Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Desarrollo de un videojuego de simulación de granjas competitivo

Development of a competitive farm simulation video game

Juan Salvador Magariños Alba

La Laguna, *26* de *mayo* de 2023

D. **Jesús Miguel Torres Jorge**, con N.I.F. 43.826.207-Y profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

“Desarrollo de un videojuego de simulación de granjas competitivo”

ha sido realizada bajo su dirección por D. **Juan Salvador Magariños Alba**,

con N.I.F. 43.493.146-P.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 21 de noviembre de 2018

Agradecimientos

Quiero dar las gracias a mi familia, sobre todo a mis padres, por su ayuda y ánimo a lo largo de toda la carrera, sin la que no habría sido posible llegar aquí.

Agradezco también a mi tutor, Jesús Torres, su guía y apoyo durante el proyecto.

Agradezco también a Rafael Arnay su guía en el desarrollo de la IA.

También quiero dar las gracias a mis amigos por su apoyo durante estos meses de trabajo, especialmente a Marta, que también hizo los iconos del juego.

Licencia

Resumen

El objetivo de este trabajo ha sido desarrollar un videojuego de granjas en el que el jugador deba enfrentarse a una inteligencia artificial. En el juego se dan al jugador una serie de recursos en forma de herramientas y espacios en los que realizar diversas actividades, y debe gestionarlos con el propósito de ganar la mayor cantidad de dinero posible. Se pueden realizar varias actividades de las que están presentes en la mayoría de juegos de gestión de granjas, como plantar cultivos, cuidar animales, y obtener recursos naturales como madera y minerales. Con estos recursos es posible fabricar herramientas que sustituyan a aquellas que se rompen por su uso, y cocinar platos que permitan al jugador recuperar energía necesaria para realizar las actividades.

Además de desarrollar el propio juego, también se ha programado la IA en base a las prioridades que daría un jugador humano a las diferentes actividades, con el objetivo de que esta pueda jugar lo mejor posible. Concretamente, se ha elegido la arquitectura de inteligencia artificial GOAP porque era la que mejor podría adaptarse a las necesidades del proyecto, pues se requería que la IA fuera capaz de modificar su comportamiento y planificar sus siguientes acciones en base a los recursos de los que dispone en cada momento.

El motor de desarrollo que se ha elegido para desarrollar el videojuego es Unity 3D, principalmente debido a la experiencia ya obtenida con dicha herramienta y con el lenguaje de programación C# en la carrera. El juego se ha programado para ser jugado en PC, pero no se descarta modificarlo en un futuro para adaptarlo a otras plataformas.

Palabras clave: videojuego, granja, gestión de recursos, inteligencia artificial, GOAP

Abstract

The main objective of this project is to develop a farming video game where the player plays against artificial intelligence. The player is given different types of resources, namely tools and locations where different activities can be done, and these resources must be managed in order to earn the highest possible amount of money. Many activities that are present in most farming video games, like growing crops, taking care of animals, and looking for natural resources like wood and minerals, are available to the player. These resources can be used to craft tools to replace those that have broken due to prolonged usage, and to cook dishes that replenish the player's energy, which is necessary for doing the mentioned tasks.

In addition to developing the game, the artificial intelligence has been programmed bearing the priorities that a human player would give to each activity in mind, in order to allow the AI to play the best it can. Furthermore, the artificial intelligence architecture GOAP was chosen since it was the one which could be best adapted to the project's requirements, because the AI should be able to modify its behavior and to plan its actions taking the available resources into account.

Unity 3D was the chosen engine for the development of this video game, because of the experience that had been previously acquired with said tool ad with the C# programming language during the degree. Even though the game has been programmed to be played on PC, it may be modified in the future to port it to other platforms.

Keywords: video game, farm, resource management, artificial intelligence, GOAP

Índice general

Capítulo 1 Introducción

En este capítulo se realizará una presentación formal del proyecto, explicando el alcance del mismo y los fundamentos del juego desarrollado. El enlace al repositorio en GitHub de este proyecto está en la bibliografía.

1.1 Bases del juego

El objetivo del juego, al igual que en la mayoría de los demás del género de simulación de granjas, es gestionar los recursos disponibles en la granja y en el escenario para obtener la mayor cantidad de dinero posible. Para ello, es necesario realizar inversiones, por ejemplo, comprando animales y semillas, esperando generar beneficios, así como explorar el mapa para conseguir recursos que permitan fabricar herramientas y recuperar energía, entre otros.

Sin embargo, los juegos clásicos de simulación de granjas tienen un elemento en común que se descartó desde la concepción del juego desarrollado para este proyecto: el paso de los días en la granja. Habitualmente, el jugador debe planificar sus acciones teniendo en cuenta el día de la semana y la estación del año en la que se encuentra para poder asistir a eventos, plantar semillas que puedan crecer en la época actual, y llevar a cabo tareas disponibles solo en momentos concretos del mes.

El primer motivo fundamental por el que se eliminaron los días y las estaciones es que el juego se ha adaptado para partidas rápidas, que duran un solo día del juego. El segundo es que era difícil de conciliar con el otro pilar sobre el que sustenta el proyecto: el desarrollo de la Inteligencia Artificial. Con el objetivo de desarrollar un videojuego más interesante, se decidió implementar una IA capaz de competir contra el jugador, de forma que se añadió al juego un componente competitivo que normalmente no está presente en los de su género.

Así, el juego coloca a dos personajes, el jugador y la IA, en un mapa compuesto por las granjas que pertenecen a cada uno de ellos, un pueblo con tiendas, dos bosques, una mina y una playa. Los dos jugadores deben realizar las actividades propias de los juegos de granja, es decir, cultivar, pescar y cuidar de animales, entre otras, con el objetivo de ser quien haya ganado más dinero al final del día.

1.2 Alcance del proyecto

Por sus fundamentos, el juego toma un carácter más similar a los juegos arcade, basados en partidas rápidas y sencillas, que en los juegos en los que alcanzar los objetivos principales puede llevar muchas horas. Por ello, al final del proyecto se debería haber realizado un mapa y una IA que permitan realizar partidas rápidas y completas. Sin embargo, idealmente el juego se desarrollará de tal forma que permita añadir mapas nuevos sin muchas complicaciones.

Capítulo 2 Estado del arte

El objetivo de este capítulo es detallar los antecedentes y el estado del arte de los tópicos principales relacionados con este TFG, que son los videojuegos de granjas y la aplicación de la inteligencia artificial al desarrollo de juegos.

2.1 Antecedentes

Los juegos de simulación de granjas aparecieron en 1996 con Harvest Moon, lanzado para la SNES de Nintendo, videojuego que daría nombre a una franquicia completa renombrada después como Story of Seasons. Desde el lanzamiento del primer Harvest Moon, han aparecido otros juegos de este género, como los Rune Factory, que son spin-offs de dicha franquicia, Stardew Valley, de 2016, y My time at Portia, de 2019.

Por otro lado, la inteligencia artificial se introdujo en el campo de los videojuegos cuando ambos aún eran muy recientes, con juegos como Nim (1951), Qwak (1974) y Pursuit (1975); pero en la época dorada de los arcades las IA de los videojuegos incrementaron su sofisticación con la introducción de los patrones en laberintos, juegos de lucha, enemigos, etc., con Space Invaders (1978) y Pac-Man (1980), entre otros. Más tarde, se comenzó a usar herramientas de inteligencia artificial como las máquinas de estados, debido al surgimiento de nuevos géneros de videojuegos en la década de 1990, y, en la actualidad, se han aplicado también muchos algoritmos y técnicas nuevas, como los algoritmos de *pathfinding* y los árboles de decisión.

2.2 Estado actual

En la actualidad, los videojuegos de granjas se han extendido, no solo en cuanto a popularidad, sino también con respecto a la cantidad de juegos de dicho género que son lanzados al mercado. Por este motivo, si bien muchos de estos siguen el esquema clásico del género, como Coral Island (2022), otros desarrolladores deciden mezclar sus mecánicas con aquellas propias de otros tipos de videojuegos. Algunos ejemplos de esto son Harvestella (2022), que integra el género de granjas con el RPG, y Sakuna: Of Rice And Ruin (2020), un juego de aventuras en 2D que hace una mezcla similar.

Con respecto al estado del arte de la aplicación de técnicas de IA al desarrollo de videojuegos, en los últimos años se ha extendido la generación procedural, que consiste en la utilización de un algoritmo para generar varios de los elementos que componen el juego, como los escenarios y los modelos, con el objetivo de hacer más variada la experiencia de juego. Por ejemplo, No Man’s Sky (2016), que consiste en explorar el universo, permite al jugador visitar diversas galaxias en las que elementos como los planetas y la música que suena al moverse por ellos han sido generados con esta técnica. Por otro lado, la IA también se está utilizando para crear NPC que se comporten de una manera más realista, no solo aumentando la complejidad de su comportamiento mediante nuevos algoritmos, sino haciendo que sean capaces de mantener conversaciones de forma similar a como lo haría un ser humano, y de reaccionar a cambios en el entorno.

Capítulo 3 Organización del proyecto

El objetivo general de este capítulo es explicar los objetivos que se intentó alcanzar durante el desarrollo del proyecto, así como la metodología seguida para ello. Además, se detallarán los principales problemas encontrados y cómo se solucionaron.

3.1 Objetivos principales y secundarios

3.1.1 Objetivos principales

A continuación, se exponen los objetivos más importantes del proyecto:

- **Desarrollar un videojuego de gestión de granjas**: con este propósito en mente, se programó un juego que tuviera los componentes principales de dicho género, es decir, la posibilidad de realizar actividades de agricultura, ganadería, pesca, tala y minería, entre otras, con el objetivo de ganar dinero que permita al jugador progresar. Si bien los juegos de granjas convencionales permiten desarrollarlas a lo largo de varios días, e incluso de estaciones, para este proyecto se buscó un enfoque más arcade en el que las partidas duran solo un día, y el objetivo es conseguir todo el dinero posible.
- **Desarrollar una IA capaz de jugar al videojuego**: en un principio, dentro del alcance del TFG solo estaría la creación del juego en sí. Sin embargo, se incluyó el desarrollo de una IA que pudiera jugar para ampliar el proyecto y porque se consideró que sería interesante profundizar en este campo y aplicarlo al trabajo, dado que la IA es un ámbito de la informática que ha cobrado importancia en los últimos años.
- **Aprender acerca del desarrollo de videojuegos**: la razón por la que se decidió hacer un TFG que versara sobre la creación de un videojuego fue debido al interés sobre el tema, así como el deseo de realizar un proyecto de mediana-gran escala en solitario, pudiendo aplicar diversos conocimientos de programación aprendidos en la carrera.

3.1.2 Objetivos secundarios

En este subapartado se enumeran objetivos que son más precisos que los explicados antes y que sirven, principalmente, para guiar el desarrollo del proyecto. Estos son:

- Diseño y creación del escenario de juego
- Programación de la interacción del jugador con el entorno
- Implementación de un sistema de inventario que permita tanto al usuario como a otros elementos del juego acceder a los objetos de los que este dispone
- Desarrollo de las diferentes actividades propias de los juegos de granja (agricultura, ganadería, pesca, tala, etc.)
- Adaptación de los sistemas desarrollados a la interacción con la IA
- Desarrollo de la IA
- Búsqueda o creación de elementos artísticos (modelos, efectos de sonido, etc.)

3.2 Metodología

Si bien no se ha seguido una metodología concreta de una forma muy estricta, la manera en la que se ha organizado el trabajo se acerca al pensamiento ágil. El trabajo realizado se puede agrupar en tres grandes bloques:

- Desarrollo de las mecánicas del juego
- Programación de la IA
- Incorporación de modelos, animaciones, etc.

El desarrollo del primer bloque fue el más largo, pues duró hasta mediados de abril. A partir de entonces se comenzó con la implementación de la IA, aunque se realizaron algunas tareas de corrección de errores y refactorización que encajan dentro del primer bloque de trabajo. Por otro lado, la búsqueda y creación de los modelos, animaciones, efectos de sonido, y otros elementos que conforman el lado artístico del videojuego se ha realizado poco a poco a lo largo del segundo cuatrimestre.

En cuanto al trabajo diario, se preparaba una lista de tareas a completar a lo largo de cada semana, y se distribuían a lo largo de la misma en función del tiempo disponible cada día.

3.3 Problemas encontrados

En líneas generales, el desarrollo del proyecto ha avanzado sin demasiadas dificultades ni ningún acontecimiento que obligara a detenerlo. Sin embargo, sí ha habido factores y problemas que han causado que el trabajo se ralentizara. Algunos de ellos son:

- **Falta de experiencia**: en el momento en el que se comenzó el desarrollo del TFG no se contaba con muchos conocimientos de Unity ni modelado 3D, además de los básicos. Afortunadamente, a lo largo de la realización del juego ha sido posible aprender mucho acerca de dichos tópicos.
- **Complejidad de la IA**: si bien la técnica GOAP, cuyo funcionamiento se explica en el apartado 4.3.1, permite generar agentes inteligentes modulares a los que es sencillo añadir nuevas acciones y metas, ha sido necesario planificar y programar el comportamiento de la IA con el objetivo de que pueda reaccionar correctamente a diferentes situaciones que puedan dificultar su trabajo.
- **Refactorizaciones**: a lo largo del proyecto se ha desarrollado el código del juego intentando evitar posibles refactorizaciones en el futuro. Sin embargo, ha sido necesario realizar algunas modificaciones importantes en momentos avanzados del trabajo, de las que cabe destacar la adición de los objetos extendidos, explicada en el apartado 4.2.2. No obstante, se ha logrado evitar que los demás cambios realizados en el código ya desarrollado no fueran menores.

Capítulo 4 Desarrollo del videojuego

Una vez explicados los antecedentes y objetivos en los que se ha basado este proyecto, en este capítulo se profundizará en las mecánicas y la implementación del mismo.

4.1 Mecánicas

Para empezar, a continuación se detallan las mecánicas principales del juego.

4.1.1 Mecánicas básicas

Para empezar, se comentará brevemente el funcionamiento básico de la partida:

- El jugador se puede mover en cualquier dirección e interactuar con los elementos del escenario utilizando un objeto o no.
- La partida comienza colocando a cada jugador en su granja con algunas semillas, alimentos y herramientas en el inventario. Desde entonces y hasta que se acabe el tiempo, deberá intentar conseguir todo el dinero que pueda.
- Puede utilizar el dinero para comprar objetos y animales. Sin embargo, el dinero que se contará al final del día es el que posea el jugador en ese momento, por lo que no se recomienda hacer inversiones cuando queda poco tiempo.
- El jugador está limitado por su energía, que se reduce cada vez que utiliza una herramienta. Si en un momento dado, el jugador quiere usar un utensilio, pero no tiene energía suficiente para hacerlo, la acción no se podrá realizar.

4.1.2 Cultivos

Una de las actividades más rentables e importantes de las que puede realizar el jugador a lo largo de la partida es plantar semillas, regarlas, esperar a que crezcan, y cosechar sus frutos, pues estos se pueden utilizar para recuperar energía, venderlos, e incluso para cocinar platos que otorguen estados beneficiosos llamados *buffs*, que pueden ayudar al jugador a trabajar de forma más eficiente.

En el juego existen tres tipos de cultivos:

- **Cultivos de una sola cosecha**: son aquellos cuya planta, como su nombre indica, solo da frutos una vez antes de destruirse. Dado que es necesario invertir constantemente en comprar semillas para tener un abastecimiento regular de este tipo de plantas, estas verduras son, por lo general, más valiosas que las demás y recuperan más energía cuando el jugador las consume. Los cultivos que pertenecen a esta categoría son la papa, la sandía, la zanahoria, la calabaza, y la cebolla. Además, las matas de hierba también pertenecen a esta categoría, y son el único cultivo que solo puede ser cosechado con una guadaña.
- **Cultivos de varias cosechas**: a diferencia de los anteriores, estas plantas no desaparecen cuando se recogen sus frutos, sino que vuelven a producirlos pasado un tiempo. Por este motivo, su valor suele ser menor que el de los cultivos que sí se destruyen y no devuelven tanta energía. Este grupo incluye las fresas,

frambuesas y pimientos.
- **Árboles frutales**: técnicamente, los árboles frutales pertenecen a la categoría anterior, dado que se pueden recoger sus frutos varias veces. No obstante, dentro del juego están en una categoría distinta, dado que los árboles no se pueden plantar en los mismos espacios que los demás cultivos, suelen tardar mucho más en crecer, dan madera cuando son talados, y la IA reacciona a estos de forma distinta que a las otras plantas. A este colectivo pertenecen la manzana, la cereza, el mango, la encina y el pino. Si bien los dos últimos no son árboles frutales en la realidad, en el juego sí se los considera como tales, dado que es posible recoger sus bellotas y piñas para plantar más árboles y conseguir madera.

El jugador tiene a su disposición un huerto con 50 espacios en los que puede plantar cultivos de las dos categorías, pero nunca árboles frutales. También posee una zona con 5 parcelas que sí admiten árboles, aunque no permiten plantar cultivos de los otros grupos. Para plantar una semilla es necesario arar una parcela con una azada y, para que pueda crecer, el espacio en el que está debe estar regado.

4.1.3 Animales

En la granja de cada jugador existe un establo frente al cual pastan los animales que posea en ese momento concreto, y que tiene un límite de espacio para acoger a los animales que el jugador compre en la tienda del pueblo. Este límite tiene un valor de 12 unidades, de forma que cada animal tiene un espacio requerido en unidades que el jugador debe tener en cuenta a la hora de comprar nuevos residentes del establo.

Los animales disponibles son:

- **Vacas**: son los animales más caros y que ocupan más espacio, pues requieren de 4 unidades libres en el establo, pero pueden producir leche a una frecuencia moderada, que luego se puede convertir en queso. Para que el jugador pueda ordeñar sus vacas, debe utilizar una cubeta lechera.
- **Ovejas**: estos animales necesitan 2 espacios libres para poder añadirlos al establo, y producen lana que se puede conseguir esquilando las ovejas con unas tijeras. Sin embargo, si bien tardan bastante tiempo en generar la lana, esta se puede utilizar para hacer tela, que se puede vender por mucho dinero, y como ingrediente de fabricación.
- **Gallinas**: son animales de los que se pueden obtener grandes beneficios, dado que solo necesitan 1 espacio en el establo y ponen huevos que se pueden utilizar para hacer mayonesa, aunque con una frecuencia baja. Además, no es necesaria ninguna herramienta para recoger los huevos, por lo que puede ayudar al jugador a ganar dinero si este se ve atrapado en una situación en la que no tiene ni herramientas ni dinero para comprarlas. Por ello, ambos jugadores empiezan con una gallina en el establo.

Cabe destacar también que los animales se pueden alimentar con cualquier producto comestible para aumentar su afinidad con el jugador, y así incrementar la posibilidad de que la calidad de sus productos sea superior. Además, los animales poseen una comida favorita y otra odiada, de forma que el alimento concreto que les da el jugador influye en si la afinidad aumenta o disminuye, aunque otro factor que se tiene en cuenta en el cálculo de esta es la calidad del producto consumido por el animal.

4.1.4 Máquinas

Además del huerto y el establo, el jugador tiene a su disposición una zona en la granja con 6 mesas sobre las que se pueden colocar máquinas que toman un objeto concreto, lo

procesan, y devuelven otro. Las máquinas solo se pueden obtener comprándolas en la tienda de máquinas del pueblo.

En el juego existen las siguientes máquinas:

- **Prensa de queso**: permiten procesar una botella de leche para convertirla en queso. El tiempo de procesamiento es moderado, pero el queso permite recuperar mucha energía.
- **Máquina de mayonesa**: como su nombre indica, puede hacer mayonesa a partir de un huevo de una forma bastante veloz. Sin embargo, recupera poca energía y su valor tampoco es muy elevado.
- **Telar**: se utilizan para convertir la lana virgen obtenida al esquilar ovejas en tela que se puede utilizar para venderse por mucho dinero o para fabricar algunas recetas.

4.1.5 Objetos e inventario

En el juego existe un gran conjunto de objetos de diferentes categorías que el jugador puede obtener. Cada tipo de objeto posee características propias, como es el caso de la potencia de golpe de las herramientas y la energía restaurada por los objetos consumibles; pero ciertas características de algunos ítems pueden variar entre instancias. Por ejemplo, dos manzanas pueden tener calidades diferentes, y un hacha puede estar más desgastada por su uso que otra.

Además, cada jugador posee un inventario, un espacio de 36 huecos en los que puede almacenar los objetos que consiga. Dependiendo del tipo de objeto, es posible almacenar hasta 999 unidades del mismo objeto en la misma ranura, siempre y cuando las características que se mencionaron anteriormente que pueden variar entre unidades de un tipo de objeto sean iguales. Por ejemplo, se pueden almacenar 90 manzanas de la calidad más alta en el mismo espacio, pero no se puede añadir una de menor calidad a ese hueco concreto.

Los tipos de objetos existentes en el juego son:

- **Objetos consumibles**: son aquellos que se pueden utilizar para recuperar energía. Incluyen tanto a los platos de cocina como a los objetos que se pueden utilizar como ingredientes, como el pescado y las frutas. Cada unidad de este tipo de objeto tiene una calidad, que afecta tanto a su valor monetario como a la energía que recupera el jugador al comerse el objeto.
- **Herramientas**: el propósito principal de estos objetos es ser usados para realizar diferentes acciones, como talar árboles y pescar. Para evitar que el jugador pueda utilizar los mismos objetos siempre y forzarle a acudir a la tienda de herramientas o a fabricarlas, cada utensilio tiene una durabilidad que se va reduciendo con cada uso hasta que se rompe y desaparece del inventario. Por otro lado, si bien las herramientas no tienen calidad como los objetos, sí que tienen asociada una valoración que indica si es mejor o peor que otras herramientas del mismo tipo. Concretamente, las hachas y los picos pueden ser de piedra, hierro y oro, en orden creciente de valoración, fuerza de golpe y de durabilidad; pero el resto de herramientas solo tienen una versión.
- **Máquinas**: a diferencia de los animales, que son enviados directamente al establo en cuanto son comprados, la maquinaria explicada en el apartado 4.1.4 se guarda en el inventario al adquirirla en la tienda de máquinas. Este tipo de objetos tiene propiedades muy diferentes a las de los demás objetos, como los objetos que sirven como input y output de la máquina, y el tiempo de procesamiento.

- **Semillas**: estos objetos solo poseen un atributo propio, que es la planta asociada a la semilla.
- **Teletransportadores**: como su nombre indica, al usar estos objetos, el jugador se teletransporta a una posición concreta. En el juego existen dos ítems de este tipo, que permiten al jugador ir a la granja o a la montaña.
- **Objetos genéricos y muebles**: son ítems que no tienen propiedades propias más allá de las que poseen todos los objetos. Sin embargo, aunque los muebles destacan por su elevado precio de venta, no se pueden comprar en ninguna tienda, por lo que el jugador puede utilizar los materiales que le sobren para fabricar muebles y ganar dinero rápidamente.

4.1.6 Compra de objetos y animales

En el mercado del pueblo se encuentran cinco tiendas que el jugador puede visitar para adquirir varios tipos de bienes. Concretamente, cuatro de los cinco locales ofrecen alimentos, máquinas, semillas y herramientas, aunque este último solo ofrece herramientas de piedra, por lo que las versiones de hierro y oro solo se pueden obtener fabricándolas directamente. Por otro lado, la quinta tienda vende animales para la granja.

4.1.7 Venta de objetos

En la entrada de la granja de cada jugador hay una caja en la que se pueden colocar los objetos que se quieren vender, siendo esta la única forma de ganar dinero del juego. Dado que los objetos que quedan en el inventario al acabar el día no se consideran vendidos, lo que se recomienda es vaciar el inventario por completo antes de finalizar la partida.

4.1.8 Recolección

Además de la pesca, que se detallará en el siguiente apartado, el jugador puede llevar a cabo varias actividades de recogida de materias primas fuera de la granja. Cabe destacar que, en la mayoría de los casos, solo se puede obtener estos recursos de las maneras que se van a describir a continuación, dado que los únicos recursos que se pueden recolectar en estado silvestre y que también se pueden adquirir en una tienda son algunas frutas.

En primer lugar, en el mapa se encuentran dos bosques, uno en la zona superior, al suroeste del lago, y otro en la costa sur, al oeste de la playa. Los recursos que se encuentran en cada uno no son exactamente iguales; pero en ambos se pueden talar árboles para obtener madera con la que fabricar herramientas. Por un lado, en el bosque de la montaña predominan los pinos y los frutos silvestres, como las fresas y las frambuesas. Por otro lado, en el de la playa abundan las encinas y los árboles frutales.

En segundo lugar, al norte del pueblo, se encuentra una gran mina que el jugador puede explorar para buscar minerales. Aunque el recorrido en sí es prácticamente lineal, la cueva está dividida en cuatro zonas, de forma que en cada una abunda un tipo de material distinto. Así, si bien casi se pueden extraer piedras de casi todas las menas de la mina, las rocas del primer sector solo dan piedras al destruirlas; mientras que las de los dos sectores siguientes sueltan principalmente hierro y oro. Finalmente, en la última sección de la cueva se pueden encontrar grandes vetas de las que se puede extraer cristal, un material muy valioso y usado como ingrediente para fabricar teletransportadores. Cabe destacar también que también se pueden encontrar menas de diamantes en la última zona de la mina.

Por último, cerca de las granjas, al sur del pueblo, hay una pequeña playa en la que se pueden encontrar objetos enterrados que ha arrastrado la corriente. El jugador puede

excavar en los montones de arena que se hallan en la playa utilizando una pala. En la arena se pueden encontrar objetos de tipos muy distintos, siendo posible encontrar desde semillas hasta minerales.

4.1.9 Pesca

Existen varias masas de agua en el mapa del juego en las que el jugador puede pescar peces de distintos tipos. Son las siguientes:

- **Mar**: es posible pescar en la costa que hay al sur del pueblo; pero solo desde la playa. En el mar se pueden encontrar anchoas, bacalaos y atunes.
- **Río bajo**: estas aguas son las que comprenden desde la cascada sur del pueblo, que funciona como desembocadura del río en el mar, hasta la cascada que hay al lado de las escaleras que llevan a la montaña, de forma que el río cruza el pueblo y pasa entre las granjas. Esta parte del río alberga truchas, sábalos y salmones.
- **Río alto**: es la sección del río que nace en el lago de la montaña y termina en la cascada del norte del pueblo. Aquí habitan esturiones, siluros y carpas.
- **Cueva**: en la zona más profunda de la mina hay un estanque en el que habitan peces poco comunes y más difíciles de capturar que el resto. Estos son el gobio luminoso y la carpa espectral.

Cabe destacar que la mayoría de los peces solo se pueden obtener pescándolos directamente en su hábitat natural, dado que los únicos peces que hay a la venta en la tienda de alimentos son la anchoa, el salmón y la carpa.

4.1.10 Fabricación y cocina

El jugador tiene a su disposición dos mesas en las que puede utilizar diferentes materiales para crear objetos: una de fabricación de ítems y otra de cocina.

Por un lado, la mesa de fabricación permite al jugador fabricar todas las herramientas disponibles en el juego, incluso aquellas que no están a la venta en la tienda. Además, si tiene los materiales necesarios, puede fabricar también los picos y hachas de hierro y oro, que solo se pueden conseguir así. También es posible fabricar teletransportadores, utilizando materiales poco comunes como los diamantes, y muebles, que sí que requieren objetos abundantes como la madera.

Por otro lado, el jugador puede utilizar la mesa de cocina para preparar platos de comida que recuperan más energía y tienen más valor económico que los ingredientes por separado. Cabe destacar también que, al consumir algunos de estos alimentos, el jugador puede obtener *buffs*, que se describirán en el apartado 4.1.12.

4.1.11 Colecciones

Se ha agrupado parte de los objetos en varias colecciones cuyo propósito es similar al de los museos de la franquicia Animal Crossing. Si bien en los juegos de dicha saga el jugador tiene que completar las salas del museo con peces, bichos, fósiles y obras de arte, en el juego de este proyecto se debe intentar vender al menos una unidad de cada objeto de las colecciones. Así, al finalizar el día, se comprobarán las que el jugador ha completado, obteniendo dinero por cada una. En total, el juego tiene registradas 5 colecciones, que son:

- **Productos animales**: leche, queso, yogur, huevo, mayonesa, lana y tela.
- **Platos de cocina**: batido, pimientos rellenos, ensalada de atún y macedonia.
- **Cultivos**: papa, sandía, zanahoria, calabaza, fresa, frambuesa, pimiento, cebolla, manzana, cereza y mango.

- **Peces**: trucha, salmón, sábalo, anchoa, atún, bacalao, esturión, siluro, carpa, carpa espectral y gobio luminoso.
- **Muebles**: sofá, silla y frutero.
- **Materiales**: madera, piedra, hierro, oro, cristal, diamante, hierba y rama.

4.1.12 *Buffs*

Como se mencionó en el apartado 4.1.10, el jugador puede obtener *buffs* al consumir algunos platos de comida. Un *buff* es un efecto temporal beneficioso que permite al jugador realizar ciertas actividades de forma más efectiva mientras dure su efecto. Los *buffs* implementados son:

- **Pies ligeros**: permiten al jugador moverse más rápido.
- **Cebo resistente**: duplica el tiempo que tarda el pez en escapar una vez ha mordido el anzuelo.
- **Atractor de peces**: reduce a la mitad el tiempo máximo que tardan los peces en picar.

4.2 Implementación

Una vez definidas las mecánicas principales del juego, el siguiente paso es explicar la manera en la que se han implementado los elementos más importantes del gameplay.

4.2.1 Jugador

Dado que el jugador puede realizar muchas acciones diferentes, y todas pueden generar efectos que modifiquen su estado interno, el GameObject que controla el jugador tiene asignados una serie de componentes, cada uno encargado de una tarea. Estos son:

- **`PlayerController`**: como su nombre indica, este componente se encarga de procesar los inputs y variables que controlan el movimiento del jugador.
- **`PlayerInteraction`**: la función de este componente es comprobar si el jugador indica que quiere realizar una interacción. Concretamente, pulsando la tecla “T” se puede utilizar el objeto equipado, decidiendo si se interactúa con algún elemento del escenario en función del tipo de objeto; mientras que la tecla “I” permite interactuar con algo sin emplear un objeto. Algunos ejemplos de interacciones son talar un árbol, abrir el arcón de ventas, y comer, siendo esta última un caso de utilización de un objeto sin interacción. Este script se comentará con más detalle en el apartado 4.2.4.
- **`PlayerState`**: este script controla si el jugador puede moverse, interactuar, o abrir el inventario en un momento concreto.
- **`ItemUser`**: el propósito de este componente es conectar otros scripts del jugador cuando el uso de un objeto lo requiere. Un ejemplo de esto es la consumición de platos de cocina, que requiere comprobar si el alimento puede aplicar *buffs*, e indicar al `BuffManager` que debe añadirlos si es así. Además, también se encarga de controlar la reducción de energía al usar herramientas, entre otras funciones.
- **`InventoryManager`**: como su nombre indica, este script se encarga de gestionar todas las operaciones que implican al inventario del jugador, incluyendo la adición y sustracción de objetos, y otras funciones que se detallarán cuando se hable del papel de este script en la IA.
- **`InventoryController`**: la función principal de este componente es controlar la apertura del HUD del inventario del jugador. Concretamente, la tecla T se utiliza tanto para abrirlo como para cerrarlo.

- **`ToolbarController`**: este script gestiona los inputs del jugador relacionados con la barra de herramientas para indicar al componente `ToolbarPanel` de la HUD de la misma cómo tiene que modificarse. Para hacer esto, en la clase `ToolbarController` se han definido eventos mediante Actions de Unity, de forma que `ToolbarPanel` se suscribe a dichos eventos y reacciona a ellos cuando se disparan. Esto se explicará con más detalle en el apartado 4.2.3.
- **`EnergyManager`**: el propósito de este script es manejar las operaciones relacionadas con la energía del jugador, tanto la adición y reducción de la misma, como la actualización del texto del HUD que indica la energía actual.
- **`MoneyManager`**: de manera similar al componente anterior, este controla todo lo relacionado con el dinero del jugador, es decir, las operaciones para añadir y restar dinero, y el texto del HUD que muestra los fondos del jugador.
- **`PlayerUIManager`**: este componente funciona como una interfaz para que otros scripts puedan activar o desactivar las partes del HUD del jugador que necesiten.
- **`FishingManager`**: la función de este script es manejar todas las operaciones relacionadas con la pesca, las cuales se explicarán más adelante, en el apartado 4.2.11.
- **`BuffManager`**: como su nombre indica, este componente almacena la información de todos los buffs aplicados al jugador en cada momento, así como el tiempo durante el cual cada uno surtirá efecto. Además, provee de una interfaz para que se puedan activar los buffs nuevos adquiridos al comer.
- **`CraftingManager`**: este componente almacena dos listas de recetas, una para las de fabricación de objetos, y otra para las de cocina, y permite fabricar objetos de cualquiera de ellas.
- **`PlayerStableManager`**: la función de este componente es controlar el estado actual del estado del jugador, así como gestionar la adición y extracción de animales del mismo. Estas acciones se llevan a cabo teniendo en cuenta la capacidad del establo, que es almacenada como propiedad en esta clase.
- **`CollectionManager`**: este script maneja las operaciones relacionadas con las colecciones de objetos vendidos por el jugador, además de almacenar un diccionario para cada una en la que se registran los objetos vendidos de la colección.

La Figura 1 muestra el HUD básico del juego, en la que aparecen la energía actual del jugador y el dinero que posee, arriba a la izquierda; y la hora actual, arriba a la derecha. La barra de herramientas, que aparece más abajo, se explicará en el apartado 4.2.3.

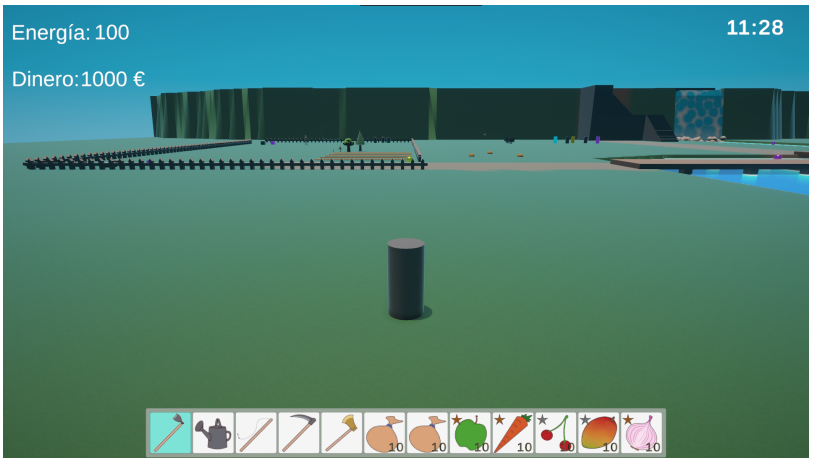


Figura 1: Captura de la interfaz del juego

4.2.2 Objetos

Se ha desarrollado una jerarquía de clases para almacenar los datos de los diferentes tipos de objetos del juego. La clase base, `Item`, contiene los datos que todos los ítems debe tener, como el nombre, los precios de compra y venta, y si se pueden apilar en el inventario, entre otros.

Inicialmente, ninguno de los objetos era instanciable, porque todos sus datos se almacenaban en `ScriptableObjects`, que son conjuntos de datos que se guardan como assets de Unity, de forma que solo existe una copia de ellos en memoria. Sin embargo, esta aproximación forzaba a que todos los objetos del mismo tipo tuvieran las mismas propiedades, por lo que se modificó para posibilitar que determinados objetos pudieran tener atributos diferentes, aunque fueran del mismo tipo. Así, se creó la clase base `ExtendedItem` y dos clases hijas, `ItemWithDurability` e `ItemWithQuality`. Los `ExtendedItem` tienen como propiedad pública el `ScriptableObject` con los datos generales del objeto, y cada clase hija puede añadir las propiedades nuevas que necesite. Concretamente, `ItemWithDurability` añadió la propiedad y los métodos necesarios para crear herramientas con durabilidad, mientras que `ItemWithQuality` hizo lo mismo para los alimentos con diferentes niveles de calidad. El diagrama de clases de ExtendedItem se puede ver en el Apéndice 8.1.

La clase base mencionada al principio de este apartado es la que contiene los datos que poseen todos los `ScriptableObjects` de los objetos. En lo que respecta a sus clases hija, las más destacables son:

- **`Consumable`**: añade dos atributos, que son la energía restaurada al consumir el alimento, y la lista de *buffs* que se aplican al hacerlo.
- **`Food`**: es una clase hija de `Consumable` y fue creada para los platos de comida, almacenando la categoría de plato a la que pertenece el objeto.
- **`Fish`**: al igual que `Food`, hereda de `Consumable` y posee dos atributos, que son el hábitat del pez y los segundos que tarda el pez en escapar tras morder el anzuelo.
- **`MachineItem`**: estos objetos simplemente almacenan el Prefab de la máquina a la que representan.
- **`Seed`**: tal y como se explicó en el apartado 4.1.5, la única propiedad que añaden las semillas es el `ScriptableObject` con los datos de la planta que generan.
- **`Tool`**: las herramientas añaden su duración máxima, la energía que consumen con su uso, un booleano para indicar si la herramienta se desgasta con cada uso independientemente de que acierte en algún objetivo que pueda reaccionar a ella, la valoración, y otro parámetro más. Las herramientas funcionan de una forma diferente al resto de objetos, dado que cada tipo de utensilio tiene una clase hija de `Tool` asociada. Esto se hizo así por comodidad, dado que, de esta forma, se puede comprobar si un objeto es de un tipo concreto simplemente analizando la clase a la que pertenece. Sin embargo, al desarrollar la IA comenzó a ser necesario realizar estas comparaciones utilizando strings, por lo que se añadió el tipo de herramienta como parámetro adicional. Queda pendiente buscar una forma de que esto no sea necesario; pero no se ha observado ningún impacto importante en el rendimiento del juego, así que no es un problema que requiera de solución urgente. Las herramientas también poseen un método adicional llamado `GetTargets`, que permite obtener una lista de objetos interactuables en base a las características de la herramienta. Un ejemplo claro de esto es la guadaña, pues en

un solo uso puede segar varias matas de hierba; mientras que un pico no puede romper varias piedras a la vez.

- **`AnimalItem`**: si bien los animales no son objetos que se puedan almacenar en el inventario, para facilitar la implementación de la tienda de animales se creó una clase hija de `Item` que guardara datos como los precios de compra y venta, así como el `ScriptableObject` con la información propia del animal, tal y como se explicará en el apartado 4.2.9.
- **`WarpItem`**: estos objetos almacenan un `Vector3` con la posición a la que el jugador se teletransportará al usarlos, y definen un método `WarpPlayer` para realizar esta función.

El diagrama de clases completo de `Item` está en el Apéndice 8.2. En él aparecen clases que no se han mencionado en la enumeración anterior, dado que son clases vacías que existen por si acaso, en un futuro, es necesario añadirles propiedades y métodos. Son:

- **`CraftingMaterial`**: es la clase propia de los materiales de fabricación, como la madera y los minerales. Hereda directamente de `Item`.
- **`Generic`**: es una clase hija de `Item` definida para objetos genéricos.
- **`Placeable`**: esta clase está asociada a los objetos colocables en el mundo. La clase `MachineItem` hereda de esta clase.
- **`FurnitureItem`**: es una clase utilizada para representar los muebles. Al igual que `MachineItem`, hereda de `Placeable`.

Antes de finalizar este apartado, es importante destacar que la calidad de los objetos `ItemWithQuality` influye en la energía que restauran. Así, la fórmula utilizada es `Ceil(energía_restaurada_base * ((calidad + 2) / 2))`, teniendo en cuenta que la calidad se representa internamente como un entero en el que el nivel de calidad básica es 0.

4.2.3 Inventario y barra de herramientas

En la Figura 2 se puede ver la interfaz del inventario del jugador.

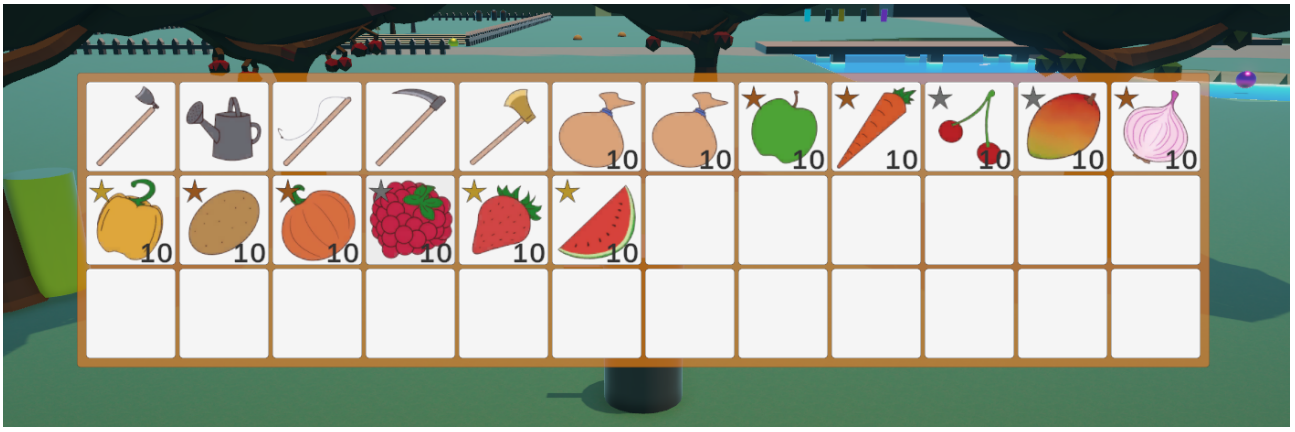


Figura 2: HUD del inventario

Tal y como se puede observar en la figura anterior, el inventario está compuesto por una lista de 36 espacios. Internamente, es una estructura de datos llamada `ItemList` formada por 36 instancias de una clase llamada `ItemSlot`, la cual almacena un `ExtendedItem` y un contador del número de unidades del mismo que hay en el hueco. Utilizando el ratón, el jugador puede hacer clic izquierdo en un objeto para colocarlo en el ratón y moverlo a otro lugar o intercambiarlo con el ítem de otra ranura; mientras que, con

un clic derecho, puede añadir objetos al espacio del ratón de uno en uno.

Por otro lado, en la Figura 3 aparece la barra de herramientas del juego, que muestra una de las filas del inventario y el objeto equipado en cada momento, el cual aparece marcado en azul. Además, es posible cambiar la fila del inventario mostrada en la barra con las teclas “R” y “F”, que permiten seleccionar las filas encima y debajo de la actual, respectivamente. Para cambiar el objeto equipado, se puede seleccionar con el ratón pulsándolo directamente en la barra de herramientas; pero las teclas “Z” y “X” permiten seleccionar como objeto equipado el que está a la izquierda y derecha del actual, respectivamente.

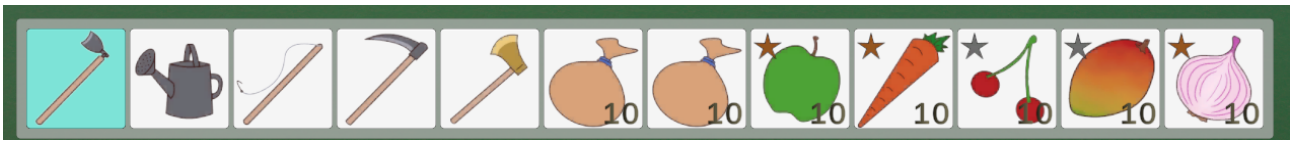


Figura 3: HUD de la barra de herramientas

Cada una de las ranuras del inventario está representada por un botón llamado `ItemSlotButton`, que contiene los iconos del objeto y de la estrella que indica su calidad (la estrella puede ser de bronce, plata u oro en orden creciente de calidad), el número de unidades en el hueco, y el marcador azul del objeto equipado. Tal y como se puede ver en la Figura 3, dependiendo del tipo de objeto se muestran o no estos elementos. Por ejemplo, las herramientas no se pueden apilar en un slot, así que no tiene sentido mostrar el número de objetos del mismo, y tampoco es necesario tener activa la estrella de la calidad, dado que es una característica propia de los alimentos.

La clase `ItemSlotButton` utiliza una herramienta de Unity muy útil que son las interfaces de ratón `IPointerClickHandler`, `IPointerEnterHandler`, e `IPointerExitHandler`, las cuales permiten definir eventos que se lanzan cuando el jugador pulsa el botón, y cuando el ratón colisiona y sale del mismo, respectivamente. Sin embargo, el inventario no se muestra únicamente cuando el jugador pulsa la tecla “M”, sino que también aparece en otras situaciones, como en el HUD de la venta de objetos. Así, se han implementado los diferentes tipos de paneles de inventario mediante la siguiente jerarquía de clases, cuyo diagrama está en el Apéndice 8.3.

- **`ItemPanel`**: es la clase base de todas las demás. Contiene la implementación básica del método `DisplayPanel`, que permite mostrar en el HUD los objetos del inventario, así como los métodos virtuales `OnButtonClick`, `OnButtonEnter` y `OnButtonExit`. La clase de cada tipo de HUD de inventario sobreescribirá estos métodos con el código que deben ejecutar los `ItemSlotButton` en cada caso.
- **`InventoryPanel`**: esta clase hija de `ItemPanel` corresponde a la interfaz de inventario más básica, que es la que aparece en la Figura 2 y se explicó anteriormente. En este HUD solo es necesario implementar `OnButtonClick`, dado que los botones no reaccionan al pasar el ratón por encima.
- **`ToolbarPanel`**: el HUD implementado por esta clase es el de la barra de herramientas, por lo que debe guardar el índice de la fila del inventario mostrada y el índice en esa fila del objeto equipado. Tal y como se mencionó en el apartado 4.2.1, esta clase está suscrita a dos eventos de `ToolbarController`, a los que reacciona utilizando los métodos `ChangeToolbarRow` y `ChangeEquippedItem`. Estas funciones permiten, como indican sus nombres, cambiar la fila del inventario que se muestra en la barra de herramientas y el índice del objeto equipado, respectivamente. Cabe destacar también que `ToolbarPanel` también implementa `OnButtonClick`, aunque, en este caso, solo fija el índice del objeto

equipado al del botón pulsado. También sobrescribe el método `DisplayPanel` para mostrar en la barra de herramientas solo los objetos que están en la fila actual.

- **`StaticInventoryPanel`**: este tipo de HUD de inventario se caracteriza por ser ignorar el input del jugador, dado que se utiliza en situaciones en las que el propósito de mostrar el inventario no va más allá de ver los objetos que tiene el jugador. Por este motivo, no sobrescribe ningún método de los eventos de ratón de `ItemPanel`. Esta HUD de inventario se utiliza en las interfaces de las tiendas de objetos y de las mesas de fabricación y cocina.
- **`SaleInventoryPanel`**: finalmente, esta HUD de inventario fue creada exclusivamente para la pantalla de venta de objetos, dado que debe interactuar con elementos que solo aparecen allí. Por eso, se explicará con más detalle en el apartado 4.2.7.

4.2.4 Interacción del jugador con otros GameObjects

El diseño de la interacción entre el jugador con los elementos del mundo en un videojuego puede ser un proceso costoso, dado que es necesario procurar que el sistema diseñado sea modular y adaptable a posibles nuevos GameObjects interactuables. Por eso, en su momento se invirtió tiempo en pensar detenidamente cómo implementar un sistema de interacción que permitiera a las nuevas mecánicas introducidas adaptarse a él con facilidad.

Hay que partir de la base de que el jugador tiene dos formas básicas de interacción, siendo estas la interacción usando el objeto equipado y sin utilizar objetos, utilizando las teclas “T” e “I”, respectivamente. Por ello, se decidió crear una clase Interactable que solo cuenta con dos métodos virtuales vacíos, `InteractWithItem` e `InteractWithoutItem`. Para cada tipo nuevo de interacción es necesario hacer una clase hija que implemente al menos uno de dichos métodos, de forma que es el propio destinatario de la acción del jugador quien decide cómo reaccionar al estímulo iniciado por este.

Por otro lado, como ya se ha mencionado anteriormente, el jugador utiliza el script `PlayerInteraction` para comenzar la interacción con los elementos del escenario. Así, para cada tipo de interacción se sigue un comportamiento distinto.

En primer lugar, si no se utiliza ningún objeto, se lanza un Raycast desde una posición por encima y delante del jugador, tal y como muestra la Figura 4. Si el objeto que colisiona con ese Raycast tiene un componente de tipo `Interactable`, es decir, un script que hereda de dicha clase, se ejecuta su implementación del método `InteractWithoutItem`. Esto quiere decir que, si el GameObject con el que se interactúa no lo implementa, no ocurrirá nada.

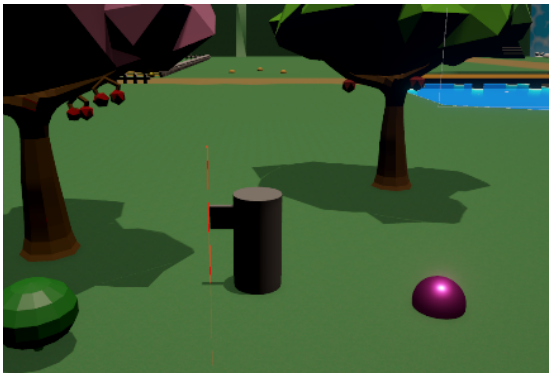


Figura 4: Raycast (en rojo) lanzado para detectar elementos con los que interactuar

En segundo lugar, si se va a utilizar el objeto equipado, se verifica antes si el espacio del inventario seleccionado está vacío o no, pues, si lo está, no se puede continuar. Si, efectivamente, hay un objeto, se comprueba si este es una herramienta y, si es así, se debe comprobar si el jugador tiene la energía suficiente como para utilizarla. En caso de que no sea así, no ocurrirá nada; pero si sí es posible usar la herramienta, se utilizará el método `GetTargets` de la misma, mencionado antes, para obtener los posibles afectados por su uso. Así, el jugador intentará interactuar con todos utilizando la herramienta. Finalmente, si el utensilio pierde resistencia con cada uso en lugar de con cada acierto en un elemento con el que puede interactuar, se reduce su durabilidad. Por otro lado, si el objeto equipado no es una herramienta, se lanza un Raycast como ocurría al interactuar sin usar ningún ítem, y, si se colisiona con algún GameObject con un componente `Interactable`, se ejecuta su `InteractWithItem`. Sin embargo, cabe destacar que, en el caso de que el Raycast no choque con ningún elemento interactivo o no choque con nada, se usará el método de `ItemUser` llamado `UseItemWithoutTarget`. Esta función permite consumir los teletransportadores y los alimentos.

4.2.5 Compra de objetos

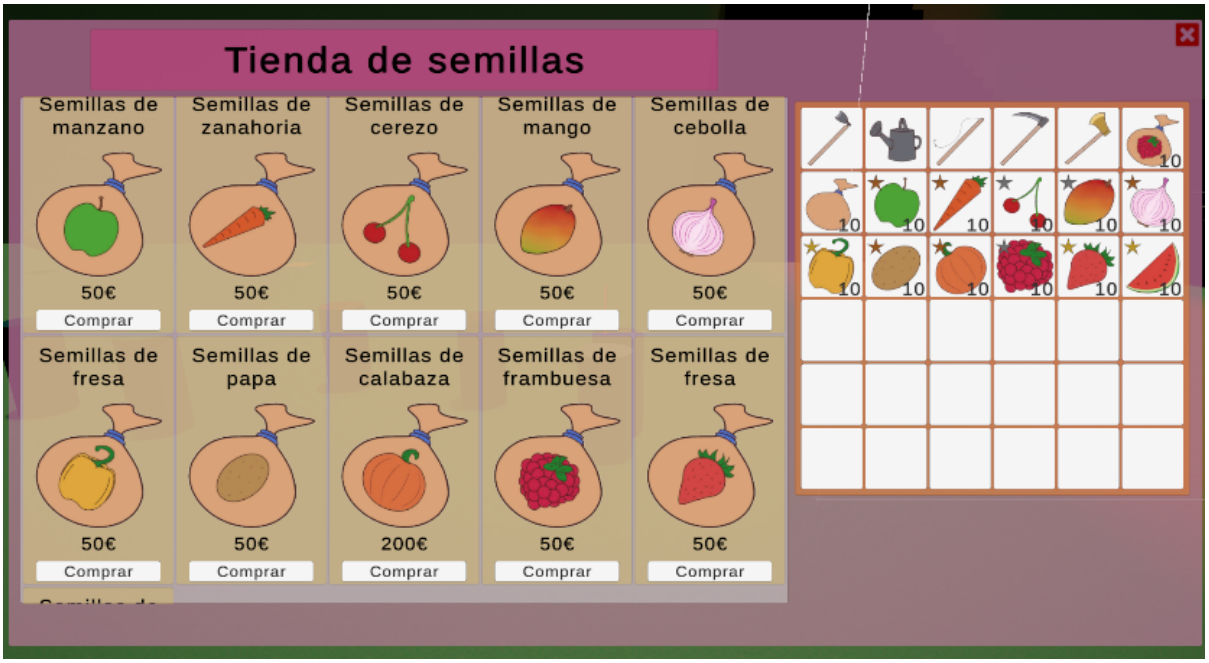


Figura 5: Interfaz de la tienda de semillas

En la Figura 5 se puede ver un ejemplo del HUD de las tiendas de objetos, en este caso, de la tienda de semillas. A la derecha aparece un panel de inventario que, como ya se mencionó anteriormente, es un `StaticInventoryPanel`, por lo que no se puede interactuar con él y solo sirve para que el jugador pueda ver cómo cambia el estado del inventario según compra objetos. Lo más destacable del HUD de la figura es el bloque que aparece a la izquierda, que corresponde a la lista de artículos de la tienda. La disposición en mosaico de las tarjetas con la información de los artículos a la venta se ha conseguido añadiendo un componente `GridLayoutGroup` al panel gris que se puede ver debajo de las tarjetas. Este componente permite organizar a los hijos de un GameObject en un Canvas siguiendo la forma de un mosaico en la que todas las piezas tienen el mismo tamaño y dejan un espacio determinado entre ellas y con los bordes del panel. Así, se ha ajustado el tamaño de las tarjetas de forma que quepan 5 por fila.

Además, el panel gris se ha colocado como hijo de un elemento adicional e invisible con dos componentes especiales: `Mask` y `ScrollRect`. Mediante el primero, es posible hacer que solo se vean los hijos del GameObject que lo lleva si estos quedan por delante de él. Por otro lado, el segundo permite mover un GameObject, ajustando previamente los ejes en los que puede desplazarse, la velocidad, etc. Combinando las funciones de estos componentes, se ha logrado que muestre solo parte de la lista de artículos disponibles, pudiendo ver el resto haciendo clic y arrastrando la lista con el ratón, o utilizando la rueda del mismo.

Por otro lado, dentro del juego, las tiendas son GameObjects interactivos. Para abrir el HUD de la Figura 5, el jugador debe interactuar con una tienda de objetos, lo que hará que el componente `ShopManager` de la misma se comunique con el script `PlayerUIManager` del jugador para abrir el HUD. Además, cada tienda tiene asociado un nombre, un `ScriptableObject` con la lista de objetos que tiene disponibles para vender, y un valor de un `Enum` llamado `ShopType`, que indica si la tienda es de objetos o animales. Así, el `ShopManager` de la tienda se comunica con el script que controla el HUD de la tienda, llamado `ShopPanel`, para cargar la lista de artículos, el inventario del jugador, y el nombre de la tienda en la interfaz. Un detalle importante es que, cuando alguien interactúa con la tienda y abre su HUD, el `ShopManager` guarda una referencia al jugador como el cliente actual. Esto le permite controlar que no puedan acceder varios jugadores a la tienda a la vez, y también le posibilita tener una referencia al GameObject con los componentes `MoneyManager`, `InventoryManager`, etc., con los que tiene que interactuar para efectuar las compras. También cabe destacar que `ShopPanel` es una clase madre y que, en realidad, la clase que se utiliza en el HUD de las tiendas de objetos se llama `ItemShopPanel`. Sin embargo, esta solo sirve para actualizar correctamente el inventario mostrado en el HUD. La otra clase hija de `ShopPanel`, `AnimalShopPanel`, se verá en el siguiente apartado. El diagrama de clases de `ShopPanel` se puede ver en el Apéndice 8.4.

Una vez el juego muestra al jugador el HUD de la tienda, este puede cerrarla con el botón X de arriba a la derecha, o comprar un objeto pulsando el botón Comprar de la tarjeta correspondiente. Cuando ocurre lo segundo, el botón indica al `ShopPanel` que el jugador quiere comprar el objeto de su tarjeta, y, a su vez, el `ShopPanel` pide al `ShopManager` que intente realizar la compra. Si todo sale bien, el `StaticInventoryPanel` de la derecha del HUD se actualiza automáticamente.

4.2.6 Compra de animales

Figura 6: Interfaz de la tienda de animales

En la Figura 6 se muestra el HUD de la tienda de animales, que, si bien es muy diferente del de las tiendas de objetos, su funcionamiento interno es bastante similar. En este caso, la interfaz es gestionada por la clase `AnimalShopPanel`, que es hija de `ShopPanel`. Sin embargo, hay algunos cambios importantes con respecto a cómo funcionaba la tienda de objetos.

En primer lugar, el cambio más evidente es la sustitución del inventario por una lista de los animales presentes en el establo, en la que cada animal se muestra junto a su valor monetario y un botón para venderlo. Internamente, al pulsar dicho botón, este indica al `AnimalShopPanel` que se debe vender el animal asociado a él, y este añade el dinero correspondiente y saca al animal del establo por medio del `MoneyManager` y del `PlayerStableManager` del jugador cliente de la tienda.

En segundo lugar, si bien este tipo de tiendas también utilizan un componente `ShopManager`, este está preparado adecuadamente tanto para cargar el HUD de la tienda de animales y no el de las de objetos, como para añadir los animales comprados al establo y no al inventario. Recordemos que, según se comentó en el apartado 4.2.2, la tienda de animales guardaba la información de estos como si fueran objetos.

Por último, debajo de la lista de animales del establo se encuentra un indicador del espacio disponible en este, que se actualiza dinámicamente según el jugador compra y vende animales en la tienda.

4.2.7 Venta de objetos

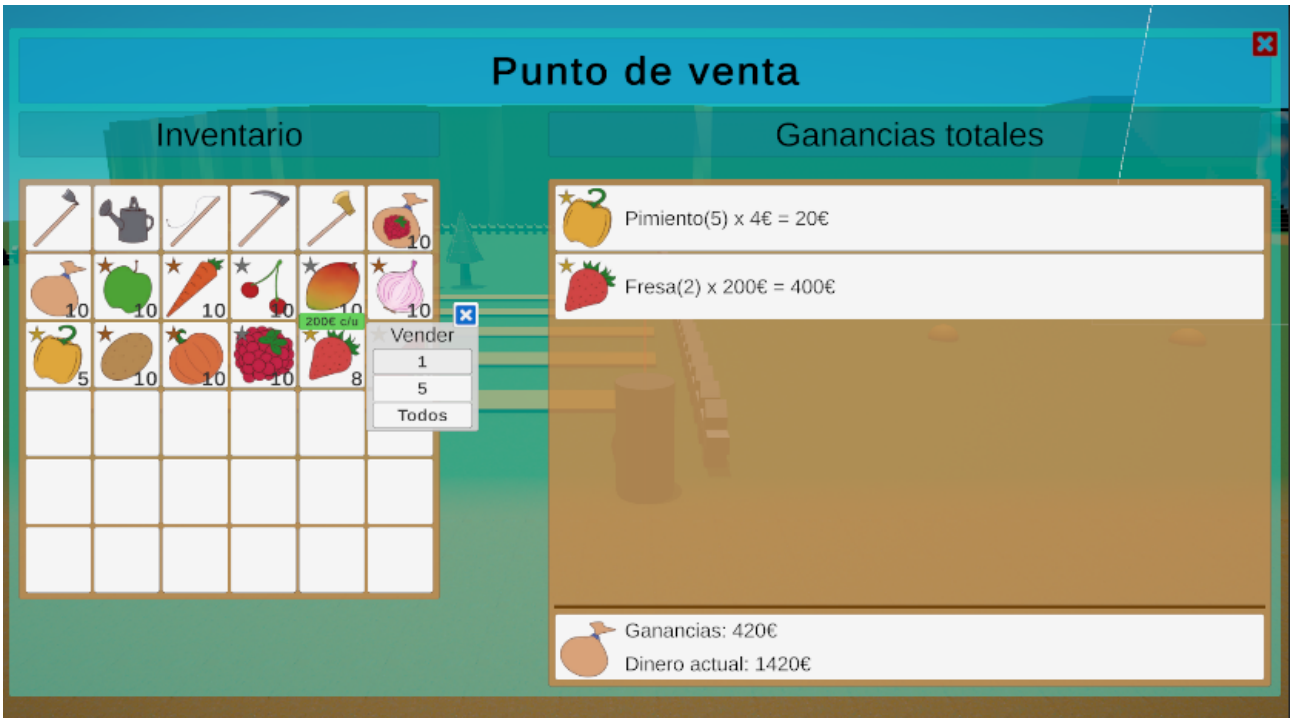


Figura 7: HUD de los puntos de venta

Como se puede ver en la Figura 7, la interfaz que utiliza el jugador para vender objetos tiene dos grandes elementos principales: el panel del inventario y la lista de objetos vendidos. Sin embargo, el HUD en general es administrado por la clase `SalePanel`; mientras que las ventas son manejadas por la clase `SaleManager`. De manera análoga a lo que ocurría con las tiendas, es posible interactuar con un punto de venta y, cuando esto ocurre, el `SaleManager` se comunica con el `PlayerUIManager` del jugador para abrir

el HUD del punto de venta con el `SalePanel`.

El panel del inventario de esta interfaz es bastante diferente de los que se han visto hasta ahora. La clase que lo implementa, `SaleInventoryPanel`, sobrescribe los tres métodos virtuales de `ItemPanel`: `OnButtonClick`, `OnButtonEnter` y `OnButtonExit`. Para empezar, cuando se pulsa el icono de un objeto del inventario, se muestra el menú pequeño que aparece en la Figura 7, y que el jugador puede utilizar para indicar la cantidad de objetos que quiere vender. El botón pulsado indica al `SaleInventoryPanel` el objeto a vender y la cantidad exacta, y este pasa a su vez esta información al `SalePanel` para comenzar la operación de venta. Primero, avisa al `SaleManager` de que debe realizar la venta, y luego actualiza la lista de objetos vendidos. Por otro lado, cuando el ratón entra en contacto con una de las ranuras del inventario, se coloca encima de ella una etiqueta verde con el precio por unidad del objeto que contiene; mientras que, cuando el ratón deja de tocar dicho hueco, la etiqueta desaparece.

El segundo elemento principal del HUD de los puntos de venta es la lista de objetos vendidos. Para realizar una venta, el `SaleManager` utiliza el método `SellItem`, que devuelve la cantidad de unidades que se han vendido. Sabiendo el objeto que se deseaba vender y qué cantidad del mismo se ha vendido, el `SalePanel` indica esta información a otra clase, `SoldItemsList`, que actualiza tanto la lista de objetos vendidos como las ganancias obtenidas durante la sesión de venta y los fondos totales del jugador. En cada una de las tarjetas de la lista de objetos vendidos se muestran el nombre del objeto y la cantidad vendida entre paréntesis, además de su valor por unidad y el total obtenido al vender esos ítems. Además, si el objeto tiene diferentes niveles de calidad, junto al icono del mismo aparecerá la estrella que indica el nivel concreto de calidad del objeto.

Por último, cabe destacar que la calidad también influye en el precio de venta de los objetos, al igual que ocurría con la energía recuperada por los alimentos. Concretamente, para calcular el dinero obtenido por unidad teniendo en cuenta la calidad del objeto, se multiplica el valor económico base por `(calidad + 2) / 2`, y se redondea por exceso el valor obtenido. Es necesario recordar que “calidad” es un entero que indica la calidad del objeto, de forma que 0 es la más baja y valores más altos corresponden a calidades mayores. Así, un objeto de calidad mínima se venderá por su valor monetario por defecto.

4.2.8 Cultivos

Algunos de los elementos que se van a ver a partir de ahora siguen un esquema que consiste en asignarles dos componentes. Uno de ellos es un Interactable, mientras que el otro es un script que gestiona el funcionamiento interno del objeto.

Antes de comenzar a detallar cómo se comportan las plantas, es necesario hacer una distinción entre cultivos y plantas silvestres:

- **Cultivos**: son plantas que necesitan estar asociadas a una parcela, pues al plantar una semilla, la planta creada se asocia directamente como hija de la parcela en la jerarquía de Unity. Al destruirlas desaparecen.
- **Plantas silvestres**: estas plantas se encuentran fuera de la granja y crecen sin necesidad de ser regadas ni de estar sobre suelo arado. Además, al romperlas, se regeneran como si se hubiera plantado una semilla nueva justo después de destruir la planta.

El primer caso que vamos a ver de este esquema es el que siguen las plantas del juego, las cuales poseen los componentes `PlantObject` y `PlantInteractable`. El primero de ellos tiene una referencia a un `ScriptableObject` de tipo `PlantData` en

el que se guarda la información de la planta en sí, es decir, el objeto que produce, las mallas y materiales de cada una de sus etapas de crecimiento, los tiempos en frames que tarda en pasar de una etapa a otra y en producir frutos, y un booleano que indica si la planta es silvestre o no, entre otros. Utilizando esta información, en cada frame se utiliza el método `Update` de `PlantObject` para controlar que el cambio de etapa de crecimiento se produzca cuando haya pasado el número de frames indicado en el `ScriptableObject`. Si la planta ya ha crecido del todo, se utiliza el mismo método para controlar cuándo produce frutos. Sin embargo, los contadores de frames que se utilizan para ello solo crecerán si la planta está regada. Un ejemplo de las diferentes etapas de crecimiento de una planta se puede ver en la Figura 8.

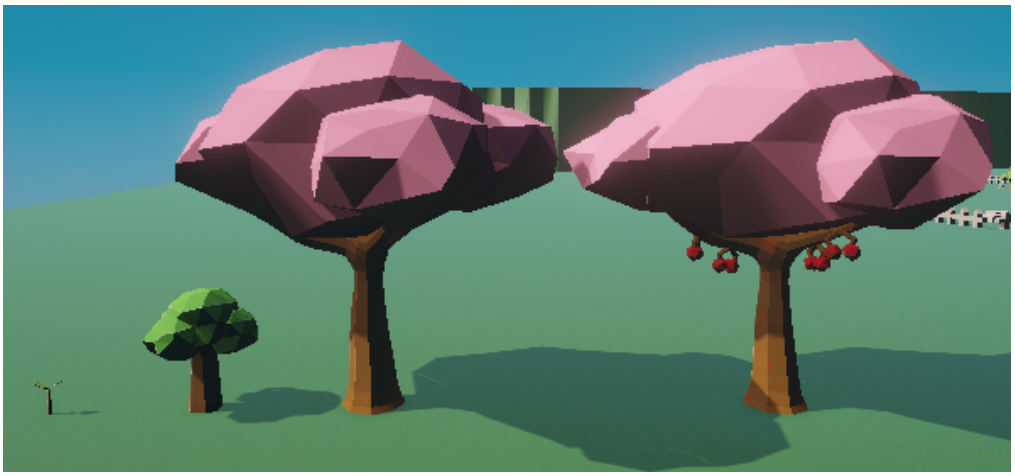


Figura 8: Etapas de crecimiento de un cerezo

Por otro lado, el jugador puede interactuar con una planta tanto utilizando un objeto como no usando ninguno. En el primer caso, la planta solo reacciona ante determinadas herramientas:

- Si se usa un hacha, se avisará a `PlantObject` de que debe reducir la resistencia restante de la planta. Si esta llega a cero, se añadirán al inventario del jugador los objetos que suelta la planta al ser talada y, tal y como se explicó antes, la planta se destruye si es un cultivo, o se regenera desde su etapa de brote si es silvestre.
- Si se usa una guadaña, primero se comprueba si la planta afectada puede cosecharse con una guadaña, según marca un booleano de `PlantData`. Si es así, se añaden al inventario del jugador los productos cosechados, y se destruye o regenera la planta según si es un cultivo o no.
- Si se usa una regadera, se indica al `GameObject` de la parcela sobre la que está la planta que debe regarse. Por supuesto, si la planta es silvestre, no ocurrirá nada.

Finalmente, se reduce la resistencia de la herramienta utilizada si es necesario hacerlo en este momento.

En cambio, si el jugador interactúa con la planta, este intentará cosechar sus productos, pero esto solo funcionará si la planta admite las cosechas sin guadaña y si tiene productos que se puedan recoger.

Para acabar, cabe destacar que las parcelas en las que se pueden plantar semillas utilizan el script `PlowableTileInteractable`, que controla si el suelo está regado y cuánto tiempo queda hasta que se seque, así como la interacción entre el jugador y la parcela. En este caso, esta solo se puede dar utilizando un objeto:

- El jugador puede utilizar una azada para arar la parcela siempre que no haya una

planta ya en ella. Además, el jugador puede usar una regadera para regarla.
- Si el objeto equipado es una semilla y la parcela está arada, la semilla es plantada, siempre y cuando el tipo de parcela sea compatible con el tipo de planta de la semilla, dado que no se pueden plantar árboles en suelos que no los admitan, y tampoco es posible plantar plantas pequeñas en parcelas que solo admiten árboles. Si la semilla se puede plantar, se instancia el Prefab de una planta genérica y se cargan los datos de la planta concreta.

4.2.9 Animales

El funcionamiento interno de los animales del juego también sigue el esquema visto con las plantas, pues se realiza comunicando un `Interactable` con otro script que gestiona el estado interno del animal. Sin embargo, en este caso, se ha definido una clase base `AnimalInteractable` de la que heredan `CowInteractable`, `SheepInteractable` y `ChickenInteractable`, de forma que cada animal puede definir cómo quiere interactuar con el jugador. El diagrama de clases de `AnimalInteractable` está en el Apéndice 8.5.

- **`CowInteractable`** es el script que gestiona la interacción con las vacas, que puede realizarse usando o no un objeto. Así, se puede utilizar una cubeta lechera para intentar ordeñar la vaca, o es posible alimentarla con un objeto `Consumable`. En este último caso se utiliza el único método que implementa `AnimalInteractable`, llamado `FeedAnimal`, el cual se explicará en detalle luego.
- **`SheepInteractable`** implementa la interacción con las ovejas, que es prácticamente idéntica a la anterior, con la salvedad de que, para obtener sus productos, es necesario utilizar unas tijeras de esquilar.
- Por último, **`ChickenInteractable`** se encarga de manejar la interacción con las gallinas, que es un poco diferente a las dos anteriores. Esto se debe, principalmente, a que para recoger los huevos no es necesario usar ninguna herramienta, por lo que se usa el método `InteractWithoutItem`. Así, `InteractWithItem` solo se emplea para alimentar a la gallina si el objeto equipado es un `Consumable`.

El otro script mencionado al principio de este apartado es `AnimalObject`, que sí es común a todos los animales. Además, guarda una referencia a un `ScriptableObject` de tipo `AnimalData`, que contiene la información básica del animal, como el objeto que producen, el número de frames que tardan en generarlo, el alimento que les gusta y el que odian, el espacio que ocupan en el establo, y el Prefab asociado al animal. Así, aunque no tengan etapas de crecimiento, en el `Update` de `AnimalObject` se actualiza un contador de frames hasta que llega al límite indicado en el `AnimalData`, momento a partir del cual se puede recoger el producto del animal.

Sin embargo, es importante resaltar el papel de la afinidad entre el jugador y el animal. Cada animal tiene un cierto nivel de afinidad con el jugador, que comienza en 0 y está limitado por una variable de `AnimalData`. La afinidad aumenta cuando se alimenta al animal lo suficiente mediante el método `FeedAnimal` de `AnimalInteractable`. Cuando se ejecuta esta función para alimentar a un animal, se guarda un entero con el incremento de afinidad basado únicamente en si al animal le gusta o no el objeto. Así, si el alimento es su favorito, la afinidad aumentará en 2; mientras que si el animal odia el objeto, la afinidad se reducirá en 1. En cualquier otro caso, la afinidad aumenta en uno. Además, si el objeto dado al animal es un `ItemWithQuality`, se suma al valor anterior su calidad. Por ejemplo, la calidad no se reduciría si se le da al animal la comida que no le

gusta, siempre y cuando no sea de la calidad más baja, cuyo valor interno es 0.

Así, la afinidad del animal influye en dos cálculos. El primero se localiza en el método `GetProduct` de `AnimalObject`, que es el que se utiliza para recoger el producto del animal si es posible y si el jugador tiene espacio en el inventario. Concretamente, la cantidad de unidades añadidas del producto es un número aleatorio entre 1 y el nivel de afinidad + 1. El segundo es dentro del `Update` ya explicado de `AnimalObject`, pues en cada frame, además de incrementar el contador, se le suma el nivel de afinidad, de forma que los animales que le tienen más cariño al jugador producen objetos más rápido.

4.2.10 Minería y búsqueda de tesoros

Debido a que ya se explicó cómo funciona la tala en el apartado 4.2.8, y a que la minería y la búsqueda de tesoros en la playa funcionan prácticamente igual, se explicarán juntas en este apartado.

En primer lugar, se detallará el funcionamiento de la minería. Para acceder a la mina, que está en una parte de la escena diferente al resto del escenario, el jugador debe dirigirse a un punto concreto, que en la Figura 9 está representado por un arco. Al entrar en contacto con el collider del interior del arco, el jugador es transportado directamente al interior de la mina, en la que hay otro punto similar que coloca al jugador enfrente del arco.

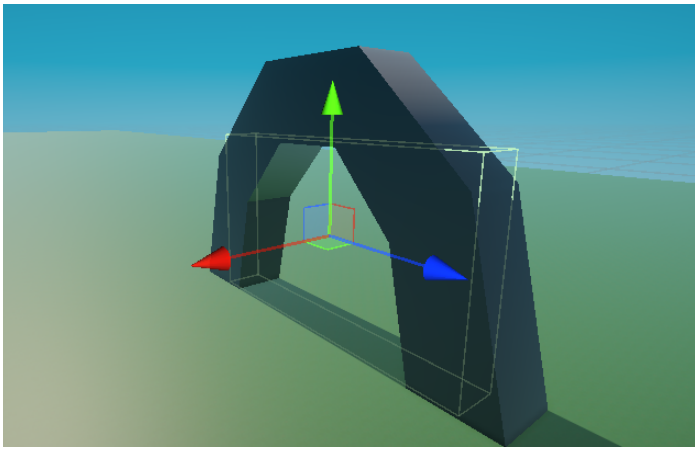


Figura 9. `Collider` de la entrada de la mina

El componente que controla el comportamiento de las menas de minerales se llama `OreObject`, y posee una referencia a una lista de `OreData`. Este es un `ScriptableObject` con los datos de la mena, es decir, su resistencia, malla, material, y dos listas, una con los materiales que se pueden obtener de ella, y otra con el número máximo de unidades que se pueden extraer por cada uno. El motivo por el que se almacena una lista de `OreData` en lugar de uno solo es que, tanto al empezar el juego, en el método `Awake` de `OreObject`, como en otras circunstancias, la mena pueda seleccionar aleatoriamente un `OreData` y utilizar su malla, material, etc., para añadir variedad. El otro momento en el que se pueden cambiar los datos de la mena es cuando esta se regenera, pues, al romperse, las menas reaparecen pasado un tiempo. Así, el `Update` de `OreObject` se encarga, principalmente, de llevar a cabo la regeneración. Para ello, en cada frame se actualiza un contador, regenerando la mena cuando este llega a un límite.

En cuanto al script `OreInteractable`, este solo permite al jugador interactuar con las menas si se está utilizando un pico. Así, si el jugador pulsa la tecla “T” teniendo equipado un pico, `OreInteractable` comprueba si la mena es vulnerable, pues no se ven

afectadas por los golpes del jugador mientras se están regenerando. Si la mena es vulnerable, se llama al método `ReceiveHit` de `OreObject`, que se encarga de reducir la resistencia de la mena en función de la potencia del pico utilizado, y, si esta llega a cero, de añadir al inventario del jugador los objetos obtenidos y de indicar que debe empezar a regenerarse.

En segundo lugar, los montones de arena que se encuentran en la playa funcionan de una forma muy similar a las menas de minerales, pero presentan algunas diferencias. Para empezar, tienen componentes `DirtPileInteractable` y `DirtPileObject`, como el resto de recursos naturales vistos hasta ahora. Sin embargo, no existe una clase `DirtPileData`, dado que en `DirtPileObject` ya se almacena una lista con los objetos que se pueden encontrar al excavar en el montón de arena, así que no es necesario un `ScriptableObject` que indique lo que contiene cada uno. Por tanto, en el método `Awake` de `DirtPileObject` se asigna un tesoro aleatoriamente, y lo mismo ocurre en el `Update`, porque los montones se regeneran como las menas.

El otro script, `DirtPileInteractable`, realiza prácticamente las mismas funciones que `OreInteractable`, con la salvedad de que el jugador necesita utilizar una pala para interactuar con la pila de arena. Al hacer esto, se llama al método `ReceiveHit` de `DirtPileObject`, que se encarga de reducir la durabilidad del montón de arena y de avisar al `InventoryManager` del jugador de que debe añadir el tesoro encontrado, siempre y cuando la resistencia del montón de arena llegue a cero.

4.2.11 Pesca

En el juego, la pesca funciona de forma algo diferente a las otras maneras de extraer recursos. Concretamente, cada masa de agua del juego tiene dos componentes, `WaterBodyInteractable` y `WaterBodyObject`, de forma que el primero se encarga de manejar la interacción entre el jugador y el agua, y el segundo gestiona el pez que muerde el anzuelo. Concretamente, el segundo script posee tres listas de peces, y cada una almacena los peces que salen por la mañana (05:00-13:00), por la tarde (13:00-19:00) y por la noche (19:00-05:00).

La diferencia principal entre el funcionamiento interno de la pesca y el agua consiste en que, cuando el jugador interactúa con el `WaterBodyInteractable` con una caña de pescar, este ejecuta el método `GetFish` de `WaterBodyObject`, el cual devuelve un pez aleatorio de la lista necesaria según la hora actual. Actualmente se pueden pescar los mismos peces en cualquier momento del día; pero el juego está preparado para establecer listas de peces distintas según la hora. Una vez `GetFish` ha devuelto un pez, el `WaterBodyInteractable` indica al componente `FishingManager` del jugador que ha comenzado la pesca con el método `StartFishing`.

El proceso de la pesca es el siguiente:

- `StartFishing` avisa al componente `PlayerState` del jugador que debe impedir que este pueda moverse, interactuar o abrir el inventario. Luego, lanza una `Coroutine` de Unity que, en primer lugar, utiliza la instrucción `yield return new WaitForSeconds(0.01f)`, para esperar una centésima de segundo antes de que comience la pesca. Esto se implementó así para evitar que el jugador pueda recoger la caña en el mismo frame en el que se empieza a pescar.
- Después de esperar, espera un número de segundos aleatorio entre 1.5 y el máximo definido por el pez. Justo después de esto, activa un booleano que indica que el pez ha picado, y, antes de hacer que el pez escape, espera los segundos que le indican sus datos.

- Desde el momento en el que el se terminan los 0.01 segundos iniciales de espera, el jugador puede pulsar la tecla “T” en cualquier momento para recoger la caña, lo que lanza el método `PullLineUp`. Si lo hace cuando está activo el booleano que indica que el pez ha picado, este es capturado con éxito, se añade al inventario, y se reinician las variables de la pesca. Independientemente de si el jugador logra pescar el pez con éxito o no, el estado del jugador se reinicia para que pueda volver a moverse, interactuar y abrir el inventario.

4.2.12 Máquinas

Internamente, las máquinas funcionan de forma similar a los cultivos, dado que estas deben colocarse sobre mesas especiales para poder usarse. Estas mesas tienen un componente `MachineStandInteractable` que solo permite interactuar con el jugador si este utiliza una máquina, que debemos recordar que se representaban en el inventario mediante `MachineItem`. Así, si el jugador interactúa con una mesa teniendo equipado uno de estos objetos, se instancia el Prefab de su máquina y se coloca como hijo de la mesa en la jerarquía de Unity, además de sacar el objeto del inventario.

Las máquinas en sí siguen el esquema que hemos visto numerosas veces hasta ahora, teniendo un `MachineInteractable` y un `MachineObject`. En este caso, el primero se encarga de las siguientes funciones:

- Si el jugador interactúa con ella usando un pico o un hacha, el GameObject de la máquina se destruye, y esta es devuelta al inventario del jugador.
- Si la máquina no se está usando en ese momento y el objeto equipado es el que la máquina toma como input de procesamiento, se introduce en ella y comienza a procesarse.
- Si el jugador interactúa con la máquina sin usar el objeto equipado, se comprueba si el producto de la máquina está listo y si el jugador tiene espacio en el inventario. Si es así, se recoge el producto, y la máquina pasa a estar libre otra vez.

El componente `MachineObject` cuenta el tiempo de procesamiento en frames, al igual que los cultivos y animales, por lo que se puede saber si el producto está listo para ser recogido, si el contador de frames es igual al número de frames necesarios para finalizar el procesamiento.

4.2.13 Fabricación y cocina

La Figura 10 muestra la interfaz de la fabricación de objetos mediante una mesa de bricolaje, que se compone por tres elementos principales. Estos son, de izquierda a derecha, la lista de recetas, cada una representada por el icono del objeto que producen, la receta concreta seleccionada y su lista de ingredientes, y un panel del inventario. Este último, como ya se ha mencionado, en un panel de tipo `StaticInventoryPanel`, así que no se puede interactuar con él.

Figura 10: HUD de la mesa de bricolaje

El funcionamiento interno de la interacción entre el HUD y el `CraftingManager`, el script que se encarga de fabricar los objetos y que posee las listas de recetas, es un poco diferente de como funcionaban las tiendas y los puntos de venta. Esto se debe, principalmente, a que `CraftingManager` no es un componente de la mesa de bricolaje en sí, sino del GameObject del jugador. Así, el único script que poseen las mesas es el `CraftingTableInteractable`, que se encarga de pedir al `PlayerUIManager` que active el HUD de la fabricación, delegando en el `CraftingManager` para que prepare la interfaz. Este, a su vez, indica al componente principal del HUD, el `CraftingPanel`, el inventario, lista de recetas y título que debe cargar, dependiendo de si la mesa es de bricolaje o cocina.

El HUD de las mesas de bricolaje es bastante sencillo. En primer lugar, la lista de recetas está formada por una lista de iconos organizados con un `GridLayout` y un `ScrollRect` de la misma forma ya vista en el HUD de las tiendas. Para ver una receta, el jugador debe pulsar sobre el icono de la que desea cargar en el panel del centro. Esto hará que el `CraftingPanel` avise al componente principal del cartel de la receta elegida, el `ChosenRecipePanel`, de que tiene que colocar la información de la receta seleccionada en él, cargando el nombre y la lista de ingredientes de la misma. Además, también muestra los dos botones que aparecen en la parte inferior de la Figura 10, que permiten fabricar 1 o 5 unidades del objeto de la receta. Cuando el jugador pulsa uno de estos botones, este avisa al `CraftingPanel` y al `ChosenRecipePanel`, quienes invocan el método `CraftRecipe` de `CraftingManager` para fabricar la receta, el cual se explicará más adelante. Finalmente, el panel del inventario se actualiza automáticamente.

El método `CraftRecipe` recibe como parámetros una receta, es decir, un objeto de la clase `Recipe`, que almacena toda la información de la misma, y el número de unidades que se quiere fabricar. Antes de crear el objeto, analiza uno a uno cada ingrediente de la lista de materiales de la receta, calculando cuántas copias del objeto final sería posible fabricar, teniendo en cuenta el número de unidades necesarias del ingrediente y la cantidad que el jugador posee en el inventario, obteniendo esta cifra con el método `GetAmountOfGivenItem` de `InventoryManager`. Es posible que no haya suficientes

unidades del ingrediente en el inventario como para fabricar la cantidad que se le pidió al método `CraftRecipe`, en cuyo caso se anota el ingrediente y el número de unidades que faltan en un diccionario. Una vez se han analizado todos los ingredientes, se comprueba si hay suficientes como para fabricar, al menos, una unidad del resultado final. Si es así, se añaden al inventario las copias que haya sido posible fabricar, y se extraen del mismo los ingredientes que se han utilizado.

Finalmente, cabe destacar que, como se ve en la Figura 11, el HUD de las mesas de cocina es exactamente igual que el de las mesas de bricolaje. Sin embargo, al abrir la interfaz, el `CraftingManager` carga las recetas de cocina en el panel de la izquierda, en lugar de las de fabricación de objetos.

Figura 11: HUD de la mesa de cocina

4.2.14 Modelado

Actualmente, la mayoría de los GameObjects que son elementos del escenario utilizan mallas básicas de Unity, como cubos, cilindros y cápsulas, debido a que se le ha dado prioridad a programar las mecánicas del juego. Sin embargo, ha sido posible modelar parte de las mallas, que se presentarán en este apartado.

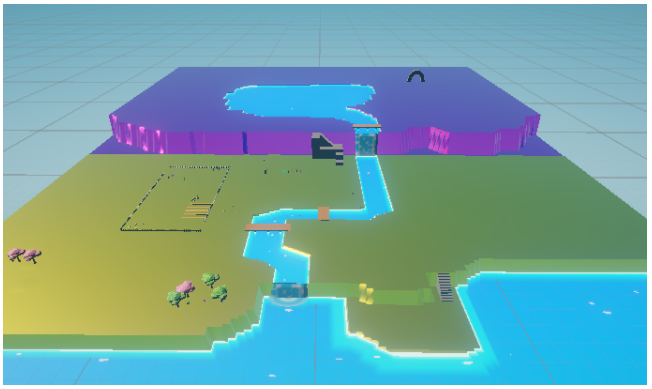


Figura 12: Vista aérea del escenario exterior

En primer lugar, la malla del escenario principal, que se puede ver en la Figura 12, está

formada por dos partes. Estas son el nivel de la costa y el de la montaña, que aparecen pintados en dorado y morado, respectivamente. Ambos han sido modelados con ProBuilder, una herramienta cuya interfaz aparece en la Figura 13, y que permite construir modelos directamente en Unity, tal y como se haría en un programa especializado en modelado 3D, como Blender o Maya. Así, para crear cada parte del escenario, se creó un plano dividido en muchos polígonos, y se extruyeron zonas grandes del mismo para formar los desniveles necesarios para formar tanto los acantilados como los cauces de los ríos. Además, también se utilizó ProBuilder para modelar las escaleras que aparecen en las Figuras 14 y 15.

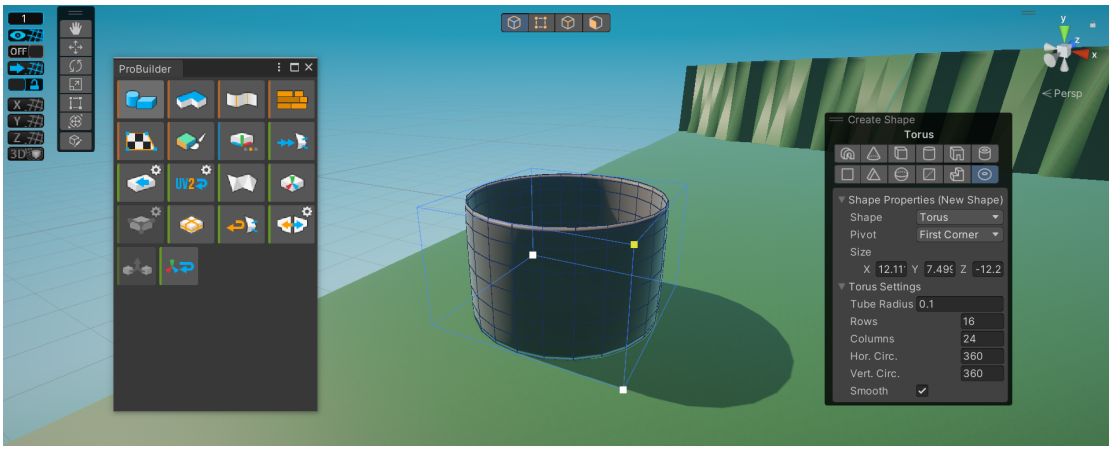


Figura 13: Ventanas de ProBuilder sobre la escena del juego

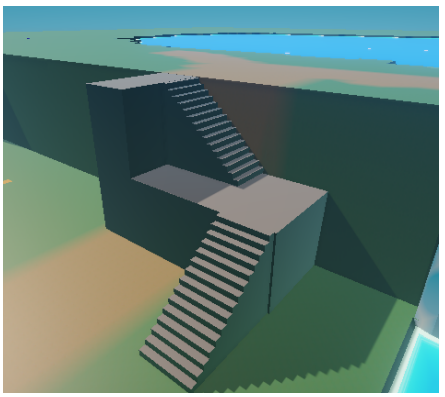


Figura 14: Escaleras que conectan el pueblo con la montaña

Para pintar el escenario hecho con ProBuilder se utilizó otra herramienta de Unity llamada PolyBrush. Al instalarla mediante el Package Manager, PolyBrush permite crear materiales a partir de un shader especial adaptado para “Texture Blend”, una técnica que permite mezclar texturas. Así, mediante la interfaz de PolyBrush que aparece en la Figura 15, es posible elegir hasta cuatro texturas diferentes y pintar una malla con ellas. Las tres texturas elegidas fueron creadas en GIMP.

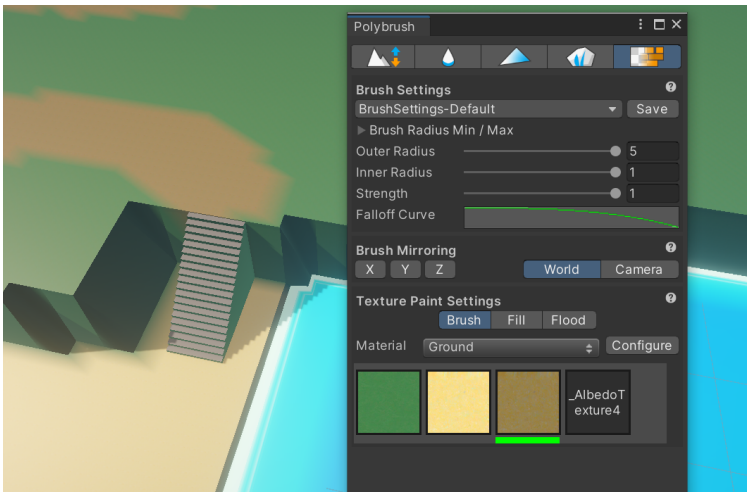


Figura 15. Ejemplo de uso de PolyBrush

Por otro lado, el resto de los modelos creados fueron desarrollados en Blender. Cabe destacar que el modelado de los personajes quedó a medias, dado que la malla fue creada con éxito; pero no se pudo acabar el rigging, weight painting, y, posteriormente, las animaciones del personaje. La Figura 16 muestra el modelo del personaje en su estado actual, con el esqueleto superpuesto.

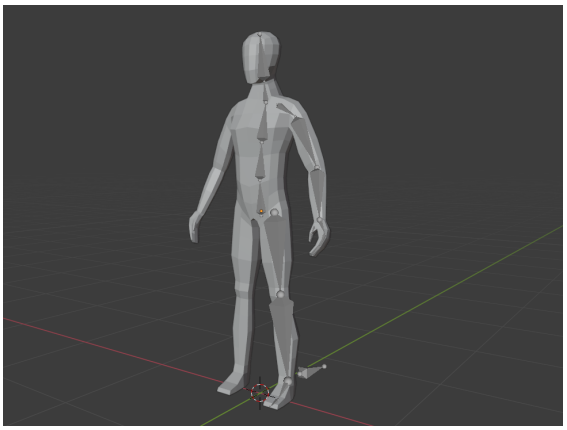


Figura 16: Modelo básico de un personaje con armazón

Además, en Blender también se realizaron los modelos de los árboles del juego. Después, para cada uno se generó un mapa UV, que es una imagen en la que aparecen las superficies y polígonos de un modelo y se utilizó para generar las texturas de cada uno de los modelos creados. Por ejemplo, para el cerezo se desarrolló el mapa UV que aparece en la Figura 17. Después se exportó su modelo en formato FBX desde Blender y se guardó el mapa UV como una imagen PNG. Así, si se importan ambos a Unity, es posible crear un material a partir del mapa y aplicarlo a la textura tal, de forma que el resultado final es el que se muestra en la Figura 17.

Figura 17: Mapa UV del cerezo y modelo final en Unity

Es importante mencionar que el modelo de la mina también fue desarrollado en Blender. Sin embargo, dado que ProBuilder permite convertir mallas creadas en `Colliders`, el `MeshCollider` de la mina fue creado directamente en Unity. El modelo original en Blender puede verse en la Figura 18; mientras que la Figura 19 muestra el `MeshCollider` creado con ProBuilder para él.

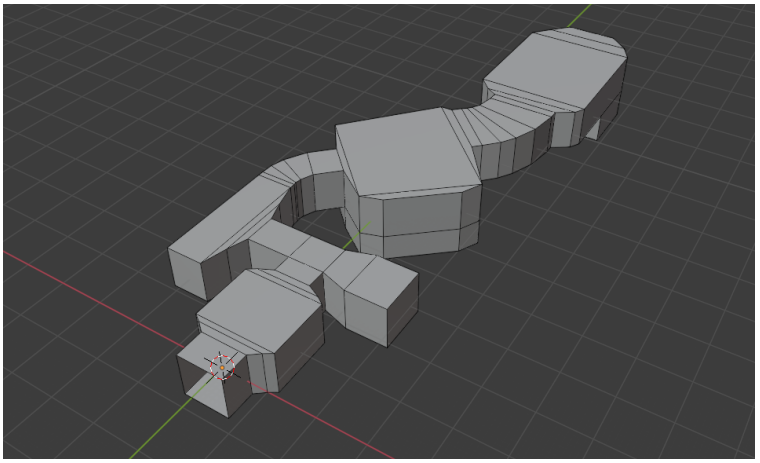


Figura 18: Modelo de la mina en Blender

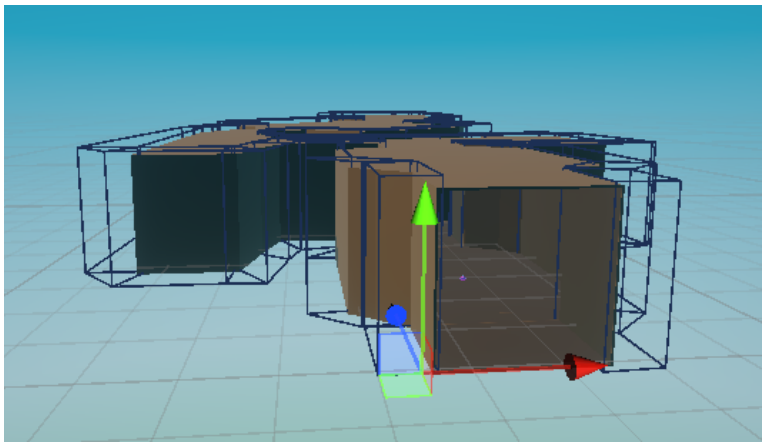


Figura 19: MeshCollider de la mina

4.2.15 Shaders

Se han desarrollado algunos shaders con el objetivo de crear materiales para el escenario con ellos. Concretamente, son los siguientes:

- **Water**: se utiliza para todas las masas de agua, a excepción de las cascadas.
- **Waterfall**: se usa en las cascadas.
- **Lava**: se emplea en la lava decorativa de la mina.
- **Skybox**: este shader se usa para el cielo, y su aspecto depende de la hora actual. Por ello, se explicará más en profundidad en el siguiente apartado.

Los shaders se pueden ver en la Figura 20.

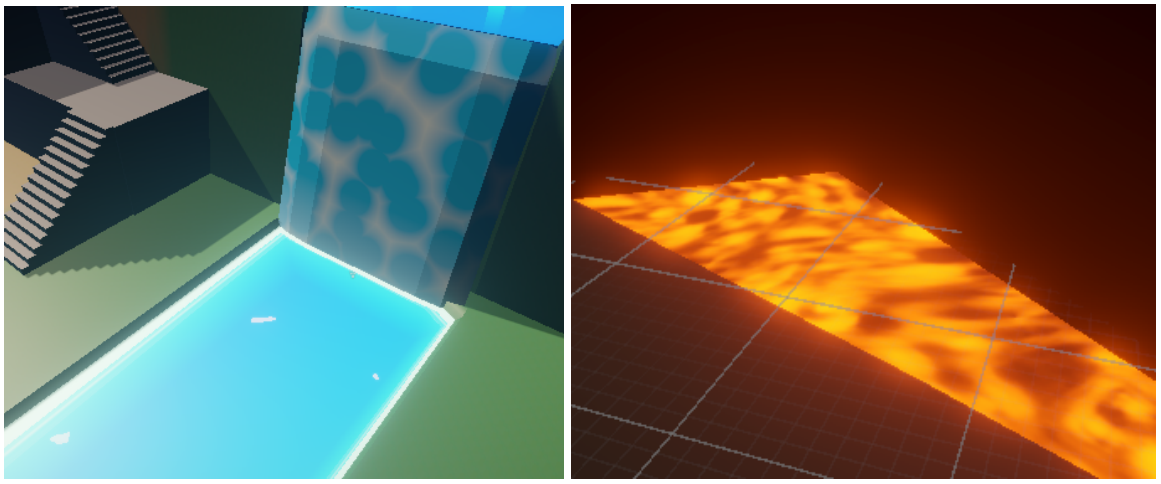


Figura 20: Shaders del agua, las cascadas, y la lava

4.2.16 Ciclo de día y noche

En el juego, el tiempo es controlado por el script `TimeManager`, un componente asignado a un GameObject sin modelo ni textura. Los datos más importantes de este fichero son:

- **Horas de comienzo y finalización de la partida**: se representan como números flotantes. Actualmente, las partidas comienzan a las 9 de la mañana, y terminan a las 9 de la noche.
- **Horas del amanecer y el atardecer**: también son números flotantes, y se corresponden con las 5 de la mañana y las 7 de la tarde, respectivamente. Para determinar si es de día o de noche, se utilizan estos valores.
- **Segundos por minuto**: es el número de segundos reales que deben transcurrir para que pase un minuto en el juego.
- **Luces solar y lunar**: son dos luces direccionales, es decir, GameObjects con el componente Light, que se almacenan junto a una `AnimationCurve` y a un `Gradient` para cada uno. La primera indica cómo cambia la intensidad de la luz a lo largo del día, mientras que el segundo hace lo mismo con el color.

Internamente, para controlar las variables relacionadas con el tiempo, se utilizan las clases de Unity `TimeSpan` y `DateTime`, dado que facilitan las operaciones que implican horas, minutos y segundos.

En el método `Awake` se invoca a la `Coroutine ControlPlayTime`, que se encarga de controlar el paso de los segundos del juego y de actualizar las luces adecuadamente. Para ello, utiliza un bucle while que para cuando la hora actual llegue a la hora de finalización de la partida. En cada iteración, la `Coroutine` espera los segundos por

minuto indicados, y luego suma un minuto a la hora actual y ejecuta el método `UpdateTime`, el cual activa o desactiva los GameObjects del Sol y la luna en función si es de día o de noche. Además, realiza dos funciones importantes.

La primera es actualizar las luces solar y lunar, utilizando los métodos `UpdateSun` y `UpdateMoon`, respectivamente. `UpdateSun` funciona de forma que si es de día se calcula, a partir de la hora actual, qué porcentaje del periodo entre el amanecer y el ocaso ha transcurrido. Ese valor se utiliza para calcular la rotación del Sol para que pueda ocultarse cuando debe, es decir, que su luz debe ser horizontal al anochecer. Además, su intensidad y su color se cambian por los valores de la `AnimationCurve` y el `Gradient` que corresponden al porcentaje obtenido. En cambio, si es de noche, la intensidad y el color del sol no se actualizan, dado que, por la noche el GameObject del Sol permanece desactivado. `UpdateMoon` hace lo mismo, pero al revés, porque su Light debe estar activada durante la noche.

La segunda función es utilizar un segundo script, llamado `SkyboxManager`, para modificar la `Skybox` dinámicamente en función de la hora actual. Para ello, divide la hora entre 24, obteniendo así el porcentaje del día completado, e interactúa con el shader Skybox mencionado en el apartado anterior. Este shader guarda los tres gradientes que se ven en la Figura 21, obteniendo el material final para el cielo sumando los tres gradientes tras multiplicar cada uno por un valor concreto.

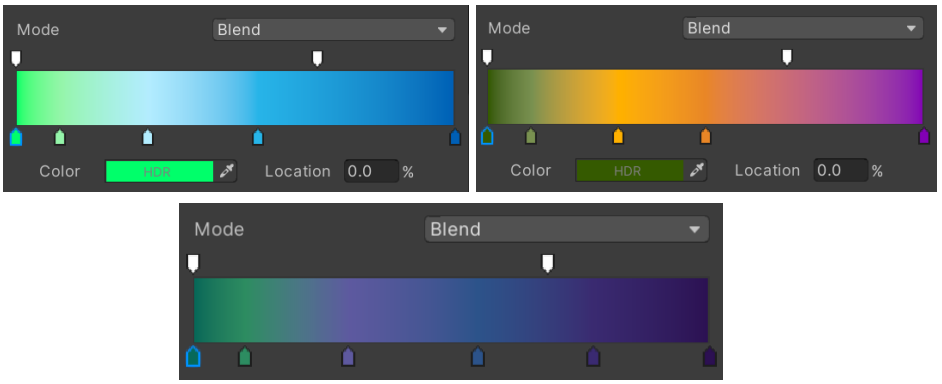
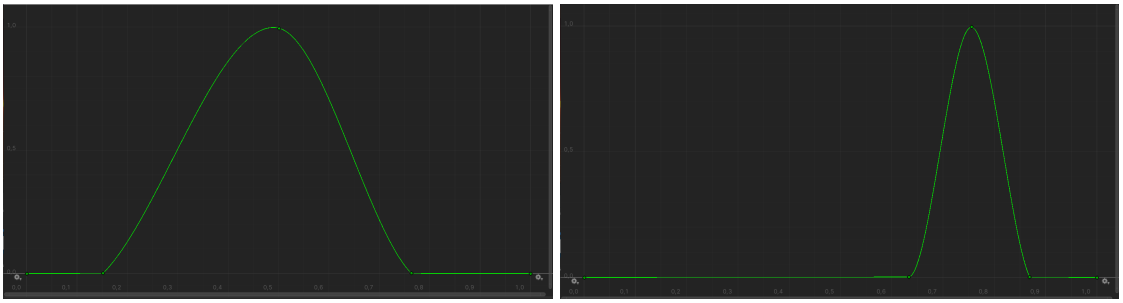


Figura 21. Gradientes de los tres cielos

El factor que multiplica a cada gradiente se obtiene de una `AnimationCurve` que almacena el componente `SkyboxManager` como propiedad. Tal y como se muestra en la Figura 22, los valores que toman las curvas indican el peso que debe tener cada gradiente en la suma final.



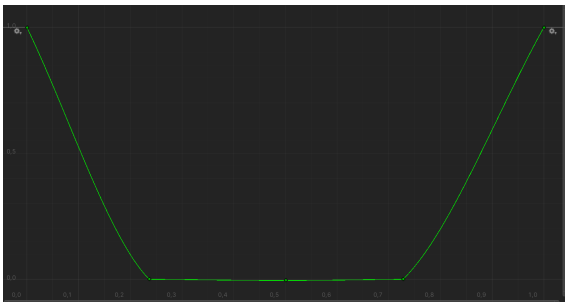


Figura 22: Curvas del peso de cada gradiente a lo largo del día (siguen el mismo orden que los gradientes de la Figura 21)

Así, el script `SkyboxManager` obtiene el valor de cada una de las curvas anteriores, y modifica las propiedades del shader que actúan como pesos en la suma según corresponda.

Un ejemplo del funcionamiento de este script es el color del cielo durante el mediodía. En ese momento del día, los valores tomados de las curvas son los que se localizan justo en el centro, por lo que el gradiente final del cielo será el que va de verde a cian, tal y como se ve en la Figura 23.

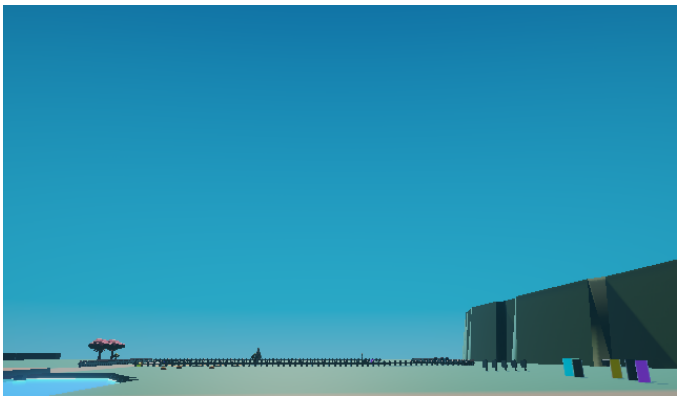


Figura 23: Gradiente del cielo durante el día

4.2.17 Cámara

Por lo general, la cámara sigue al jugador en todo momento gracias al script `CameraController`, que le permite moverse en cada frame siguiendo un GameObject. La cámara se localiza casi siempre, en coordenadas globales, en la posición (posición del GameObject seguido) + (0, 5, -10). Además, siempre está rotada 15 grados en el eje X, de forma que mira hacia el jugador, como se ve en la Figura 24.

Figura 24: Vista desde la cámara sin objetos por delante

Esto se lleva a cabo en el método `LateUpdate` de `CameraController`. En cada frame se lanza un Raycast hacia atrás desde la posición que está tres unidades en el eje Y por encima del jugador. Dicho Raycast se mueve 10 unidades en el sentido negativo del eje Z global (en la Figura 24, es hacia la cámara; mientras que, en la Figura 25, es hacia la parte de abajo de la imagen), y su objetivo es detectar si hay algo en medio de la posición por defecto de esta y el jugador. Así, si el Raycast colisiona con algo, la cámara se mueve directamente encima del jugador mirando hacia él. Sin embargo, si el Raycast no detecta nada, la cámara se mueve a su posición normal.

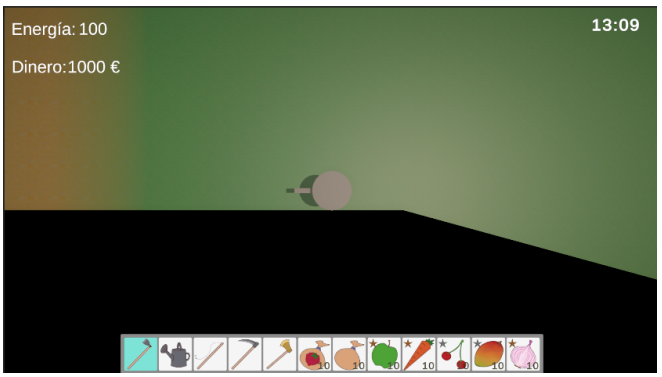


Figura 25. Vista de pájaro desde la cámara

Por ejemplo, en la Figura 25, la cámara se ha movido encima del jugador debido a que hay un objeto delante de él.

4.3 Inteligencia Artificial

Esta sección se dedicará a explicar en profundidad la Inteligencia Artificial desarrollada para el proyecto.

4.3.1 ¿En qué consiste el GOAP?

Para implementar la IA se ha elegido la técnica GOAP (Goal Oriented Action Planning), que originalmente fue creada por Jeff Orkin para la IA del videojuego F.E.A.R. Esta arquitectura se fundamenta en establecer una serie de metas y de acciones, de forma que el agente puede hacer lo siguiente:

- **Elegir dinámicamente una meta de entre las disponibles**: cada una tiene una prioridad, lo que permite al agente determinar un objetivo a corto plazo.
- **Planificar una secuencia de acciones para alcanzar dicha meta**: cada acción tiene un coste asociado, así que el agente puede intentar formar un plan que le permita alcanzar la meta propuesta, eligiendo el que minimiza la suma de los costes de sus acciones si es posible lograr la meta de varias formas.

Además de las listas de acciones y metas, el agente cuenta con una base de conocimiento en la que guarda sus percepciones acerca del mundo, de forma que puede utilizar esa información para saber qué acciones son posibles y cuáles no.

El motivo por el que se ha elegido utilizar esta técnica para implementar la IA del proyecto, en lugar de otros sistemas, como las máquinas de estados, es que GOAP es muy modular, dado que es posible crear nuevas acciones y añadirlas al sistema con mucha facilidad. Así, a diferencia de las máquinas de estado, GOAP no necesita mantener un grafo que crece en complejidad a medida que se añaden acciones nuevas, porque la secuencia de acciones a tomar se decide en tiempo de planificación.

4.3.2 Implementación

Para desarrollar el GOAP del proyecto se ha partido del framework de GOAP implementado por Penny de Byl en su minicurso de Youtube “Goal Oriented Action Planning”, en el cual se crea una simulación de un hospital. Los scripts más importantes del modelo son los siguientes, aunque es necesario aclarar que algunos se han eliminado, dado que no eran necesarios para el proyecto:

- **`GAgent`**: es el cerebro del agente, dado que en él se integran los demás componentes y se decide, mediante el método `LateUpdate`, lo que hace la IA en cada momento. Así, si hay acciones por realizar, gestiona su ejecución; mientras que, si no hay, ordena las metas disponibles en orden descendente de prioridad y va utilizando el componente `GPlanner` para trazar un plan para cada una hasta que encuentra una con un plan que se puede realizar. `GAgent` es una clase base, dado que es posible definir varios tipos de agentes. Además, cada meta está representada por una instancia de la clase `SubGoal`, que almacena su nombre y prioridad.
- **`GPlanner`**: es la clase que se encarga de la función de planificación del GOAP. Para llevarla a cabo, el `GAgent` le indica qué acciones hay disponibles, cuál es el estado actual del mundo, y qué meta se quiere alcanzar. Basándose en esa información, `GPlanner` construye un grafo en forma de árbol y selecciona el nodo hoja con el camino hacia la raíz más corto, pues este corresponderá a la secuencia de acciones de menor coste.
- **`GWorld`**: es el componente que representa el estado del mundo. En el framework, las bases de conocimiento se representan mediante instancias de la clase `WorldStates`, que constituye una interfaz para un `Dictionary<string, int>`. Cabe destacar que `GWorld` no se utiliza en la adaptación del framework realizada, dado que toda la información que utiliza el agente procede de las llamadas “beliefs”, que son las creencias del agente. El motivo por el que no se utiliza `GWorld` es que, en principio, no va a haber varios agentes que necesiten comunicarse entre sí con una clase que realice las funciones de “blackboard”, sino que basta con que el único agente que hay pueda almacenar y extraer creencias en su propia base de conocimiento, que es la que se utilizará para determinar qué acciones son viables.
- **`GAction`**: es una clase base a partir de la cual se puede implementar el resto de acciones del agente. En el framework, el modus operandi del agente dado una acción siempre es: ejecutar acciones antes de moverse, luego, moverse hacia el objetivo de la acción, y, cuando haya alcanzado el destino, realizar otra serie de acciones para finalizar. Así, las nuevas acciones definidas a partir de `GAction` deben especificar qué se debe hacer en el primer y el último paso.

Para mayor comodidad durante el desarrollo de la IA, se ha añadido un componente adicional llamado `Cache`, cuyo objetivo es almacenar información que puede estar en la base de conocimiento de las beliefs dado que no se puede representar mediante parejas `<string, int>`. Algunos ejemplos de información guardada en la caché son las comidas favoritas y odiadas que se hayan averiguado de los animales, y una referencia al `ShopManager` de la tienda que se quiere visitar. Además, se ha creado una clase hija de GAgent llamada Farmer, en la que se definen las metas del agente y se añaden algunas beliefs iniciales, como la energía y el dinero con los que empiezan los jugadores.

Es importante destacar que algunos de los componentes del GameObject del jugador que fueron descritos en el apartado 4.2.1 se han eliminado del agente, dado que no se utilizan. Un ejemplo de esto es el `PlayerUIManager`, pues la IA interactúa directamente

con los Manager adecuados en cada caso, sin utilizar ningún tipo de HUD. Por esto, los scripts se han adaptado para evitar las acciones relacionadas con el HUD si el tag del GameObject del que son componentes es “AIPlayer”. Por otro lado, también se ha añadido métodos nuevos a clases como `InventoryManager`, con el objetivo de hacer más sencillas las operaciones de la IA. Por ejemplo, `InventoryManager` ahora puede anotar en las beliefs del agente el número de objetos de un tipo que hay en el inventario.

4.3.3 Acciones de la IA

En este apartado se describirán las diferentes acciones de la IA que se han implementado, enfocando la explicación especialmente hacia el razonamiento de alto nivel que debe seguir el agente para afrontar cada situación. Todas son clases hijas de `GAction`.

Acciones relacionadas con el establo

La primera acción de este grupo es **`GoToStable`**, que permite al agente desplazarse hasta el establo de su granja. Sin embargo, esto solo se lleva a cabo si en el establo hay, al menos, un animal, pues carecería de sentido ir si no se cumpliera esta condición. Es importante resaltar que el agente comienza con una gallina en el establo para que tenga, al menos, una fuente de ingresos, por lo que lo anterior siempre se cumplirá.

La segunda acción relacionada con el establo es **`TakeCareOfAnimals`**. Una vez el agente está en el establo, consulta en la caché, que es donde se almacena la lista de animales en el establo, si hay alguno que no haya sido alimentado recientemente o cuyo producto esté listo para ser recogido. Si es así, la IA se desplaza hasta el animal y se prepara para interactuar. Se sabe si un animal ha comido hace poco gracias a un booleano añadido a `AnimalObject` que lo indica.

Si hace mucho que el animal fue alimentado por última vez, el agente busca en la caché sus comidas favorita y odiada, y encarga al `InventoryManager` que busque un alimento basado en esa información, priorizando aquellos de mayor calidad. Si hay, al menos, un alimento que valga la pena ofrecer al animal, se le da, apuntando en la caché si es su comida favorita u odiada en el caso de que se desconozca.

Además, si el producto del animal está listo para ser recogido, el agente se equipará con la herramienta necesaria para hacerlo, aunque, como se pueden recoger los huevos de las gallinas sin necesidad de ninguna herramienta, esto no se hace si el animal en cuestión es una. Si el agente no tiene la herramienta necesaria, la apunta en la lista de herramientas faltantes de la caché.

Acciones relacionadas con la búsqueda de tesoros

El agente posee una acción, **`CheckBeach`**, que le permite ir a la playa y buscar montones de arena por excavar. A diferencia de otras acciones utilizadas para ir a un lugar concreto, esta no requiere realizar comprobaciones previas al trayecto. Una vez el agente está en la playa, utiliza un `Physics.OverlapSphere` para buscar los montones de arena, anotando en la caché los que se han encontrado y en las beliefs la cantidad de estos.

Por otro lado, la acción que permite al agente excavar los montones de arena encontrados, **`DigFoundPiles`**, es bastante sencilla. Así, antes de ir hacia una de estas pilas, comprueba si tiene una pala en el inventario, pues no tendría sentido ir hacia el objetivo si no es así. Si no posee ninguna pala, anota que necesita una en la lista de herramientas faltantes y cancela la acción; pero, si tiene una pala y quedan montones por revisar, va hacia uno de ellos e intenta extraer su tesoro. Para ello, sigue excavando hasta

que logre sacarlo, cambiando la pala por otra si se rompe antes de lograrlo. Sin embargo, si esto ocurre y al agente no le quedan palas, la acción se cancela.

Acciones relacionadas con el bricolaje

El agente utiliza la acción **`GoToCraftingTable`** para desplazarse hasta la mesa de bricolaje de su granja. No tiene condiciones que impidan ir, pero, antes de hacerlo, la IA hace un recuento de las herramientas que no tiene, por si acaso falta alguna por anotar en la caché.

Una vez está en la mesa de bricolaje, el agente utiliza la acción **`Craft`** para fabricar lo que necesite de la siguiente forma:

- En primer lugar, recorre la lista de herramientas faltantes de la caché e intenta fabricar una unidad de cada una. Si no es posible fabricar alguna de ellas, el agente anota en las beliefs los nombres de los materiales que le faltan.
- Una vez revisada la lista de herramientas, obtiene de `CraftingManager` todas las recetas de los teletransportadores y muebles, e intenta fabricarlos todos, aunque es poco probable que pueda hacerlo. Sin embargo, si no puede fabricar alguno, no se apuntan los ingredientes que faltan en las beliefs, dado que no son bienes de primera necesidad, como las herramientas.

Acciones relacionadas con la cocina

En este caso, la acción que permite al agente ir a la cocina, **`GoToCookingTable`**, no realiza ninguna tarea de preprocesado ni comprueba que se cumpla ninguna condición específica.

Del mismo modo, **`Cook`**, la acción para cocinar, es más sencilla que `Craft`, dado que no es posible obtener diferentes tipos de objetos de la cocina, al contrario que con el bricolaje. Así, antes de cocinar, el agente obtiene del `CraftingManager` la lista de recetas de cocina y las ordena en orden decreciente según la energía restaurada por los platos, de forma que intenta cocinar antes aquellos que más energía dan. Una vez ordenada la lista, la recorre intentando cocinar cada plato.

Acciones relacionadas con la pesca

Mediante la acción **`GoToFishingSpot`**, el agente puede ir a un punto de pesca. Para ello, obtiene de la caché la localización del punto más cercano; pero, antes de ir hacia allí, comprueba si tiene una caña en el inventario. Si no es el caso, apunta en la lista de herramientas pendientes de la caché que necesita una caña de pescar, y cancela la acción.

Si el agente ha ido a un punto de pesca, utiliza la acción **`TryToCatchFish`** para intentar pescar un solo pez. Sin embargo, la IA intentará utilizar esta acción varias veces, anotando en las beliefs el número de peces que ha pescado, de forma que para cuando llegue a 5. Dado que es posible que la caña se rompa entre capturas consecutivas, el agente vuelve a comprobar si tiene una caña y, si es así, la equipa y lanza el cebo al agua. Después, haciendo uso de una `Coroutine`, espera un número de segundos aleatorio entre 1.4 y 5, que es, aproximadamente, el tiempo que tardan en picar los peces. Cuando termina la espera, recoge la caña, añadiendo el nuevo pez al inventario y aumentando el contador de peces de las beliefs.

Acciones relacionadas con la exploración de los bosques

En este caso, la acción que utiliza el agente para ir a un bosque, llamada **`GoToForest`**,

utiliza técnicas ya vistas en otras acciones. Así, el agente siempre se dirigirá hacia el bosque más cercano y, al llegar, usará un `Physics.OverlapSphere` para apuntar en la caché los GameObjects de todas las plantas del bosque.

Una vez hecho esto, el agente repetirá la siguiente acción, **`GatherForestResources`**, hasta que no queden plantas sin revisar. Dado que en los bosques se pueden encontrar plantas de diferentes tipos, el agente tiene un comportamiento definido para cada uno:

- Si la planta es un **árbol no frutal**, es decir, una encina o un pino, el agente lo talará. Para hacer esto, comprobará si tiene un hacha en el inventario, añadiéndola a la lista de herramientas pendientes si no es así. Después, el agente golpeará el árbol con el hacha hasta que termine de talarlo o la IA se quede sin hachas o sin energía suficiente como para utilizar una. Al final, si ha logrado talar el árbol o si no quedan hachas, este se quita de la lista de plantas pendientes por revisar.
- Si la planta es una **planta silvestre con frutos**, como un arbusto de frambuesas o un manzano, simplemente los cosecha y quita la planta de la lista.
- Por último, si la planta es una **mata de hierba** que ha crecido del todo, el agente intentará cortarla con una guadaña, apuntando en la caché que necesita una si no cuenta con ninguna. Independientemente del resultado, la mata se extrae de la lista de plantas pendientes.

Acciones relacionadas con el uso de máquinas

En la caché se guardan dos listas relacionadas con las máquinas que posee el agente, una para las mesas que están ocupadas por una máquina, y otra para aquellas que no lo están. Así, **`GoToMachineArea`**, que es la acción que permite al agente ir a la zona donde están las máquinas de la granja, comprueba antes de ir si se dan a la vez las siguientes condiciones:

- No hay ninguna máquina en la zona de máquinas, en uso o vacía.
- No hay máquinas en el inventario.

Si se cumplen a la vez, no tiene sentido ir a la zona de máquinas, así que se cancela la acción.

Por otro lado, una vez en el área de máquinas, el agente utiliza la acción **`CheckMachines`** para realizar sus tareas. Concretamente, cada vez que se ejecuta esta acción, ocurre lo siguiente:

- Antes de dirigirse a ninguna mesa concreta, el agente comprueba si tiene alguna máquina en el inventario y, si es así, si hay alguna mesa libre. Si se dan estas dos condiciones, la IA irá a esa mesa, colocará la máquina sobre ella, y la marcará como mesa en uso.
- En cambio, si no hay mesas libres o no hay máquinas en el inventario, se elige una máquina que no haya sido revisada recientemente y se desplaza hacia ella. Al llegar, el agente comprueba si la máquina está procesando un producto. Si es así y el producto está listo, la IA lo recoge e intenta introducir otra copia del input de la máquina. Si, por el contrario, la máquina no se está utilizando, el agente simplemente coloca el objeto necesario dentro para que empiece a procesarse, si es que el agente posee una unidad de dicho objeto.

Acciones relacionadas con la exploración de la mina

Dado que la mina está en una parte del escenario diferente al resto de localizaciones,

se han definido dos acciones nuevas llamadas **`EnterTheMines`** y **`ExitTheMines`**, que permiten al agente desplazarse hasta las zonas que hacen que los jugadores entren y salgan de la mina, respectivamente. Sin embargo, hay que destacar un detalle, que consiste en que, al entrar en la mina, la caché almacena una lista con las localizaciones de todos los sectores que quedan por visitar dentro de la mina.

Así, la acción **`GoToMiningArea`** toma uno de esos sectores, que normalmente están almacenados por orden de profundidad, y envía al agente a ese lugar. Una vez allí, el agente utiliza un `Physics.OverlapSphere` para obtener la información de todas las menas del área en la que está, anotando las referencias a sus GameObject en la caché, y el número total de menas encontradas en las beliefs.

Finalmente, para picar una mena se utiliza la acción **`ExtractFoundOres`**. Para hacer esto, el agente va hacia una de las que están registradas en la caché y la golpea repetidamente con un pico hasta que se rompa, parando si, en algún momento, se queda sin picos. Por último, si el agente ha logrado extraer los minerales, saca la mena de la lista y reduce el número de menas por romper de las beliefs. La idea detrás de este funcionamiento es que el agente explore los cuatro sectores de la mina en orden, de forma que, si está en uno concreto, rompa todas las menas que encuentra antes de pasar al siguiente.

Acciones relacionadas con la agricultura

Son las acciones que el agente realiza en el huerto de la granja. Para desplazarse hasta allí, se utiliza la acción **`GoToOrchard`**, que no necesita llevar a cabo tareas de preprocesamiento ni postprocesamiento.

Por otro lado, existen dos acciones que el agente puede realizar en el huerto:

- **`PlantAndWaterSeed`**: consiste en tomar del inventario una semilla, plantarla en una parcela que esté libre, y regarla para que crezca. Por ello, antes de moverse a ningún sitio, el agente debe comprobar que tiene una semilla, una azada, y una regadera, y que hay, al menos, un espacio libre en el huerto. Como ya se ha visto, si falta alguna de las herramientas, se añade a la lista de la caché. Además, si alguna de las cuatro condiciones anteriores no se cumple, se cancela la acción; pero, si se cumplen todas, el agente se dirigirá a la parcela elegida, la arará y regará, y plantará la semilla.
- **`WaterAndHarvestPlant`**: esta acción sirve para regar y cosechar las plantas que lo necesiten, y sigue una filosofía similar a la anterior, en el sentido de que primero comprueba si hay plantas que necesiten ser regadas o que hayan dado frutos. Si encuentra una planta con el suelo seco, también comprueba si tiene una regadera, cancelando la acción si no es así. Si el agente tiene los medios para interactuar con la planta, se mueve hacia ella, y la riega y/o cosecha sus frutos, dependiendo de lo que necesite.

Acciones relacionadas con la compra de objetos y animales

Dado que existen cinco tiendas, se ha programado una pareja de acciones para cada una de ellas, siguiendo el esquema que se ha utilizado hasta ahora de juntar una acción para ir a un lugar con otra que indica qué hay que hacer allí. Así, las acciones que pertenecen al primer grupo son:

- **`GoToFoodShop`**: esta acción lleva al agente a la tienda de comida, y no realiza ninguna tarea, ni antes ni después de hacerlo.
- **`GoToMachineShop`**: al igual que la anterior, esta acción, que permite al agente

dirigirse a la tienda de máquinas, no necesita ningún preprocesado ni postprocesado.

- **`GoToSeedShop`**: es la acción que usa el agente para ir a la tienda de semillas. Una vez allí, este anota todos los artículos de la tienda en la caché, pues la compra de semillas se hace de forma distinta a la compra de comida y máquinas.
- **`GoToAnimalShop`**: esta acción permite al agente ir a la tienda de animales, y también añade sus artículos a la caché una vez la IA ha llegado allí.
- **`GoToToolShop`**: al igual que las dos anteriores, esta acción, que lleva al agente a la tienda de herramientas, hace que la IA apunte los objetos disponibles en la caché una vez allí. Sin embargo, antes de ir realiza un recuento de las herramientas que le faltan, por si acaso alguna no esté anotada en la caché.

Por otro lado, las acciones que permiten al agente comprar los distintos artículos son bastante más complejas, y se explican a continuación:

- **`BuyFood`**: esta acción es utilizada para comprar comida, y funciona de una manera distinta a casi todas las demás, dado que, antes de adquirir nada, ordena la lista de productos de la tienda aleatoriamente, con el objetivo de no comprar siempre los mismos alimentos. Una vez hecho esto, el agente recorrerá la lista intentando comprar dos unidades de cada producto, parando cuando no quede espacio en el inventario o los fondos bajen por debajo de 500.
- **`BuyMachines`**: la implementación de esta acción, cuyo propósito es la compra de máquinas, es ligeramente diferente al resto. Esto se debe a que, antes de adquirir algo, el agente construye un conjunto con todos los tipos de máquinas que tiene en el inventario o en una mesa de la granja. Tomando como referencia este conjunto, el agente comprará máquinas de la tienda dando prioridad a aquellas que no tenga, aunque con ciertas limitaciones, pues no adquirirá más de dos máquinas y parará cuando no tenga espacio en el inventario o su dinero esté por debajo de 1000.
- **`BuySeeds`**: esta acción se usa para comprar semillas y es prácticamente idéntica a `BuyFood`, ya que los artículos de la tienda son ordenados aleatoriamente antes de comenzar a comprar. Sin embargo, en este caso, el agente intentará comprar 5 unidades de cada semilla, y para cuando el dinero actual cae por debajo de 200, dado que las semillas son más baratas que los alimentos.
- **`BuyAnimals`**: esta acción está implementada de forma que el agente compra solo un animal para evitar quedarse sin dinero muy rápido, dado que los animales suelen ser caros. Para hacer esto, la IA recorre la lista de animales a la venta buscando el mejor siguiendo los siguientes criterios:
    - La regla general consiste en dar prioridad a los animales que no están en el establo, información que también está en la caché.
    - Si el agente tiene que elegir entre dos animales que están en el establo, o entre dos de los cuales no se posee ninguna unidad, se tomará una decisión basándose en sus características. Concretamente, se elegirá el animal que menos espacio ocupe en el establo, y, si este valor es igual para ambos, se elige el más barato. Si ambos cuestan lo mismo, se elige aleatoriamente.
- **`BuyTools`**: el funcionamiento de esta última acción, que permite al agente comprar herramientas, es relativamente sencillo. Gracias a que la caché guarda una lista con los utensilios que el agente no tiene, lo que debe hacer es recorrer la lista de herramientas de la tienda comprobando si es una de las que no tiene, comprando una unidad si es así. En este caso, el agente cancela la acción si sus fondos caen por debajo de 300 o si se queda sin espacio en el inventario.

Acciones relacionadas con la venta de objetos

En primer lugar, el agente puede ir al arcón de venta de su granja utilizando la acción **`GoToSalePoint`**, que no necesita realizar ninguna tarea antes ni después de que el agente vaya a dicho lugar.

Las acciones que involucran vender objetos del inventario tienen un funcionamiento muy sencillo, pero, en esta ocasión, son dos:

- Si el agente quiere vender solo parte del contenido del inventario, se utiliza la acción **`SellSomeItems`**, que recorre todas las ranuras del mismo y decide qué hacer según el tipo de objeto que contenga. Si este es un `FurnitureItem` o un `Generic`, la IA venderá todas las unidades del espacio; mientras que, si es un `Consumable`, solo venderá la mitad. Además, en caso de que el `Consumable` sea un plato de comida, es decir, de la clase `Food`, no lo venderá, dado que es mejor reservarlos para que el agente se los coma cuando los necesite.
- Por otro lado, la acción **`SellAllItems`** se utiliza para vender todos los objetos del inventario, incluyendo las herramientas, teletransportadores, y demás objetos valiosos. Para evitar que el agente se quede sin poder hacer nada, antes de ejecutar esta acción, se comprueba en las beliefs que queda poco tiempo para que finalice la partida. Esto se conseguiría haciendo que el `TimeManager` descrito en el apartado 4.2.16 lance un evento cuando quede, aproximadamente, una hora para que termine la partida. Entonces, el agente recibiría el evento y anotaría en las beliefs que la partida está a punto de terminar; pero esto no se ha llegado a implementar.

Acciones adicionales

La última acción es **`Eat`**, que permite al agente comerse un único alimento. Curiosamente, esta es la única acción que no requiere que el agente se desplace a ningún lugar, sino que se puede realizar en cualquier momento. Para llevarla a cabo, el agente encarga a InventoryManager que busque el mejor alimento en cuanto a energía restaurada, teniendo en cuenta la fórmula que se utiliza para calcularla cuando los alimentos están asociados a una calidad. Si encuentra uno, se lo come, finalizando la acción.

Capítulo 5 Conclusiones y líneas futuras

En conclusión, si bien no se ha conseguido desarrollar una versión jugable del juego, se ha creado una base sólida cuyas mecánicas ya están implementadas, lo que permitirá continuar con el desarrollo del juego en un futuro sin tener que empezar de cero. Además, la mayoría de las funciones que fueron planteadas en los objetivos secundarios han sido implementadas con éxito, y, aunque no se ha terminado la IA, sí que se han programado las acciones que debería ser capaz de realizar. El desarrollo de este proyecto también me ha permitido adquirir muchos conocimientos de muchas áreas relacionadas con los videojuegos, como el modelado 3D, la programación y la Inteligencia Artificial.

Por otro lado, la principal conclusión personal que puedo extraer del trabajo realizado es que, gracias a la experiencia que se adquiere, como se mencionó antes, desarrollar un videojuego es un trabajo muy enriquecedor. Sin embargo, también es muy exigente, dado que requiere mucho esfuerzo y dedicación, especialmente si se desarrolla en solitario. Afortunadamente, mi interés hacia el campo ha permitido que haya disfrutado de la experiencia y me ha motivado a lo largo de estos meses de trabajo.

En lo que respecta a los planes para el futuro del proyecto, considero que intentar terminarlo sería lo correcto. Sin embargo, también sería necesario corregir ciertas partes del código que no funcionan como deberían o que pueden implementarse de otra forma más óptima o legible. Por otro lado, el juego necesita pasar por una larga etapa de pruebas para asegurar que todo lo implementado funciona correctamente, dado que no se ha podido testear el juego en profundidad. Concretamente, aunque se han desarrollado todas las acciones de la Inteligencia Artificial, es necesario enfrentarla a diferentes situaciones y asegurar que su reacción en todas es la esperada.

Sin embargo, también hay muchas ideas que hubo que descartar por falta de tiempo, y que me gustaría poder añadir al juego en un futuro. Por ejemplo, se podría introducir conversaciones con los aldeanos y un sistema de trueques para darle aleatoriedad al juego. Además, el juego también podría beneficiarse mucho de algunas mejoras de Quality Of Life, como añadir un sistema de avisos que indiquen los objetos que entran en el inventario. Otra posible ampliación del juego que podría ser interesante de realizar una vez exista una versión jugable del mismo consistiría en hacer que el juego tenga un modo multijugador online.

Capítulo 6 Summary and Conclusions

In conclusion, even though developing a fully playable version of the game was not possible, I was able to create one whose mechanics have been implemented. This will allow me to continue developing the game in the near future without having to do it from scratch. Besides, most of the functions that were specified in the secondary objectives have been implemented successfully, and the intelligent agent's actions have been programmed as well, though the AI itself was not finished. What is more, during the development of this project, I was able to learn a lot about many areas related to video games, like 3D modeling, programming, and Artificial Intelligence.

Moreover, the most important conclusion that I can get from this project is that due to the massive amount of knowledge that can be gained, as I said before, developing a videogame is a very enriching experience. However, it is very demanding, since it requires a lot of effort and dedication, especially if one does it alone. Fortunately, my interest in this field has allowed me to enjoy the experience to stay motivated during these months.

In regards to the future of the project, I think that trying to finish it would be the right thing to do. Nevertheless, it is also necessary to correct parts of the code that do not work as expected and other ones that could be implemented in a way that makes them more optimal or readable. On the other hand, the game needs to be tested to check that everything that has been implemented works correctly, since I could not test it enough these days. The part that needs to be tested the most is the Artificial Intelligence, since, even though all of its actions have been implemented, it should be tested using different case scenarios in order to check that the AI reacts correctly to all of them.

However, there are many ideas that had to be discarded due to the lack of time that I would like to implement in the game somewhere in the future. For example, I would like to add a dialogue system that allows the player to talk to the villagers, and a trade system as well, in order to give randomness to the game. Besides, some Quality Of Life improvements are needed, for instance, a warning system that tells the player when an item is added to the inventory. Another improvement that could be interesting to do once the game is fully playable would be to add an online multiplayer mode.

Capítulo 7 Presupuesto

Para finalizar, en este capítulo se estima un presupuesto necesario para cubrir los gastos del desarrollo del TFG.

En la Tabla 7.1 se recoge, para cada tarea realizada en el proyecto, una aproximación de las horas invertidas en ella y su precio, estableciendo unos 20 € como el precio por hora.

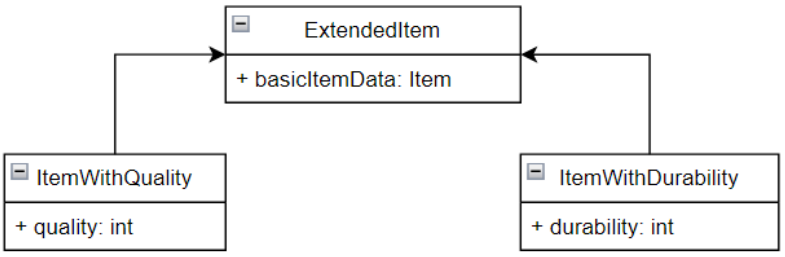
Tarea	Horas invertidas	Importe
Diseño	70	20 € x 70h = 1400 €
Programación de mecánicas	300	20 € x 300 h = 6000 €
Programación de IA	100	20 € x 100 h = 2000 €
Iconos de objetos	-	20 €

Tabla 7.1: Resumen de tipos

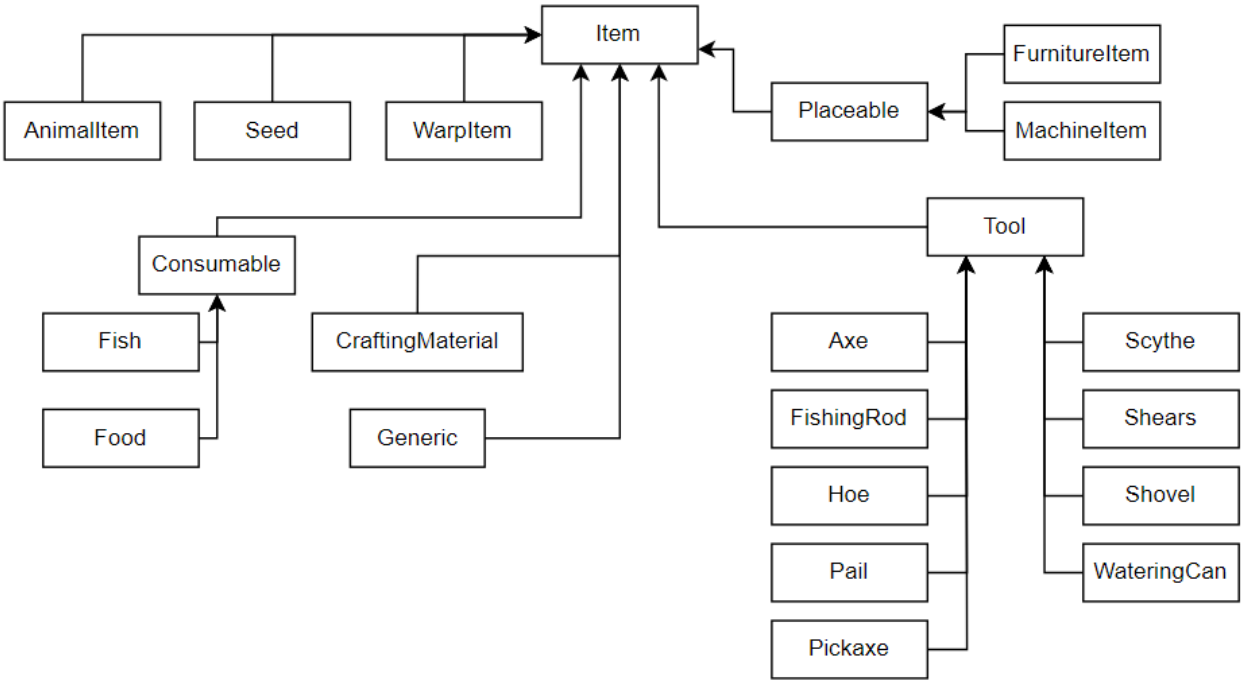
Así, al sumar todos los importes calculados, se obtiene que el presupuesto final es 9420 €.

Capítulo 8 Diagramas de clases

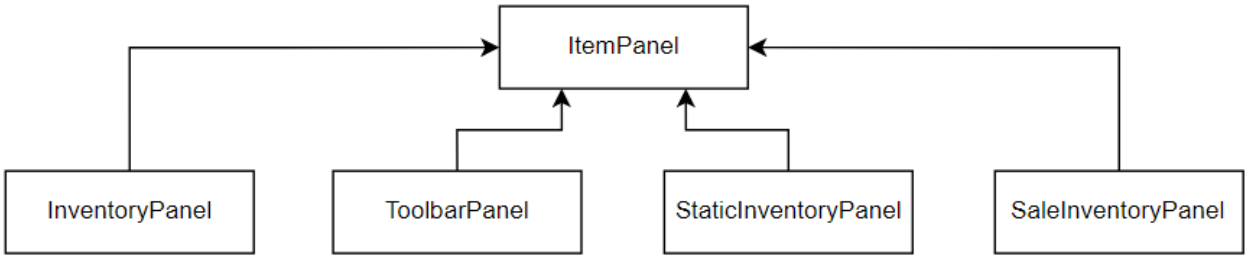
8.1 Diagrama de clases de ExtendedItem



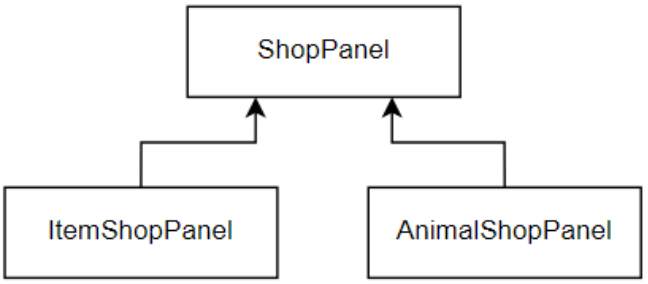
8.2 Diagrama de clases de Item (simplificado)



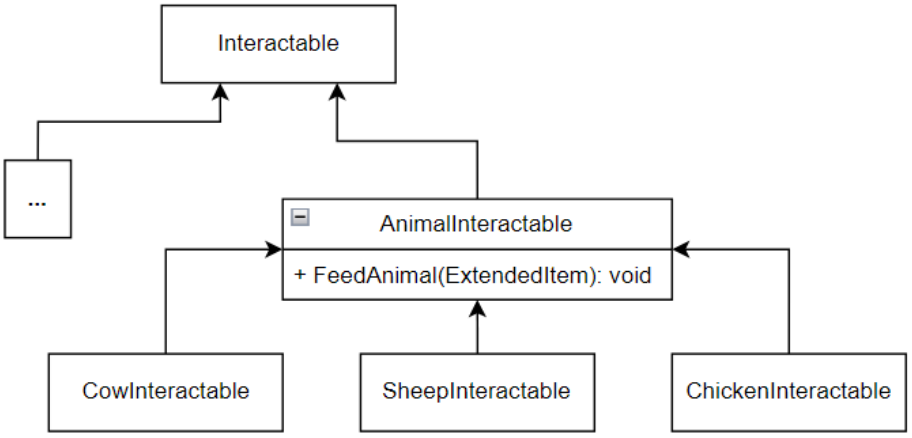
8.3 Diagrama de clases de ItemPanel (simplificado)



8.4 Diagrama de clases de ShopPanel (simplificado)



8.5 Diagrama de clases de AnimalInteractable (simplificado)



Bibliografía

Enlace al repositorio en GitHub del proyecto: https://github.com/alu0101352145/TFG.git

[1] Plataforma de desarrollo en tiempo real de Unity | Motor de VR, AR, 3D y 2D (s.f.). Recuperado de https://unity.com/es

[2] Unity - Scripting API, Unity - Manual: Unity User Manual 2021.3 (LTS) (s.f.). Recuperado de https://docs.unity3d.com/ScriptReference/

[3] Artificial intelligence in video games. [En Wikipedia]. Recuperado (2023, Mayo 14) de https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games

[4] Category:Farming video games [En Wikipedia]. Recuperado (2023, Mayo 14) de https://en.wikipedia.org/wiki/Category:Farming_video_games

[5] Jones, J. (2023, Abril 17). The Future Of AI In Gaming, GameDesigning de https://www.gamedesigning.org/gaming/ai-in-gaming/

[6] González, M. (2023, Enero 26). Cómo se aplica la Inteligencia Artificial en los videojuegos - IIC. Instituto de Ingeniería Del Conocimiento. https://www.iic.uam.es/noticias/como-aplica-inteligencia-artificial-en-videojuegos/

[7] [Greg Dev Stuff]. (2020, Septiembre 28). Stardew Valley like Game in Unity Episode 5 Inventory System Part 1 [Video]. Recuperado de https://www.youtube.com/watch?v=mJLuKrk__tk

[8] Lague, S. [Sebastian Lague]. (2017, Julio 20). RPG graphics E01: Character model [Blender] [Vídeo]. Recuperado de https://www.youtube.com/watch?v=aAO4C_8y0w8

[9] Gius Caminiti [Gius Caminiti]. (2021, Febrero 9). Shader de agua estilizada con Unity Shader Graph [Video]. Recuperado de https://www.youtube.com/watch?v=jRuCQnp78gk

[10] [REM118]. (2019, Junio 10). Unity 3d Shader Graph Lava [Video]. Recuperado de https://www.youtube.com/watch?v=w7qDzz5K8Gg

[11] [Ketra Games]. (2021, Octubre 21). Creating a Day/Night Cycle (Unity Tutorial) [Video]. Recuperado de https://www.youtube.com/watch?v=L4t2c1_Szdk

[12] Owens, B. (2014, Abril 23). Goal Oriented Action Planning for a Smarter AI, Game Development Envato Tuts+ de https://gamedevelopment.tutsplus.com/tutorials/goal-oriented-action-planning-for-a-smarter-ai--cms-20793

[13] De Byl, P. [Holistic3D]. (2022, Agosto 5). Goal-Oriented Action Planning Part 1 [Vídeo]. Recuperado de https://www.youtube.com/watch?v=tdBWk2OVCWc

[14] An Introduction to GOAP - Unity Learn (s.f.). Recuperado de https://learn.unity.com/tutorial/an-introduction-to-goap#

[15] Orkin, J. (s.f.). Goal-Oriented Action Planning (GOAP). Recuperado de http://alumni.media.mit.edu/~jorkin/goap.html

[16] McManus, I. [Iain McManus]. (2021, Diciembre 16). Unity AI Tutorial: Navigation System [Vídeo]. Recuperado de https://www.youtube.com/watch?v=8C5TI0Tam6o

[17] McManus, I. [Iain McManus]. (2021, Septiembre 9). Unity AI Tutorial: Goal Oriented Action Planning [Vídeo]. Recuperado de https://www.youtube.com/watch?v=Q7aHXn_LypI