallos, se optimizaron cuando fue posible los cálculos y las estructuras de datos, dando como resultado una mejora de rendimiento de un 2254%.

4.1.3. Datos resultantes

Se ha mencionado varias veces a lo largo del documento que el objetivo de este programa es exportar tres tipos de datos, todos relacionados con las criaturas y las especies que componen debido al tema de estudio. Son los siguientes:

- El mundo, como dos archivos JSON. Uno es el mapa de regiones, que se necesita para reutilizar el mundo en otra simulación, y el otro es el mundo como tal. Ambos archivos tienen los nombres y la estructura especificada en el apartado 4.1.1. Se exportan ambos archivos por su utilidad para crear repetir una misma simulación con las criaturas también exportadas, o para realizar nuevas simulaciones en un mundo ya configurado. También puede resultar útil ver las

características del terreno generado si se ha creado con un mapa de alturas previo. Ejemplos de estos archivos se encuentran en el apartado A del apéndice.

- La información de las especies vivas cuando termina la simulación. Se exportan todas las criaturas que estén vivas al acabar la simulación, y en caso de ser menos de las especies con las que se ha iniciado la simulación, se exportan las que hayan existido durante ésta que más tiempo hayan existido. Cada especie se exporta como un archivo JSON y contiene la información del representante de la especie, la cual contiene dos elementos: el nombre de la especie y las cualidades dado el cromosoma original. El cálculo de estas cualidades se hace en base a la clase instanciada, como se explicó anteriormente en este capítulo. No se incluye en la información a exportar ni el cromosoma base de la especie ni los atributos y habilidades que describe, pues se consideró que son elementos estrictamente relacionados con la programación evolutiva y por lo cual no especialmente útiles fuera de la simulación. Se entiende que si se quiere evolucionar las criaturas se usaría el código del simulador, el cual también se proporciona. Las cualidades, por otra parte, describen cómo se relaciona la especie con su entorno y con cuánta competencia, así que se supone información más práctica. La interpretación de estas cualidades recae en el usuario del programa. La incorporación de los elementos excluidos, de desearse, sería fácilmente incorporada sin embargo.

  Los archivos de cada especie tienen como nombre *Species_X.json* donde *X* es un número que representa el orden en el que se han exportado. Éstos archivos están estructurados como un único objeto con la forma que aparece en la Tabla 4.5.

Atributo	Descripción
name	El nombre de la especie
stats	Cualidades de la especie, obviando los aspectos internos del cromosoma que se han comentado antes. Las cualidades se estructuran como un array en el que están los siguientes elementos: ● *Gender:* Valor numérico que representa un valor enumerado. Los posibles valores son: ○ 0: Macho. ○ 1: Hembra. ● *Diet:* Valor numérico que representa un valor enumerado. Los posibles valores son:

	○ 0: Herbívoro. ○ 1: Omnívoro. ○ 2: Carnívoro. ● *Scavenger:* Valor numérico de coma flotante. ● *MaxHealth:* Valor numérico de coma flotante. ● *CurrHealth:* Valor numérico de coma flotante. ● *Damage:* Valor numérico entero. ● *Armor:* Valor numérico entero. ● *Perforation:* Valor numérico entero. ● *Venom:* Valor numérico de coma flotante. ● *Counter:* Valor numérico de coma flotante. ● *AerialSpeed:* Valor numérico entero. ● *ArborealSpeed:* Valor numérico entero. ● *GroundSpeed:* Valor numérico entero. ● *AirReach:* Valor booleano. ● *TreeReach:* Valor booleano. ● *MaxEnergy:* Valor numérico de coma flotante. ● *CurrEnergy:* Valor numérico de coma flotante. ● *EnergyExpense:* Valor numérico de coma flotante. ● *MaxHydration:* Valor numérico de coma flotante. ● *CurrHydration:* Valor numérico de coma flotante. ● *HydrationExpense:* Valor numérico de coma flotante. ● *MaxRest:* Valor numérico de coma flotante. ● *CurrRest:* Valor numérico de coma flotante. ● *RestRecovery:*Valor numérico de coma flotante. ● *RestExpense:*Valor numérico de coma flotante. ● *Camouflage:* Valor numérico entero.

	• *Aggressiveness:* Valor numérico entero. • *Intimidation:* Valor numérico entero. • *Perception:* Valor numérico entero. • *MaxPerception:* Valor numérico entero. • *Size:* Valor numérico entero. • *LifeSpan:* Valor numérico entero. • *CurrAge:* Valor numérico entero. • *Limbs:* Valor numérico entero. • *Metabolism:* Valor numérico entero. • *MinTemperature:* Valor numérico de coma flotante. • *MaxTemperature:* Valor numérico de coma flotante. • *IdealTemperature:* Valor numérico de coma flotante. • *Hair:* Valor booleano. • *Knowledge:* Valor numérico entero. • *Paternity:* Valor numérico entero., • *HealthRegeneration:* Valor numérico de coma flotante. • *MaxSpeed:* Valor numérico de coma flotante. • *TimeBetweenHeats:* Valor numérico entero. • *InHeat:* Valor booleano. • *Upright:* Valor booleano.

Tabla 4.5. Atributos del objeto dentro de *Species_X.json*

Existen ejemplos sobre los archivos de especies en el apartado A del apéndice.

- Los árboles filogenéticos de las especies, que describen las relaciones entre las especies con el paso del tiempo. Hay un único archivo que contiene toda la información y dentro de este hay tantos árboles como especies al inicio de la simulación, de las que salen las especies derivadas. Se exportan como un archivo llamado *tree.txt* en el que cada especie está ordenada jerárquicamente. También

muestra el tiempo de vida de cada especie, con el primer ciclo de simulación en que aparece y el último antes de desaparecer. El apartado A del apéndice contiene un ejemplo de este archivo.

- Imágenes del mapa. Éstas imágenes se generan para que el usuario tenga una idea general de cómo es el mapa en el que se realiza la simulación. Se generan como archivos .png son representados en las Figuras 4.17 y 4.18.

  Cabe destacar que el programa en modo de depuración genera más mapas que han sido utilizados para revisar la generación del terreno, estos mapas se pueden ver en el apartado A del apéndice.

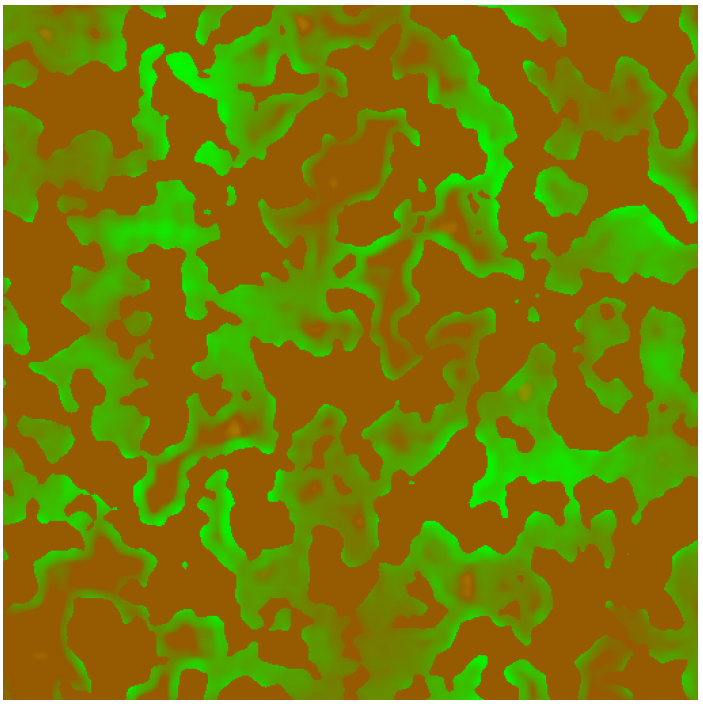


Figura 4.17. Mapa de relieve y flora (*TerrainTexture.png*)

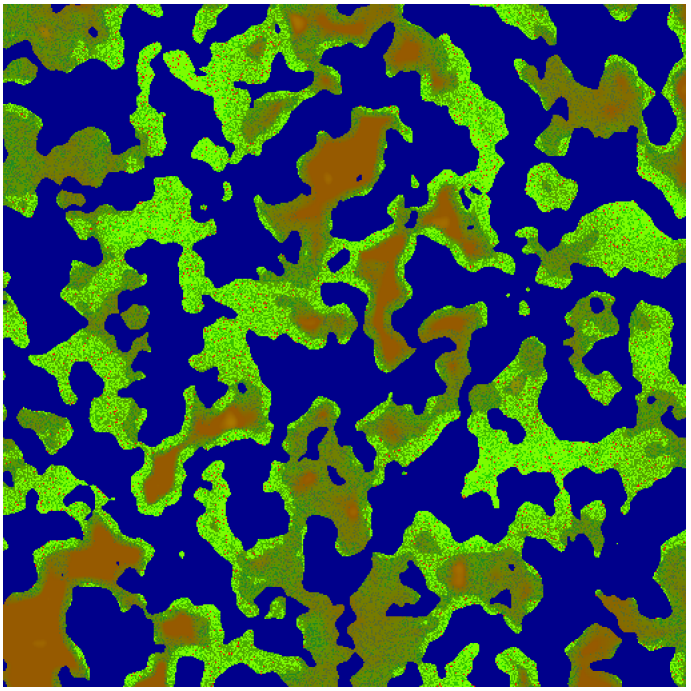


Figura 4.18. Representación del mapa con flora, agua y relieve (*Map.png*)

4.2. Prototipo de demostración del sistema evolutivo

Como ejemplo de uso, se propuso integrar el sistema en Unity. Se escogió Unity porque es una de las plataformas de desarrollo más populares y de uso más común en la actualidad. Además, presenta como principales ventajas el uso de C# como lenguaje de scripting, lo que facilita la integración del sistema, y la familiaridad de los integrantes del proyecto con el motor. El desarrollo del prototipo en Unity se realizó en paralelo con el de la propia simulación. A parte de permitir la visualización de la simulación en tiempo real, se quería proporcionar un ejemplo de la interpretación de sus datos, representando las criaturas gráficamente como se muestra a continuación.

4.2.1. Creación de criaturas con los datos de salida

Como el sistema evolutivo está diseñado para funcionar mediante datos, no es necesario acoplar el sistema a ningún proyecto de Unity u otro tipo de proyecto para hacer uso de él. Una vez se configure una simulación y se complete la ejecución de ésta, se exportarán los datos obtenidos y la interpretación de estos recae en el usuario.

El objetivo del primer prototipo en Unity incide en este aspecto, la interpretación de los datos obtenidos por el sistema para conseguir una representación visual ejemplar en 3D de las criaturas. Se propuso hacerlo en tres dimensiones porque la simulación se realiza técnicamente en el mismo espacio, aunque los datos obtenidos no están sujetos a su interpretación para un espacio tridimensional. En primer lugar, se hicieron simulaciones

para obtener los datos de las especies, como se explica en el apartado 4.1.3. Luego, se creó un script en Unity para la interpretación de los datos de salida, con el que se creó un primer prototipo con los recursos que provee Unity por defecto; dando lugar al resultado, que se puede ver en la Figura 4.19.

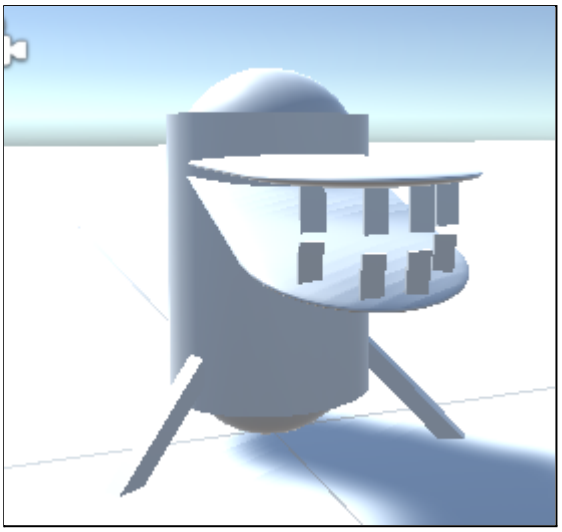
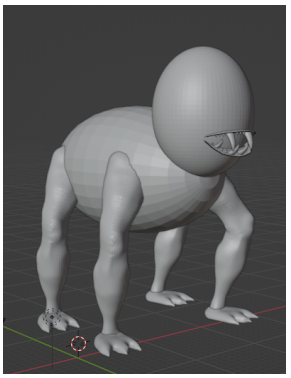
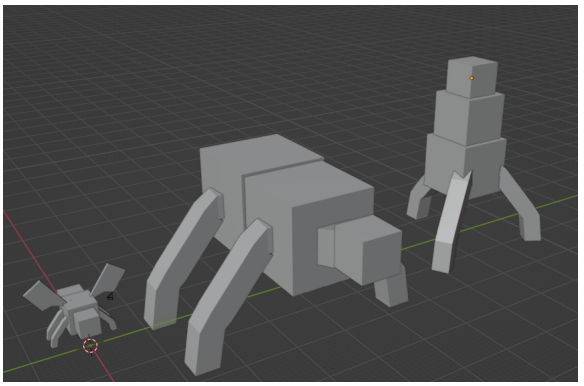


Figura 4.19. Prototipo inicial de criatura (Interpretando datos de archivo)

Aunque se puede suponer la representación visual de una criatura en el prototipo inicial, la representación no se consideró apropiada a la información dada en los archivos, en gran parte por la limitación que suponía usar sólo los recursos de Unity. Además, no se representaba ninguna característica propia de las criaturas, así que todas serían fundamentalmente iguales.

A continuación, se propusieron varios modelos 3D propios para la representación, con el objetivo de tener un margen mayor para una representación adecuada de la información dada en los archivos. Los dos prototipos que se plantearon se pueden observar en las Figuras 4.20 y 4.21.





Figuras 4.20 y 4.21. Prototipos de representación de criaturas

Se descartó el estilo de la Figura 4.20 debido a que su alto número de polígonos dificultaría la simulación simultánea de gran cantidad de criaturas, y la creación de nuevas partes implicaría un mayor gasto de tiempo. Además, y aunque es un aspecto fundamentalmente subjetivo, el intento de realismo dio pie a una figura muy desagradable de ver, lo cual es otro aspecto negativo. Por otro lado, tras experimentar con el estilo basado en polígonos de la Figura 4.21 se acabó optando por este, la razón tras esta decisión siendo la relativa sencillez a la hora de crear nuevas piezas, su menor coste en ejecución, y, además, el hecho de que la realización de uniones de distintas partes para la creación de criaturas sería más sencilla y más estéticamente consistente. Tras la elección del estilo, se crearon modelos independientes para cada parte significativa de las criaturas, pudiendo haber varios modelos asociados a una misma parte. Por ejemplo, se podrían usar dos modelos diferentes para las alas de las criaturas, pudiendo así elegir qué modelo utilizar según la cualidad de velocidad en aire que esté presente en el archivo.

Finalmente, fueron creados modelos listados en el apartado A del apéndice que representan ciertas características de las criaturas, aunque no todas, ya que no es el objetivo de este proyecto conseguir una interpretación idónea, sino datos útiles que se ejemplifican en un prototipo de demostración. Se han hecho los siguientes modelos a parte del cuerpo básico formado por cubos:

- Alas, que representan si una criatura puede volar o no. Hay dos modelos, y representan distintas inversiones en esta habilidad, usando una para especies competentes en el vuelo y otra para especies sobresalientes en esto.

- Patas, que representan el número de miembros de una criatura. Hay tres modelos para las patas con el objetivo de dar variedad y diferenciar las especies.

- Bocas, que representan visualmente la dieta de una especie. Los carnívoros tienen dientes pequeños y colmillos protuberantes; los omnívoros tienen dientes de un tamaño mediano y colmillos más pequeños que los carnívoros, marcando un punto intermedio, y los herbívoros tienen dientes grandes y cuadrados. Aunque no necesariamente realista es una forma simple de representar estas diferencias.

- Pelo, que representa precisamente si la criatura tiene o no, y a falta de una implementación más sencilla, dado que no hay un gran conocimiento de modelado en el grupo, se representa con un tupé.

- Barba, que representa si la criatura ha vivido una gran cantidad de tiempo en relación a su ciclo de vida. Hay constancia de que puede dar lugar a confusiones con el pelo, pero de nuevo era la implementación más sencilla y realmente los modelos no son parte del estudio como tal.

- Púas, que representan si tienen una criatura tiene o no, se plasma visualmente con pinchos que salen del tronco del modelo.

Se introdujeron en el proyecto de Unity, se adaptó el script de la interpretación de datos para hacer un uso dinámico de estos modelos y se consiguió una representación funcional de las criaturas. A continuación, en la Figura 4.22, un ejemplo de una criatura que ejemplifica los puntos descritos anteriormente.

Posteriormente se añadió una barra de vida que proporciona el nombre de la especie de un individuo. Tras obtener resultados exitosos en la interpretación de las criaturas, se propuso como nueva meta representar toda la simulación (criaturas y terreno) en vivo para así tener una representación mucho más visual de lo que ocurre en la simulación. Con esto, se conseguiría también una exhibición que se asemejaría más a un mundo virtual coherente y también se confirma otro de los posibles usos del sistema, que es acoplarlo a proyectos externos.

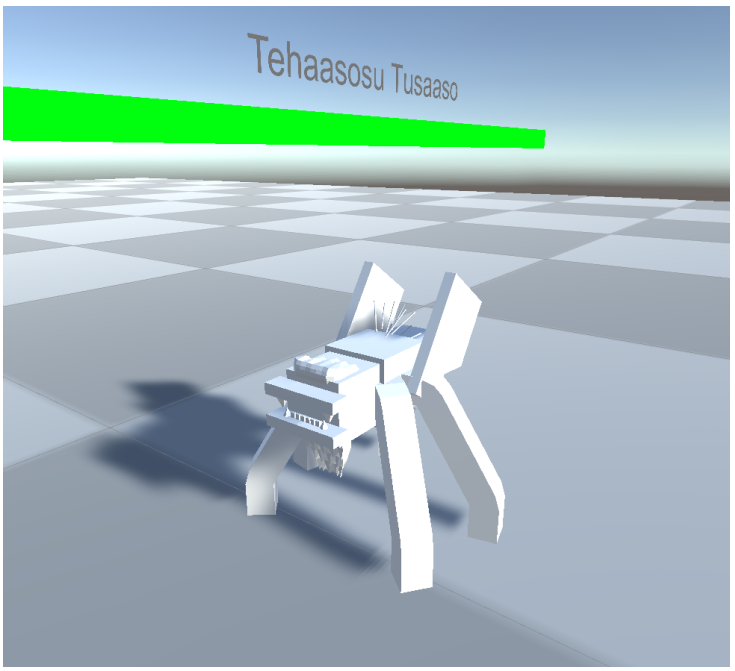


Figura 4.22. Representación gráfica de una criatura (Interpretando datos de archivo)

4.2.2. Representación visual de la simulación

Para esta demostración se utilizó una escena diferente dentro del proyecto de Unity en la que también se representa el mundo físico y no solo las criaturas. Una vez importada la librería con el código de la simulación en el proyecto de Unity fue necesario implementar algo de infraestructura en el proyecto Unity para adaptar el código fuente del sistema evolutivo.

Existe un script principal que gestiona la configuración y gestión de la simulación. Se trata del script *SimulationManager* que sigue el patrón *singleton*. Con esto, el resto de scripts pueden acceder a la información relevante de la simulación a través de la

instancia de este *script*. Se pueden configurar mediante este script, desde el editor de Unity, los archivos de entrada necesarios para la simulación, que son los mismos vistos en el apartado 4.1.1, salvo el mapa de alturas. También se pueden configurar otros parámetros que se explican más adelante.

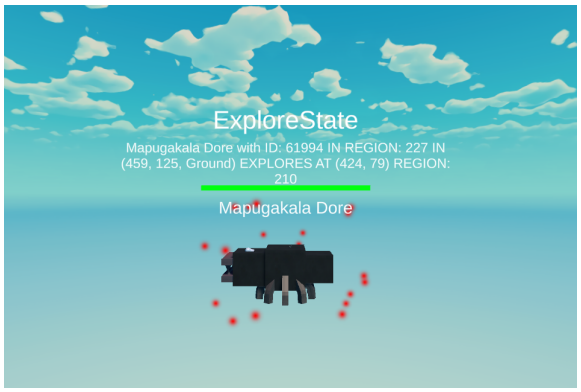
Este script no es el que contiene el sistema evolutivo per sé, el script *UnitySimulation* es una adaptación de la simulación evolutiva para el proyecto de Unity. Ésta simulación es creada y configurada según lo que esté en el editor por el *SimulationManager*. Provee de la misma funcionalidad que la simulación del sistema, y se inicializa proporcionando los archivos de entrada independientes, en vez de el directorio donde encontrarlos, que es otra opción que permite la simulación. La simulación se puede inicializar tanto proporcionando los archivos como los directorios donde se encuentran los archivos, se hace uso de la segunda opción ya que desde el editor de Unity es más cómodo acoplar los archivos independientes. Este script también tiene funcionalidad adicional, que consiste en métodos para obtener datos sobre la simulación, y una función alternativa para avanzar la ejecución de la simulación, para poder ser utilizada en la corrutina en la que se ejecuta la simulación. Ésta corrutina es lanzada por el *SimulationManager* y es necesaria para el movimiento de las criaturas, que se realiza mediante interpolaciones lineales.

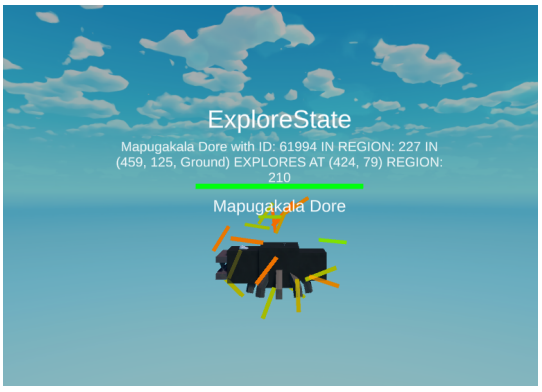
Existen, además, tres scripts para gestionar los principales aspectos de la simulación:

- *WorldCreaturesManager*, que se encarga de crear y actualizar los objetos que representan visualmente las criaturas de la simulación. Para ello, destruye o crea las criaturas según el estado de la simulación. Tiene que estar asociado al *SimulationManager* desde el editor para poder actualizarse tras cada ciclo de simulación. Además, desde el editor, se puede configurar el objeto que se utiliza para la representación (*Prefab* de Unity), junto con la altura a la que se simula el vuelo y el movimiento a través de los árboles. La representación de las criaturas se extendió respecto a la versión final vista en la Figura 4.22, se interpretan las mismas características, pero ahora se obtienen los datos de la propia ejecución de la simulación en lugar de un scripts externo. Además se interpretan datos extra para representar el estado en el que se encuentran las criaturas y la información referente a este estado con texto por encima de la criatura, como se puede ver en la Figura 4.23. Se puede observar también en la figura que las criaturas están texturizadas, éstas texturas se pueden cambiar desde el editor igualmente.

Para simular un movimiento de las criaturas, se utiliza una interpolación lineal entre la última y la nueva posición de la criatura. Esta interpolación depende del tiempo que pase entre cada ciclo de simulación, como se verá más adelante en este apartado. Además, las criaturas también tienen efectos de partículas asociados a recibir un ataque, a alimentarse y a beber agua, como se puede ver en las Figuras 4.24 y 4.25 y la Figura 4.26.

Figura 4.23. Representación gráfica de dos criaturas en la simulación de Unity





Figuras 4.24 y 4.25. Efectos de partículas de las criaturas al recibir daño y al alimentarse

Figura 4.26. Efectos de partículas de las criaturas al beber agua

- *WorldCorpseManager*, cuya función es crear y destruir los objetos que representan gráficamente los cadáveres de la simulación.Tiene que estar asociado al *SimulationManager* desde el editor para poder actualizarse tras cada ciclo de simulación Además, desde el editor se puede configurar el objeto que se utiliza para la representación (*Prefab* de Unity). En la Figura 4.27 se puede observar cómo se ven estos cadáveres.

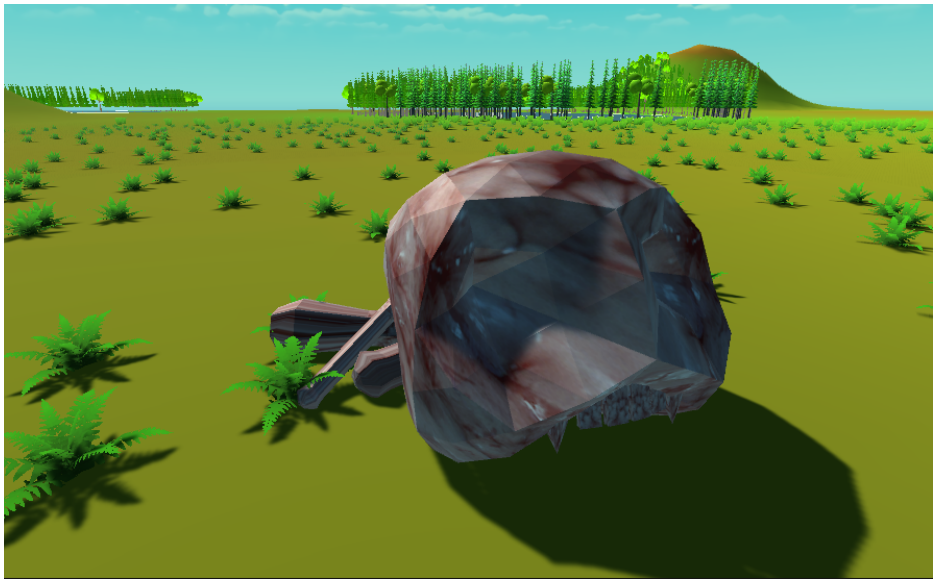


Figura 4.27. Representación gráfica de un cadáver en la simulación de Unity

- *WorldGenerator* se encarga de la representación visual del terreno y la flora presente en el mundo de la simulación. La implementación es casi directa, a causa de que el mapa que se genera es del mismo tipo que se pide en los terrenos de Unity.

Se simula adicionalmente paso del tiempo entre día y noche, mediante el script *DayNightCycle*. Este script utiliza la información proporcionada por la simulación para cambiar dinámicamente el shader del Skybox de Unity según el momento del día en el

que se encuentre la simulación, transicionando entre dos texturas diferentes y modificando la intensidad de la fuente de luz de la escena.

Para poder ver la simulación, es necesario poner desde el editor, en el *SimulationManager*, el número de especies originales y el número de individuos iniciales por especie. Además, se puede hacer que se simulen años previos a la representación, también desde el editor. Ésta opción no es muy recomendable ya que esta simulación previa suele tener una duración muy extensa.

Por último, se puede configurar igualmente la velocidad a la que se quiere ver la simulación, ya que puede resultar muy lenta en la mayoría de momentos. Desde el editor existe un parámetro *TicksBetweenSteps* que permite configurar el tiempo (en segundos) que pasa entre cada ciclo de la simulación. Este tiempo puede no ser real si hay un gran número de criaturas y el ciclo de la simulación tarda más de lo establecido en el *TicksBetweenSteps*. Éste parámetro afecta principalmente a las interpolaciones que hace cada criatura para su movimiento.

Figura 4.28. Vista general de la representación en Unity

Una vez la simulación termina, es decir, todas las criaturas han muerto, aparece un panel con un botón que permite reiniciar la simulación, en el mismo mapa pero con nuevas criaturas.

Figura 4.29. Panel de reinicio de simulación en Unity

Además, para la representación en Unity se utilizaron recursos externos cuya atribución se menciona en el proyecto y a continuación:

- Fantasy Adventure Environment (Staggart Creations)[5]
- Free Stylized Skybox (Yuki2022)[6]
- Tiling Texture Pack 18 (Yughues)[7]
- Tiling Texture Pack 24 (Yughues)[8]
- Woodland Animals Texture Pack (poss)[9]
- Gory Flesh Texture (Adrian Dierigl)[10]

[5] https://assetstore.unity.com/packages/3d/environments/fantasy/fantasy-adventure-environment-70354
[6] https://assetstore.unity.com/packages/2d/textures-materials/sky/free-stylized-skybox-212257
[7] https://opengameart.org/content/tilling-texture-pack-18
[8] https://opengameart.org/content/tilling-texture-pack-24
[9] https://opengameart.org/content/woodland-animals-texture-pack
[10] https://www.artstation.com/artwork/KaPWKo

5. Pruebas

Una vez el programa era funcional, para garantizar tanto la aptitud de los resultados de nuestro proyecto como su correcto funcionamiento, recurrimos a una serie de pruebas con distintos sistemas y el análisis de los datos producidos por los mismos.

El correcto funcionamiento de nuestro proyecto fue comprobado mediante un sistema de eventos y telemetría implementado en el programa, mientras que la aptitud de nuestras criaturas a la hora de poblar un entorno digital fue medida mediante pruebas con usuarios potenciales de nuestro público objetivo.

5.1. Evaluación mediante un sistema de telemetría

Un problema persistente a la hora de producir generaciones de criaturas mediante la simulación es la finalización prematura de ésta por una extinción repentina de todas las criaturas. Al ocurrir esto, las criaturas exportadas son aquellas que más tiempo han sobrevivido, lo que anula el propósito del proyecto, que es la evolución de éstas. Para poder averiguar las razones por las cuales ocurre esto, se decidió instrumentalizar el proyecto con un sistema de telemetría mediante el cual se pretendía averiguar cuáles eran los factores ambientales que contribuyen en mayor medida a las muertes de las criaturas.

Mediante un proyecto de telemetría que se ha implementado con el propósito de depurar la simulación, se ha realizado un seguimiento de la ejecución de la aplicación mediante una serie de eventos, que son almacenados en archivos para su posterior análisis. Estos eventos, que se añaden al sistema de telemetría desde la parte relevante del código, tienen en común los atributos de la Tabla 5.1.

Atributo	Descripción
Timestamp	Momento en el que se produce el evento. Se guarda en milisegundos. Sirve para guardar un orden cronológico que facilita el entendimiento de los datos que se recogen.
Tipo de evento	Cada evento tiene un tipo que permite identificarlos y agruparlos para su posterior procesamiento y análisis.
Identificador de sesión	Se utiliza la clase Guid de Microsoft que genera una combinación única en la práctica, que permite diferenciar e

	identificar las distintas sesiones, siendo una sesión una ejecución del programa.

Tabla 5.1. Atributos del *TrackerEvent* básico

A partir de esta estructura, se crean clases para cada evento específico que se necesita. Los eventos del sistema *SessionStart* y *SessionEnd* no amplían esta clase base.

Los eventos pertinentes a las criaturas heredan de la clase *CreatureEvent*, que añade atributos adicionales que aparecen en la Tabla 5.2.

Atributo	Descripción
Tick	Ciclo de la simulación el el cual se ha producido el evento.
CreatureID	Número que identifica a la criatura específica en la simulación.
SpeciesName	Nombre de la especie a la que pertenece la criatura.
X	Posición en el eje X del mapa de la criatura.
Y	Posición en el eje Y del mapa de la criatura.

Tabla 5.2. Atributos de *CreatureEvent*

Los eventos relevantes a las plantas heredan de la clase *PlantEvent*, que añade los siguientes atributos propios de la Tabla 5.3.

Atributo	Descripción
Tick	Ciclo de la simulación el el cual se ha producido el evento.
PlantID	Número que identifica a la planta específica en la simulación.
X	Posición en el eje X del mapa de la planta.
Y	Posición en el eje Y del mapa de la planta.

Tabla 5.3. Atributos de *PlantEvent*

A continuación se listan los distintos eventos según su tipo, con la información adicional que añaden a la de su clase base, descritos en la Tabla 5.4.

Evento	Descripción
CreatureAdult	Evento que hereda de *CreatureEvent*, y se lanza cuando una criatura deja de considerarse una cría. No añade parámetros adicionales.
CreatureApplyPoison	Evento que hereda de *CreatureEvent*, y se lanza cuando una criatura aplica veneno a otra. Añade los parámetros *VictimID* y *VictimSpecies* (el identificador y la especie de la criatura víctima del veneno), damage (el daño del veneno por ciclo) y duration (duración en ciclos del veneno).
CreatureAttack	Evento que hereda de *CreatureEvent*, y se lanza cuando una criatura ataca a otra. Añade los parámetros *VictimID* y *VictimSpecies* (el identificador y la especie de la criatura víctima del ataque), damage (el daño base del ataque) y penetration (cantidad de daño que penetrará la armadura de la criatura enemiga).
CreatureBirth	Evento que hereda de *CreatureEvent*, y se lanza cuando una criatura nace. Añade todas las características de la criatura.
CreatureDeath	Evento que hereda de *CreatureEvent*, y se lanza cuando una criatura muere. Añade características relevantes a su muerte: *DeathType* (enum que contiene todas las posibles fuentes de daño letal), *KillerID* (identificador de la criatura que ha causado la muerte, -1 si no es por daño de otra criatura) y *KillerDmg* (cantidad de daño que ha causado la muerte).
CreatureMating	Evento que hereda de *CreatureEvent*, y se lanza cuando una criatura tiene descendencia por reproducirse con otra. Añade las características *MateID* y *MateSpecies* (identificador y especie de la criatura con la que se ha apareado), y *ChildNumber* (cantidad de hijos producida).
CreatureReceiveDamage	Evento que hereda de *CreatureEvent*, y se lanza cuando una criatura recibe daño de cualquier fuente. Añade las características *AttackerID* (identificador de la criatura que ha causado el daño, -1 si no es por daño de otra criatura), *damage* (cantidad de daño causada) y *RemainingHP* (puntos de vida restantes en la criatura).
CreatureStateEntry	Evento que hereda de *CreatureEvent*, y se lanza cuando una criatura entra en un nuevo estado de su máquina de estados de comportamiento. Añade la característica *State*, que contiene el nombre del estado al que entra.

CreatureStateEntryExplore	Evento que hereda de *CreatureStateEntry*, y se lanza cuando una criatura entra específicamente en el estado *Explore*, que usa para buscar recursos que no tiene. Añade la característica *ResourceNeeded*, que contiene el nombre del recurso que necesita encontrar.
CreatureStateEntryNotSafe	Evento que hereda de *CreatureStateEntry*, y se lanza cuando una criatura entra en los estados considerados peligrosos (de ataque, persecución o huida). Añade las características *RivalID* y *RivalSpeciesName*, que son el identificador y el nombre de la especie de su adversario (es decir, la criatura que le ha hecho pasar a este estado).
PlantEaten	Evento que hereda de *PlantEvent*, y se lanza cuando una planta es comida por completo. No añade parámetros adicionales.
SessionSample	Evento que hereda de *TrackerEvent*, y se lanza periódicamente durante la simulación. Contiene los números *Tick* y *NumCreatures*, que son el ciclo en el cual se ha lanzado el evento y la cantidad de criaturas vivas en la simulación, respectivamente. Además contiene el número *EatenPlantsRatio*, que indica el porcentaje de plantas comestibles que se encuentran completamente consumidas en ese momento.
SimulationStart	Evento que hereda de *TrackerEvent*, y se lanza al comenzar la simulación. Contiene los números *YearsTick*, *TotalTicks* y TotalEdiblePlants, que son la cantidad de ciclos que dura un año, la cantidad de ciclos totales que durará la simulación y el número total de plantas comestibles iniciales, respectivamente.
SimulationEnd	Evento que hereda de *TrackerEvent*, y se lanza al finalizar la simulación. Añade los atributos adicionales *TicksSimulated*, *NumCreaturesAlive* y NumSpeciesAlive, que son números que contienen la cantidad de ciclos que se han simulado, la cantidad de criaturas vivas al final de la simulación y la cantidad de especies presentes en estas criaturas, respectivamente.
SessionStart	Evento que hereda del *TrackerEvent* básico. Representa el inicio de la aplicación de simulación.
SessionEnd	Evento que hereda del *TrackerEvent* básico. Representa el fin de la aplicación de simulación.

Tabla 5.4. Eventos de telemetría

El análisis de los eventos se hace a través de Python, ya que tiene una conveniente selección de librerías con este propósito, como la que se ha elegido usar: Plotly[11]. La particularidad y ventaja de Plotly sobre otras librerías de representación gráfica de datos, aunque no es la única, es que las figuras que genera son interactivas, proyectan datos concretos al pasar el ratón por los gráficos e incluye funciones como hacer zoom. De esta forma, se hizo un archivo .py que usa Plotly que lee, interpreta y representa los datos que se describen en forma de eventos para que se puedan interpretar y responder las preguntas que nos habíamos propuesto investigar. Aparte, para facilitar la presentación de estos datos se ha usado también el entorno de *Jupyter Notebook*, que permite la división del código en celdas que se pueden ejecutar de manera independiente. Esto ayuda a obtener solo la información que se desea, y a hacerlo de forma dinámica, que demuestra ser útil cuando se quiere conocer únicamente información sobre dos especies concretas, por ejemplo, y no todas. Con esta implementación se crea una especie de interfaz, donde el "usuario" o la persona que interpreta los datos solo tiene que llamar a métodos del .py a través del *Jupyter Notebook*, sin necesidad de tener que programar en profundidad, por así decirlo.

Cabe mencionar que utilizar el sistema de telemetría consiste en abrir el archivo *Analysis.ipynb* que se encuentra en la carpeta *Análisis* de la carpeta raíz del proyecto, mediante *Jupyter Lab* o algún otro entorno para *JupyterNotebooks*. Una vez abierto, se necesita poner el identificador de la sesión (las cuales se generan tras una ejecución del programa en el la carpeta Output en el directorio de salida de datos del programa) como el valor de *session* en la primera celda de código. Finalmente, ejecutar las celdas, haciendo siempre la primera para que cargue las librerías y procese los datos.

5.2. Evaluación con usuarios

Con el fin de comprobar si las criaturas producidas por este proyecto podrían usarse en otros de cara a usuarios, se han realizado pruebas con una serie de individuos. Estas pruebas consistieron en una demostración controlable de una ejecución en Unity, en la cual se mostraba una serie de criaturas en un mismo entorno. Se les mostraron criaturas que habían sido producidas mediante evolución de la simulación. Tras la demostración, se les entregó una encuesta a los probadores con preguntas sobre sus opiniones sobre la misma, cuyas respuestas fueron almacenadas por Google Forms. Los individuos sobre los que se realizaron las pruebas fueron alumnos del Grado de Desarrollo de Videojuegos de la Universidad Complutense de Madrid. Estos fueron escogidos por su disponibilidad, y por su familiaridad con los videojuegos. En total, se evaluó a 7 individuos ya que no había más disponibles.

Una vez el probador se encontraba en el equipo en el que se realizaba la prueba, un integrante del grupo le hacía un breve resumen del propósito y funcionamiento del proyecto. Esta explicación estaba redactada de antemano y siempre era igual. Tras esto, se ponía en marcha la ejecución de una build de Unity con unas criaturas anteriormente escogidas. Durante la prueba, los integrantes del grupo resolvían las preguntas de los

[11] https://plotly.com/

probadores, e indicaban zonas o comportamientos de interés de la simulación para que los probadores las observaran.

Una vez pasados 10 minutos de prueba, la simulación era detenida, y se entregaba a los probadores un formulario de Google Forms. Las preguntas se evalúan mediante una escala entre 1 y 6, siendo 1 poco y 6 mucho (en el contexto de la pregunta).

6. Resultados

Una de las partes más importantes de este proyecto es comprobar que las criaturas evolucionadas sean del gusto de los usuarios y simulen un comportamiento creíble en el entorno en el que se encuentran. Para esto lo primero que hay que hacer es que unas criaturas generadas inicialmente de manera aleatoria, evolucionen a lo largo de la simulación, por lo que otro paso de suma importancia es que la simulación dure un largo periodo de tiempo y de tiempo a la evolución de las criaturas.

Los resultados obtenidos sobre el desarrollo del proyecto son los siguientes, donde el primer estudio se centra en averiguar por qué acababa la simulación antes de tiempo en el periodo de depuración y cómo se podría solucionar. La muerte prematura de una gran cantidad de criaturas impide la evolución de éstas, así que era de gran interés arreglar los problemas del código. La segunda evaluación se centra en las pruebas realizadas a usuarios, comprobando si las criaturas que han evolucionado funcionan correctamente y es del agrado de los probadores.

6.1. Depuración de errores usando el sistema de telemetría

Con el objetivo de ver por qué se morían las criaturas prematuramente dando lugar a una simulación defectuosa, nos realizamos una serie de preguntas, que respondimos con conclusiones obtenidas mediante los datos generados. Las gráficas proporcionadas muestran tan sólo un porcentaje de las especies para mayor legibilidad, pero las gráficas completas se crean tras una ejecución del Notebook de Jupyter.

Las preguntas y resultados fueron los siguientes:

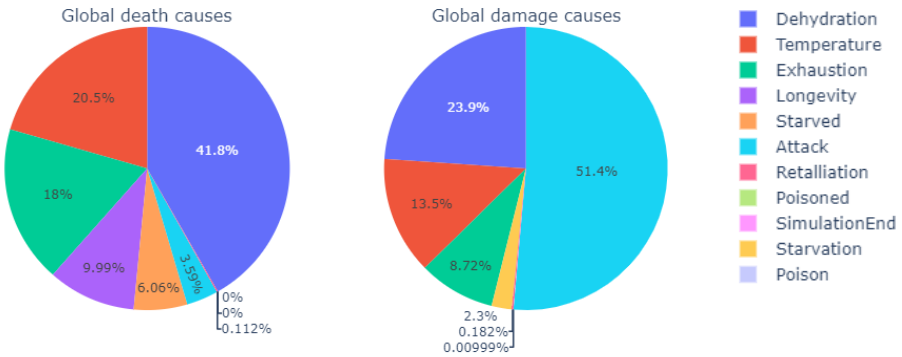


Figura 6.1. Muertes globales de las criaturas según su causa

- ¿Influye la temperatura demasiado en su supervivencia?

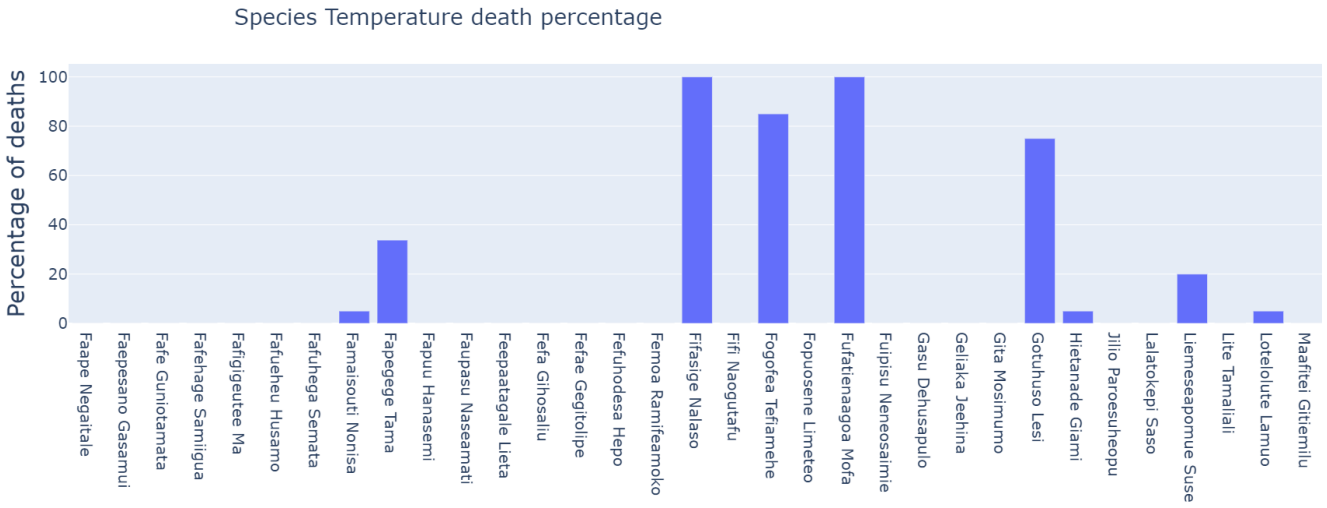


Figura 6.2. Porcentaje de muertes totales por diferencia térmica en parte de las especies analizadas

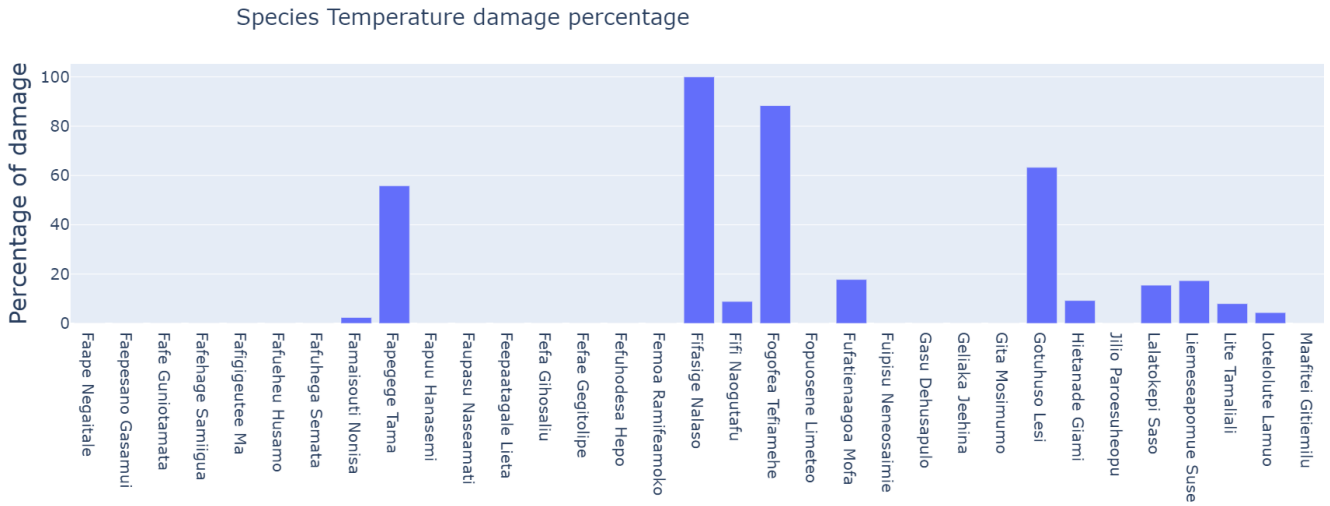


Figura 6.3. Porcentaje de daño total por diferencia térmica en parte las especies analizadas

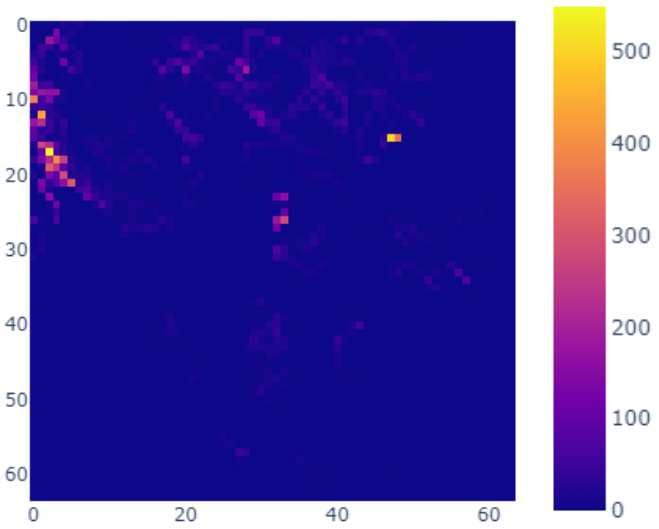


Figura 6.4. Mapa de calor de daño recibido por diferencia térmica

La conclusión que obtuvimos es que sí. De hecho fue lo primero que se detectó mediante la telemetría. Se espera que la temperatura sea un factor menor en la supervivencia de las especies, orientado a dividir el territorio en secciones en base a esta y fragmentar de esta forma la distribución de las especies en el mundo. De esta forma, se dividirían las especies en base a las temperaturas que han evolucionado para soportar y evitarían de forma natural la centralización de la competición por recursos.

Se ha hecho que al generar las criaturas al inicio de la simulación se seleccionen posiciones aleatorias hasta encontrar alguna cuya temperatura es apta para su supervivencia (tiene un límite de intentos por si la criatura tiene un aguante térmico muy pobre en relación al mapa generado). Se ha hecho que haya un número limitado de veces que se intenta colocar en una posición adecuada, pues hay ocasiones en que la criatura tiene un aguante térmico muy pobre en relación al mapa generado y prácticamente en cualquier zona va a recibir daño y acabará muriendo por temperatura.

Después de realizar este cambio, disminuyó la cantidad de daño recibido y el número de muertes causadas por temperatura, pero sigue siendo uno de los más elevados, por lo que necesita cambios en otros aspectos de la simulación.

- ¿Influye la necesidad y disponibilidad del agua demasiado en su supervivencia?

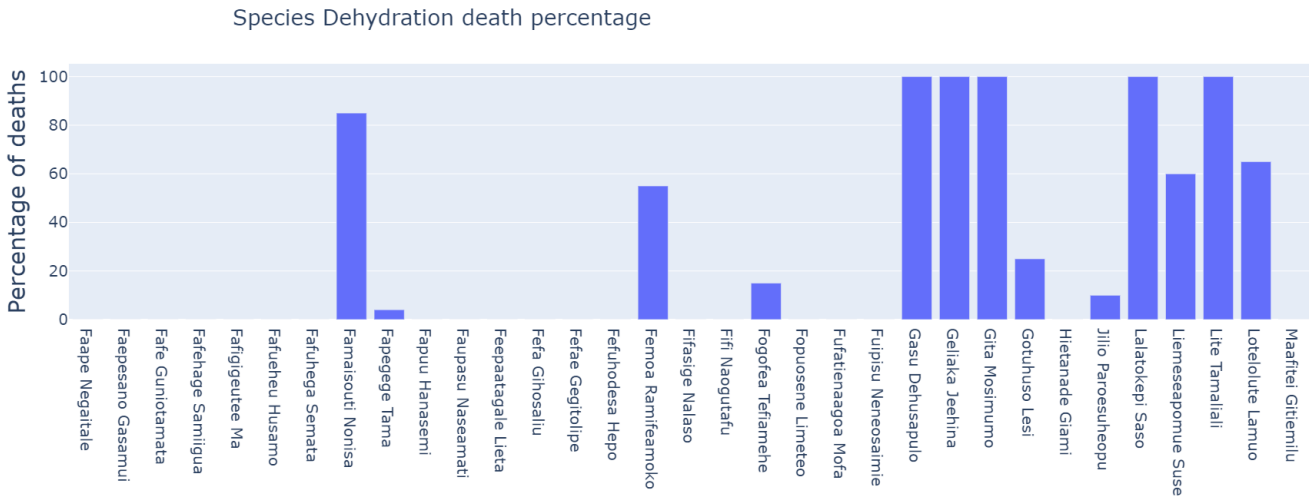


Figura 6.5. Porcentaje de muertes totales por sed en parte de las especies analizadas

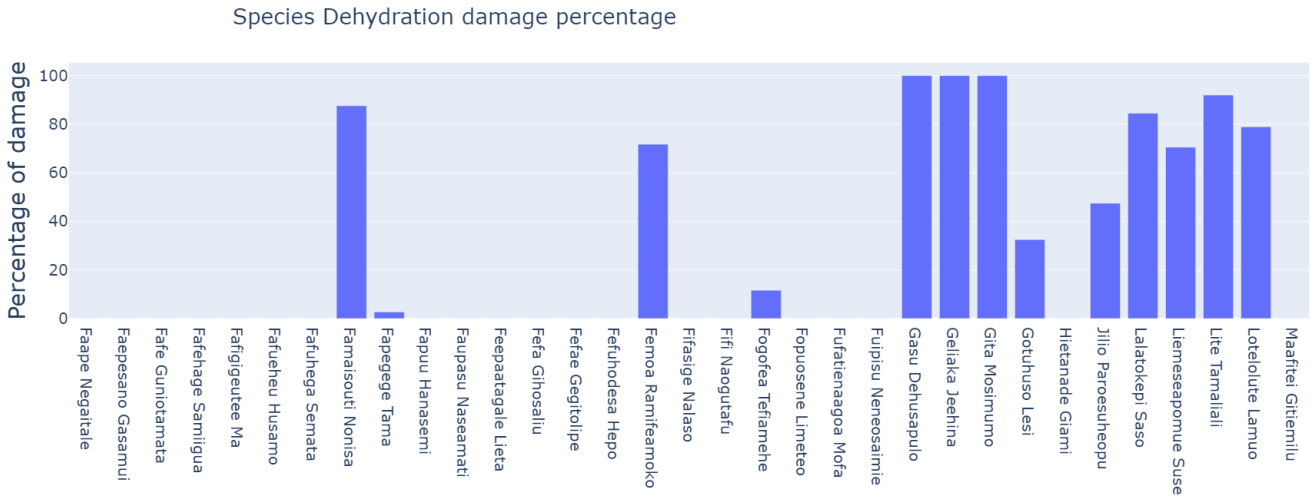


Figura 6.6. Porcentaje de daño total por sed en parte de las especies analizadas

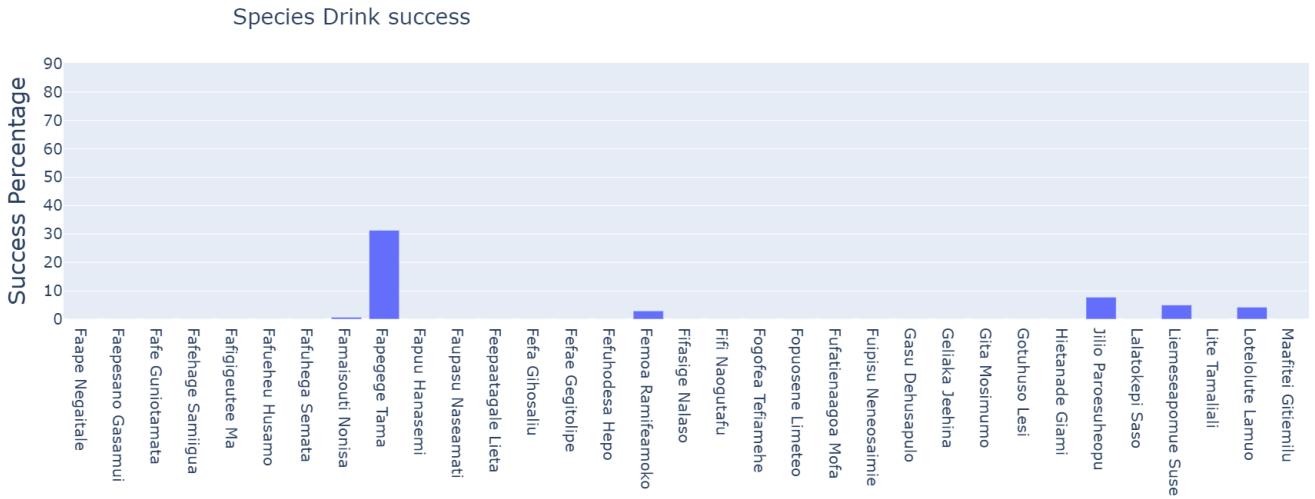


Figura 6.7. Porcentaje de éxito al ir a beber agua en parte de las especies

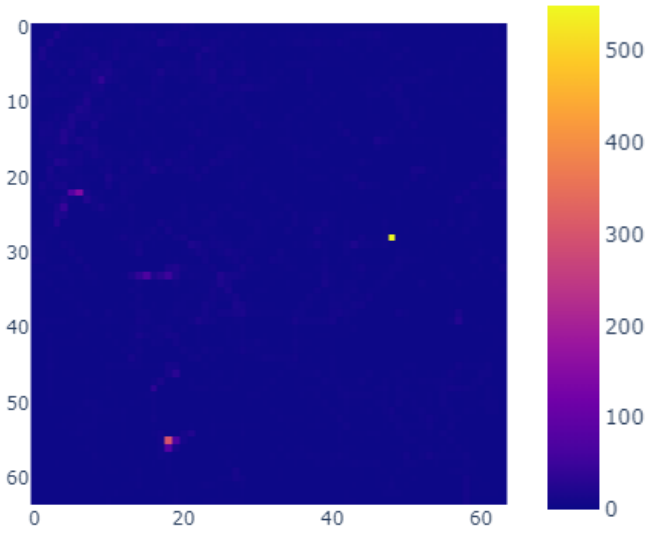


Figura 6.8. Mapa de calor de daño recibido por deshidratación

Tras analizar las gráficas anteriores llegamos a la conclusión de que sí afecta demasiado. Una de las mayores causas de muerte es la sed, cosa que se sospechaba incluso antes de realizar este proyecto. El agua es el recurso más importante que las criaturas necesitan, siendo el que se agota más rápido (3 veces más rápido que la energía otorgada por la comida), y para poder obtenerla debe irse a zonas específicas del mapa que la contienen. El mapa de calor nos muestra que todas las muertes por sed han ocurrido en zonas alejadas del agua, es decir, que las criaturas no han podido encontrarlas.

Los gráficos también muestran que muchas de las criaturas han visto sus intentos de beber interrumpidos. Esto se debe con total seguridad a un fallo en la máquina de estados de las criaturas, ya sea porque alguna transición no se realiza correctamente o porque las prioridades están mal asignadas. Esto ocurre cuando una criatura va a beber y es atacada, por ejemplo, deja de intentar beber para atacar o defenderse. De la misma manera, si tiene sueño pero le entra sed a posteriori, deja de intentar dormir para buscar agua, ya que es más prioritario. Una mala ordenación de las prioridades podría explicar por qué deja de intentar beber tan a menudo.

Se ajustaron las necesidades energéticas, que permitieron a la criatura tener unas necesidades más razonables y ajustadas a nuestra idea de una simulación ideal. Además, se arreglaron fallos en transiciones de los estados relacionados con la sed, que permitieron obtener un porcentaje de éxito a la hora de beber superior en las distintas especies.

- ¿Influye la necesidad y disponibilidad de alimento demasiado en la supervivencia de las criaturas herbívoras?

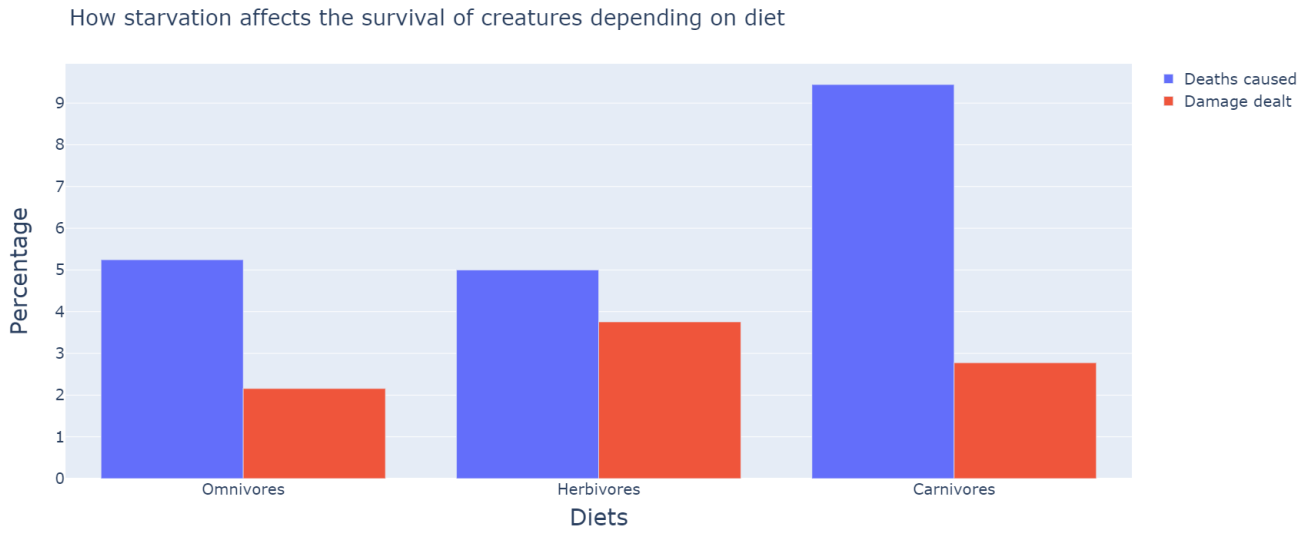


Figura 6.9. Porcentaje de daños y muertes de las especies organizadas por dieta

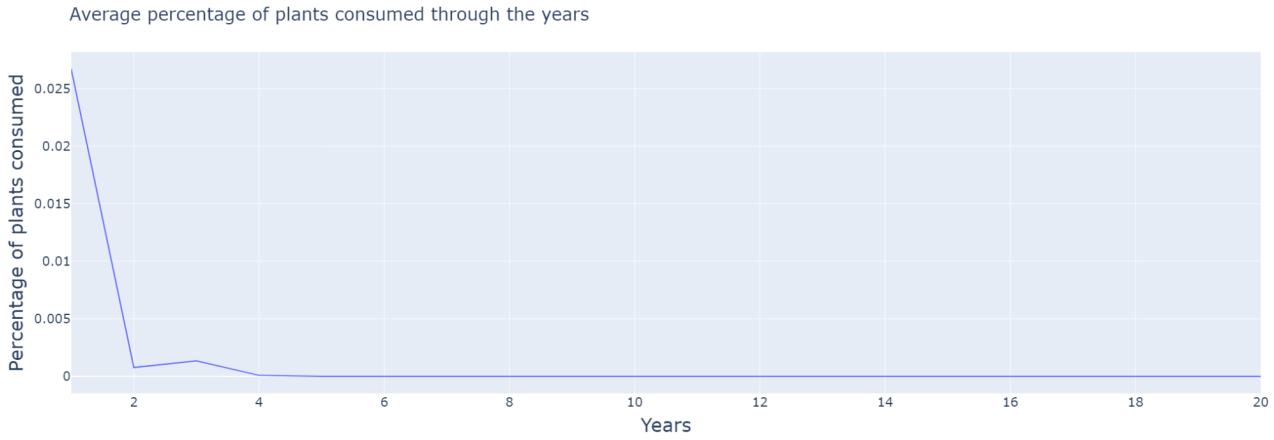


Figura 6.10. Porcentaje de plantas comestibles consumidas por completo frente al total del terreno

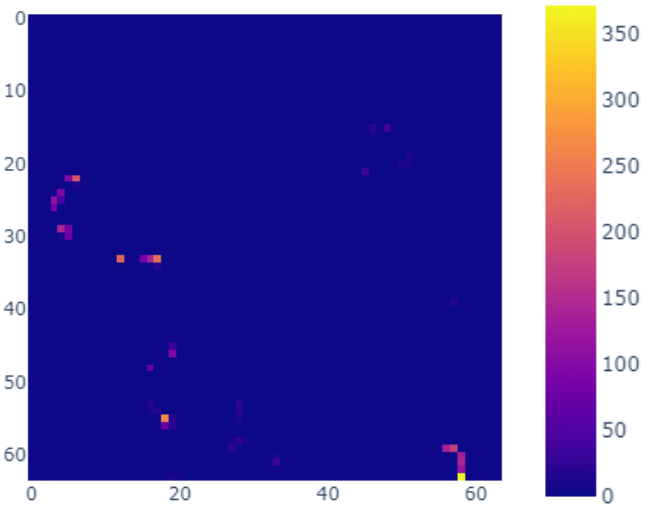


Figura 6.11. Mapa de calor de daño por hambruna

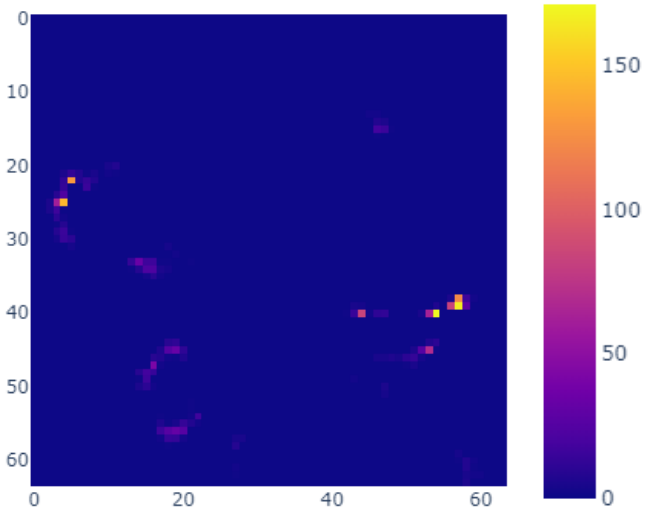


Figura 6.12. Mapa de calor de plantas comidas

A diferencia de las preguntas anteriores, esta necesidad no es relevante. Las criaturas herbívoras pueden obtener comida mucho más fácilmente que las criaturas con otras dietas debido a la disponibilidad de plantas en el entorno y lo rápido que se regeneran. Todo esto es visible en el porcentaje de plantas comidas, y el mapa que muestra dónde son comidas: las criaturas herbívoras comen pocas de las plantas disponibles (siempre por debajo del 0.5% de las plantas totales), y siempre en la misma zona, lo que indica que esta zona les suple con la nutrición suficiente de manera constante.

Se puede observar que puede haber pocas zonas con suficiente comida para sostener la vida, ya que en el mapa de calor de plantas consumidas se come constantemente la/s misma/s, y en el mapa de calor del hambre solo hay puntos cálidos en las posiciones alejadas de este foco de alimento. En el mapa de hambre se incluyen todas las criaturas, pero hay dos posibilidades (dado que los omnívoros se incluyen en las dos dietas restantes): que mueran las criaturas herbívoras, y entonces no hay muchos lugares con alimentos suficientes para esta dieta, o son carnívoras y no encuentran presas, por lo cual el resto de criaturas se concentran en otros puntos, casi con total seguridad el foco en el mapa de plantas consumidas. De esta manera, se puede llegar a la conclusión de que quitando algún punto con una gran concentración de plantas tiende a no haber muchas alternativas en cuanto a lugares de alimentación para herbívoros.

Debido a que quizá influía demasiado poco, se aumentó el tiempo de regeneración de las plantas y se redujo la nutrición otorgada por las mismas para que las criaturas herbívoras se vieran obligadas a explorar y a no volverse sedentarias. Esto influyó positivamente en la simulación, haciendo que las criaturas comieran una cantidad mayor de plantas en zonas menos concentradas. Además, se ajustaron los gastos energéticos para que se aproximaran más a las cantidades deseadas.

- ¿Influye la necesidad y disponibilidad de alimento demasiado en la supervivencia de las criaturas carnívoras?

  Ver Figura 6.9.

  En líneas generales, es evidente que el hambre no es muy influyente. Sin embargo, la extraordinariamente alta mortalidad inicial debido a sed y temperatura en la simulación puede estar comprometiendo estos datos. También, las criaturas carnívoras son las que empiezan a atacar a otras para comer, lo que hace que entren en combate y mueran en él a menudo (aunque esto se verá más adelante). Si la mayoría de criaturas carnívoras mueren por otras causas, las estadísticas recogidas para esta métrica no son representativas, y dado que estos factores de media representan un % de las muertes en las criaturas, se complica la obtención de conclusiones.

  A pesar de esto, en la mayoría de pruebas el hambre influye en gran medida en los carnívoros respecto al resto de dietas. Esto se debe a que muchas criaturas mueren al inicio de la simulación, reduciendo en gran medida la población global y volviendo de esta forma muy escasa la comida para las criaturas carnívoras. Si se mejora la supervivencia en base a las otras métricas, parece razonable decir que las criaturas carnívoras no están tan por detrás que las otras dietas, aunque sí en una desventaja notable, concluyendo así que el hambre sí importa demasiado en su supervivencia. La solución más razonable en este caso sería aumentar la nutrición que proporcionan los cadáveres o el tiempo que duran los cadáveres, o de no ser posible, reducir el consumo calórico de las criaturas carnívoras. De esta manera, se reduce la cantidad de veces que tienen que cazar en su vida,

reduciendo la probabilidad de que mueran de hambre y recompensando una dieta más arriesgada con mayores recompensas, para equilibrarla con las demás.

Tras el ajuste a los gastos energéticos previamente mencionado, las criaturas carnívoras también se vieron beneficiadas positivamente viendo sus muertes y daños por hambre reducidos.

- ¿Influye la necesidad y disponibilidad de alimento demasiado en la supervivencia de las criaturas omnívoras?

  Ver Figura 6.9 y Figura 6.10.

  No, de hecho influye mucho menos de lo esperado. El análisis de los datos de telemetría demuestran un error de lógica a la hora de programar las dietas, y es que es objetivamente mejor ser omnívoro que herbívoro o carnívoro. Esto se debe a que los omnívoros no tienen restricciones en sus dietas, y por lo cual tienen lo bueno de ambos mundos. Esto también es así en la vida real, pero al no especializarse en un tipo de alimento sacrifican de una manera u otra la efectividad de estos. Además las criaturas omnívoras se beneficiaron de las medidas tomadas para las carnívoras y herbívoras, y les afecta de manera positiva también los ajustes a los valores relacionados con las plantas. Como consecuencia, decidimos representar esto de forma interna haciendo que la nutrición que recibe una criatura omnívora de cualquier fuente de alimento se ve penalizada en relación a las otras dietas, actualmente por un 30%.

- ¿Hay suficientes nacimientos para asegurar la prosperidad de la especie?

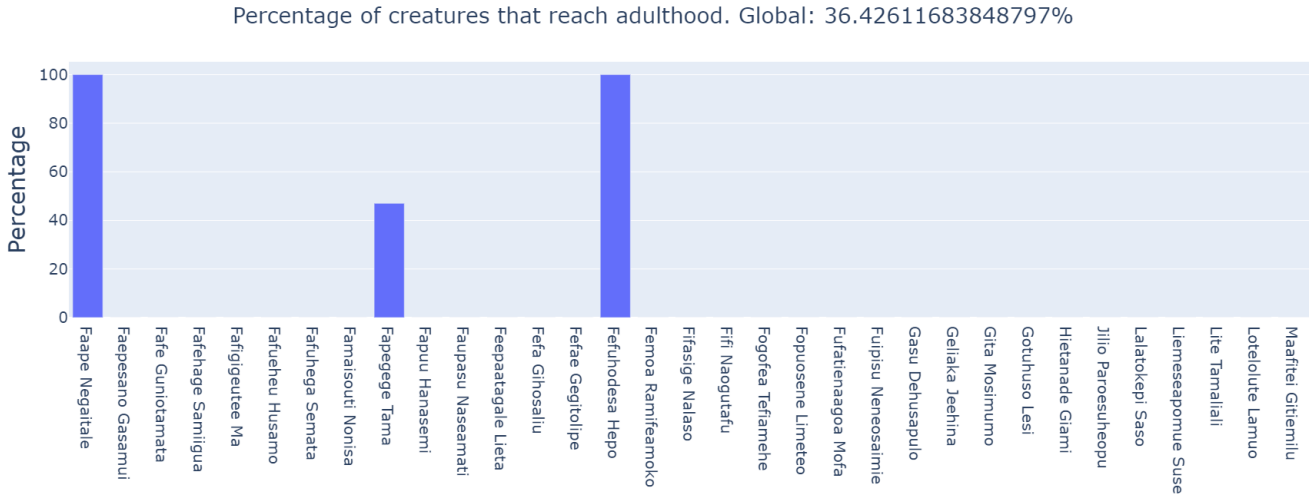


Figura 6.13. Porcentaje de criaturas que llegan a ser adultas en parte de las especies

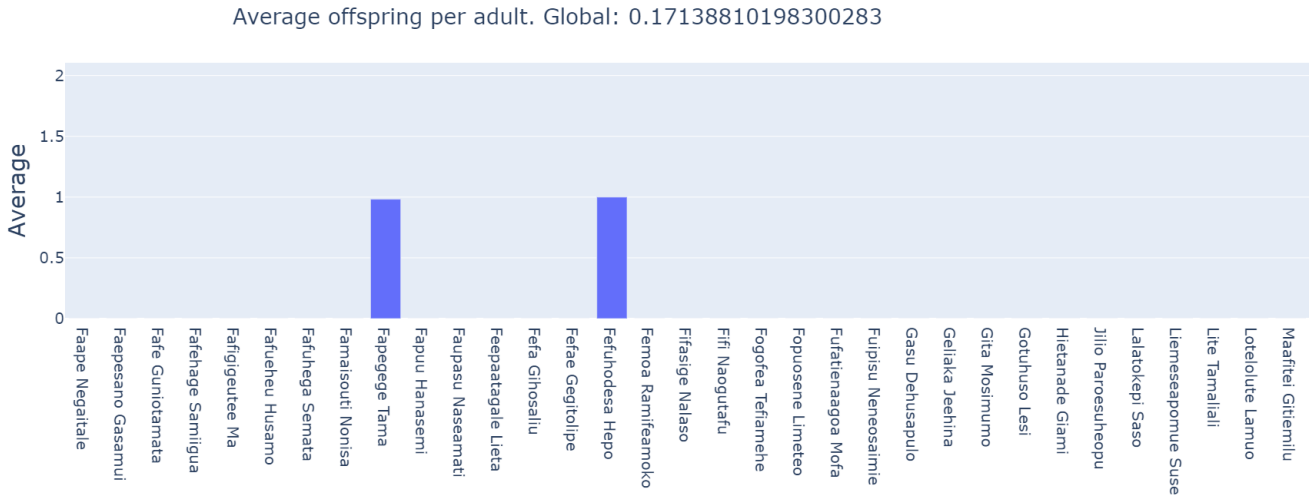


Figura 6.14. Número medio de hijos engendrados por adulto en parte de las especies

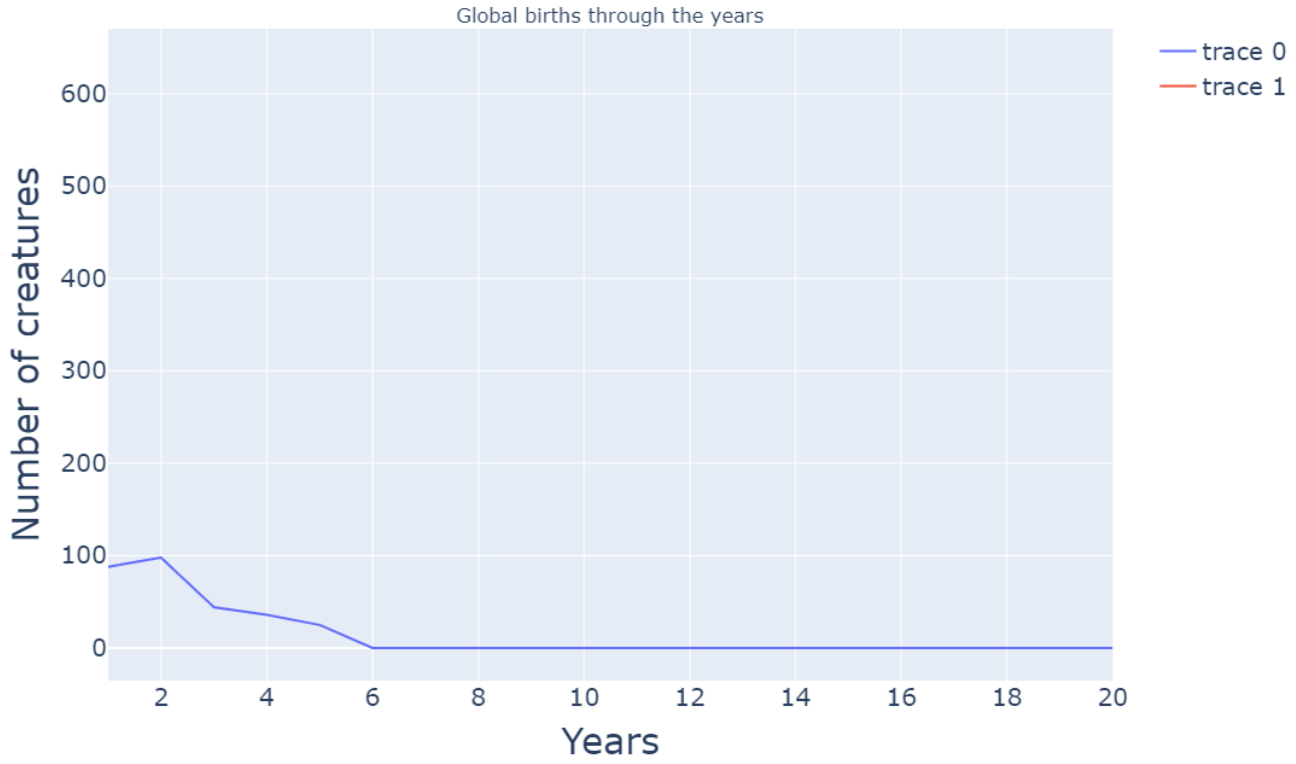


Figura 6.15. Número de nacimientos totales por año de simulación

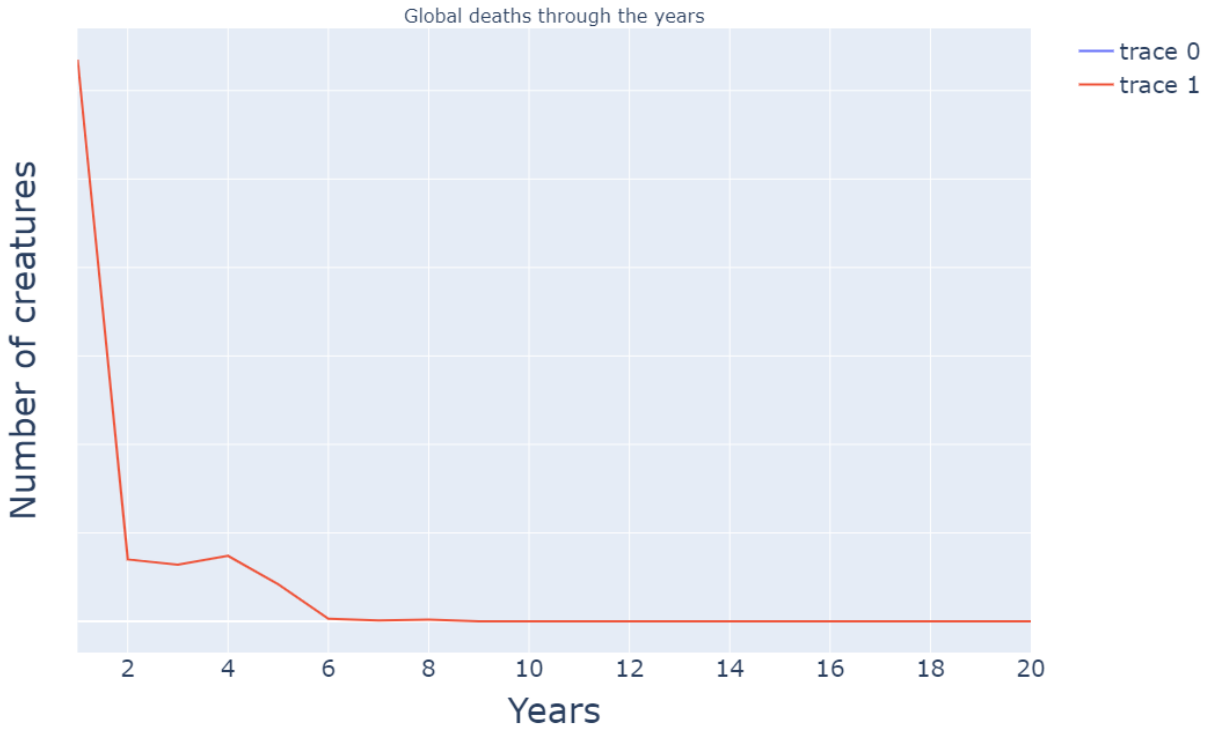


Figura 6.16. Número de muertes totales por año de simulación

Esta pregunta es difícil de concluir porque en las pruebas la mortalidad de las criaturas es súbita, causando la extinción de especies enteras antes de que puedan reproducirse de forma efectiva. Sin embargo, se ha visto que la relación entre los adultos de una especie y el número de descendientes que tienen de media es muy baja aún cuando la especie sobrevive durante suficiente tiempo para reproducirse.

Por otro lado, la probabilidad de que las crías de una especie lleguen a la adultez, suponiendo que sobreviven, es muy baja de media también, pero este es otro factor condicionado en gran medida por la gran tasa de mortalidad en el estado actual de la simulación.

La respuesta evidente comparando los gráficos de muertes y nacimientos es evidentemente no, pero no se puede llegar a la conclusión de que haya que reducir el tiempo que se tarda en llegar a la adultez o aumentar el número de hijos que aparecen cada vez que se da una reproducción, por ejemplo. De hecho, en simulaciones extraordinarias donde una o varias especies sobrevivían al primer año con una cierta cantidad de miembros, la población global crecía de forma desmedida, hasta dificultar la ejecución del programa.

En conclusión, esta pregunta no se ha podido responder de forma satisfactoria porque se ve totalmente condicionada por el estado actual de la simulación, que presenta una serie de características, la increíblemente alta mortalidad principalmente, que no permiten la obtención de datos fiables. Habría que modificar el programa en base a los resultados obtenidos en los apartados anteriores y volver a analizar los datos y métricas resultantes.

Gracias al sistema de telemetría y al análisis, se han descubierto también otros aspectos y errores externos al objetivo principal, pero que influyen en la simulación. Algunos de estos errores han sido arreglados durante el desarrollo, ya que no permitían que la simulación tuviera resultados aceptables para evaluar el objetivo original. Por ejemplo, cuando se comenzó a hacer uso de los eventos y la telemetría, se hizo evidente que había transiciones de la máquina de estados de las criaturas que no estaban como se esperaba. Si no fuese por el sistema de telemetría, probablemente no habría sido muy difícil o imposible detectar estos errores, pues la manera de ver lo que ha hecho una criatura durante su vida es muy útil. Por ejemplo, muchas criaturas morían de hambre o sed mientras dormían, o que cambiaban constantemente de un estado A a otro B y en el siguiente ciclo del B al A hasta que se morían sin hacer nada. Había otro caso en el cual una especie puede ser erradicada por ataques de otras que no se ajusta a ésto: cuando una criatura nace con suficiente diferencia genética de su padre se la considera una especie nueva. La violencia entre padres e hijos está deshabilitada por código, al igual que en miembros de la misma especie. Sin embargo, si una criatura tiene un hijo de su misma especie, y otro que muta lo suficiente para no serlo, al no ser de la misma especie ni tener relación de paternidad, nada impide a dos criaturas hermanas atacarse. Una vez esto ocurre, la criatura de la misma especie es ayudada por su padre, que ataca a su hijo mutado por solicitud de su otro hijo, saltándose así las restricciones de violencia paternal y acabando con una nueva especie nada más nacer. Por lo tanto, se debe evaluar mejor las relaciones entre especies, considerando distancia en el árbol genealógico como medida. Esto fue detectado comparando los id de las criaturas a las que atacaban, y la relación de parentesco entre criaturas fue ajustada para impedir que padres, hijos y hermanos fueran capaces de combatir entre sí.

Otro de los aspectos que resultó ser de especial interés pero no están relacionados directamente con el objetivo son las escaramuzas iniciales. Si se observan los archivos de las especies que han muerto en su gran mayoría por ataques de otras criaturas, se ve que todas las muertes han tenido lugar en los primeros ciclos (muy pronto en la simulación). Esto se debe a cómo se genera el mundo: al poner una gran cantidad de especies en un mismo lugar, muchas de ellas se encuentran ocupando el mismo espacio. Si una criatura se encuentra amenazada por otra de una especie diferente, la ataca, solicitando ayuda de los miembros de su especie cercanos. De la misma manera, los que han sido atacados se defienden, solicitando ayuda también de sus aliados. Esta concentración crea finalmente una gran batalla, en la que especies enteras se atacan y defienden hasta que no queda ningún miembro superviviente, tan sólo porque se encontraban a una distancia que consideraban peligrosa. Por ello, se debe idear alguna manera de repartir las criaturas de tal manera que aquellas más agresivas estén aisladas inicialmente, para permitir un desarrollo inicial más estable.

6.2. Resultados mediante pruebas con usuarios

Dado el contexto de pruebas con usuarios que se estableció en el apartado 5.2, realizaron dichas pruebas mostrando una ejecución en Unity, se les realizó una serie de preguntas que nos ayudaron a sacar conclusiones y saber si las criaturas que había durante la prueba, que eran criaturas ya evolucionadas, parecían realmente evolucionadas y podríamos concluir que el proyecto cumple uno de sus objetivos principales. La primera sección de la encuesta sirve para formar un perfil del individuo encuestado mediante su conocimiento de los temas y entornos relacionados con la misma.

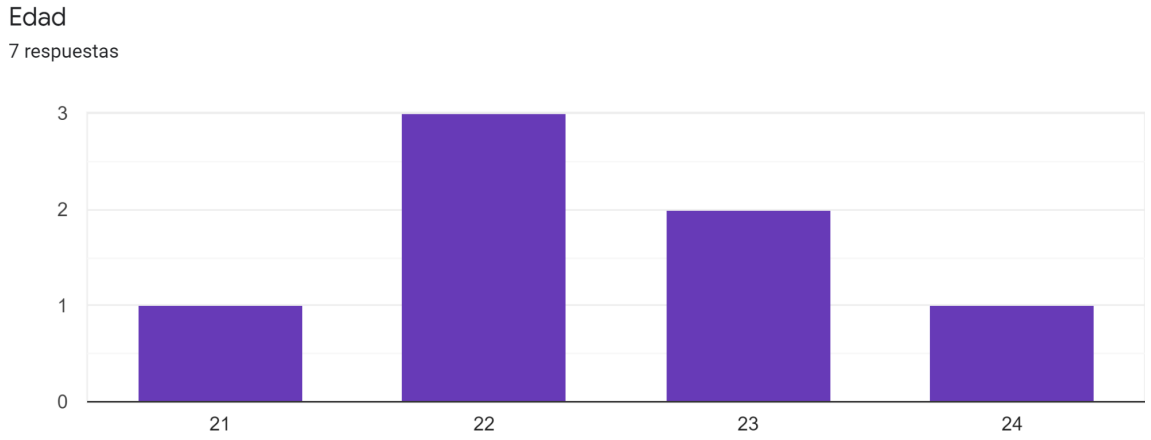


Figura 6.17. Distribución de las respuestas

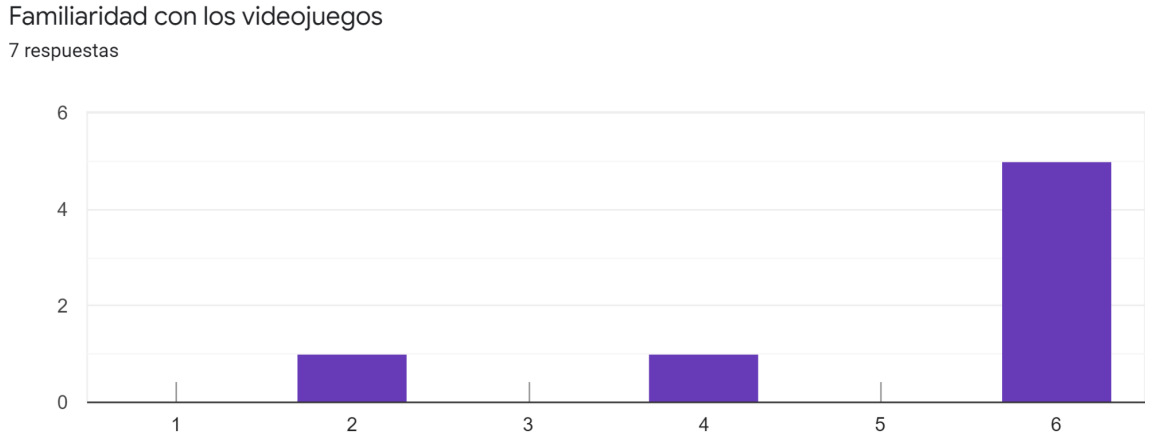


Figura 6.18. Distribución de las respuestas

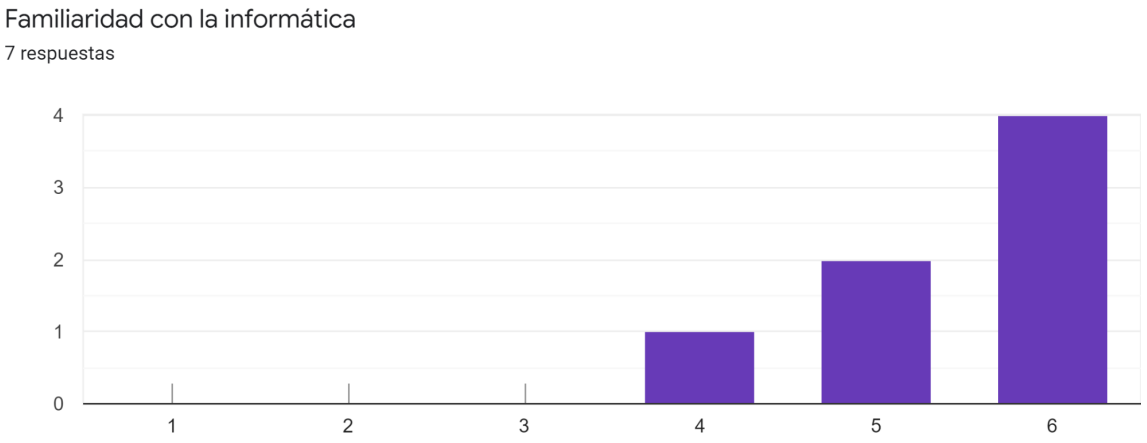


Figura 6.19. Distribución de las respuestas

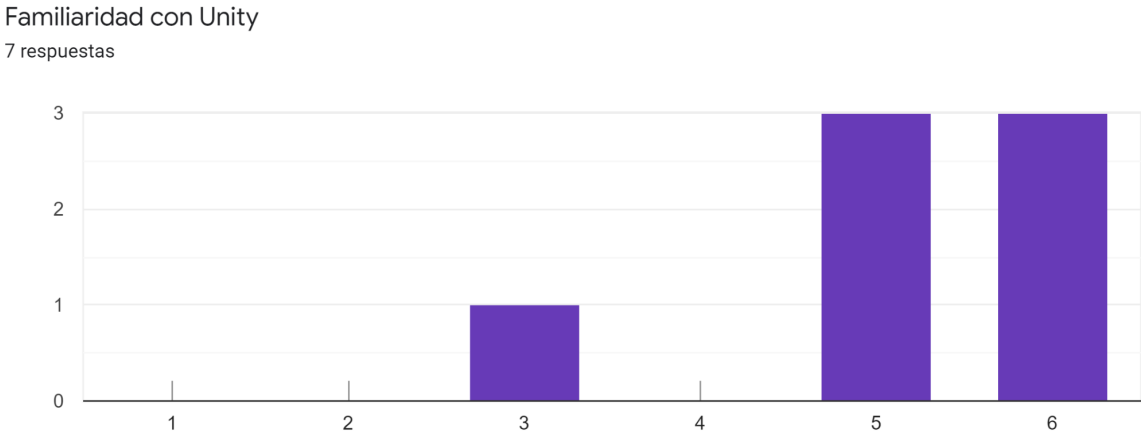


Figura 6.20. Distribución de las respuestas

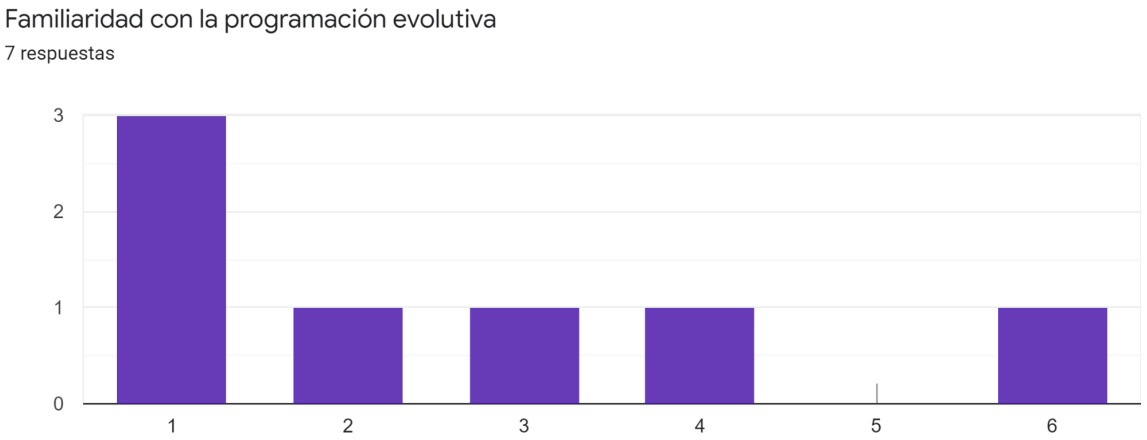


Figura 6.21. Distribución de las respuestas

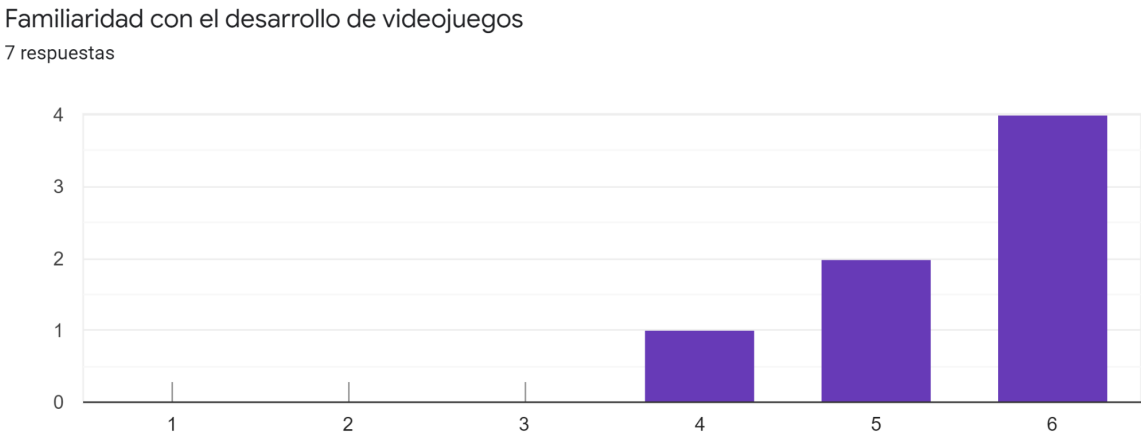


Figura 6.22. Distribución de las respuestas

Estas respuestas nos permitieron contrastar las respuestas de cada individuo con su conocimiento, dando más fiabilidad a aquellos familiarizados con el entorno para el cual nuestro proyecto está destinado.

La segunda sección de la encuesta consta de preguntas relacionadas con los datos producidos por nuestra simulación y con la manera con la que se han plasmado estos en Unity. En esta sección, la escala lineal de las preguntas representa en su menor valor el estar completamente en desacuerdo, y en su máximo valor estar completamente de acuerdo.

Figura 6.23. Distribución de las respuestas

La pregunta representada por la Figura 6.23 evalúa si el terreno que se ha generado en la aplicación, su texturizado y la representación de la flora y agua en él son creíbles. Como podemos observar, todos los encuestados están de acuerdo en mayor o menor medida con esta noción, por lo que podemos concluir que el terreno se genera y muestra satisfactoriamente según nuestras necesidades.

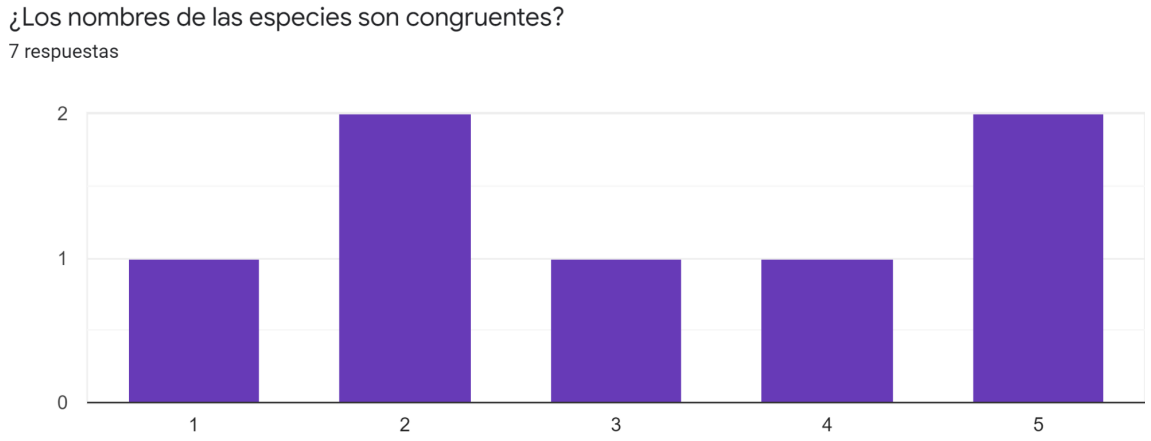


Figura 6.24. Distribución de las respuestas de los encuestados

La pregunta representada por la Figura 6.24 evalúa si los nombres de las especies son diferenciables, y hasta cierto punto si tienen similitud con los nombres científicos de animales reales (como el gato doméstico, o *Felis silvestris catus*). Como se puede observar, las opiniones son más variadas en esta pregunta. Podemos considerar que

parte de la razón es por un fraseo extraño, o por el hecho de que nuestro sistema de generación de nombres no intenta asemejarse al latín. Por ende, aunque podemos deducir que no se asemeja al sistema científico, no podemos decir que está mal con seguridad.

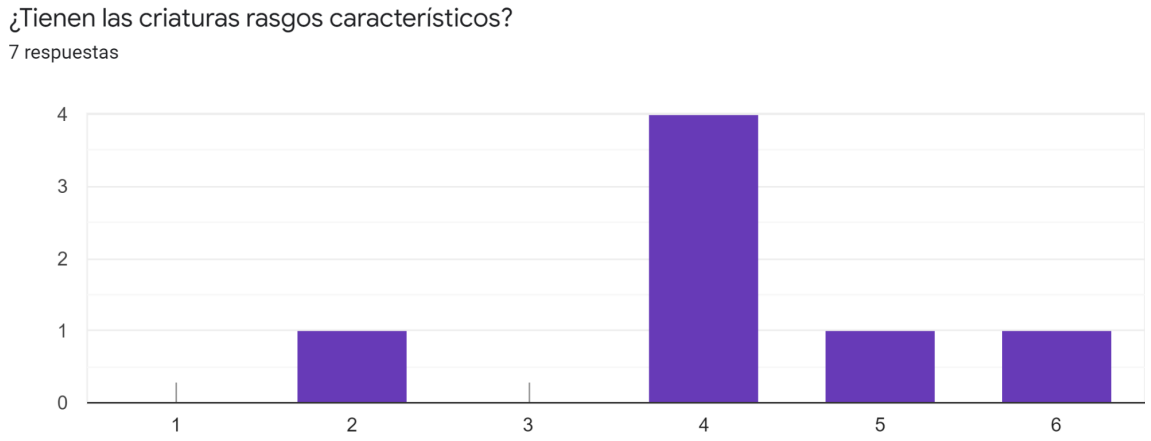


Figura 6.25. Distribución de las respuestas de los encuestados

La pregunta representada por la Figura 6.25 evalúa si las criaturas se diferencian entre sí en la representación, dando la impresión de un ecosistema variado. La opinión general es positiva; a pesar de esto, su valor es bajo de media, de lo que podemos concluir que debe hacerse un esfuerzo extra para diferenciarlas. Sobre esta cuestión en específico recibimos retroalimentación concreta, citando los colores dados a las criaturas como causa de confusión. Por ello, se implementó un sistema de pigmentación en la representación que modifica la textura dada a las criaturas con un color según sus atributos, ayudando a la distinción visual.

Figura 6.26. Distribución de las respuestas de los encuestados

La pregunta representada por la Figura 6.26 evalúa si se han representado las distintas habilidades y rasgos presentes en el cromosoma de cada criatura de manera que estos son discernibles. Las respuestas a esta pregunta son mixtas, por razones sobre las que se nos proporcionó retroalimentación: los elementos que representaban las habilidades en las criaturas eran pequeños o de forma poco definida (por ejemplo, en ese momento la habilidad de pelo era representada por un pequeño flequillo blanco apenas visible). Sin embargo, la mayoría de los votos tiende a estar a favor de la noción, así que aunque imperfecto el sistema implementado durante las pruebas resultó ser funcional.

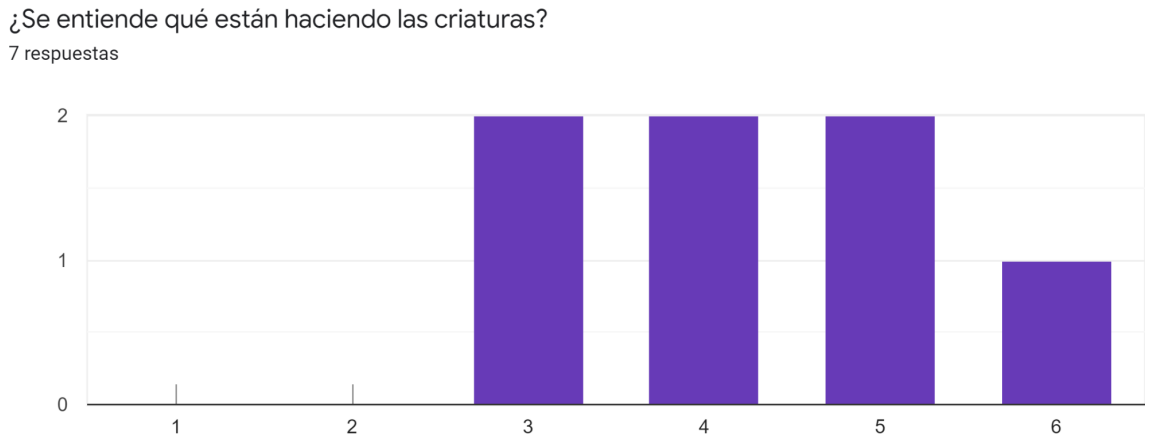


Figura 6.27. Distribución de las respuestas de los encuestados

La pregunta representada por la Figura 6.27 evalúa si se puede distinguir a simple vista el estado de una criatura y la acción que está realizando en ese momento. En la simulación, además de una serie de efectos de partículas que acompañan a ciertas acciones (como el combate entre criaturas), se muestra un texto descriptivo encima de las mismas que indica el estado en el que se encuentran. Podemos deducir que la respuesta positiva a esta pregunta se debe a estos elementos, pero que se podrían refinar ligeramente, ya que pueden ser difíciles de ver cuando muchas criaturas se encuentran juntas.

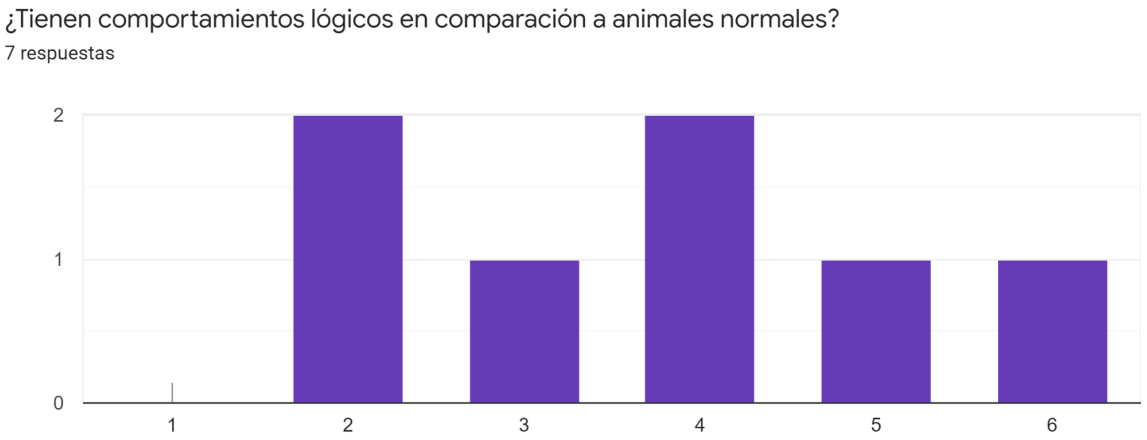


Figura 6.28. Distribución de las respuestas de los encuestados

La pregunta representada por la Figura 6.28 evalúa si el comportamiento de nuestras criaturas es comparable al de animales reales. Las respuestas son mixtas. Es muy complicado traducir el comportamiento animal a una simulación precisa; además de que nuestra simulación, por funcionar mediante *ticks*, la hace más "robótica". La media tiende a ser positiva, sin embargo, así que se puede suponer que hasta cierto punto sí se pueden apreciar comportamientos lógicos en las criaturas.

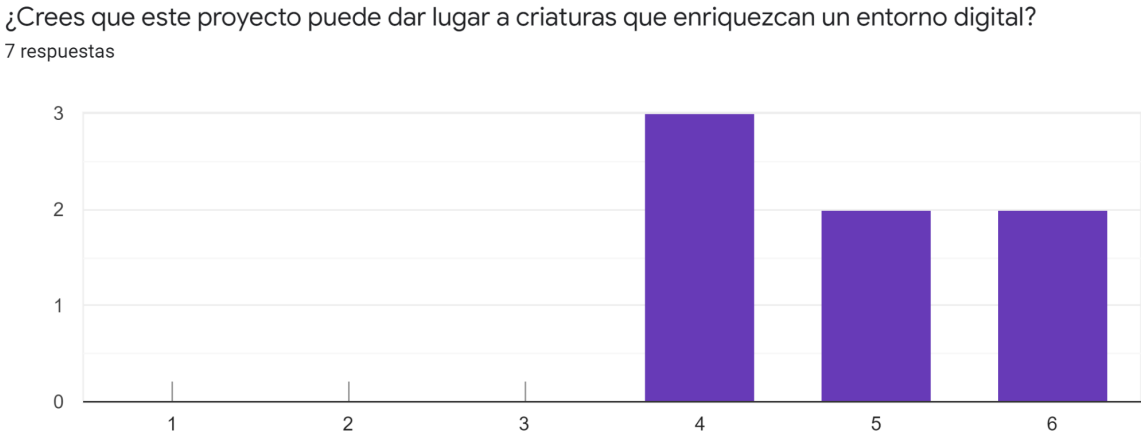


Figura 6.29. Distribución de las respuestas de los encuestados

La pregunta representada por la Figura 6.29 nos sirve para evaluar si estamos bien encaminados para cumplir el objetivo planteado por nuestro proyecto.

Afortunadamente, la respuesta a esta pregunta es mayormente positiva, por lo que podemos pensar que nuestro objetivo se ha cumplido.

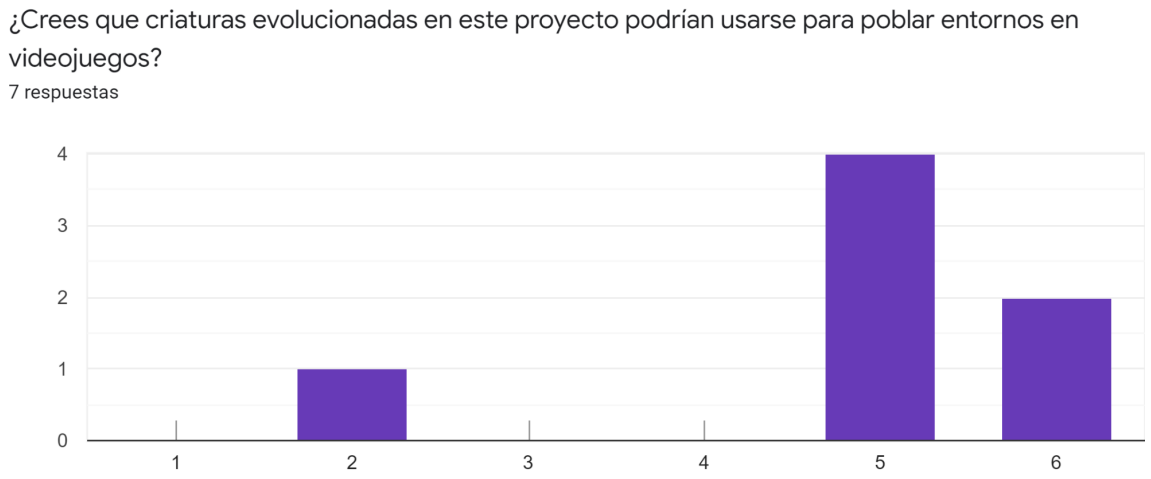


Figura 6.30. Distribución de las respuestas de los encuestados

La pregunta representada por la Figura 6.30 profundiza sobre la anterior, refiriéndose explícitamente a los videojuegos. Las respuestas, otorgadas por personas relacionadas con el mundo del videojuego y que están familiarizadas con él son mayoritariamente muy positivas. Esto refuerza la conclusión anterior, y da fuerza a la suposición de que desarrolladores del sector verían valor en la herramienta proporcionada por nuestro proyecto.

7. Discusión

Como se ha podido observar en los resultados anteriores, todavía hay algunos apartados de este proyecto que se pueden y deben pulir. Sin ir más lejos, las pruebas de usuario descritas en el anterior apartado, aunque puedan considerarse concluyentes, están severamente limitadas por el número de individuos sobre los que se han realizado. Es sabido que en el caso de la estadística, sobre todo cuando se está evaluando algo pensado de cara al público general, cuantas más opiniones se proporcionen mejor será la conclusión que se saque en base a los datos. Por lo cual, serían necesarias muchas más personas para poder llevar a cabo un estudio con el cual se pudiesen sacar conclusiones fehacientes.

Comenzando con la generación del terreno y mundo, este se genera de forma satisfactoria generando sectores del mundo suficientemente diferentes entre sí en cuanto a propiedades como temperatura, humedad y altitud. También genera la flora de forma que se corresponda correctamente a la esperada por estos factores. La generación de masas de agua, en cambio, es insuficiente. El generador no es capaz de generar ríos que den sentido a los lagos, y el nivel del agua es constante, impidiendo generar depresiones o lagos de montañas. No obstante, la poca presencia de agua se puede solventar modificando los valores del generador, aunque eso no soluciona los problemas mencionados anteriormente.

Siguiendo las definiciones expuestas en el apartado 2.1 sobre qué es un ser vivo, este programa realiza una recreación de una criatura que encaja cómodamente en las definiciones que se han considerado más relevantes en el contexto de este estudio. No se cumplen siguiendo un sistema basado en el realismo, ya que esto requiere un nivel de detalle contraproducente. Solo la representación acertada de un sistema metabólico resultaría extremadamente costoso. En lugar de esto, se han creado una serie de características que simplifican en gran medida la mayoría de procesos biológicos de las criaturas, de forma que se simulan necesidades y también una capacidad de improvisación.

De los siete pilares que caracterizan la vida según la definición que se consideró más pertinente, las criaturas simuladas son planes organizados con la información y comportamientos necesarios para suplir sus necesidades y permitir la reproducción; son capaces de sufrir cambios para aumentar su probabilidad de supervivencia, tanto de forma individual, recordando experiencias que basan sus acciones para evitar peligros o frecuentar lugares seguros, como a nivel de especie, evolucionando para adaptarse a su entorno gracias al proceso de evolución. También requieren energía en forma de

recursos como alimento o hidratación, que permiten la simulación de un sistema metabólico, el cual a su vez simula un proceso de regeneración, curando daños recibidos mientras se cumplan las necesidades básicas de una criatura. Los elementos que se acaban de mencionar componen cinco de los siete pilares. Sin embargo, hay efectivamente dos pilares que no se han representado. Esto se debe a que están estrictamente relacionados con los procesos biológicos, y más concretamente químicos, que toman lugar en un ser vivo en todo momento. En otras palabras, las criaturas no están compartimentalizadas ni cuentan con formas de reclusión. Esto se debe a un aspecto anteriormente mencionado: el extremo coste de simular de forma realista eventos como reacciones químicas, por lo que estos pilares se han ignorado conscientemente.

Se concluye, entonces, que de los siete pilares se representan de forma exitosa cinco, la mayoría, y los que se ignoran se han dejado de lado a propósito debido a limitaciones técnicas justificables. Al fin y al cabo, la generación con la implementación actual ya tarda una cantidad de tiempo notable. Si se hiciera más realista, esta llegaría a un punto que roza lo irrazonable. Por lo cual, las criaturas que genera este proyecto se pueden considerar de forma justificable “seres vivos”, por lo menos en el contexto informático.

Por otra parte, la generación de las criaturas, haciendo uso de un proceso procedimental para crear especies primigenias en el comienzo de la simulación, también se considera adecuada, pues las criaturas generadas son diferentes dentro de los confines y límites establecidos por las implementaciones del cromosoma y máquina de estados. Aún así, se ha intentado hacer estos elementos lo más flexibles posibles, de forma que se puedan configurar y personalizar. El cromosoma cuenta en el momento de redacción de este documento con 27 genes diferentes capaces de estar relacionados entre sí, y la máquina de estados recibe una gran cantidad de valores que definen los comportamientos de los seres vivos. Además, también es configurable el mundo y su generación, teniendo varios tipos de mapas o incluso recibiendo propios del usuario, al igual que permite que este establezca las funciones y parámetros relacionados con la generación del mundo. Así pues, se considera que el programa se puede adaptar a las necesidades del usuario. No obstante, también es cierto que a pesar de ser configurable, requiere un esfuerzo mayor de instrumentalización para hacer uso de todos los aspectos que se pueden configurar. Por ejemplo, la creación de una generación del mundo o una implementación propia de las cualidades de las criaturas requieren de la creación de clases nuevas, lo que conlleva la modificación de la librería.

A mayores, cuando una simulación acaba tras el tiempo estipulado y las criaturas han conseguido sobrevivir, suponiendo que no se ha creado un mundo donde la supervivencia es trivial, se demuestra en el acto que estas criaturas han evolucionado y se han adaptado a su entorno. Como consecuencia, cuando la simulación se completa exitosamente, las especies restantes se consideran evolucionadas. Por otro lado, conseguir que el programa simule la cantidad de años dada por el usuario no siempre se logra, cosa que, aún natural en una simulación con estas funciones, resulta inconveniente, pues la idea es que se deje la simulación en ejecución sin la supervisión del usuario. Si el programa termina porque ha tomado lugar una extinción y ya no quedan criaturas vivas, esto es un evento razonable dada la intención de la simulación,

pero actualmente requiere de un reinicio manual, y por lo cual hay que revisar el estado de la simulación o uno se arriesga a perder bastante tiempo en vano. Sin embargo, reiniciar la simulación tampoco es una solución perfecta: si reiniciamos por completo la simulación podría darse el caso de que el programa entrase en un bucle infinito de reinicios; y si se introdujera un nuevo grupo de criaturas sobre el mundo ya existente (como en el apocalipsis de la programación evolutiva), los datos producidos irían en contra del propósito del proyecto pues se estaría ignorando deliberadamente la extinción anterior. Es por esto que habría que alcanzar un mayor grado de depuración en el código y ajustar algunas transiciones de la máquina de estados con el fin de que la mayor cantidad de simulaciones sean exitosas.

Cabe mencionar también que el comportamiento que poseen las criaturas es a grandes rasgos el esperado, consiguiendo dotarlas de una gran cantidad de funcionalidades basadas en el cumplimiento de necesidades, lo cual genera acciones realistas en los individuos. Por ejemplo, las criaturas son capaces de determinar mediante algoritmos de búsqueda de caminos trayectos a la hora de moverse por el mapa, haciendo uso de sus capacidades de vuelo o escalada de manera apropiada. De esta manera, los movimientos que realizan resultan coherentes y realistas, lo cual ayuda a cumplir la hipótesis establecida, además que se ayuda a optimizar la supervivencia de las criaturas. Sin embargo, algunas prioridades o respuestas a situaciones externas se manejan de tal manera que causan problemas para la coherencia del comportamiento de la criatura o su supervivencia, por lo que deberían ser revisadas para mejorar todavía más el funcionamiento del sistema. Se considera, a pesar de los elementos mejorables en la implementación actual, exitosa la obtención de comportamientos inteligentes. A esto se puede sumar el correcto funcionamiento de los algoritmos y representación genéticos, que también se incluyen en el campo de la inteligencia artificial.

La representación de *Unity*, aunque básica, representa correctamente a las criaturas y el mundo a grandes rasgos. Debido a su sencillez, deja que desear en el apartado gráfico de las criaturas, ya que hay una cierta escasez de modelos y texturas, y los que hay no son de alto nivel. Sin embargo, el proyecto de *Unity* no es el eje central del estudio, sino un mero prototipo demostrativo de la utilidad de los datos resultantes, al igual que un ejemplo simple de cómo se podrían interpretar. Aún con esto, cuenta con fallos que no se pueden achacar al papel secundario de esta escena. Al generarse procedimentalmente las criaturas, ocasionalmente se da el caso de que ciertas piezas que componen a los individuos de una especie se solapen con u oculten dentro de otras de mayor tamaño.

8. Conclusiones y trabajo futuro

Una vez presentados todos los elementos previos y propios del desarrollo, resta alcanzar conclusiones en base a la información descrita hasta este punto en el documento y discutir sobre posibles expansiones que enriquezcan el proyecto en un futuro.

8.1. Conclusiones del proyecto

Este es un estudio sobre la creación de un generador procedimental de criaturas basado en datos que se pueden importar e interpretar. Un animal es la consecuencia de un proceso de adaptación en su entorno, y por lo cual sus propiedades representan hasta cierto punto las características de este. Este es un factor que se analiza y explota, recibiendo un terreno con unas características y creando una serie de criaturas que interactúen entre sí y con el mundo que habitan, simulando la evolución para crear un ecosistema creíble, realista y sostenible. Para lograrlo, se ha investigado y hecho uso de inteligencia artificial, generación procedimental como tal, ingeniería de comportamientos, y programación evolutiva, además del motor externo que se ha utilizado para mostrar los datos en un entorno tridimensional en tiempo real.

Para evaluar el correcto funcionamiento de este programa se han usado dos métodos: una prueba con usuarios, en la que se pide a personas ajenas al proyecto que evalúen los datos resultantes en base a su experiencia, y usar un sistema de telemetría basado en eventos que permite la recolección de datos para su posterior análisis. De esta forma se puede llegar a conclusiones que permitan la mejora y depurado del proyecto.

Así pues, se concluye que la contribución de este estudio es positiva porque se cumplen los objetivos del proyecto, siendo los principales y más notables la generación de un mundo virtual coherente con las variables de temperatura, humedad y altitud, que permiten la generación de una flora y posteriormente fauna. Más importante todavía, las criaturas que componen esta fauna interactúan exitosamente entre sí y con su entorno, describiendo comportamientos lógicos y adquiriendo y modificando sus características físicas en base al proceso de evolución dictado por la selección natural. La simulación funciona correctamente y produce datos válidos para su interpretación en otros entornos, como podrían ser otras simulaciones o videojuegos. El funcionamiento del sistema se ha conseguido y comprobado haciendo uso del previamente descrito sistema de telemetría, y la evaluación con usuarios, aunque no consta de tantos individuos como sería deseable, presenta igualmente datos empíricos que apoyan la idea del uso del

proyecto, si bien es necesario mejorar ciertos elementos, la mayoría centrado en la depuración de los comportamientos de las criaturas y los valores de la simulación.

8.2. Trabajo futuro

Dadas las dimensiones del proyecto, aún teniendo un equipo de desarrollo de un tamaño considerable, hay una notable cantidad de aspectos y detalles que se han dejado al margen durante este estudio. Ejemplos de esto son la falta de un depurado intensivo del código por falta de tiempo, que utilizando el sistema de telemetría no sería particularmente complejo; un mayor y mejor equilibrado de las funcionalidades de las distintas habilidades y cualidades de las criaturas, ya que es razonable suponer que tras una cierta cantidad de casos se empezará a ver cómo unas son más recurrentes que otras; el sistema de combate, que se puede y de hecho está pensado para ser ampliado, añadiendo nuevos efectos de estado (actualmente solo está el veneno), etc; más cualidades y habilidades, con el objetivo de representar una mayor cantidad de características animales…

El pulido del proyecto en sí representa una gran cantidad de tiempo y trabajo que invertir en un futuro para que la simulación alcance el mayor nivel de eficacia posible a la hora de generar criaturas. Sin embargo, se pueden desarrollar otros aspectos no relacionados con el contenido actual, como podrían ser expansiones de la vida animal en otros medios. A pesar de que se representa movimiento aéreo y arbóreo junto al terrestre, se hace un gran hincapié en el último, y los otros son adaptaciones de este. Como consecuencia, no hay una complejidad notable, por ejemplo, en la interacción entre criaturas aéreas, ya que no tienen comportamientos específicos. Esto lleva a la idea de crear distintos “mundos” dependiendo del medio, separando completamente el funcionamiento de estos. Ejemplos de esto podrían ser divisiones del espacio aéreo en altitudes y propiedades genéticas que dicten cómo y dónde pueden volar las criaturas, o especies subterráneas que habitan el subsuelo.

Este último concepto, llevado a un extremo, se puede extrapolar a la creación de un ecosistema acuático. Explicado de forma sencilla, la simulación actual, con sus tres medios (aéreo, arbóreo y terrestre), representaría una parte de la nueva simulación, y la otra daría lugar a un medio acuático con sus propias divisiones del espacio en el que un nuevo proceso evolutivo tendría lugar. De forma similar, tendría diferentes temperaturas, entidades generadoras de nutrientes, como podrían ser esponjas, y altitudes, que definirían elementos fundamentales para la supervivencia de las criaturas. Por ejemplo, en altitudes altas existen organismos que realizan la fotosíntesis, mientras que en las profundidades no llega luz natural y los animales se alimentan de la materia orgánica residual que desciende desde capas más cercanas al nivel del mar. Esto daría lugar a nuevas interacciones entre los dos tipos de ecosistemas, además de una nueva generación de terreno, dando lugar a ríos y nuevas y distintas masas de agua, creando una simulación más unificada.

Todas estas ideas están basadas en traducir el funcionamiento de la vida y los ecosistemas naturales reales a una simulación, los cuales son una fuente sin fin de

posibles ampliaciones. Al fin y al cabo, representar la naturaleza en un entorno virtual es una tarea prácticamente infinita en su complejidad.

Otra dirección en la que se puede expandir el proyecto es precisamente el contrario al que se ha estado discutiendo hasta el momento. En vez de intentar recrear los entornos y comportamientos naturales se puede intentar crear nuevos, sin basarse completamente en casos ya existentes. Esto puede presentar un valor notable, ya que se podrían generar posibles evoluciones que todavía no se han dado en la vida real, siempre y cuando se implemente correctamente el sistema previamente descrito. Sin embargo, esta es precisamente la dificultad de esta idea: implementar comportamientos que ni los creadores son capaces de imaginar. Se requeriría de una reconstrucción de la máquina de estados y el sistema de cualidades y habilidades, pero esta sería una ampliación viable y de gran valor.

9. Contribución de los miembros

9.1. Daniel Cortijo Gamboa

Al igual que el resto de integrantes del grupo, Daniel participó en la repartición y planteamiento de tareas, estando siempre presente en las reuniones y revisiones semanales y colaborando con otros integrantes cuando la tarea lo requería.

En el inicio del proyecto, Daniel tomó la tarea de crear el entorno físico base en el cual las criaturas se desarrollarían. Acompañó en esta tarea a Georgi, cuyo trabajo previo en el campo de la generación de terreno fue de gran valor para el proceso. Juntos crearon la primera iteración de lo que acabaría siendo el mundo, una matriz simple de casillas en la que se colocó un objeto que se movía de manera aleatoria sobre el mismo: la criatura primigenia, todavía carente de comportamientos complejos. Posteriormente, haciendo uso de ruido de Perlin, implementaron un proceso simple de generación de terreno que rellenaba los atributos del mapa de manera aleatoria pero coherente. Esta generación se perfeccionaría más adelante.

Daniel y Georgi continuaron su asociación para implementar un sistema de coste energético para las posibles acciones de las criaturas, que hasta ahora se hacían arbitrariamente cada ciclo de la simulación. Implementaron un sistema de energía en la criatura y costes para cada acción realizable que fue la base de los comportamientos de las mismas. Más tarde y con ayuda de aquellos compañeros que habían implementado la máquina de estados de la criatura, unificaron el sistema de energía con la misma para obtener el sistema de acciones que se usa actualmente en el proyecto.

Más adelante, Daniel colaboró en la implementación de los primeros estados de la criatura con Eloy y Pablo, dando lugar a un comportamiento lógico en la criatura. Daniel fue responsable de los comportamientos de los estados Combat y Escape en todas sus iteraciones, actualizándolos conforme la manera de percibir el mundo de la criatura se volvía más compleja. Realizó ajustes al ser actualizados el sistema de navegación de la criatura y su sistema de prioridades. Para evitar errores con el estado de huida, implementó un sistema a prueba de fallos para impedía que la criatura fuera capaz de quedarse de forma permanente en un esfuerzo fútil de huír de un agresor si estaba acorralada. Adaptó el estado de combate para que relajara la caza por parte de criaturas carnívoras, y asistió en la programación de combate en manada. También se ocupó de ajustar las transiciones de la máquina, sobre dentro de los estados de combate y huida y aquellas que los conectaban con otros estados.

Daniel creó el sistema de efectos de estado de las criaturas, que les permite tener modificadores de estado con diversos efectos al ser aplicados o quitados o mientras duren. Implementó también el único efecto usado actualmente en la simulación, el de veneno, y permitió su aplicación en la acción de combate de la máquina de estados. Implementó efectos de estado adicionales, como el parásito, que fueron descartados.

Daniel creó la primera iteración del sistema de interacción entre criaturas, que con ayuda de Pablo fue reelaborado dando lugar al que se usa actualmente, permitiendo asignar ciertos métodos a interacciones entre criaturas. Principalmente, Daniel implementó las interacciones relacionadas con el combate: el recibimiento de daño, el envenenamiento y la devolución de daño por púas, que son la base de la interacción interespecie actual.

Daniel fue responsable de las necesidades corporales de las criaturas, ideando las fórmulas que dan lugar a los gastos por las criaturas. En un principio se optó por valores solamente dependientes de atributos de las criaturas, pero al ver resultados demasiado extremos con frecuencia se pasó a un sistema que modulaba un valor base según las características de las criaturas con límites establecidos. Además, hizo que las criaturas pudieran regenerar su salud.

Daniel asistió también a Georgi en el ajuste de las plantas comestibles, modificando las plantas existentes para permitir la distinción entre plantas no comidas y ya comidas, y haciendo que estas últimas crecieran de nuevo. Le dio utilidad a los puntos de vida de las mismas, previamente sin uso, para que las plantas fueran consumidas por partes.

El sistema utilizado en el sistema de telemetría, tanto para saber las causas de daño como las de muerte, fue ideado por Daniel, con el fin de saber por qué las criaturas morían y qué comportamientos se habían de cambiar para evitarlos. Este sistema fue expandido con la introducción del tracker de eventos, lo que permitió un ajuste del proyecto para unos resultados más satisfactorios. Con dicho sistema simuló numerosas instancias de la simulación permitiendo su análisis y ajuste posterior.

Además, Daniel introdujo el apocalipsis a la simulación como alternativa a una ejecución fallida; una nueva generación de criaturas sobre el mismo terreno de de simulación en caso de que todas las criaturas se extinguieran con el fin de poder otorgar resultados satisfactorios pese al fracaso inicial, aunque finalmente fuera descartado tras consideración del supervisor.

Durante todo el proyecto, Daniel participó activamente en la resolución de errores menores o tareas pendientes que fueron surgiendo durante el desarrollo. Esto implicó la transición de múltiples variables del código a parámetros configurables, a la vez que comprobaciones en el Validator de que las mismas estaban correctamente configuradas, y el suplemento de comentarios a partes del código que carecían de ellos.

Por último, Daniel contribuyó notablemente a la memoria en distintos apartados. Su contribución fue constante a lo largo del proyecto, incluyendo la asistencia a otros compañeros revisando y corrigiendo sus aportaciones además de los comentarios dejados por el supervisor.

9.2. Andrés de la Cuesta López

En conjunto al resto de compañeros del proyecto, participó activamente en el proceso de planificación semanal y mensual de las tareas a realizar, participando siempre en las tareas relacionadas con las metodologías ágiles utilizadas durante el desarrollo del proyecto.

Al iniciar el desarrollo del proyecto, nos dividimos en parejas, siendo el encargado principal junto a Daniel González Cerdeiras de la parte relacionada con la computación evolutiva. A diferencia de Daniel, Andrés ya tenía conocimientos previos sobre este tema, por lo que la mayoría de las partes más importantes sobre este tipo de algoritmos como pueden ser el cruce, la mutación o la selección ya los conocía. Empezaron usando una librería que facilitaba toda la arquitectura base de la computación evolutiva, pero dado que en nuestro proyecto hay muchas partes que no necesitamos como puede ser tener una población fija o criaturas a seleccionar para la reproducción, se decidió que era mejor no utilizar la librería y hacerlo todo desde el principio con una implementación propia, teniendo así mucho más control sobre lo que se puede hacer con el sistema.

El desarrollo que se hizo sobre la programación evolutiva fueron los cromosomas que contienen las criaturas, que tuvieron que rehacer dos veces debido a fallos encontrados sobre éstos, y las funciones de cruce y de mutación utilizadas durante la reproducción de dos criaturas.

Continuando con la parte del cromosoma, trabajando todavía con Daniel González, idearon y desarrollaron la manera de interpretar el cromosoma y pasarlo a los atributos que contienen las criaturas. Ambos se encargaron de realizar la transformación del cromosoma a estas cualidades de las criaturas, asegurándose de que todo era correcto, aunque a lo largo del desarrollo del proyecto se han ido modificando alguna de estas transformaciones entre el resto de los miembros con el objetivo de que no hubiese ninguna característica mal interpretada o que nos interesaba hacerlo de otra manera.

Después de esto, idearon y realizaron el historial de las criaturas que hay durante la simulación, separando las criaturas existentes y las que se han extinguido durante toda la simulación, con el objetivo de poder exportar las criaturas que se quieren y poder crear el árbol filogenético, constituyendo dos de los datos de salida generados por el programa.

Con el fin de hacer más configurable las cualidades que extraen del cromosoma, concretamente las habilidades, decidió hacer un archivo JSON en el que aparecieran todas las habilidades y el porcentaje necesario para desbloquearlo, haciendo así que puedan haber habilidades que el usuario pueda considerar más fácil de desbloquear o más difícil.

Tras tener un sistema de evolución genética funcional y la configuración de los cromosomas y la lectura de éste a cualidades de la criatura, se dio por finalizado este apartado genético del desarrollo y Andrés se puso a desarrollar en otros ámbitos. Empezó a ayudar a otros compañeros, y pasó a estar junto a Pablo y Daniel Cortijo haciendo la máquina de estados de las criaturas.

Andrés hizo gran parte de los estados y transiciones que hay en el macro estado *SAFE,* como pueden ser los relacionados con la reproducción, alimentación y descanso. También fue el encargado de realizar los estados y transiciones que las criaturas tienen debido a la habilidad *Paternidad*.

En este momento, la máquina de estados ya era funcional y Daniel González había implementado la memoria de las criaturas, que se tuvo que rehacer debido al alto consumo de RAM que necesitaba, así que en conjunto a él y también a Pablo, se pusieron a rehacer la memoría hasta alcanzar el estado final, teniendo solo que retocar alguna función más adelante. Junto a Pablo, realizó bastantes iteraciones sobre cómo elegir un punto al que las criaturas fuesen a explorar.

A estas alturas del proyecto se implementó el Apocalipsis, que venía a ser reiniciar la simulación si las criaturas se morían antes de acabar la ejecución, siendo Andrés el que hizo la parte relacionada con la taxonomía, es decir, exportar las especies que más tiempo habían sobrevivido y el árbol filogenético.

Tras esto, el proyecto se encontraba en un punto en el que prácticamente sólo quedaba probar el correcto funcionamiento de la máquina de estados y de la simulación en sí, así que en conjunto a la mayoría de los integrantes del proyecto, se puso a probar, encontrar errores y arreglarlos. De forma paralela, encontró errores en la parte de Unity, como que las posiciones del mapa y de las criaturas no se correspondían a las que deberían ser, que la información que aparece encima de las criaturas no se actualizaba correctamente…

Es en este punto cuando, ya varios meses después de dar por finalizado el desarrollo sobre el cromosoma, encontró con que había algunos fallos en el cromosoma en cómo se calculaban las relaciones entre genes, haciendo que algunos genes siempre estuvieran al máximo de su capacidad o al mínimo, debido a la dependencia que tenían. Inmediatamente tras ver esto, se puso a arreglarlo.

Ayudó activamente a idear y realizar las pruebas de usuario utilizadas casi al final del proyecto. Respecto a la parte de Telemetría, Andrés realizó gran parte de los eventos e instrumentalización del proyecto, y en conjunto con el resto del equipo, sacó conclusiones de los datos obtenidos y encontró gracias a este sistema errores que no se habían observado en la máquina de estados de las criaturas.

9.3. Daniel González Cerdeiras

Lo primero que cabe mencionar sobre las aportaciones de Daniel es la idea del proyecto en sí, por lo que ideó todo el concepto del proyecto. Por supuesto, todo el grupo realizó los cambios pertinentes en base al criterio mayoritario y se realizaron algunas modificaciones tras reuniones con el tutor.

Al igual que el resto de miembros, participó activamente en el proceso de planificación de una metodología, tomando parte en las discusiones sobre qué herramientas usar y cómo usarlas, siendo un miembro activo en la división del trabajo y participando en todas las actividades relacionadas con las metodologías ágiles que han tomado lugar en el proyecto.

Una vez comenzado el desarrollo del proyecto, este miembro del equipo hizo una gran cantidad de programación relacionada con el apartado de computación evolutiva, junto con Andrés de la Cuesta López. En sus contribuciones se concretan otros aspectos de este desarrollo. Llevar a cabo esta programación requirió de un estudio más detallado que el inicial que se hizo sobre la premisa del estudio, aprendiendo sobre los diferentes medios y técnicas que permiten la implementación de una genética artificial. Estudió una pluralidad de implementaciones con distintos objetivos y funciones con el fin de informarse sobre cómo se debería realizar esta parte del proyecto de forma coherente de cara a los intereses de este.

Esta programación evolutiva comprende la concepción de la estructura del cromosoma al igual que cómo este debería ser llevado a cabo y funciones genéticas, que dictan la forma en la que los cromosomas de las criaturas se mezclan, recombinan y mutan en la reproducción en forma de funciones de cruce y mutación. Por supuesto, no solo colaboró en la creación de estos elementos, sino también en su pulido y puesta en marcha, solvencia de errores o fallos en el código y comprobación del funcionamiento y utilidad.

Posteriormente, aprovechando su conocimiento sobre el sistema genético de las criaturas, con sus cromosomas, genes y funciones genéticas, ideó y realizó la creación de los atributos, trabajando todavía con Andrés. Recordemos que estos son las interpretaciones de la información de los genes a valores que la lógica de la simulación puede utilizar para calcular las propiedades y características de cada criatura. Estas características fueron denominadas como “cualidades”, y al igual que con los atributos fueron elementos desarrollados por Daniel y Andrés. Pusieron todos los cálculos iniciales de estas propiedades para todas las criaturas, y se aseguraron de su correcto funcionamiento aunque a posteriori hubiera que modificar o reemplazar algunos cálculos que se hicieron en un primer momento.

Además, todavía trabajando con Andrés, creó un historial de criaturas que guarda constancia de las especies que existen en un momento dado de la ejecución de la simulación y las que se han extinguido, manteniendo las relaciones entre las criaturas para poder crear un árbol filogenético, que constituye parte de los datos de salida del programa. Para poder hacer esto, tuvo que establecer qué constituye una especie y cómo se diferencian dos especies, dando lugar a la comparación genética de especies por porcentaje de similitud en el que se basa el programa.

Continuando con la temática de la programación evolutiva, pero saliendo a un ámbito más general, se decidió pronto durante el desarrollo que los cromosomas y sus peculiaridades debían poder ser configurables fácilmente por el usuario, y esto dió pie a Daniel a hacer el sistema de exportación e importación del programa. No fue necesario debido al uso de librerías externas que tuvieran que realizar manualmente la interpretación de los archivos JSON que se decidieron usar, pero el propio sistema requiere de unas clases tanto de utilidad como de objetos exportables necesarias para exportar e importar los datos que se decidieron.

A continuación, según el proyecto tomaba forma, se propuso crear la entrada y salida de datos del programa, tanto por consola como creando “prompts” de Windows para facilitarle la tarea al usuario. Lo realizó pidiendo a través de los medios previamente

mencionados unos ciertos datos al usuario, comprobando la validez de estos y estableciéndose como datos del programa para su ejecución. Posteriormente, tras una notable cantidad de tiempo de desarrollo, volvió a este sistema para actualizarlo con todos los nuevos o modificados datos que diferenciaban la simulación actual de la pretérita. En cierta relación con este punto, ayudó a Georgi a crear un Validator, que evalúa la validez de los datos y archivos que el usuario le pasa al programa.

Mientras la máquina de estados estaba en desarrollo, Daniel creó la iteración inicial del sistema de memoria, basado en ser lo más preciso posible, pero dado el consumo de memoría que esto requería hubo que desechar esta implementación y crear una nueva. En este momento, para acelerar el desarrollo de esta, Andrés y Pablo se unieron a Daniel para poder tener lo antes posible una máquina de estados funcional, ya que esta depende de la memoria y mente, concepto que apareció en la segunda iteración.

Finalmente, en cuanto a implementación básica se refiere, Daniel realizó junto con Eloy la interpretación de los datos de las criaturas en Unity, creando modelos físicos en base a sus características. A parte de esto, Daniel, que cabe destacar que no es artista, modeló las partes de las criaturas que no son formas básicas, y además implementó un shader en la textura del skybox (el cielo) de Unity para representar visualmente el ciclo día y noche.

A partir de este punto, los esfuerzos de Daniel se centraron en dos objetivos concretos: depurar el código y redactar la memoria. Durante una cierta cantidad de tiempo, estuvo arreglando los errores que surgían con las ejecuciones del código, modificando estados, transiciones, la mente, la memoria y los algoritmos de esta última mayoritariamente. A parte de arreglar y actualizar varias partes de la máquina de estados, introdujo algunas mejoras en los comportamientos, como tácticas de manada, que hace que las criaturas de la misma especie se defiendan entre sí. También, al igual que el resto de integrantes del grupo, colaboró en el planteamiento, recogida y análisis de datos tanto del sistema de telemetría como de la evaluación con usuarios.

Por último, Daniel ha dedicado una gran parte de su tiempo de desarrollo en la memoria, habiendo escrito la mayor parte de esta y supervisado el resto. Esta es una división de trabajo establecida en el grupo de forma interna, ya que demostró ser la persona con la mejor expresión escrita del grupo. Como consecuencia, y aunque sí tomó parte en estas partes del proceso, véase el punto anterior, Daniel ha participado en menor medida en comparación a otros miembros en la depuración de errores, y particularmente en el "balanceado" de la simulación, ajustando valores y algoritmos para que esta tenga una distribución "normal" de características entre las criaturas, tasas de muerte, etc.

9.4. Georgi Mednikov

Al igual que el resto de integrantes del grupo, Georgi Mednikov participó en la repartición y planteamiento de tareas, estando siempre presente en las reuniones y revisiones semanales y colaborando con otros integrantes cuando la tarea lo requería.

Las aportaciones de Georgi Mednikov comenzaron con la implementación del movimiento de las criaturas, para ello tuvo que crear la base del terreno, en aquel

momento plano, haciendo que las criaturas se moviesen de manera aleatoria en la primera iteración. Posteriormente, ideó e implementó el núcleo de la generación de terreno, el cual se refinería durante los meses venideros.

A continuación, se centró en expandir la capacidad de actuar de las criaturas, añadiendo el sistema de *puntos de acción* y metabolismo. Al mismo tiempo definió las unidades de tiempo en la simulación, introduciendo así el ciclo día/noche.

En un comienzo, la visualización del terreno se realizaba por consola, esto cambió cuando implementó la posibilidad de exportar los terrenos a BMP, eliminando así la necesidad de generar mapas de solo 32x32 y mejorando la visualización. A su misma vez, comenzó a parametrizar la generación de terreno para que fuese más modular.

Después, dejó de lado durante un tiempo la generación de terrenos para implementar la longevidad de las criaturas, añadiendo la fórmula para disminuir proporcionalmente los atributos cuan más joven fuese la criatura. En ese espacio de tiempo también creó la clase *RandomGenerator* la cual unifica todos los generadores aleatorios del proyecto en uno solo.

Tras esto, sus esfuerzos se volvieron a centrar en la generación de terreno, esta vez implementado una primitiva generación de flora, para ello tuvo que implementar también los diferentes tipos de plantas presentes en el proyecto.

A consecuencia de la modularidad de la generación, la cual admite que se le cambien las funciones, Georgi Mednikov creó e implementó la clase *Validator*, la cual validaba si las funciones que recibía la generación eran válidas. También, debido al número incremental de parámetros que empezaba a recibir la generación de terreno, creó la clase *WorldGenConfig*, que contenía toda la información necesaria para generar terreno, permitiendo también poner valores por defecto.

Una vez más, sus esfuerzos se centraron en la implementación del movimiento de las criaturas, implementando esta vez *A** de bajo nivel, para ello tuvo que idear e implementar las fórmulas que posteriormente serían usadas para el movimiento. Este proceso fue largo debido a la complejidad de la búsqueda de una heurística personalizada para la simulación. Aunque sus esfuerzos se centrasen en otras cosas, continuó trabajando en *A** arreglando los fallos que iban surgiendo.

Pasó a continuación a trabajar en el prototipo de demostración de Unity, implementando una manera de visualizar el mapa usando los terrenos de Unity, esto le llevó a buscar los modelos de plantas que se ven actualmente en el proyecto. Para terminar con el aspecto visual del proyecto hizo que el mapa exportase una textura para su posterior uso en Unity.

Como ya se menciona la generación de terreno, este está inspirado en el sistema de clasificación de zonas de vida de Holdridge, para realizar una comparación de nuestro sistema con el original creó una mapa que se exporta el cual representa las zonas descritas por Holdridge.

A consecuencia de la creciente llegada de informes de que *A** se quedaba bloqueado en un bucle infinito, Georgi Mednikov centró todos sus esfuerzos en arreglar este error. Al inspeccionar el error descubrió que *A** no se bloqueaba si no que tardaba mucho si el destino se encontraba muy lejos del origen. Esto le llevó a la implementación de las regiones de Voronoi y la implementación de *A** de alto nivel, arreglando los problemas ya mencionados.

Su trabajo prosiguió con la creación de las plantillas de generación de terreno. Esto se hizo para poder explorar otro tipo de generación de terreno, ya que las criaturas morían muy rápido, Georgi Mednikov pensó que podría ser por falta de agua y por ello implementó plantillas que tenían más agua de promedio. Durante este proceso también hizo que el mapa se exportara a JSON junto con Daniel González.

Con la implementación de las plantillas, se reportaron nuevos fallos de *A**. Georgi se centró en investigar estos errores, lo cual le llevó varias semanas, para finalmente determinar que procedía de la máquina de estados y no se *A** como se había informado. Esto se debe a que los nuevos tipos de mapa presentaban una nueva gama de restricciones y la máquina de estados no estaba preparada. Lo cual llevó a cambiar todos los estados que dependieran de *A**.

Con *A** arreglado, Georgi comenzó con la optimización del proyecto para que fuese más rápido, ya que una serie de estudios preliminares mostraban que el proyecto tardaría 30 días en terminar una ejecución. Este proceso demostró tener una complejidad alta, ya que los cambios que hiciese no tenían que afectar a cómo funcionaba el programa. Aun con estas restricciones consiguió disminuir el tiempo necesario a 33 horas haciendo uso de un profiler para localizar zonas problemáticas de código.

Finalmente, sus esfuerzos se centraron en realizar simulaciones y arreglar los errores que surgían. Al mismo tiempo, realizó cambios en los valores por defecto de criaturas, ayudando a mejorar su supervivencia en la simulación.

De manera paralela, escribió en la memoria y ayudó en la implementación del sistema de telemetría del proyecto, haciendo arreglos al proyecto según se necesitase.

9.5. Eloy Moreno Cortijo

Eloy, junto al resto de integrantes del grupo, ayudó en la repartición de tareas y estuvo presente en todas las reuniones y las revisiones semanales, y colaboró con otros integrantes para realizar tareas de mayor tamaño.

En un principio, creó la estructura del proyecto junto a Pablo, dividiéndolo en tres proyectos. Uno de los proyectos incluye el núcleo de la simulación, que internamente tiene una lista de entidades, las cuales se dividen entre plantas y criaturas, que implementan las interfaces apropiadas para poder referenciarlas y actualizarlas. El segundo proyecto se encarga de iniciar la simulación, tomando la librería que genera el primer proyecto, y crear una parte visual utilizando *Windows Forms*. El tercer proyecto se encarga también de ejecutar la simulación, pero utiliza la consola de *Windows* como

base, mostrando la situación del mundo y la posición de las criaturas utilizando caracteres.

Más adelante, también junto con Pablo, Eloy implementó la máquina de estados de las criaturas primero "envolviendo" una librería de uso libre, y más adelante eliminando esta librería, pues su implementación limitaba la funcionalidad necesaria para el proyecto. Tras esto, ambos crearon una máquina de estados propia que se ajustaba a las necesidades del proyecto. Además, junto a Daniel Cortijo y Pablo implementó los primeros estados de la máquina, con transiciones iniciales y temporales, y colaboró en la integración de los puntos de acción en la misma. Posteriormente, Eloy realizó también ajustes a los distintos estados conforme la máquina iba actualizándose y expandiéndose.

Mientras creaba la máquina de estados, Eloy creó la percepción de las criaturas, consistente en recorrer una sección del mundo alrededor de estas y guardando aquellas entidades que se encuentren en esa posición. Para hacer esto, tuvo que modificar las entidades para incluir su posición en el mundo. Además, modificó cómo se almacenaban las entidades y criaturas del mundo para facilitar la búsqueda.

A continuación, modificó cómo varían las características de las criaturas según su edad e ideó una manera de exportar la máquina de estados para poder visualizarla, y comenzó con la representación visual de la simulación en *Unity*.

Creó la representación de las criaturas en *Unity* junto a Daniel González, primero utilizando las formas básicas del motor y leyendo los archivos que exporta la simulación, y más adelante con modelos creados por Daniel. La representación visual en *Unity* no estaba unida al proyecto de simulación, por lo que se necesitaron estructuras auxiliares que se eliminaron tras la unión.

Junto a Georgi, mientras creaba la representación de *Unity*, implementó la versión inicial del algoritmo de *A**, que considera todo el mapa para generar el camino, utilizando para ello una versión inicial de la heurística utilizada, y el cálculo de caminos aéreos utilizando el algoritmo de Bresenham. También arregló los errores que iban surgiendo de *A**, tanto los que provenían de la heurística utilizada como los producidos por los estados de las criaturas, y los producidos por el movimiento aéreo.

Tras esto, movió los parámetros de la simulación a un archivo que lee esta al principio mediante el *UniverseParametersManager*. Estos parámetros antes estaban junto al código, por lo que era necesario recompilar el proyecto si se quería modificar alguno de estos, y era necesario encontrar el parámetro en el mismo. Modificó el *Validador* para que comprobase que los *UniversalParameters* que lee de un archivo estén en rango o no tengan valores erróneos o imposibles. Este proceso se extendió varias semanas a causa de la gran cantidad de parámetros a incluir, esto le llevó a buscar en profundidad por el código aletargando el final de la tarea.

Una vez se comenzó a hacer simulaciones, arregló los errores que iban surgiendo en los estados y transiciones junto al resto de miembros del equipo, según iban surgiendo haciendo uso del *Pair Programming*. Cabe destacar el estado *GoToDrink,* el cual

provocó que se hicieran cambios en varias partes del código, y se planteara una pregunta de investigación cuando se hizo un análisis de la telemetría.

Incluyó en la representación de *Unity* un movimiento semi-continuo de las criaturas y una representación del estado de las mismas, modificando ligeramente el color del cuerpo según las características de las criaturas; y creó un sistema de eventos para mostrar cuándo una criatura ataca a otra (aunque este sistema acabó se eliminando), y creó sistemas de partículas para cuando la criatura come, bebe y ataca, mostrando su vida actual y su estado de la máquina de estados. Para facilitar las pruebas de usuario que se hicieron, incluyó un movimiento en primera persona en la representación. Junto a Georgi, supervisó e incluyó los modelos finales de las plantas de la representación de *Unity*. Junto a Pablo y Andrés arregló la posición en la que se colocan los objetos de la representación cuando el tamaño del mapa no tiene el valor por defecto.

Finalmente, se encargó en parte de eliminar código temporal o incompleto, y de escribir y revisar la memoria.

9.6. Pablo Rodríguez-Bobada García-Muñoz

Al igual que el resto de miembros, Pablo participó activamente en el proceso de planificación de una metodología, tomando parte en las discusiones sobre qué herramientas usar y cómo usarlas, siendo un miembro activo en la división del trabajo y participando en todas las actividades relacionadas con las metodologías ágiles que han tomado lugar en el proyecto.

Al comenzar el proyecto, se encargó junto con Eloy Moreno de establecer la arquitectura del proyecto, que luego se fue adaptando según se fue desarrollando el proyecto. Esto consistió en la concepción de los proyectos que se iban a realizar: el de la simulación que genera una librería dinámica y luego otros que lo utilizan para su representación gráfica) y la estructura del proyecto de simulación. Esta consiste en una simulación que contiene dos elementos principales, el mundo, que contiene las entidades que lo habitan; divididas en criaturas y entidades estáticas (plantas y cadáveres). Ambos se encargaron de la primera iteración de la simulación que englobó las de estructuras anteriores y definió el *pipeline* de ejecución de la misma. Pablo se encargó de la última iteración sobre la simulación, que corregía algunos aspectos de la misma y creó el *EventSimulation*, que mediante el uso de eventos, proporcionaba un fácil acceso a la información de la simulación mientras se está ejecutando.

Pablo estableció una primera implementación de las entidades de los cadáveres. También se encargó de la primera iteración de las interacciones de los cadáveres con las criaturas cuando éste sistema estuvo implementado, y más tarde Daniel Cortijo se encargó de la refactorización de estas interacciones. Respecto al sistema de interacciones, se encargó de la implementación del mismo junto con Daniel Cortijo, y de una posterior adaptación que dió lugar al sistema final de interacciones.

Más tarde, se encargó junto con Eloy Moreno de la primera iteración de la máquina de estados que definiría el comportamiento de las criaturas. Con Eloy y con Daniel Cortijo

también se encargó de la implementación de los primeros estados. Pablo se encargó más tarde de la refactorización de la máquina de estados, que eliminaba la librería auxiliar que se estaba usando y se adaptaba de una mejor forma a los estados jerárquicos y al sistema de coste de acciones ideados para las criaturas.

Realizó la refactorización de la memoria de las criaturas con Daniel González y con Andrés de la Cuesta, para llegar a la mente final de las criaturas. También se encargó de arreglar fallos menores sobre la misma y, junto a Andrés, realizó las varias iteraciones en mente y estados de la criatura para lograr crear una acción basada en la exploración de recursos que funcionara correctamente, lo cual resultó complejo y no llegó a funcionar como se esperaba. Posteriormente, tras la adaptación por Georgi Mednikov del algoritmo de *A** para utilizar regiones, se encargó de la refactorización final del susodicho estado *Explore*, que hacía uso de estas.

Pablo también se encargó, aunque en menor medida respecto a otros compañeros, de arreglar y corregir errores derivados de otros estados y transiciones de la máquina de estados de las criaturas.

Pablo se encargó de la refactorización de la gestión de entidades, en la que se optó por usar identificadores (IDs) para cada entidad de la simulación, para evitar comportamientos indeseados relacionados con el sistema de referencias de C#, ya que surgían errores basados en la permanencia de información que debía ser eliminada.

Pablo se encargó de la implementación del proyecto de telemetría, salvo de gran parte de los eventos. Esto consistió en la creación de las principales estructuras que componen el sistema de telemetría: el *Tracker*, el sistema de persistencia y el sistema de serialización. También realizó la integración del proyecto en la simulación, junto con Daniel Cortijo y Andrés de la Cuesta, para lanzar eventos en los momentos que se requirieron durante la ejecución. Tras la recogida de eventos en las ejecuciones, se encargó de la interpretación de las trazas generadas, mediante Python. Para ello creó el script de Python e implementó el código pertinente para la obtención y filtrado de los datos obtenidos de las trazas, que luego se usaron para la representación visual de los datos obtenidos, junto a Daniel González y Georgi Mednikov. También, junto con Daniel González, se encargó de trasladar lo conseguido desde el script de python al *JupyterNotebook* que se utilizó para facilitar el análisis de los datos.

Pablo, como el resto del equipo, ayudó activamente a realizar la evaluación del sistema con usuarios. Se encargó de dar instrucciones a los probadores en parte de las pruebas de usuario y de la posterior entrevista a los mismos.

Respecto al proyecto de Unity, se encargó de la estructuración del proyecto de Unity para hacer uso de la librería de simulación; además de la implementación de la funcionalidad necesaria para representar simulación en Unity, junto con el resto del equipo.

Ésto supone la integración de la mayor parte de los sistemas que interpretan la información interna de la simulación para luego conseguir la representación de la misma. Entre estos sistemas comprenden el manager general de la simulación, que se encarga de crear y ejecutar la simulación, junto con el resto de managers; el manager de

criaturas, que se encarga de interpretar la información interna de la simulación para crear, destruir y actualizar los objetos de Unity que representan las mismas; y el manager de cadáveres, que se encarga de interpretar la información interna de la simulación para crear y destruir los objetos de Unity que representan los mismos.

Por último, Pablo contribuyó a la memoria en distintos apartados, principalmente los relacionados con la ejecución del programa y el ejemplo de uso en Unity. Además, supervisó varios apartados, los relacionados principalmente con la parte que ha realizado en el proyecto.

La contribución de Pablo fue constante durante todo el proyecto. Participó activamente en la resolución de errores menores o tareas pendientes que fueron surgiendo durante el desarrollo, prestando y recibiendo ayuda por parte de todo el equipo.

Summary of the study in English

10. Introduction

This project aims to achieve the creation and documentation of the development of a procedural generator of criatures, which also simulates an environment given a certain terrain, logically establishing a virtual world containing flora that bases the existence of the fauna. Given this context, after some time of simulating, new creatures will have appeared, adapted to survive, able to live in harmony not only with their environment but also with themselves, thus creating a balanced ecosystem. Subsequently, these creatures, exported as data, can be imported and used as the user wishes.

Procedural generation is the process of creating content based on an algorithm instead of direct human invention. It is a tool frequently used in computer programs with a series of characteristics that allow its use due to the advantages this process presents, which will be explained shortly. Procedural generation is particularly notable in the videogame industry, owing to the importance of content in a title. Furthermore, replayability is heavily valued, in other words, it is considered important that a game can entertain in different ways along many playthroughs. This is especially true within certain genres, such as roguelikes, because its premise is one of self contained and unique games, no matter how many are played. As one can suppose, an automated generation is superior to manual labour when it comes to speed and quantity of resources created, which is why procedural generation is held in high regard among developers.

This tool presents a great amount of advantages (Short & Adams, 2017a) and the most relevant are going to be discussed next. Firstly, the correct use of procedural generation means a reduction in development time due to the automatic generation of content. It is also replaceable, precisely due to its efficiency. If a given result is not satisfactory, new ones can be produced until one meets the standards. The variety this process generates in turn generates replayability because the content can be created ipso facto among a great amount of possibilities, therefore remaining unpredictable for longer. Furthermore, a procedural generator is reusable in other projects or content fields as well as consistent. This is a consequence of applying an algorithm; the rules established in one cannot be broken when used. Results are notably realistic when used to simulate real elements, forgive the redundancy, precisely because of the last point made, plus it can generate content in big scales, such as galaxies, if desired. Lastly, procedural generation can help surpass technical limitations, for example, through the avoidance of previously saving content by creating it while running a program.

However, this process presents a series of challenges and inconveniences which usually limit procedural generation to videogames and simulations (Short & Adams, 2017a): the

generation is automated, for better and worse, so the resulting content can be deemed rushed or unpolished by Quality Assurance. Ironically, even though one advantage it has is cutting the developing time, it can actually increase it if an effective algorithm proves elusive, thus requiring a great amount of modifications and even so it might not be good enough. On the other hand, funneling too many resources into a procedural generator can create deficits in other areas of a project, leading to an incomplete experience so matter how polished the generator is. Likewise, too simple of a generator creates close to random content, which is equally as unsatisfactory. Lastly, this process does not fit well in projects with a very specific intent, such as graphic novels particularly focused on the story or artistic direction, or competitive games, which need to be carefully created and balanced.

In short, procedural generators are a situational tool, but very valuable under the right circumstances. It can generate anything from music, levels or characters. Within the previously mentioned videogame industry, the title that best exemplifies this concept is No Man's Sky, whose premise consists of exploring a practically endless galaxy, with countless planets, each with its own fauna, flora and unique characteristics, all of which are created making use of procedural generation. This way, it can be ensured that there are not two people who have experienced exactly the same game.

10.1. Motivation

This project pretends to encompass the procedural generation of a specific type of content, NPCs or Non Playable Characters, which is to say entities the player has no control over.

This field in and of itself is not exploited, since there are, as far as we could find, no such type of generator for anyone to use, whether freely or behind payment. This is a consequence of each project having its own artistic direction, as well as its characters their own characteristic traits. In other words, it generally is pointless to make a generic generator of characters because each developer has their own necessities. It is precisely because of this that this study does not aim to create characters per se, but animals. The fundamental difference is that human characters need to present a personality through their design, whereas creatures inhabiting a world only need to enrich it with detail. It is precisely due to this that these creatures are made to evolve in a terrain with certain properties; to give realism and credibility to it, and in the act, to themselves. Furthermore, the generator is based on data, which means that the results can be interpreted, for example physically, however one wants.

In short, this study aims to program a generator of a type that does not exist generically so that it can be used for a variety of purposes, creating an example of a viable generator in a plurality of projects by capitalizing in the animal's role in the world, creating realistic creatures while giving the user the capability to do whatever they want with the resulting data. Its most notable use of this would be in game developing, being possibly used in games with a similar theme to the previously mentioned No Man's Sky.

10.2. Hypothesis

The basic hypothesis presented in this project is that, given an initial terrain, whether it is given by the user or generated by the program, a world can be simulated inhabited by some initial creatures with their own chromosome to subsequently evolve them in said simulation. Specifically, it is desired to demonstrate and exemplify the design and creation of a procedural generator that creates and simulates the aforementioned elements in order to export the resulting information into data structures to be used in external projects. At the same time, it is desired that this generation is viewed as a viable tool when it comes to the development of content, and that users as well as developers see the potential in a process such as the previously described. It is particularly designed to be used in videogames, as a way to fill a virtual world with the developed creatures so that a realistic and coherent simulation can be done, even though the animals are in a way "alien" to this world.

As a consequence, the established hypothesis dictates that the resulting system will allow the generation of evolved creatures adapted to their environment. These will be able to be imported into and interpreted in other projects, such as simulations and videogames, and will have credible characteristics. This way, users that interact with said project will perceive plausible animals that provide their environment with realism and unity.

Said hypothesis implies the translation of factors such as animal behaviors or natural selection into a simulated environment, as well as the necessary genetic functions of crossover and mutation for the creature's reproduction, leading to new generations with modifications in relation to the previous ones, thus allowing for adaptation.

10.3. Objectives

This project's objectives are the following:

- The generation of a physical world with height, humidity and temperature maps, along with flora according to the terrain's properties.
- Condensation and reproduction of the features and characteristics that define an animal.
- Representation of said characteristics in a structure, a chromosome, based on evolutionary computing, along with its functions of crossover and mutation, as to simulate reproduction.
- Logical navigation of the criatures through the terrain using the A star algorithm with custom heuristics.
- Implementation of an interaction system between a creature and the terrain, flora and other entities in the world.
- Simulation of logical behaviors for the creatures based on fulfilling their basic needs, simulating their vital cycles and thus applying natural selection.

- Implementation of a system that simulates the passing of time in the world, creating a day/night cycle and a turn system for the creatures actions.
- Exportation of the creatures to a generic format based on data to facilitate their use. Exportation of the simulation to a library.
- Exemplary demonstration of a possible interpretation of the resulting data to an external engine, as well as the exportation of the simulation to said engine.

10.4. Methodology

The first point to discuss is the environment on which the program is going to be developed. This includes both the programming language as well as the programming environment itself. Four languages were considered for this study: C++, C#, Java and Python. The first option is the most efficient when it comes to speed. However, it comes with a shortcoming when it comes to developing time, since it is the lowest level language among the four options. Python could potentially reduce the time needed to complete an execution of the program by optimizing the mathematical operations through the use of Numpy, an extremely efficient library, but the language in and of itself is slow by nature, as well as somewhat alien to the members of the group. Lastly, C# and Java are both object oriented languages, with the notable distinction that the developing group has more experience and a deeper understanding of C#. It has a garbage collector and a large amount of potentially useful libraries, which greatly reduces development time. Furthermore, it is the language that is used for scripting in Unity, the engine that will be used for the demonstrating prototype.

Knowing this, C# was chosen from among the candidates, since it is somewhat in the middle between running and development speed, plus it is the language used in Unity, the engine used to graphically represent the simulation, so certain possible interpretation steps between languages can be skipped. Once the language is chosen, the development environment is easily chosen, in this case Visual Studio, because all members of the group have access to it and are familiar with its inner workings.

C# could present, however, a limitation in development related to the creatures. Due to how references work in this language, when a creature would have to be eliminated from the simulation, it would not as long as there is another reference somewhere, for example in the case that one creature remembers another. However, a dead creature should at all levels be erased. C# provides ways to deal with problems of this nature, but this seems like a cumbersome solution, halfway between C++ and C# work. This implies the implementation of an ID system for the entities in the world, so that when an entity wants to acquire information about another it first must try to use the latter's ID, and in the case that it has perished the data would not be provided.

The first element necessary to begin the fulfilling of the previously described objectives is the generation of the world. This commences with some basic data that describes a map of heights and water masses through a bidimensional matrix. It is intended to be a possibility for the user to give this map to the program is they see so fit, and if none is

provided a random one is generated using Perlin noise and a certain amount of configurable parameters to customize the result of the generation which, as many other aspects in this project, the user will be able to modify. Once the map of height and masses of water is obtained, it will be used to calculate semi-randomly maps of humidity and temperature that will be used as a last step in the creation of a basic flora, which is the objective of many basic interactions of the creatures.

Once a terrain already exists with its properties as well as a flora with which the creatures can interactuate, the next step is to generate unique genetics to each species. This is represented through a chromosome, a structure that contains all the information that defines how a creature is and what characteristics it has. In the first generation of each species, random chromosomes are created for them, since there are no progenitors on which to base the genetics of these individuals. These creatures are the primals that will evolve to create new species that will adapt to their environment progressively better.

Then, as many individuals belonging to as many species as the user has established will be distributed around the map. All individuals of the same species appear together at first so that they have a common starting point and are more likely to find each other if needed, and species will appear somewhat distributed around the world in order to use most of the generated terrain and avoid concentration of species in specific locations, which can lead to almost immediate extinctions. The first generation of each species shares the same chromosome, representing a common point in the phylogenetic tree of the supposed previous evolution that led them to their present state. As is reasonable, the sexes of the creatures are randomly assinged so that half of the population tends to be male and the other half female.

While the simulation is running, a name will be assigned to each species in order to identify them and preserve a legible history, keeping track of the different existing and extinct species and progressively creating their respective phylogenetic trees. This way, the user will be given the possibility of knowing the evolutionary history of the creatures, in case it is of any use.

When the simulation ends, a stream of data will be exported: the generated world's information, in case one wants the information of the environment used in the simulation; the information of the creatures once they have gone through the evolutive process that can later be interpreted as one deems fit for the project in question, and the aforementioned phylogenetic trees.

Lastly, in order to exemplify a possible use of the resulting data and display live the simulation on which this project is based, the data and the program's code will be imported to an external engine, Unity, to recreate both the world and the creatures, representing several of their attributes in their physic form and running all this in real time. This way both the simulation in and of itself as well as how the data can be interpreted as one desires are demonstrated. The version of Unity that will be used is Unity 2020.3.29.f1. In order to fulfill the necessity of tridimensional models for said recreation of the simulation, Blender, a modeling program, will be utilized. Using it, the

models that make up the representation of the creatures will be designed by the developing team.

Pivotal Tracker will be used to create and organize the tasks that make up the project, at the same time that allows for awareness of the developing speed. Semanal iterations will be carried out, after which the validity of the stories will be judged and new ones will be chosen for the next iteration. The duration of the stories is assigned following a point system, which represents an estimation of the time cost of said task.

Github is going to be used as a version control system through a public repository containing both the simulation project as well as the adaptation to Unity. The repository of the study can be accessed through the following link: https://github.com/georgimednikov/TFG2122.

The developing process has been planned based on the timeline in Table 10.1, knowing that certain periods of time the team would have less time to develop due to academic reasons.

Date	Project progress
May	Initial conception of the idea, creation of a work group and search for a project director.
November	Study about the state of the art and the branches of knowledge related to the project.
December	Definition of the inner workings and detail of the program and the posterior organization of the development in Pivotal Tracker.
January	Preparation for development: Creation of the projects in Visual Studio and Unity using a repository on Github
February	Development beginning: Implementation of the evolutionary programming related to the project and the creation of the world generation. Generate primitive creatures in the world combining these two elements.
March	Creation of a state machine that gives behaviors to the creatures and perfecting of the previously created content as errors or needs arise. Import and export data structures to be utilized in Unity, plus a basic interpretation of them.
April	Perfecting of the creature's simulation in the base project and their representation as well as their actions in Unity. Error fixing and beginning of the memory writing.

May	Tests with users through the representation in Unity. Termination of the memory and fixing of the remaining errors in the code.
June	Publication of the project and defense of the study.

Table 10.1. Chronogram of the project

10.5. Document structure

Chapter 1, as seen before, is a general summary of the project answering the fundamental questions of what, how and why, while at the same time adding context to the idea and briefly explaining not only the program to create, but also this document.

Chapter 2 continues the introduction, talking about the current situation of this type of project in the world of computer science, or more specifically, the videogame industry, with the objective of giving more context to the motivation and the hypothesis previously presented. Before this, however, it is debated about the current concept of what constitutes a living being and what are their fundamental characteristics, to set a foundation for the study.

Chapter 3 contains the bulk of the project, describing the entire process of generating and simulating an inhabited world, and therefore it is going to be described in more depth. It has been ordered imitating the order of events when the program runs, for consistency's sake. As a consequence, in subchapter 3.1 the structure of the project is discussed. After that commences the description of the simulation in subchapter 3.2, which details the creation of the world and flora, describing the variables that modify these, how the user can customize them and how they are used.

Subchapter 3.3 revolves around the creatures from a genetic point of view. The beginning of the section focuses on the evolutionary aspect of the project, discussing the implementation of the genetic information, along with its structure and representation. Later in the same subchapter, it is talked about the interpretation of this data and its translation to statistics and information that can be used in the creature's state machine, along with an explanation of why this system was chosen.

Subchapter 3.4 details the creature's logic and "thought process" during their life cycle, which includes descriptions about how they gain and lose information, navigate the world and attempt to fulfill their needs through a state machine that defines their behavior.

Chapter 4 revolves around the user's interaction with the program: what data can be input and modified to customize the simulation; how the program's execution works and has been perfected, and lastly the resulting data is discussed as well as how it can be exported to be used as the user deems fit. This last concept is exemplified in the

adaptation to Unity, where the steps to visually recreate the world and creatures are described, and the reproduction of the simulation's code live.

Once all the information related to the creation and execution of the program is presented, the study is closed with the following chapters: chapter 5, which explores the tests run in order to evaluate the program's performance; chapter 6, which describes the results obtained and their usability, and lastly chapter 7, in which the validity and contribution of the study to the academic field are discussed.

To conclude, the last chapters, 8 and 9, end the memory summarizing the conclusions of the project, possible expansions and improvements, along with the individual contributions of each member of the group during the development of the study.

11. Conclusions and future work

This is the final chapter of this study. Once all elements previous and related to the development have been presented, the only remaining aspect is to reach conclusions based on the information described so far in this document and discuss possible expansions that may enrich this project in the future.

11.1. Conclusions

This is a study about the creation of a procedural generator of creatures based on data that can be imported and interpreted. An animal is the consequence of a process of adaptation to its environment, and therefore its properties represent to a certain extent the characteristics of the latter. This is a factor that is analyzed and exploited, receiving a terrain with specific characteristics and creating a series of creatures that interact with each other as well as with the world they inhabit, simulating evolution in order to create a believable, realistic and sustainable ecosystem. To achieve this, research has been carried out about artificial intelligence, procedural generation in and of itself, behavioral engineering and evolutionary computing, plus the external engine used to show the data in a real-time, tridimensional environment.

In order to evaluate the proper functioning of this program, two methods have been used: test with users, in which people alien to the project are asked to evaluate the resulting data based on their experience, and a telemetry system based on events, which allows for the recollection of data for its posterior analysis. This way, conclusions that allow for updates and improvements on the project to be reached.

Thus, it is concluded that the contribution of this study is positive because the objectives of the projects have been fulfilled, being the main and most notable the generation of a coherent virtual world with variables for temperature, humidity and altitude, that allow for the generation of a flora and a fauna. More importantly yet, the creatures that make up this fauna successfully interact among themselves and with their environment, describing logical behaviors and acquiring as well as modifying their physical characteristics based on the evolutionary process dictated by natural selection. The simulation works properly and produces valid data for its interpretation in other environments, such as other simulations or videogames. The correct working of the system has been tested and achieved making use of the previously mentioned telemetry system, and the user tests, even though there were not as many as one would hope, still present empirical evidence that supports the idea of the use of this project, although

certain elements need to be improved on, most of which fall under the depuration of the creature's behaviors and the values of the simulation.

11.2. Future work

Given the dimensions of the project, even with a developing team of a considerable size, there is a decent amount of details and aspects that have been left aside during this study. Examples of this are the lack of intensive debugging of the code due to a lack of time, which would not be particularly complicated thanks to the telemetry system; a bigger and better balancing of the functionalities of the different abilities and qualities of the creatures, since it is reasonable to presume that after a certain amount of cases it will start to appear obvious that some are more recurrent than others; the combat system, which can and is designed to be expanded, adding new state effects (currently, poison is the only one), etc; more qualities and abilities, with the objective of representing a larger amount of animal characteristics...

The polishing of the project in and of itself represents a great amount of time and effort to invest in the future so that the simulation reaches the largest amount of efficiency possible when it comes to generating creatures. However, other aspects not related to the current content can be developed, such as expansions of animal life into other mediums. Even though aerial and arboreal movement is represented along with the terrestrial, a lot of focus has been placed on the latter, and the others are mere adaptations. As a consequence, there is not a notable complexity, for example, in the interaction between airborne creatures, since there are none specific behaviors defined. This leads to the idea of creating new "worlds" depending on the medium, completely separating their functioning. Examples of this might be new divisions of airspace based on altitude and genetic properties that dictate how and where creatures can fly, or subterranean species that inhabit the underground.

This last concept, carried to the extreme, can be extrapolated to the creation of an aquatic ecosystem. Simply explained, the current simulation, with its three mediums (aerial, arboreal, and terrestrial), would make up part of the new simulation, and the other would be an aquatic medium with its own divisions of space in which a new evolutionary process would take place. In a similar manner, it would have different temperatures, nutrient-generating entities, such as sponges, and altitudes, which would define fundamental elements of the survival of the creatures. For example, in high altitudes there are organisms that carry out photosynthesis, whereas in the depths there is no natural light and animals feed on residual organic matter that descends from above, layers closer to the sea level. This would clue in new interactions between the two ecosystems, as well as a new terrain generation, creating rivers and new and distinct masses of water and therefore a more unified simulation.

All these ideas are based on translating the functioning of life and real natural ecosystems into a simulation, which are endless sources of possible extensions. After all, representing nature in a virtual environment is a task practically infinite in its complexity.

A different direction in which this project can be expanded is precisely the opposite of what has been discussed up until this point. Instead of trying to recreate natural environments and behaviors, it can be attempted to create new ones without basing them on already existing cases. This can have a notable importance and value, since possible evolutions that have not happened in real life yet might take place in the simulation, as long as this system is implemented correctly and successfully. However, this is precisely the difficulty of this idea: implementing behaviors that not even the developers can imagine. This would require a reconstruction of the state machine and the system of abilities and qualities, but it would be a viable and highly valuable expansion.

Bibliografía

A, S., RAYMOND, & W, J. J. R., JOHN. (2003). *Física volumen 1*. Ediciones Paraninfo, S.A.

Algoritmo de búsqueda A*. (2022). En *Wikipedia, la enciclopedia libre*. https://es.wikipedia.org/w/index.php?title=Algoritmo_de_b%C3%BAsqueda_A*&oldid=140834447

Algoritmo de gráfico de segmento de Bresenham. (2005). https://es.frwiki.wiki/wiki/Algorithme_de_trac%C3%A9_de_segment_de_Bresenham

Andersson, M., & Iwasa, Y. (1995). *Sexual Selection*.

Azulejos Wang. (2021, octubre 29). Wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Azulejos_Wang&oldid=139368967

Barriga, N. A. (2019). A Short Introduction to Procedural Content Generation Algorithms for Videogames. *International Journal on Artificial Intelligence Tools*, *28*(02), 1930001. https://doi.org/10.1142/S0218213019300011

Binary space partitioning. (2022, abril 3). Wikipedia. https://en.wikipedia.org/w/index.php?title=Binary_space_partitioning&oldid=1080819490

Black, J. L. (2014). *Brief history and future of animal simulation models for science and application*.

Box blur. (2021, diciembre 6). Wikipedia. https://en.wikipedia.org/w/index.php?title=Box_blur&oldid=1058880555

Bresenham, J. E. (1965). *Algorithm for computer control of a digital plotter*. https://web.archive.org/web/20080528040104/http://www.research.ibm.com/journal/sj/041/ibmsjIVRIC.pdf

Brezzi, F., F. Chan, T., & Griebel, M. (1959). *Numerische Mathematik l*.

Camino aleatorio. (2021, abril 24). Wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Camino_aleatorio&oldid=135029342

Clarke, A., & Fraser, K. P. P. (2004). Why does metabolism scale with temperature? *Functional Ecology*, *18*(2), 243-251. https://doi.org/10.1111/j.0269-8463.2004.00841.x

Coello Coello, C. A. (2019). *Computación Evolutiva*.

Cui, X., & Shi, H. (2010). *A*-based Pathfinding in Modern Computer Games. 11*.

Darwin, C. (1988). *El Origen de las Especies* (Spanish Edition). S.L.U. Espasa Libros.

Diamond-square algorithm. (2021, noviembre 30). Wikipedia. https://en.wikipedia.org/w/index.php?title=Diamond-square_algorithm&oldid=1057847114

dotnet-bot. (s. f.). *Random Clase (System)*. Recuperado 3 de abril de 2022, de https://docs.microsoft.com/es-es/dotnet/api/system.random

Dunhill, J. (2022). *First Fully Complete Human Genome Has Been Published After 20 Years*. https://www.iflscience.com/health-and-medicine/first-fully-complete-human-genome-has-been-published-after-20-years/

El Juego de la Vida de Conway. (2022). En *Wikipedia, la enciclopedia libre*. https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life

Figure 1. Illustration of Dijkstra's algorithm. (2019). ResearchGate. https://www.researchgate.net/figure/Illustration-of-Dijkstras-algorithm_fig1_331484960

Goldschmidt, R. (2009). *The Material Basis of Evolution* (Illustrated Edition). Yale University Press.

Gould, S. J., & Eldredge, N. (1993). *Punctuated equilibrium comes of age*.

Gwiazda, T. D. (2006). *Genetic Algortihms Reference* (First Edition). Tomaszgwiazda E-Books.

Holdridge life zones. (2022, abril 19). Wikipedia. https://en.wikipedia.org/w/index.php?title=Holdridge_life_zones&oldid=1083567853

Koshland, D. E. (2002). *Los Siete Pilares de la Vida*.

Kosko, B., & Isaka, S. (1993). Fuzzy Logic. *Scientific American*, *269*(1), 76-81.

Langton, C. G. (1995). *Artificial Life*.

Libnoise: Tutorial 4: Modifying the parameters of the noise module. (s. f.). Recuperado 16 de abril de 2022, de http://libnoise.sourceforge.net/tutorials/tutorial4.html

Marbate, P., & Gupta, R. (2013). *Fortune's Method: An Efficient Method For Voronoi Diagram Construction*. *12*, 7.

Mayr, E. (1996). *What Is A Species, and What Is Not?*

Obitko, M. (1998). *XI. Crossover and Mutation*.

Pavlov, I. P. (1929). *Los reflejos condicionados* (First Edition). Ediciones Morata.

Pedersen, L. D., Sørensen, A. C., Henryon, M., Ansari-Mahyari, S., & Berg, P. (2009). ADAM: A computer program to simulate selective breeding schemes for animals. *Livestock Science*, *121*(2), 343-344. https://doi.org/10.1016/j.livsci.2008.06.028

Pierre. (2019, junio 23). *Vagabond – Dungeon and Cave Generation – Part 1*. Pvigier's Blog. https://pvigier.github.io/2019/06/23/vagabond-dungeon-cave-generation.html

Rajput, R. K. (1996). *Engineering Thermodynamics* (Third Edition). Laxmi Publications.

Romero Dopico, M. (s. f.). *El Juego de la Vida*.

Russell, S. J., Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4.ª ed.). Pearson.

Sagan, C. (1970). *Definitions of Life*.

Short, T. X., & Adams, T. (2017a). *Procedural Generation in Game Design* (First Edition). CRC Press.

Short, T. X., & Adams, T. (2017b). *Procedural Generation in Game Design* (First Edition). CRC Press.

Sistema-L. (2022, febrero 20). Wikipedia, la enciclopedia libre. https://es.wikipedia.org/w/index.php?title=Sistema-L&oldid=141801563

Theiβen, G. (2005). *The proper place of hopeful monsters in evolutionary biology*.

Thompson, A. (2016). *Scientists Have Made Transistors Smaller Than We Thought Possible*. https://www.popularmechanics.com/technology/a23353/1nm-transistor-gate/

Toro, C., & Buriak, J. M. (2018). Wired Chemistry: The Synthetic Chemistry that Made Silver Nanowire-Based Electrodes Possible. *Chemistry of Materials*, *30*(15), 4875-4876. https://doi.org/10.1021/acs.chemmater.8b02612

Unity. (2022). *Unity—Inner workings of the Navigation System*. https://docs.unity3d.com/Manual/nav-InnerWorkings.html

van der Linden, R., Lopes, R., & Bidarra, R. (2014). Procedural Generation of Dungeons. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(1), 78-89. https://doi.org/10.1109/TCIAIG.2013.2290371

White et al. (2021, septiembre 30). *Evolución 101! - Understanding Evolution*. https://evolution.berkeley.edu/bienvenido-a-la-evolucion-101/

Apéndice

A. Archivos


```
[
  [
    {
      "height": 0.5,
      "humidity": 0.5,
      "temperature": 0.5,
      "flora": 0.29730177875068026,
      "plant": {
        "type": 0,
        "eaten": false,
        "x": 0,
        "y": 0,
        "ID": 0,
        "maxHp": 10.0,
        "curHp": 10.0
      },
      "isWater": false,
      "regionId": 0
    },
    {
      "height": 0.5,
      "humidity": 0.5049999356269836,
```

```
      "temperature": 0.49999943375587463,
      "flora": 0.3025239866131627,
      "plant": null,
      "isWater": false,
      "regionId": 0
    },
    (...)
  ],
  (...)
]
```

Figura A.1. Objetos dentro del archivo de entrada y salida *World.json* que representan las casillas del mapa

```
[
  {
    "spawnPoint": {
      "X": 24.0,
      "Y": 15.0
    },
    "links": {
      "1": [
        {
          "X": 15.0,
          "Y": 23.0
        },
        {
          "X": 16.0,
          "Y": 23.0
        },
        (...)
```

```
      ],
      (...)
    }
  },
  (...)
]
```

Figura A.2. Objeto dentro del archivo de entrada y salida *RegionMap.json*

```
{
 [
  0.1, 0.2,
  0.1, 0.2
 ]
}
```

Figura A.3. Archivo de entrada *HeightMap.json* de tamaño 2x2

```
[
    {
        "feature":"Strength",
        "maxValue":50,
        "relations":[]
    },
    {
        "feature":"Size",
        "maxValue":200,
        "relations":
        [
```

```
            {
                "feature":"Constitution",
                "percentage":0.4
            },
            {
                "feature":"Strength",
                "percentage":0.25
            }
        ]
    },
    (...)
]
```

Figura A.4. Extracto de un archivo de entrada *Chromosome.json*

```
[
  {
    "Item1": "Arboreal",
    "Item2": 0.4
  },
  {
    "Item1": "Wings",
    "Item2": 0.4
  },
  {
    "Item1": "Venomous",
    "Item2": 0.4
  },
  {
    "Item1": "NightVision",
    "Item2": 0.4
```

```
  },
  {
    "Item1": "Horns",
    "Item2": 0.4
  },
  {
    "Item1": "Mimic",
    "Item2": 0.4
  },
  {
    "Item1": "Upright",
    "Item2": 0.4
  },
  {
    "Item1": "Thorns",
    "Item2": 0.4
  },
  {
    "Item1": "Scavenger",
    "Item2": 0.4
  },
  {
    "Item1": "Hair",
    "Item2": 0.4
  },
  {
    "Item1": "Paternity",
    "Item2": 0.4
  }
]
```


Figura A.5. Archivo de entrada *AbilityUnlock.json*

```
{
  "ticksPerHour": 50,
  "hoursPerDay": 24,
  "daysPerYear": 365,
  "morningStart": 6.5,
  "nightStart": 20.0,
  "grassHp": 10,
  "bushHp": 20,
  "eTreeHp": 50,
  "grassNutritionalValue": 2.8,
  "bushNutritionValue": 6.4,
  "eTreeNutritionalValue": 13.2,
  "grassHoursTillGrowth": 24.0,
  "bushHoursTillGrowth": 120.0,
  "eTreeHoursTillGrowth": 360.0,
  "abilityUnlockPercentage": 0.4,
  "minHealth": 10,
  "healthGainMultiplier": 2,
  "healthRegeneration": 0.1,
  "maxLimbs": 10,
  "resourceAmount": 100,
  "minPerception": 5,
  "maxPerception": 10,
  "minLifeSpan": 0.5,
  "exhaustToSleepRatio": 3.0,
  "hoursTilExhaustion": 28.0,
  "perceptionWithoutNightVision": 0.6,
  "minPerceptionWithNightVision": 0.7,
  "minMobilityMedium": 0.6,
```

```
"mobilityPenalty": 0.7,
"yearsBetweenHeats": 0.05,
"maxChildNumber": 5,
"ticksToReproduce": 20,
"maxSpeed": 1.5,
"hornIntimidationMultiplier": 1.5,
"hairTemperatureMultiplier": 0.2,
"restRegenerationThreshold": 0.7,
"energyRegenerationThreshold": 0.85,
"hydrationRegenerationThreshold": 0.85,
"regenerationRate": 0.01,
"hoursTilStarvation": 124.0,
"thirstToHungerRatio": 3.0,
"maxTemperatureDifference": 0.2,
"minHealthTemperatureDamage": 0.01,
"maxHealthTemperatureDamage": 0.02,
"venomDamageMultiplier": 0.25,
"omnivorousNutritionMultiplier": 0.7,
"newbornStatMultiplier": 0.33,
"adulthoodThreshold": 0.2,
"tiredThreshold": 0.45,
"exhaustThreshold": 0.12,
"hungryThreshold": 0.55,
"veryHungryThreshold": 0.15,
"thirstyThreshold": 0.6,
"veryThirstyThreshold": 0.2,
"treeMovementPenalty": 0.7,
"knowledgeTickMultiplier": 500,
"aggressivenessToRadiusMultiplier": 0.33,
"maxResourcesRemembered": 5,
"maxPositionsRemembered": 10,
"baseActionCost": 1000,
```

```
  "venomCostMultiplier": 100,
  "chaseCostMultiplier": 0.4,
  "fleeingCostMultiplier": 0.75,
  "drinkingCostMultiplier": 10,
  "drinkingMultiplier": 10,
  "eatingCostMultiplier": 10,
  "sleepingCostMultiplier": 10,
  "mutationChance": 0.1,
  "adjacentLength": 1,
  "actionPerceptionPercentage": 0.75,
  "sleepingExpenseReduction": 0.1,
  "fleeingTransitionMultiplier": 4,
  "stopEatingTransitionEnergyPercentage": 1.0,
  "combatTransitionHealthThresholdMultiplier": 50.0,
  "maxMenaceIntimidationMultiplierBasedOnMissingHealth": 2.0,
  "experienceMaxAggresivenessMultiplier": 0.2,
  "maxDistanceToStartFollowParent": 10,
  "maxDistanceToStopFollowParent": 3,
  "rotStartMultiplier": 0.01,
  "corpseNutritionPointsMultiplier": 80.0,
  "percentageSimilaritySpecies": 0.9
}
```


Figura A.6. Archivo de entrada *UniverseParameters.json*


```
{
  "name": "Talagae Gogage",
  "stats": {
    "Gender": 0,
    "Diet": 0,
```

```
    "Scavenger": 0.0,
    "MaxHealth": 70.0,
    "CurrHealth": 70.0,
    "Damage": 16,
    "Armor": 11,
    "Perforation": 5,
    "Venom": 5.0,
    "Counter": 0.0,
    "AerialSpeed": -1,
    "ArborealSpeed": 66,
    "GroundSpeed": 46,
    "AirReach": false,
    "TreeReach": true,
    "MaxEnergy": 100.0,
    "CurrEnergy": 98.51227,
    "EnergyExpense": 0.014879032,
    "MaxHydration": 100.0,
    "CurrHydration": 95.53604,
    "HydrationExpense": 0.044637095,
    "MaxRest": 100.0,
    "CurrRest": 94.0712,
    "RestRecovery": 0.17785715,
    "RestExpense": 0.059285715,
    "Camouflage": 14,
    "Aggressiveness": 10,
    "Intimidation": 0,
    "Perception": 7,
    "MaxPerception": 10,
    "Size": 125,
    "LifeSpan": 4599000,
    "CurrAge": 1149850,
    "Limbs": 3,
```

```
    "Metabolism": 68,
    "MinTemperature": 0.16666666666666666,
    "MaxTemperature": 0.2777777777777778,
    "IdealTemperature": 0.2222222222222222,
    "Hair": false,
    "Knowledge": 19,
    "Paternity": 6,
    "HealthRegeneration": 0.1,
    "MaxSpeed": 1.5,
    "TimeBetweenHeats": 4380,
    "InHeat": false,
    "Upright": false
  }
}
```


Figura A.7. Achivo de salida *Species_X.json*


```
├──Simahoaguni Ti    First born tick: 0    Last alive tick: 1746
├──Fuli Fasesose    First born tick: 0    Last alive tick: 238
├──Seasu Toniutusieko    First born tick: 0    Last alive tick: 87986
├──Hularihati Loewio    First born tick: 0    Last alive tick: 309
├──Poima Hepetureyu    First born tick: 0    Last alive tick: 1933
├──Sasu Toheheso    First born tick: 0    Last alive tick: 881
├──Nesahaa Lusuli    First born tick: 0    Last alive tick: 1490
├──Lomimamegaa Rema    First born tick: 0    Last alive tick: 298
├──Hakisuse Seguiwa    First born tick: 0    Last alive tick: 128
├──Peesigo Remuleupe    First born tick: 0    Last alive tick: 2451
├──Lumahileguu Lomi    First born tick: 0    Last alive tick: 553
├──Pifida Nalaesuna    First born tick: 0    Last alive tick: 262800
│    └──Peme Negasaumate    First born tick: 262354    Last alive tick: 262800
```

```
|     └──Pemegu Hesaisua    First born tick: 262354    Last alive tick: 262800
|     └──Fefanuga Meusina    First born tick: 261857    Last alive tick: 262800
|     └──Feamona Nalutane    First born tick: 261857    Last alive tick: 262800
|     └──Pelinae Humeheute    First born tick: 261857    Last alive tick: 262800
|     └──Fuma Gunamisui    First born tick: 261857    Last alive tick: 262800
|     └──Falu Nohesata    First born tick: 260584    Last alive tick: 262800
|     └──Fuminogu Lagu    First born tick: 260584    Last alive tick: 262800
|     └──Femugihelu Susu    First born tick: 252872    Last alive tick: 254288
|     └──Famonegie Reaho    First born tick: 252872    Last alive tick: 256957
|     └──Palinoahumi Sela    First born tick: 252872    Last alive tick: 255822
|     └──Pielati Nalilese    First born tick: 249856    Last alive tick: 253455
|     └──Falatee Nemiseila    First born tick: 242992    Last alive tick: 244489
|     └──Mama Nigesase    First born tick: 242315    Last alive tick: 246221
|     └──Pulege Naelulogi    First born tick: 240410    Last alive tick: 262800
|     └──Maemi Nohatumaugi    First born tick: 240410    Last alive tick: 242513
|     └──Piloota Tisoteitu    First born tick: 233127    Last alive tick: 233389
|     └──Fulegiegeto Toto    First born tick: 233127    Last alive tick: 236888
|     └──Pusuta Gagileteu    First born tick: 229031    Last alive tick: 262800
|     └──Fealeogugoseo Siteo    First born tick: 229031    Last alive tick: 232332
|     └──Pomehoheale Tiu    First born tick: 224949    Last alive tick: 226515
|     └──Momii Ganafaga    First born tick: 224949    Last alive tick: 228035
|     └──Famonuugui Mahi    First born tick: 224230    Last alive tick: 226848
|     └──Pasiaho Helietehee    First born tick: 224230    Last alive tick: 228341
|     └──Fimani Gelolasa    First born tick: 224230    Last alive tick: 262800
|     └──Faluonugeemu Ha    First born tick: 224230    Last alive tick: 228110
|     └──Falaunide Soato    First born tick: 221299    Last alive tick: 262800
|     └──Famaigi Gafataa    First born tick: 219181    Last alive tick: 221036
|     └──Mamanenufee He    First born tick: 219112    Last alive tick: 219331
|     └──Fumonugoli Su    First born tick: 213966    Last alive tick: 217936
|     └──Palagogu Leleo    First born tick: 213966    Last alive tick: 215321
|     └──Femenuhe Mututa    First born tick: 213966    Last alive tick: 218321
|     └──Piileni Tatisahu    First born tick: 213966    Last alive tick: 217884
```

```
|     └──Fimuegau Niimuti    First born tick: 212260    Last alive tick: 215866
|     └──Fimo Nitelitane    First born tick: 212260    Last alive tick: 229739
|     └──Pomo Nogemaoluhi    First born tick: 210278    Last alive tick: 212027
|     └──Mominanuseo Se    First born tick: 210278    Last alive tick: 262800
|     └──Pumahehi Tasole    First born tick: 210278    Last alive tick: 235521
|     └──Memi Gihotusanai    First born tick: 210278    Last alive tick: 213619
|     └──Fimetenumau Sata    First born tick: 210009    Last alive tick: 211802
|     └──Fimuanedolue Ga    First born tick: 210009    Last alive tick: 214571
|     └──Pela Neguletoige    First born tick: 204702    Last alive tick: 206715
|     └──Paleniegiili Laho    First born tick: 204702    Last alive tick: 252605
|     └──Falaahegisa Sesi    First born tick: 204702    Last alive tick: 208668
|     └──Mamunegasi Ti    First born tick: 201148    Last alive tick: 206270
|     └──Femi Nunamisuti    First born tick: 201148    Last alive tick: 262800
|     └──Fiefo Nenamusego    First born tick: 201148    Last alive tick: 262800
|     └──Faumono Gehafesi    First born tick: 199546    Last alive tick: 203093
|     └──Faminudisa Sola    First born tick: 199546    Last alive tick: 219231
|     |     └──Fumei Goduumete    First born tick: 219112    Last alive tick: 219757
|     └──Peleiheuga Lusute    First born tick: 199546    Last alive tick: 202445
|     └──Foomu Nihasitite    First born tick: 195602    Last alive tick: 236740
|     |     └──Femiga Noulosia    First born tick: 229031    Last alive tick: 232036
|     |     └──Pailugogasue Lete    First born tick: 229031    Last alive tick: 230026
|     └──Femiu Gigelalete    First born tick: 195602    Last alive tick: 197814
|     └──Muamanotiso Seola    First born tick: 195602    Last alive tick: 199651
|     └──Famo Nahilaahu    First born tick: 195602    Last alive tick: 199192
|     └──Paluehehолu Lege    First born tick: 192127    Last alive tick: 198617
|     └──Mami Neudaosutuni    First born tick: 192127    Last alive tick: 236315
|     |     └──Pelasinusae Faha    First born tick: 234165    Last alive tick: 235440
|     |     └──Famatu Nilutaece    First born tick: 221299    Last alive tick: 221575
|     |     └──Mamaneine Mutoigi    First born tick: 212260    Last alive tick: 212504
|     └──Folotenia Soimietee    First born tick: 192127    Last alive tick: 195613
|     └──Femage Nelula    First born tick: 190529    Last alive tick: 195462
|     └──Palaati Nisimite    First born tick: 190529    Last alive tick: 262800
```

```
|     └──Pialehoha Liusumi    First born tick: 186090    Last alive tick: 186340
|     └──Miemone Noitututu    First born tick: 182267    Last alive tick: 205161
|     └──Famoguogife Tu    First born tick: 181419    Last alive tick: 183678
|     └──Fefi Gadofiosa    First born tick: 181419    Last alive tick: 262800
|     |     └──Foleho Galimege    First born tick: 251906    Last alive tick: 253423
|     |     └──Faminenulo Sela    First born tick: 234131    Last alive tick: 235016
|     └──Fimenu Gososetia    First born tick: 176378    Last alive tick: 178507
|     └──Famania Gisasete    First born tick: 176378    Last alive tick: 178091
|     └──Fama Nageusimego    First born tick: 176378    Last alive tick: 176461
|     └──Pesetugigeo Letia    First born tick: 176378    Last alive tick: 179709
|     └──Pumunugisa Letia    First born tick: 173238    Last alive tick: 262800
|     |     └──Fulagagoli Sigo    First born tick: 234165    Last alive tick: 238673
|     └──Famoni Nemati    First born tick: 167349    Last alive tick: 171211
|     └──Fefoga Galesuco    First born tick: 163017    Last alive tick: 167158
├──Jupaomago Tejabao    First born tick: 0    Last alive tick: 164250
├──Hiusi Getapomawa    First born tick: 0    Last alive tick: 500
├──Femuepa Tomige    First born tick: 0    Last alive tick: 771
├──Tagiliteri Mima    First born tick: 0    Last alive tick: 259
├──Refetufere Ho    First born tick: 0    Last alive tick: 922
├──Saomirimoso Tiota    First born tick: 0    Last alive tick: 185
├──Hatete Sulepehi    First born tick: 0    Last alive tick: 1939
├──Famipimemau To    First born tick: 0    Last alive tick: 671
├──Japaime Gafesu    First born tick: 0    Last alive tick: 32850
├──Gisupote Hagi    First born tick: 0    Last alive tick: 429
├──Rasiogo Pegulago    First born tick: 0    Last alive tick: 188
├──Hetujoo Tefaremu    First born tick: 0    Last alive tick: 500
├──Mupusiriidu Sa    First born tick: 0    Last alive tick: 341
├──Guatesohesa Mege    First born tick: 0    Last alive tick: 1242
├──Kofoimula Pofaati    First born tick: 0    Last alive tick: 2963
├──Nineniiherae Satia    First born tick: 0    Last alive tick: 545
├──Gunusuhasae Gupa    First born tick: 0    Last alive tick: 5828
├──Salu Mototome    First born tick: 0    Last alive tick: 262800
```

Figura A.8. Ejemplo de archivo de salida *Tree.txt*

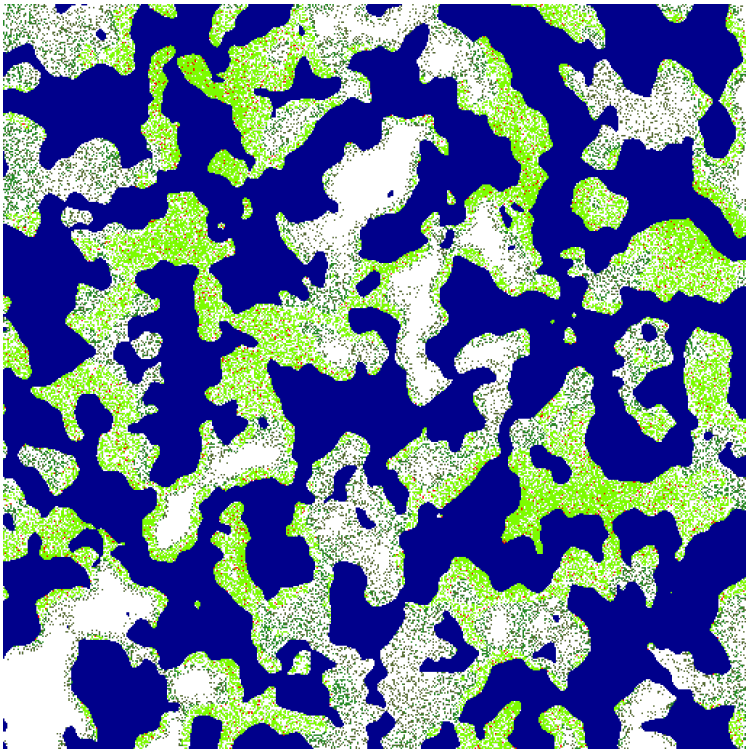


Figura A.9. Mapa de flora y agua sin terreno de depuración (*Flora.png*)

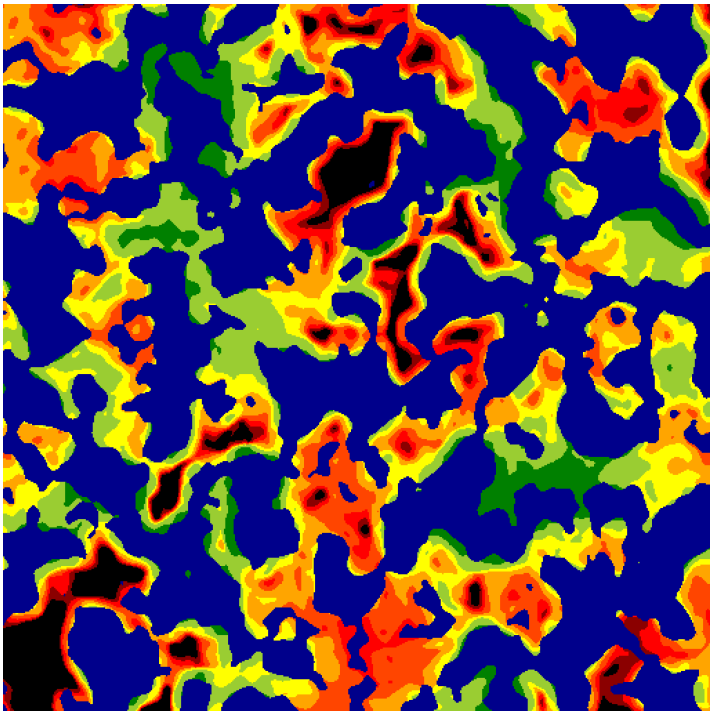


Figura A.10. Mapa de densidad de flora de depuración (*FloraProb.png*)

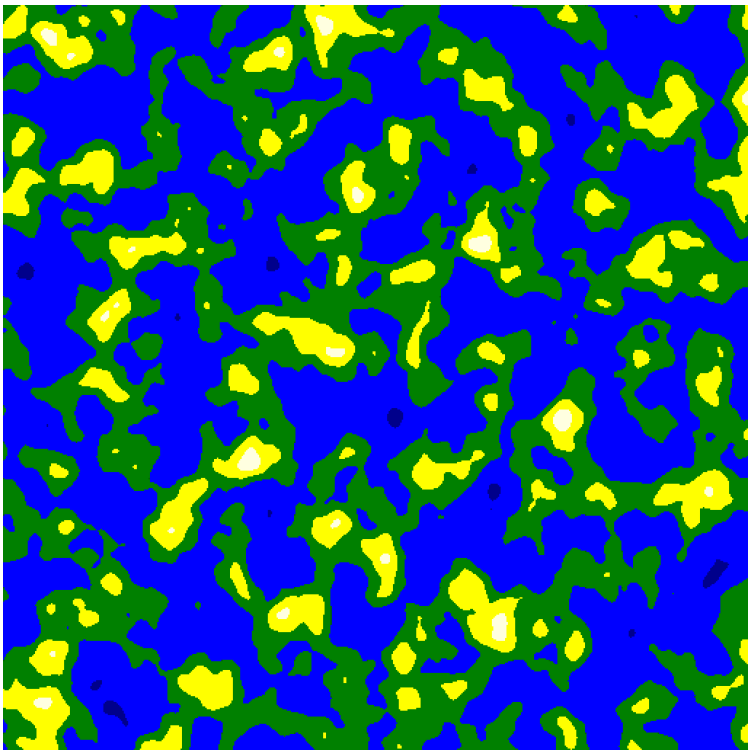


Figura A.11. Mapa de alturas de depuración (*Height.png*)

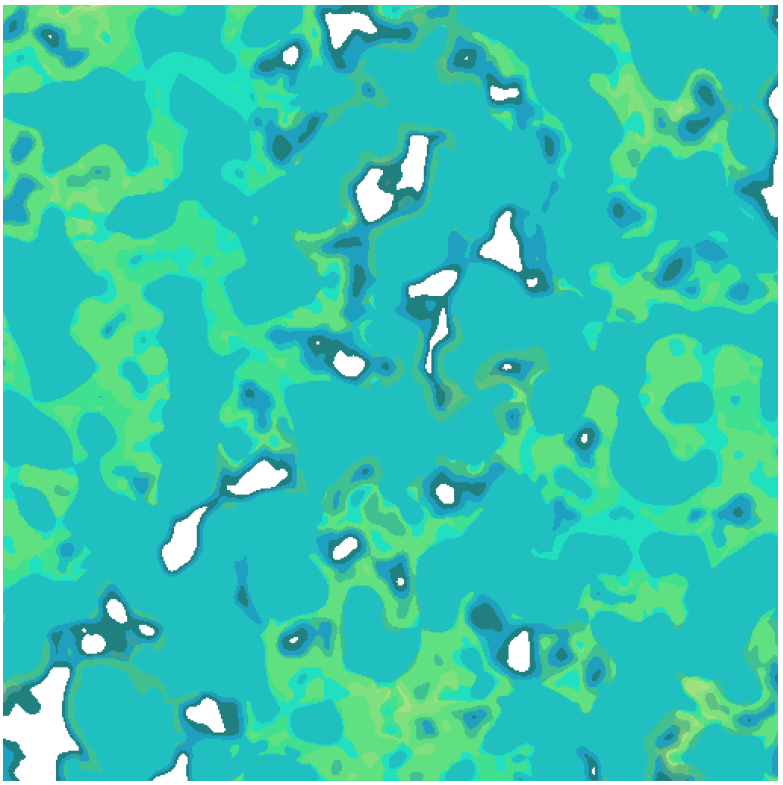


Figura A.12. Mapa de biomas según la clasificación de Holdridge de depuración (*HoldridgeBiome.png*)

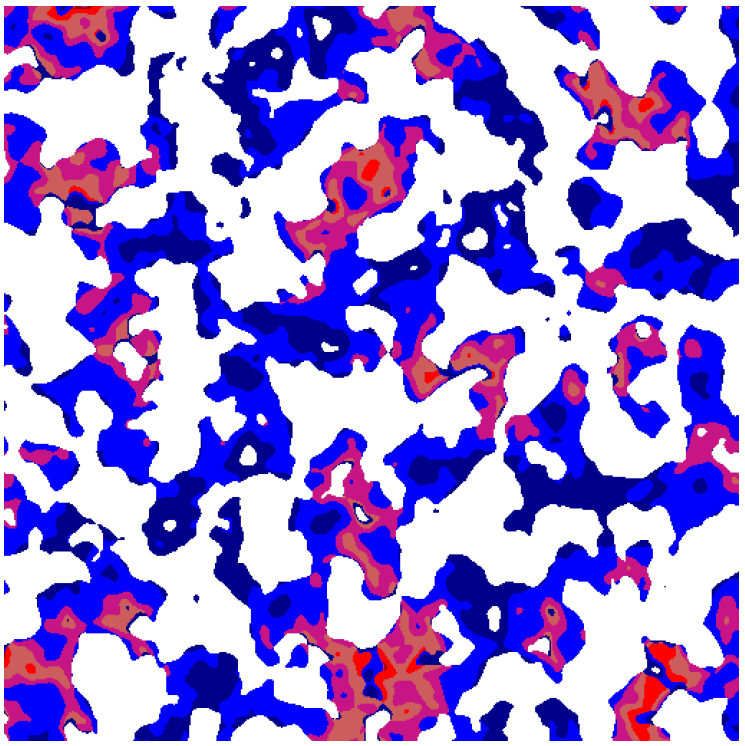


Figura A.13. Mapa de humedades de depuración (*Humidity.png*)

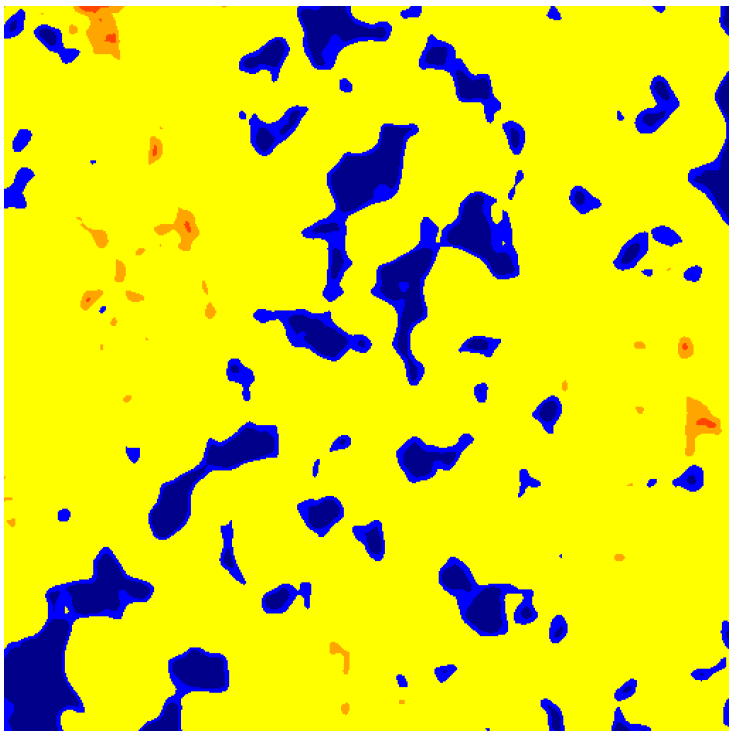


Figura A.14. Mapa de temperaturas de depuración (*Temp.png*)

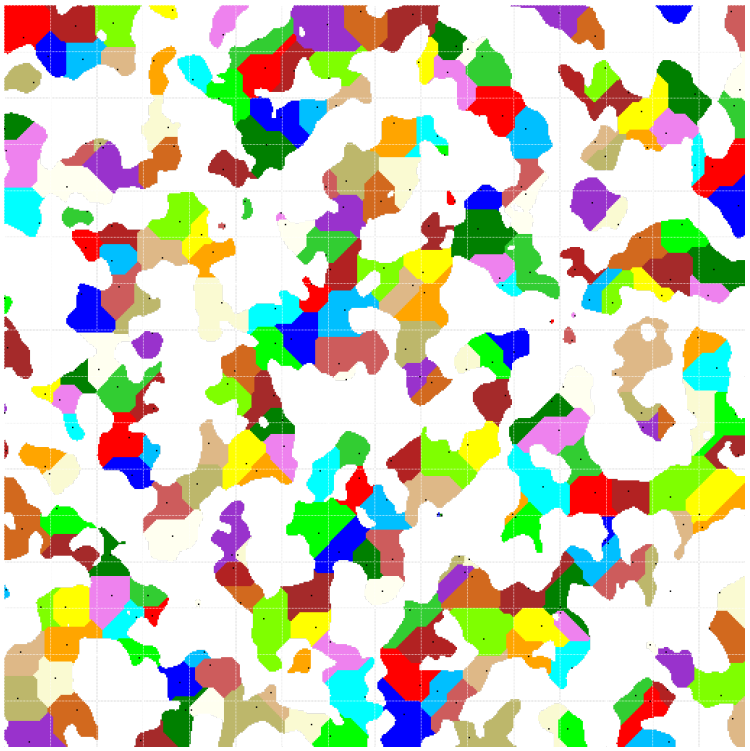


Figura A.15. Mapa de regiones de Voronoi de depuración (*VoronoiRegions.png*)

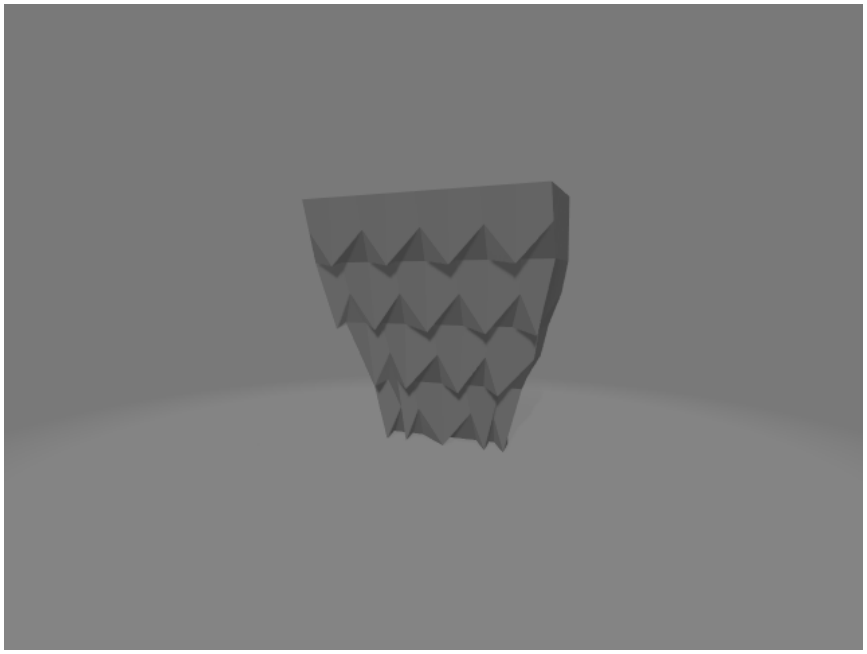


Figura A.16. Modelo de barba utilizado en la exhibición de Unity

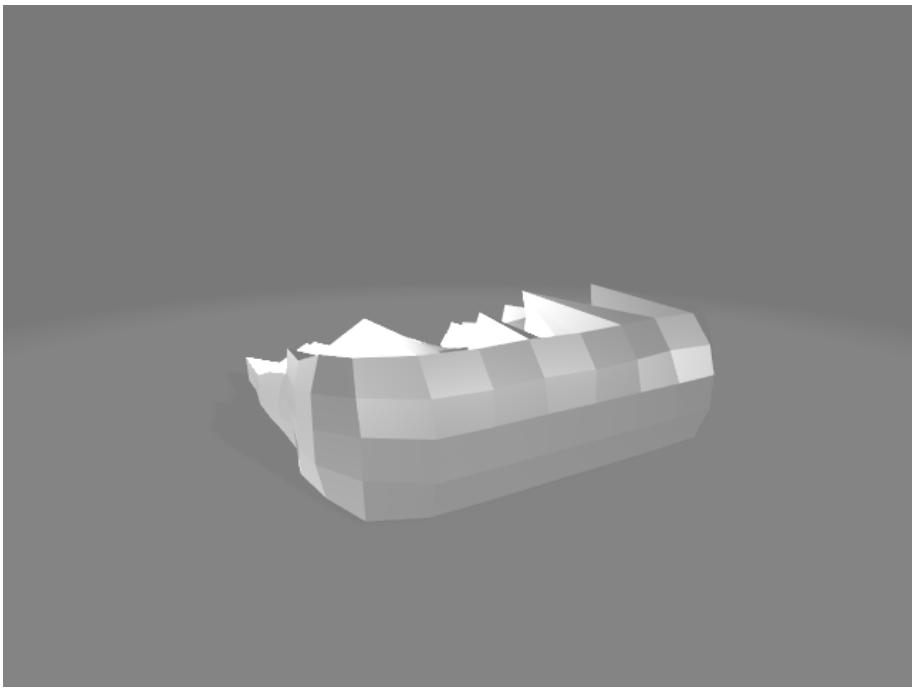


Figura A.17. Modelo de pelo utilizado en la exhibición de Unity

Figura A.18. Primer modelo de pata utilizado en la exhibición de Unity

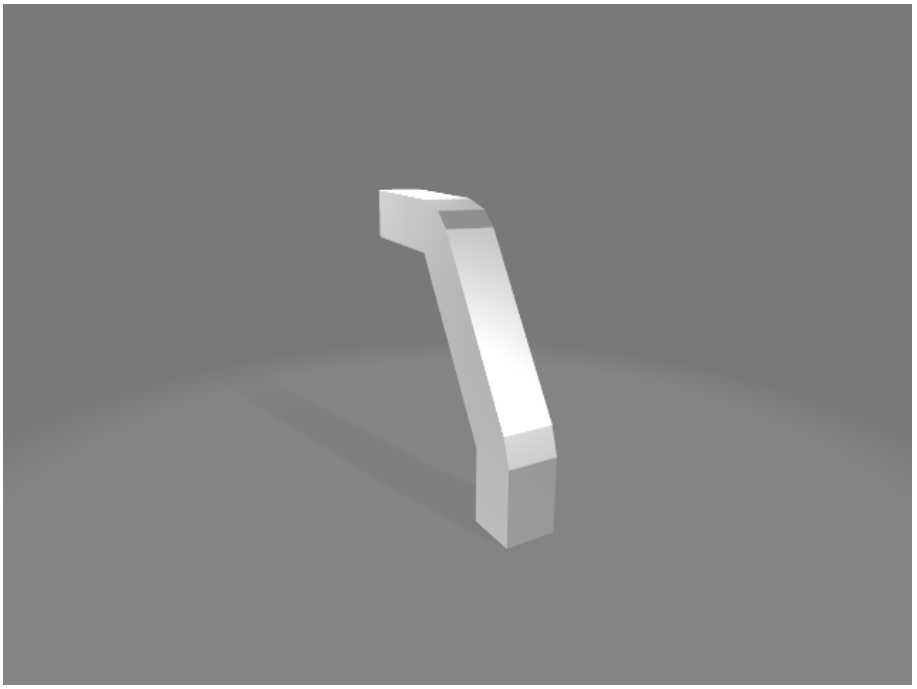


Figura A.19. Segundo modelo de pata utilizado en la exhibición de Unity

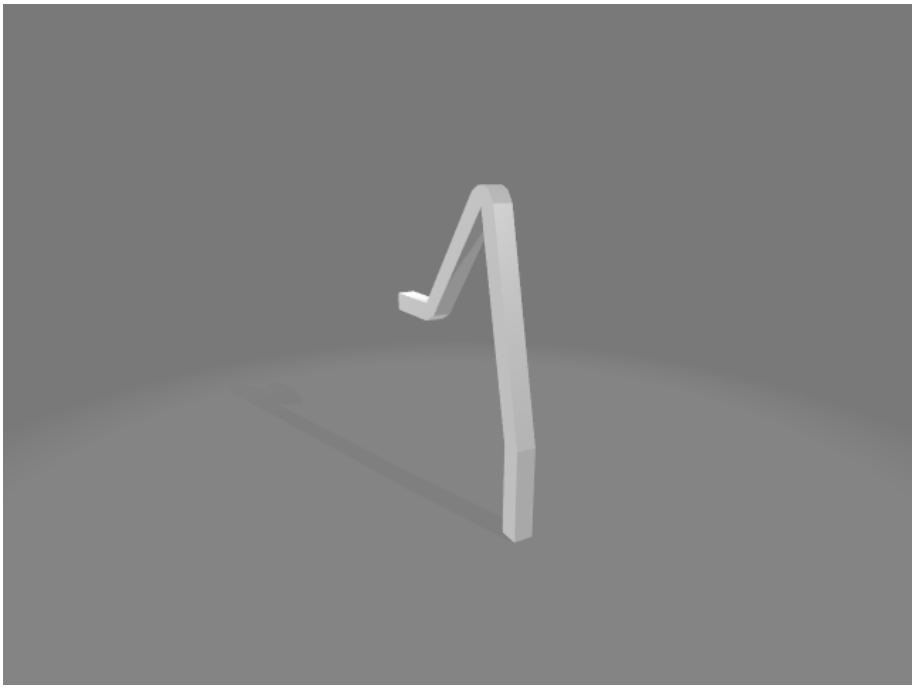


Figura A.20. Tercer modelo de pata utilizado en la exhibición de Unity

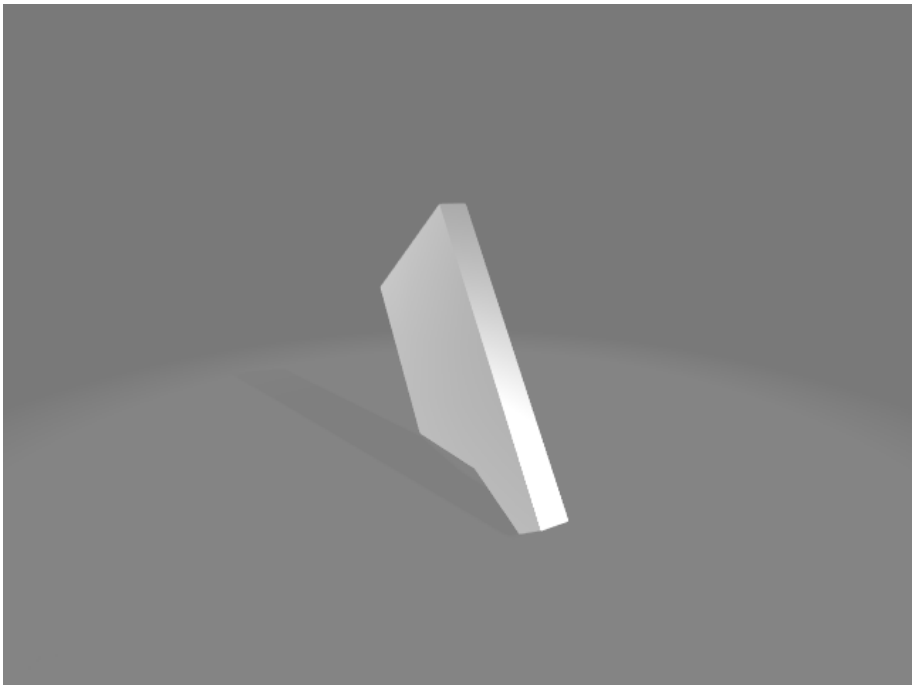


Figura A.21. Primer modelo de ala utilizado en la exhibición de Unity

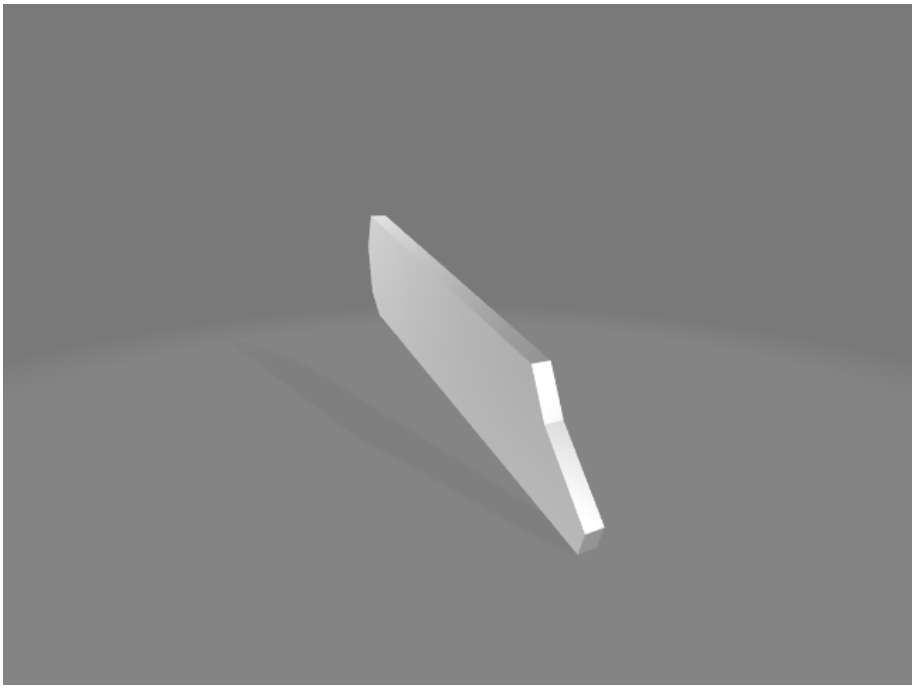


Figura A.22. Segundo modelo de ala utilizado en la exhibición de Unity

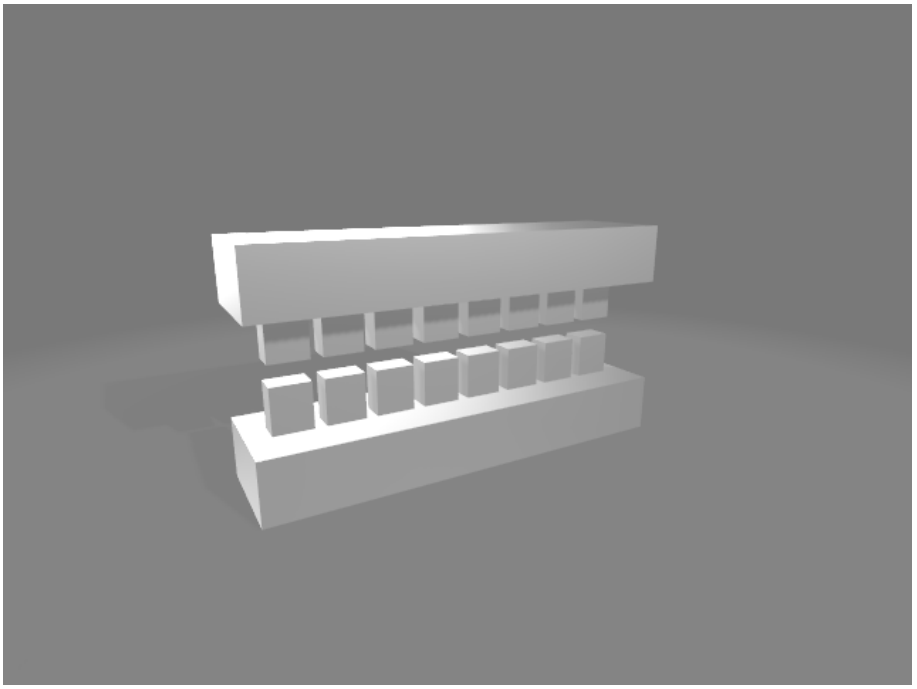


Figura A.23. Modelo de boca para criaturas herbívoras utilizado en la exhibición de Unity

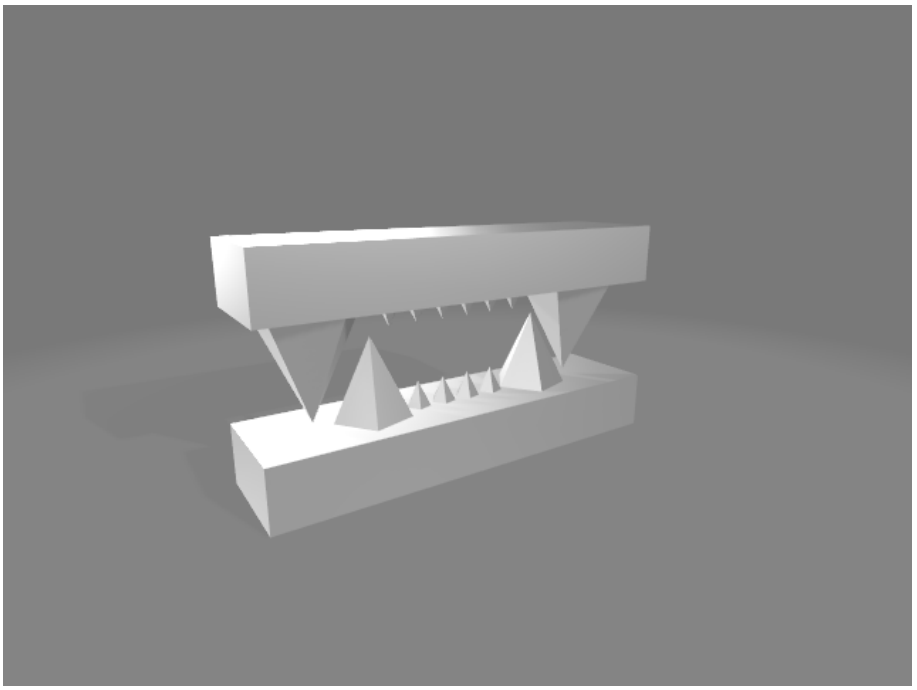


Figura A.24. Modelo de boca para criaturas carnívoras utilizado en la exhibición de Unity

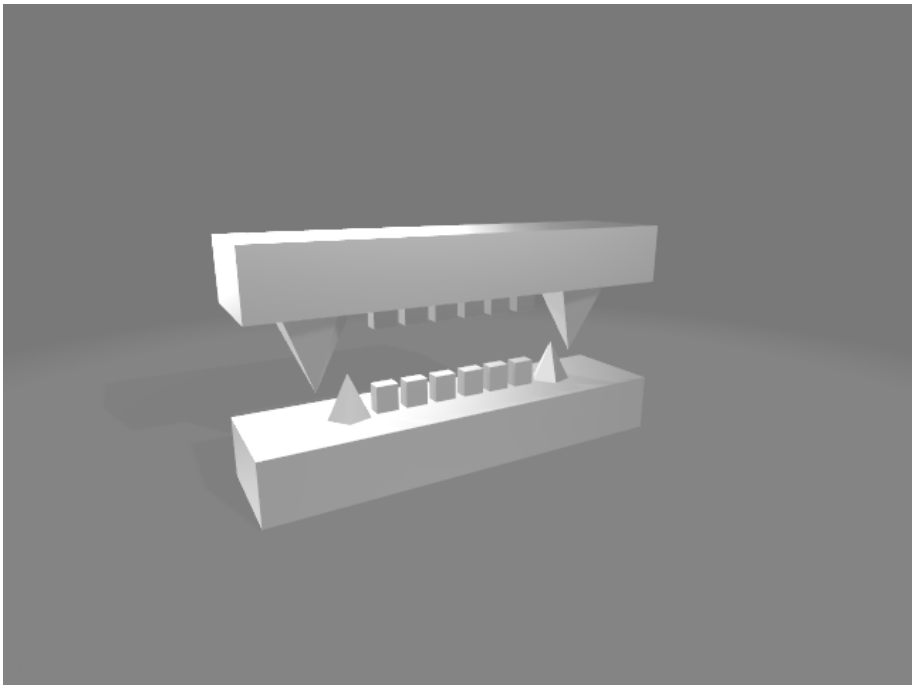


Figura A.25. Modelo de boca para criaturas omnívoras utilizado en la exhibición de Unity

B. Documentos Adicionales

$$f(x,\ y)\ =\ (1\ +\ x)\ *\ (y\ -\ avgT)\ +\ avgT : \forall x, y \in [0, 1]$$

Ecuación B.1. Suavizado de Temperatura

```
double treeThreshold = 0.4, bushThreshold = 0.2;
if (floraProb <= treeThreshold)
{
    if (RandomGenerator.NextDouble() <= ((1 / (treeThreshold -
bushThreshold)) * (floraProb - bushThreshold)))
        return 1;
    else
        return 0;
}
else
{
    if (RandomGenerator.NextDouble() < 0.95)
        return 2;
    else
        return 3;
}
```

Figura B.1. Función de selección de flora

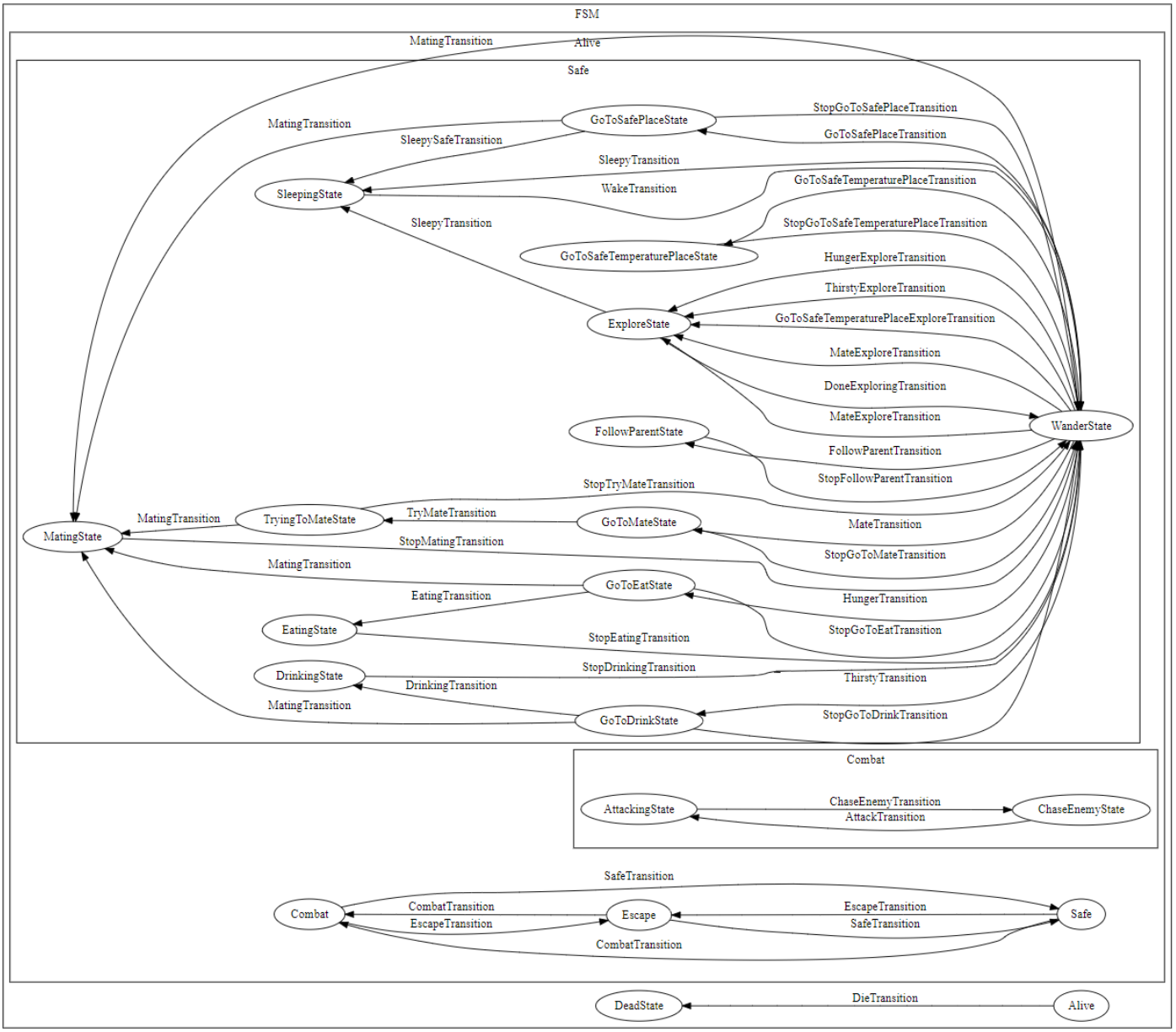


Figura B.2. Máquina de estados completa

Estado	Transiciones
Macroestado *Alive*	Transición *DieTransition*: Transición que lleva al estado *Dead*. Su condición se cumple cuando la vida de la criatura es inferior o igual a 0.

Macroestado *Escape*	Transición *CombatTransition*: Transición que lleva al macroestado *Combat*. Su condición se cumple si tiene otra criatura signada como adversario y considera que, con ayuda de miembros de su especie cercanos, si los hay, sería capaz de acabar con dicho adversario; si se encuentra amenazado por una criatura y no tiene camino por el cual huir; o si tiene una criatura asignada como presa potencial y tiene suficiente hambre como para necesitar cazar (y le es más rentable que comer una planta, ya sea por su dieta o por proximidad).
	Transición *SafeTransition*: Transición que lleva al macroestado "Safe". Su condición se cumple cuando la criatura no tiene ninguna otra criatura asignada como adversario, y no se encuentra amenazada.
Macroestado *Combat*	Transición *SafeTransition*: Ver macroestado *Escape*.
	Transición *EscapeTransition*: Transición que lleva al macroestado *Escape*. Su condición se cumple si la criatura no tiene adversario, pero se siente amenazada y tiene camino posible por el cual huir; o si tiene un adversario pero cree que, aun teniendo ayuda de criaturas de su misma especie a su alrededor, sería incapaz de acabar con él.
Estado *ChaseEnemy*	Transición *AttackTransition*: Transición que lleva al estado *Attacking*. Su condición se cumple cuando el adversario de la criatura se encuentra a su alcance y puede ser atacado por la misma.
Estado *Attacking*	Transición *ChaseEnemyTransition*: Transición que lleva al estado *ChaseEnemy*. Su condición se cumple cuando el adversario de la criatura no se encuentra a su alcance por lo que para poder atacar debe acercarse a él.
Macroestado *Safe*	Transición *EscapeTransition*: Ver macroestado *Combat*.
	Transición *CombatTransition*: Ver macroestado *Escape*.

Estado *Drinking*	Transición *StopDrinkingTransition*: Transición que va al estado *Wander*. Su condición se cumple cuando la sed de la criatura ha sido saciada por completo o no tiene sed y tiene hambre o está muy cansada.
Estado *Eating*	Transición *StopEatingTransition*: Transición que va al estado *Wander*. Su condición se cumple cuando la criatura sacia su hambre hasta un umbral específico, o si el alimento que busca comer está demasiado lejos o si no está hambriento y tiene sed o está muy cansada.
Estado *Explore*	Transición *DoneExploringTransition*: Transición que lleva al estado *Wander*. Su condición se cumple una vez el objetivo de explorar de la criatura ha sido cumplido: si estaba hambrienta y ha encontrado comida, si estaba sedienta y ha encontrado agua, si estaba cansada y ha encontrado un lugar donde dormir, o si quería buscar a alguien con quien aparearse y lo ha encontrado.
	Transición *SleepyTransition*: Transición que lleva al estado *Sleeping*. Su condición se cumple si la criatura está extremadamente cansada sin estar muy hambrienta o sedienta.
Estado *FollowParent*	Transición *StopFollowParentTransition*: Transición que lleva al estado *Wander*. Su condición se cumple una vez la criatura o deja de ser considerada una cría, o no tiene padre, o desconoce la posición de su padre, o está suficientemente cerca de su padre o no lo puede alcanzar.
Estado *GoToDrink*	Transición *StopGoToDrinkTransition*: Transición que lleva al estado *Wander*. Su condición se cumple si la criatura desconoce una posición de agua, o está extremadamente cansada, o si no está sedienta y en su lugar está muy hambrienta.
	Transición *DrinkingTransition*: Transición que lleva al estado *Drinking*. Su condición se cumple cuando se encuentra a una distancia de la fuente de agua a la que está yendo que le permite beber de la misma.
	Transición *MatingTransition*: Transición que lleva al estado *Mating*. Su condición se cumple si la criatura se está apareando en ese momento.

Estado *GoToEat*	Transición *StopGoToEatTransition*: Transición que lleva al estado *Wander*. Su condición se cumple cuando la criatura no tiene alimento que comer, cuando está extremadamente cansado o si no está muy hambrienta y en su lugar está muy sedienta.
	Transición *EatingTransition*: Transición que lleva al estado *Eating*. Su condición se cumple cuando se encuentra a una distancia del alimento más cercano que le permite comérselo.
	Transición *MatingTransition*: Ver estado *GoToDrink*.
Estado *GoToMate*	Transición *StopGoToMateTransition*: Transición que lleva al estado Wander. Su condición se cumple cuando no tiene ninguna criatura con la que reproducirse, o tiene alguna necesidad (hambre, sed, o cansancio).
	Transición *TryMateTransition*: Transición que lleva al estado *TryMate*. Su condición se cumple cuando tiene una criatura con la que reproducirse y ésta se encuentra a una distancia suficientemente corta.
Estado *GoToSafePlace*	Transición *StopGoToSafePlaceTransition*: Transición que lleva al estado *Wander*. Su condición se cumple cuando la criatura deja de estar cansada o cuando no conoce un lugar seguro donde dormir.
	Transición *SleepySafeTransition*: Transición que lleva al estado *Sleeping*. Su condición se cumple si la criatura está cansada y se encuentra en una posición adyacente a una zona segura, o se encuentra extremadamente cansada.
	Transición *MatingTransition*: Ver estado *GoToDrink*.
Estado *GoToSafeTemepraturePlace*	Transición *StopGoToSafeTemperaturePlaceTransition*: Transición que lleva al estado *Wander*. Su condición se cumple cuando la criatura se encuentra en un lugar cuya temperatura es apta para su supervivencia.
Estado *Mating*	Transición *StopMatingTransition*: Transición que lleva al estado *Wander*. Su condición se cumple cuando la criatura ya no está en proceso de aparearse, o no tiene criatura con quien aparearse.

Estado *Sleeping*	Transición *WakeTransition*: Transición que lleva al estado *Wander*. Su condición se cumple cuando la criatura ha descansado por completo o no está cansada y tiene hambre o está cansada.
Estado *TryMate*	Transición *StopTryMateTransition*: Transición que lleva al estado *Wander*. Su condición se cumple cuando la criatura no tiene con quién aparearse, o no se está apareando, o tiene alguna necesidad (hambre, sed, o cansancio).
	Transición *MatingTransition*: Ver estado *GoToDrink*.
Estado *Wander*	Transición *FollowParentTransiition*: Transición que lleva al estado *FollowParent*. Su condición se cumple cuando la criatura es una cría y tiene constancia de su progenitor, y se encuentra además a una distancia demasiado alejada del mismo.
	Transición *GoToSafeTempPlaceTransition*: Transición que lleva al estado *GoToSafeTemperaturePlace*. Su condición se cumple cuando la criatura se encuentra en una posición cuya temperatura no es apta, y conoce una posición en la cual sí lo es.
	Transición *GoToSafeTemperaturePlaceExploreTransition*: Transición que lleva al estado *Explore*. Su condición se cumple cuando la criatura se encuentra en una posición cuya temperatura no es apta, y no conoce una posición en la cual sí lo es.
	Transición *SleepyTransition*: Ver estado *Explore*.
	Transición *SafePlaceExploreTransition*: Transición que lleva al estado *Explore*. Su condición se cumple cuando la criatura está cansada y no conoce una posición segura en la que pueda dormir.
	Transición *GoToSafePlaceTransition*: Transición que lleva al estado *GoToSafePlace*. Su condición se cumple cuando la criatura no conoce una posición segura, y está cansada o en una posición peligrosa.
	Transición *DrinkingExploreTransition*: Transición que lleva al estado *Explore*. Su condición se cumple

	cuando la criatura está sedienta y no conoce una fuente de agua en la cual beber.
	Transición *ThirstyTransition*: Transición que lleva al estado *GoToDrink*. Su condición se cumple cuando la criatura está sedienta y conoce una fuente de agua en la cual beber.
	Transición *HungerExploreTransition*: Transición que lleva al estado *Explore*. Su condición se cumple cuando la criatura está hambrienta y no conoce un objetivo para alimentarse.
	Transición *HungerTransition*: Transición que lleva al estado *GoToEat*. Su condición se cumple cuando la criatura está hambrienta y conoce un objetivo para alimentarse.
	Transición *MatingExploreTransition*: Transición que lleva al estado *Explore*. Su condición se cumple cuando la criatura es de género másculino, no tiene ninguna necesidad (hambre, sed o cansancio) y no conoce ninguna criatura con la cual pueda aparearse.
	Transición *GoToMateTransition*: Transición que lleva al estado *GoToMate*. Su condición se cumple cuando la criatura es de género másculino, no es una cría y conoce alguna criatura con la cual pueda aparearse.
	Transición *MatingTransition*: Ver estado *GoToDrink*.

Tabla B.1. Transiciones de cada estado

Atributo	Descripción
ticksPerHour	Número de ciclos que se corresponden a una hora de simulación.
hoursPerDay	Número de horas que se corresponden con un día de

	simulación.
daysPerYear	Número de días que se corresponden con un año de simulación.
morningStart	Hora a la que comienza el día
nightStart	Hora a la que comienza la noche
grassHp	"Vida" de la hierba. Deriva en la cantidad de veces que se puede comer antes de ser consumida por completo.
bushHp	"Vida" de los arbustos. Deriva en la cantidad de veces que se pueden comer antes de ser consumidos por completo.
eTreeHp	"Vida" de los árboles con fruto comestible. Deriva en la cantidad de veces que se pueden comer antes de ser consumidos por completo.
grassNutritionalValue	Cantidad de alimento que proporciona la hierba.
bushNutritionValue	Cantidad de alimento que proporciona un arbusto.
eTreeNutritionalValue	Cantidad de alimento que proporciona un árbol con frutos comestibles.
grassHoursTillGrowth	Número de horas necesarias para la regeneración de la hierba tras ser consumida por completo.
bushHoursTillGrowth	Número de horas necesarias para la regeneración de los arbustos tras ser consumidos por completo.
eTreeHoursTillGrowth	Número de horas necesarias para la regeneración de los árboles con fruto comestible tras ser consumidos por completo.
abilityUnlockPercentage	Cantidad relativa al máximo del cromosoma que tiene que ser igualada o superada para disponer de una habilidad. Sólo será usado si no se provee del archivo que configura estos valores.
minHealth	Mínimo valor de vida que tiene que tener una criatura.

healthGainMultiplier	Punto de vida máxima por cada punto de constitución tenga la criatura.
healthRegeneration	Cantidad relativa de vida que se regenera cuando la criatura está sana.
maxLimbs	Número máximo de extremidades que puede llegar a tener una criatura.
resourceAmount	Máxima cantidad de cada necesidad vital (energía, descanso e hidratación).
minPerception	Mínimo radio de percepción de las criaturas.
maxPerception	Máximo radio de percepción de las criaturas.
minLifeSpan	Número mínimo de años de vida de las criaturas.
exhaustToSleepRatio	Multiplicador que indica lo rápido que se descansa al dormir en relación al gasto energético de estar despierto. Es decir, si una criatura consume una cantidad X de energía al estar despierta, mientras duerme recuperará una cantidad X multiplicada por este número.
hoursTilExhaustion	Número de horas que han de pasar para que las criaturas se agoten por completo.
perceptionWithouNight Vision	Cantidad relativa de percepción que tienen las criaturas sin visión nocturna al anochecer.
minPerceptionWithNight Vision	Cantidad relativa de percepción que tienen las criaturas con visión nocturna al anochecer.
minMobilityMedium	Movilidad mínima que puede alcanzar una criatura al moverse por terreno que penaliza su movimiento. Valores entre 0.6 y 1.
mobilityPenalty	Penalización de movilidad terrestre para las criaturas que pueden moverse en otros medios.

yearsBetweenHeats	Tiempo que necesitan las criaturas para poder volver a entrar en celo.
maxChildNumber	Número máximo de crías que pueden haber en nuevas camadas.
ticksToReproduce	Ticks de la simulación que dura la cópula entre criaturas.
maxSpeed	Multiplicador de la velocidad que establece la velocidad máxima de las criaturas.
hornIntimidation Multiplier	Aumento de la intimidación en las criaturas que tienen cuernos.
hairTemperature Multiplier	Aumento de la temperatura percibida por las criaturas que tienen pelo.
restRegeneration Threshold	Umbral de descanso a partir del cual una criatura considera estar descansada y puede regenerar su salud.
energyRegeneration Threshold	Umbral de energía a partir del cual una criatura considera estar bien nutrida y puede regenerar su salud.
hydrationRegeneration Threshold	Umbral de hidratación a partir del cual una criatura considera estar hidratada y puede regenerar su salud.
regenerationRate	Porcentaje de la vida máxima que se puede regenerar cada ciclo.
hoursTilStarvation	Horas que las criaturas pueden estar sin comer hasta empezar a morir de hambre.
thirstToHungerRatio	Velocidad a la que se deshidratan las criaturas en relación con el hambre que tienen.

maxTemperature Difference	Grados por encima del límite de cada criatura que influyen en la penalización por estar en zonas de temperaturas extremas.
minHealthTemperature Damage	Mínima cantidad de daño por vida máxima hecho al estar en temperaturas extremas.
maxHealthTemperature Damage	Máxima cantidad de daño por vida máxima hecho al estar en temperaturas extremas.
venomDamageMultiplier	Multiplicador del que se deriva el daño producido por veneno. Este se obtiene multiplicando la característica numérica de veneno de la criatura por este número.
omnivorousNutrition Multiplier	Penalización a la nutrición de las criaturas omnívoras.
newbornStatMultiplier	Reducción de las cualidades de las criaturas recién nacidas.
adulthoodThreshold	Umbral de la vida de la criatura a partir de la cual se considera adulta.
tiredThreshold	Umbral a partir del cual una criatura considera que necesita dormir.
exhaustThreshold	Umbral a partir del cual una criatura considera estar cansada.
hungryThreshold	Umbral a partir del cual una criatura considera que tiene hambre.
veryHungryThreshold	Umbral a partir del cual una criatura considera que necesita comer urgentemente.
thirstyThreshold	Umbral a partir del cual una criatura considera que tiene sed.

veryThirstyThreshold	Umbral a partir del cual una criatura considera que necesita beber agua urgentemente.
treeMovementPenalty	Penalización para las criaturas no arbóreas que atraviesan zonas con árboles.
knowledgeTickMultiplier	Tiempo en ciclos que puede recordar una criatura cualquier cosa. Está relacionado con el conocimiento de las mismas.
aggressivenessToRadius Multiplier	Multiplicador del radio de percepción que puede considerar una criatura como peligrosa.
maxResources Remembered	Número máximo de recursos que puede recordar una criatura.
maxPositions Remembered	Número máximo de posiciones que puede recordar una criatura.
baseActionCost	Costa base de cualquier acción.
venomCostMultiplier	Coste extra de aplicar veneno.
chaseCostMultiplier	Modificador del coste de perseguir a otras criaturas
fleeingCostMultiplier	Modificador del coste de huir de otras criaturas.
drinkingCostMultiplier	Modificador del coste de beber agua.
drinkingMultiplier	Modificador de la cantidad de agua que se recupera al beber.
eatingCostMultiplier	Modificador del coste de comer.
sleepingCostMultiplier	Modificador del coste de dormir.
mutationChance	Probabilidad que tiene una criatura de mutar al nacer.

adjacentLength	Cantidad mínima de casillas de distancia hacia un punto a partir de la cual una criatura puede considerarse adyacente e interactuar físicamente con el mismo.
actionPerception Percentage	Modificación del radio de percepción cuando una criatura está realizando alguna acción.
stopEatingTransition EnergyPercentage	Porcentaje de energía máxima para dejar de comer.
sleepingExpense Reduction	Reducción al gasto de recursos cuando las criaturas duermen.
fleeingTransition Multiplier	Multiplicador para calcular la agresividad de las criaturas mientras huyen.
combatTransitionHealth ThresholdMultiplier	Multiplicador para calcular si ataca o no en función a la vida que le queda.
maxMenaceIntimidation MultiplierBasedOn MissingHealth	Multiplicador para calcular la intimidación en función a la vida que le falte a la criatura.
experienceMax AggresivenessMultiplier	Porcentaje de la agresividad máxima usada para calcular experiencias.
rotStartMultiplier	Umbral relativo al tiempo de los cadáveres a partir del cual empiezan a pudrirse.
corpseNutritionPoints Multiplier	Modificación de la cantidad de alimento que proveen los cadáveres.
percentageSimilarity Species	Porcentaje de similitud a partir del cuál un se considera una nueva especie.

Tabla B.2. Descripción del archivo *UniverseParameters.json*