UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Desarrollo de un videojuego de simulación agrícola en Unity

Trabajo Fin de Grado

Grado en Tecnología Digital y Multimedia

AUTOR/A: Nuevo Orden, David

Tutor/a: González Mollá, Jorge

CURSO ACADÉMICO: 2022/2023

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE INGENIERÍA DE TELECOMUNICACIÓN

Universitat Politècnica de València

Escuela técnica Superior de Ingeniería de Telecomunicación

Desarrollo de un Videojuego de Simulación Agrícola en Unity

Trabajo Fin de Grado

Grado en Tecnología Digital y Multimedia

AUTOR/A: David Nuevo Orden

TUTOR/A: Jorge Gonzalez Molla

CURSO ACADÉMICO: 2022-2023

Resumen

Los videojuegos sobre simular la vida en una granja son, a día de hoy, uno de los géneros más clásicos y que siguen teniendo relevancia en el sector. Estos videojuegos brindan un sentimiento de tranquilidad al consumidor a la vez que lo anima a conseguir mayores logros dentro del mundo virtual.

Este trabajo se centra en la realización de un videojuego orientado en gestionar una granja, lo que conlleva cuidar plantaciones, criar animales y decorar el espacio. El jugador deberá tomar decisiones sobre qué necesita plantar en cada época del año, cómo administrar sus ingresos y en qué dedica el tiempo disponible.Este proyecto hace uso de Unity como eje central para el desarrollo. Se harán uso de distintos scripts en C# para las diferentes mecánicas y para la programación de la lógica de funcionamiento del videojuego. Se utilizarán también elementos como Tilemap para hacer el mapeado del juego.

Resum

Els videojocs sobre simular la vida en una granja son, avui, un dels gèneres més clàssics i que seguixen tenint relevancia al sector. Aquests videojocs brinden un sentiment de tranquilitat al consumidor a la vegada que l'anima a aconseguir assoliments més grans dins del món virtual.

Aquest treball es centra en la realització d'un videojoc orientat en gestionar una granja, el que comporta cuidar les plantacions, criar animals i decorar el espai. El jugador haurà de prendre decisions sobre què necesita plantar en cada època de l'any, com administrar els seus ingresos i en què dedica el temps disponible. Aquest projecte fa ús de Unity com eix central per al desenvolupament. S'utilitzaràn diversos scripts en C# per a les diferents mecàniques del joc i per a la programació de la lògica de funcionament del videojoc. Es faran usos també d'elements com Tilemap per fer el mapa del joc.

Abstract

Video games about simulating life on a farm are, to this day, one of the most classic genres and one that continues to be relevant in the sector. These video games provide a feeling of tranquility to the consumer while encouraging him to achieve greater achievements within the virtual world.

This work focuses on the realization of a video game oriented to manage a farm, which involves taking care of plantations, raising animals and decorating the space. The player will have to make decisions about what he needs to plant at each time of the year, how to manage his income and where he spends the available time. This project makes use of Unity as the central axis for development. Different scripts in C# will be used for the different mechanics and for the programming of the operating logic of the video game. Elements such as Tilemap will also be used to map the game.

1-Introducción..... 1
2-Motivación y objetivos..... 1
3-Estado del Arte..... 2
3.1- Referentes históricos..... 2
3.1.1- Harvest Moon..... 2
3.1.2- Animal Crossing..... 4
3.1.3- Stardew Valley..... 6
4-Planificación y especificación..... 9
4.1- Tecnologías..... 9
4.1.1- Motores de juego..... 9
4.1.2- Visual Studio..... 11
4.2- Estilo artístico del videojuego..... 11
4.3- Metodología de trabajo..... 13
4.4- Planificación de tareas..... 13
4.5- Planificación de Costos..... 19
5-Diseño del proyecto..... 22
6-Implementación..... 23
6.1- Iteración 1- Movimiento Personaje..... 23
6.2- Iteración 2- Objetos Coleccionables..... 26
6.3- Iteración 3- Creación Layout Granja..... 28
6.4- Iteración 4- Inventario Prototipo 1..... 31
6.5- Iteración 5- Interfaz Inventario..... 32
6.6- Iteración 6- Soltar Objetos..... 35
6.7- Iteración 7- Mecánica de Arar tierra..... 39
6.8- Iteración 8- Sistema de Objetos..... 41
6.9- Iteración 9- Barra de Herramientas..... 44
6.10- Iteración 10- Inventario Final..... 46
6.11- Iteración 11- Arrastrar y soltar (Drag and Drop)..... 47
6.12- Iteración 12- Conexión Inventarios..... 49
6.13- Iteración 13- Día y Noche..... 51
6.14- Iteración 14- Sistema de tiempo interno..... 54
6.15- Iteración 15- Sembrar semillas..... 55
6.16- Iteración 16- Funcionalidad Herramientas..... 56
6.17- Iteración 17- Crecimiento cultivos..... 61
6.18- Iteración 18- Cosechar cultivos..... 63
6.19- Iteraciones extra..... 65
7- Líneas futuras..... 67
8- Conclusiones..... 68
9- Bibliografía..... 72
10- Enlaces a archivos..... 74

1-Introducción

La propuesta inicial de este proyecto es la creación de un prototipo de videojuego de simulación agrícola, centrándose en la programación de la lógica de funcionamiento de la mayoría de las mecánicas esenciales para un videojuego de esta temática. Con esto, se sentarán las bases para la ampliación del proyecto a futuro y llevar así la creación del videojuego completo.

El género de simulador agrícola se caracteriza por la gestión de los diferentes elementos a largo plazo, dotando así a los videojuegos del género de un tono pausado y sosegado, que transmite calma y tranquilidad. Crear tu propia granja, decorarla, construir edificios agrícolas, decorar tus propiedades, cuidar tus plantaciones, construir relaciones con los NPC y explorar el mundo suelen ser dinámicas generales en el género. Con esto en mente, el objetivo general de estos juegos difiere mucho de los otros géneros, donde se suelen tener unas finalidades mucho más concretas y a corto plazo.

Esta mezcla de elementos hace que acudan a este tipo de videojuegos jugadores que busquen tranquilidad y relajación, desconectar un poco del mundo y jugar a un juego sin ningún tipo de presión o objetivo claro. Por esto, se trata de dotar al juego de infinidad de posibilidades en torno a la gestión de la partida de cada jugador, sin hacer imprescindible cualquier elemento del juego, dando así la libertad necesaria a los usuarios, sin ningún tipo de obligación o atadura en cuanto a la forma de afrontar el transcurso de la partida.

En esta memoria se describirá el proceso de planteamiento, diseño y desarrollo del proyecto. Se mostrarán las diferentes ideas propuestas, así como la perspectiva de cómo abordar cada mecánica y dinámica realizada.

2-Motivación y objetivos.

Los videojuegos de simulación agrícola son considerados, a día de hoy, uno de los géneros más clásicos de la historia de los videojuegos. A pesar de esto, siempre han tenido su público y su hueco en el mercado que no se ha visto afectado por las tendencias de las diferentes épocas. Gracias a esto, cada año se lanzan una cantidad considerable de estos videojuegos, creando cada vez más aficionados al género.

El objetivo del proyecto es diseñar e implementar las diferentes mecánicas y dinámicas generales características de este tipo de videojuegos. Para esto, se creará un escenario donde el usuario podrá probar todos los elementos desarrollados en una especie de demo técnica. Se dejan de lado pues, aspectos relevantes en el desarrollo de videojuegos tales como el diseño de una historia, diseño del mapeado completo y sistema de misiones para abordar el proyecto desde un punto más técnico.

Entre las mecánicas y dinámicas iniciales que se quieren abordar, se encuentran el diseño de un nivel que se corresponderá con la granja del jugador, que contendrá diferentes elementos característicos del género, desarrollo e implementación de la mecánica de plantación y cultivo de diferentes vegetales, recolección de objetos, diseño y desarrollo de una interfaz para el usuario y creación de un sistema de inventarios.

3-Estado del Arte

El marco teórico de este trabajo se centra en estudiar la historia de los videojuegos del género propuesto, estudiando los referentes históricos y los lanzamientos más consolidados en el ámbito, consiguiendo tener una perspectiva general del nicho y entender mejor el desarrollo del proyecto.

3.1- Referentes históricos

3.1.1- Harvest Moon

Aunque pueda parecer que los videojuegos de gestión agrícola estuvieron presentes desde los inicios de la industria, siendo estos fácilmente imaginables en máquinas de recreativa de la época, no fue hasta 1996, cuando en Japón se publicó el videojuego "Harvest Moon[1]", el primer y máximo referente del género, el cual fue tan inspirador y único, que inspiró a todos los demás videojuegos de temática similar e incluso es considerado el mayor exponente a día de hoy.

Figura 1. Carátula Harvest Moon.

El juego consiste en cuidar una granja que el protagonista ha heredado de su abuelo. Hay que restaurar el terreno donde se encuentra la granja, ya que esta está en un estado deplorable debido a que ha estado años sin mantenimiento. El jugador dedica el tiempo virtual de cada día en el juego a hacer las diferentes tareas disponibles, como limpiar el terreno, plantar cultivos, cosecharlos, regarlos, criar animales, ir a los festivales del pueblo del lado, construir relaciones con los personajes de ese pueblo o explorar.
A los animales se les tiene que dar de comer, a las plantaciones hay que regarlas para que sigan creciendo y hay que hablar con los personajes no jugables (NPC) para mejorar las relaciones. El juego

quería dar la sensación al jugador de que estaba haciendo las tareas del día a día, para que, con tiempo y esfuerzo, construyera una granja de ensueño.

Figura 2. Gameplay Harvest Moon.

Planificar la mejor cosecha,vender los cultivos e introducir herramientas de recolección distinguidas para cada tipo de material, picos para las piedras, hachas para los árboles y una hoz para arar la tierra fueron de los elementos más característicos del título.

Además, el ir construyendo una relación con los habitantes del pueblo a medida que se iba mejorando la granja aportaba un sentimiento de comunidad y hacía sentir al jugador una parte más del pequeño pueblo.

La primera alpha del juego consistía en una porción pequeña de tierra, con unas piedras y árboles para poder probar las mecánicas esenciales del título, como limpiar el terreno, arar la tierra o plantar cultivos teniendo tan solo unas pocas animaciones creadas.

Después del lanzamiento, el videojuego tuvo tanto éxito que salió internacionalmente y, con el paso de los años, se han creado numerosas secuelas, como “Harvest Moon: Island of Happiness[2]”, lanzado en 2008 para nintendo DS, o “Harvest Moon: Light of Hope[3]", lanzado en 2018 para PlayStation 4 y nintendo Switch.

Figura 3. Harvest Moon Nintendo Switch.

3.1.2- Animal Crossing

Animal Crossing es, sin duda alguna, la saga de videojuegos del género de gestión agrícola más conocida del mundo. Desarrollada por la mismísima Nintendo, Animal Crossing vio la luz por primera vez en 2001 para las plataformas Nintendo 64 y GameCube y, como solía ser habitual en la época, fue lanzado únicamente en Japón. No fue hasta un año más tarde, en 2002, después de comprobar que el juego daba beneficios, que se exportó al resto del mundo.

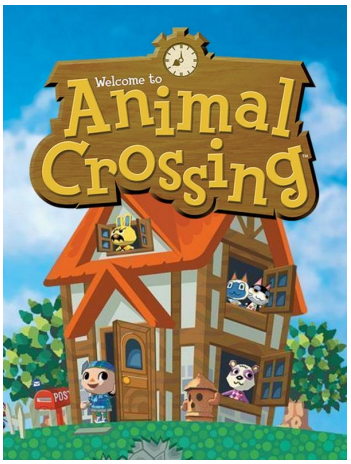


Figura 4. Carátula Animal Crossing 2001.

En un principio, Animal Crossing iba a ser un videojuego RPG sobre mazmorras, pero debido al fracaso de los juegos que estaban lanzando en la plataforma, decidieron crear algo completamente nuevo, que aludiera tanto a las personas habituales en el mundo de los videojuegos como a las personas totalmente ajenas a este ámbito, cosa que logró de manera excepcional. Aún a día de hoy, esta dualidad de público objetivo sigue vigente en la saga, haciendo que millones de personas que no suelen jugar a videojuegos le den una oportunidad a Animal Crossing.

La trama de Animal Crossing, como en la mayoría de juegos del género, es simple, llegas a un pueblo donde habitan animales y formas parte del pueblo, donde creas tu granja, interactúas con animales y exploras el pueblo mientras mejoras tus relaciones con los NPC.

Una de las dinámicas más características del título original fue el jugar en tiempo real, gracias al reloj interno de la gamecube. Esto significaba que si el usuario jugaba por la noche, en el juego era de noche, los eventos anuales como Navidad o Halloween coinciden con las fechas en la realidad.

Figura 5. Invierno en Animal Crossing.

Los jugadores tienen su propia casa, que es el segundo elemento más importante del juego, ya que pueden decorar totalmente a su gusto, con pinturas de pared, muebles y accesorios.

En el mundo virtual los jugadores pueden pescar, obtener insectos, fósiles y pinturas que se podían donar a un museo donde quedaban expuestos para que todos los habitantes del pueblo pudieran visitarlo y ver los diferentes elementos entregados.

Figura 6. Museo en Animal Crossing.

Al principio del juego había 6 habitantes en el pueblo, que más tarde se podrían aumentar hasta 15. Con estos habitantes se podía hablar, ofrecerles ayuda en diferentes tareas, comerciar y escribirles cartas.

Animal Crossing fue un gran éxito en ventas y críticas, obteniendo un 87 sobre 100 en metacritic, página referente en cuanto a análisis de videojuegos, donde se juntan todas las críticas de las diferentes revistas y webs más importantes del sector además de ofrecer otra nota dada únicamente por los usuarios.

Con el paso de los años, se han lanzado cuatro entregas más de la saga principal, una para cada una de las cuatro grandes consolas de Nintendo: Animal Crossing Wild World para Nintendo DS, Animal Crossing City Folk para Nintendo Wii, Animal Crossing New Leaf para Nintendo 3DS y Animal Crossing New Horizons para Nintendo Switch.

Estos lanzamientos han mantenido las características mencionadas anteriormente del título original, siendo fieles a sus raíces pero añadiendo nuevas mecánicas y refinando la fórmula hasta obtener juegos sobresalientes.Todas las entregas han sido de los juegos más vendidos para esa plataforma, vendiendo en Nintendo Switch casi 50 millones de copias y tienen de media una nota de 84 '9 puntos sobre 100, lo cual la convierte en una de las sagas de videojuegos más exitosas de la historia.

Figura 7. Ilustración Animal Crossing 2020.

3.1.3- Stardew Valley

Si antes hemos dicho que Animal Crossing era la saga de videojuegos más conocida del género, es seguro afirmar que Stardew Valley es el mejor juego de simulación agrícola.

Figura 8. Carátula Stardew Valley.

Publicado en febrero de 2016 en Steam, la plataforma de videojuegos más importante en el mundo del PC, y siendo desarrollado por una única persona, Eric Barone, conocido como ConcernedApe, el cual reveló que le llevó cuatro años y medio, dedicando 10 horas diarias, todo el desarrollo del juego, desde el apartado artístico hasta los aspectos más profundos de la programación, todo hecho por él mismo.

Stardew Valley podría considerarse un tributo a Harvest Moon, por la cantidad de elementos que comparten, con dos décadas de diferencia entre lanzamientos. El videojuego ensalza los elementos de su predecesor y añade cantidades ingentes de contenido nuevo y de actividades a realizar, creando un videojuego que conlleva unas 100 o 200 horas de contenido en una sola partida, y altamente rejugable.

Figura 9. Gameplay Stardew Valley.

Como en Harvest Moon, la trama y el sistema principal de Stardew Valley es heredar la granja del abuelo del protagonista, que está en un estado deplorable, e intentar activarla de nuevo a través de la limpieza del terreno, decoración, haciendo cultivos, criando animales y construyendo edificios mientras el jugador va construyendo relaciones personales con la gente del pueblo.

El paso del tiempo divide el juego en cuatro estaciones: otoño, invierno, primavera y verano. En cada una de estas, se pueden plantar cultivos diferentes, ofreciendo gran variedad de estos. También, el aspecto del pueblo y del mundo cambian según la estación. Además, en cada estación hay uno o dos eventos con los personajes del pueblo, como el festival de pascua, donde hay una búsqueda de huevos por el pueblo o el festival de la cosecha en otoño, donde se exponen las mejores cosechas que un jurado puntúa.

Figura 10. Estaciones Stardew Valley.

Se introduce una mazmorra donde se pueden encontrar materiales para mejorar las herramientas y hacer objetos para la granja que está confeccionada por cien niveles diferentes, divididos por capas, añadiendo una componente RPG básica al juego.

También introduce numerosos eventos ya sea con cada uno de los personajes del pueblo, que se desbloquean al llegar a cierto grado de amistad con ellos, como eventos por el mundo, relacionados con magia arcana y espíritus ancestrales.

Cada una de las actividades tiene una gran cantidad de contenido, la pesca, por ejemplo, tiene más de 50 especies que se encuentran en los diferentes tipos de agua en la que el jugador pesque.

Figura 11. Pesca Stardew Valley.

Uno de los sistemas clave del videojuego es el sistema social, donde se ve reflejado el grado de amistad o rivalidad con cada uno de los integrantes del pueblo, desbloqueando diálogos únicos dependiendo de esta estadística. Cada personaje tiene más de 300 líneas de diálogo diferentes, y eventos exclusivos asociados a cada uno.

Figura 12. Diálogos Stardew Valley.

4-Planificación y especificación

4.1- Tecnologías

4.1.1- Motores de juego

Un motor de videojuego es un framework o un grupo de herramientas que permiten a los programadores realizar un desarrollo de un videojuego de una manera más eficaz. Entre las tareas de las que se suelen encargar los motores de videojuegos están la simulación de físicas realistas, añadir y editar sonido, renderizar gráficos, gestionar y administrar la memoria y editar escenarios gráficos. Estos elementos son poco relevantes desde un punto de vista de diseño pero sumamente importantes para el funcionamiento general de cualquier videojuego. Con esto en mente, podemos determinar que su función primordial es la de quitar carga de trabajo a los programadores.

Estos motores deben encargarse del renderizado, para manejar los gráficos del videojuego, que pueden ser en 2D o en 3D y que, a la vez, habilitan al programador de añadir o editar elementos gráficos a las escenas o niveles.

Una de las partes más importantes de estos elementos es el motor de físicas, para simular de una manera creíble interacciones como colisiones, gravedad, lanzamientos y peso.

Además de esto, deben tener una interfaz que sea lo más intuitiva posible para los programadores y permita acceder a todas las funciones del motor de una manera rápida y sencilla, para agilizar el trabajo de estos.

Para acabar, los motores gráficos deben tener un sistema de animación para el control de estas y un sistema para manejar los elementos de código de programación, los scripts.

Analizando el mercado, podemos ver la infinidad de motores gráficos que existen, como el RE Engine de Capcom, Rockstar Advanced Game Engine, Frostbite, Game Maker, UbiArt Framework, Unreal Engine o Unity3D. Cada uno tiene sus fortalezas y flaquezas con respecto a los otros, algunos incluso son de uso exclusivo por las diferentes compañías. En este proyecto analizaremos los últimos dos mencionados ya que son los más utilizados a día de hoy por la gran mayoría de desarrolladores, Unity y Unreal Engine 5.

Unreal Engine

Motor ampliamente conocido a día de hoy por todos los desarrolladores, creado por Epic Games, empresa de videojuegos, en 1998 y actualizado cada ciertos años hasta la versión más actual: Unreal

Engine 5, del que se están empezando a vislumbrar los primeros videojuegos realizados con este motor.

Está altamente orientado al desarrollo en 3D, con una potencia gráfica abismal, de lo mejor del mercado. Utilizado por la mayoría de juegos triple A, los juegos con mayor presupuesto del mercado.

Entre sus ventajas destacamos su licencia gratuita y el material de aprendizaje creado en torno a este motor gráfico, además de la enorme potencia mencionada anteriormente.

En este motor, se utiliza el lenguaje C++, el cual es de los más utilizados a nivel global en multitud de ámbitos. Este lenguaje se ha estudiado en el grado de Tecnología Digital y Multimedia por lo que podemos valorarlo como ventaja.

Destacar que este motor está muy extendido en el sector, por lo que hay muchos tutoriales y guías hechos por la comunidad, que pueden servir tanto para los inicios de programación como para aspectos mucho más profesionales y avanzados.

Unity

Unity es probablemente el motor más utilizado en el mundo por los desarrolladores. Creado por la compañía Unity Technologies y está dedicado tanto al desarrollo de videojuegos como para escenarios en RV y experiencias interactivas.

Orientado tanto al contenido en 3D como en 2D. Utilizado en infinidad de proyectos debido a su versatilidad y su reputación.

Al ser el motor más famoso de la industria, la cantidad de contenido como tutoriales, documentación, guías y assets de la propia tienda supera al del Unreal Engine y a todos los otros motores del sector.

Su licencia también es gratuita y permite la comercialización de los productos hechos con este motor con la condición de que no se superen los 100.000€ de beneficios, por lo que es tanto adecuado para los proyectos pequeños que no se pueden permitir licencias de motores como para proyectos más grandes, ya que al superar esa cantidad no debería de haber problema para adquirir la licencia de pago del motor gráfico.

El lenguaje de programación que utiliza es el C#, derivado del C++, por lo que también es conocido por los estudiantes de GTDM.
La tienda de Unity ofrece muchos elementos a un precio asequible muy potentes, como por ejemplo, Pixel Crushers, un asset que permite gestionar toda la dinámica de los diálogos del juego de una manera muy ágil y altamente personalizable.

Motor escogido

El motor gráfico que se ha seleccionado ha sido Unity. Los factores importantes han sido la mayor facilidad que ofrece a la hora del desarrollo, la mayor cantidad de recursos y documentación

disponibles en el sector, así como los diferentes tutoriales en plataformas como youtube de prácticamente cualquier cosa que existe en el desarrollo de videojuegos.

No obstante, los dos factores que han decantado la balanza a favor de este motor han sido la experiencia previa que se tiene con el manejo de este motor, ya que se ha utilizado durante varias asignaturas de la carrera, como Realidad Virtual o Desarrollo de videojuegos, y la mejor capacidad que tiene el motor de abordar proyectos en 2D, que ha sido el modo de trabajo escogido también debido a su mayor simplicidad en comparación al 3D. Destacar también que los referentes históricos mencionados en el capítulo anterior, Harvest Moon y Stardew Valley, son proyectos realizados en dos dimensiones.

4.1.2- Visual Studio

Un IDE ,entorno de desarrollo integrado (Integrated Development Environment), es una aplicación visual que facilita el desarrollo de código de programación a los desarrolladores. Sus principales funciones es la de acortar varias tareas repetitivas a la hora de escribir líneas de código o implementar funciones, permitiendo escribir parte de los términos a los que se quiere hacer referencia y que el autorellenado los complete por el programador.

Además, son muy útiles para buscar referencias dentro de los diferentes archivos de programación, para llamar a las diferentes propiedades de clases o de otros scripts y su objetivo principal es ahorrar tiempo a los desarrolladores.

El entorno de desarrollo integrado por defecto de Unity es Visual Studio, por lo que se ha escogido como IDE del proyecto, ya que se cuenta con la experiencia previa al trabajar con él en otras asignaturas y usar su variante Visual Studio Code para todas las otras asignaturas que tienen que ver con la programación en general.

Además, cuenta con un depurador que resulta muy útil a la hora de encontrar errores de programación al tener varios scripts funcionando simultáneamente como va a ser el caso de este proyecto.

4.2- Estilo artístico del videojuego

Si analizamos los estilos artísticos de los referentes históricos mencionados en el punto 3.1, podemos agrupar a Harvest Moon y Stardew Valley en un estilo artístico retro-pixelado y a Animal Crossing en un estilo más caricaturesco .

Después de un estudio de varios estilos artísticos propios del género y de las tendencias de los últimos años, se ha decidido apostar por una estética denominada “cozy art”, que hace referencia a aquellas que transmiten un sentimiento acogedor y cómodo. Esto encaja a la perfección con el proyecto ya que los juegos del género quieren transmitir calma y serenidad. Este estilo artístico se caracteriza por el uso de colores pastel, creando una sensación de suavidad

Se ha decidido para el desarrollo de este proyecto comprar los assets gráficos en la tienda de itch.io, por lo que se ha indagado en la tienda viendo las diferentes posibilidades que habían a un precio asequible para el usuario, no más de 20€. Esto se debe en parte a la inexperiencia del desarrollador a la hora de crear arte digital y al deseo de este de obtener un proyecto más agradable visualmente, teniendo en cuenta la gran cantidad de artistas que publican sus creaciones en varias tiendas del sector, sabiendo así, que el producto artístico está ya preparado para ser implementado en videojuegos y todas las consecuencias que eso conlleva, como por ejemplo, un personaje, deberá de tener 24 dibujos o más diferentes en varias posiciones para las animaciones que conlleva el 2D.

Finalmente, se han encontrado dos paquetes de recursos gráficos que encajan a la perfección con el proyecto, los cuales son “Cozy Farm Asset Pack[0]” y “Cozy People Asset Pack[0]”, ambos creados por *Shubibubi*[0]. Ambos se pueden adquirir en la tienda en dos versiones, la gratuita que lleva una cantidad reducida de recursos gráficos y la completa que está el paquete completo. Las versiones completas cuestan 5€, por lo que cumplen el requisito de ser asequibles y además ofrecen licencia completa para la comercialización de cualquier videojuego realizado con estos recursos gráficos.

El primer paquete cuenta con infinidad de elementos relacionados con granjas, como animales, cultivos, verduras, frutas, edificios de agricultura, suelos y elementos para la creación de un mapeado ya dividido en pequeñas porciones.

Figura 13. Paquete de Recursos Gráficos Agrícolas.

El segundo, “Cozy People Asset Pack” va más dirigido a la creación de personajes y las animaciones de estos al usar cualquier herramienta o al hacer diferentes acciones como caminar, saltar o pescar, por lo que resulta ideal para varias necesidades del proyecto.

Figura 14. Paquete de Recursos Gráficos de Personajes.

4.3- Metodología de trabajo

Se ha optado por seguir una metodología de trabajo iterativa o incremental, que consiste en ir sumando elementos o funcionalidades una a una, es decir, planificar un elemento, diseñarlo, implementar ese elemento y probar su correcto funcionamiento. Una vez que la funcionalidad funcione como debe, se pasa al siguiente elemento. Se establecerán una serie de objetivos para abordar durante el transcurso del proyecto.

Se ha escogido esta modalidad por ser la más adecuada para el desarrollo de un videojuego por un principiante en programación, simplifica las implementaciones ya que suceden de una en una y se evitan problemas derivados de llevar a cabo varias modificaciones a la vez.

Con esta metodología es más sencillo medir el progreso y la evolución del proyecto y permite ver el comportamiento de las diferentes funcionalidades cohesionadas entre ellas, ya que en este proyecto funcionarán varios componentes de programación simultáneamente o se llamarán unos a otros dependiendo de las interacciones del usuario.

Con respecto a esta modalidad, la principal ventaja es que se puede observar la evolución del proyecto con cada iteración, viendo aumentado el valor del trabajo. Además, se reduce la complejidad del desarrollo ya que se van añadiendo las funcionalidades poco a poco y se pueden delimitar de una manera mucho más sencilla los posibles errores en la programación.

En el caso concreto de este proyecto, se va a apostar por iteraciones cortas, abordando en cada una mecánicas concretas, en vez de intentar agrupar cada mecánica en algún grupo mayor.

Al ser un trabajo individual, se suprime la principal desventaja o dificultad de utilizar esta metodología, que es la del estrecho contacto necesario con el cliente, ya que no es necesario su visto bueno con respecto a cada iteración diseñada.

Planteado esto, el inconveniente que surge es la dificultad de definir qué tan grande es una funcionalidad, por lo que se deberá reflexionar adecuadamente en el proceso de planteamiento de cada iteración.

4.4- Planificación de tareas

La descomposición en tareas del proyecto planteado queda de la siguiente manera:

Identificador	Tarea
1	Pre-Producción
1.1	Investigación Referentes Históricos
1.2	Redacción Marco Teórico
1.3	Redacción Introducción
1.4	Redacción Motivaciones y Objetivos
1.5	Planificación
1.5.1	Asignación de Tareas
1.5.2	Recursos
1.5.3	Tiempo
1.5.4	Costes
1.6	Redacción Tecnologías
1.7	Redacción Estilo Artístico
2	Producción
2.1	Análisis General
2.2	Desarrollo Iteración 1: Movimiento Personaje
2.2.1	Planteamiento Mecánica
2.2.2	Selección sprites
2.2.3	Desarollo logica del movimiento (script)
2.2.4	Configurar Animator Unity
2.2.5	Pruebas Iteracion
2.3	Desarrollo Iteración 2: Objetos Coleccionables
2.3.1	Planteamiento Mecánica
2.3.2	Selección sprites
2.3.3	Desarollo logica (script)
2.3.4	Configurar elementos en Unity
2.3.5	Pruebas Iteracion
2.4	Desarrollo Iteración 3: Creación Layout Granja
2.4.1	Planteamiento
2.4.2	Configurar Tilemap
2.4.3	Diseño del mapa
2.4.4	Creación distintas Capas

Figura 15. Distribución Tareas 1.

2.5	Desarrollo Iteración 4: Inventario Propotipo 1
2.5.1	Planteamiento Mecánica
2.5.2	Selección elementos gráficos
2.5.3	Desarollo logica (script): Inventory
2.5.4	Configurar elementos en Unity: Canvas, Imagenes
2.5.5	Pruebas Iteracion
2.6	Desarrollo Iteración 5: Interfaz Inventario
2.6.1	Planteamiento Mecánica
2.6.2	Selección elementos gráficos
2.6.3	Desarollo logica (script): UI_Manager
2.6.4	Configurar elementos en Unity: Canvas, Imagenes
2.6.5	Pruebas Iteracion
2.7	Desarrollo Iteración 6: Soltar Objetos
2.7.1	Planteamiento Mecánica
2.7.2	Selección elementos gráficos
2.7.3	Desarollo logica (script): Player.cs, Inventory.cs
2.7.4	Configurar elementos en Unity
2.7.5	Pruebas Iteracion
2.8	Desarrollo Iteración 7: Mecánica de Arar tierra
2.8.1	Planteamiento Mecánica
2.8.2	Selección elementos gráficos
2.8.3	Desarollo logica (script): TileManager.cs
2.8.4	Configurar elementos en Unity: Tilemap, GameManager
2.8.5	Pruebas Iteracion
2.9	Desarrollo Iteración 8: Sistema de Objetos
2.9.1	Planteamiento Mecánica
2.9.2	Selección elementos gráficos
2.9.3	Desarollo logica (script): Item.cs, ItemData.cs, ItemManager.cs
2.9.4	Configurar elementos en Unity: Prefabs, creación Items
2.9.5	Pruebas Iteracion

Figura 16. Distribución Tareas 2.

2.10.0	Desarrollo Iteración 9:Inventario Final
2.10.1	Planteamiento Dinámica
2.10.2	Selección elementos gráficos
2.10.3	Desarollo logica (script): InventoryManager.cs, Inventory.cs
2.10.4	Configurar elementos en Unity
2.10.5	Pruebas Iteracion
2.11	Desarrollo Iteración 10: Barra de Herramientas
2.11.1	Planteamiento Mecánica
2.11.2	Selección elementos gráficos
2.11.3	Desarollo logica (script): ToolBar_UI.cs, Inventory.cs
2.11.4	Configurar elementos en Unity
2.11.5	Pruebas Iteracion
2.12	Desarrollo Iteración 11: Arrastrar y soltar(Drag and Drop)
2.12.1	Planteamiento Mecánica
2.12.2	Selección elementos gráficos
2.12.3	Desarollo logica (script): Player.cs, Inventory.cs, InventoryManager.cs
2.12.4	Configurar elementos en Unity:Canvas, GameManager
2.12.5	Pruebas Iteracion
2.13	Desarrollo Iteración 12: Conexión Inventarios
2.13.1	Planteamiento Mecánica
2.13.2	Selección elementos gráficos
2.13.3	Desarollo logica (script): Inventory.cs, InventoryManager.cs
2.13.4	Configurar elementos en Unity
2.13.5	Pruebas Iteracion
2.14	Desarrollo Iteración 13: Dia y Noche
2.14.1	Planteamiento Mecánica
2.14.2	Selección elementos gráficos
2.14.3	Desarollo logica (script): DayTimeController.cs
2.14.4	Configurar elementos en Unity: URP, Light
2.14.5	Pruebas Iteracion

Figura 17. Distribución Tareas 3.

2.15	Desarrollo Iteración 14: Sistema de tiempo interno
2.15.1	Planteamiento Mecánica
2.15.2	Selección elementos gráficos
2.15.3	Desarollo logica (script): TimeAgent.cs
2.15.4	Configurar elementos en Unity: GameManager
2.15.5	Pruebas Iteracion
2.16	Desarrollo Iteración 15: Sembrar semillas
2.16.1	Planteamiento Mecánica
2.16.2	Selección elementos gráficos
2.16.3	Desarollo logica (script): Player.cs, TileManager.cs
2.16.4	Configurar elementos en Unity: Tilemaps
2.16.5	Pruebas Iteracion
2.17	Desarrollo Iteración 16: Funcionalidad Herramientas
2.17.1	Planteamiento Mecánica
2.17.2	Selección elementos gráficos
2.17.3	Desarollo logica (script): GatherResourceNode.cs,ToolManager.cs
2.17.4	Configurar elementos en Unity: ToolActions
2.17.5	Pruebas Iteracion
2.18	Desarrollo Iteración 17: Crecimiento cultivos
2.18.1	Planteamiento Mecánica
2.18.2	Selección elementos gráficos
2.18.3	Desarollo logica (script): TileManager.cs, ToolManager.cs,Player.cs
2.18.4	Configurar elementos en Unity
2.18.5	Pruebas Iteracion
2.19	Desarrollo Iteración 18: Cosechar cultivos
2.19.1	Planteamiento Mecánica
2.19.2	Selección elementos gráficos
2.19.3	Desarollo logica (script): OnTilePickUp.cs, TileManager.cs
2.19.4	Configurar elementos en Unity
2.19.5	Pruebas Iteracion
3	Post-Producción
3.1	Pruebas de Prototipo Final
3.2	Creación Ejecutable
3.3	Redacción Memoria

Figura 18. Distribución Tareas 4.

Además, se ha hecho una estimación del tiempo necesario para llevar a cabo cada una de estas tareas. Se han agrupado los pasos de cada iteración para obtener una idea general de lo que conlleva cada una de las 18 iteraciones propuestas.

Al final de la estimación, se ha añadido un margen de un 25% pensando en la alta probabilidad de la aparición de problemas no previstos debido a la poca experiencia del programador.

Identificador	Tarea	Tiempo previsto (horas)
1	Pre-Producción	48
1.1	Investigación Referentes Históricos	10
1.2	Redacción Marco Teórico	3
1.3	Redacción Introducción	2
1.4	Redacción Motivaciones y Objetivos	3
1.5	Planificación	24
1.5.1	Asignación de Tareas	8
1.5.2	Recursos	5
1.5.3	Tiempo	6
1.5.4	Costes	5
1.6	Redacción Tecnologías	4
1.7	Redacción Estilo Artístico	2
2	Producción	240
2.1	Análisis General	30
2.2	Desarrollo Iteración 1: Movimiento Personaje	6
2.3	Desarrollo Iteración 2: Objetos Coleccionables	10
2.4	Desarrollo Iteración 3: Creación Layout Granja	8
2.5	Desarrollo Iteración 4: Inventario Propotipo 1	15
2.6	Desarrollo Iteración 5: Interfaz Inventario	10
2.7	Desarrollo Iteración 6: Soltar Objetos	7
2.8	Desarrollo Iteración 7: Mecánica de Arar tierra	10
2.9	Desarrollo Iteración 8: Sistema de Objetos	16
2.10.0	Desarrollo Iteración 9:Inventario Final	20
2.11	Desarrollo Iteración 10: Barra de Herramientas	9
2.12	Desarrollo Iteración 11: Arrastrar y soltar(Drag and Drop)	8

Figura 19. Distribución Tiempo Tareas 1.

2.13	Desarrollo Iteración 12: Conexión Inventarios	11
2.14	Desarrollo Iteración 13: Dia y Noche	8
2.15	Desarrollo Iteración 14: Sistema de tiempo interno	8
2.16	Desarrollo Iteración 15: Sembrar semillas	9
2.17	Desarrollo Iteración 16: Funcionalidad Herramientas	24
2.18	Desarrollo Iteración 17: Crecimiento cultivos	15
2.19	Desarrollo Iteración 18: Cosechar cultivos	16
3	Post-Producción	53
3.1	Pruebas de Prototipo Final	10
3.2	Creación Ejecutable	3
3.3	Redacción Memoria	40
	TOTAL HORAS	**341**
	TOTAL HORAS despues del 25%	**426.25**

Figura 20. Distribución Tiempo Tareas 2.

Después de sumar el 25% a cada tarea obteniendo las horas necesarias se plantea el siguiente diagrama de Gantt:

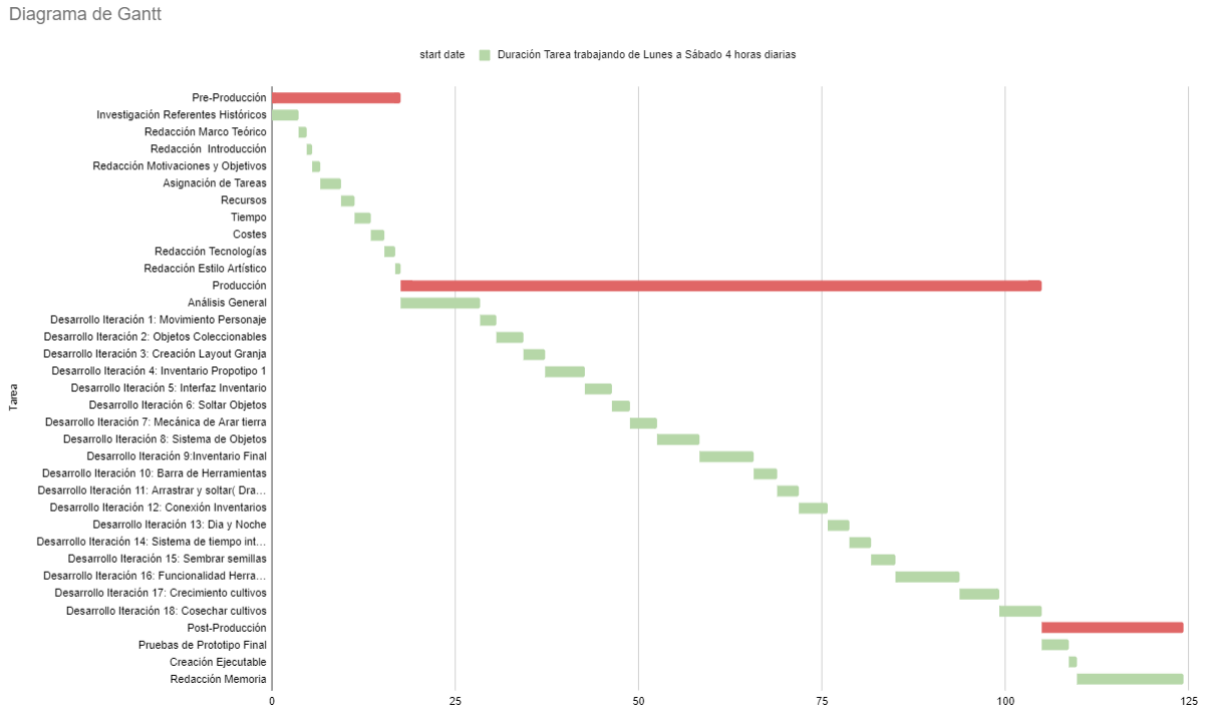


Figura 21. Diagrama de Gantt del proyecto.

En el momento de hacer diagrama de Gantt se ha propuesto como eje de tiempo los días necesarios trabajando de lunes a sábado durante cuatro horas cada día. Siguiendo estos parámetros, son necesarios 125 días, lo que se traduce en 4 meses y 3 días, por lo que, al comenzar el proyecto a mediados de mayo se estima su finalización a mediados de agosto, siendo la convocatoria de septiembre el objetivo para la defensa del proyecto.

En caso de necesitar más horas de las previstas se puede introducir el domingo como día para avanzar más trabajo o se pueden incrementar las horas de los jueves, viernes y sábados debido a la disponibilidad del desarrollador.

4.5- Planificación de Costos

Para llevar a cabo la estimación de costes, se va a tener en cuenta el material informático utilizado, el costo del software, de los diferentes recursos gráficos y el sueldo de un programador junior, es decir, con poca o ninguna experiencia, y el de un diseñador de videojuegos junior.

Por lo que respecta al coste de los materiales del proyecto, sean estos de hardware o de software, se ha realizado la siguiente tabla:

	Costes Material		
Nombre	Precio por unidad	Cantidad	Coste
Ordenador Personal	1.200€	1	1.200€
Gimp- GNU Image Manipulation Program	0€	1	0€
Unity 2021.3.13f1	0€	1	0€
Recursos gráficos	5€	2	10€
Visual Studio 2019	0€	1	0€
Coste Total			1.210€

Figura 22. Costos Materiales.

Como podemos observar, una gran ventaja en el planteamiento del proyecto es la nula necesidad de uso de software de pago, se ha escogido todo el software teniendo en cuenta su posibilidad de ser gratuito, por lo que reduce enormemente el gasto inicial e incrementa la viabilidad de este proyecto.

Después, en el coste humano se ha tenido en cuenta la necesidad de únicamente dos perfiles, el de diseñador de videojuegos y el de programador junior. El primero estará presente durante las tres fases del proyecto y el segundo únicamente en la segunda fase, la de producción. Para el programador junior se ha encontrado esta media de sueldo:

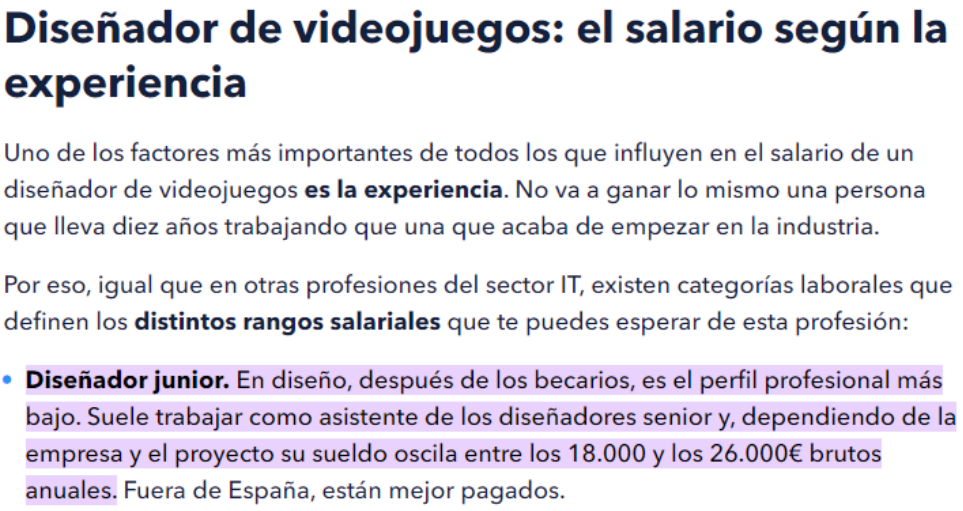
¿Cuál es el sueldo de un programador especializado en Unity?

El sueldo de un programador con Unity depende de distintos factores, los cuáles desarrollaremos más adelante. Teniendo esto en cuenta, es complicado hablar sobre una cifra exacta para la remuneración de estos profesionales. Por eso, tras distintas consultas en distintos portales de empleo, **hemos elaborado una horquilla salarial.**

Teniendo en cuenta distintas ofertas de empleo de programación de videojuegos, con especialidad en Unity, podemos hablar de que el sueldo de estos profesionales se mueve entre los **18.700€ y los 29.430€ brutos anuales.**

Figura 23. Artículo sobre sueldo de programador.

Y para el diseñador de videojuegos se ha obtenido esta información:



Diseñador de videojuegos: el salario según la experiencia

Uno de los factores más importantes de todos los que influyen en el salario de un diseñador de videojuegos **es la experiencia**. No va a ganar lo mismo una persona que lleva diez años trabajando que una que acaba de empezar en la industria.

Por eso, igual que en otras profesiones del sector IT, existen categorías laborales que definen los **distintos rangos salariales** que te puedes esperar de esta profesión:

- **Diseñador junior.** En diseño, después de los becarios, es el perfil profesional más bajo. Suele trabajar como asistente de los diseñadores senior y, dependiendo de la empresa y el proyecto su sueldo oscila entre los 18.000 y los 26.000€ brutos anuales. Fuera de España, están mejor pagados.

Figura 24. Artículo sobre sueldo de diseñador 1.

Diseñador de videojuegos: ¿cuánto cobra de media?

Como uno de los especialistas que juega un papel central en la creación de estos productos audiovisuales, vamos a la horquilla de cuánto puede cobrar un diseñador de videojuegos en España.

Actualmente, el sueldo medio inicial de un diseñador de videojuegos en España se sitúa en torno a los 18.000 euros brutos anuales, pero puede llegar incluso hasta y los 50.000.

Hay una diferencia abismal en este rango, ¿verdad? El motivo principal de este desfase es la experiencia que se tenga las espaldas y, por supuesto, el talento.

Figura 25. Artículo sobre sueldo de diseñador 2.

Al encontrarse los dos perfiles necesarios en un rango similar de coste, se va a estimar un salario de 20.000€ anuales, al tratarse de un proyecto bastante comedido. Con esto, podemos concluir la siguiente tabla con los costes humanos:

	Costes Humanos			
Nombre	Coste mensual	Horas	Coste por hora	Coste
Diseñador Videojuegos	1.665€	426	10.4€	4.430€
Programador Junior Unity	1.665€	300	10.4€	3.120€
Coste Total				7.550€
Coste Total + IVA 21%				9.130€

Figura 26. Costes Humanos.

Si sumamos estos costes a los anteriores y agregamos un margen del 15% obtenemos:

	Costes Totales			
Coste Materiales				1.210€
Costes Humanos sin IVA				7.550€
Coste Total sin margen				8.760€
Margen 15%				1.315€
Coste Total con Margen				10.075€
Coste total con Margen e IVA				12.190€

Figura 27. Costes Totales.

5-Diseño del proyecto

Como se ha mencionado anteriormente, el objetivo es el desarrollo de un prototipo de un videojuego de simulación agrícola al estilo de los máximos referentes del género, como Animal Crossing o Stardew Valley.

Para seguir el ejemplo de los mayores exponentes del género, se ha concluido en diseñar una demo similar a la que publicó en su día el estudio a cargo del primer Harvest Moon, cuya finalidad fue la de mostrar las diferentes mecánicas base de su videojuego a los aficionados y accionistas de la compañía.

Por lo tanto, el prototipo deberá de contar con un nivel general donde el jugador sea capaz de moverse libremente por este y, al mismo tiempo, probar las diferentes mecánicas que se desarrollarán en los incrementos propuestos durante la planificación del proyecto.

Para un desarrollo satisfactorio del trabajo, se han propuesto 18 incrementos o iteraciones a desarrollar, algunas de las cuales estarán estrechamente relacionadas con otras para poder vislumbrar el funcionamiento cruzado entre estas y poder discernir de una manera más clara cómo funcionaría el proyecto completo.

Destacar que cada uno de estas iteraciones se planeará, diseñará y programará teniendo en cuenta el factor de la escalabilidad, pensando en una futura expansión del proyecto hacía un videojuego completo y así evitar un posible rediseño necesario de los sistemas base planteados en este trabajo.

Como referencia para el planteamiento de las diferentes iteraciones se ha escogido el videojuego Stardew Valley ya que es el referente más actual y el usuario dispone de amplios conocimientos del título, por lo que es capaz de observar y entender cuales son las mecánicas más importantes del juego y desgranarlas en los diferentes elementos necesarios de programación para el desarrollo de estas.

Con respecto al diseño puramente artístico, como se ha comentado anteriormente, se han adquirido dos paquetes de recursos gráficos pensados íntegramente para el desarrollo de videojuegos por lo que la tarea artística se reduce exponencialmente.

Dentro de la explicación de cada iteración se hará referencia a la fase de diseño dentro del subapartado de “planteamiento de mecánica”. Esto se debe a la metodología escogida, donde se planeará y desarrollará cada iteración de manera iterativa, una después de la otra.

Abordados los puntos anteriores, estas son las 18 iteraciones planificadas:

Desarrollo Iteración 1: Movimiento Personaje
Desarrollo Iteración 2: Objetos Coleccionables
Desarrollo Iteración 3: Creación Layout Granja
Desarrollo Iteración 4: Inventario Prototipo 1
Desarrollo Iteración 5: Interfaz Inventario
Desarrollo Iteración 6: Soltar Objetos
Desarrollo Iteración 7: Mecánica de Arar tierra
Desarrollo Iteración 8: Sistema de Objetos
Desarrollo Iteración 9: Barra de Herramientas
Desarrollo Iteración 10: Inventario Final
Desarrollo Iteración 11: Arrastrar y soltar (Drag and Drop)
Desarrollo Iteración 12: Conexión Inventarios
Desarrollo Iteración 13: Día y Noche
Desarrollo Iteración 14: Sistema de tiempo interno
Desarrollo Iteración 15: Sembrar semillas
Desarrollo Iteración 16: Funcionalidad Herramientas
Desarrollo Iteración 17: Crecimiento cultivos
Desarrollo Iteración 18: Cosechar cultivos

Tabla 1. Tabla Iteraciones del Proyecto.

6-Implementación

6.1- Iteración 1- Movimiento Personaje

Planteamiento Mecánica

Para este primer incremento se desea la realización del movimiento básico del personaje controlable por el jugador. Los objetivos que se definen son los de mover al personaje en las direcciones de arriba, izquierda, abajo y derecha según el usuario pulse las teclas WASD, respectivamente.

Si el jugador presiona a la vez dos de las teclas asignadas a las direcciones, el personaje se desplazará en dirección diagonal, según las dos teclas que se presionen, por ejemplo: al presionar W y A, el personaje de desplazará diagonalmente hacia arriba a la izquierda.

También se desea que según la dirección en la que se esté trasladando el personaje se reproduzca una animación determinada para cada una de las direcciones disponibles, por lo que se deberán crear a

partir de los sprites (imágenes que se pueden mover independientemente del fondo) del paquete de recursos adquirido.

Estas animaciones se deberán configurar para que viren de unas a otras según ciertas condiciones que se cambiarán desde los scripts.

Selección sprites

Después de analizar el paquete de recursos y determinar la naturaleza de este trabajo, tratándose de un prototipo, se ha escogido un archivo de sprites de un personaje simple, sin decoración, prendas o pelo en este, ya que cumple perfectamente la función deseada y además es más apropiado para que, en la realización del proyecto completo, se pueda crear una dinámica que permita la personalización del personaje sobre esta misma base.

Por lo tanto, estos son los sprites que se utilizarán para esta iteración:

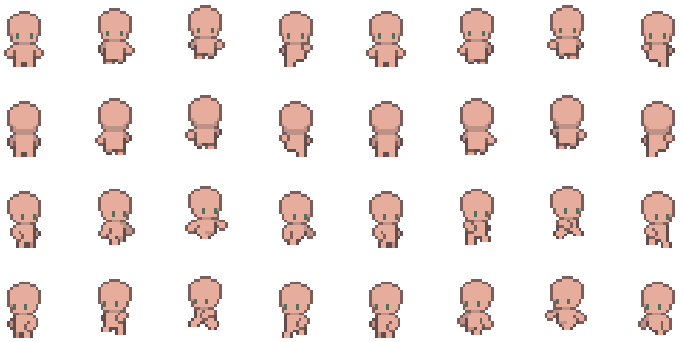


Figura 28. Sprites Personaje.

Como podemos observar, hay un total de 32 sprites en un solo archivo, esto se debe a que se necesitarán 8 sprites para cada una de las cuatro animaciones relacionadas con el movimiento del personaje.

Desarrollo scripts

Para el desarrollo de la lógica de funcionamiento de esta iteración se ha creado el script de “Movement.cs” . Dentro de este archivo se encuentra el código que es capaz de detectar en todo momento la posición de nuestro personaje dentro del juego y analiza si esta es diferente a una anterior y, en caso afirmativo, llama a una función para actualizar la posición del personaje y a otra para reproducir la animación deseada:

```
15      // Update is called once per frame
        Mensaje de Unity | 0 referencias
16      private void Update()
17      {
18          float horizontal = Input.GetAxisRaw("Horizontal");
19          float vertical = Input.GetAxisRaw("Vertical");
20
21          direction = new Vector3(horizontal,vertical);
22
23          AnimateMovement(direction);
24          if (horizontal != 0 || vertical != 0)
25          {
26              lastMotionVector = new Vector2(horizontal, vertical).normalized;
27              animator.SetFloat("lastHorizontal", horizontal);
28              animator.SetFloat("lastVertical", vertical);
29          }
30      }
```

Figura 29. Fragmento Código Movement.cs.

Por lo que respecta a la función de animar el movimiento, esta se encarga de actualizar los valores de verticalidad y horizontalidad del jugador en el "animator", que es la componente del unity encargada de las animaciones.

Configurar Unity

Una vez hecha la lógica de funcionamiento, se deben crear y configurar las animaciones de las direcciones y asignarle el script creado en el apartado anterior al componente del personaje del juego, para que sea capaz de reconocer su posición.

A la hora de configurar las animaciones, se ha decidido crear un controlador para todas las animaciones del personaje, previniendo posibles necesidades.

Para este controlador se crea un estado donde el personaje aparecerá quieto (Idle) y un árbol de estados para controlar las cuatro direcciones:

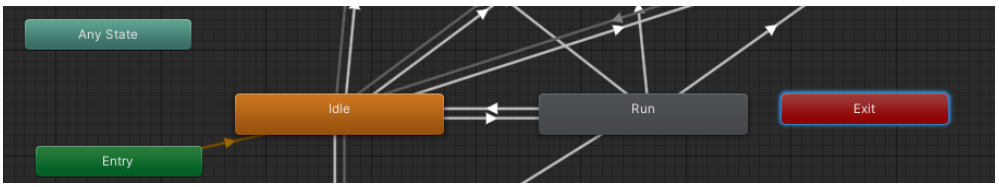


Figura 30. Captura de pantalla Animator General.

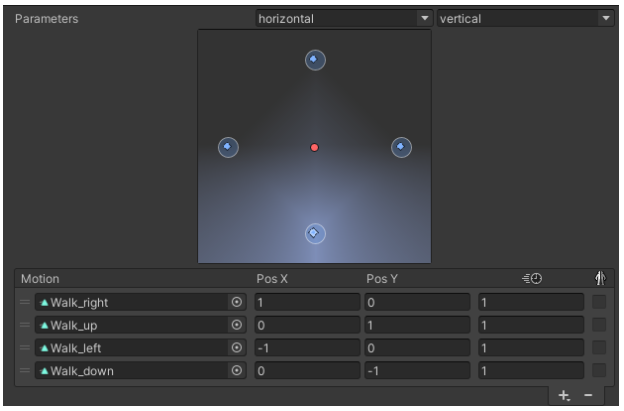


Figura 31. Captura de pantalla Animator Direcciones.

Dentro del árbol de estados configuramos los valores necesarios para que se reproduzcan cada una de las animaciones, que serán dados por el script del punto anterior, el cual pasará automáticamente los valores de verticalidad y horizontalidad.

Pruebas

En el momento de las pruebas, el único error que se encontró fue la mala configuración de las direcciones izquierda y derecha, que aparecían en el momento incorrecto al estar puestas al revés.

6.2- Iteración 2- Objetos Coleccionables

Planteamiento Mecánica

La idea de esta dinámica es darle la capacidad a nuestro personaje de recoger ciertos objetos que se encuentren en el suelo del mapa y, en un futuro, añadirlos al inventario. Por lo tanto, necesitamos definir el tipo de objetos que podrán ser recogidos por el personaje y darle la capacidad al personaje de detectar estos objetos, para eso se añadirán cajas de físicas y colisiones a estos elementos y Unity se encargará automáticamente de detectar si el personaje está encima de un objeto.

Selección sprites

En la selección de sprites únicamente seleccionaremos un objeto a modo de prueba, para poder crear un prefab posteriormente y poder añadir más tipos de objetos. Se ha escogido una imagen de una bolsa de semillas de zanahoria:

Figura 32. Sprite Semillas de Zanahoria.

Desarrollo scripts

Para la parte de la lógica de funcionamiento, se ha creado el script "Collectable.cs", el cual se encarga únicamente de detectar la colisión producida por el jugador una vez se sitúa encima de algún objeto asociado a este código:

```
6   public class Collectable: MonoBehaviour
7   {
8
        Mensaje de Unity | 0 referencias
9       private void OnTriggerEnter2D(Collider2D collision)
10      {
11          Player player = collision.GetComponent<Player>();
12          Destroy(this.gameObject);
```

Figura 33. Fragmento Código Collectable.cs.

Además, se destruirá el objeto una vez se detecte esta colisión, haciéndolo desaparecer de la escena. Más adelante, en el momento que se cree el sistema de inventario, antes de destruir el objeto se llamará a una función para añadir el objeto al inventario del jugador.

Configurar Unity

Esta vez, la configuración es simple, se añade la imagen de las semillas a la escena, lo que creará un componente al cual le asignaremos el código del apartado anterior. Además, a este componente se le añadirá también un "Box Collider 2D" y un "RigidBody 2D", que se encargarán de detectar las colisiones y las físicas del objeto. Nos aseguramos de marcar la casilla de "Is Trigger" para que el jugador pueda atravesar estos objetos en vez de actuar como si fueran inamovibles.

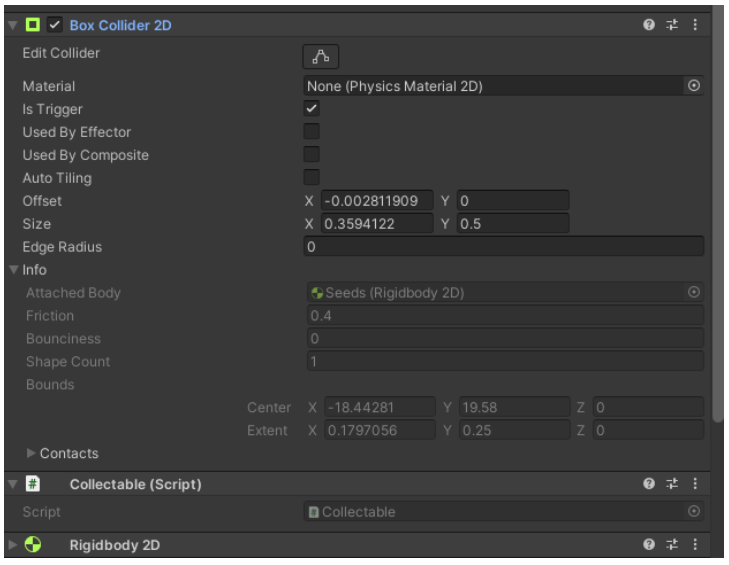


Figura 34. Físicas de Recogibles.

Pruebas

Las pruebas han consistido en ejecutar el juego y comprobar que las semillas son destruidas al pasar por encima de ellas, lo cual ha resultado sin imprevistos.

6.3- Iteración 3- Creación Layout Granja

Planteamiento

En esta iteración se busca diseñar el nivel básico de la granja en la que se encuentra el jugador, tendremos en cuenta referentes de niveles de Stardew Valley para el diseño de un mapeado que contenga ciertas características:

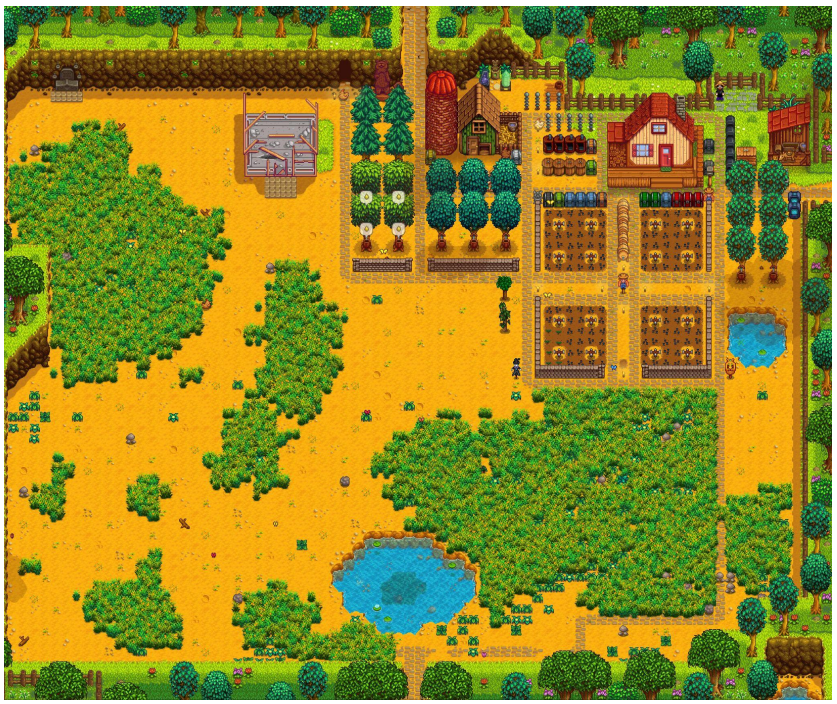


Figura 35. Ejemplo Mapeado Stardew Valley.

Como se puede apreciar en la imagen con la estructura del mapa en Stardew Valley, el mapeado debe contar con ciertos elementos, como pueden ser el agua, la disposición de una casa, amplios terrenos sobre los que construir o cultivar, y una serie de caminos por lo que el jugador pueda salir hacia otras partes del juego.

Además, se va a implementar un elemento divisorio en el mapeado de este proyecto, para que el jugador en cierto momento del juego completo sea capaz de desbloquear más parte de su granja personal, introduciendo un objetivo indirecto que hará que el usuario tenga ganas de cumplir los requisitos necesarios para ampliar su terreno jugable.

Selección sprites

Para el desarrollo de esta iteración se va a emplear Tilemaps, una herramienta que permite la creación de mapas a partir de pequeños cuadros, teniendo un uso similar a una libreta de dibujo. Esta es la paleta que se utilizará para dibujar:

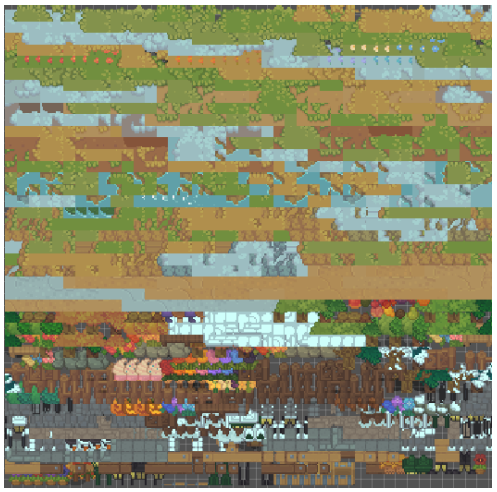


Figura 36. Paleta de Recuadros disponibles.

Y esta es la imágen seleccionada para la casa del jugador:

Figura 37. Sprite Casa Jugador.

Desarrollo

Esta iteración no cuenta con la necesidad de desarrollo de lógica de funcionamiento, por lo que se hará referencia al desarrollo del mapeado en sí, el cual se ha estructurado en capas. Se ha dividido por razones de comunidad a la hora de hacer alusión futuramente a estas capas para añadir o modificar elementos de estas desde los scripts.

Así ha finalizado la creación del mapeado:

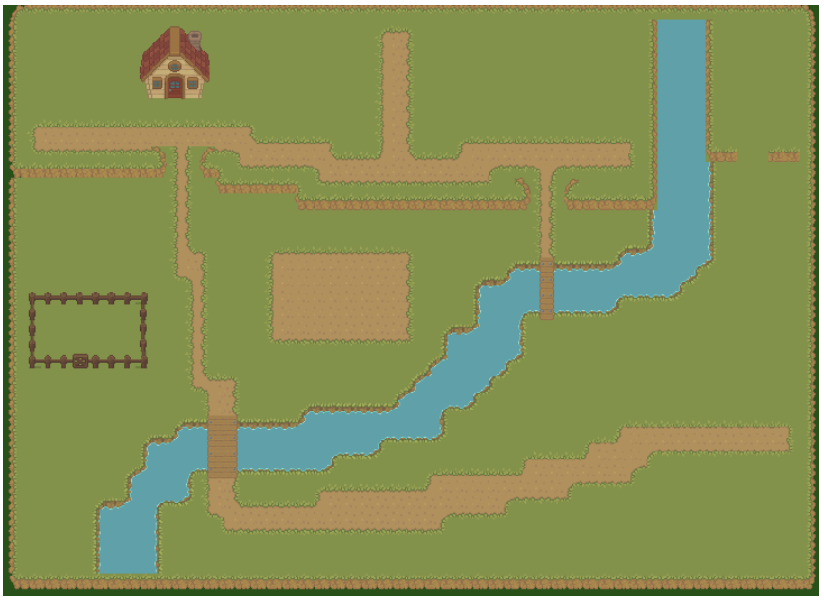


Figura 38. Mapeado desarrollado.

Se ha abogado por un diseño simple pero que incluyera los elementos mencionados anteriormente. Por una parte nos encontramos la casa del personaje arriba a la izquierda. Por otra parte, encontramos tres caminos por los que el jugador podrá salir en el videojuego completo y por último visualizamos el elemento divisorio que hemos planteado, en este caso un río. La idea principal es que los dos puentes que observamos estén rotos al inicio y el jugador sea capaz de repararlos con ciertos requisitos.

Configurar Unity

La configuración de Unity en este incremento se trata de la creación de las diferentes capas Tilemap y agruparlas todas bajo un mismo objeto vacío.

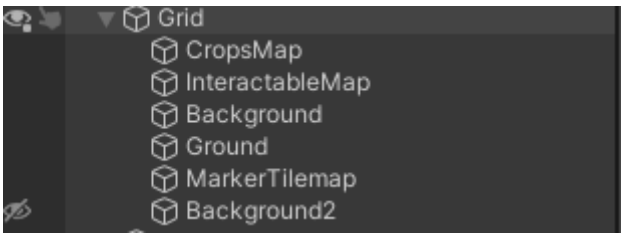


Figura 39. Capas del Mapeado.

Se crean varias capas que de momento no tendrán ningún uso como “CropsMap” para futuras iteraciones.

Además, se han añadido las barreras físicas de los bordes del mapa o de los precipicios para que el jugador no sea capaz de atravesar estos bordes con el personaje.

Pruebas

Se ha ejecutado el juego y se ha comprobado la correcta representación del mapeado y el buen funcionamiento de las barreras invisibles.

6.4- Iteración 4- Inventario Prototipo 1

Planteamiento Mecánica

El objetivo de este incremento es la creación de un primer sistema de inventario para poder llevar a cabo la realización de las siguientes iteraciones y proveer al desarrollador una mejor idea de las necesidades del sistema final de inventario.

Este inventario estará formado por diferentes espacios donde se irán almacenando los diferentes objetos en la lista de los diferentes espacios, actualizando el valor de estos.

Una vez realizado esto se añadirá la parte de poder añadir objetos al inventario dentro de la segunda iteración, para poder comprobar si los objetos se añaden correctamente

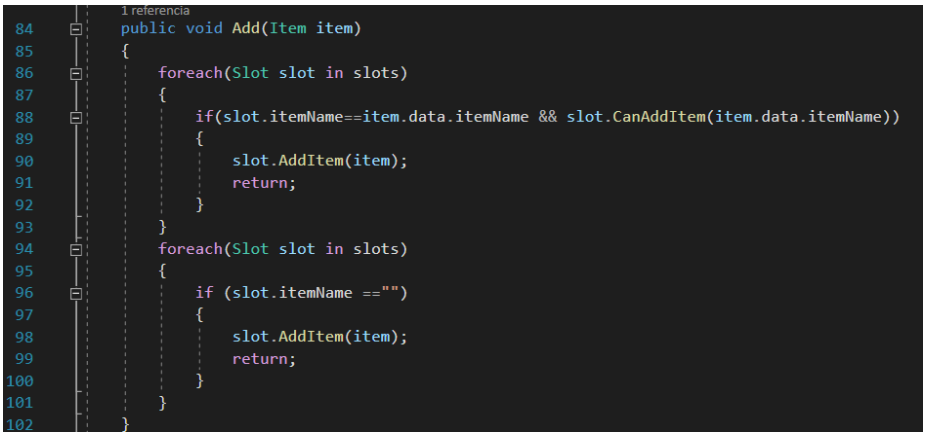
Desarrollo scripts

Se ha creado y desarrollado el script “Inventory.cs”. Este código lo que hace es crear la clase “slot” que se refiere a los diferentes espacios del inventario. Esta clase cuenta con una serie de elementos como lo serán el icono del objeto, la cantidad de este o el nombre:

```
5   public class Inventory
6   {
7       [System.Serializable]
        10 referencias
8       public class Slot
9       {
10          public string itemName;
11          public int count;
12          public int MaxAllowed;
13          public Sprite icon;
14          public Item item;
            1 referencia
15          public Slot()
16          {
17              itemName = "";
18              count = 0;
19              MaxAllowed = 2;
20              item = null;
21
22          }
```

Figura 40. Fragmento Código Inventory.cs.

También se han implementado las funciones para poder añadir los objetos a los inventarios. Dentro de estas funciones se hace referencia a otras funciones como “CanAddItem” cuya función es devolver verdadero o falso dependiendo de si es posible añadir cierto item en el inventario, para poder saber si el espacio de inventario ya está ocupado por otro objeto y pasar al siguiente.



```
1 referencia
public void Add(Item item)
{
    foreach(Slot slot in slots)
    {
        if(slot.itemName==item.data.itemName && slot.CanAddItem(item.data.itemName))
        {
            slot.AddItem(item);
            return;
        }
    }
    foreach(Slot slot in slots)
    {
        if (slot.itemName =="")
        {
            slot.AddItem(item);
            return;
        }
    }
}
```

Figura 41. Función Add() Inventory.cs.

Pruebas

Para probar el buen funcionamiento de la iteración se ha realizado la misma prueba que en la iteración 2, añadiendo una línea de código que imprima en todo momento el inventario actual del jugador. De esta manera podemos observar que se añaden los objetos de manera correcta y deseada.

6.5- Iteración 5- Interfaz Inventario

Planteamiento Dinámica

Para lo que conlleva este incremento se va a realizar una interfaz de inventario, lo que permitirá al jugador ver la representación visual de los objetos que tiene en su haber.

Esta interfaz estará asociada a la tecla TAB, con lo que el usuario podrá pulsar para abrir Inventario y volver a presionar para cerrarlo.

Para la representación visual se utilizarán los elementos de interfaz de Unity, tales como canvas, imágenes y paneles.

En cuanto al diseño, se tendrá como referencia el inventario del videojuego Stardew Valley, Incluyendo algunos elementos distintivos como pueden ser el espacio para los objetos, el botón de cerrar y se dejará suficiente espacio para poder implementar más funciones como la vista del propio personaje, las distintas pestañas del inventario, información general o el botón para los logros.

Figura 42. Ejemplo Inventario Stardew Valley.

Selección sprites

Al tratarse aún de un inventario prototipo, se ha desplazado la selección de sprites y recursos gráficos al incremento donde se desarrolla el inventario final, por lo que en esta iteración se usarán elementos básicos de Unity como cuadros de color blanco o gris, ya que no tenemos ninguna necesidad estética y lo más problable es que más adelante se tuviera que cambiar las componentes gráficas de la interfaz.

Desarrollo scripts

En cuanto al apartado de la lógica de funcionamiento, se crearán dos scripts: Inventory_UI.cs y Slot_UI.cs. Al hacer esta división nos aseguramos de tener un script que se encargue de la interfaz del inventario y otro de la interfaz de los espacios, de manera que en el futuro si se añaden más inventarios o se necesita que algún elemento cuente con espacios de inventario serán accesibles de manera totalmente independiente.

El primer script se encargará de actualizar el inventario cada vez que el proyecto lo requiera, como puede ser al recoger un objeto, y de crear cada uno de los espacios del inventario:

```
23      private void Start()
24      {
25          inventory = GameManager.instance.player.inventory.GetInventoryByName(inventoryName);
26
27          SetupSlots();
28          Refresh();
29
30
31      }
32
        4 referencias
33      public void Refresh()
34      {
35          if (slots.Count == inventory.slots.Count)
36          {
37              for(int i = 0; i < slots.Count; i++)
38              {
39                  if(inventory.slots[i].itemName != "")
40                  {
41                      slots[i].SetItem(inventory.slots[i]);
42                  }
43                  else
44                  {
45                      slots[i].SetEmpty();
46                  }
47              }
48          }
49
50      }
```

Figura 43. Fragmento Código Inventory_UI.cs.

El segundo script contará con las funciones SetItem y SetEmpty que se encargarán de establecer los objetos que se recojan en su espacio correspondiente,y de eliminar objetos del inventario y por tanto, establecer de nuevo el espacio como vacío, respectivamente.

Dentro de este script declaramos las componentes de cada espacio, que son el identificador del espacio, el inventario al que pertenecen, el icono y un texto con la cantidad que se tiene de ellos.

```
        Script de Unity (3 referencias de recurso) | 9 referencias
 7      public class Slot_UI : MonoBehaviour
 8      {
 9          public int slotID;
10          public Inventory inventory;
11          public Image itemIcon;
12          public TextMeshProUGUI quantityText;
13          [SerializeField] private GameObject highlight;
            1 referencia
14          public void SetItem(Inventory.Slot slot)
15          {
16              if (slot != null)
17              {
18                  itemIcon.sprite = slot.icon;
19                  itemIcon.color = new Color(1, 1, 1, 1);
20                  quantityText.text = slot.count.ToString();
21              }
22          }
            1 referencia
23          public void SetEmpty()
24          {
25              itemIcon.sprite = null;
26              itemIcon.color = new Color(1, 1, 1, 0);
27              quantityText.text = "";
28          }
```

Figura 44. Fragmento Código Slot_UI.cs.

Configurar Unity

Dentro de Unity, creamos un Canvas, dentro del cual pondremos cualquier tipo de interfaz que hagamos en nuestro proyecto. En el interior de este, crearemos un objeto vacío que representara todo nuestro inventario y, dentro, pondremos un panel donde se encontrarán los futuros elementos gráficos de nuestro inventario.

En el panel, crearemos una imagen gris que servirá de fondo y dentro de esta se crearán los slots (espacios de inventario). Los slots serán otra imagen que contendrán otra imagen y un texto que serán los de los objetos que tengamos en el inventario y, en el caso de no tener algún objeto en ese slot, serán invisibles. Estos espacios los hacemos "prefab", para que al hacer varios duplicados de estos slots puedan ser editados de una manera más sencilla, sin necesidad de editar uno a uno.

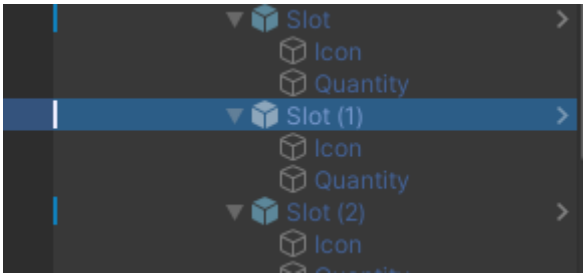


Figura 45. Jerarquía Slots.

Pruebas

Para las pruebas, se ha comprobado que el la interfaz aparece y desaparece al pulsar el tabulador y que representa correctamente los objetos si es que tenemos alguno en el inventario.

6.6- Iteración 6- Soltar Objetos

Planteamiento Mecánica

La idea básica para esta mecánica es la de dotar al usuario de la posibilidad de quitar algún objeto del inventario y que este aparezca en el mundo otra vez. Es la mecánica inversa a la de recoger objetos.

Aunque en temática puedan ser similares, a la hora de implementarla difiere de su antónima. Primero se deberá de quitar el objeto de su espacio correspondiente en el inventario y seguidamente deberemos hacer que Unity sea capaz de reconocer de qué objeto se trata y "crearlo" en el suelo, haciendo que aparezca lo suficientemente lejos del jugador para que este no lo recoja automáticamente.

Desarrollo scripts

Para el desarrollo de esta iteración necesitaremos modificar los scripts Inventory.cs y Inventory_UI.cs.

En el primero introduciremos la función Remove, qué estará sobrecargada, es decir, estará implementada dos veces, con diferentes parámetros de entrada para diferenciarlas y su desempeño es simple, llamar a RemoveItem, la otra función que creamos en este script, el número de veces que se quiera eliminar cierto objeto. RemoveItem se encargará de reducir la cantidad de unidades del objeto que estemos soltando y en caso de llegar a 0, asignar a ese slot o espacio de inventario los valores nulos.

```
4 referencias
103     public void Remove(int index)
104     {
105         slots[index].RemoveItem();
106
107
108     }
        1 referencia
109     public void Remove(int index,int numToRemove)
110     {
111         if(slots[index].count >= numToRemove)
112         {
113             for(int i=0; i < numToRemove; i++)
114             {
115                 Remove(index);
116             }
117         }
118     }
119
```

Figura 46. Funciones Remove() Inventory.cs.

Por lo que respecta al script de Inventory_UI.cs, se ha creado la función Remove para llamar a su homóloga dentro de Inventory.cs. Esta tendrá como parámetro de entrada el número de índice del slot del inventario, para saber cual eliminar en cada momento.

Para la parte de crear el objeto de nuevo en el suelo, creamos el script ItemManager.cs, el cual se encargará de gestionar todos los objetos del juego, creando un diccionario y guardando los datos de cada uno de los diferentes objetos para poder acceder a su información a la hora de crear algún objeto.

También se ha añadido una función para obtener el elemento “Item” dado un nombre, esto quiere decir que al llamar a esta función con el parámetro de un nombre en específico, obtendremos los demás datos del objeto, como pueden ser su icono u otros elementos que le asignemos a los objetos.

```
10      private void Awake()
11      {
12          foreach(Item item in items)
13          {
14              AddItem(item);
15              Debug.Log(item.data.itemName);
16          }
17      }
        1 referencia
18      private void AddItem(Item item)
19      {
20          if (!nameToItemDict.ContainsKey(item.data.itemName))
21          {
22              nameToItemDict.Add(item.data.itemName, item);
23          }
24      }
        1 referencia
25      public Item GetItemByName(string key)
26      {
27          if (nameToItemDict.ContainsKey(key))
28          {
29              return nameToItemDict[key];
30          }
31          return null;
32      }
33  }
```

Figura 47. Funciones Inventory_UI.cs.

Dentro de la función que hemos creado en Inventory_UI.cs, podemos obtener el nombre del objeto que estamos intentado eliminar accediendo al espacio y con esto, podemos obtener el objeto llamando a la función “GetItemByName” de ItemManager. Después, llamamos a una función llamada DropItem que crearemos dentro de otro script.

Ahora actualizaremos el script de Player.cs y añadiremos el código necesario para hacer que el objeto aparezca en el suelo. Lo hacemos en este script y no en los que hemos creado anteriormente ya que necesitamos acceder a la posición del jugador para saber dónde crear el objeto que soltemos. Se crea la función DropItem que se encarga de obtener la posición del jugador y de Instanciar (crear) el objeto añadiendo un margen para que aparezca fuera del rango de alcance del jugador.

```
2 referencias
public void DropItem(Item item)
{
    Vector2 spawnLocation = transform.position;

    Vector2 spawnOffset = Random.insideUnitCircle;
    if (spawnOffset.x < 0)
    {
        spawnOffset.x = (float)(spawnOffset.x - 0.3);
    }
    if (spawnOffset.x >= 0)
    {
        spawnOffset.x = (float)(spawnOffset.x + 0.3);
    }
    if (spawnOffset.y < 0)
    {
        spawnOffset.y = (float)(spawnOffset.y - 0.3);
    }
    if (spawnOffset.y >= 0)
    {
        spawnOffset.y = (float)(spawnOffset.y + 0.3);
    }

    //spawnOffset.x = (float)(spawnOffset.x + 0.1);
    //spawnOffset.y = (float)(spawnOffset.y + 0.1);
    //Debug.Log(spawnOffset);
    Item droppedItem= Instantiate(item, spawnLocation + spawnOffset,
        Quaternion.identity);
    droppedItem.rb2d.AddForce(spawnOffset * .2f, ForceMode2D.Impulse);
}
```

Figura 48. Función DropItem() Player.cs.

Configurar Unity

En Unity, añadimos los objetos a la lista del script de ItemManager.cs. Es muy importante añadir el objeto desde los prefabs, ya que si lo añadimos desde la escena, pueden haber errores. Esto se debe a que si el objeto de la escena desaparece, al ser recogido por ejemplo, se borraría también de la lista y al intentar soltar el objeto, obtendremos un error.

Pruebas

Se ha añadido un botón dentro del prefab de Slot, que llamará a las función de Remove. Se ha ejecutado el juego y se ha visto que al pulsar este botón, los objetos desaparecen del inventario y se crean en el suelo.

Se ha presentado el error de que al intentar eliminar un objeto del inventario, si se había añadido desde la escena nos devolvía un error ya que al recogerlo para añadirlo al inventario borraba el objeto de la lista.

6.7- Iteración 7- Mecánica de Arar tierra

Planteamiento Mecánica

Para este incremento se desarrollará la funcionalidad de arar la tierra. Para ello se ha pensado crear una nueva capa de mapeado y delimitar cuál será la zona en la que se podrá realizar esta acción. La zona escogida es la siguiente:

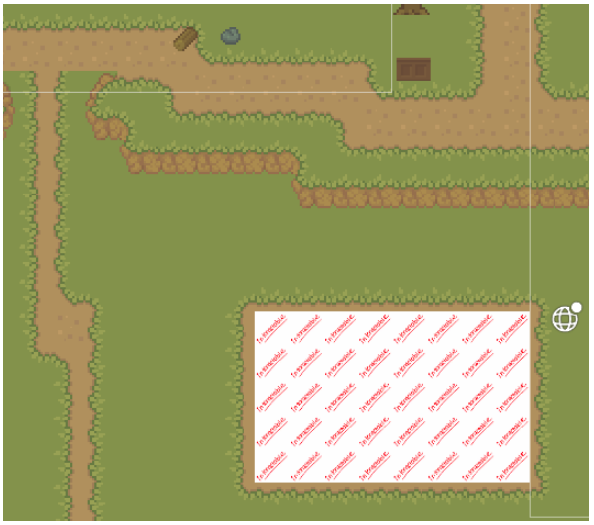


Figura 49. Capa Interactuable Tilemap.

Después, se le permitirá al usuario, dentro de esta zona, pulsar una tecla (barra espaciadora) y cambiar el recuadro de tierra normal a tierra arada.

Además, se creará la animación de la azada y se reproducirá cada vez que el usuario are un recuadro de tierra.

Selección sprites

Para determinar cuáles serán los recuadros con los que el usuario podrá interactuar y cúal será el recuadro para representar la tierra arada, usaremos los siguientes sprites:

Figura 50. Sprites necesarios para Arar Tierra.

Y para la animación de la azada, la obtenemos del paquete de recursos que hemos adquirido del proyecto:

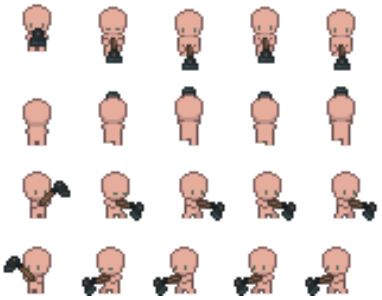


Figura 51. Sprites Animación Azada.

Desarrollo scripts

En cuanto al código, actualizaremos de nuevo el script de Player.cs y crearemos el de TileManager.cs.

En el primero, revisaremos en cada fracción de segundo si el usuario ha pulsado la tecla espacio y en caso de ser así, se obtendrá el recuadro al que está apuntado el jugador con el ratón y se llamará a la función que se encargará de arar la tierra , que explicaremos a continuación.

```
36      if (Input.GetKeyDown(KeyCode.Space) && selectable == true)
37      {
38
39          Vector3Int position2= GetTileBase(Input.mousePosition);
40
41          Vector3Int positionPlayer = GetTileBase(direction);
42          //Debug.Log("positionPlayer:" + positionPlayer);
43          //Debug.Log("positionMouse:" + position2);
44
45          auxHor = position2.x - (positionPlayer.x+7);
46          auxVer = position2.y - (positionPlayer.y+4);
47          Vector2Int toolDirection = new Vector2Int(auxHor, auxVer);
48          //Debug.Log("Auxs:"+auxHor +","+ auxVer);
49
50
51          GameManager.instance.toolManager.UseTool(position2, inventory.toolbar, direction,toolDirection);
```

Figura 52. Fragmento Código Player.cs.

Por lo que respecta al script de TileManager.cs, este se encargará de guardar leer las posiciones del mapeado en las que hayamos determinado que serán interactuables.

Tendrá una función para comprobar si un recuadro está dentro de las posiciones mencionadas anteriores y otra para cambiar el recuadro de tierra normal a tierra arada en caso de que sí que esté la posición seleccionada dentro de las posiciones interactuables.

```
101     public void PlowTile(Vector3Int position,Vector2Int toolDirection)
102     {
103         if (crops.ContainsKey((Vector2Int)position))
104         {
105             return;
106         }
107
108         interactableMap.SetTile(position, plowed);
```

Figura 53. Función PlowTile() TileManager.cs.

Configurar Unity

Dentro de Unity creamos la nueva capa de Tilemap, y configuramos las variables del script creado en el apartado anterior, como son el sprite de tierra arada o la capa del mapa a la que hace referencia en el código.

También se ha creado y diseñado la animación de la azada, que ha quedado de la siguiente manera en el animador de Unity:

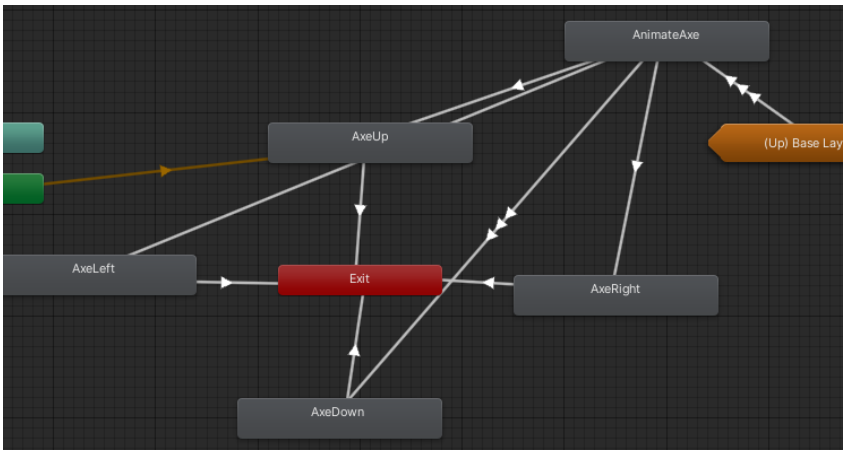


Figura 54. Animator de la Azada.

Pruebas

Para las pruebas se ejecuta el juego y se comprueba el buen funcionamiento dentro de la zona designada para arar. Además, pese a no estar implementada en el código, se comprueba que la animación de la azada se vea correctamente con unos botones auxiliares que se han creado para la prueba y se han borrado posteriormente.

6.8- Iteración 8- Sistema de Objetos

Planteamiento Mecánica

Hasta este momento y, a pesar de que en el código mostrado no se ha reflejado, no existía un sistema de objetos. A la hora de referirse a un objeto, realmente teníamos que acceder a su componente de Collectable.cs, creado en la iteración 2. Debido a esto, acceder a ciertas propiedades o al pensar en añadir otro tipo de objetos con funciones diferentes se complicaba sobremanera.

Por ello, en esta iteración se va a realizar un sistema de objetos completo, donde cada objeto del juego tendrá asociado un "ItemData", un ScriptableObject de Unity. Este objeto es un script que se utiliza para almacenar datos de entidades de una manera mucho más accesible y potente. Dentro de este script definiremos cuales son las características que podrán tener cualquiera de nuestros objetos en el

juego y Unity nos permitirá acceder a estos datos como si fuera una librería, de manera muy sencilla, accesible desde cualquier script, objeto o escena sin necesidad de conectarlo de ninguna manera.

Este sistema nos posibilitará, por ejemplo, asignar o no funcionalidades a los objetos. Esto es de gran ayuda porque a la hora de implementar las herramientas agrícolas, cada una tendrá su propia funcionalidad diferente, como puede ser talar un árbol con el hacha o arar la tierra con la azada.

Además, nos permitirá crear más objetos de una manera sencilla y que consumirá menos recursos.

Desarrollo scripts

Se han implementado los scripts de ItemData.cs y Item.cs, además de hacer modificaciones prácticamente a la totalidad de los otros scripts para referenciar de la nueva forma los objetos.

ItemData.cs será el lugar donde se crearán las instancias de nuestros objetos, por lo que crearemos un menú para poder configurar estos parámetros desde Unity y propondremos los parámetros básicos de los objetos que tenemos hasta ahora: el nombre del objeto, su icono y el tipo. Más adelante, se añadirá nueva información necesaria para los objetos, como pueden ser las funcionalidades de las herramientas, el tipo de cultivo, el precio o el prefab asociado.

```
6       [CreateAssetMenu(fileName = "Item Data", menuName = "Item Data", order = 50)]
        Script de Unity | 4 referencias
7   public class ItemData : ScriptableObject
8   {
9       public string itemName = "Item Name";
10      public Sprite icon;
11      public string itemType = "Item Type";
12      public ToolAction onAction;
13      public ToolAction onTilemapAction;
14      public ToolAction onItemUsed;
15      public Crop crop;
16      public int price;
17      public GameObject prefab;
18  }
```

Figura 55. Fragmento Código ItemData.cs.

Para Item.cs, impondremos la necesidad de tener un cuerpo de físicas asociado al objeto al que se asocie dentro del unity para poder acceder a él en el momento que se detecten las colisiones y, además, le asignaremos un ItemData de los que crearemos para cada objeto del videojuego.

```
4       [RequireComponent(typeof(Rigidbody2D))]
        Script de Unity (7 referencias de recurso) | 33 referencias
5   public class Item : MonoBehaviour
6   {
7       public ItemData data;
8       [HideInInspector]public Rigidbody2D rb2d;
        Mensaje de Unity | 0 referencias
9       private void Awake()
10      {
11          rb2d = GetComponent<Rigidbody2D>();
12      }
13  }
14
```

Figura 56. Fragmento Código Item.cs.

Configurar Unity

En Unity, procederemos a crear los diferentes ItemData para cada objeto y los asociaremos a los prefabs con los objetos ya creados:

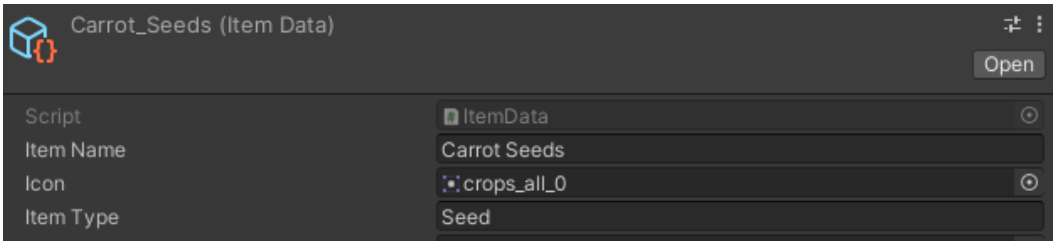


Figura 57. Item Data dentro de Unity.

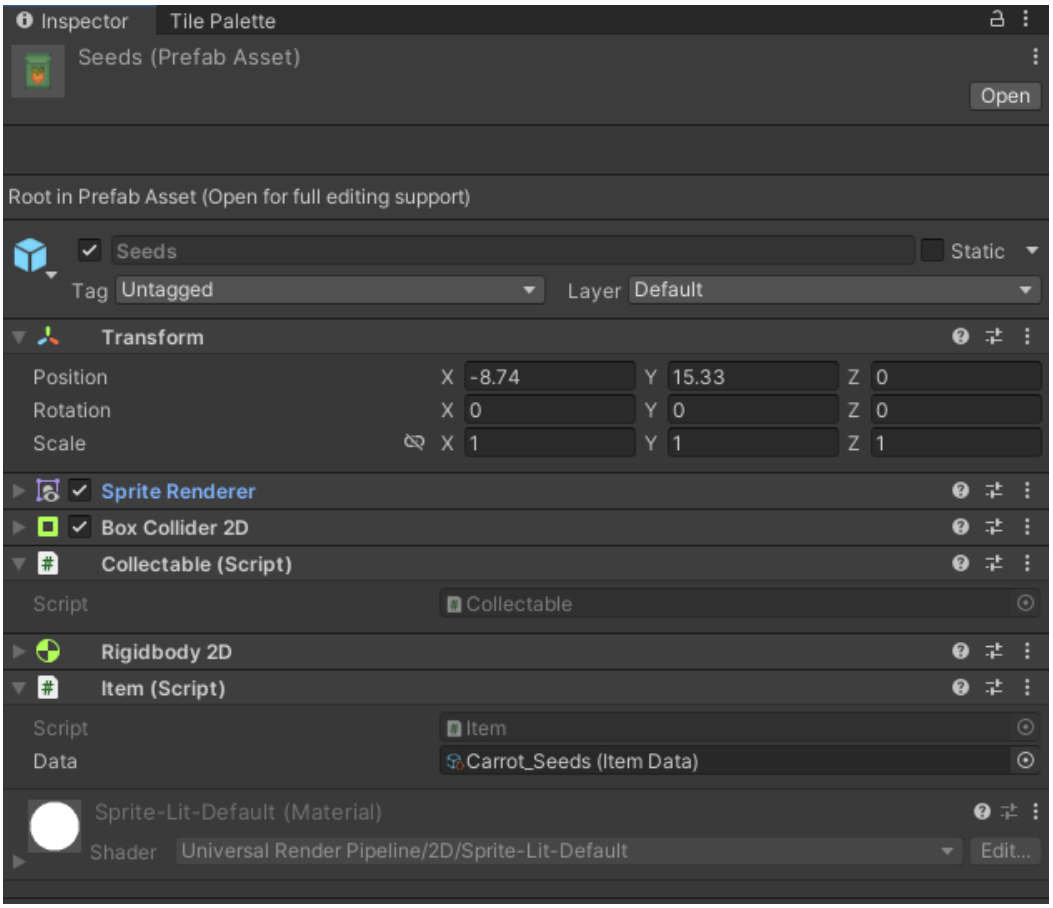


Figura 58. Inspector de Item dentro de Unity.

Pruebas

Las pruebas realizadas han sido probar el funcionamiento de todas las iteraciones que se han implementado hasta el momento, para asegurar que todo el funcionamiento sigue correcto después de los cambios a, prácticamente, la totalidad de los scripts.

6.9- Iteración 9- Barra de Herramientas

Planteamiento Mecánica

En este incremento implementaremos el diseño, creación y desarrollo de la barra de herramientas, muy habitual en todo tipo de videojuegos y, imprescindible, en un juego de estas características.

Una barra de herramientas es como una especie de inventario más reducido, que usualmente aparece en la parte inferior central de la pantalla. El jugador es capaz de acceder a los diferentes espacios de inventario de esta barra pulsando los botones numéricos del teclado, permitiendo un juego más fluido sin la necesidad de parar para usar cada uno de los objetos desde el inventario principal.

La barra de herramientas que se ha pensado consta de 9 espacios, correspondientes a los primeros nueve números. Además, se implementará un marcador, para poder ver cual es el espacio que se tiene seleccionado en cada momento.

Selección sprites

Para la selección de sprites se ha elegido el siguiente del paquete de recursos gráficos:

Figura 59. Sprite Barra de Herramientas.

Además, para el remarcador, se ha creado un sprite sencillo de un borde de recuadro blanco, que dentro del unity se le reducirá la opacidad para que no resalte sobremanera.

Desarrollo scripts

En cuanto al desarrollo de código, se ha creado el script Toolbar_UI.cs. Dentro de este se crea una lista con los diferentes slots de la barra de herramientas y se comprobará en cada fracción de segundo si el jugador presiona uno de los nueve números para poder poner el marcador y cambiar el slot seleccionado, que será una variable estática desde la que podrán acceder otros scripts.

```
5   public class Toolbar_UI : MonoBehaviour
6   {
7       [SerializeField] private List<Slot_UI> toolbarSlots = new List<Slot_UI>();
8
9       public static Slot_UI selectedSlot;
10      public static int numSlot;
11
12
        Mensaje de Unity | 0 referencias
13      private void Start()
14      {
15          SelectSlot(0);
16      }
        Mensaje de Unity | 0 referencias
17      private void Update()
18      {
19          CheckAlphaNumericKeys();
20      }
21
        10 referencias
22      public void SelectSlot(int index)
23      {
24          if (toolbarSlots.Count == 9)
25          {
26              if (selectedSlot != null)
27              {
28                  selectedSlot.SetHighlight(false);
29              }
30              selectedSlot = toolbarSlots[index];
31              numSlot = index;
32              //Debug.Log("SelectedSlot" + selectedSlot.name);
33              selectedSlot.SetHighlight(true);
```

Figura 60. Fragmento Código Toolbar_UI.cs.

Configurar Unity

Dentro de Unity, lo primero es crear un nuevo panel dentro del canvas de la iteración 4, donde asignaremos como imagen de fondo el sprite seleccionado para la barra de herramientas.

Dentro, asignaremos los nueve slots, que serán casi idénticos a los del inventario, pero que tendrán unos pocos cambios, por lo que crearemos un prefab específico para los espacios de la barra de herramienta que serán una copia de los del inventario. A este nuevo prefab le añadiremos un texto en la parte superior izquierda para poner los números de la barra de herramientas (1-9) y también le asignaremos la imagen con el marcador, que estará desactivado ya que el código se encargará de activarlo dependiendo del espacio seleccionado.

Así es como se ve la barra de herramientas dentro del juego con el slot 4 seleccionado:

Figura 61. Captura de Pantalla de la Barra de Herramientas dentro del Juego.

Pruebas

Para las pruebas, se ha ejecutado el juego y se ha comprobado que se pueda cambiar el marcador presionando todos los números.

6.10- Iteración 10- Inventario Final

Planteamiento Mecánica

En este incremento ahondaremos el apartado del inventario visto en la iteración 4.

Ahora que el proyecto está lo suficientemente avanzado y debido a que las siguientes iteraciones tratarán temas asociados al inventario, es un buen momento para diseñar y desarrollar el inventario final, donde pondremos los elementos gráficos finales y prepararemos adecuadamente el elemento para los próximos pasos.

Selección sprites

Como es habitual, vamos a utilizar algunos de los sprites que hemos adquirido, se han seleccionado los siguientes para conformar un inventario:

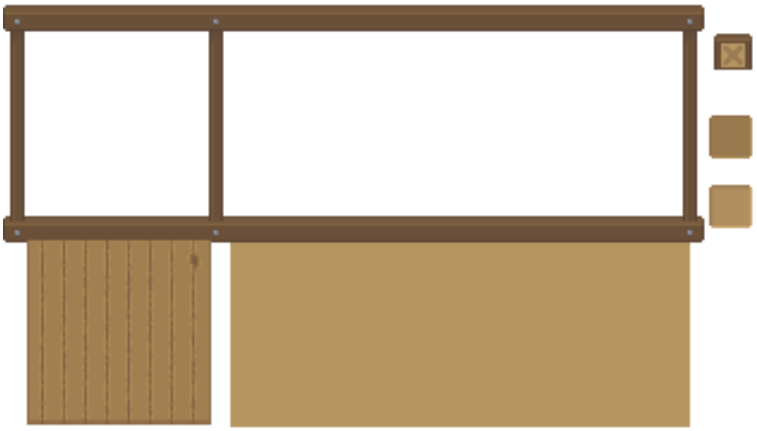


Figura 62. Sprites Inventario.

Como se puede apreciar, tenemos el inventario dividido en secciones, como el marco, los fondos o el botón de cerrar. El inventario en sí, está dividido en dos secciones, la de la derecha, que es más amplia, se corresponde con la sección de los slots, y la de la izquierda queda libre para poder implementar la ventana de personaje en el desarrollo del juego completo.

Configurar Unity

Dentro de Unity, añadiremos todos los componentes gráficos y los asociaremos al marco del inventario. También modificaremos el prefab de los slots y pondremos las imágenes definitivas de los cajones del inventario. Añadiremos 27 slots al inventario y los alineamos para que queden ordenados en tres filas y ocupen todo el espacio de la derecha.

Implementaremos el botón de cerrar el inventario y le añadiremos la función de Toggle Inventory, que es la que nos permite abrir o cerrar inventario.

Pruebas

Se vuelve a comprobar que el inventario funcione correctamente, que se abra y se cierre al pulsar tabulador, que los slots aparezcan en su lugar, que los objetos se añadan al inventario y salgan representados.

6.11- Iteración 11- Arrastrar y soltar (Drag and Drop)

Planteamiento Mecánica

En este incremento se desarrollará la mecánica de Drag and Drop, que permitirá al jugador arrastrar y soltar objetos en los diferentes espacios del inventario o fuera de él.

El usuario será capaz de cambiar los objetos entre un slot a otro sin ningún tipo de problema, siempre que estén en el mismo inventario, por ejemplo: un objeto está en el primer slot del inventario, el jugador lo puede cambiar a cualquier otro slot del inventario pero no podrá moverlos a la barra de herramientas. Este tema se abordará en la siguiente iteración de conexión de inventarios.

Para abordar esta mecánica, usaremos la componente Event Trigger de Unity, que nos permite detectar cuando comienza un arrastre, cuando se sigue arrastrando, cuando se acaba de arrastrar y cuando se suelta.

Desarrollo scripts

Se modificarán los scripts de Inventory_UI.cs y Inventory.cs.

En el primero crearemos las funciones que gestionarán los cuatro eventos que pondremos en el Event Trigger mencionado anteriormente. Se creará una variable “DraggedSlot” que incluirá información de cuál es el slot con el que se está interactuando en cada momento.

Al comenzar a arrastrar, se creará una instancia con el icono del objeto que estemos arrastrando para poder observarlo mientras se arrastra el objeto.

```
81      public void SlotBeginDrag(Slot_UI slot)
82      {
83          UI_Manager.draggedSlot = slot;
84          UI_Manager.draggedIcon = Instantiate(UI_Manager.draggedSlot.itemIcon);
85          UI_Manager.draggedIcon.transform.SetParent(canvas.transform);
86          UI_Manager.draggedIcon.raycastTarget = false;
87          UI_Manager.draggedIcon.rectTransform.sizeDelta = new Vector2(50f, 50f);
88          MoveToMousePosition(UI_Manager.draggedIcon.gameObject);
89          print("Start Drag:" + inventory.slots[UI_Manager.draggedSlot.slotID].itemName);
90
91      }
```

Figura 63. Función SlotBeginDrag() Inventory_UI.cs.

En cuanto al script de Inventory.cs, se creará la función MoveSlot, que tendrá la función de gestionar el movimiento de los objetos de un slot a otro, recibirá los parámetros: índice del slot inicial, índice slot finaly el número de objetos que se mueven.

```
120     public void MoveSlot(int fromIndex, int toIndex, Inventory toInventory, int numToMove=1)
121     {
122         Slot fromSlot = slots[fromIndex];
123         Slot toSlot = toInventory.slots[toIndex];
124
125         for (int i = 0; i < numToMove; i++)
126         {
127             if (toSlot.IsEmpty || toSlot.CanAddItem(fromSlot.itemName))
128             {
129                 toSlot.AddItem(fromSlot.itemName, fromSlot.icon, fromSlot.MaxAllowed,fromSlot.item);
130                 fromSlot.RemoveItem();
131             }
132         }
133     }
```

Figura 64. Función MoveSlot() Inventory.cs.

Creamos el script UI_Manager.cs, añadiremos en la actualización en cada fracción de segundo, una comprobación para saber si el usuario está pulsando shift cuando va a seleccionar un slot y en caso afirmativo, establecerá una variable global para poder saber si el usuario quiere mover un solo objeto de un espacio de inventario o quiere moverlos todos.

```
26          if (Input.GetKey(KeyCode.LeftShift))
27          {
28              dragSingle = true;
29          }
30          else
31          {
32              dragSingle = false;
33          }
```

Figura 65. Fragmento Código UI_Manager.cs.

Configurar Unity

En Unity, configuraremos el Event Trigger con los eventos mencionados y asociaremos las funciones que hemos creado para poder mover y soltar objetos:

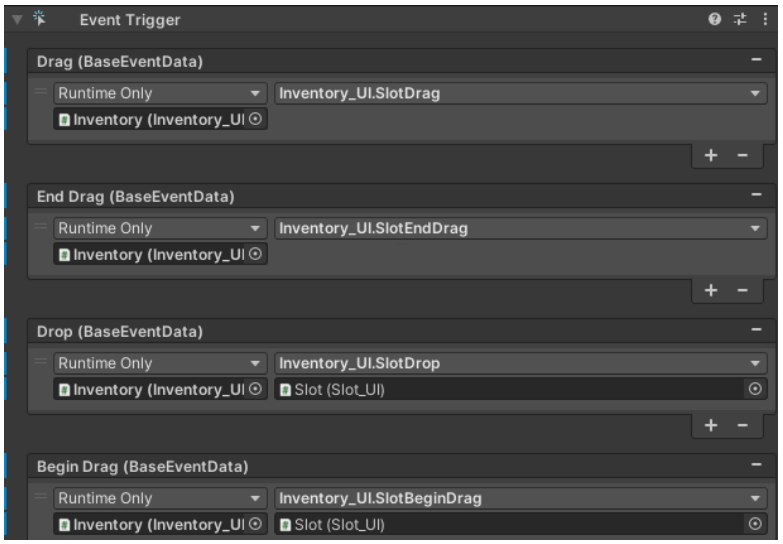


Figura 66. Event Trigger Slots.

También añadiremos un panel invisible detrás del inventario al que le asociaremos la función de Remove, lo que quiere decir que al soltar un objeto fuera del inventario, es decir, dentro del panel invisible, soltaremos el objeto y aparecerá en el suelo:

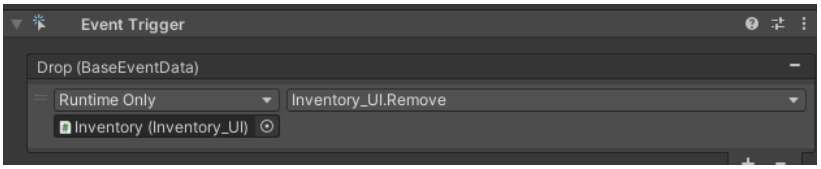


Figura 67. Event Trigger Panel de Soltar Objetos.

Pruebas

Se ha comprobado el correcto funcionamiento de los cuatro eventos programados implementando mensajes con consola para comprobar que detectaba de manera correcta todos los casos. También se ha comprobado que funciona al soltar objetos al suelo.

6.12- Iteración 12- Conexión Inventarios

Planteamiento Mecánica

El objetivo de esta iteración es retocar el proyecto para ser capaces de poder mover objetos entre el inventario y la barra de herramientas. Desde esta última solo queremos poder mover objetos, no deseamos poder soltar objetos en el suelo.

Para ello, deberemos cambiar un poco las funciones creadas en el apartado anterior y configurar correctamente una manera para poder detectar en qué inventario interactuamos en cada momento.

Desarrollo scripts

En esta parte nos centramos principalmente en el script creado en el apartado anterior, UI_Manager.cs. Dentro de este vamos a crear unas funciones que nos permitirán actualizar el inventario deseado o actualizarlos todos a la vez:

```
51      public void RefreshInventoryUI(string inventoryName)
52      {
53          if (inventoryUIByName.ContainsKey(inventoryName))
54          {
55              inventoryUIByName[inventoryName].Refresh();
56          }
57      }
        1 referencia
58      public void RefreshAll()
59      {
60          foreach(KeyValuePair<string, Inventory_UI>keyValuePair in inventoryUIByName)
61          {
62              keyValuePair.Value.Refresh();
63          }
64      }
```

Figura 68. Funciones Refresh() UI_Manager.cs.

Además, se ha implementado una función para obtener la componente Inventory_UI dado el nombre de inventario, lo que nos permitirá saber claramente el inventario con el que estamos interactuando:

```
65      public Inventory_UI GetInventoryUI(string inventoryName)
66      {
67          if (inventoryUIByName.ContainsKey(inventoryName))
68          {
69              return inventoryUIByName[inventoryName];
70          }
71          Debug.LogWarning("There is no inventory ui for " + inventoryName);
72          return null;
73      }
```

Figura 69. Función GetInventoryUI UI_Manager.cs.

Dentro del script Inventory_UI.cs se ha modificado la función SlotDrop para que pueda obtener correctamente los valores del inventario inicial y del inventario final:

```
106     public void SlotDrop(Slot_UI slot)
107     {
108         if (UI_Manager.dragSingle)
109         {
110             UI_Manager.draggedSlot.inventory.MoveSlot(UI_Manager.draggedSlot.slotID, slot.slotID, slot.inventory);
111         }
112         else
113         {
114             UI_Manager.draggedSlot.inventory.MoveSlot(UI_Manager.draggedSlot.slotID, slot.slotID, slot.inventory,
115                 UI_Manager.draggedSlot.inventory.slots[UI_Manager.draggedSlot.slotID].count);
116         }
117         GameManager.instance.uiManager.RefreshAll();
118     }
```

Figura 70. Función SlotDrop Inventory_UI.cs.

Configurar Unity

En Unity se ha configurado el Event Trigger para los slots de la barra de herramientas.

Pruebas

Se ha comprobado el funcionamiento del traspaso de objetos entre inventarios.

6.13- Iteración 13- Día y Noche

Planteamiento Mecánica

En este incremento se va a desarrollar la dinámica de Día y Noche, lo que conlleva la creación de un sistema de tiempo propio del juego y nos exige la implementación de algún módulo que gestione el apartado de la iluminación.

Se ha escogido el paquete de Universal RP para el control de la iluminación necesaria.

También se implementará en este punto la introducción en la interfaz de los diseños para mostrar la hora del día y el día del mes, por lo que se desarrollará el código para gestionar los segundos, minutos, horas o días.

Selección sprites

Para la interfaz que contiene la fecha y hora se han escogido estos sprites del paquete gráfico:

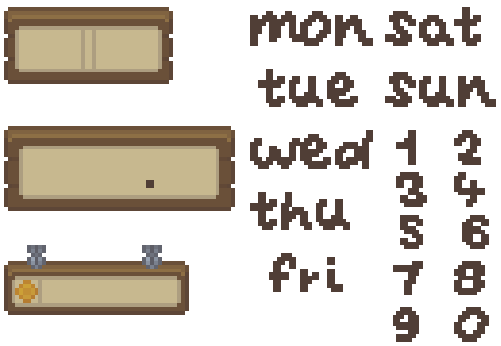


Figura 71. Sprites Interfaz Fecha y Hora.

Se utilizarán los recuadros de la izquierda para la interfaz, los letreros con los días de la semana en inglés se introducirán como imágenes que se seleccionarán una vez pase el día y los números se convertirán a formato de tipografía de texto con la página online gratuita Calligraphr para configurarlo posteriormente en Unity.

Desarrollo scripts

Se ha desarrollado el script de DayTimeController.cs, donde gestionaremos todo lo relacionado con el paso de segundos, minutos, horas y días.

```
private void Update()
{
    time += Time.deltaTime * timeScale;
    TimeValueCalculation();
    DayLight();
    if (time > secondInDay)
    {
        NextDay();
    }

    TimeAgents();
}
1 referencia
private void TimeValueCalculation()
{
    int hh = (int)Hours;
    int mm = (int)Minutes;
    textHour.text = hh.ToString("00");
    textMinutes.text =  mm.ToString("00");
}
```

Figura 72. Fragmento Código DayTimeController.cs.

También tendremos una función encargada de gestionar la el grado de luz del día según la hora que sea y otra para gestionar el cambio de día, donde será necesario cambiar la imagen con el día de la semana y escribir con texto el número del día del mes:

```
90      private void DayLight()
91      {
92          float v = nightTimeCurve.Evaluate(Hours);
93          Color c = Color.Lerp(dayLightColor, nightLightColor, v);
94          globalLight.color = c;
95      }
        1 referencia
96      private void NextDay()
97      {
98          time = 0;
99          days += 1;
100         dayOfWeek = (days-1) % 7;
101         textNumDay.text = days.ToString("00");
102         dayOfWeekIm.sprite=daySprites[dayOfWeek];
103     }
104 }
```

Figura 73. Funciones DayLight() y NextDay() DayTimeController.cs.

Configurar Unity

Dentro de Unity, deberemos importar los sprites para la interfaz y los días de las semanas e importar la nueva tipografía con los 10 carácteres numéricos.

Después, añadimos al canvas la nueva parte de la interfaz, a la que hemos denominado "TimeBox". La configuraremos para que en el primer recuadro salgan las horas y minutos con la tipografía creada. En el segundo cuadro contendrá una imagen con el día de la semana y un texto con el número del día respecto al mes. El tercer cuadro lo dejaremos en blanco.

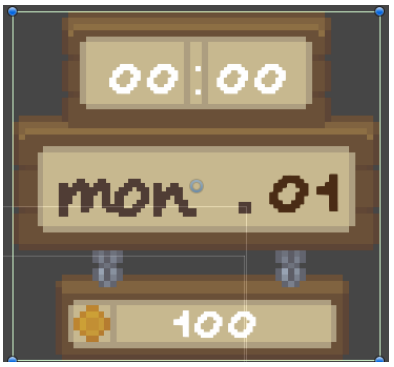


Figura 74. Interfaz Fecha y Hora.

Después, configuraremos los parámetros que necesitamos para el script creado en el apartado anterior, que son principalmente los diferentes textos de la interfaz, la lista de las imágenes de los días de la semana y la escala de tiempo:

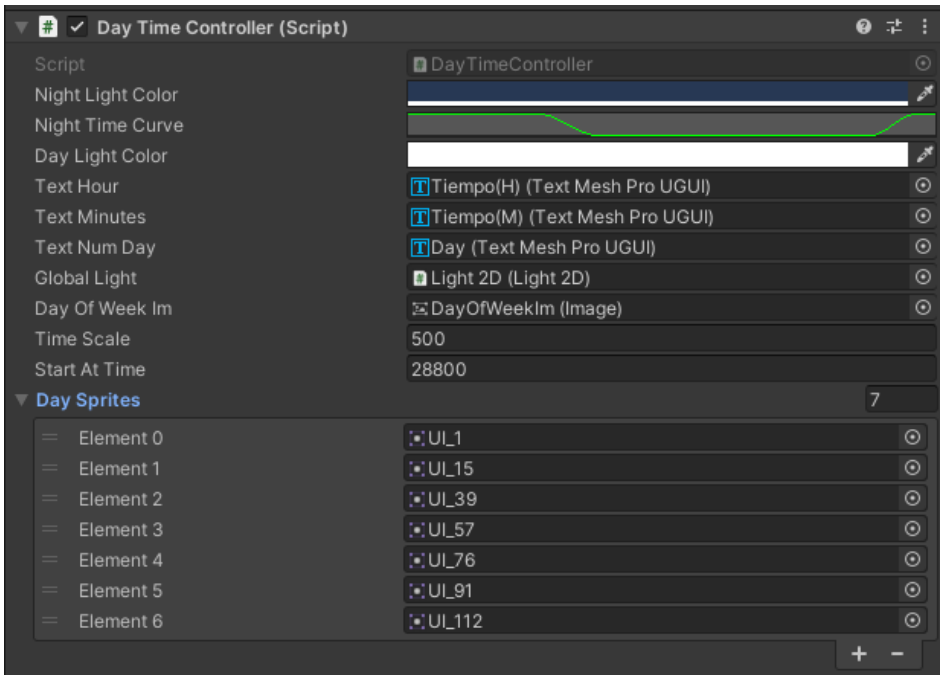


Figura 75. Configuración Script DayTimeController.cs desde Unity.

Pruebas

Se ha comprobado el correcto paso del tiempo dentro del juego y su correcta representación en la interfaz.

6.14- Iteración 14- Sistema de tiempo interno

Planteamiento Mecánica

En el desarrollo de este incremento, planteamos la manera de gestionar el tiempo interno del juego, es decir, que los elementos del juego puedan ser capaces de hacer algún tipo de cosa después de un tiempo específico.

Esto será necesario para la iteración de los cultivos y, además, será esencial para el videojuego completo, donde se implementará multitud de eventos o interacciones que necesiten tener en cuenta el tiempo.

Desarrollo scripts

Desarrollaremos el script TimeAgent.cs, que será el script que permitirá a los componentes acceder al tiempo interno del juego.

También modificaremos el script creado en la iteración anterior, DayTimeController.cs, para permitir a los elementos del juego subscribirse al tiempo virtual, ya que este se gestiona desde este código.

```
31      private void Awake()
32      {
33          agents = new List<TimeAgent>();
34      }
        Mensaje de Unity | 0 referencias
35      private void Start()
36      {
37          time = startAtTime;
38      }
        1 referencia
39      public void Subscribe(TimeAgent timeAgent)
40      {
41          agents.Add(timeAgent);
42      }
        1 referencia
43      public void Unsubscribe(TimeAgent timeAgent)
44      {
45          agents.Remove(timeAgent);
46      }
```

Figura 76. Código Tiempo Interno DayTimeController.cs.

El script TimeAgent.cs, básicamente, sobrescribirá el elemento del juego al que esté unido, al sistema de tiempo interno:

```
9       public Action onTimeTick;
        Mensaje de Unity | 0 referencias
10      private void Start()
11      {
12          Init();
13      }
        2 referencias
14      public void Init()
15      {
16          GameManager.instance.dayTimeController.Subscribe(this);
17      }
        1 referencia
18      public void Invoke()
19      {
20          onTimeTick?.Invoke();
21      }
22
        Mensaje de Unity | 0 referencias
23      private void OnDestroy()
24      {
25          GameManager.instance.dayTimeController.Unsubscribe(this);
26      }
```

Figura 77. Fragmento Código TimeAgent.cs.

Pruebas

Para probar que el sistema funcione bien, se ha creado una prueba sencilla, poner un cuadrado en el suelo que sea capaz de crear semillas de zanahoria cada vez que pase un tiempo específico.

6.15- Iteración 15- Sembrar semillas

Planteamiento Mecánica

El desarrollo de está mecánica puede considerarse una pequeña expansión de la iteración 7, arar la tierra.

El funcionamiento será muy similar pero incluiremos una condición al pulsar la tecla espaciadora para comprobar si la tierra con la que se quiere interactuar está ya arada y en caso afirmativo, llamar a la función para sembrar en vez de la de arar.

Selección sprites

Se usará un sprite de semilla básico para comprobar el funcionamiento de la mecánica:

Figura 78. Sprite Semillas.

Desarrollo scripts

Modificaremos el script de Player.cs para añadir el condicional y el script TileManager.cs para añadir la funcionalidad de plantar semillas, además de añadir otro condicional en este script para futuras iteraciones o para el proyecto completo. En el momento de plantar la semilla, en vez de cambiar el sprite dentro de la capa del mapeado InteractableMap, lo cambiaremos en la capa cropsMap, que será el que utilicemos exclusivamente para los cultivos, ya que nos interesa tenerlos separados por si en un futuro se añade alguna funcionalidad que pueda interactuar en alguna capa específica:

```
118     public void SeedTile(Vector3Int position, Crop toSeed)
119     {
120
121         cropsMap.SetTile(position, seeded);
122         //Debug.Log("PonemosSeedTILEMANAGER");
123         crops[(Vector2Int)position].crop = toSeed;
124
125     }
        2 referencias
126     public bool Check(Vector3Int position)
127     {
```

Figura 79. Función SeedTile() TileManager.cs.

Configurar Unity

Dentro del unity configuraremos el sprite de la semilla y lo añadimos como variable al script TileManager.cs

Pruebas

Se ha comprobado que al pulsar espacio apuntando a una tierra ya arada, se planta la semilla.

6.16- Iteración 16- Funcionalidad Herramientas

Planteamiento Mecánica

Antes de pasar a la implementación del crecimiento de cultivos, vamos a desarrollar el sistema completo de la funcionalidad de herramientas u otros elementos.

Esta iteración es una de las más importantes de este proyecto, ya que se ha pensado de una manera que es perfectamente escalable para un videojuego completo y es muy polivalente.

Este sistema dotará a los objetos de funcionalidades específicas, es decir, al hacha le dará la mecánica de poder talar un árbol, al azada de arar la tierra y las hará exclusivas de estas herramientas.

Dividiremos el sistema en dos partes, los elementos que interactúan con el mapeado, como serán el azada o las semillas, y los elementos que interactúan con otros elementos, como el hacha o el pico.

Esta diferenciación se debe a que el primer tipo necesita acceder a un recuadro del mapa en específico de alguna capa tilemap, mientras que el segundo accede a elementos que no estén dentro de los recuadros del tilemap, siendo elementos externos, como un árbol, la casa del jugador o el jugador en sí.

Selección sprites

En este incremento se han implementado elementos como rocas, árboles, trozos de piedra, trozos de madera, hacha, pico y azada, estos son sus respectivos sprites, originarios del paquete de recursos adquirido:

Figura 80. Sprites Varios.

Desarrollo scripts y Unity

El primer paso será desarrollar el script que contendrá la clase a la que pertenecerán todas las funcionalidades. Este script será un Scriptable Object, lo cual nos permitirá crear varias instancias de éste con datos propios. Definiremos las funciones OnApply y OnApplyToTilemap, para los dos tipos de interacción que hemos definido en el planteamiento:

```
6   public class ToolAction : ScriptableObject
7   {
        2 referencias
8       public virtual bool OnApply(Vector2 worldPoint,Vector2Int toolDirection, Animator animator)
9       {
10          Debug.LogWarning("OnApply is not implemented");
11          return true;
12      }
13
        5 referencias
14      public virtual bool OnApplyToTilemap(Vector3Int tilemapPosition,Tilemap tilemap,Item item,Vector2Int toolDirection)
15      {
16          Debug.LogWarning("OnApplyToTilemap is not implemented");
17          return true;
18      }
```

Figura 81. Fragmento Código ToolAction.cs.

Estas funciones son virtuales, lo que significa que se pueden volver a definir en los scripts que sean miembros de ToolAction con sus funciones específicas.

Para el primer tipo de funcionalidad, creamos un script, toolhit.cs, que detectará si hay colisión entre el la herramienta y otro elemento con físicas y, crearemos el script ResourceNode.cs, que será miembro de toolhit volver a implementar la función de hit para que al golpear, el elemento golpeado se destruya y cree otros elementos en el suelo:

```
11      public override void Hit()
12      {
13          Vector2 position = this.transform.position;
14          GameManager.instance.player.DropItemPosition(pickUpDrop, dropCount,position);
15          Destroy(gameObject);
16      }
        2 referencias
17      public override bool CanBeHit(List<ResourceNodeType> canBeHit)
18      {
19          return canBeHit.Contains(nodeType);
20      }
21  }
```

Figura 82. Fragmento Código ResourceNode.cs.

Esto se utilizará, por ejemplo, cuando un pico golpea una roca, ésta se destruirá y saldrán fragmentos de piedras.

Después, crearemos el script GatherResourceNode.cs, miembro de ToolAction. Este sobreescribirá la función OnApply para detectar si existe colisión entre herramienta y jugador llamando al script anterior, reproducirá la animación de la herramienta que se esté usando y golpeará al elemento, haciendo que este se rompa y descomponga en piezas más pequeñas:

```
21          Collider2D[] colliders = Physics2D.OverlapCircleAll(worldPoint, sizeInteractableArea);
22          foreach (Collider2D c in colliders)
23          {
24              ToolHit hit = c.GetComponent<ToolHit>();
25              if (hit != null)
26              {
27                  if (hit.CanBeHit(canHitNodesOfType) == true)
28                  {
29
30                      animator.SetInteger("toolDirHorizontal", toolDirection.x);
31                      animator.SetInteger("toolDirVertical", toolDirection.y);
32                      animator.SetTrigger(animationName);
33                      hit.Hit();
```

Figura 83. Fragmento Código GatherResourceNode.cs.

Ahora, en Unity, solo resta crear “ToolActions” desde el inspector y configurar las funcionalidades CutTree y SmashRock:

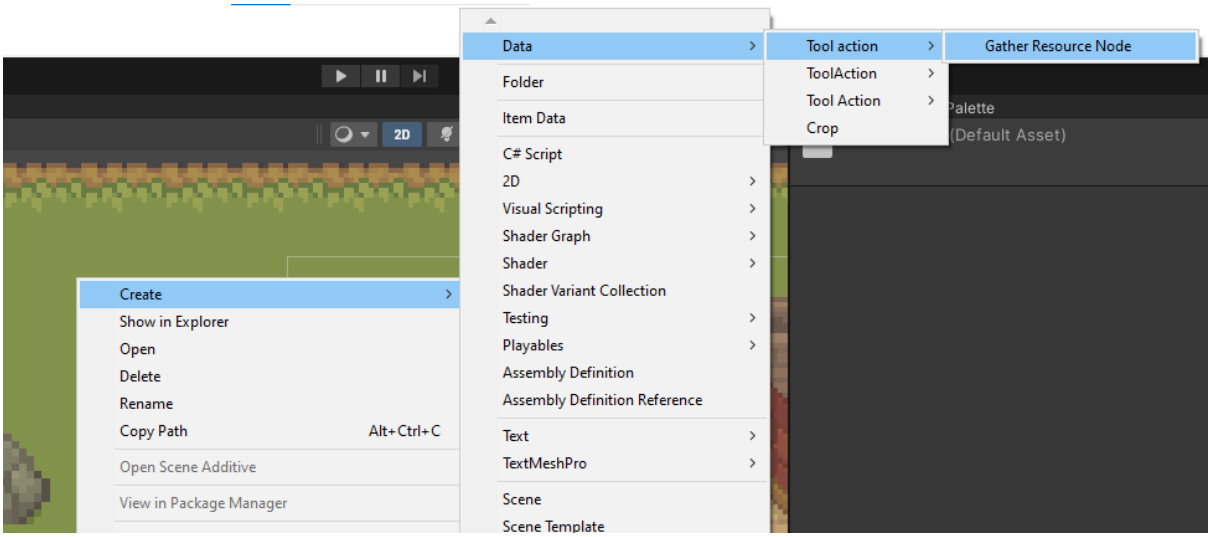


Figura 84. Captura Creación ToolAction.

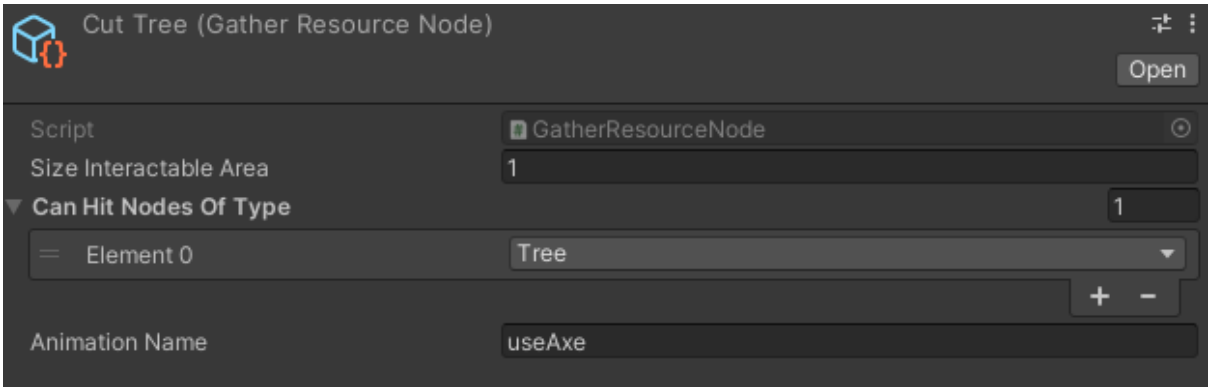


Figura 85. Captura Configuración GatherResourceNode en Unity.

Para el segundo tipo, los que interactúan con los recuadros del mapa, se crean los scripts PlowTile.cs y SeedTile.cs, miembros de ToolAction, que lo único que harán será sobrescribir la función de OnApplyToTilemap y llamará a las funciones de PlowTile y SeedTile dentro de TileManager.cs:

```
7   public class PlowTile : ToolAction
8   {
9
        3 referencias
10      public override bool OnApplyToTilemap(Vector3Int tilemapPosition, Tilemap interactableTilemap,Item item,Vector2Int toolDirection)
11      {
12          if (GameManager.instance.tileManager.Check(tilemapPosition))
13          {
14
15          }
16          else { GameManager.instance.tileManager.PlowTile(tilemapPosition,toolDirection); return true; }
17          //Debug.Log("FasePlowTile");
18          return false;
19      }
```

Figura 86. Fragmento Código PlowTile.cs.

Después, en el script Player.cs cambiaremos el modo de llamar a las herramientas y haremos que pulsando V se llame al primer tipo y pulsando la barra espaciadora al segundo:

```
36          if (Input.GetKeyDown(KeyCode.Space) && selectable == true)
37          {
38              Vector3Int position2= GetTileBase(Input.mousePosition);
39              Vector3Int positionPlayer = GetTileBase(direction);
40
41              auxHor = position2.x - (positionPlayer.x+7);
42              auxVer = position2.y - (positionPlayer.y+4);
43              Vector2Int toolDirection = new Vector2Int(auxHor, auxVer);
44
45              GameManager.instance.toolManager.UseTool(position2, inventory.toolbar, direction,toolDirection);
46          }
47          if (Input.GetKeyDown(KeyCode.V) && selectable == true)
48          {
49              Vector3Int position2 = GetTileBase(Input.mousePosition);
50              Vector3Int positionPlayer = GetTileBase(direction);
51              auxHor = position2.x - (positionPlayer.x + 7);
52              auxVer = position2.y - (positionPlayer.y + 4);
53              Vector2Int toolDirection = new Vector2Int(auxHor, auxVer);
54
55              GameManager.instance.toolManager.UseTool2(inventory.toolbar,toolDirection);
```

Figura 87. Fragmento Código llamar Funcionalidades Herramientas Player.cs.

Por último, en Unity configuraremos los objetos para añadirles su funcionalidad, para ello añadiremos nuevas variable a ItemData.cs para representar las funcionalidades:

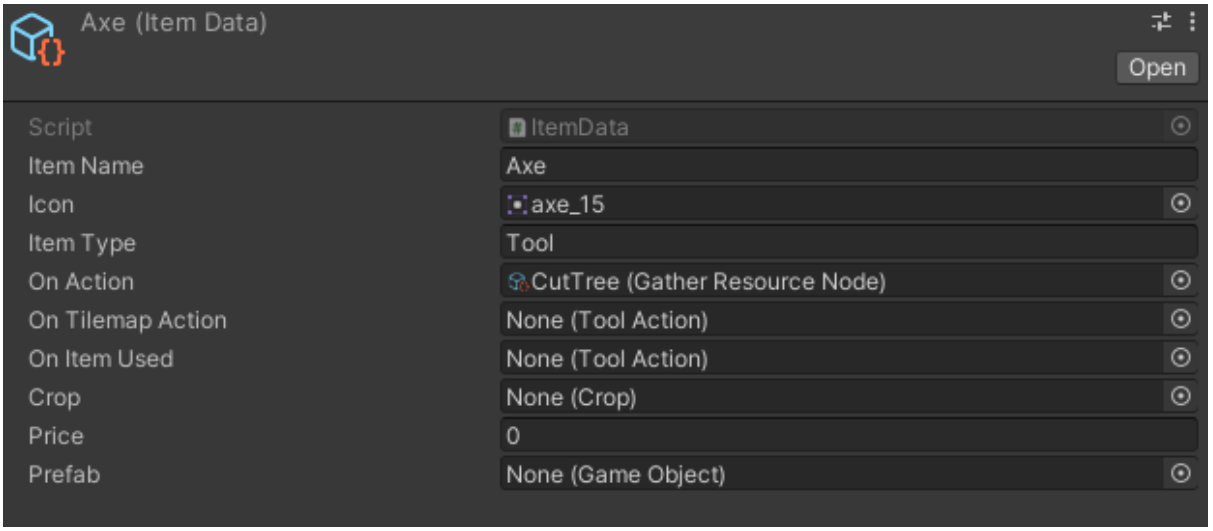


Figura 88. Configuración Objeto Herramienta desde Unity.

Pruebas

Para las pruebas se han testeado todas las herramientas con sus respectivos elementos, también se ha comprobado que solo la herramienta asociada pueda interactuar con su elemento asociado, haciendo que, por ejemplo, el pico pueda interactuar con las rocas pero no con los árboles.

Escalabilidad

Como se ha mencionado en el planteamiento, está mecánica es altamente escalable para un desarrollo de juego completo.

Por ejemplo, si en un futuro se añadieran menas de minerales, lo único que habría que hacer en tema de herramientas sería añadirlas a la lista de elementos interactuables del pico, consiguiendo así, una extrema simplicidad a la hora de escalar el videojuego con multitud de elementos.
Si se quisieran añadir más funcionalidades también sería sencillo, habría que determinar si esa funcionalidad interactuar con el mapeado o con otro elemento y después, crear un solo script que gestione la funcionalidad. Ejemplo: se implementa una regadora, esta interactúa con el mapeado, se crearía el script WaterPlant.cs que lo único que tendría que hacer es sobreescribir la función OnApplyToTilemap y generar el código para regar los cultivos.

6.17- Iteración 17- Crecimiento cultivos

Planteamiento Mecánica

En este incremento vamos a desarrollar el crecimiento de los cultivos. Esto consiste en hacer que las semillas que se han sembrado, se conviertan en cultivos de su tipo correspondiente y crezcan con el paso del tiempo.Por ello, los scripts irán asociados al agente de tiempo creado anteriormente.

Los sprites de los cultivos irán cambiando según en la fase de crecimiento en la que se encuentren. Cada vegetal tendrá entre 4 y 6 fases de crecimiento y un tiempo específico para cada una.

Al ser los cultivos instancias similares de unos a otros, necesitaremos crear un scriptable object que sirva de base para todos ellos, donde se definan el tiempo de crecimiento, la semilla asociada, el fruto asociado y los sprites pertinentes.

Selección sprites

Para los recursos gráficos se han escogido los siguientes del pack de recursos adquirido:

Figura 89. Sprites Cultivos.

Desarrollo scripts

En cuanto al desarrollo de scripts, se modificarán los scripts de ItemData.cs y TileManager.cs, además de la creación de Crop.cs.

Por lo que respecta a Crop.cs, este será el Scriptable Object mencionado anteriormente, que contendrá las características básicas de cada cultivo:

```
6   public class Crop : ScriptableObject
7   {
8       public int timeToGrow=10;
9       public ItemData yield;
10      //public Item toSpawn;
11      public int count = 1;
12      public List<Sprite> sprites;
13      public List<int> growthStageTime;
14  }
```

Figura 90. Fragmento Código Crop.cs.

En las modificaciones, ItemData.cs recibirá un nuevo parámetro: crop, qué hará referencia a si ese objeto tiene un cultivo asociado o no.

Por otra parte, en TileManager.cs, configuraremos la función de PlowTile, para que una vez el jugador are la tierra, se cree un objeto prefab invisible en la posición de la tierra arada y se añada a una lista con los demás cultivos correspondientes. Además, se implementará la función que gestionará los cultivos a lo largo del tiempo, comprobando en qué momento ha pasado suficiente tiempo para cambiar de fase de crecimiento o en qué momento está listo para cosechar:

```
64  public void Tick()
65  {
66      foreach(CropTile cropTile in crops.Values)
67      {
68          if (cropTile.crop == null) { continue; }
69          if (cropTile.Complete)
70          {
71              Debug.Log("Im done growing");
72              continue;
73          }
74          cropTile.growTimer += 1;
75
76          if (cropTile.growTimer >= cropTile.crop.growthStageTime[cropTile.growStage])
77          {
78
79              if (cropTile.growStage == 0) { cropsMap.SetTile(cropsInv[cropTile], null); }
80
81              cropTile.renderer.gameObject.SetActive(true);
82              cropTile.renderer.sprite = cropTile.crop.sprites[cropTile.growStage];
83              cropTile.growStage += 1;
84              Debug.Log("Pasando a fase:" + cropTile.growStage);
85          }
```

Figura 91. Función Tick() TileManager.cs.

Configurar Unity

En Unity, crearemos un nuevo objeto Crop y asignaremos los valores correspondientes para que se asocie con el primer cultivo, las zanahorias:

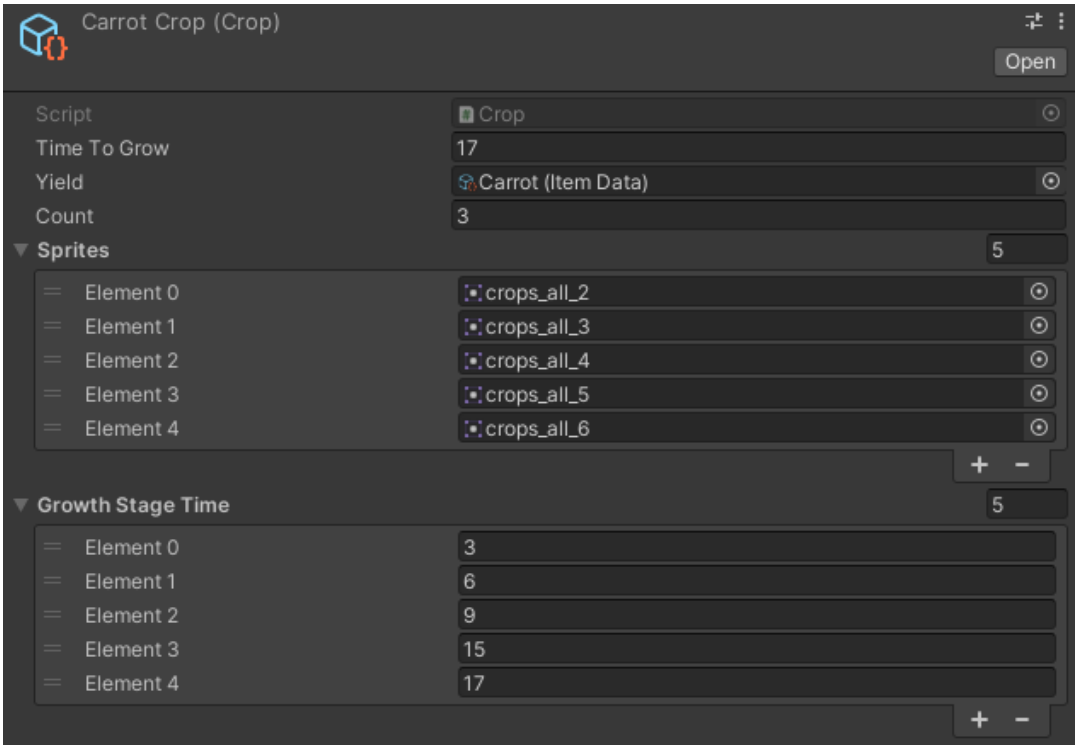


Figura 92. Configuración Elemento Crop desde Unity.

Pruebas

Se ha configurado un mensaje cada vez que el cultivo pasa de fase y se ha ejecutado el juego, se ha probado la mecánica de plantar semillas y se ha visto el correcto funcionamiento del crecimiento de cultivos.

6.18- Iteración 18- Cosechar cultivos

Planteamiento Mecánica

Se va a abordar la última iteración planificada, cosechar los cultivos una vez estén en su fase final de crecimiento.

Para conseguir esto, se va a implementar una nueva funcionalidad de las mencionadas en el incremento 16, concretamente OnTilePickUp. Como su nombre indica, interactúa con los recuadros del mapeado y sirve para recoger el elemento que esté sobre el recuadro con el que se interactúa.

Esta funcionalidad no la asociaremos a ninguna herramienta por el momento, sino que comprobaremos que cuando se pulse la barra espaciadora el jugador no tenga ningún objeto en su mano, lo que equivale a no tener nada equipado en la casilla seleccionada de la barra de herramientas.

Se quiere llevar a cabo la manera de poder interactuar con los cultivos ya maduros y poder cosecharlos para obtener varios de sus frutos.

Desarrollo scripts

Se creará el script de TilePickUpAction que lo que hará será lo mismo que SeedTile.cs o PlowTile.cs, sobrescribir la función OnApplyToTilemap y llamará a una función de TileManager que se encargará de la cosecha en sí.

En cuanto a la modificación de TileManager.cs, se añadirá la función OnPickUp, que lo primero que hace es comprobar si en ese recuadro existe un cultivo y en caso afirmativo, destruir el cultivo y crear sus frutos en el suelo, además de poner los valores de el cultivo en ese recuadro a cero:

```
130     internal void PickUp(Vector3Int tilemapPosition)
131     {
132         Vector2Int position = (Vector2Int)tilemapPosition;
133         if (crops.ContainsKey(position) == false) { return; }
134 
135         CropTile cropTile = crops[position];
136 
137         if (cropTile.Complete)
138         {
139             Debug.Log("croptile.crop.count:" + cropTile.crop.count);
140             GameManager.instance.itemSpawnManager.SpawnItem(tilemapPosition, cropTile.crop.yield, cropTile.crop.count);
141             cropsMap.SetTile(tilemapPosition, null);
142             cropTile.Harvested();
143         }
144     }
145 }
```

Figura 93. Función PickUp() TileManager.cs.

Configurar Unity

Una vez en Unity, se configurarán los objetos que suelta el cultivo, en este caso zanahorias, y se creará la ToolAction de Harvest.

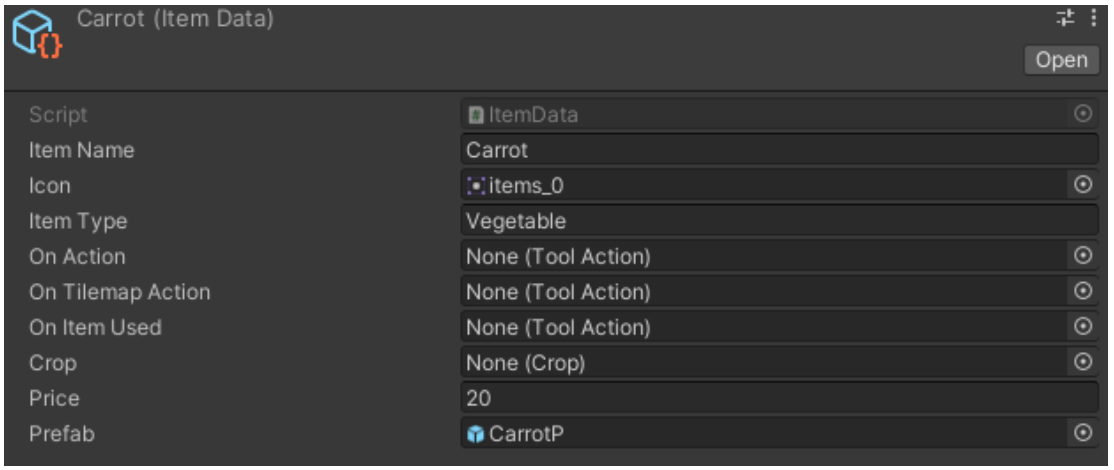


Figura 94. Configuración Objeto Vegetal en Unity.

Pruebas

Se ha comprobado que al pulsar la tecla espacio sobre un cultivo ya maduro este se destruye y se crean los frutos en el suelo listos para añadirlos en el inventario.

6.19- Iteraciones extra

Se han planteado varias iteraciones extra, que no estaban planificadas originalmente, esto se debe al que el desarrollador creía conveniente el estudio, diseño e implementación de estas mecánicas para acabar de redondear el prototipo y obtener un mejor resultado.

A continuación se van a explicar brevemente en qué consisten estos incrementos:

Extra 1- Tienda

Se ha implementado una versión básica de tienda, donde el usuario puede acceder con un click para comprar semillas o vender cultivos.

Para ello, ha sido necesario crear un pequeño sistema de economía, que suma o resta monedas al comprar o vender artículos en la tienda. La cantidad de monedas total aparece en la interfaz del día y la hora, justo debajo en otro recuadro, como ya se vió en la iteración 13. Además se ha creado un script para controlar en todo momento el incremento o decremento de las monedas y una función para actualizar la interfaz con cada cambio: CurrencyManager.cs

Se compone de dos interfaces conectadas entre sí, la primera es la de comprar. En ella el usuario puede pulsar los botones bajo los iconos con las semillas para comprar los objetos que aparecen en esos iconos. Se restará automáticamente el coste del producto que quiera adquirir a su cantidad total de dinero:

Figura 95. Captura Interfaz Tienda 1.

A la segunda interfaz, se accede con el botón de la derecha de la primera, y nos lleva a la interfaz de venta. Esta interfaz cuenta con un inventario de un único slot, donde podremos arrastrar los cultivos de nuestro inventario que queramos vender. Para venderlos, pulsaremos el botón que se encuentra justo al lado del slot y automáticamente destruirá el objeto y nos dará monedas según su valor, si es que tiene alguno:

Figura 96. Captura Interfaz Tienda 2.

Extra 2- Animales

Se ha comenzado a indagar en la dinámica de los animales, debido a que es una componente muy característica de los juegos del género. Se ha querido implementar una primera versión donde en el prototipo podamos observar una vaca dentro de un recinto.

La vaca tiene animaciones de movimiento propias y se ha generado un script para que cada 2 segundos, la vaca se mueva en una dirección, en caso de no tener un obstáculo justo en la dirección prevista, en ese caso, se quedará quieta. Después de cada animación, la vaca se quedará quieta durante 1.5 segundos aproximadamente.

El usuario será capaz de interactuar con ella pulsando la tecla E mientras se tiene el cursor del ratón sobre la vaca. Hacer esto resultará en la aparición de una viñeta con un icono de un corazón sobre la cabeza de la vaca.

Figura 97. Captura Interacción Animales.

7- Líneas futuras

Una vez realizados todos los incrementos planteados y teniendo una perspectiva global del prototipo y de lo que puede llegar a ser el videojuego completo, podemos afirmar que el futuro de este proyecto es muy amplio.

Se trata de un desarrollo de videojuego que, en el mejor de los casos, necesitaría otros dos o tres años para ver el final del desarrollo. En este prototipo se han realizado parte de las mecánicas más importantes que suelen tener títulos similares, pero son necesarias muchas más para obtener un juego que podamos considerar completo.

Sistema de combate, calabozos para asaltar, enemigos, sistema de estadísticas y habilidades, minijuegos , multijugador, diseño del mundo completo, sistema social o NPCs son algunas de las dinámicas que se pueden pensar en un primer momento como necesarias y que no están planteadas en este proyecto.

El deseo del desarrollador de continuar con el proyecto mientras lo compagina con la vida laboral que está por comenzar es clara, aunque primero habría que definir de una manera más clara cuál va a ser el rumbo del destino de este videojuego.

Sería necesario plantear cuáles van a ser los elementos distintivos de este videojuego sobre las otras decenas de juegos de este género que están saliendo en los últimos años debido a la influencia innegable de Stardew Valley, videojuego que generó una ingente cantidad de seguidores, entre los que me incluyo, y que resulta en toda una inspiración de superación al ver como una única persona desarrolló todo el proyecto desde la habitación de su casa.

Tener elementos distintivos no es realmente necesario, como hemos podido ver en el referente mencionado anteriormente, que es prácticamente un calco del Harvest Moon de la década de los 90, con mucho más contenido.

A medida que avance el tiempo, dentro de lo posible, se irán implementando algunas o todas las dinámicas mencionadas al principio de este capítulo, con la esperanza de ser capaz de publicar este juego en Steam para que los usuarios puedan disfrutar de lo que se espera, sea un buen juego.

Para terminar, como ideas que quedan en el tintero, se puede intentar compaginar un juego de esté estilo con un sistema de mazmorra vertical donde el jugador debe de escalar una torre de 100 pisos para recuperar sus recuerdos e ir ganando más materiales para progresar tanto en su granja, como en su armamento para afrontar los desafíos de la torre, al más puro estilo de los mangas coreanos.

Durante el ascenso del jugador por la torre, este encontraría además, personajes secundarios importantes para el devenir de la historia, que pueden cumplir una función en el pueblo cerca de la granja, por ejemplo, además de añadir más y más interés por escalar la torre.

Este sistema podría crear un loop jugable muy interesante para el usuario, que debería gestionar la dualidad de las dos ideas, por una parte la tranquilidad y el sosiego de la vida agrícola y, por otra, la desesperación e intensidad de escalar la torre, con momentos realmente intensos.

8- Conclusiones

Primero, desde un punto de vista más objetivo, podemos afirmar que el proyecto, pese a ser el primer videojuego o prototipo de este, que se ha realizado de forma individual, ha salido bastante bien. Se han conseguido todos los puntos propuestos, incluso se han añadido un par más para obtener un resultado más completo.

El desarrollo ha ido bien respecto al tiempo previsto, no se han necesitado muchas más horas de las previstas en la planificación inicial, aunque sí que se han dedicado infinidad de horas a mirar más tutoriales o documentación porque resultaba realmente interesante.

Se ha obtenido un prototipo final que cumple con las expectativas fijadas, el ser una demo para que se puedan probar las cosas más básicas de un juego de este estilo. Al ser un proyecto tan enfocado a la programación, se ha complicado el hecho de tener que explicar y reflejar el trabajo realizado, aunque creo que se ha conseguido también de una manera bastante satisfactoria.

Para concluir el proyecto, me gustaría compartir el sentimiento de realización que siento cuando miro todo lo que se ha sido capaz de crear desde 0. Todos los tutoriales, documentación, foros y pruebas han desembocado en un proyecto del que me siento realmente orgulloso. Se puede notar el esfuerzo y dedicación que hay detrás de cada una de las diferentes partes que se han querido realizar y todo el aprendizaje que se ha llevado a cabo.

También me quedo con todo lo aprendido durante el transcurso de este desarrollo, me he dado cuenta de la infinidad de cosas que se realizan en los videojuegos y he aprendido la manera de observar cada elemento para diseccionar en los diferentes aspectos necesarios para implementarlo.

He aprendido multitud de sistemas, trucos y estructuras de programación que, aunque había estudiado anteriormente en la carrera, tengo la sensación de que ahora soy mucho mejor programador que antes del proyecto. Antes, aunque tenía nociones de programación, me veía muy perdido a la hora de hacer cualquier cosa por mi cuenta, sin una guía, compañeros a los que consultar o profesores con los que contactar. No me imaginaba abordando un proyecto de programación de esta envergadura por ninguna cosa del mundo. Ahora, con toda la experiencia obtenida, y a punto de empezar mi carrera en el sector, me veo mucho más confiado y tengo las cosas muchísimo más claras de lo que soy capaz de hacer, he obtenido una especie de seguridad.

Gracias a este proyecto he acabado de confirmar que lo que realmente me gusta es el desarrollo de videojuegos, la libertad creativa, capacidad de resolución de problemas, las distintas maneras de abordar cualquier cosa y la infinidad de proyectos de la comunidad hacen que el desarrollo de videojuegos se convierta en una maravilla.

Claro está que en determinados puntos se siente desesperanza por no poder realizar alguna cosa que te habías propuesto, cuando estás una semana con el mismo problema y nadie en el mundo parece tener una respuesta a él, pero los momentos agradables superan con creces a los malos. Ese sentimiento de victoria cuando resuelves un error con el que te has estado peleando toda la semana, esa satisfacción al ver que lo que te habías imaginado en tu cabeza va tomando forma en un pequeño videojuego que lleva una parte de ti en su ser, son sentimientos realmente indescriptibles.

Por último, me gustaría agradecer a infinidad de gente, a mi familia, que me ha apoyado durante toda carrera, aunque aún no sepan bien que he estudiado, a mis amigos de toda la vida, que han estado ahí para cualquier cosa que he necesitado y a mis amigos de la universidad, que me han ayudado en cualquier consulta que he podido tener. También me gustaría agradecer a mi tutor del TFG, Jorge, por aceptar ser mi tutor y darme las indicaciones que he necesitado, pese a ser yo bastante alma libre en cuanto al proyecto en sí.

Figura 1. Carátula Harvest Moon. 2
Figura 2. Gameplay Harvest Moon. 3
Figura 3. Harvest Moon Nintendo Switch. 3
Figura 4. Carátula Animal Crossing 2001. 4
Figura 5. Invierno en Animal Crossing. 5
Figura 6. Museo en Animal Crossing. 5
Figura 7. Ilustración Animal Crossing 2020. 6
Figura 8. Carátula Stardew Valley. 6
Figura 9. Gameplay Stardew Valley. 7
Figura 10. Estaciones Stardew Valley. 7
Figura 11. Pesca Stardew Valley. 8
Figura 12. Diálogos Stardew Valley. 8
Figura 13. Paquete de Recursos Gráficos Agrícolas. 12
Figura 14. Paquete de Recursos Gráficos de Personajes. 12
Figura 18. Distribución Tareas 4. 17
Figura 19. Distribución Tiempo Tareas 1. 18
Figura 21. Diagrama de Gantt del proyecto. 19
Figura 22. Costos Materiales. 20
Figura 23. Artículo sobre sueldo de programador. 20
Figura 24. Artículo sobre sueldo de diseñador 1. 21
Figura 25. Artículo sobre sueldo de diseñador 2. 21
Figura 26. Costes Humanos. 21
Figura 27. Costes Totales. 22
Figura 28. Sprites Personaje. 24
Figura 29. Fragmento Código Movement.cs. 25
Figura 30. Captura de pantalla Animator General. 25
Figura 31. Captura de pantalla Animator Direcciones. 26
Figura 32. Sprite Semillas de Zanahoria. 26
Figura 33. Fragmento Código Collectable.cs. 27
Figura 34. Físicas de Recogibles. 27
Figura 35. Ejemplo Mapeado Stardew Valley. 28
Figura 36. Paleta de Recuadros disponibles. 29
Figura 37. Sprite Casa Jugador. 29
Figura 38. Mapeado desarrollado. 30
Figura 39. Capas del Mapeado. 30
Figura 40. Fragmento Código Inventory.cs. 31
Figura 41. Función Add() Inventory.cs. 32
Figura 42. Ejemplo Inventario Stardew Valley. 33
Figura 43. Fragmento Código Inventory_UI.cs. 34

Figura 45. Jerarquía Slots. 35
Figura 46. Funciones Remove() Inventory.cs. 36
Figura 47. Funciones Inventory_UI.cs. 37
Figura 48. Función DropItem() Player.cs. 38
Figura 49. Capa Interactuable Tilemap. 39
Figura 50. Sprites necesarios para Arar Tierra. 39
Figura 51. Sprites Animación Azada. 40
Figura 52. Fragmento Código Player.cs. 40
Figura 54. Animator de la Azada. 41
Figura 55. Fragmento Código ItemData.cs. 42
Figura 57. Item Data dentro de Unity. 43
Figura 58. Inspector de Item dentro de Unity. 43
Figura 59. Sprite Barra de Herramientas. 44
Figura 60. Fragmento Código Toolbar_UI.cs. 45
Figura 61. Captura de Pantalla de la Barra de Herramientas dentro del Juego. 45
Figura 62. Sprites Inventario. 46
Figura 63. Función SlotBeginDrag() Inventory_UI.cs. 48
Figura 64. Función MoveSlot() Inventory.cs. 48
Figura 65. Fragmento Código UI_Manager.cs. 48
Figura 66. Event Trigger Slots. 49
Figura 67. Event Trigger Panel de Soltar Objetos. 49
Figura 68. Funciones Refresh() UI_Manager.cs. 50
Figura 69. Función GetInventoryUI UI_Manager.cs. 50
Figura 71. Sprites Interfaz Fecha y Hora. 51
Figura 72. Fragmento Código DayTimeController.cs. 52
Figura 73. Funciones DayLight() y NextDay() DayTimeController.cs. 52
Figura 74. Interfaz Fecha y Hora. 53
Figura 76. Código Tiempo Interno DayTimeController.cs. 54
Figura 77. Fragmento Código TimeAgent.cs. 55
Figura 78. Sprite Semillas. 56
Figura 79. Función SeedTile() TileManager.cs. 56
Figura 80. Sprites Varios. 57
Figura 81. Fragmento Código ToolAction.cs. 57
Figura 82. Fragmento Código ResourceNode.cs. 58
Figura 83. Fragmento Código GatherResourceNode.cs. 58
Figura 84. Captura Creación ToolAction. 59
Figura 85. Captura Configuración GatherResourceNode en Unity. 59
Figura 86. Fragmento Código PlowTile.cs. 59
Figura 87. Fragmento Código llamar Funcionalidades Herramientas Player.cs. 60
Figura 88. Configuración Objeto Herramienta desde Unity. 60
Figura 89. Sprites Cultivos. 61
Figura 90. Fragmento Código Crop.cs. 62

Figura 92. Configuración Elemento Crop desde Unity. 63
Figura 93. Función PickUp() TileManager.cs. 64
Figura 94. Configuración Objeto Vegetal en Unity. 65
Figura 95. Captura Interfaz Tienda 1. 66
Figura 96. Captura Interfaz Tienda 2. 66
Figura 97. Captura Interacción Animales. 67

9- Bibliografía

[1] OVERCLOCKERS UK. *The history of Harvest Moon, and how it became Story Of Seasons-Jacob Smith.* 9 de septiembre de 2022. https://www.overclockers.co.uk/blog/the-history-of-harvest-moon-and-how-it-became-story-of-seasons/

[2] NPC GAMING GROUP. *The History of Harvest Moon - The Game That Inspired Stardew Valley!* -[video]. *YouTube.* 04 de octubre de 2020. Disponible en: https://www.youtube.com/watch?v=e4xpSCp-u4E

[3] RETROGAMER TEAM. *The history of Animal Crossing: How the series evolved from an N64 oddity to a Nintendo Switch system seller.* 14 de abril de 2021. https://www.gamesradar.com/the-history-of-animal-crossing/

[4] NINTENDOLIFE. *Feature: Animal Crossing: A Brief History-Alex Olney & Gavin Lane-* 11 de marzo de 2020. https://www.nintendolife.com/news/2020/03/feature_animal_crossing_a_brief_history

[5] 3DJUEGOS. *¿Qué es un motor de juego? Estos 5 son los más importantes de la industria-Diego Emegé-* 28 de abril de 2022. https://www.3djuegosguias.com/otros-generos/que-motor-juego-estos-5-importantes-industria

[6] OBSERVATORIO GATE. *¿Qué es un motor de videojuegos?-Alberto Carrasco Carrasco-* 4 de julio de 2018. https://blogs.upm.es/observatoriogate/2018/07/04/que-es-un-motor-de-videojuegos/

[7] ITCH.IO. *Paquete de Recursos Gráficos Cozy Farm-Shubibubi. Disponible en:* https://shubibubi.itch.io/cozy-farm

[8] ITCH.IO. *Paquete de Recursos Gráficos Cozy People-Shubibubi. Disponible en:* https://shubibubi.itch.io/cozy-people

[9] TOKYO SCHOOL. *¿Cuál es el sueldo de un programador con Unity?.* https://www.tokioschool.com/formaciones/cursos-videojuegos/programacion-unity/sueldo/

[10] TOKYO SCHOOL. *Sueldo de un diseñador de videojuegos: ¿cuánto cobra?.* https://www.tokioschool.com/formaciones/cursos-videojuegos/diseno/sueldo/

[11] CURSOS.COM . *Sueldo de un diseñador de videojuegos* https://cursos.com/cursos/informatica/diseno-videojuegos/sueldo/#:~:text=Dise%C3%B1ador%20junior.,los%2026.000%E2%82%AC%20brutos%20anuales

[12] CALLIGRAPHR . *Herramienta para desarrollar fuentes de texto. Disponible en:* https://www.calligraphr.com/es/

[13] UNITY. *The Sprite Editor - Unity Official Tutorials*-[video]. *YouTube.* 10 de marzo de 2014. Disponible en: https://www.youtube.com/watch?v=gbgIA3pwpHc

[14] FORUMS UNITY. *Calling function from other scripts c#*-[forum]. Disponible en: https://forum.unity.com/threads/calling-function-from-other-scripts-c.57072/

[15] BMO. *Quick and Easy Unity Debugging Basics*-[video]. *YouTube.* 20 de abril de 2022. Disponible en: https://www.youtube.com/watch?v=wt2YdrUlzgg

[16] UNITY DOCUMENTATION. *Serialización de Script*-[documentación]. Disponible en: https://docs.unity3d.com/es/530/Manual/script-Serialization.html

[17] TERRESQUALL. *Creating a Farming RPG (like Harvest Moon) in Unity*-[playlist]. *YouTube.* https://www.youtube.com/playlist?list=PLgXA5L5ma2Bu1sWc_-ZGRuPSUEAKPktHF

[18] UNITY DOCUMENTATION. *Static GameObjects*-[documentación]. Disponible en: https://docs.unity3d.com/Manual/StaticObjects.html

[19] BRACKEYS. *2D Animation in Unity (Tutorial)*-[video]. *YouTube.* 05 de agosto de 2018. Disponible en: https://www.youtube.com/watch?v=hkaysu1Z-N8

[20] COCO CODE. *Unity INVENTORY: A Definitive Tutorial*-[video]. *YouTube.* 29 de septiembre de 2022. Disponible en: https://www.youtube.com/watch?v=oJAE6CbsQQA

[21] COCO CODE. *Drag and drop in Unity UI - create your own inventory UI!* -[video]. *YouTube.* 08 de septiembre de 2022. Disponible en: https://www.youtube.com/watch?v=kWRyZ3hb1Vc

[22] TARODEV. *How to make a menu in Unity - UI Tutorial*-[video]. *YouTube.* 04 de septiembre de 2021. Disponible en: https://www.youtube.com/watch?v=lF26yGJbsQk

[23] GAMEDEV . *Classes in Unity - internal vs public.* https://gamedev.stackexchange.com/questions/146940/classes-in-unity-internal-vs-public

[24] UNITY DOCUMENTATION. *Universal Render Pipeline overview*-[documentación]. Disponible en: https://docs.unity3d.com/Packages/com.unity.render-pipelines.universal@16.0/manual/#:~:text=The%20Universal%20Render%20Pipeline%20

[25] JAVA T POINT. *Unity UI Text*-[documentación]. Disponible en: https://www.javatpoint.com/unity-ui-text#:~:text=To%20insert%20a%20Text%20UI,to%20show%20in%20that%20field

[26] UNITY. *C# Overriding in Unity! - Intermediate Scripting Tutorial*-[video]. *YouTube.* 18 de julio de 2019. Disponible en: https://www.youtube.com/watch?v=h0J4gs4DW5A

[27] FORUMS UNITY. *[c#] How can I let something happen after a small delay?*-[forum]. https://discussions.unity.com/t/c-how-can-i-let-something-happen-after-a-small-delay/117594

[28] FORUMS UNITY. *Instatiating a GameObject from a Scriptable Object*-[forum]. https://forum.unity.com/threads/instatiating-a-gameobject-from-a-scriptable-object-to-a-parent-causing-weird-spawn-positions.560413/

[29] UNITY DOCUMENTATION. *Font Asset Creator*-[documentación]. Disponible en: https://docs.unity3d.com/Packages/com.unity.textmeshpro@4.0/manual/FontAssetsCreator.html

[30] UNITY DOCUMENTATION. *Event Trigger*-[documentación]. Disponible en: https://docs.unity3d.com/es/2018.4/Manual/script-EventTrigger.html

[31] GAMEDEVBEGINNER. *Scriptable Objects in Unity (how and when to use them)*-John French-22 de septiembre de 2022: https://gamedevbeginner.com/scriptable-objects-in-unity/

[32] GAMEDEVBEGINNER. *How to use Sorting Layers in Unity*-John French-02 de diciembre de 2021. Disponible en: https://gamedevbeginner.com/how-to-use-sorting-layers-in-unity/

[33] FRIENDLYCOSMONAUT. *Farming RPG Tutorial: GMS2*-[playlist]. *YouTube.* https://www.youtube.com/playlist?list=PLSFMekK0JFgzbFfj1vAsyluKTymnBiriY

10- Enlaces a archivos

- Enlace Proyecto de Unity: https://drive.google.com/file/d/1fFHc8KXPElQ1BbkLgXy59jQZ_B9h36UB/view?usp=sharing
- Enlace a Ejecutable: Dentro hay un readme.txt con los controles. El ejecutable es “FarmingGame.exe” : https://drive.google.com/file/d/1KLYJDh4P0SdIaT4rHV8nTg7I9WFqjGZh/view?usp=sharing