TRABAJOS FIN DE GRADO

Universidad de Jaén

Escuela Politécnica Superior de Jaén

DESARROLLO DE UN PROTOTIPO DE VIDEOJUEGO

Autor: José Luis Díaz Verdejo

Grado: Grado en Ingeniería Informática

Tutores: Juan Roberto Jiménez Pérez y Francisco Daniel Pérez Cano

Departamento del director: Informática

Fecha: Septiembre de 2024

Licencia CC

CREA

(Página intencionalmente en blanco)

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Informática

Don Juan Roberto Jimenez Pérez y Don Francisco Daniel Pérez Cano, tutores del Proyecto Fin de Grado titulado: “**Desarrollo de un prototipo de videojuego**”, que presenta José Luis Díaz Verdejo, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, Septiembre de 2024

El alumno: Los tutores:

José Luis Díaz Verdejo

Juan Roberto Jiménez Pérez

Y

Francisco Daniel Pérez Cano

Agradecimientos.

Me gustaría dar las gracias a todas las personas que me han apoyado durante el transcurso de este largo camino a través de los años invertidos en la Universidad.

A mis padres y a mi hermana por haber sido mi apoyo durante todos estos años. En especial a mi padre, que aunque ya no esté, espero que pueda sentirse orgulloso.

A mis tutores por toda su paciencia y por su guía para realizar este proyecto. A Francisco Daniel, sin su ayuda y sus ánimos esto no hubiera sido posible.

A mis amigos y compañeros, José Jiménez por convencerme para entrar en la universidad y Ángel Conde por haber sido mi mentor.

Índice de contenido.

Agradecimientos. 3
1. Introducción 9
 1.1. Objetivos 13
 1.2. Metodología. 13
2. Análisis 15
 2.1. Estado del arte en juegos de construcción de poblados 15
 2.2. Software utilizado. 20
 2.3. Análisis de requisitos. 23
  2.3.1. Requisitos funcionales. 25
  2.3.2. Requisitos no funcionales. 26
 2.4. Planificación. 27
  2.4.1. Estimación de tiempo 28
  2.4.2. Diagrama de Gantt. 30
 2.5. Estimación de costes. 36
  2.5.1. Costes asociados al software. 36
  2.5.2. Costes asociados al hardware. 36
  2.5.3. Costes asociados al personal 37
  2.5.4. Coste total. 39
3. Diseño. 41
 3.1. Diagrama de casos de uso principal. 41
 3.2. Diagrama de componentes 42
 3.3. Primer incremento: Terreno. 43
 3.4. Segundo incremento: Sistema de cuadrícula. 44
 3.5. Tercer incremento: Sistema de cámara. 45
 3.6. Cuarto incremento: Sistema de construcción. 46
 3.7. Quinto incremento: Sistema de agentes. 48
 3.8. Sexto incremento: Creación de mundo y economía. 49
 3.9. Séptimo incremento: Interfaz. 50
4. Desarrollo. 53
 4.1. Primer incremento: terreno. 54
 4.2. Segundo incremento: Sistema de cuadrícula. 57
 4.3. Tercer incremento: Sistema de cámaras. 63

4.4. Cuarto incremento: Sistema de construcción. 67
4.5. Quinto incremento: Sistema de agentes. 70
4.6. Sexto incremento: Generación de mundo y economía. 77
4.6.1. Generación de mundo. 77
4.6.2. Economía del juego. 81
4.7. Séptimo incremento: Interfaz. 82
4.8. Octavo incremento: Ajustes finales y pruebas. 86
4.8.1. Pruebas. 86
4.8.2. Corrección de errores. 88
5. Conclusiones y trabajo futuro. 91
Apéndice I. Guía de instalación. 93
Apéndice II. Guía de usuario. 95
Apéndice III. Recursos del juego no referenciados. 98
Apéndice IV. Descripción del material entregado. 99
Bibliografía 100

Índice de ilustraciones.

Ilustración 1.1: Ganancias en la industria del videojuego por año...................... 9
Ilustración 1.2: Número de jugadores de videojuegos a través del tiempo. 10
Ilustración 1.3: Número de videojuegos con la etiqueta "Indie" en Steam por año. 11
Ilustración 1.4: Número de lanzamientos de videojuegos con la etiqueta City Builder en Steam por año. 12
Ilustración 1.5: Esquematización del modelo incremental [9]. 14
Ilustración 2.1: SimCity, 1989. 15
Ilustración 2.2: Captura de pantalla del videojuego Banished. 17
Ilustración 2.3: Captura de pantalla del videojuego Foundation. 18
Ilustración 2.4: Captura del juego Cities: Skylines. 19
Ilustración 2.5: Captura de pantalla del programa Visual Studio 2019. 22
Ilustración 2.6: Captura de pantalla del programa Blender. 23
Ilustración 2.7: Sistema de cuadrícula del juego Cities: Skylines. 24
Ilustración 2.8: Sistema de recursos del juego Anno 1404. 25
Ilustración 2.9: Diagrama de Gantt, parte 1. 31
Ilustración 2.10: Diagrama de Gantt, parte 2. 32
Ilustración 2.11: Diagrama de Gantt, parte 3. 33
Ilustración 2.12: Diagrama de Gantt, parte 4. 34
Ilustración 2.13: Diagrama de Gantt, parte 5. 34
Ilustración 3.1: Diagrama de casos de uso del prototipo. 42
Ilustración 3.2: Diagrama de componentes del sistema. 43
Ilustración 3.3: Boceto del sistema de cuadrícula. 44
Ilustración 3.4: Sistema de cámara del videojuego Age of Empires II. 46
Ilustración 3.5: Diagrama de actividades del sistema de construcción. 47
Ilustración 3.6: Diagrama de actividad del sistema de agentes. 49
Ilustración 3.7: Diseño de la interfaz del menú principal. 51
Ilustración 3.8: Diseño de la interfaz del juego. 52
Ilustración 4.1: Herramienta de Unity para modificar el terreno........................ 56
Ilustración 4.2: Resultado final del terreno del prototipo. 56
Ilustración 4.3: Shader para representar visualmente las casillas. 58
Ilustración 4.4: Visualización del sistema de cuadrícula en el mapa. 59
Ilustración 4.5: Captura del script “ObjectsDatabaseSO”. 60
Ilustración 4.6: Uso del patrón Singleton de la clase GridData......................... 61
Ilustración 4.7: Campo de visión de la cámara principal del juego. 64
Ilustración 4.8: Líneas de código asociadas a los límites del mundo. 64
Ilustración 4.9: Vista de la cámara principal. 65
Ilustración 4.10: Campo de visión de la cámara del mini mapa........................ 66
Ilustración 4.11: Funcionamiento de la cámara del mini mapa. 66
Ilustración 4.12: Assets utilizados para las casas. 67
Ilustración 4.13: Edificios colocados en el prototipo. 69

Ilustración 4.14: Edificios presentes en el sistema mediante el inspector. 69
Ilustración 4.15: Parámetros de la superfie de la malla de navegación y localización del botón "bake". 71
Ilustración 4.16: NavMesh de la escena. 72
Ilustración 4.17: Parámetros y estética del NavMeshAgent. 73
Ilustración 4.18: Instancia de un agente leñador. 74
Ilustración 4.19: Flujo de animaciones de los agentes. 76
Ilustración 4.20: Assets empleados para los recursos existentes en el juego. . 77
Ilustración 4.21: Recursos del juego tras haber modificado su escala y rotación. 79
Ilustración 4.22: Resultado de la generación de mundo. 80
Ilustración 4.23: Interfaz del menú inicial. 83
Ilustración 4.24: Panel superior de la interfaz principal del juego. 84
Ilustración 4.25: Panel inferior de la interfaz principal del juego. 84
Ilustración 4.26: Resultado del mini mapa en la interfaz principal del juego. ... 85
Ilustración 4.27: Composición final de la interfaz en el prototipo. 86
Ilustración 4.28:Solución propuesta para el error con el sistema de construcción. 89
Ilustración 4.29: Línea de código necesaria para actualizar la malla de navegación. 90
Ilustración A II. 1: Menú inicial de la aplicación.. 95
Ilustración A II. 2: Vista principal del juego.. 96
Ilustración A II. 3: Vista del menú construcción del juego.. 97

Índice de tablas.

Tabla 2.1: Tiempo estimado total del proyecto. 30
Tabla 2.2: Estimación de costes del software. 36
Tabla 2.3: Estimación de costes del hardware. 37
Tabla 2.4: Estimación de costes del personal. 39
Tabla 2.5: Resumen del coste total del proyecto.. 40
Tabla 4.1: Coste de los edificios... 87

1. Introducción

La industria del videojuego a día de hoy es una industria totalmente consolidada, y al contrario que otros mercados digitales, las revoluciones y cambios tecnológicos como han sido la irrupción de los teléfonos inteligentes o la apuesta de compañías como Microsoft de dar la posibilidad de jugar en la nube no han hecho más que paulatinamente atraer a nuevos usuarios al mercado e incrementar la facturación y beneficios tanto de grandes compañías con miles de trabajadores como de pequeños desarrolladores llamados Indie, término procedente de la abreviatura de la palabra Independiente.

Se estima que la industria en 2023, tras la pequeña recesión vivida en 2022 a causa del impacto en los mercados de la invasión rusa de Ucrania, tuvo unas ganancias de 249,58 mil millones de dólares americanos y que sus expectativas de crecimiento sean positivas en los años venideros como podemos ver en la Ilustración 1.1 [1].

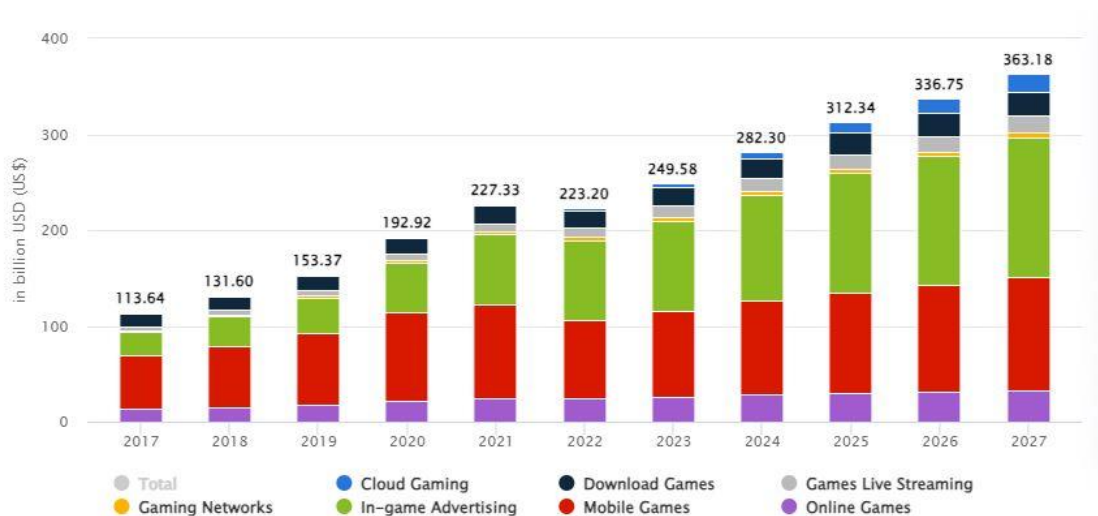


Ilustración 1.1: Ganancias en la industria del videojuego por año.

También se estima que el número de jugadores crece año tras año, ya que mientras las formas tradicionales de este tipo de ocio tienen un crecimiento más modesto, la llamada “casualización” en los videojuegos con los juegos móviles, la mejora de accesibilidad y nuevos controles adaptados y la reducción del costo de acceso al sector tanto en hardware como en software nos da una previsión

de 1,472.0 millones de usuarios para el año 2027, o lo que es lo mismo, un crecimiento de aproximadamente un 92,72% en un plazo de 10 años partiendo de 2017 como podemos observar en la Ilustración 1.2 [2].

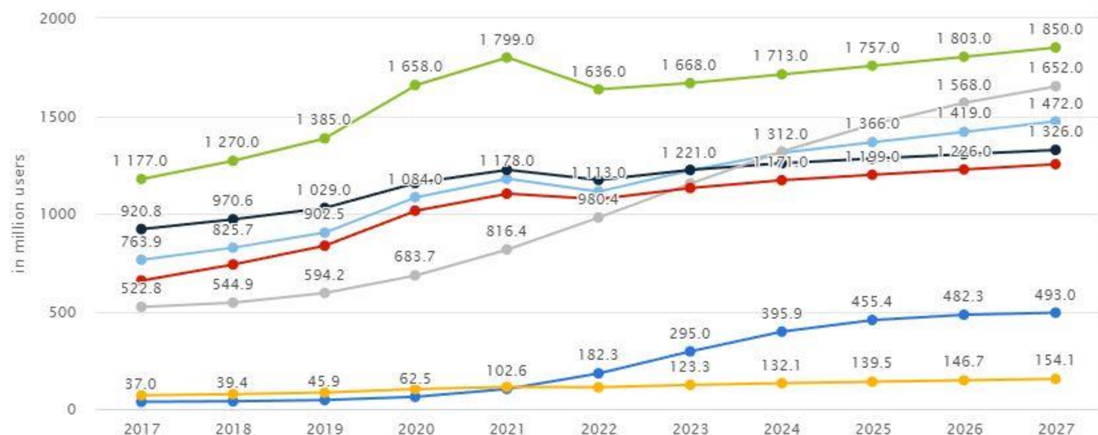


Ilustración 1.2: Número de jugadores de videojuegos a través del tiempo.

Los llamados desarrolladores o estudios *Indie* son aquellos que bien trabajan solos o bien el equipo consta de unas pocas personas. La proliferación de este tipo de estudios se debe principalmente a la accesibilidad a información y recursos para el aprendizaje de nuevas tecnologías como la programación, el modelado 3D o el dibujo digital, teniendo muchos repositorios, foros y canales donde no solo hay información, si no donde desarrolladores se ayudan mutuamente.

El coste de utilización de este tipo de software en muchos casos es gratis o con costes reducidos al desarrollo *Indie*. Por ejemplo, para el modelado 3D existe Blender, una herramienta gratuita y de fuentes abiertas muy potente, para el dibujo digital está Krita, también gratuita y de fuentes abiertas y para el desarrollo de videojuegos existen dos potentes motores ampliamente utilizados como son Unity, propietario pero gratuito bajo algunas condiciones como no sobrepasar los 100.000$ de facturación [3] o Unreal Engine, también propietario y gratuito siempre que no superes el millón de dólares en ingresos brutos anuales [4].

Siendo el coste del software empresarial una gran barrera para el aprendizaje y puesta en marcha de pequeños proyectos, la ausencia de este

gasto ha permitido la popularización de este tipo de desarrollos. Como podemos ver en la Ilustración 1.3, el número de juegos lanzados en la principal plataforma de distribución digital, Steam, es de aproximadamente 6.000 videojuegos al año [5] obteniendo el 47% de todas las ventas en la plataforma a lo largo de este año [6].

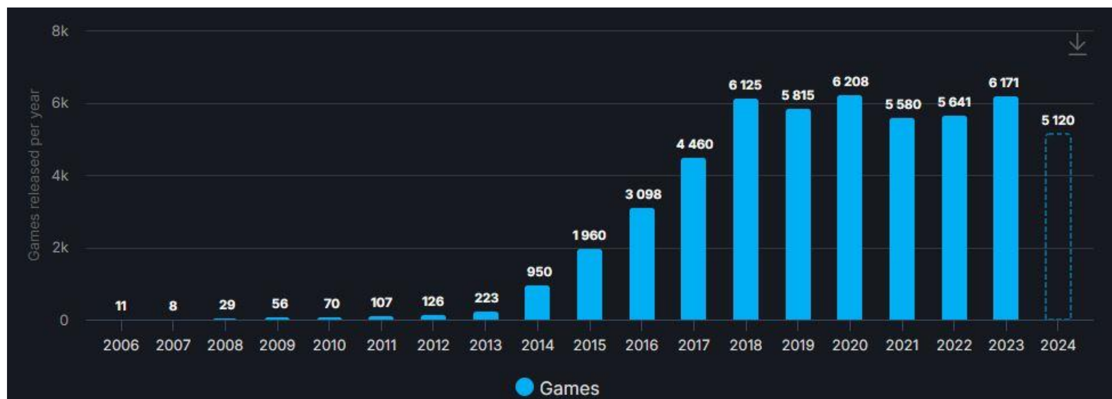


Ilustración 1.3: Número de videojuegos con la etiqueta "Indie" en Steam por año.

Por último, ocupándonos ya del género elegido para el desarrollo del prototipo, el de construcción de ciudades, representa solo aproximadamente el 2% del total de juegos de la plataforma Steam [7] con alrededor de 2.300 juegos en total.

Complementariamente, como podemos observar en la Ilustración 1.4, es un tipo juego que, aunque sus precursores y primeros ejemplos datan de los años 80 y 90, no ha sido hasta recientemente y gracias a un caso en especial llamado *Banished* (ver el apartado Estado del Arte en la sección de Análisis más adelante) que el género ha conseguido despegar.

Este tipo de juegos aún no se han popularizado entre las grandes masas de jugadores y salvo contadas excepciones no suelen tener una gran repercusión mediática, por lo que todos estos datos nos catalogan actualmente el género como “género nicho”. Esto quiere decir que están enfocados a un público específico y reducido pero que por normal general es exigente y fiel al género.

Ilustración 1.4: Número de lanzamientos de videojuegos con la etiqueta City Builder en Steam por año.

Para finalizar el Grado en Ingeniería Informática he decidido realizar este Trabajo de Fin de Grado que se centra en el desarrollo de un prototipo de un videojuego de construcción de ciudades.

En este documento se detalla el progreso seguido para la creación del prototipo con el siguiente orden:

- Identificar las necesidades básicas del prototipo y documentar el proceso de creación del mismo.
- Análisis del estado del arte en el sector, mecánicas a aplicar, software a utilizar y planificación del proyecto.
- Diseño de la interfaz, mecánicas empleadas y factores a considerar en el prototipo de videojuego.
- Proceso de implementación y creación del prototipo gracias a la metodología estudiada.
- Fase de pruebas y ajuste de balance para la economía y progresión del videojuego.

1.1. Objetivos

El principal objetivo de este proyecto es la creación de un prototipo de videojuego de construcción de ciudades para PC.

Dada la limitación de horas para la realización de un proyecto de tanta envergadura como es un videojuego en 3D, este prototipo intentará tener en cuenta en todo momento la escalabilidad de la aplicación para tener una base sólida de la que poder partir para la consecución final de un videojuego, teniendo las mecánicas básicas de este tipo de juegos como la construcción de edificios, la recolección de recursos y la inteligencia artificial básica de los agentes.

El usuario podrá seleccionar y construir en un mapa edificios a cambio de recursos que serán recolectados mediante el uso de agentes autónomos.

El proceso de estudio del mercado y el análisis y posterior utilización de software empleado en la industria me dará una base de experiencia para la preparación de un futuro laboral en la misma. Además, la consecución del prototipo y su probable continuación posterior hasta alcanzar un producto final me darán un elemento importante en mi porfolio personal.

1.2. Metodología.

Para la realización de este prototipo he considerado que la mejor opción es utilizar una metodología incremental de desarrollo software [8]. En esta metodología el proyecto se subdivide en pequeños módulos o incrementos que añaden partes de funcionalidades hasta obtener el total (Ilustración 1.5). Una de las características más importantes de dicha metodología es que cada incremento es independiente y funcional por sí solo, por lo que la sensación de avance en el proyecto es palpable en cada entrega.

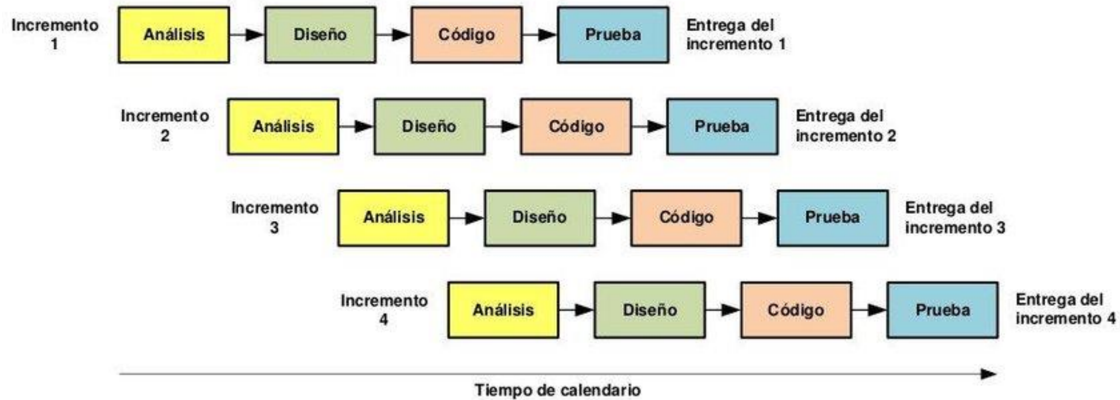


Ilustración 1.5: Esquematización del modelo incremental [9].

Esta metodología es idónea para el desarrollo de videojuegos, ya que normalmente estos están compuestos por diferentes partes (inteligencia artificial de enemigos, diseño de niveles etc.) que se pueden tratar como los incrementos anteriormente vistos, por lo que cada incremento se puede ajustar a cada uno de ellas.

Como podemos observar en la ilustración anterior, la planificación se divide en los siguientes puntos:

- Análisis: Recopilar y priorizar las necesidades de cada iteración.
- Diseño: Diseñar la funcionalidad propia de cada incremento.
- Código: Desarrollo del incremento con los datos previamente obtenidos.
- Prueba: Realización de pruebas para ver el correcto funcionamiento del incremento desarrollado.

2. Análisis

En este apartado procedo a explicar el proceso de estudio previo, planificación y razonamiento a aplicar para la construcción del prototipo, así como las mecánicas y sistemas que voy a emplear.

2.1. Estado del arte en juegos de construcción de poblados

Los juegos de construcción de ciudades han evolucionado mucho a través del tiempo, desde el precursor del género como Utopía (1982) [10] en la que se mezclaba la estrategia en tiempo real, también llamada RTS, con las primeras nociones sobre construir ciudades, o SimCity (1989, Ilustración 2.1) que por primera vez unió la construcción de ciudades con la gestión de recursos, además de enfatizar la continua construcción, desarrollo y expansión de la ciudad en lugar de establecer condiciones de victoria [11]. Posteriormente estas dos características se convertirían en estándares a seguir por multitud de título a lo largo de los años.

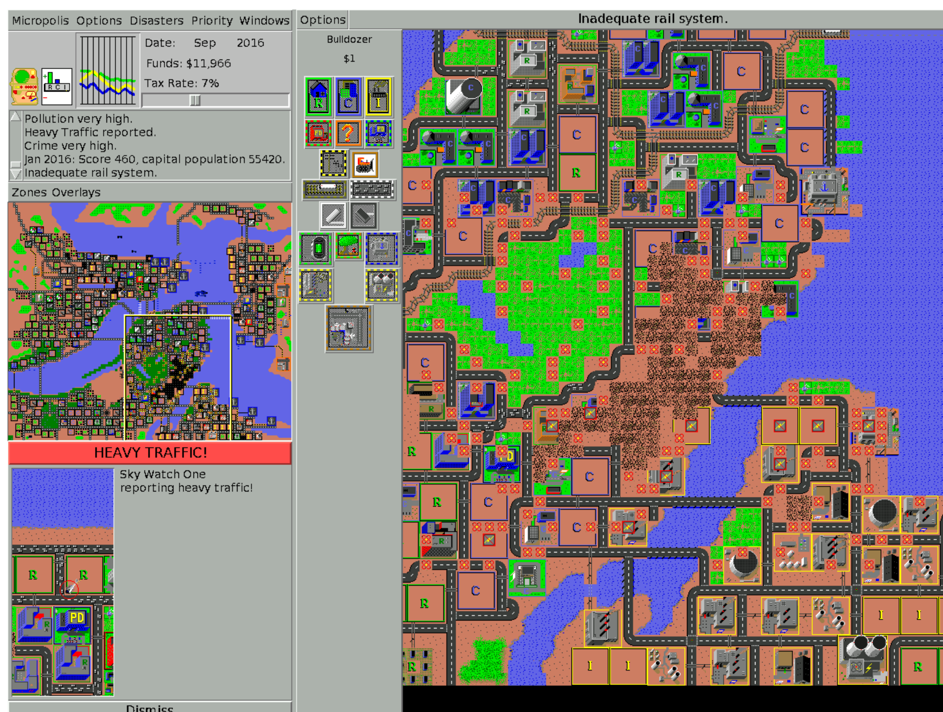


Ilustración 2.1: SimCity, 1989.

En los últimos años el género ha gozado de cierta popularidad gracias a la revitalización dada por el juego *Banished*. Voy a hacer un pequeño análisis de *Banished, Foundation y Cities Skylines*, juegos que han influido en la realización del prototipo. Y más concretamente los sistemas clave que voy a emplear para darle forma al mismo, que son el sistema de construcción, la economía y el comportamiento de los agentes.

Banished (Ilustración 2.2) [12] es un videojuego lanzado en 2014 basado en una aldea medieval realista cuyo enfoque en la dificultad de la supervivencia de sus ciudadanos a través de la gestión de recursos le dieron un toque diferente y desafiante al jugador. Además, fue desarrollado por una sola persona con unas ventas estimadas de 2.7 millones de copias [13], siendo un gran éxito, lo que influyó a muchos desarrolladores a seguir su camino y comenzar a desarrollar sus propios títulos.

Como resumen de sus sistemas tenemos:

- Sistema de construcción: el sistema es el clásico por casillas. Seleccionas el edificio a construir, su localización y los agentes se encargan en despejar la zona, llevar los recursos y construirlo.
- Economía: resta importancia a la acumulación de riquezas para enfocarse en la obtención de recursos para la supervivencia de los agentes, siendo vitales elementos como la comida o leña para calefacción.

  Hay diferentes edificios para cada uno de los recursos a obtener.

- Agentes: el jugador les asigna el trabajo a ejercer y se dedican exclusivamente a ello, como puede ser constructor o minero.

  Siguen un sistema realista de vida, en el que envejecen, reproducen si viven en una casa con una persona en edad adulta del género opuesto y mueren incrementando la probabilidad en función a mayor edad.

Ilustración 2.2: Captura de pantalla del videojuego Banished.

Foundation, Ilustración 2.3 [14], es un juego aún en desarrollo, pero con versión jugable, cuya principal característica es su sistema de construcción autónoma sin celdas delimitadas por zonas que el jugador dibuja en el mapa. Otra característica importante es la creación de caminos mediante su propia generación a través de un sistema de camino de hormigas en el que su tamaño depende del número de agentes que transiten, dándole un aspecto orgánico al poblado.

Emplea los siguientes sistemas:

- Sistema de construcción: se divide en dos mecánicas.

  Para los edificios residenciales se pinta una zona en el mapa y los propios agentes se encargan de construir las viviendas necesarias en función al valor de la zona, encontrando más atractivo vivir cerca de mercados o iglesias que de fábricas.

  Para los edificios de producción y recolección el jugador los coloca manualmente.

- Economía: sigue un sistema de recolección y procesamiento de recursos para obtener nuevos y construir edificios de mayor nivel. También existe un mercado con el que comerciar tus recursos y obtener oro que puedes emplear tanto para comprar recursos faltantes o ampliar tu territorio.
- Agentes: los agentes son semiautónomos, el jugador selecciona su profesión, la cual mejora su eficiencia mediante el tiempo y un sistema de niveles, y ellos solos se encargan de trabajar, recolectar y construir.

La población depende de un sistema de felicidad que se da cuando obtienen los recursos necesarios en función de su estatus (a mayor estatus mayores necesidades de diferentes recursos). Su forma de aumentar es mediante las migraciones.

Ilustración 2.3: Captura de pantalla del videojuego Foundation.

Por último, Cities: Skylines, Ilustración 2.4 [15], es un juego lanzado en 2015 basado en la época contemporánea realizado en el motor de videojuegos Unity. 9 años después de su salida sigue siendo el juego de construcción de ciudades puro, es decir, sin otros géneros como secundarios, más jugado [16], incluso superando la segunda parte lanzada recientemente.

Tiene como principales sistemas:

- Sistema de construcción: Tiene un sistema de construcción en cuadrícula en el que el jugador selecciona las zonas en la que quiere edificar en función de tres tipos básicos; viviendas, comercios o industria. La propia inteligencia artificial del juego es la encargada de levantar los edificios en función a la demanda de la propia ciudad.
- Economía: Sigue un sistema de economía básico actual, en el que en función del número de ciudadanos y el nivel de sus viviendas se recaudan impuestos que posteriormente el jugador utiliza para ampliar el mapa o mantener los servicios públicos como por ejemplo la educación o el transporte público.
- Agentes: Los agentes son en gran medida decorativos. Su principal función es decorativa, y si bien se mueven desde su vivienda a su lugar de trabajo, lo hacen sin tener en cuenta factores como horarios. La única función útil que el juego utiliza es guardar su nivel de vivienda para asignarle la generación de dinero.

Ilustración 2.4: Captura del juego Cities: Skylines.

2.2. Software utilizado.

En este apartado se realizará un pequeño análisis sobre todo el software a emplear en el desarrollo del prototipo.

En primer lugar, para la elección del motor de videojuegos voy a descartar los motores propietarios como Frostbite (exclusivo de EA) o Decima (Guerrilla Games) ya que no son motores a los que el público general tenga acceso. También voy a descartar motores como Godot, que, si bien es gratuito, está enfocado a juegos 2D, al igual que GameMaker.

Esto nos deja con dos opciones interesantes. Unity y Unreal Engine son los motores más populares en la actualidad. Cada uno de ellos tiene ventajas y desventajas. También es importante saber que ambos motores son gratuitos como indiqué en la introducción para proyectos menores, por lo que el precio a priori no sería un motivo de elección de uno u otro.

Comenzando por Unreal Engine, su principal característica es el apartado visual, con el que puedes llegar a conseguir niveles de detalle y calidad superiores gracias a la potencia gráfica y de renderizado que posee el motor [17]. Otra de sus principales características es el sistema de “Blueprint”, con el que puedes crear el juego sin necesidad de escribir código, siendo distintivo de otros motores y dando la posibilidad a diseñadores o artistas de poder crear juegos.

Como principal desventaja hay que mencionar la elevada curva de aprendizaje inicial, que necesitaría un gran tiempo de adaptación antes de poder comenzar el proyecto. También es necesario comentar que, al tener un apartado visual superior, el equipo necesario para poder ejecutar sus productos es mayor al necesitar un mayor rendimiento.

Unity es un motor multiplaforma que destaca por su facilidad de uso y su amplia comunidad que repercute en un mayor número de recursos, tanto de pago, como principalmente gratuitos para la creación de juegos. Ese gran número de usuarios hace que la cantidad de información en línea (guías,

tutoriales, resolución de dudas, etc.) que un desarrollador puede encontrar sea mucho mayor que para el resto de motores.

Además, Unity es el motor que se utiliza en la asignatura de cuarto año Desarrollo de Videojuegos de la Universidad de Jaén, por lo que los alumnos que hemos cursado dicha asignatura tenemos al menos una base de inicio.

La principal desventaja que Unity posee es el menor rendimiento. Al ser una herramienta multiplataforma, los motores especializados ofrecen un rendimiento mucho mayor, además, el acceso al código base de Unity está restringido solamente a las licencias empresariales de pago [18], por lo que los que quieran aventurarse a indagar para intentar mejorar el rendimiento de algún aspecto en concreto necesitan pagar un elevado coste.

Habiendo visto ambos motores y sus ventajas y desventajas, mi elección para la realización del proyecto será **Unity**, tanto por sus ventajas, ya que se adecua mejor a lo que tengo pensado hacer, como por el conocimiento que poseo previamente del motor gracias a la asignatura de Desarrollo de Videojuegos y de algún pequeño proyecto personal.

Como entorno de programación voy a utilizar **Microsoft Visual Studio** (Ilustración 2.5). Es un programa en el que podemos programar en C#, el lenguaje utilizado para Unity, es gratuito, potente, y Unity recomienda su uso para sus proyectos. Además, es el programa en el que aprendí a programar cuando hice el Ciclo Formativo de Grado Superior de Desarrollo de Aplicaciones Informáticas antes de entrar en la Universidad de Jaén, por lo que tengo gran familiaridad con él.

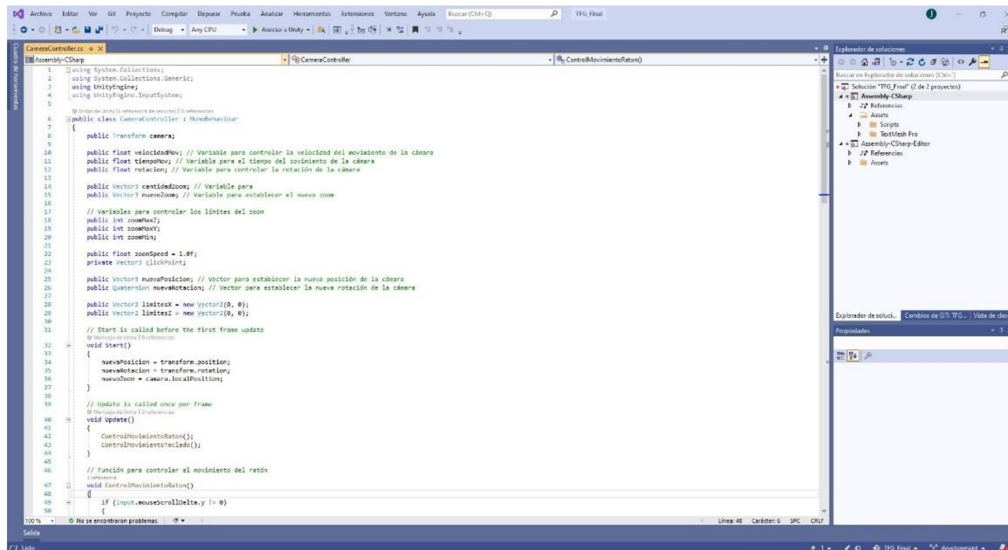


Ilustración 2.5: Captura de pantalla del programa Visual Studio 2019.

Para la creación de la interfaz voy a utilizar el programa de edición de imágenes digitales **GIMP**. Si bien no es tan potente como otros programas de edición de imagen como puede ser Photoshop, es gratuito, de fuentes abiertas y también he trabajo con anterioridad con él.

Los recursos (o *Assets*) van a ser gratuitos, pero en caso de que sea necesario modificar algún detalle, como la referencia al origen, se utilizará también **Blender** (Ilustración 2.6). Blender es otro programa gratuito de fuentes abiertas especializado al modelado, animación y creación de entornos tridimensionales. Es un programa potente con el que se trabaja profesionalmente y en que existen multitud de recursos tanto para su aprendizaje como para encontrar ayuda si fuese necesaria.

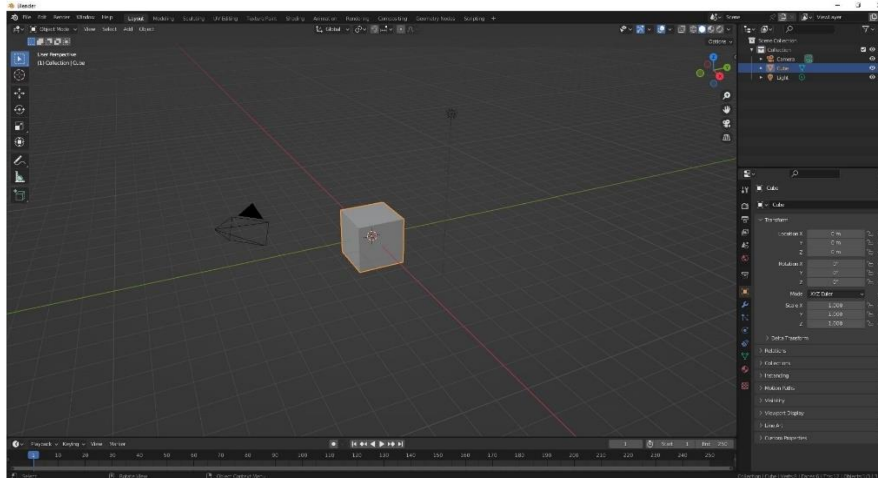


Ilustración 2.6: Captura de pantalla del programa Blender.

2.3. Análisis de requisitos.

Habiendo ya analizado el sector de los videojuegos de construcción de ciudades y decidido las herramientas para llevar a cabo la tarea, me dispongo a darle forma a las principales características del prototipo a realizar.

Primero, por la simplicidad y las posibilidades de almacenar información de los diferentes elementos que compongan el mapa en una estructura de datos con mayor simplicidad, voy a utilizar un sistema de cuadrícula (o "*Grid*") como hemos visto en ejemplos como *Banished* o *Cities: Skylines* (Ilustración 2.7).

Esto también dará mayor sencillez al usuario al construir edificios, ya que la cuadrícula es un sistema más intuitivo que simplemente elegir zonas en el mundo abierto.

Ilustración 2.7: Sistema de cuadrícula del juego Cities: Skylines.

Una de las mecánicas principales también será la recolección de recursos diseminados sobre el mapa. Los agentes serán autónomos en este cometido, por lo que el jugador no tendrá control sobre ellos. Los agentes se encargarán de buscar los recursos en el mapa dependiendo de su trabajo y recolectarlos.

Dicho esto, la economía del juego también dependerá de dichos recursos, siendo posible intercambiarlos por nuevos edificios.

La ambientación será medieval. Al tener recursos por el mapa y basar la economía en recursos en lugar de establecer una moneda, considero que este tipo de ambientación y estética se adecua mejor. También, al ambientarse en una temática medieval, el sistema de recursos puede comenzar siendo más básico, como puede ser por ejemplo obtener madera de un árbol para construir una casa.

Como he escrito anteriormente en el apartado de Estado del arte, en este género no tiene por qué haber condiciones de victoria, por lo que voy a prescindir de ellas. La idea del prototipo es tener un sistema sólido de construcción con la mayor escalabilidad posible, ya que, debido al limitado número de horas de este tipo de proyecto, sería imposible crear sistemas de recolección y procesamiento

complejos, como podemos ver en la Ilustración 2.8 del videojuego Anno 1404 [19].

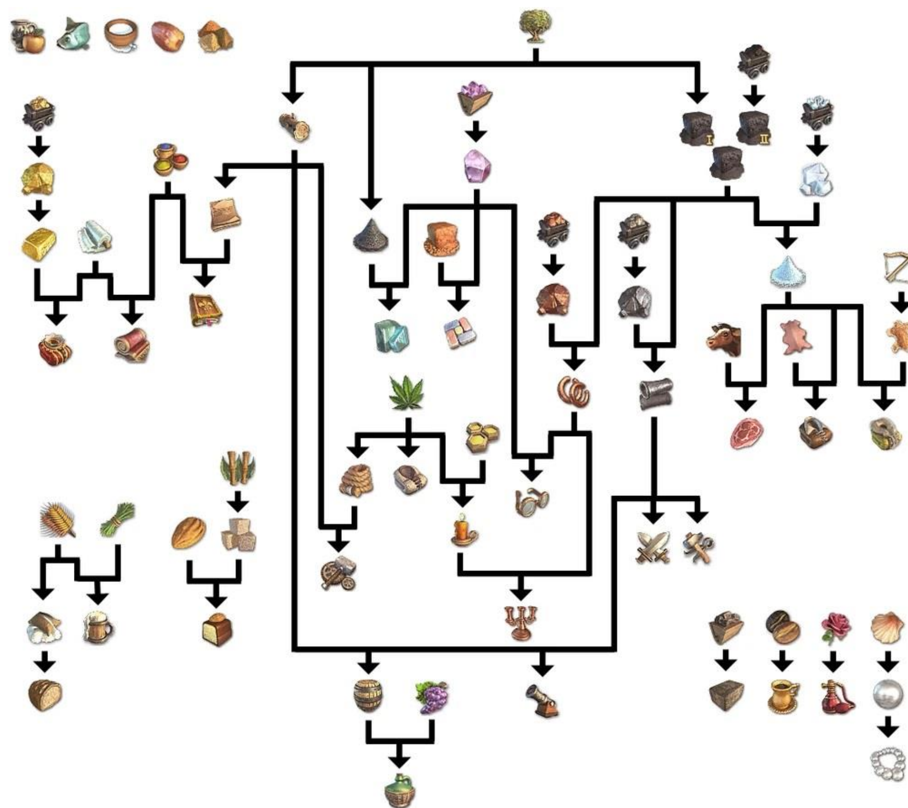


Ilustración 2.8: Sistema de recursos del juego Anno 1404.

2.3.1. Requisitos funcionales.

Los requisitos funcionales son aquellos requisitos que especifican las funciones y características que debe tener el programa para cumplir con las necesidades básicas. Dichas funcionalidades deben de estar completas obligatoriamente en el producto final [20].

Para el prototipo los requisitos funcionales serán los siguientes:

- El prototipo debe permitir al usuario empezar un nuevo mapa o salir.
- El prototipo debe permitir al usuario mover la cámara horizontal y verticalmente.
- El prototipo debe permitir al usuario rotar la cámara.
- El prototipo debe permitir al usuario entrar al modo de construcción.

- El prototipo debe permitir al usuario salir del modo de construcción.
- El prototipo debe permitir al usuario colocar edificios en el mapa.
- El prototipo debe permitir al usuario visualizar la correcta información de los recursos obtenidos.
- El prototipo debe permitir al usuario desactivar la música del juego.
- El prototipo debe permitir al usuario salir de la aplicación.

2.3.2. Requisitos no funcionales.

Los requisitos no funcionales, también llamados requisitos de calidad, son aquellos que describen como debe comportarse el sistema para satisfacer al usuario final [20].

Para el prototipo los requisitos no funcionales serán los siguientes:

- **Escalabilidad:** El prototipo debe tener en cuenta en todo momento la posibilidad de la ampliación de los sistemas ya existentes en la aplicación.
- **Robustez:** El prototipo debe ser capaz de tolerar e impedir los fallos del usuario.
- **Usabilidad:** El prototipo debe de ser intuitivo de utilizar.
- **Portabilidad:** El prototipo debe ser capaz de funcionar en sistemas con hardware modesto.
- **Mantenibilidad:** El prototipo debe ser fácil de modificar y actualizar.

Teniendo ya todos los requisitos analizados e identificados, puedo proceder a explicar la planificación a emplear y posteriormente pasar a la fase del diseño del propio prototipo.

2.4. Planificación.

En este apartado voy a detallar la planificación que deberá seguir el proyecto.

Como indiqué previamente en el punto 1.3. Metodología, voy a utilizar metodología incremental de desarrollo software. Aunque en el desarrollo de esta metodología se contemple la división del análisis y diseño por separado en cada incremento, en este documento voy a exponer estas etapas cada una en conjunto para mayor cohesión, y voy a explicar los incrementos de la fase de “Código” por separado. Además, la realización de esta documentación se hará paralelamente al desarrollo del prototipo.

Así bien, una vez aclarado el punto anterior y en función a la metodología incremental la implementación del prototipo quedará en el siguiente orden:

- Diseño, implementación y prueba del **terreno** sobre el que se apoyará el prototipo.
- Diseño, implementación y prueba del **sistema de cuadrícula** (o *Grid*).
- Diseño, implementación y prueba del **sistema de cámara**.
- Diseño, implementación y prueba del **sistema de construcción**.
- Diseño, implementación y prueba del **sistema de agentes**.
- Diseño, implementación y prueba de la **creación de mundo** (distribución de recursos por el mapa) **y economía**.
- Diseño, implementación y prueba de la **interfaz** del prototipo.
- Etapa final de ajustes de balanceo.

Una vez terminada cada incremento del prototipo, el desarrollador probará cada aspecto creado para comprobar su correcto funcionamiento. La etapa final de ajustes de balance será una excepción, ya que será un usuario externo el que la probará para así obtener una retroalimentación y una doble comprobación sobre los sistemas empleados y posibles fallos pasados por alto en la versión final del prototipo que el desarrollador haya podido pasar por alto.

2.4.1. Estimación de tiempo

En este apartado estimaré una aproximación del esfuerzo en tiempo que supondrá la realización del Proyecto de Fin de Grado. Voy a realizar una tabla para tener todos los datos organizados. Dentro de dicha tabla voy a desglosar el tiempo empleado en sus subapartados correspondientes por incremento, análisis, diseño e implementación, salvo el último, ya que al estar dedicado a pruebas y ajustes carece de los dos primeros. El resultado se mostrará en la tabla 2.1.

Finalmente, también voy a incluir un diagrama de Gantt para reflejar dicho esfuerzo a lo largo del tiempo.

<table>
<tr><th>Actividad</th><th colspan="2">Tiempo estimado</th></tr>
<tr><td>Redacción de la documentación</td><td colspan="2">72 horas</td></tr>
<tr><td>Análisis preliminar</td><td colspan="2">40 horas</td></tr>
<tr><td colspan="2">Incremento 1: Terreno</td><td rowspan="4">6 horas</td></tr>
<tr><td>Análisis</td><td>2 horas</td></tr>
<tr><td>Diseño</td><td>2 horas</td></tr>
<tr><td>Implementación</td><td>2 horas</td></tr>
<tr><td colspan="2">Incremento 2: Sistema de cuadrícula</td><td rowspan="2">48 horas</td></tr>
<tr><td>Análisis</td><td>12 horas</td></tr>
</table>

Diseño	8 horas	
Implementación	28 horas	
Incremento 3: Sistema de cámara		20 horas
Análisis	4 horas	
Diseño	4 horas	
Implementación	12 horas	
Incremento 4: Sistema de construcción		64 horas
Análisis	12 horas	
Diseño	16 horas	
Implementación	36 horas	
Incremento 5: Sistema de agentes		60 horas
Análisis	4 horas	
Diseño	8 horas	
Implementación	48 horas	
Incremento 6: Creación mundo y economía		32 horas
Análisis	4 horas	

Diseño	4 horas	
Implementación	24 horas	
Incremento 7: Interfaz		32 horas
Análisis	8 horas	
Diseño	16 horas	
Implementación	8 horas	
Incremento 8: Ajustes finales		16 horas
ESFUERZO TOTAL	**390 horas**	

Tabla 2.1: Tiempo estimado total del proyecto.

2.4.2. Diagrama de Gantt.

En este apartado podremos ver el diagrama de Gantt obtenido de la estimación anterior. En él se detallan tanto los apartados vistos anteriormente como los que no han necesitado un desglose previo. Para su mejor visualización se ha necesitado dividirlo en varias partes, las ilustraciones 2.9, 2.10, 2.11, 2.12 y 2.13. Al final del apartado habrá un enlace con el acceso a las imágenes en formato PNG para poder observarlo con mayor detalle.

Las partes de análisis están representadas en verde, las de diseño, en rojo, y las de desarrollo, en azul. La redacción de la documentación aparece en la parte inferior del diagrama, representada con un color turquesa.

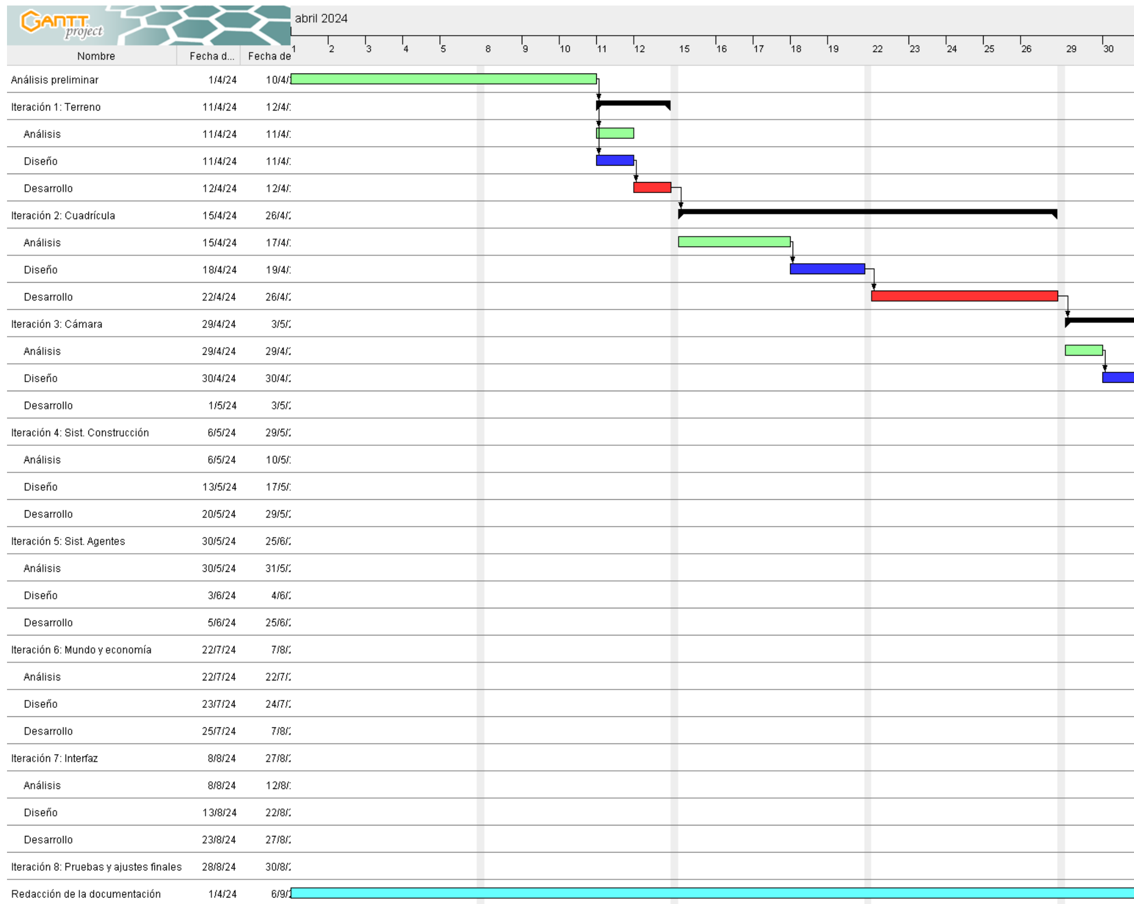


Ilustración 2.9: Diagrama de Gantt, parte 1.

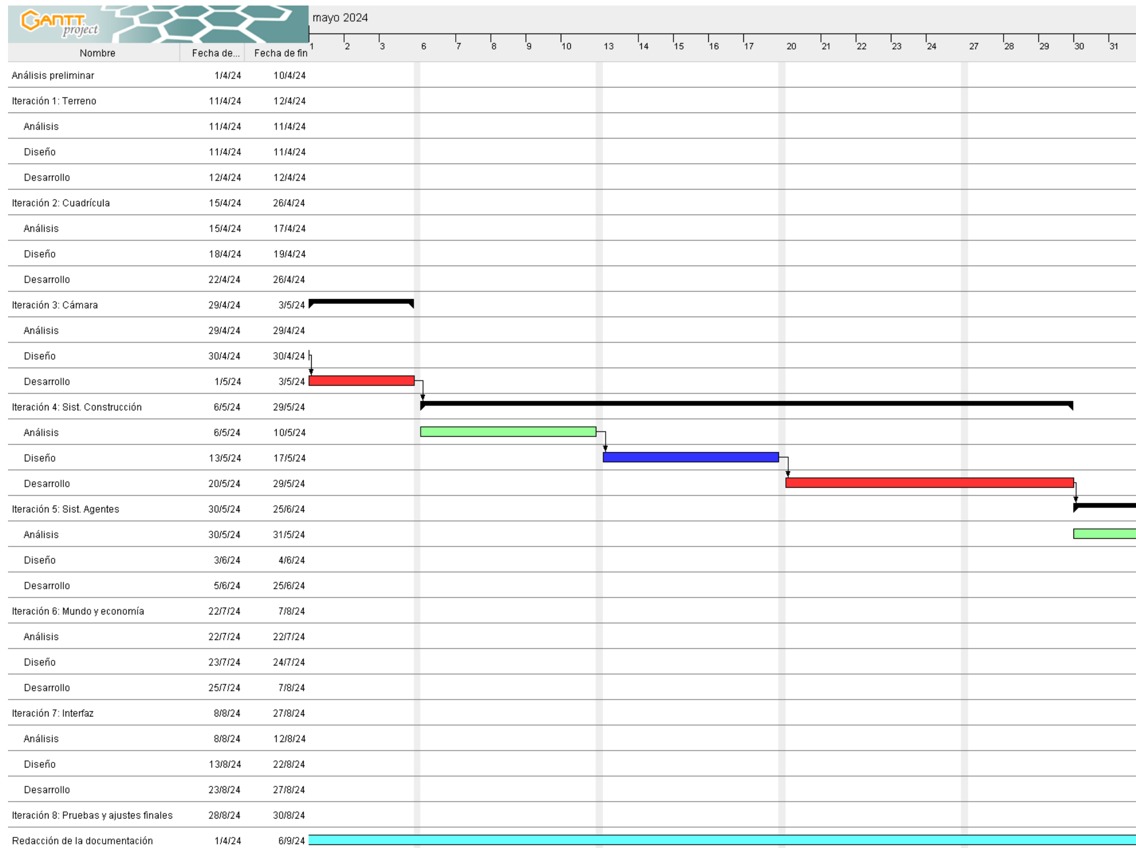
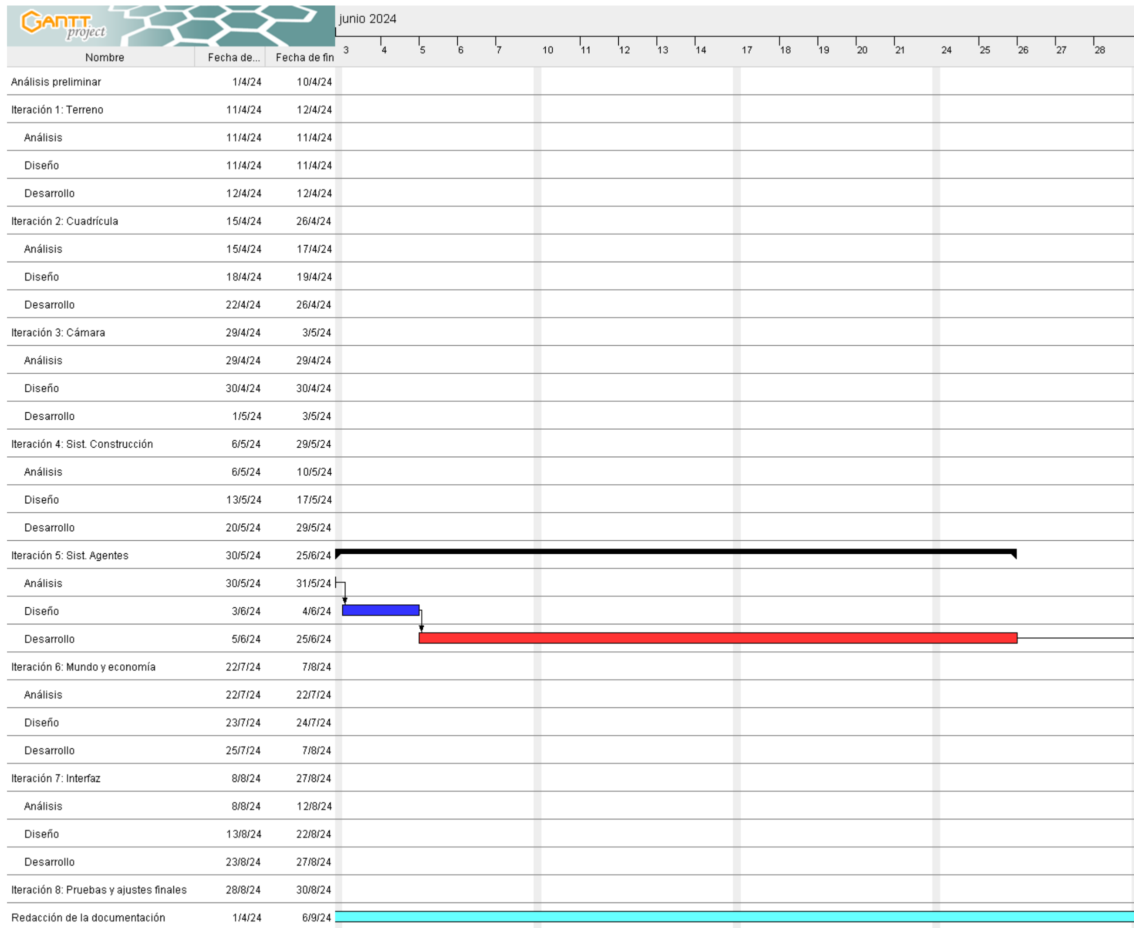


Ilustración 2.10: Diagrama de Gantt, parte 2.



Nombre	Fecha de...	Fecha de fin
Análisis preliminar	1/4/24	10/4/24
Iteración 1: Terreno	11/4/24	12/4/24
Análisis	11/4/24	11/4/24
Diseño	11/4/24	11/4/24
Desarrollo	12/4/24	12/4/24
Iteración 2: Cuadrícula	15/4/24	26/4/24
Análisis	15/4/24	17/4/24
Diseño	18/4/24	19/4/24
Desarrollo	22/4/24	26/4/24
Iteración 3: Cámara	29/4/24	3/5/24
Análisis	29/4/24	29/4/24
Diseño	30/4/24	30/4/24
Desarrollo	1/5/24	3/5/24
Iteración 4: Sist. Construcción	6/5/24	29/5/24
Análisis	6/5/24	10/5/24
Diseño	13/5/24	17/5/24
Desarrollo	20/5/24	29/5/24
Iteración 5: Sist. Agentes	30/5/24	25/6/24
Análisis	30/5/24	31/5/24
Diseño	3/6/24	4/6/24
Desarrollo	5/6/24	25/6/24
Iteración 6: Mundo y economía	22/7/24	7/8/24
Análisis	22/7/24	22/7/24
Diseño	23/7/24	24/7/24
Desarrollo	25/7/24	7/8/24
Iteración 7: Interfaz	8/8/24	27/8/24
Análisis	8/8/24	12/8/24
Diseño	13/8/24	22/8/24
Desarrollo	23/8/24	27/8/24
Iteración 8: Pruebas y ajustes finales	28/8/24	30/8/24
Redacción de la documentación	1/4/24	6/9/24

Ilustración 2.11: Diagrama de Gantt, parte 3.

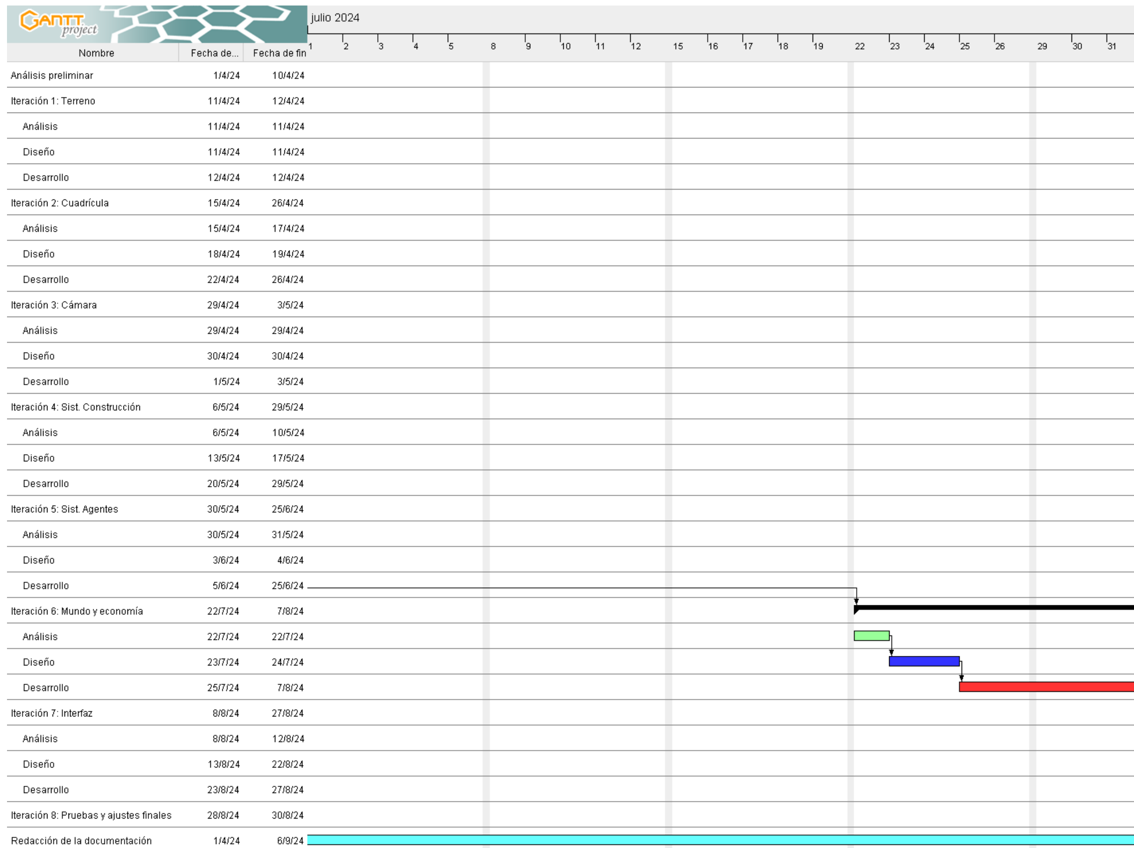


Ilustración 2.12: Diagrama de Gantt, parte 4.

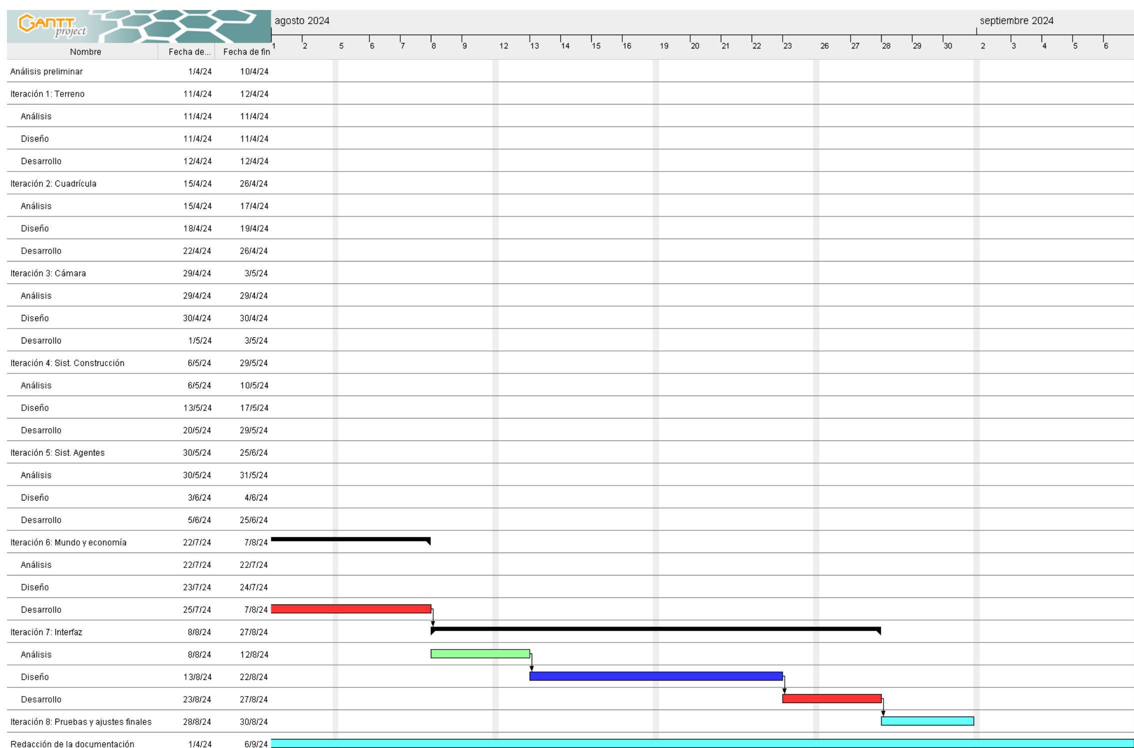


Ilustración 2.13: Diagrama de Gantt, parte 5.

En el mes de julio existe un parón planeado con antelación en la actividad debido a un viaje.

Enlace a los diagramas en formato PNG para su mejor visualización:

https://drive.google.com/drive/folders/1czihgcjHeoxeLQwABtT4bM641rVg3R1n?usp=drive_link

2.5. Estimación de costes.

En este apartado estimaré los costes asociados a la realización del Proyecto de Fin de Grado. Dividiré el apartado en los recursos utilizados que son; los costes asociados al software, los asociados al hardware empleado y los costes asociados al personal.

2.5.1. Costes asociados al software.

Al ser un proyecto universitario he utilizado programas gratuitos en su totalidad, por lo que el coste asociado es 0€ como podemos ver en la tabla 2.2.

Programa	Coste
Unity	0€
Visual Studio 2019	0€
GIMP	0€
Blender	0€
COSTE TOTAL	**0€**

Tabla 2.2: Estimación de costes del software.

2.5.2. Costes asociados al hardware.

Para los costes asociados al hardware se utilizarán los equipos necesarios utilizados durante la consecución del proyecto. En este caso se componen de dos componentes, el ordenador personal y la tableta digitalizadora del desarrollador del proyecto.

Hay que tener en cuenta que para estimar el coste en este apartado no se contempla el coste de adquisición del hardware empleado, ya que es utilizado

para más proyectos. En su lugar se utiliza el coste de amortización definido por los días de utilización de cada equipo.

La amortización se realiza en función de la vida útil del equipo, en el caso del ordenador personal está en torno a los 5 años de duración [21], mientras que en el caso de la tableta digitalizadora esta se extiende hasta los 10 años [22].

Con los datos anteriores y teniendo en cuenta de que la duración de la realización del proyecto es de aproximadamente 5 meses. Lo que nos lleva a deducir que la utilización del ordenador personal durante ese tiempo será constante, mientras que la utilización de la tableta será ocasional ya que solo se utilizará en el diseño de la interfaz. Con esto podemos realizar la estimación definida en la tabla 2.3.

Hardware	Días utilizado	€/día	Coste
Ordenador personal	150	0,90€	135€
Tableta digitalizadora	5	0,03€	0,15€
COSTE TOTAL			**135,15€**

Tabla 2.3: Estimación de costes del hardware.

2.5.3. Costes asociados al personal

Para los costes asociados al personal se contempla un equipo básico de desarrollo, aunque al ser un Proyecto de Fin de Grado ha sido realizado por un único alumno. Por lo que el alumno tomará todos los roles disponibles del citado equipo.

Hay que tener en cuenta que, si bien los siguientes roles trabajarán el proyecto, no lo harán a tiempo completo en él ya que hay que asumir que ciertos roles en las empresas trabajan en más de un proyecto a la vez.

Estos roles son:

- **Jefe de proyecto**: Es el encargado de supervisar todo el desarrollo del proyecto, además de encargarse de la realización de la documentación correspondiente.
- **Diseñador**: Encargado de analizar y diseñar el prototipo de videojuego.
- **Diseñador gráfico**: Es el encargado de realizar todo el trabajo visual, tanto de interfaz como de modificación de recursos del prototipo (o *Assets*).
- **Programador**: Es el encargado de llevar a cabo el desarrollo e implementación de la visión del diseñador.
- **QA Tester**: Es el encargado de comprobar y analizar la calidad del producto, es decir, el correcto funcionamiento del prototipo.

Por tanto, teniendo en cuenta los diferentes roles con su tiempo asociado y que la duración del proyecto es de unos 5 meses nos quedarían los datos de la tabla 2.4, donde obtenemos el coste del empleado obteniendo el porcentaje de dedicación al proyecto con respecto a su sueldo mensual, multiplicado por los 5 meses de duración del proyecto.

Hay que agregar también que, a la fecha de realización del proyecto, 2024, el salario mínimo bruto anual es de 15.876€ [23], y que los sueldos por especialidad vienen dados por las medias de ofertas reales de la web Tecnoempleo en la actualidad [24]. Por simplicidad voy a considerar 12 pagas anuales.

Rol	Sueldo bruto mensual	Tiempo dedicado al proyecto	Coste
Jefe de proyecto	3.650€	15%	2.737,50€
Diseñador	2.633€	25%	3.291,25€
Diseñador gráfico	1.323€	10%	661,50€
Programador	2.000€	70%	7.000,00€
QA tester	1.323€	5%	330,75€
COSTE TOTAL			**14.021,00€**

Tabla 2.4: Estimación de costes del personal.

2.5.4. Coste total.

Finalmente, para obtener el coste total del desarrollo del proyecto es necesario calcular también los costes indirectos del mismo.

Este tipo de gastos son los que engloban los gastos adicionales como pueden ser alquiler, electricidad, internet etc. Se supondrán un 10%. También se va a incluir un 10% como margen de maniobra para reducir riesgos y otro 10% asignado a los beneficios de la empresa.

Por tanto, estos costes ascienden al 30% del total.

Como nota, todos los costes calculadores incluyen el porcentaje de I.V.A. correspondiente.

Una vez habiendo analizado y desglosado los costes de cada apartado individualmente, junto con los costes indirectos, estamos en posición de poder calcular el coste total de desarrollo del Proyecto de Fin de Grado.

La tabla 2.5 muestra un resumen del coste total del proyecto:

Costes asociados	Coste
Software	0,00€
Hardware	135,15€
Personal	14.021,00€
COSTE BASE	14.156,15€
Gastos indirectos (10%)	1.415,62€
Margen de maniobra (10%)	1.415,62€
Beneficios (10%)	1.415,62€
COSTE TOTAL	**18.403,01€**

Tabla 2.5: Resumen del coste total del proyecto.

3. Diseño

En este apartado se explicarán las decisiones que he tomado para el diseño de las diferentes iteraciones del prototipo.

Primero, para la mejor visualización de mis ideas utilizaré un diagrama de casos de uso para la visión general de la aplicación. También incluiré un diagrama de componentes para mostrar los diferentes elementos que componen el prototipo y la relación entre ellos. Finalmente explicaré el planteamiento del diseño de los incrementos pensados anteriormente utilizando diagramas de actividades para los sistemas que lo requieran.

3.1. Diagrama de casos de uso principal.

Habiendo analizado todo lo requerido para la creación del prototipo he creado un diagrama de casos de uso representado en la ilustración 3.1. En él podemos ver las acciones que el jugador podrá realizar, que serán iniciar o salir del juego, mover la cámara por el mapa y la posibilidad de construir edificios siempre que las condiciones lo permitan.

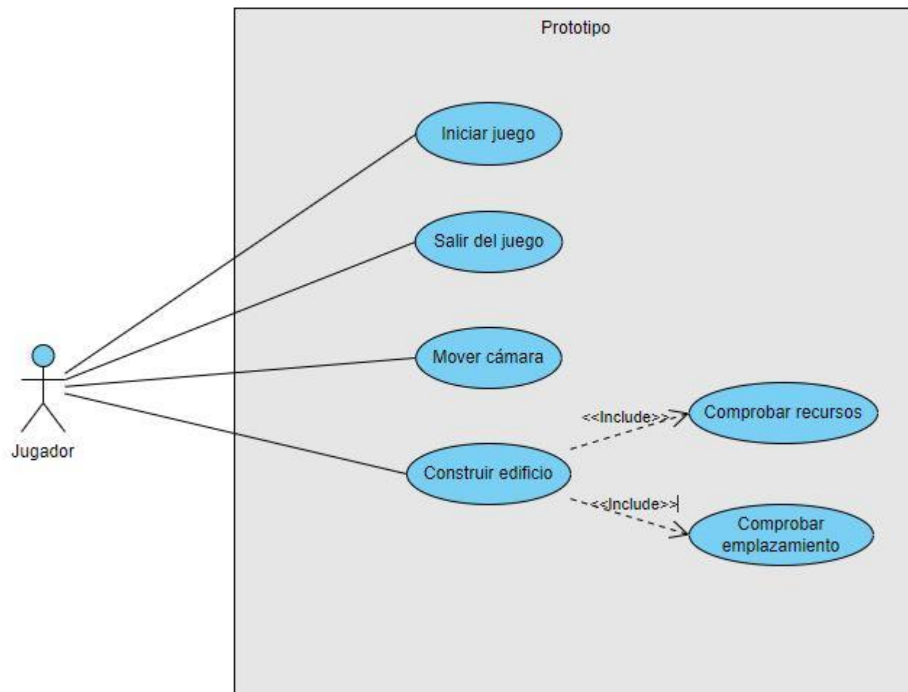


Ilustración 3.1: Diagrama de casos de uso del prototipo.

3.2. Diagrama de componentes

Para detallar el diseño de prototipos he creado un diagrama de componentes en el que se detallan los diferentes sistemas y controladores que tendrá el sistema del prototipo, así como las relaciones entre ellos. El diagrama lo podremos ver en la ilustración 3.2.

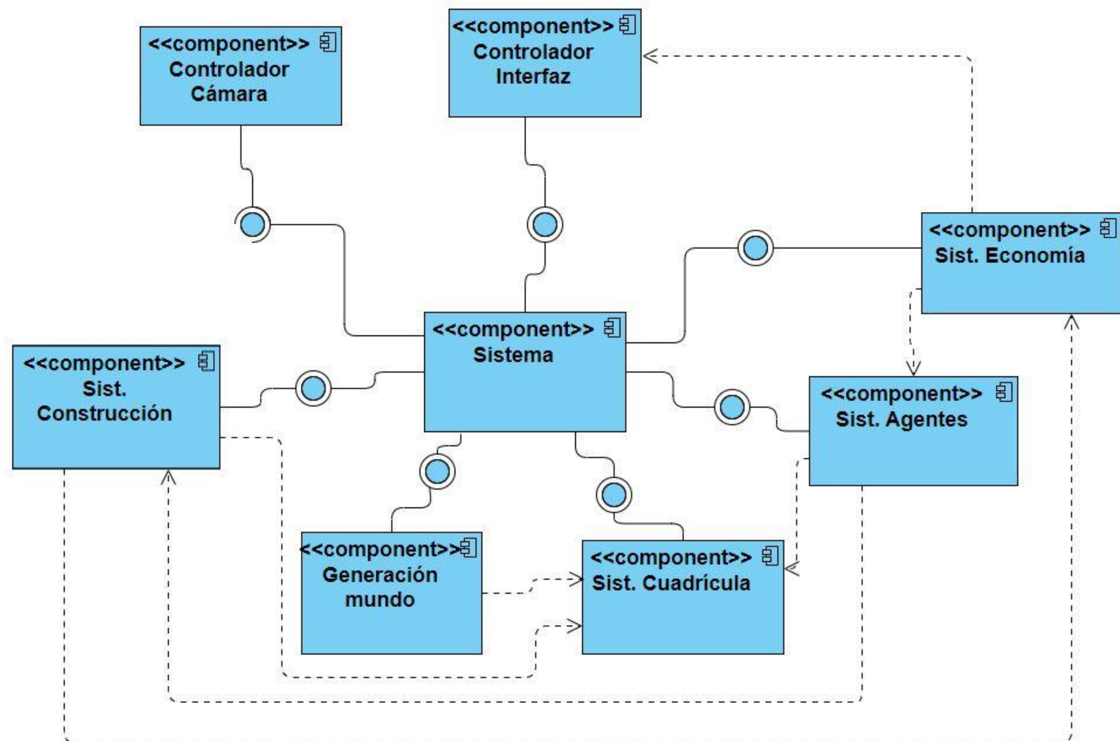


Ilustración 3.2: Diagrama de componentes del sistema.

3.3. Primer incremento: Terreno.

Para el diseño del terreno en el que se va a asentar el prototipo de videojuego he elegido un mapa plano con una dimensión suficiente para la colocación de multitud de recursos y edificios.

Esto se debe a que, al no tener condiciones de victoria, hacer un mapa de pequeñas dimensiones resultaría en partidas excepcionalmente cortas teniendo en cuenta que en juegos de este tipo los mapas pueden superar con holgura las 10 horas de duración.

La elección de que todo el terreno sea plano es para evitar problemas derivados a las pendientes y la construcción, ya que los desniveles van a proporcionar retos superiores al tiempo del que dispongo. Alrededor del mapa, es decir, en los límites, voy a modelar el propio terreno para crear montañas o algunos accidentes geográficos que den sensación de finitud.

3.4. Segundo incremento: Sistema de cuadrícula.

Para el diseño del sistema de cuadrícula voy a utilizar una malla rectangular uniforme. Esta va a estar asentada en el terreno previamente creado y servirá de nexo de unión entre el apartado visual y la lógica del prototipo. Su función, por tanto, no es meramente estética.

Visualmente se mostrará solamente cuando el jugador quiera colocar algún edificio, en un modo de “vista de construcción”, del que se podrá salir para volver a visualizar el mapa normalmente.

En el apartado lógico, voy a permitir la ocupación de un solo elemento, sea recurso o edificio en cada casilla. Al ser un rectángulo uniforme puedo utilizarlo a mi favor y considerar la posición de dichas casillas como localizador único con el formato (X, Y, Z), al ser un entorno en tres dimensiones (aunque al trabajar sobre el mismo plano la Y siempre será la misma). Se puede ver un boceto ilustrativo de la cuadrícula en la ilustración 3.3.

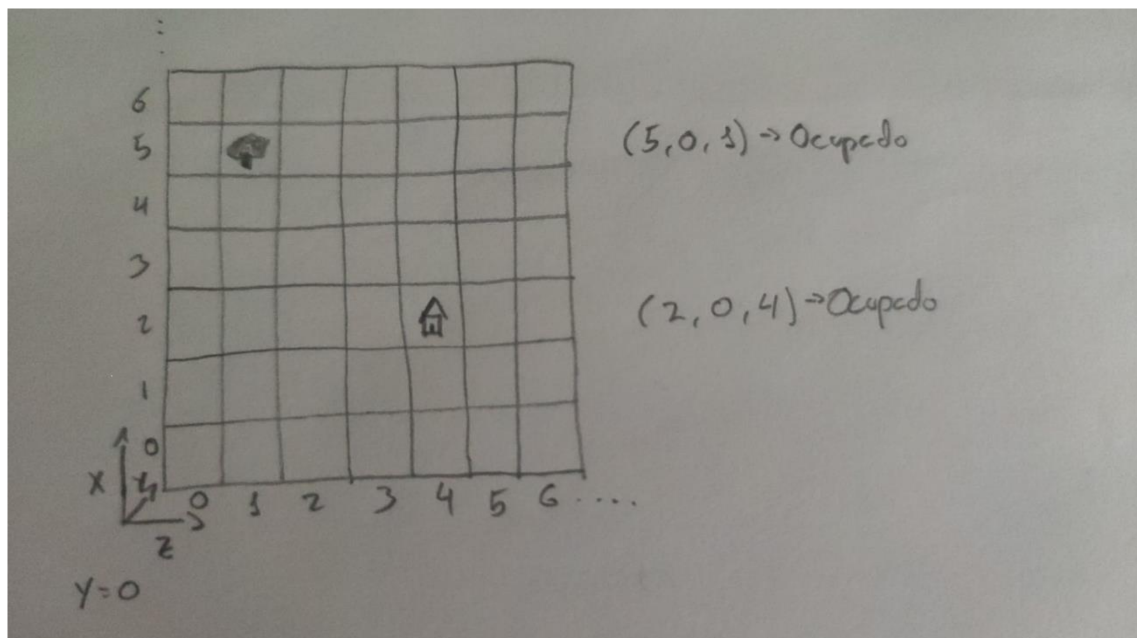


Ilustración 3.3: Boceto del sistema de cuadrícula.

En esta iteración también veo necesario el diseño de una base de datos propia de Unity, ya que va a ser necesaria. Esta pequeña base de datos almacenará los recursos y será muy simple, ya que solo se utilizará para

almacenar unos pocos elementos al ser la lógica posterior en la implementación la encargada de almacenarlo todo.

Estará compuesta por los siguientes campos:

- ID: Un identificador único.
- Nombre: El nombre del objeto a almacenar.
- Tamaño X e Y: Tamaño del objeto que ocupará en el sistema de casillas.
- Prefab: Asset del objeto a almacenar.
- Valores: Valores de los recursos del objeto.

3.5. Tercer incremento: Sistema de cámara.

Para el diseño del sistema de cámaras que el prototipo va a utilizar he pensado que la mejor opción sea la utilización de dos cámaras, una principal que mostrará la vista del juego, y otra secundaria que cumplirá una función de mini mapa.

La cámara principal será una cámara isométrica. Esta cámara ofrece una perspectiva inclinada con unos ángulos que pueden variar generalmente entre 30º a 45º consiguiendo mostrar en pantalla una gran cantidad de información del mapa. La cámara isométrica también tiene como característica que mantiene una posición fija y el jugador, al moverla, consigue el efecto de moverse alrededor del mapa.

En la ilustración 3.4 podemos observar mejor la vista aproximada de una cámara de este tipo que el conocido juego Age of Empires II utiliza.

Ilustración 3.4: Sistema de cámara del videojuego Age of Empires II.

Para el mini mapa, voy a utilizar una cámara cenital. Este tipo de cámara, al tener una perspectiva lejana y perpendicular al plano del mapa, dará información del mundo y los elementos que lo componen de una forma más esquemática. Al ser también un recurso muy utilizado en juegos de este tipo considero idóneo su creación.

3.6. Cuarto incremento: Sistema de construcción.

Para el diseño del sistema de construcción voy a utilizar dos tipos de edificios con distintas dimensiones cada uno. Esto me permitirá probar la robustez del sistema y la interconexión con la cuadrícula al comprobar que independientemente del tamaño de edificio todo sea funcional.

Un edificio será para leñadores y otro para mineros.

Cuando el jugador seleccione un edificio a construir, se pasará a un modo de “vista de construcción” en el que se podrá ver la cuadrícula, como antes mencioné. Esto facilitará la colocación del edificio al dar una mayor potencia visual al proceso.

Al seleccionar el edificio a construir e intentar colocarlo en el mapa, el sistema primero deberá de comprobar si la casilla (o casillas, en el caso de tener un tamaño superior a una) está ocupada. Si ésta está ocupada no dejará construirlo, mientras que si está disponible debe pasar a la siguiente fase, que es comprobar la disponibilidad de recursos necesarios. Si estos requisitos se cumplen, se procederá a colocar el edificio en el mapa.

En la ilustración 3.5 podemos ver el comportamiento del sistema de construcción mediante un diagrama de actividades.

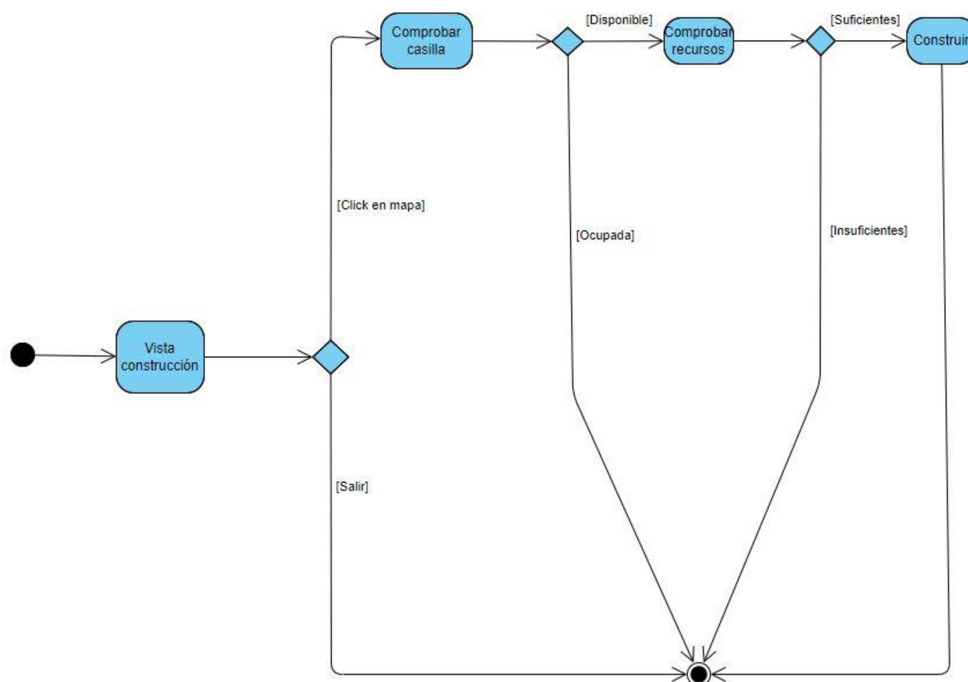


Ilustración 3.5: Diagrama de actividades del sistema de construcción.

Este sistema va a ser el nexo de unión entre el sistema anterior y el siguiente, de agentes, ya que pretendo que, en fase posterior del desarrollo del sistema de agentes, estos se creen justo al lado del edificio construido.

3.7. Quinto incremento: Sistema de agentes.

Para el diseño del sistema de agentes, como previamente he comentado, voy a generar un agente por cada edificio que coloque en el mapa.

El agente creado será autónomo y tendrá un “trabajo” asociado al edificio construido.

Su ciclo de trabajo consistirá en buscar el recurso asociado más cercano, moverse hacia dicho recurso y comenzar a recolectar hasta agotarlo. Una vez hecho esto volverá al comienzo del ciclo. Dicho ciclo será repetido hasta que no queden más recursos que recolectar en el mapa, momento en el que el agente finalizará su labor y quedará en estado de espera.

Solo se permitirá que un agente trabaje en un recurso a la vez, si todos están ocupados también quedará en estado de espera.

Para visualizar mejor el comportamiento de los agentes, he creado un diagrama de actividad que podemos observar en la ilustración 3.6.

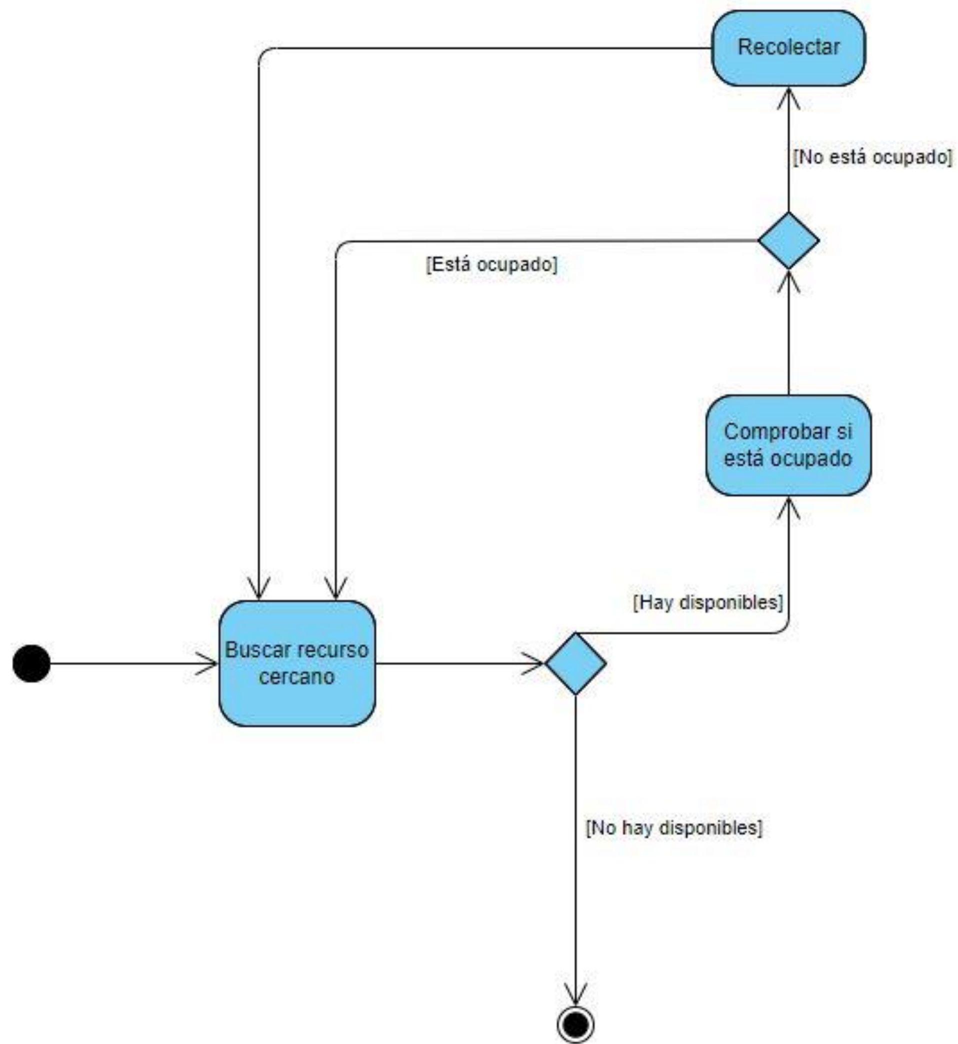


Ilustración 3.6: Diagrama de actividad del sistema de agentes.

3.8. Sexto incremento: Creación de mundo y economía.

Para el diseño del mundo voy a añadir dos tipos de recursos diferentes, madera y piedra. Con ellos vamos a poder construir los edificios necesarios para crear nuevos agentes y recolectar más recursos. Como indiqué previamente, al estar ambientado en un entorno medieval he elegido los recursos más básicos posibles. Además, al venir de elementos muy comunes como son los árboles y

las rocas, debería haber multitud de recursos disponibles en internet para poder representarlo todo gratuitamente.

He considerado que la mejor opción para distribuirlos por el mundo es la de asignarlos aleatoriamente mediante una función. Al tener un sistema de cuadrícula será más sencillo recorrer el mapa colocando un recurso por casilla. Hacerlo manualmente supondría un esfuerzo en el tiempo demasiado elevado, además, al tener solo un nivel, el mapa sería siempre el mismo, lo que supondría demasiada repetitividad.

En lo que respecta a la economía, el prototipo se basará en esos dos recursos, piedra y madera. Teniendo esos dos recursos me aseguro que los agentes puedan recolectar más de uno, y los edificios necesiten varios para construirse también, por lo que me aseguro la escalabilidad como propiedad para un futuro.

Los recursos estarán disponibles en árboles y rocas que los agentes deberán recolectar, y serán utilizados para la construcción de los edificios. Cada edificio tendrá un coste diferente.

3.9. Séptimo incremento: Interfaz.

El último incremento a tratar en el diseño es la interfaz. El diseño estará dividido en dos partes. El primero será la parte del menú inicial del juego, el segundo la interfaz propia del juego.

La interfaz del menú principal consistirá en dos botones, sus funciones serán entrar al propio mapa del juego o salir de la aplicación. También tendrá un botón para activar o desactivar el sonido de una canción en el menú.

Al ser una parte puramente visual es más sencillo mostrarlo con el boceto mostrado en la ilustración 3.7.

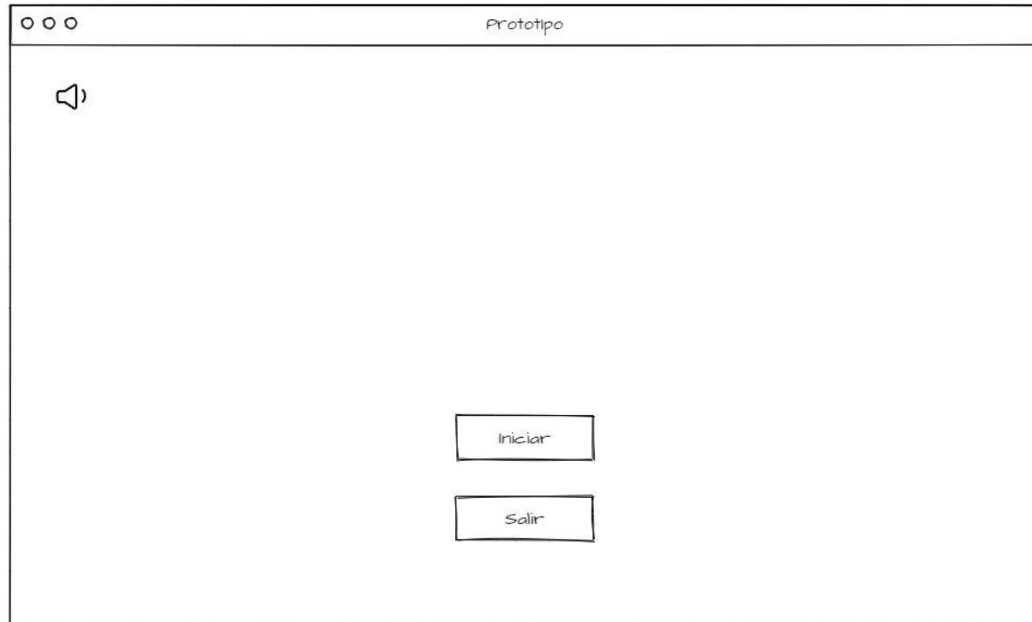


Ilustración 3.7: Diseño de la interfaz del menú principal.

La interfaz del propio juego estará dividida en varios componentes.

- En la parte superior central mostrará la información de los recursos y agentes presentes en el juego.
- En la parte superior derecha mostrará el mini mapa.
- En la parte inferior mostrará dos botones para representar sendos edificios para su construcción.

En la ilustración 3.8 podremos ver la representación de dicha interfaz.

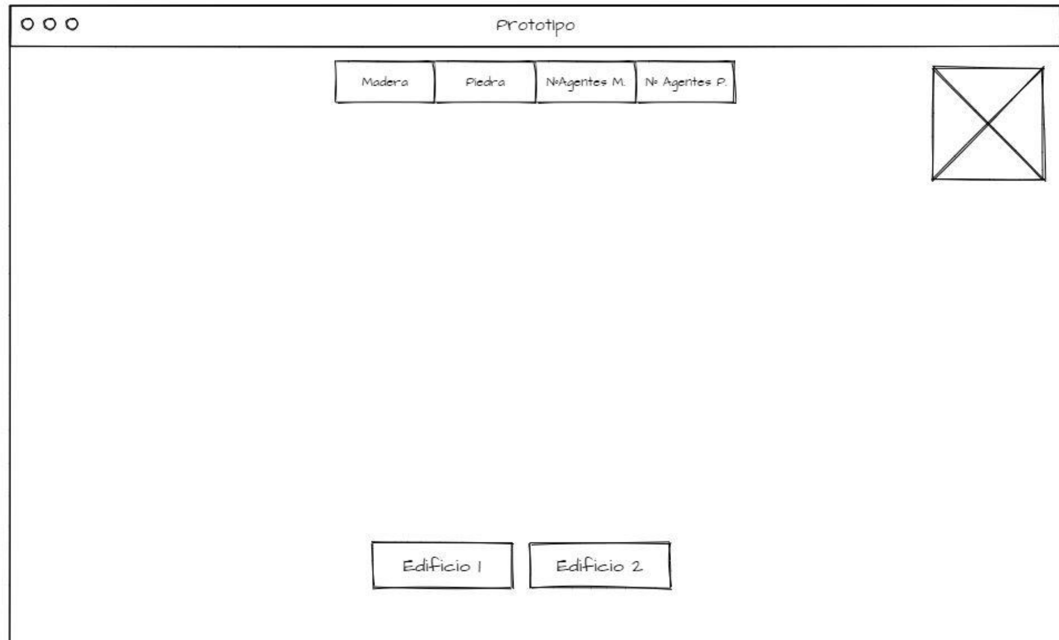


Ilustración 3.8: Diseño de la interfaz del juego.

4. Desarrollo.

En este apartado se explicará todo el proceso de desarrollo del prototipo siguiendo la metodología incremental definida anteriormente, en el apartado de Análisis. También, como vimos en la subsección de estimación de tiempo del mismo apartado, algunos incrementos serán más ligeros que otros, ya que elementos como el sistema de agentes, por ejemplo, van a suponer una carga de trabajo muy elevada.

Esta sección, al igual que en el apartado anterior de Diseño, estará dividida en los apartados de los incrementos planificados, estos son:

- Primer incremento: Terreno.
- Segundo incremento: Sistema de cuadrícula.
- Tercer incremento: Sistema de cámara.
- Cuarto incremento: Sistema de construcción.
- Quinto incremento: Sistema de agentes.
- Sexto incremento: Creación mundo y economía.
- Séptimo incremento: Interfaz.
- Octavo incremento: Ajustes finales y balanceo.

Para el desarrollo de la aplicación vamos a utilizar con frecuencia el patrón de diseño creacional Singleton. Este patrón va a ser central en la consecución final de la aplicación, ya que asegura que la clase solo tenga una única instancia y proporciona un acceso global a la misma [25]. La utilización del patrón Singleton tiene las siguientes ventajas:

- **Acceso global**: Puedes acceder a la instancia de una clase Singleton desde cualquier parte de tu código, lo que facilita la comunicación y el uso compartido de datos y funcionalidades importantes.
- **Control de la instancia**: El patrón Singleton garantiza que solo haya una única instancia de una clase en todo momento, lo que evita problemas de duplicación o conflictos de datos.

- **Eficiencia**: Al utilizar una única instancia en lugar de crear múltiples instancias de una clase, puedes ahorrar recursos y mejorar el rendimiento de tu juego.

4.1. Primer incremento: terreno.

Al ser la primera iteración del proyecto con el motor Unity, lo primero ha sido crear las bases comunes que suelen tener la mayoría de juegos.

Primero, he creado un nuevo proyecto 3D. El proyecto contará con un menú principal que dará paso al juego, por lo que primero he renombrado la escena existente a “Main Menu”. Seguidamente he añadido una nueva escena al proyecto, la cual he denominado “Main”, al ser la principal en la que se desarrollará el juego.

Después, he activado la librería de Unity “TextMeshPro”, ya que la conozco de trabajos previos y da buenos resultados. Esta librería ofrece una mejora a los elementos de texto la interfaz de Unity (los llamados “label”), mejorándolos visualmente al añadir una mejora del renderizado sin perder rendimiento. También permite modificar en mayor profundidad el texto creado, dando una mayor funcionalidad al formato del texto. Por último, es ampliamente compatible con multitud de fuentes y sprites, por lo que, al tenerla ya añadida al proyecto, este queda mejor preparado a ampliaciones futuras.

También he creado la estructura de carpetas necesarias para la organización del proyecto, como por ejemplo una carpeta para los scripts, otra para las imágenes etc.

Seguidamente, he añadido un objeto vacío a la escena llamado “GameManager”, como nos enseñaron en la asignatura de Desarrollo de Videojuegos. Este elemento es fundamental y sirve para organizar y gestionar varios aspectos del juego.

El “GameManager” actúa como el núcleo central del proyecto, centralizando buena parte de la lógica posterior que tendrá el juego. Esto no solo facilita la gestión de diferentes componentes y sistemas, sino que también mejora la claridad y la estructura del código, haciendo que el desarrollo sea más eficiente y menos propenso a errores, ayudando así a uno de los requisitos no funcionales más importantes, como la robustez.

Además, teniendo en cuenta un principio tan crucial como la escalabilidad, contar con un elemento como el “GameManager” es extremadamente útil. Permite añadir nuevas funcionalidades de manera ordenada y sin desorganizar toda la estructura del proyecto. Esto asegura que el juego pueda crecer y evolucionar con el tiempo, incorporando nuevas características y mejoras sin comprometer la estabilidad y la coherencia del código existente.

Finalmente, en el “GameManager”, al ir enlazando los diferentes scripts asociados, permite tener una mayor usabilidad y mantenibilidad, ya que los parámetros a introducir para el funcionamiento del prototipo serán fácilmente localizables, lo que implica una mayor facilidad de uso.

Una vez hechos los pasos previos pasamos a la creación del terreno en sí. Para ello, hay que añadir un nuevo objeto 3D llamado “Terrain”. Este elemento servirá como base al mapeado al tener un collider incorporado para que elementos del juego como los agentes tengan una base.

Con el terreno ya creado, aunque plano, el siguiente paso es darle una textura y color apropiados. Para ello he utilizado un paquete de texturas gratuitas de la AssetStore de Unity [26] que simula hierba.

Finalmente, un plano era poco interesante para el proyecto, así que he creado bordes montañosos para delimitar los límites del terreno, haciendo una especie de valle en el centro que es donde se desarrollará el juego. Dichas montañas están creadas con la propia herramienta que Unity ofrece para modificar los terrenos como podemos ver en la ilustración 4.1. Con ella puedes desde pintar la superficie, o como en este caso, aumentar o disminuir el nivel del terreno haciendo uso de un pincel para ello.

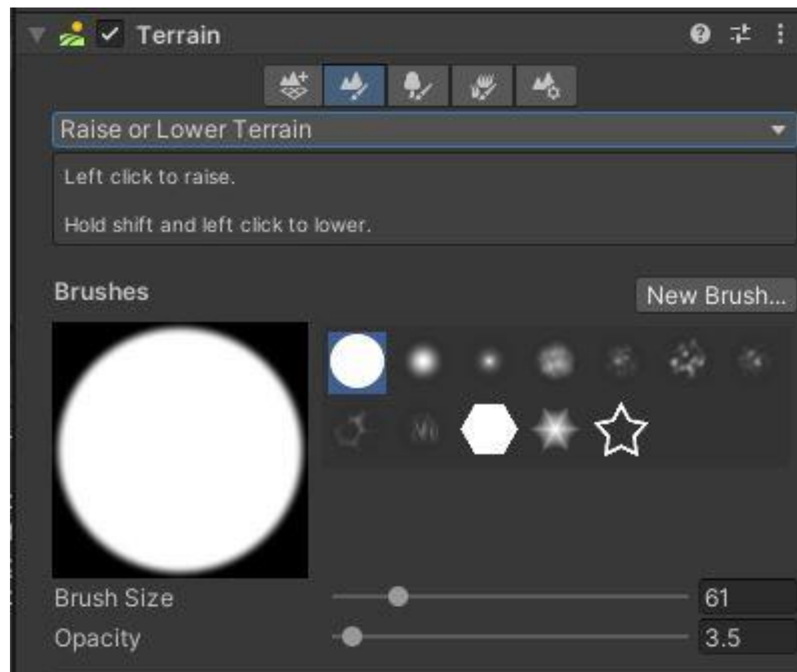


Ilustración 4.1: Herramienta de Unity para modificar el terreno.

Estas montañas estarán estrechamente ligadas con la posterior iteración del sistema de cámara, ya que deberán interactuar para establecer los límites.

El resultado final del terreno creado lo podemos ver en la ilustración 4.2.

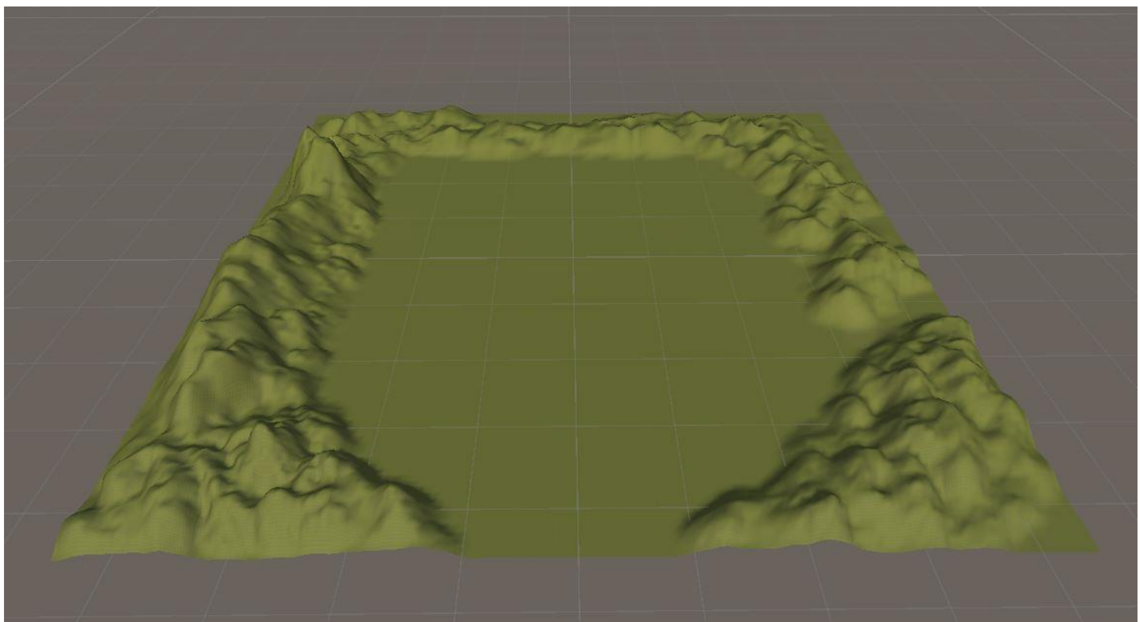


Ilustración 4.2: Resultado final del terreno del prototipo.

4.2. Segundo incremento: Sistema de cuadrícula.

Para la segunda iteración, el sistema de cuadrícula (o “Grid”), primero vamos a crear un objeto vacío llamado “BuildSystem” que albergará la lógica de todo el sistema de cuadrícula, además de posteriormente también incluir el sistema de construcción. Si bien el “GameManager” sirve para concentrar la lógica general del prototipo, considero que tanto el sistema de cuadrícula y construcción son unos sistemas muy complejos, además de estar relacionados entre sí, así que lo mejor es almacenar todas sus partes organizadamente dentro de dicho objeto.

Una vez creado el objeto “BuildSystem” se crea un nuevo objeto vacío llamado “GridParent”, en el que alojaré la propia cuadrícula. Para ello vamos a crear dos objetos dentro de este, “GridVisualization” y “Grid”.

El primero, “GridVisualization”, será un plano que servirá para mostrar el aspecto visual de la cuadrícula. Para ajustarse al terreno le damos un tamaño de 50x50 en los ejes X y Z. Su posición será 0 en los ejes X y Z, y 0.015 en el eje Y. Esto ayudará a situarlo justo por encima del terreno para que así no se solapen. Para la parte visual creamos un pequeño shader rectangular de tamaño 1x1 de color blanco (ilustración 4.3). Esto servirá para poder ver la cuadrícula en el mundo.

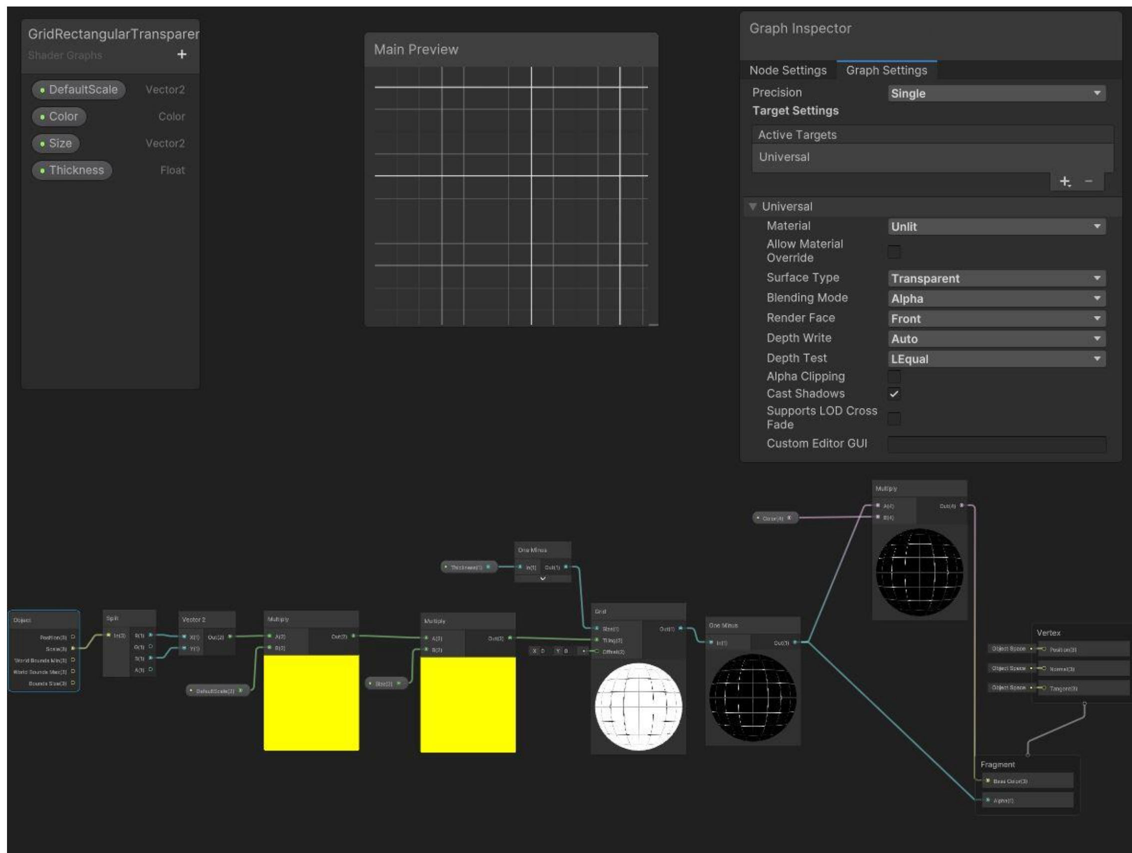


Ilustración 4.3: Shader para representar visualmente las casillas.

El segundo, “Grid”, será donde añadamos el sistema de casillas propiamente dicho. Para hacerlo, tenemos que añadir un componente al objeto llamado “Grid”. Para ajustarse a las dimensiones del terreno previamente hecho, le damos un tamaño de 10x10. Este componente es importante ya que es el que nos proporciona la posibilidad de conversión de coordenadas en un plano normal a casillas y viceversa, lo que nos aportará mayor consistencia cuando pasemos a colocar elementos en el mapa. La vista del mapa con el sistema de casillas se puede observar en la ilustración 4.4.

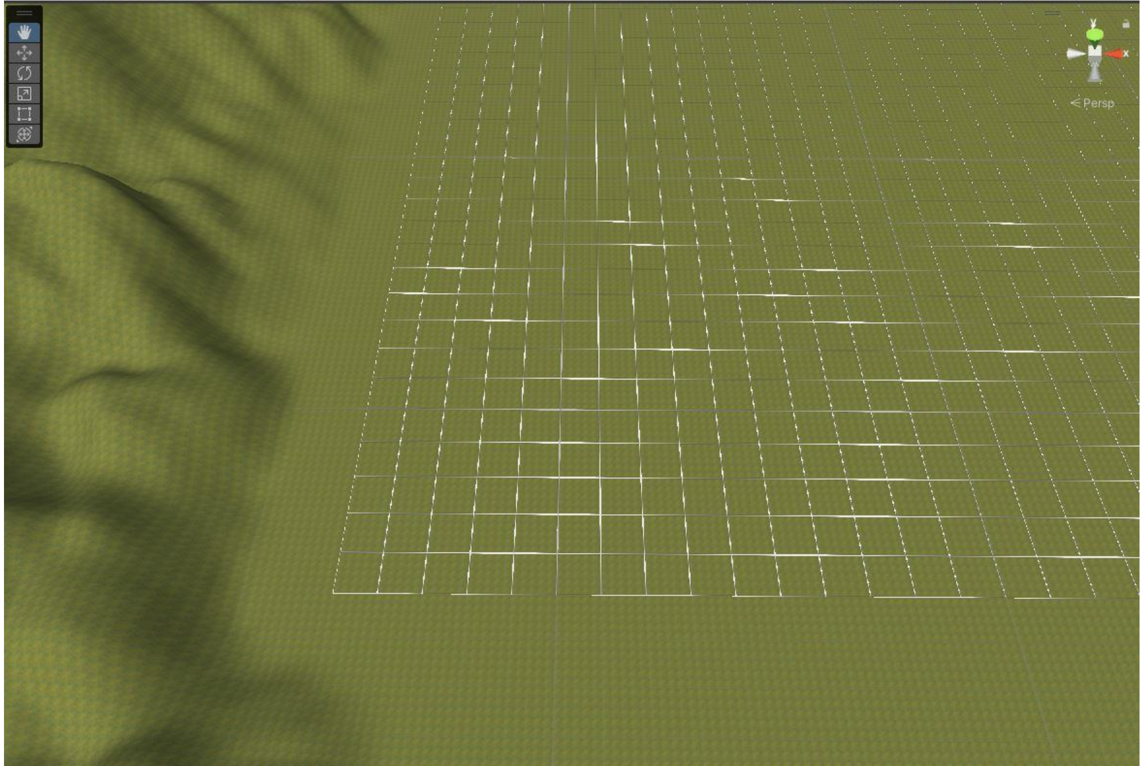


Ilustración 4.4: Visualización del sistema de cuadrícula en el mapa.

En esta iteración también es necesaria la creación de una base de datos propia para almacenar los componentes necesarios para el funcionamiento del proyecto. Se hará mediante “ScriptableObject”, un contenedor de datos propio de Unity en el que se pueden almacenar grandes cantidades de datos de manera independiente a las instancias de clase. Es útil ya que reduce enormemente el uso de memoria en el proyecto al evitar copias de valores innecesarias [27].

En este caso en concreto su uso es interesante ya que voy a tener el mismo prefab repetido numerosas veces sin modificaciones. De esta manera, en lugar de que se instancie el prefab copiando sus propios datos, obteniendo multitud de datos duplicados, al utilizar este sistema los datos de los objetos almacenados se obtendrán por referencia.

Para crear dicha base de datos, primero vamos a crear un script llamado “ObjectDatabaseSO” con los siguientes valores:

- Nombre: Nombre del objeto almacenado.
- ID: ID del objeto.

- Tamaño: Número de casillas que ocupará en el sistema con formato X, Y.
- Prefab: Recurso del objeto.
- Valor madera: Valor o coste del objeto almacenado.
- Valor piedra: Valor o coste del objeto.

El script queda como se muestra en la ilustración 4.5.

```
1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using UnityEngine;
5
6  [CreateAssetMenu]
   Script de Unity | 5 referencias
7  public class ObjectsDatabaseSO : ScriptableObject
8  {
9      public List<ObjectData> objectsData;
10 }
11
12 [Serializable]
   1 referencia
13 public class ObjectData
14 {
15     [field: SerializeField]
       0 referencias
16     public string nombre { get; private set; }
17     [field: SerializeField]
       10 referencias
18     public int id { get; private set; }
19     [field: SerializeField]
       4 referencias
20     public Vector2Int tamanio { get; private set; } = Vector2Int.one;
21     [field: SerializeField]
       1 referencia
22     public GameObject prefab { get; private set; }
23     [field: SerializeField]
       2 referencias
24     public int valorMadera { get; private set; }
25     [field: SerializeField]
       2 referencias
26     public int valorPiedra { get; private set; }
27 }
28
```

Ilustración 4.5: Captura del script “ObjectsDatabaseSO”.

Gracias al “ScriptableObject” creado podemos tener nuestra propia base de datos que se adapte a las necesidades del prototipo. Al crear una base de datos mediante este proceso, se simplifica mucho la acción de añadir nuevos

elementos a la base de datos, ya que solo hay que seleccionarla mediante el inspector de Unity y tendremos acceso de manera visual a la misma, pudiendo añadir nuevos elementos con un simple click.

Además, otra de las ventajas de crear una base de datos mediante este sistema es que, en el caso de modificación o ampliación de la aplicación, solo necesitaríamos modificar nuestro “ScriptableObject” y los campos asociados en la base de datos para adecuarlo a las nuevas necesidades. Logrando así también cumplir con los requisitos no funcionales propuestos, como mantenibilidad y escalabilidad respectivamente.

Para dotar de la lógica necesaria y conseguir que las casillas almacenen información vamos a crear un nuevo script llamado “GridData”. En él crearemos una estructura de datos para almacenar los elementos presentes en el mapa y así unir la parte visual con la parte lógica.

Esta clase seguirá el patrón Singleton previamente explicado, por lo que la instancia quedará como podemos ver en la ilustración 4.6.

```
public static GridData Instance
{
    get
    {
        // Si no hay una instancia existente, crea una nueva
        if (instance == null)
        {
            instance = new GridData();
        }
        return instance;
    }
}

1 referencia
private GridData() { }
```

Ilustración 4.6: Uso del patrón Singleton de la clase GridData.

Hecho esto, lo siguiente es crear una estructura de datos que será la encargada de guardar en cada casilla el objeto colocado en el mapa asociado a la base de datos creada previamente. La estructura de datos será un “Dictionary”, ya que se adapta bien a la estructura del sistema de cuadrícula al poder

almacenar en su llave las coordenadas de la casilla en formato Vector3(X, Y, Z), y en su valor una clase con el objeto colocado en el mapa. Dicho esto, es necesario crear dicha clase. Es una pequeña clase en la que solo existirán los métodos básicos get, set y su constructor. Constará del ID asociado al objeto en la base de datos creada anteriormente, un ID propio del objeto insertado y una lista de Vector3 de las posiciones a ocupar, ya que los objetos asociados pueden ocupar más de una casilla.

Habiendo ya establecido la conexión entre la parte visual del mundo con la parte lógica de la aplicación podemos explicar el valor de la clase creada previamente.

La funcionalidad de esta clase es clave, ya que es la encargada de añadir correctamente los objetos que creemos tanto en el sistema de construcción (edificios) como en la generación de mundo (objetos contenedores de recursos como árboles) a la estructura de datos asociada al sistema de casillas.

Para la correcta inserción de objetos en la estructura de datos nos apoyaremos en funciones de apoyo para comprobar que, en el caso contemplado de objetos de diferente tamaño, sea posible comprobar que las casillas adyacentes necesarias también cumplan los requisitos establecidos.

Una vez conseguido el correcto funcionamiento de esta funcionalidad, dotamos al sistema de la robustez necesaria para que en las siguientes iteraciones nos podamos apoyar en el sistema de casillas sin esperar fallos.

4.3. Tercer incremento: Sistema de cámaras.

Ésta tercera iteración, el sistema de cámaras, consta de dos cámaras más su posterior funcionamiento con sus scripts asociados. La primera cámara es la cámara principal del juego, que será la que el jugador vea en todo momento. La segunda será una cámara auxiliar en la esquina superior derecha que mostrará un mini mapa que se irá moviendo teniendo como referencia la cámara principal.

Primero, es necesario crear un objeto vacío que actúe como grúa para anclar la cámara. Dentro de ese objeto añadimos la propia cámara de Unity y la denominamos “Main Camera”. Para ajustarla en el mundo la retrasamos respecto al origen y la elevamos sobre el plano, las coordenadas XYZ quedan (0, 50, -50). Esta cámara, como vimos en el apartado de diseño, es una cámara con vista isométrica, por lo que el ángulo de visión está enfocado hacía el eje X. Para conseguir este efecto cambiamos la rotación del eje X hasta los 45º.

Para controlar la cámara creamos un script, “CameraController” en el que dotamos de funcionalidad el movimiento de esta.

En la ilustración 4.7 podemos observar el campo de visión de la cámara principal del juego con respecto al mapa.

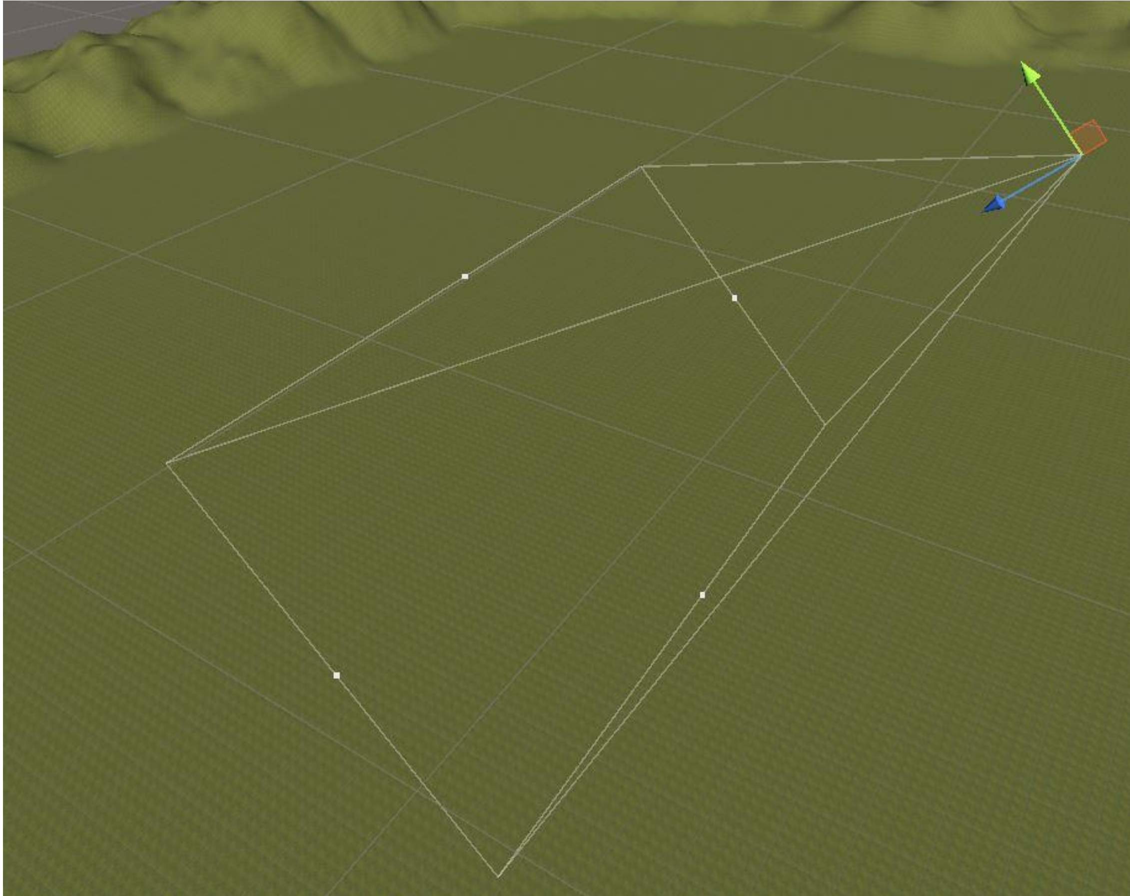


Ilustración 4.7: Campo de visión de la cámara principal del juego.

La cámara principal, además, está relacionado con el terreno previamente implementado en la primera iteración. Esto es debido a que los límites establecidos en el terreno también son los límites hasta los que la cámara puede desplazarse. Esto lo podemos observar en las dos líneas de código, una para cada eje de movimiento, de la ilustración 4.8.

```
nuevaPosicion.x = Mathf.Clamp(nuevaPosicion.x, limitesX.x, limitesX.y);
nuevaPosicion.z = Mathf.Clamp(nuevaPosicion.z, limitesZ.x, limitesZ.y);
```

Ilustración 4.8: Líneas de código asociadas a los límites del mundo.

Finalmente, el resultado de la cámara principal quedará como podremos apreciar en la ilustración 4.9.

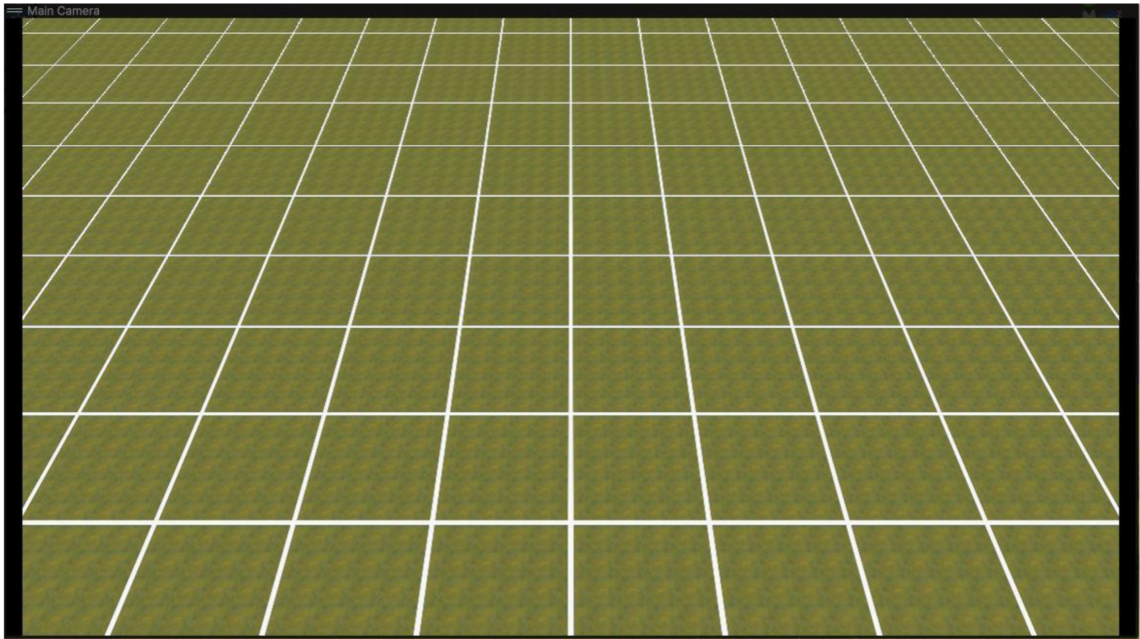


Ilustración 4.9: Vista de la cámara principal.

Para la segunda cámara existente en el prototipo, la cámara del mini mapa hay que añadir otra cámara al proyecto. Debe quedar fuera de la grúa de la cámara principal ya que, si bien el movimiento está asociado, si estuviese dentro no se movería de la forma adecuada. Por lo tanto, añadimos una nueva cámara de Unity llamada “Minimap Camera”.

Esta cámara, al contrario que la principal, tiene una proyección ortogonal al tener que implementar una vista cenital. Para ello, dentro de las propiedades de la cámara debemos cambiar el tipo de proyección y ajustar los valores de tamaño, dejándolo en 60 y de rango de visión, que dejamos ajustados entre -100 y 100. Estos últimos valores limitan el rango en el que la cámara renderiza los objetos existentes en su plano de cámara. Ajustar estos valores es necesario siempre, ya que, aunque en este momento sea un terreno simple, tener unos valores desproporcionados puede desembocar en una pérdida de rendimiento de la aplicación. Podemos observar el campo de visión con los valores establecidos en la ilustración 4.10.

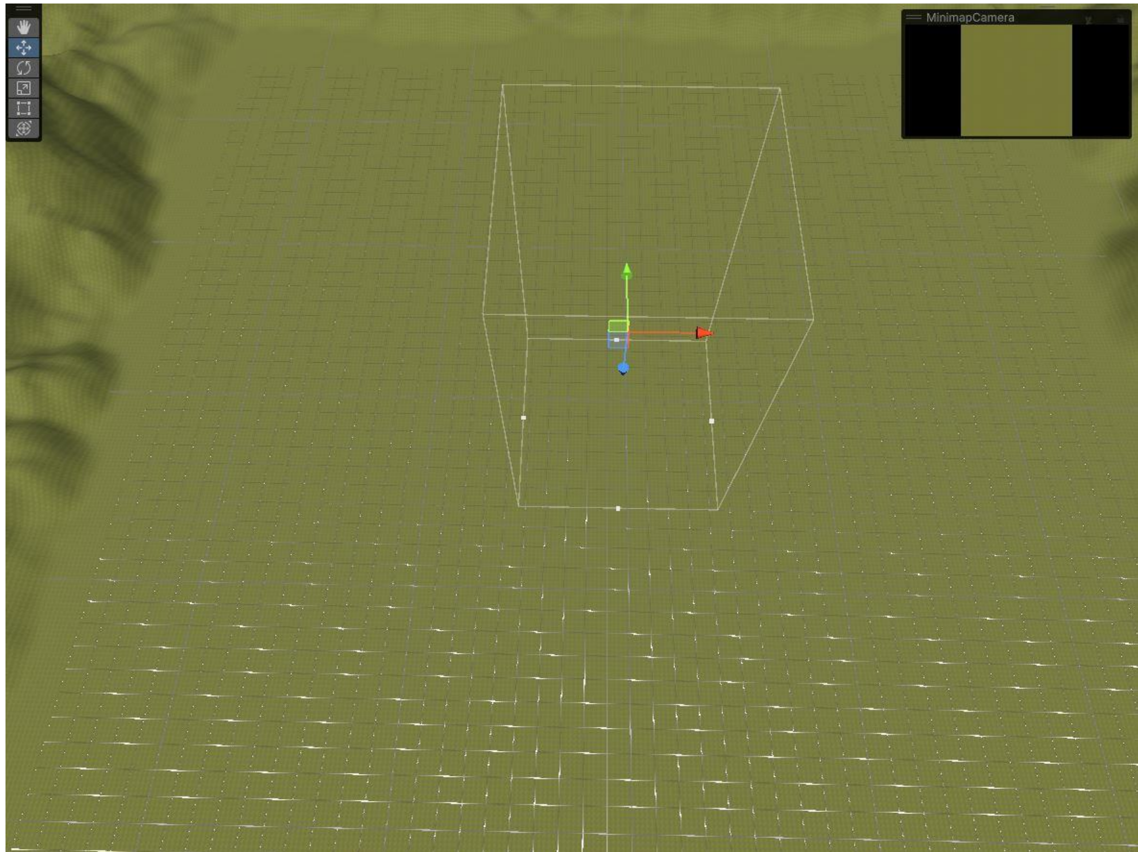


Ilustración 4.10: Campo de visión de la cámara del mini mapa.

Con los valores ya ajustados en el editor de Unity, necesitamos crear un pequeño script para implementar el comportamiento de la cámara. La cámara del mini mapa está directamente asociada a la cámara principal, ya que el mini mapa debe de mostrar desde una perspectiva diferente la información del mundo. Por ello solo hay que añadir un desplazamiento en el eje Z con respecto a la posición de la cámara principal. Podemos observar el funcionamiento de dicho script en la ilustración 4.11.

```
void LateUpdate()
{
    Vector3 newPosition = mainCamera.position;
    newPosition.z += 80f;

    minimapCamera.position = newPosition;
}
```

Ilustración 4.11: Funcionamiento de la cámara del mini mapa.

4.4. Cuarto incremento: Sistema de construcción.

El sistema de construcción es uno de los ejes centrales del proyecto ya que de él depende buena parte de la lógica de la aplicación. Este sistema debe integrar el sistema de cuadrícula previamente creado para la correcta colocación de objetos en las casillas y su estructura de datos interna.

Los assets elegidos [28] para representar los edificios los podemos observar en la ilustración 4.12. La primera casa empezando por la izquierda, ocupará un espacio de 1x1 en el sistema de cuadrícula y representará a los agentes recolectores de madera, mientras que la segunda será de 2x2 y representará a los agentes recolectores de piedra.

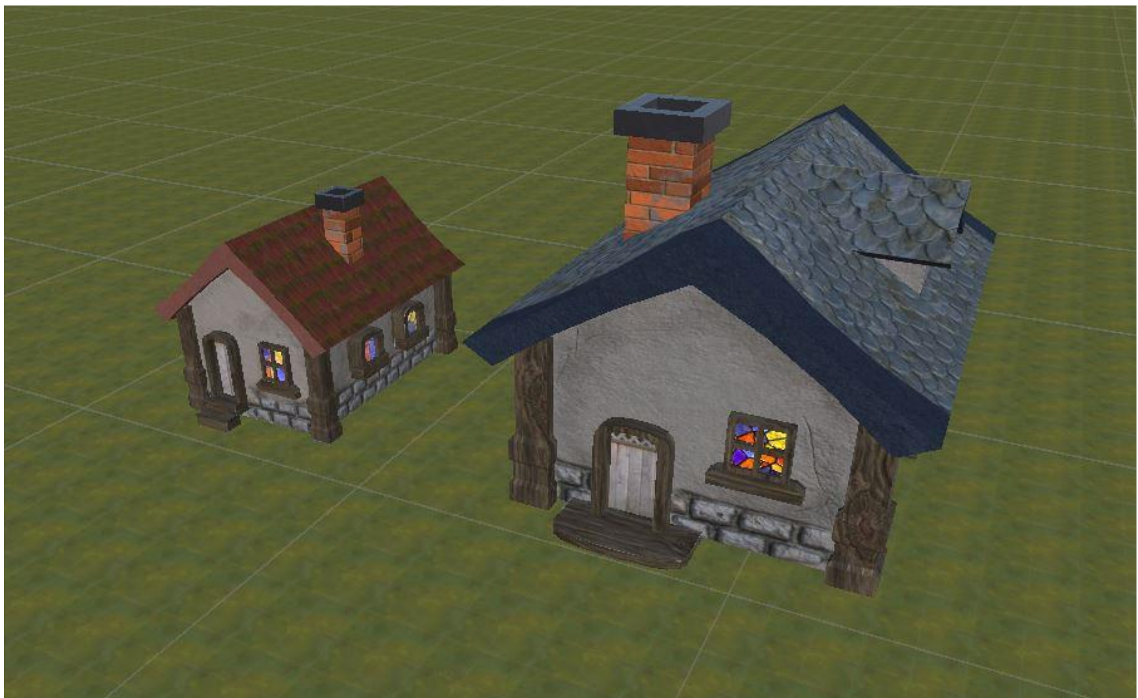


Ilustración 4.12: Assets utilizados para las casas.

Para comenzar el sistema de construcción, necesitamos crear un script que funcione como punto de entrada a las órdenes del usuario. Dicho script será "PlacementSystem". En él tendremos varias funciones que establecerán la transición del ratón mediante el sistema de "Inputs" de Unity con sus coordenadas absolutas al sistema de coordenadas del sistema de cuadrícula. En

esta función también se creará la llamada a una función para comenzar el proceso de colocación de objetos. Para ello me he servido de una pequeña interfaz para añadir consistencia a la aplicación.

Dicha interfaz establecerá un método, “OnAction”, que será el encargado de gestionar la colocación del objeto en el sistema.

Para añadir el objeto al sistema, primero, como vimos previamente en el apartado de Diseño, tendremos que comprobar si las casillas ya están ocupadas. Para ello debemos comprobar el “GridData” previamente creado en el sistema de cuadrícula y ver la disponibilidad de dichas casillas. Para ello, haciendo uso de una llamada a la instancia a dicha función, comprobamos mediante el ID del objeto asociado a la base de datos el tamaño del objeto y vemos si es posible colocarlo en la cuadrícula.

En el caso de que estén libres, pasamos a comprobar si existen recursos disponibles en el sistema. Al ser la economía un paso posterior del desarrollo, por ahora los recursos necesarios para construir un edificio serán de 0 y más adelante nos ocuparemos de darle un valor adecuado al objeto en la base de datos.

Si ambas comprobaciones son correctas, pasamos a añadir el objeto tanto al mundo en la posición indicada, como al “Dictionary” del “GridData” que almacena todos los objetos colocados en el sistema. Para añadir los objetos al mundo nos ayudamos de una pequeña clase llamada “ObjectPlacer” cuya función será tanto la de instanciar el prefab en el mundo, como de añadirlo a una lista de objetos presentes para así tener un feedback visual de los elementos presentes en el juego.

Como podemos observar en la ilustración 4.13. Los objetos se colocan correctamente dentro del sistema de cuadrícula, centrados con respecto a su tamaño y evitando colocarse uno encima de otro.

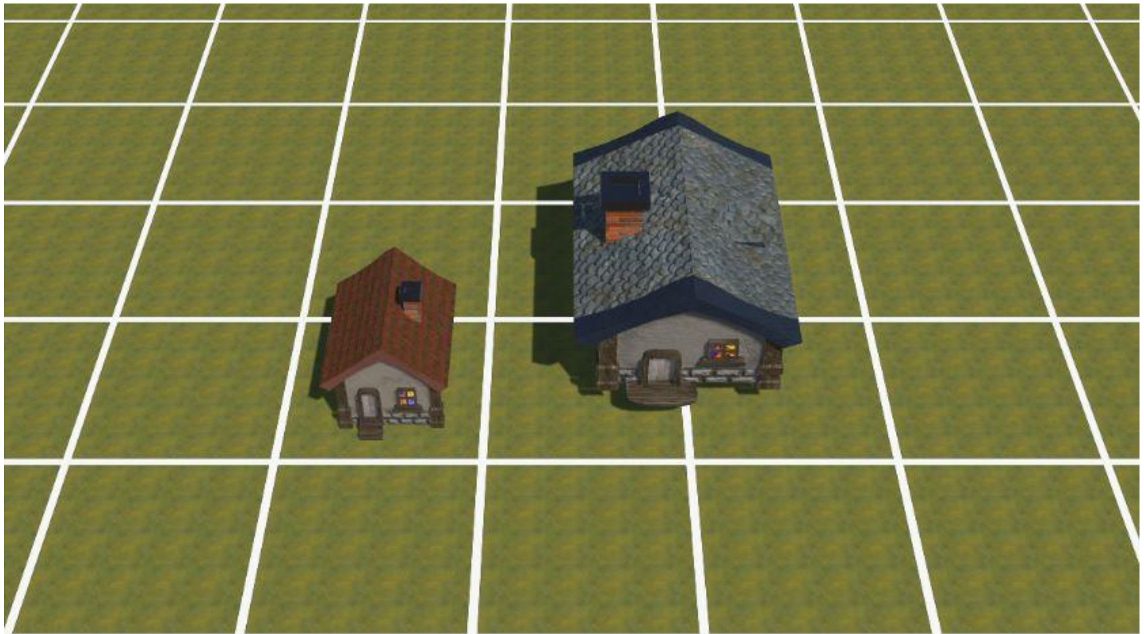


Ilustración 4.13: Edificios colocados en el prototipo.

También, como podemos apreciar en la ilustración 4.14, los objetos se añaden correctamente al sistema y así los podemos visualizar en el propio inspector de Unity.

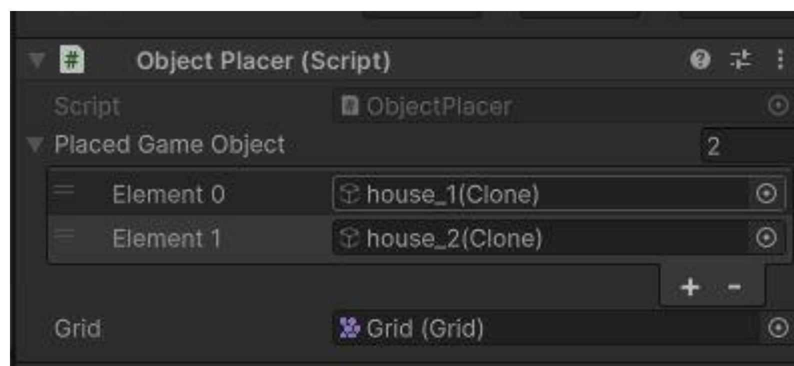


Ilustración 4.14: Edificios presentes en el sistema mediante el inspector.

4.5. Quinto incremento: Sistema de agentes.

Para el Sistema de agentes necesitamos establecer las librerías de las que va a depender su navegación por el mapa y la superficie inteligente que recorrerán.

La primera que se va a utilizar es la librería de Unity denominada “AI.Navigation”. Esta librería proporciona la generación automática de la malla de navegación (o “NavMesh”) a partir de la geometría de la escena [29]. Dicha malla será la encargada de proporcionar a los agentes la superficie útil en la que poder moverse.

Para crearla, necesitamos añadir un nuevo objeto vacío a la escena principal y añadir el componente “NavMeshSurface”. Una vez añadida, necesitamos ajustar los parámetros para que la malla no se solape con los edificios que construyamos o los recursos que se generen. También para que los agentes del tipo “humanoide”, que son los que se van a utilizar, consideren la superficie como transitable. Hecho esto, debemos de hacer click en la opción “bake” para que finalmente genere la malla. Este paso solo se debe de hacer una vez, ya que la malla es inteligente y se debe adaptar a los cambios que se incluyan en el mapa mediante llamadas en código a su función propia de actualización.

En la ilustración 4.15 podemos ver la parametrización y la localización del botón “Bake” de la malla de navegación citada anteriormente.

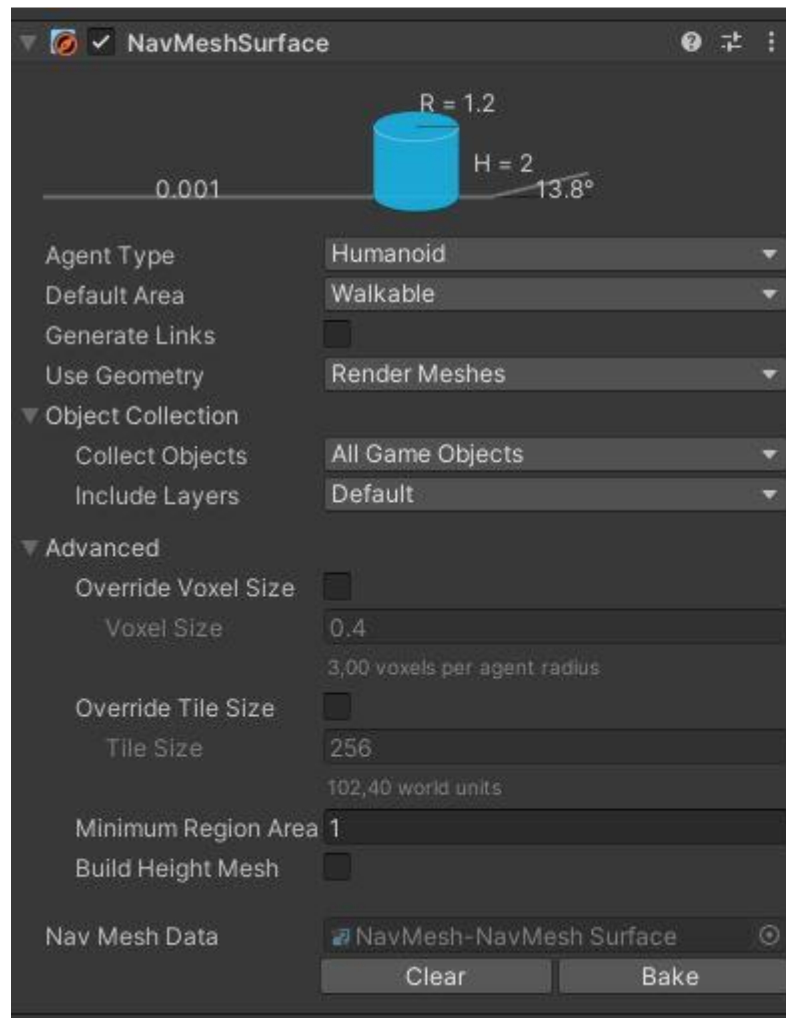


Ilustración 4.15: Parámetros de la superfie de la malla de navegación y localización del botón "bake".

El resultado de los ajustes lo podemos observar en la ilustración 4.16. En ella vemos como la superficie, de un calor azul-verdoso, respeta los elementos que aparecen en el mapa, dejando márgenes suficientes (dados los parámetros ajustados) para que los agentes puedan transitar el terreno.

Ilustración 4.16: NavMesh de la escena.

La segunda librería que se va a utilizar es la incorpora el motor por defecto, “UnityEngine.AI”. La librería se utiliza principalmente para implementar características de navegación y búsqueda de caminos en los juegos [30]. Esta librería es la que da la posibilidad de añadir agentes inteligentes al juego, los llamados “NavMeshAgent” con funciones propias de búsqueda de caminos.

Para crear los agentes que se van a utilizar, primero era necesario dotarlos de un apartado visual, para ello voy a utilizar assets gratuitos de la tienda de Unity [31]. Para utilizarlos como nuestros agentes por defecto y establecerlos como “NavMeshAgent” necesitamos convertirlos a prefab para poder instanciarlos con código. Una vez hecho añadimos el componente “Nav Mesh Agent”, encargado de convertir al agente y le damos los parámetros adecuados. El resultado lo podemos observar en la ilustración 4.17.

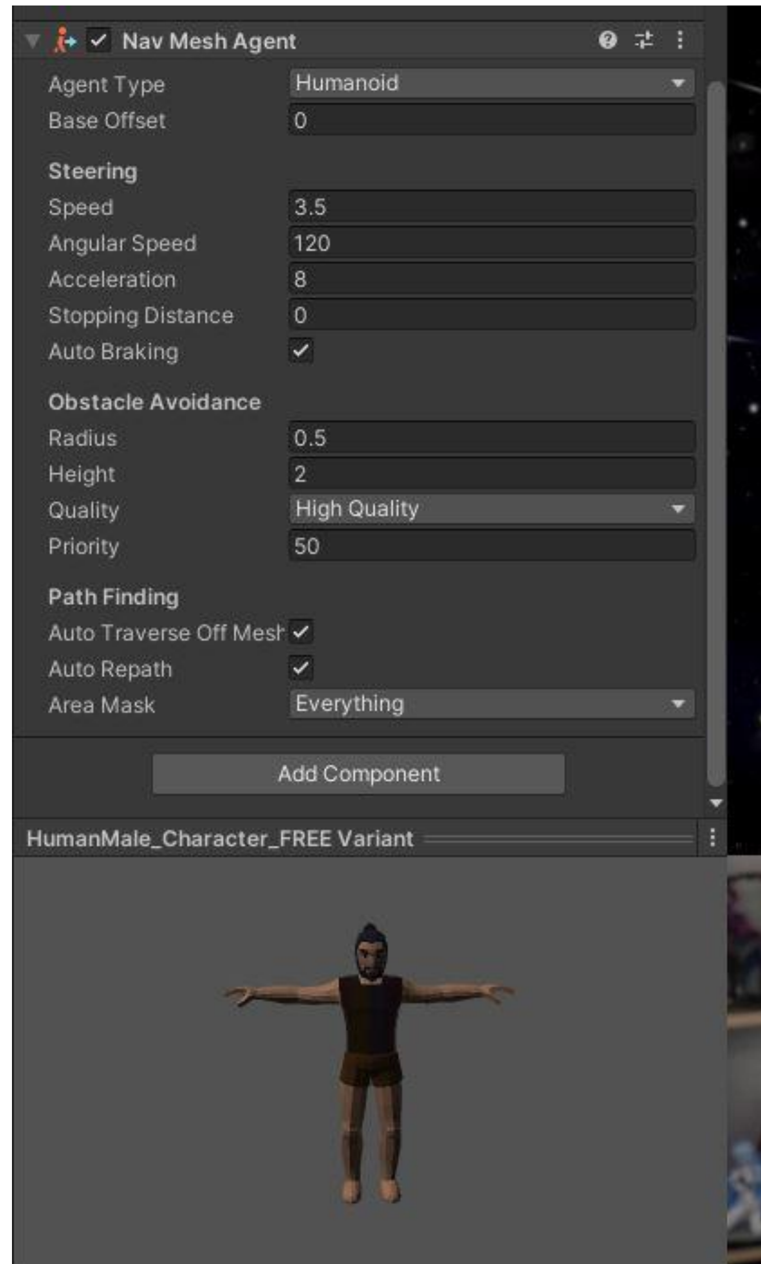


Ilustración 4.17: Parámetros y estética del NavMeshAgent.

Una vez hechos los pasos previos, pasamos a dotar de funcionalidad a los agentes. Necesitamos crear un nuevo script anclado a nuestro “GameManager”, “AgentSystem” en el que estará el código.

El sistema de agentes tiene dos características, la primera es que también utiliza el patrón Singleton, y la segunda, es que al necesitar múltiples agentes trabajando en paralelo será necesario dotarlos de concurrencia. En Unity, para establecer el uso de corutinas es necesario la utilización de las funciones

IEnumerator. Estas funciones permiten ejecutar código que puede pausarse y reanudarse en diferentes puntos, lo que es útil para crear efectos de tiempo, secuencias de eventos y evitar que el código se sobrescriba [32].

Como vimos anteriormente en el apartado anterior de Diseño, los agentes se crearán con un comportamiento diferente asociado a cada edificio, es decir, un agente creado en el edificio de leñadores trabajará recolectando madera y un agente creado en el edificio de mineros trabajará picando piedra. Por lo tanto, las llamadas a la función se harán al construir el edificio.

La idea de implementación de tener dos tipos de agentes con dos edificios asociados distintos se debe a intentar dotar a la aplicación de una mayor escalabilidad. Al tener dos tipos de agentes cada uno creado de diferentes edificios, y que sea funcional, me permite comprobar que efectivamente es posible crear agentes con diversos comportamientos, lo que a futuro serviría para poder añadir nuevos edificios y agentes con funcionalidades distintas.

Dentro del script de nuestro sistema de agentes, lo instanciamos con los parámetros necesarios. En la ilustración 4.18 podemos ver dicha instancia, aunque lo importante de ese fragmento de código es la última línea que es la encargada de instanciar al agente con la corutina.

```
public void WoodAgentInstantiator(Vector3 gridPosition)
{
    NavMeshAgent newWoodAgent = Instantiate(woodAgentPrefab, gridPosition, Quaternion.identity);
    agentList.Add(newWoodAgent);
    StartCoroutine(WoodAgentBehaviour(newWoodAgent));
}
```

Ilustración 4.18: Instancia de un agente leñador.

Una vez instanciado, hay que crear el comportamiento del agente, para ello necesitamos crear primero una serie de puntos en el mapa para simular que son los recursos, ya que en la siguiente iteración será la encargada de la creación del mundo y de instanciar dichos recursos. Esta función es fácilmente reemplazable una vez los recursos reales estén generados, así que no supone demasiado tiempo extra.

Para comenzar a dotar de comportamiento al agente, lo primero es buscar el recurso asociado más cercano, una vez encontrado, utilizamos la función que la plataforma Unity proporciona a sus agentes para la búsqueda de caminos y con ella se pueda mover automáticamente hacia el recurso evitando los obstáculos gracias a la malla anteriormente creada.

Seguidamente, el agente comenzará a trabajar en el recurso esperando 10 segundos, considerando que es la simulación del trabajo y eliminando el recurso de la estructura de datos en el que estaba almacenado una vez pasado dicho tiempo. Para terminar el comportamiento, el agente volverá al comienzo de la función siempre y cuando sigan quedando recursos para recolectar, y volverá a repetir la función hasta que se agoten.

Para finalizar el sistema de agentes también voy a añadir animaciones básicas para mejorar el aspecto visual. Para ello he añadido assets de animaciones de la tienda de Unity [33]. Para que sean funcionales, necesitamos añadir al prefab del agente creado anteriormente el componente “Animator” de Unity. Dentro del sistema de animaciones de Unity, creamos el flujo de animación utilizando las animaciones del paquete obtenido en la tienda de Unity.

El resultado lo podemos ver en la ilustración 4.19. En ella podemos observar como el agente comienza en una posición de espera (o “idle”), de ella pasa a la animación de correr, y en función del tipo de agente pasará a la animación de trabajo distinta. Dichas animaciones pueden volver también hacia la anterior ya que contemplo la casuística de que el mapa quede sin recursos, y, por tanto, tenga que volver a una posición de espera.

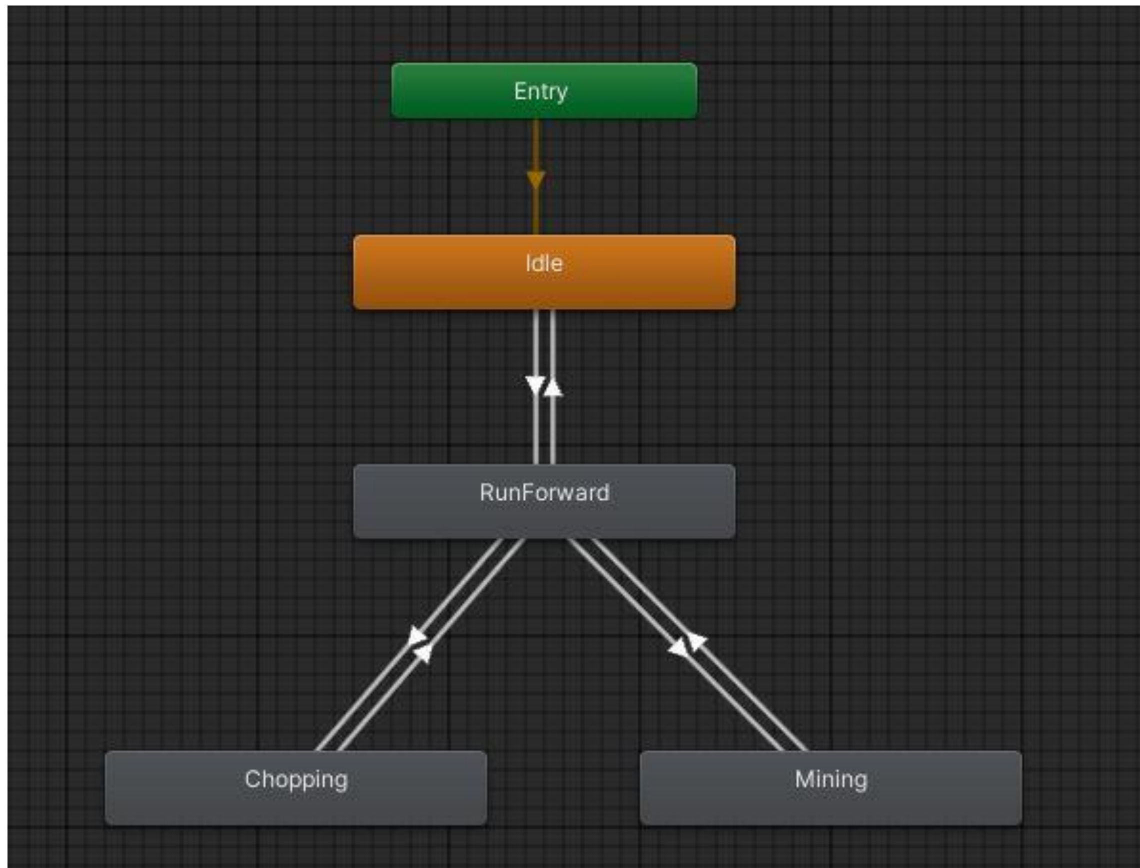


Ilustración 4.19: Flujo de animaciones de los agentes.

Con esto, y a falta de dotar al sistema con su propia economía, tenemos el sistema de agentes funcional moviéndose alrededor del mundo.

4.6. Sexto incremento: Generación de mundo y economía.

Este apartado se dividirá en dos puntos, ya que, si bien ambos elementos están relacionados, tienen presencia suficiente para poder hacerlo.

4.6.1. Generación de mundo.

Para generar el mundo, o concretamente los elementos que lo componen, vamos a crear un script que, dado un número pasado por el inspector de Unity, genere en posiciones aleatorias dentro de la cuadrícula ese número de cada tipo de recurso, en este caso, árboles y rocas. Podemos ver los recursos obtenidos en la tienda de Unity [34][35] en la ilustración 4.20.

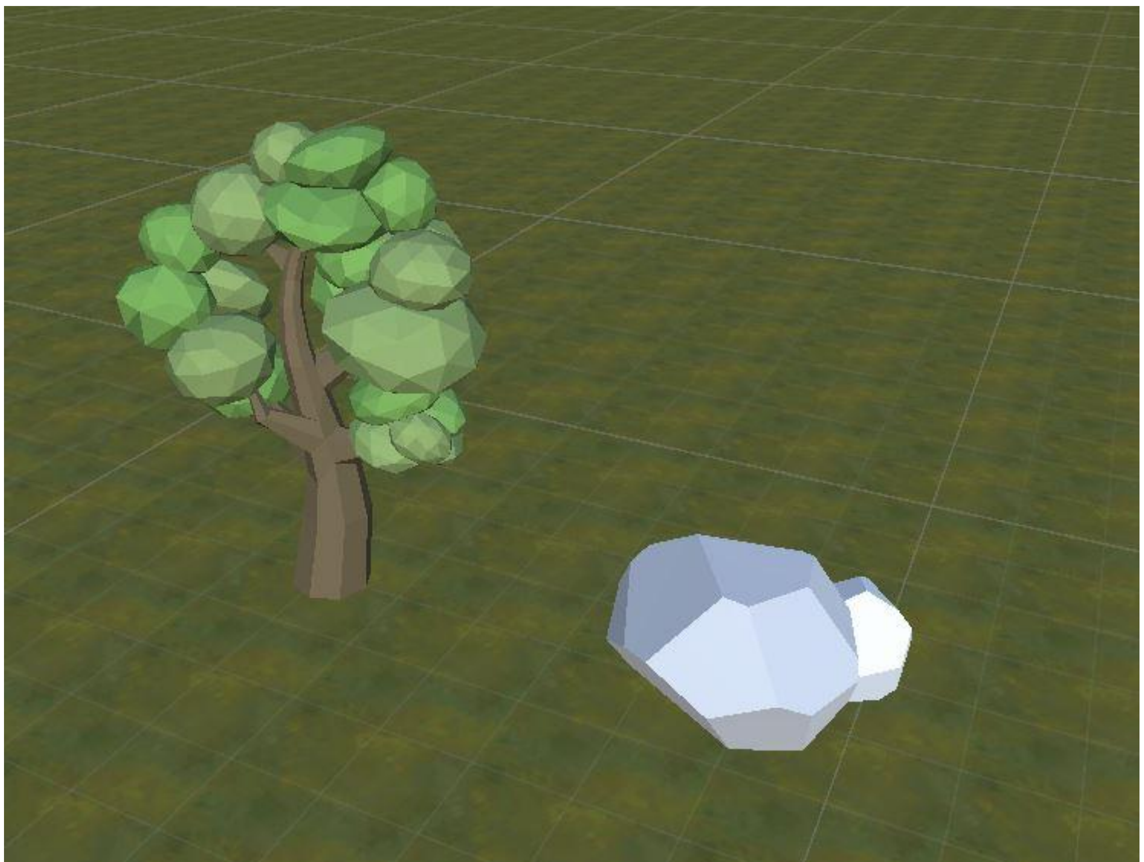


Ilustración 4.20: Assets empleados para los recursos existentes en el juego.

El sistema usado para la colocación de estos objetos en el mundo es similar al proceso que vimos previamente en el sistema de construcción, con la única particularidad de que estos objetos siempre tienen de tamaño 1 casilla.

El tamaño inicial de cada recurso será, teniendo en cuenta que tenemos un sistema de cuadrícula de 50x50, es decir, con 2.500 casillas en total, de 800 unidades por tipo de recurso. La pauta de colocación como he comentado es similar a la construcción y sigue sus mismas reglas, por lo que si una casilla está ocupada no se podrá colocar nada encima. He considerado que para ahorrar tiempo de procesamiento y para que, aunque sea en posiciones aleatorias, no tenga siempre el mismo número de objetos, obviar cuando una casilla esté ocupada y no volver a intentar colocar el mismo objeto. Así que finalmente cuando ejecutemos la función obtendremos un total de aproximadamente 1.600 objetos en el mapa.

También he considerado que, para darle un aspecto más orgánico al mundo, cada objeto insertado en el mundo tendrá un tamaño y rotación diferentes.

Para el tamaño ha sido necesario la creación de una pequeña función auxiliar que modifica la escala de los objetos con un valor aleatorio entre unos rangos determinados. Al modificar la escala de los objetos, es necesario tener en cuenta la proporción original del objeto, ya que si los valores fuesen completamente aleatorios dichas proporciones se romperían y el resultado no sería visualmente atractivo. Para respetar las proporciones simplemente ha sido necesario asignar el mismo valor aleatorio a los ejes X y Z (ancho y largo), y agregar un valor fijo como suma al valor aleatorio anterior en el eje Y (altura). Para la rotación basta con establecer un valor entre 0º y 360º para conseguir buenos resultados.

Una vez realizados dichos cambios, conseguimos establecer un aspecto heterogéneo en los objetos que componen el mundo del juego rompiendo la uniformidad y mejorando notablemente el aspecto visual del juego, como podemos apreciar en la ilustración 4.21.

Ilustración 4.21: Recursos del juego tras haber modificado su escala y rotación.

Los objetos contenedores de recursos también se adaptan a la malla de navegación creada anteriormente, por lo que los agentes pueden navegar libremente esquivando los obstáculos.

El resultado final, desde la vista con el “NavMesh” activado la podemos ver en la ilustración 4.22. En el podemos ver también los assets utilizados de la tienda de Unity.

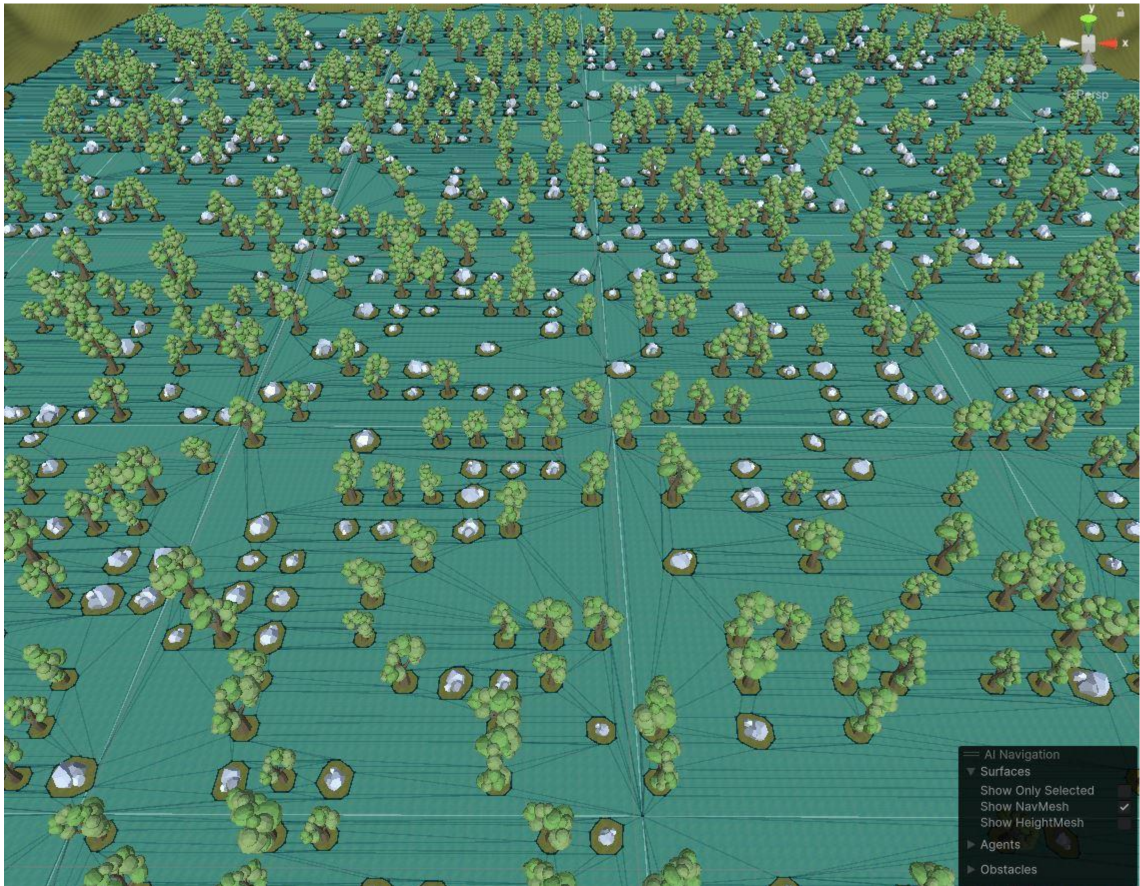


Ilustración 4.22: Resultado de la generación de mundo.

Los recursos generados al comienzo de la ejecución son todos los recursos que habrá durante el transcurso de la partida, es decir, no se regenerarán.

La idea de poder regenerar recursos fue algo contemplado como añadido en la implementación, pero finalmente ha quedado descartada. Si bien en algunos juegos del género de construcción de ciudades es posible la regeneración de recursos, como la replantación de árboles, por ejemplo. Esto siempre suele estar apoyado mediante el uso de agentes y edificios especializados para conseguir ese propósito, lo que implicaría una sobrecarga de trabajo excesiva ya que sería necesario añadir cambios sustanciales tanto al sistema de construcción como al sistema de agentes.

4.6.2. Economía del juego.

Para la economía del juego hay que dotar de valor a los objetos existentes en la base de datos y añadir la lógica necesaria para que el sistema funcione.

Primero es necesario añadir un nuevo script al “GameManager” para controlar la economía. En el vamos a asignar como valores iniciales al total de nuestros recursos de 100 de madera y piedra.

Después, a los edificios le vamos a poner el siguiente coste:

- Edificio del leñador: 60 de madera y 40 de piedra.
- Edificio del minero: 40 de madera y 60 de piedra.

Con esos valores nos aseguramos de que haga lo que haga el jugador, siempre va a poder colocar un edificio de cada tipo para así asegurarse de que los agentes pueden comenzar a recolectar los árboles y rocas.

Para añadir los valores de dichos edificios a la lógica, solo hay que añadir una función en el script para, al haber preparado previamente la comprobación al construir el edificio, hacer una llamada a dicha función que compruebe si hay recursos suficientes.

Seguidamente, es necesario dar valor también a los árboles y rocas para que los agentes puedan crear valor al recolectarlos. Le daremos un valor de 10 unidades de recurso. Ya que los agentes tienen como tiempo asociado al ciclo de recolección 10 segundos, nos asegura que dichos agentes recolecten a una velocidad alta.

Cuando los agentes terminen de recolectar el valor que el objeto posee, el objeto será eliminado del mapa y de la estructura de datos interna. Una vez eliminado, la malla de navegación se adaptará a los cambios y lo que antes estaba ocupado por el recurso, volverá a ser transitable por los agentes.

También, al haber liberado el espacio, será posible construir en el nuevo espacio vacío.

4.7. Séptimo incremento: Interfaz.

La implementación de la interfaz establecerá finalmente la conexión entre la información del juego y el jugador mediante el uso de elementos visuales.

Como primer paso, debemos de establecer el Menú inicial cuya escena establecimos al comienzo de la fase de desarrollo. Para ello, debemos añadir a la escena un elemento “canvas” del apartado UI de Unity. El “canvas” será el espacio de trabajo a la que añadir los diferentes elementos que compongan la interfaz de dicho menú.

Seguidamente, para que una imagen se pueda insertar en la interfaz de Unity primero es necesario convertirla a Sprite. Para ello debemos agregar la imagen al proyecto y en el propio editor de Unity cambiar la propiedad “texture” a Sprite (2D and UI). Este paso es necesario para cada imagen que haya que incluir en la interfaz. Hecho esto, ya es posible añadir la imagen de fondo del menú.

A continuación, hay que añadir botones para dotar al menú de funcionalidad. La funcionalidad prevista en fase de diseño es la de entrar al propio juego y salir de la aplicación, así que es necesario la creación de dos botones.

El primero, llamado “Inicio”, dará paso a la escena principal que contiene el juego. Para ello hay que añadir a la función “OnClick” del botón una llamada a una función que de paso a la escena.

El segundo, llamado “Salir”, cerrará la aplicación al pulsarlo. Para ello también debemos añadir otra función al “OnClick” del botón.

Para finalizar la interfaz del menú inicial he añadido música de fondo y un botón para controlar la reproducción y dar también posibilidad de silenciarla. Para añadir sonidos a la escena debemos añadir a la escena el elemento de Unity llamado “AudioSource”, y establecer como clip en bucle la pista de audio. En la parte superior izquierda he añadido un botón con el icono intercambiable de sonido y silencio. Con él se podrá silenciar la reproducción del audio.

El resultado final de la composición del menú inicial lo podemos observar en la ilustración 4.23.

Ilustración 4.23: Interfaz del menú inicial.

La interfaz de la escena principal del juego está dividida en tres secciones:

- Panel superior: en el que se mostrará la información sobre la economía y agentes activos en el juego.
- Panel inferior: en el que se establecerán los botones de construcción de los dos edificios presentes en el prototipo.
- Mini mapa: visión de la cámara del mini mapa creada previamente en la parte superior derecha.

En el panel superior, la información se mostrará mediante la combinación de texto e iconos que representen los recursos y tipos de agentes. Como imagen de fondo tendrá un pergamino para conseguir una estética uniforme con el resto de la interfaz. El resultado del panel superior lo podremos ver en la ilustración 4.24.

Ilustración 4.24: Panel superior de la interfaz principal del juego.

Para el panel inferior, se utilizarán los propios edificios como botones de construcción, además de texto para especificar cada edificio. Para unir estos elementos con el sistema de construcción hay que modificar el evento “OnClick” de dichos botones con el script del sistema de construcción previamente creado mediante la llamada a la función “StartPlacement()” que da comienzo al proceso. Para seguir con la misma estética general, se ha utilizado otra imagen de pergamino para la imagen de fondo del panel inferior. El resultado lo podemos ver en la ilustración 4.25.

Ilustración 4.25: Panel inferior de la interfaz principal del juego.

Para el mini mapa, simplemente vamos a establecer un borde de un color que respete la estética y colocar la cámara auxiliar previamente creada. El resultado obtenido aparece en la ilustración 4.26.

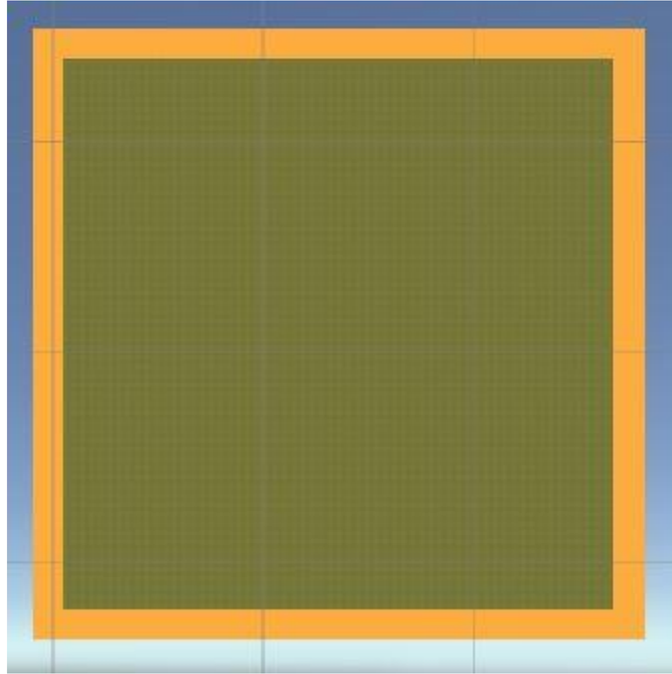


Ilustración 4.26: Resultado del mini mapa en la interfaz principal del juego.

Finalmente, la integración de la interfaz con la lógica del juego se hará mediante el script “UIController”, que se comunicará con el script en el que controlamos la economía para así modificar los textos del panel superior con cada evento que suponga una modificación en la economía.

La composición de la interfaz resultante integrada en juego la podemos ver en la ilustración 4.27.

Ilustración 4.27: Composición final de la interfaz en el prototipo.

4.8. Octavo incremento: Ajustes finales y pruebas.

En este apartado se expondrán las pruebas de ajuste y balanceo final a las que el prototipo se ha visto sometido, además de implementar soluciones para errores no detectados sobre iteraciones previas.

4.8.1. Pruebas.

Para finalizar el desarrollo de la aplicación, es necesario realizar una fase de pruebas previas para ajustar el balance de la economía del juego. Para la realización de las pruebas vamos a obviar el tiempo de desplazamiento, ya que este será muy variable.

Basándonos en los valores creados al inicio, tenemos los siguientes valores base de los edificios (tabla 4.2):

Nombre	Costo madera	Costo piedra
Casa de leñador	60	40
Casa de minero	40	60

Tabla 4.1: Coste de los edificios.

Además, cada árbol y roca tiene un valor asociado de 10 unidades cada uno, existen aproximadamente 800 de cada tipo, y un agente recolecta 10 unidades cada 10 segundos.

Con los valores iniciales, con un solo agente, se tarda aproximadamente 133 minutos en vaciar el mapa de recursos ((800x10) /60 = ~133).

Dado que la mecánica del juego es construir más edificios y estos añadir más agentes, la relación entre el tiempo y los agentes del juego sigue una función inversamente proporcional (T(n) = 1 / n), donde T= tiempo y n = número de agentes. El agotamiento de objetos contenedores de recursos es demasiado rápido cuando comenzamos a colocar edificios. Esto convierte los valores iniciales en inviables.

Por ello, vamos a establecer cambios en los valores. El nuevo valor de unidades de recurso por objeto pasará a ser de 100, mientras que los agentes tardarán 3 segundos por ciclo de recolección. Aumentaremos en un edificio de cada tipo cuando consigamos 100 de cada recurso.

Los nuevos valores nos dan un tiempo de partida de aproximadamente 4 minutos, lo que sigue siendo insuficiente.

Nuevamente volvemos a cambiar los valores. Esta vez la cantidad de recursos por objeto será de 200, y el ciclo de recolección por agenté pasará a los 5 segundos. Esta vez, la duración de las partidas se incrementan hasta alcanzar aproximadamente los 15 minutos, lo que considero aceptable, por lo que dichos valores van a ser los elegidos.

4.8.2. Corrección de errores.

Una vez terminado el prototipo, fueron detectados dos errores, el primero afectaba el correcto funcionamiento del juego, y el segundo ralentizaba el flujo del juego de una manera inaceptable para un proyecto de esta envergadura.

El primer error hacía que la construcción no tuviera en cuenta si los agentes estaban en el espacio en el que se iba a colocar el edificio, lo que causaba que los agentes podían quedar atrapados indefinidamente dentro. Esto se debía a que, al utilizar el sistema de cuadrícula, toda la lógica de construcción se basa en el código creado en lugar de utilizar, como frecuentemente se hace, la propia detección de colisiones que Unity ofrece.

La solución propuesta fue la de incluir una nueva comprobación a las ya existentes en el sistema de construcción en la que saber en el momento de construir las posiciones exactas de los agentes existentes dentro de nuestro sistema de cuadrícula, es decir, tomar también como “ocupadas” las casillas en las que en el momento de intentar colocar el edificio haya un agente moviéndose por allí.

Para ello fue necesaria la creación de una modificación en el script del sistema de agentes para, en el momento de la instancia de los agentes, crear una lista con el total de agentes presentes en el juego. Para completar la solución, también fue necesaria la creación de una función en el mismo script que recorriera la lista obteniendo sus posiciones y devolviéndolas. Dicha función sería llamada desde el sistema de construcción. La función la podemos ver en la ilustración 4.28.

```
// Función para comprobar las posiciones de los agentes
1 referencia
public List<Vector3Int> GetAgentsPositions()
{
    List<Vector3Int> positions = new List<Vector3Int>();
    foreach (NavMeshAgent agent in agentList)
    {
        positions.Add(grid.WorldToCell(agent.transform.position));
    }
    return positions;
}
```

Ilustración 4.28:Solución propuesta para el error con el sistema de construcción.

El segundo error consistía en la ralentización de toda la aplicación al construir un edificio o eliminar un recurso del mapa, es decir, al modificar el total de elementos existentes en la escena. Esto se debía a la forma en la que se trataba la generación de la malla navegable, o “NavMesh”, que se generaba completamente cada vez que se añadía o eliminaba algún obstáculo a la misma, como eran los edificios o los objetos de recursos.

En la fase inicial del desarrollo, cuando se creó el sistema de construcción, no había muchos elementos en escena, pero posteriormente, al implementar la generación de mundo y aumentar drásticamente el número de objetos que modificaban la malla, esto pasó al ser un problema al tener que generarla continuamente. Este problema generaba micro cortes en el desarrollo del juego dando como resultado una mala experiencia para el jugador.

Este problema, pese a su aparente sencillez, me supuso todo un reto de conseguir arreglar y una lección más de aprendizaje en la búsqueda de recursos en línea.

La solución propuesta encontrada más común consistía en dividir la malla de navegación en cuadrículas más pequeñas e ir regenerándolas en aquellas zonas en las que se produjesen los cambios. Esta solución fue descartada ya que implicaría eliminar parte de la escalabilidad, al tener que establecer manualmente diversas mallas independientes, y en caso de ampliar el mapa, reformular toda la estructura de mallas.

Finalmente, la solución estaba en el lugar más obvio, la documentación de Unity. Una de las características que se añadió en versiones posteriores del "NavMesh" ofrece una solución a la actualización de la malla. Para ello solo es necesario llamar a la función que podemos ver en la ilustración 4.29.

```
navMeshSurface.UpdateNavMesh(navMeshSurface.navMeshData);
```

Ilustración 4.29: Línea de código necesaria para actualizar la malla de navegación.

Esta función a la que pasamos los datos de nuestra propia malla de navegación hace que al actualizarla, solo se modifique la parte afectada, consiguiendo así que la eficiencia mejore notablemente.

5. Conclusiones y trabajo futuro.

Normalmente se considera el desarrollo de videojuegos como uno de los productos software más complejos, ya que es necesario tener conocimientos en multitud de campos. El desarrollo de este Proyecto de Fin de Grado no ha sido una excepción y he podido comprobar la dificultad que un proyecto de este tipo puede entrañar.

La consecución del prototipo de videojuego no ha sido solo un trabajo para demostrar el conocimiento aprendido en las asignaturas cursadas en el grado, sino que además ha servido para seguir adquiriendo conocimiento en el inmenso mundo del desarrollo software.

Este proyecto también ha servido para analizar el mercado del sector, y ver que, aunque haya géneros que no lleguen a millones de jugadores, siempre es posible encontrar tu hueco en el mercado si haces un producto lo suficientemente bueno.

Uno de los aspectos que como alumnos más ignoramos, la planificación y el diseño previo, han resultado ser vitales para completar el proyecto con éxito, ya que consiguieron el propósito de aclarar las ideas y en el momento de la fase de desarrollo, saber lo que se debía de hacer.

Hace casi 15 años, cuando me inicié en el mundo de la programación, no me imaginé que, siendo una de mis mayores pasiones el mundo de los videojuegos, iba a poder realizar por mí mismo uno. Si bien lo logrado aún carece aún de algunos elementos importantes para conseguir la gamificación apropiada para que el público general se fije en él, el prototipo sienta unas bases para conseguir productos finales completos.

Habiendo obtenido elementos versátiles como un sistema de cuadrícula funcional, es posible no solo aplicarlo a juegos de construcción de ciudades, sino a diversos géneros tan dispares como juegos de estrategia, de defensa de torres etc.

Habiendo dicho todo esto, el prototipo solo es el comienzo de la aplicación futura. Como trabajo futuro pienso seguir trabajando en él y como propuestas de trabajo surgen las siguientes:

- Ampliación de recursos existentes en el juego.
- Aumento de la complejidad de obtención de recursos (procesado, conversión, etc.).
- Ampliación de nuevos tipos de agentes, como por ejemplo los constructores, agricultores etc.
- Ampliación de nuevos tipos de edificios para sostener los puntos anteriores.
- Inclusión de un sistema de caminos que doten a los agentes de mayor velocidad cuando transiten por él.
- Sustitución de assets del juego por diseños propios.

Apéndice I. Guía de instalación.

En este apéndice se explicarán los pasos a seguir para la ejecución del software creado en el Trabajo de Fin de Grado.

El prototipo puede probarse de dos maneras diferentes:

La primera es mediante el uso del ejecutable incluido en la entrega. Para ejecutar la aplicación utilizando este método solo es necesario descomprimir el archivo.zip y ejecutar el .exe de su interior. Esta es la opción recomendada si solo se quiere probar el prototipo.

La segunda opción es mediante la instalación de la herramienta de desarrollo utilizada para su creación, Unity. En este método tendremos acceso al código fuente de la aplicación, también tendremos la posibilidad de modificar diversos valores presentes desde el inspector de Unity como valores iniciales de recursos, coste de los edificios etc. Para la instalación de Unity debemos seguir los del siguiente apartado.

1. Guía de instalación del motor de desarrollo Unity.

Para instalar Unity en su sistema debe seguir los siguientes pasos. Cabe destacar que para poder ejecutar la plataforma de desarrollo es necesario la creación de una cuenta de Unity:

1. Acceder a la página web https://unity.com/es/download
2. Pulsar el botón de descarga, esto comenzará la descarga de la versión más reciente del programa “Unity Hub”.
3. Instalar el programa “Unity Hub” siguiendo los pasos recomendados.
4. Entrar a la siguiente web https://unity.com/releases/editor/archive
5. Instalar la versión de Unity que prefiera. Como recomendación, la versión en la que el proyecto fue creado es la “**2022.2.18f1**”, por lo que

recomiendo encarecidamente que si no se desean tener problemas de compatibilidad se descargue dicha versión del editor.

6. Descargar desde la entrega del proyecto, el proyecto de Unity en formato .zip y guardarlo en su sistema.

Una vez seguidos los pasos anteriores, el usuario deberá abrir el programa “Unity Hub” previamente descargado y pulsar desde la parte superior derecha “Add Project from disk”.

Una vez hecho el proyecto con todos los recursos necesarios podrá ser modificado, probado o utilizado desde el motor Unity.

Apéndice II. Guía de usuario.

En este apéndice se explicarán las acciones que el usuario puede realizar en el software desarrollado en el Trabajo de Fin de Grado. Se mostrarán las acciones posibles en las dos escenas que lo componen, el menú inicial y la escena principal del juego.

Al entrar en la aplicación el usuario se encontrará con el menú principal mostrado en la ilustración A II. 1.

Ilustración A II. 1: Menú inicial de la aplicación.

Las acciones que el usuario puede llevar a cabo son:

1. Hacer click en el botón "Iniciar". Inicia el juego.
2. Hacer click en el botón "Salir". Cierra la aplicación.
3. Hacer click en el icono de sonido de arriba a la izquierda. Silencia el sonido, si se vuelve a pulsar, el sonido se vuelve a activar.

Si el usuario avanza en la pantalla anterior, pasará a la escena principal del juego y podrá ver la escena que aparece en la ilustración A II. 2.

Ilustración A II. 2: Vista principal del juego.

En la vista principal del juego, el usuario puede controlar la cámara con las siguientes teclas:

1. Tecla W o flecha arriba: Desplazamiento hacia arriba.
2. Tecla S o flecha abajo: Desplazamiento hacia abajo.
3. Tecla A o flecha izquierda: Desplazamiento hacia la izquierda.
4. Tecla D o flecha derecha: Desplazamiento hacia la derecha.
5. Tecla Q: Rotación hacia la izquierda.
6. Tecla E: Rotación hacia la derecha.

El usuario también puede hacer click sobre alguno de los edificios, “Casa de Leñador” o “Casa de Minero”, para entrar en el modo de construcción cuya vista podremos observar en la ilustración A II. 3.

Ilustración A II. 3: Vista del modo construcción del juego.

El usuario en esta vista también puede desplazarse por el mundo con los controles de cámara previamente explicados.

El usuario puede hacer click en un edificio y luego en la casilla que desee construirlo, y siempre que se cumplan las condiciones requeridas el edificio se colocará en el mapa.

Si el usuario lo desea puede presionar la tecla “escape” para salir del modo construcción.

Para cerrar la aplicación en cualquier momento el usuario deberá pulsar la combinación de teclas “Alt + F4”.

Apéndice III. Recursos del juego no referenciados.

En este apéndice se expondrán los recursos utilizados en el proyecto que no han sido referenciados en la memoria.

Icono de madera: https://icon-library.com/icon/wood-icon-png-20.html

Icono de piedra: Icongeek_26. https://www.flaticon.com/free-icon/stone_4405457

Icono de hacha: Freepik. https://www.flaticon.com/free-icon/axe_809139?related_id=809037

Icono de pico: Freepik. https://www.flaticon.com/free-icon/pickaxe_3380348

Icono de sonido: Pixel perfect. https://www.flaticon.com/free-icon/volume_727269

Icono de silencio: Pixel perfect. https://www.flaticon.com/free-icon/mute_727240

Música del menú inicial: Yevhen Onoychenko. https://pixabay.com/es/users/onoychenkomusic-24430395/?utm_source=link-attribution&utm_medium=referral&utm_campaign=music&utm_content=136824

Apéndice IV. Descripción del material entregado.

En este apéndice se listarán los archivos que se han entregado como Trabajo de Fin de Grado:

- **Memoria del proyecto**: TFG_Diaz_Verdejo_Jose_Luis.pdf
- **Código fuente del proyecto**: Proyecto.zip
- **Ejecutable**: Ejecutable.zip
- **Vídeo de demostración**: video_demostracion.mp4

Bibliografía

[1] Statista Market Insights (2024, marzo). Worldwide revenue by market. https://www.statista.com/outlook/dmo/digital-media/video-games/worldwide#revenue [Último acceso: 8 abril 2024]

[2] Statista Market Inisights (2024, marzo). Worldwide users by market. https://www.statista.com/outlook/dmo/digital-media/video-games/worldwide#users [Último acceso: 8 abril 2024]

[3] Unity Technologies. Compare Unity plans. https://unity.com/products/compare-plans [Último acceso: 8 abril 2024]

[4] Epic Games Inc. Opciones de licencia de Unreal Engine. https://www.unrealengine.com/es-ES/license [Último acceso: 8 abril 2024]

[5] SteamDB. Steam Indie Game Releases by Month. https://steamdb.info/stats/releases/?tagid=492 [Último acceso: 8 abril 2024]

[6] Eliza Crichton-Stuart (2024, 10 marzo) Indie Games Claim 31% of All Steam Revenue. https://gam3s.gg/news/indie-games-claim-31-perfect-of-all-steam-revenue/ [Último acceso: 9 abril 2024]

[7] Games-Stats.com City Builder Games – Steam Marketing Tool. https://games-stats.com/steam/?tag=city-builder [Último acceso: 9 abril 2024]

[8] Gabriel Mancuzo (2021, 13 mayo) Qué es el modelo incremental. https://blog.comparasoftware.com/que-es-el-modelo-incremental/ [Último acceso: 11 abril 2024]

[9] Alejandro Aldás Alarcón (2016, abril) Modelo de proceso incremental. https://www.researchgate.net/figure/Figura-10-Modelo-de-proceso-incremental-Fuente_fig6_326571456 [Último acceso: 11 abril 2024]

[10] Mark J. P. Wolf (2012, 16 agosto) Encyclopedia of Video Games: The Culture, Technology, and Art of Gaming Volume 1. https://www.researchgate.net/publication/273946688_Encyclopedia_of_video_games_the_culture_technology_and_art_of_gaming [Último acceso: 11 abril 2024]

[11] Richard Moss (2015, 10 noviembre) From SimCity to, well, SimCity: The history of city-building games. https://arstechnica.com/gaming/2015/10/from-simcity-to-well-simcity-the-history-of-city-building-games/3/ [Último acceso: 11 abril 2024]

[12] Shining Rock Software LLC (2014, 18 febrero) Banished Game. https://store.steampowered.com/app/242920/Banished/ [Último acceso: 11 abril 2024]

[13] Video Game Insights (2024) Banished – Steam Stats. https://vginsights.com/game/242920?utm_source=SteamDB [Último acceso: 11 abril 2024]

[14] Polymorph Games (2024) Foundation Game. https://www.polymorph.games/en/ [Último acceso: 11 abril 2024]

[15] Colossal Order Ltd. (2015, 15 marzo) Cities: Skylines Game. https://store.steampowered.com/app/255710/Cities_Skylines/ [Último acceso: 11 abril 2024]

[16] SteamDB (2024). Most played City Builder games. https://steamdb.info/charts/?tagid=4328 [Último acceso: 11 abril 2024]

[17] Eventyr (2023, 12 abril). Game Development with Unreal Engine: Pros and Cons. https://eventyr.pro/blog/game-development-with-unreal-engine-pros-and-cons/ [Último acceso: 18 abril 2024]

[18] Kevuru Games (2023, 16 marzo). Unity vs Unreal Engine: Pros and Cons. https://kevurugames.com/blog/unity-vs-unreal-engine-pros-and-cons/ [Último acceso: 18 abril 2024]

[19] FANDOM Games Community (2023, 16 agosto) Anno 1404 production chains. https://anno1404.fandom.com/wiki/Production_chains [Último acceso: 21 abril 2024]

[20] GeekforGeeks (2024, 21 enero) Functional vs Non Functional Requirements. https://www.geeksforgeeks.org/functional-vs-non-functional-requirements/ [Último acceso: 22 abril 2024]

[21] Techaway (2024, 7 marzo) ¿Sabes cuál es la vida media de un ordenador? https://techaway.es/vida-media-ordenador/#:~:text=En%20general%2C%20se%20estima%20que,adecuado%20y%20se%20actualizan%20peri%C3%B3dicamente. [Último acceso: 10 marzo 2024]

[22] Infortisa (2023, 12 abril) ¿Cuál es la diferencia entre las tablets y las tabletas gráficas? https://blog.infortisa.com/cual-es-la-diferencia-entre-las-tablet-y-las-tabletas-graficas/ [Último acceso: 10 marzo 2024]

[23] Gobierno de España (2024, 6 febrero) El Gobierno aprueba la subida del SMI. https://www.lamoncloa.gob.es/consejodeministros/resumenes/Paginas/2024/060224-rp-cministros.aspx [Último acceso: 10 marzo 2024]

[24] Tecnoempleo (2024) Salarios por funciones profesionales IT https://www.tecnoempleo.com/tecnocalculadora.php [Último acceso: 10 marzo 2024]

[25] Blog Desarrollo de los videojuegos (2023, 12 julio) La guía completa para comprender el patrón Singleton. https://desarrollodelosvideojuegos.com/desenmascarando-los-singleton-en-c-la-guia-completa-para-comprender-el-patron-singleton/ [Último acceso: 24 abril 2024]

[26] Chromisu (2023, 23 enero) Handpainted Grass & Ground Textures. https://assetstore.unity.com/packages/2d/textures-

materials/nature/handpainted-grass-ground-textures-187634 [Último acceso: 14 abril 2024]

[27] Unity Documentation (2018, 15 octubre) ScriptableObjects. https://docs.unity3d.com/Manual/class-ScriptableObject.html [Último acceso: 22 abril 2024]

[28] Eva3D (2019, 31 enero) Fantasy Houses Exterior. https://assetstore.unity.com/packages/3d/environments/fantasy/fantasy-houses-exterior-138320 [Último acceso: 21 mayo 2024]

[29] Unity Documentation (2022) AI Navigation. https://docs.unity3d.com/Packages/com.unity.ai.navigation@2.0/manual/index.html [Último acceso: 15 agosto 2024]

[30] Unity Documentation (2024, 9 marzo). AI. https://docs.unity3d.com/2022.3/Documentation/Manual/com.unity.modules.ai.html [Último acceso: 14 junio 2024]

[31] Blink (2022, 19 abril) FREE Low Poly Human - RPG Character. https://assetstore.unity.com/packages/3d/characters/humanoids/fantasy/free-low-poly-human-rpg-character-219979 [Último acceso: 6 junio 2024]

[32] Unity Documentation (2024, 9 marzo) MonoBehaviour.StartCoroutine. https://docs.unity3d.com/ScriptReference/MonoBehaviour.StartCoroutine.html [Último acceso: 8 junio 2024]

[33] Explosive (2019, 3 septiembre) Crafting Mecanim Animation Pack FREE. https://assetstore.unity.com/packages/3d/animations/crafting-mecanim-animation-pack-free-45047 [Último acceso: 25 junio 2024]

[34] Ada_King (2017, 4 diciembre) Free Trees. https://assetstore.unity.com/packages/3d/vegetation/trees/free-trees-103208 [Último acceso: 25 agosto 2024]

[35] Broken Vector (2018, 10 julio) Low Poly Rock Pack. https://assetstore.unity.com/packages/3d/environments/low-poly-rock-pack-57874 [Último acceso: 25 agosto 2024]