UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

HERRAMIENTAS PARA EL DESARROLLO DE VIDEOJUEGOS CON PROYECCIÓN ISOMÉTRICA

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

GONZALO ANDRÉS URIBE MONTERO

PROFESORA GUÍA:
NANCY HITSCHFELD KAHLER

MIEMBROS DE LA COMISIÓN:
MARÍA RIVARA ZUÑIGA
JÉRÉMY BARBAY

SANTIAGO DE CHILE
2020

RESUMEN DE LA MEMORIA PARA OPTAR
AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN
POR: GONZALO ANDRÉS URIBE MONTERO
FECHA: SEPTIEMBRE, 2020
PROF. GUÍA: NANCY HITSCHFELD KAHLER

HERRAMIENTAS PARA EL DESARROLLO DE VIDEOJUEGOS CON PROYECCIÓN ISOMÉTRICA

Existen muchos videojuegos que utilizan una proyección isométrica para simular un espacio tridimensional mediante el uso de imágenes bidimensionales. El proceso que se realiza para conseguir esta ilusión ya está bien documentado por varias fuentes, pues se ha usado de manera continua a lo largo de los años para distintos videojuegos. No obstante, a pesar de la popularidad de este proceso, no existen herramientas que permitan utilizar esta proyección de manera sencilla. Esto significa que cada vez que se utiliza, deben reimplementarse funcionalidades básicas y estandarizadas que fácilmente podrían compartirse entre distintos videojuegos.

En esta memoria se implementa un conjunto de herramientas que permite abstraer a los desarrolladores de los detalles que conlleva la proyección isométrica, se optimiza su integración con el motor de videojuegos *Unity* y se diseña una arquitectura general que podrá ser reutilizada en otros motores.

La solución desarrollada consiste en una serie de componentes modulares e independientes que permiten incorporar en Unity de manera granular las distintas funcionalidades que se requieren en los juegos isométricos tales como transformación entre coordenadas isométricas y cartesianas, posicionamiento de elementos en el nivel, separación del modelo y la vista, serialización de niveles y construcción procedural de sprites en función de su entorno.

Las herramientas se probaron desarrollando prototipos de videojuegos que simulan escenarios reales de uso, lo cual permitió validar que la integración funciona sin interrumpir el flujo normal de desarrollo y que efectivamente disminuyen la cantidad de código escrito por los desarrolladores.

Agradezco a mis padres por siempre priorizar mis estudios y preocuparse de que no me falte nada para poder seguir adelante con ellos.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

1.1. Motivación

Tanto las fotografías como los gráficos 3D por computadora utilizan algún tipo de proyección. Esta permite visualizar en una superficie bidimensional, como una pantalla, objetos tridimensionales. Si bien existen varios tipos distintos de proyecciones, una de las más utilizadas en videojuegos es la isométrica.

Esta es una proyección en la que los tres ejes ortogonales forman ángulos de 120 grados al proyectarse y los objetos miden lo mismo independiente de su distancia con el observador, como se muestra en la Figura 1.1.

Figura 1.1: Escena que utiliza la proyección isométrica.

La proyección isométrica fue muy popular antes de la masificación de los motores de renderizado tridimensional ya que permitía mostrar escenas en tres dimensiones usando imágenes bidimensionales y sin la necesidad de escalarlas, lo cual habría sido un gran problema debido

a la baja resolución que tenían las imágenes de la época.

Aunque con el aumento de la resolución con la que trabajan los computadores, las técnicas de alisado de bordes y los motores de renderizado 3D, esta proyección ya no es necesaria para visualizar espacios tridimensionales, de todas formas, sigue utilizándose para muchos juegos. A pesar de que ésta proyección ha mantenido cierta popularidad hasta la actualidad (ver Figura 1.2), no existen herramientas modernas con buen soporte para juegos isométricos. Esto significa que para hacer uno de estos, los desarrolladores deben invertir su tiempo en implementar funcionalidades estándar que ya se han programado muchas veces. Por ejemplo, el posicionamiento de sprites y la construcción procedural de distintos sprites para que sean consistentes con sus vecinos.

Figura 1.2: Juegos con proyección isométrica: Zaxxon (1982), Populous (1989), Age Of Empires II (1999) y Project Zomboid (2020), respectivamente.

En esta memoria se crea un conjunto de herramientas que permite a los desarrolladores abstraerse de las complicaciones específicas de esta proyección para poder concentrarse en implementar aquellos elementos únicos de su aplicación, aumentando así su productividad.

Para conseguir esto se modelan los distintos elementos de los juegos isométricos de manera que los programadores puedan interactuar con ellos de manera intuitiva y sean renderizados de manera automática.

Estas herramientas están diseñadas para el motor Unity y se pueden importar directamente en los proyectos que lo necesiten.

1.2. Estado del Arte

1.2.1. Herramientas integradas en motores

Para diseñar una herramienta como ésta, primero se investigaron las alternativas ya existentes. Para esto se buscaron los motores de videojuegos más utilizados en la actualidad y se averiguó la disponibilidad de herramientas para juegos isométricos en cada uno de ellos.

Dos de las más grandes plataformas de distribución de videojuegos entregan al público estadísticas relacionadas con los juegos que publican [2] [14]. De aquellos datos se pudo extraer la popularidad de los motores actuales, la cual se muestra en la Figura 1.3.

Se hizo una investigación respecto a las capacidades y disponibilidad de cada uno de los motores para ver si permiten el desarrollo de juegos isométricos. Esto permitió descartar inmediatamente los motores Source, CryEngine y Gamebryo por no tener soporte para juegos 2D [1] [4] [15]. También se descartó el motor Anvil por no estar disponible para el público [11], ya que es propiedad de Ubisoft.

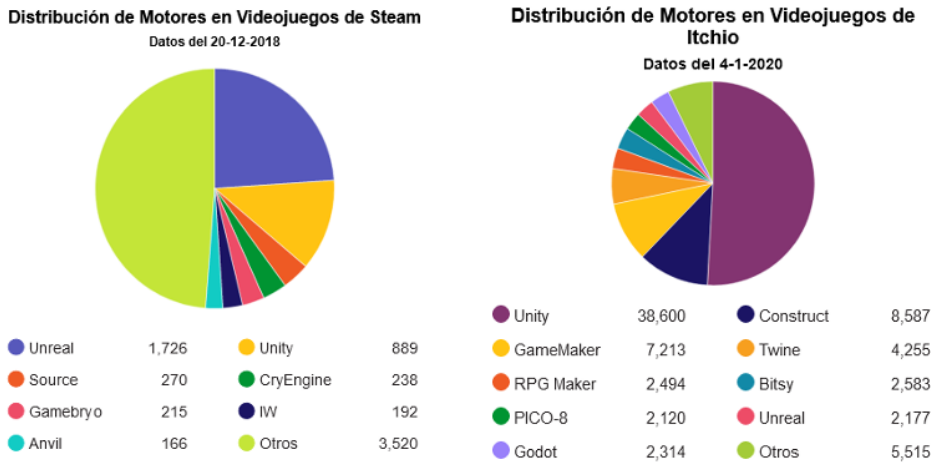

Figura 1.3: Popularidad de motores de videojuegos en distintos mercados.

Finalmente, se descartaron los motores RPG Maker, Twin, IW y Bitsy por estar diseñados para géneros específicos de juegos y carecer de la flexibilidad necesaria para crear un juego isométrico [3] [7] [10] [9].

De los motores restantes, solo Unity y Godot tienen herramientas para este tipo de juegos [5]. Estas herramientas permiten crear niveles posicionando sprites en la pantalla y renderizándolos en el orden correcto. El problema con ellas es que no tienen ningún tipo de interacción con el resto del motor [8]. Luego de que los objetos quedan posicionados no hay nada que indique que están simulando una perspectiva isométrica, ni que permita trabajar con ellos de esta forma.

Motor	Soporte 2D	Público	Múltiples Géneros	Herramientas existentes
Unreal	Si	Si	Si	No
Unity	Si	Si	Si	Si
Source	No	Si	Si	No
CryEngine	No	Si	Si	No
Gamebryo	No	No	Si	No
IW	No	Si	Si	No
Anvil	No	No	Si	No
Construct	Si	Si	Si	No
GameMaker	Si	Si	Si	No
Twine	No	Si	No	No
RPG Maker	Si	Si	No	No
Bitsy	Si	Si	No	No
PICO-8	Si	Si	Si	No
Godot	Si	Si	Si	Si

Tabla 1.1: Soporte para desarrollo de juegos isometricos en distintos motores.

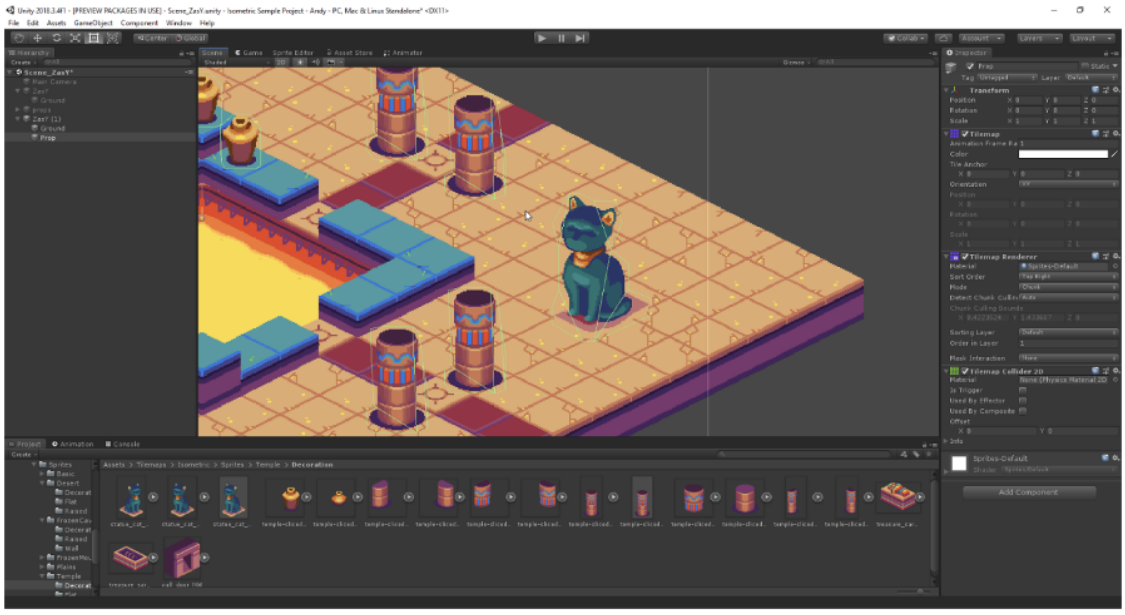

Figura 1.4: Editor de mapas isométricos de Unity.

1.2.2. Herramientas autónomas

Además de las herramientas integradas con los motores de videojuegos, se buscaron herramientas autónomas que permitan ayudar en el proceso de crear juegos isométricos.

La única herramienta que se encontró fue Tiled: un programa que permite crear y visualizar niveles para luego exportarlos en formato json o csv. Esta herramienta permite crear mapas isométricos, pero tampoco permite trabajar con ellos en mayor profundidad. De hecho, para importar los niveles a cualquier motor se debe implementar como mínimo un algoritmo de transformación entre coordenadas cartesianas e isométricas.

1.3. Objetivos

1.3.1. Objetivo General

El objetivo principal de esta memoria es diseñar e implementar un conjunto de herramientas que permiten el desarrollo de juegos isométricos sin tener que preocuparse de las dificultades que conlleva esta proyección. Estás herramientas fueron publicadas bajo una licencia de código abierto.

Estas herramientas fueron desarrolladas para funcionar con Unity, pues es el motor con mayor cantidad de usuarios en la actualidad, cumple con los requisitos para poder desarrollar juegos isométricos y, además, posee una buena plataforma de distribución de herramientas.

Sin embargo, la arquitectura y diseño de clases fueron modelados de forma genérica, esto permite que sirvan como guía para implementar las estas mismas herramientas en otros motores que soporten gráficos 2D.

Para esto se trabajó de manera iterativa, donde primero se construyó e implementó una arquitectura que permite entender los elementos que componen un juego isométrico y sus relaciones entre ellos, la cual puede ser utilizada como modelo para otros motores. Luego se realizó una segunda iteración, en la cual se implementaron optimizaciones especificas para Unity que mejoran el rendimiento de las herramientas y su usabilidad.

1.3.2. Objetivos Específicos

- Implementar herramientas que permitan crear niveles isométricos, modificarlos e interactuar con sus distintos elementos.
- Diseñar una arquitectura genérica que permita implementar las herramientas en otros motores.
- Mantener la flexibilidad de las herramientas, permitiéndoles acomodarse a distintos tipos de juegos en proyección isométrica.
- Organizar el proyecto de tal manera que permita ser extendido, dando facilidades para integrarse con distintas tecnologías.
- Lograr que las herramientas sean independientes entre sí, permitiendo usar solo las que sean necesarias para cada proyecto.
- Asegurarse de que las herramientas compartan la filosofía de diseño del motor de videojuegos para el que se vayan a implementar, buscando que su uso resulte natural a los programadores con experiencia en el motor.

Si bien, la implementación del proyecto tiene una dificultad considerable, se espera que el mayor desafío de esta memoria sea diseñar la arquitectura del proyecto de tal manera que sea modular y su uso resulte intuitivo y así, efectivamente, logre aumentar la productividad de los desarrolladores.

1.4. Estructura de la memoria

Lo que resta del documento se divide en seis capítulos. El Capítulo 2 presenta los antecedentes necesarios para entender la perspectiva isométrica y Unity, el motor que se utiliza durante el desarrollo del proyecto. En el Capítulo 3 se analizan los casos de uso y se diseña la arquitectura general del motor, lo cual se implementa en el Capítulo 4. El Capítulo 5 muestra el desarrollo de una prueba de concepto. Con los aprendizajes obtenidos con la prueba de concepto, en el Capítulo 6 se implementan optimizaciones especificas para el motor Unity. En el Capítulo 7 se valida la utilidad del proyecto mediante la creación de pequeños prototipos que utilizan las herramientas creadas en la memoria y en el Capítulo 8 se presentan las conclusiones.

Capítulo 2

Antecedentes

2.1. Proyección isométrica en videojuegos

En estricto rigor, la proyección isométrica se define como una proyección axonométrica en la que existe un ángulo de 120 grados entre cada uno de sus tres ejes. En el ámbito de la ingeniería y el diseño estos ángulos iguales son de suma utilidad, pues permiten mostrar de manera simple y natural las dimensiones y los tamaños, pero al desarrollar videojuegos se utiliza una variación en la que los ángulos están levemente alterados.

La razón para este cambio es que una linea con un ángulo de 120 grados posee un patrón de píxeles complejo, lo cual es muy notorio en bajas resoluciones y no se considera agradable a la vista. En videojuegos, el ángulo utilizado en la mayoría de los casos es de 126.87 grados, el cual genera lineas con patrones mucho mas simples y estéticamente agradables, como se puede ver en la Figura 2.1. Como en este caso las imágenes cumplen una función mas artística y menos técnica, la pequeña distorsión que conlleva este cambio no es relevante.

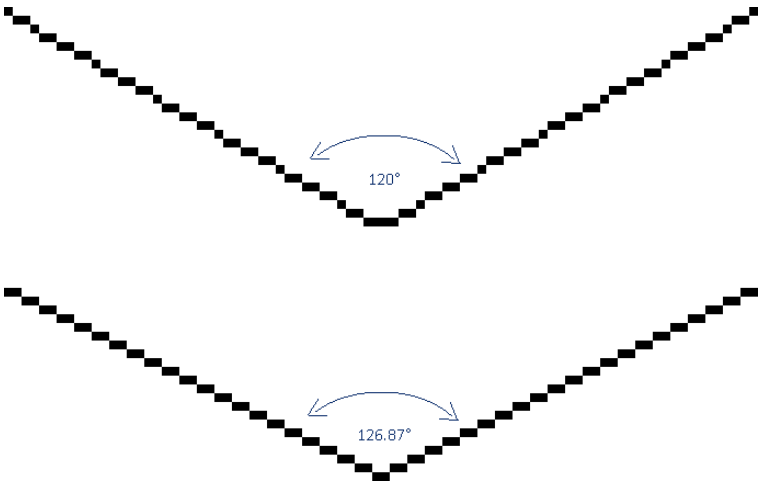

Figura 2.1: Diferencia entre lineas de 120 y 126.87 grados en baja resolución.

A pesar de que esta proyección es una variación de la isométrica, en esta memoria nos referiremos a ella simplemente como proyección isométrica.

2.2. Desarrollo en Unity

A pesar de que el lenguaje principal en el que se programa dentro de Unity es C#, un lenguaje orientado a objetos, todos sus elementos están construidos con una arquitectura orientada a componentes. Esta arquitectura facilita la reutilización de código de manera especialmente efectiva cuando se trabaja en desarrollo de videojuegos.

2.2.1. GameObjects

El bloque básico con el que se construyen todos los elementos que forman parte de un juego en Unity es el GameObject. A él se pueden agregar distintos componentes que le entreguen las funcionalidades necesarias.

Los GameObjects se encargan de mantener una lista de los componentes vinculados a sí mismo y de enviarles mensajes cuando ocurren eventos importantes.

Por ejemplo, si se quiere mostrar un sprite en la pantalla, se debe crear un GameObject y agregarle un componente SpriteRenderer. Este se podrá configurar con la información necesaria y mostrará el sprite en la pantalla.

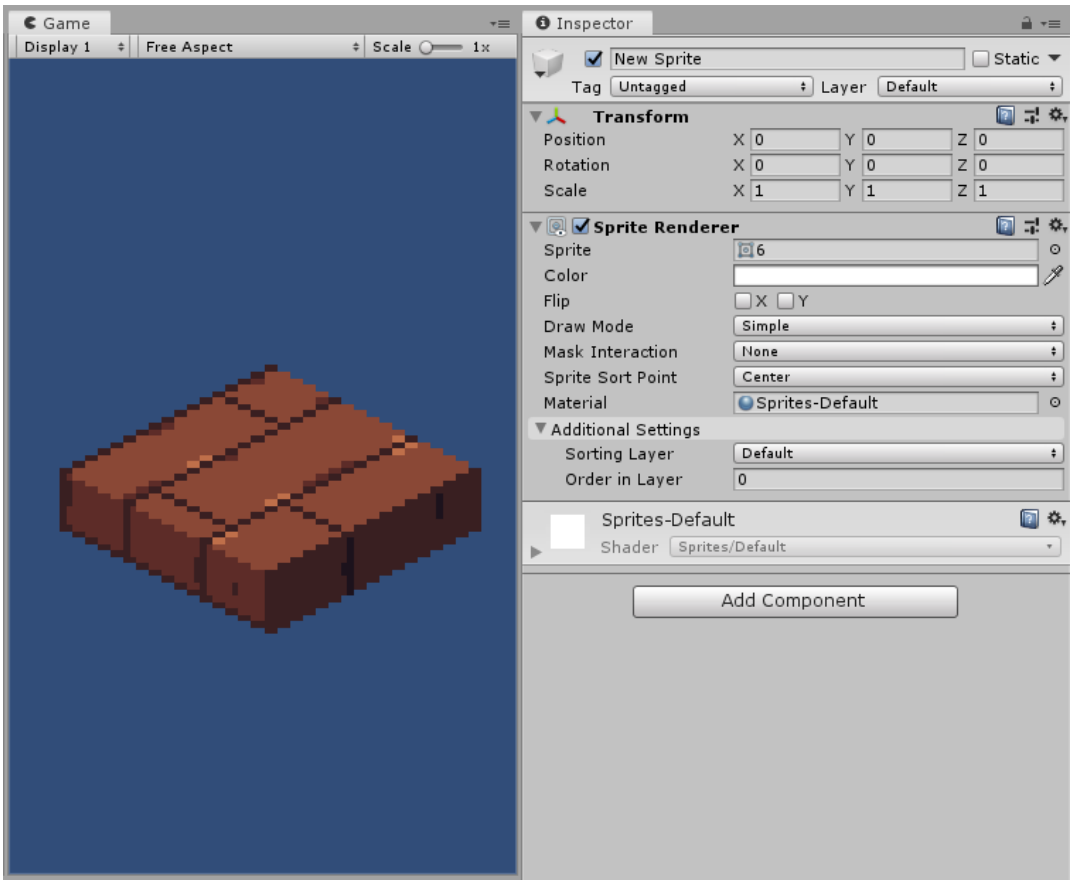

Figura 2.2: Un GameObject con un SpriteRenderer.

Como se puede ver en la Figura 2.2, el GameObject también tiene un componente llamado Transform. Este componente se encarga de posicionar al objeto dentro del espacio del juego

y es automáticamente añadido a todos los GameObjects.

Los componentes son elementos sumamente modulares y reutilizables que, la mayor parte del tiempo, cumplen funciones básicas. Algunos de los componentes más utilizados en Unity son: los Colliders, que permiten detectar colisiones con otros objetos; los RigidBodies, que agregan propiedades físicas al objeto y permiten simularlo; los MeshRenderer, que renderizan mallas geométricas dentro del juego y los Animators, que permiten reproducir animaciones tanto en modelos 3D como en sprites.

2.2.2. Orientación a componentes

Al desarrollar una aplicación basada en componentes, las funcionalidades complejas se separan en varios componentes reutilizables y modulares que pueden interactuar entre ellos. Luego, al momento de necesitar elementos con comportamientos complejos o especializados se agregan varios componentes a un GameObject consiguiendo el funcionamiento necesario por medio de la combinación de estos.

En el ciclo de desarrollo regular de un juego para Unity, los programadores pasan la mayoría de su tiempo escribiendo nuevos componentes que les entreguen las funcionalidades necesarias para sus propios juegos. Para crear un componente nuevo basta con crear una clase que herede de MonoBehaviour, una clase que entrega Unity. Con esto, Unity se encargará enviar los mensajes necesarios al nuevo componente y de serializar sus propiedades públicas para que se vean en la interfaz gráfica. En la Figura 2.2 se puede ver que el componente Transform expone sus propiedades *Position*, *Rotation* y *Scale* mediante el mecanismo de serialización. Por ejemplo, si se necesita crear un auto que se mueva por la pantalla se tendrá que crear un componente que escuche la entrada del usuario y que interactúe con el Rigidbody del GameObject para aplicarle las fuerzas necesarias.

Si el problema a solucionar es más general, se crean componentes más flexibles que puedan ser reutilizados en otros GameObjects. Por ejemplo, si se necesita hacer un sistema de combate, uno de los pasos sería crear un componente "Puntos de Vida" que se pueda agregar tanto al personaje del jugador como a los enemigos y que esté encargado de llevar la cuenta de la cantidad de vida de cada GameObject, de tener funciones para restar y agregar vida de distintas formas y de llamar las funciones correspondientes cuando los puntos de vida sean menores a cero.

En el caso de que el problema sea más general todavía, es probable que existan librerías de componentes que ya implementen las funcionalidades necesarias, las cuales se descargarían para ahorrar trabajo.

Trabajar con componentes de esta manera también permite realizar cambios al proyecto sin tener que tocar el código. Pues con un buen grupo de componentes, un diseñador puede crear un nivel completo combinando componentes de la manera que necesite y editando sus propiedades desde la interfaz de Unity.

2.2.3. Prefabs

Los Prefabs de Unity son un sistema que permite crear, configurar y almacenar un GameObject junto a todos sus componentes y los valores de sus propiedades convirtiéndolo en un recurso reutilizable. El Prefab funciona como una plantilla con la cual se pueden crear instancias de GameObjects preconfigurados.

Si bien los Prefabs son muy utilizados para poblar una escena con GameObjects manualmente, lo interesante para este proyecto es su capacidad de crear GameObjects preconfigurados en tiempo de ejecución.

Como se muestra en la Figura 2.3, los Prefabs pueden ser editados de manera sencilla mediante la interfaz de Unity de manera similar a los GameObjects.

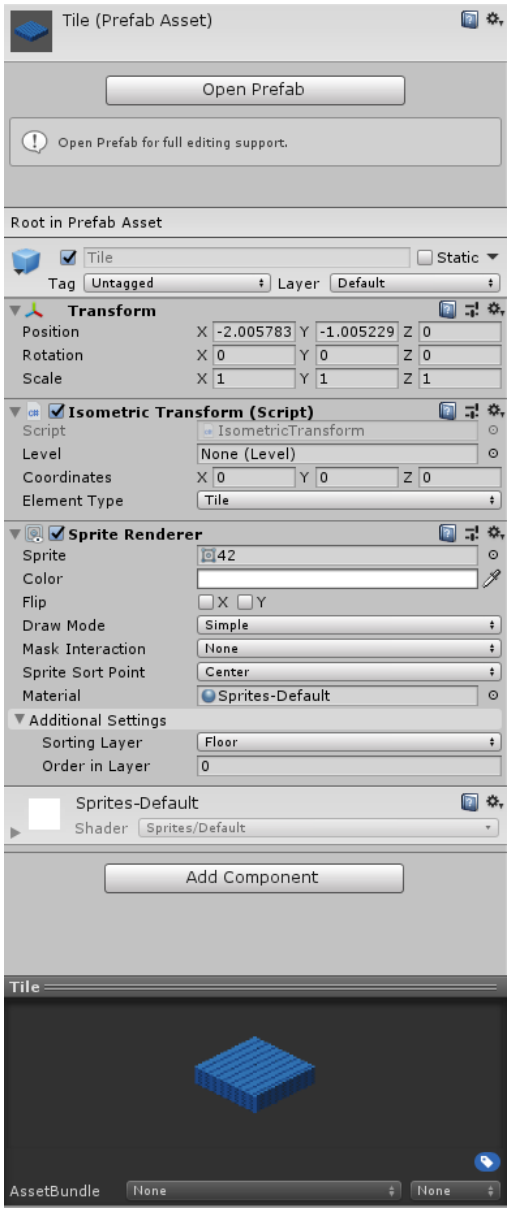

Figura 2.3: Visualización de un Prefab en la interfaz gráfica de Unity.

2.2.4. Espacios de coordenadas en Unity

Unity trabaja con tres sistemas de coordenadas distintas: las coordenadas de pantalla, globales y locales. El primer sistema es principalmente utilizado para programar elementos de la interfaz gráfica de los videojuegos, y los otros dos se usan para posicionar sprites, modelos tridimensionales, luces y otros elementos propios del mundo del juego.

Las coordenadas globales son coordenadas tridimensionales cartesianas con un origen estático, a diferencia de las coordenadas locales que se calculan con respecto a un GameObject. El componente Transform tiene la capacidad de transformar entre coordenadas locales relativas a su propio GameObject y coordenadas globales.

Las coordenadas de pantalla son cartesianas y bidimensionales. A cada píxel de la ventana de juego le corresponde una posición (x, y) y la posición del cursor solo se puede obtener en coordenadas de pantalla. Para transformar entre estas coordenadas y otro tipo se necesita un componente *Camera* para saber que punto del mundo le corresponde a cada píxel en la ventana.

Capítulo 3

Análisis y Diseño

3.1. Casos de uso

El primer paso en este proyecto fue realizar un análisis del problema que se quiere solucionar. Ya que las herramientas pretenden optimizar el proceso de desarrollo de videojuegos isométricos, se analizaron primero los actores que se verán afectados por ellas.

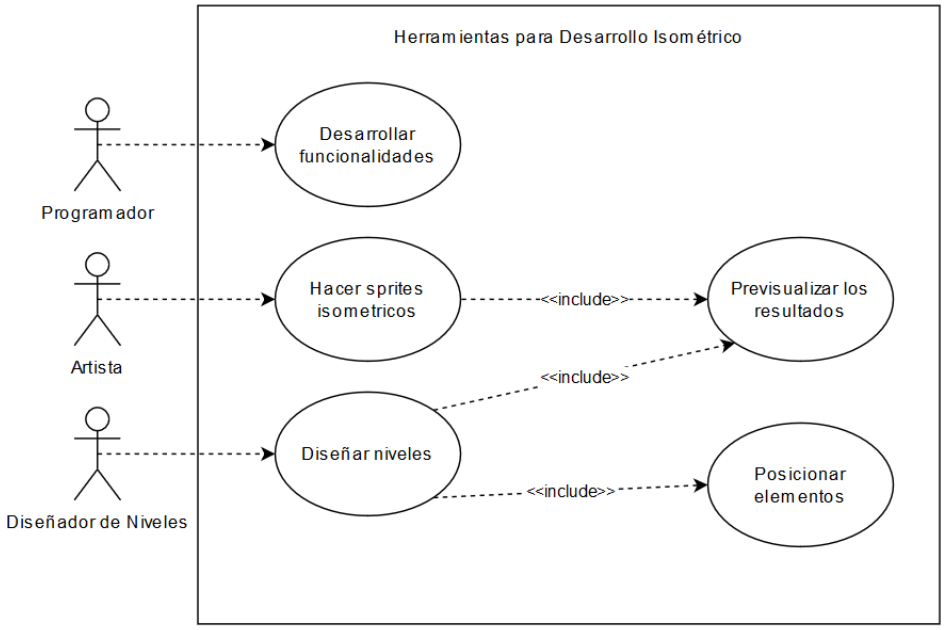

Figura 3.1: Diagrama de casos de uso de las herramientas.

Como se puede ver en la Figura 3.1, el alcance de las herramientas no termina con los programadores. Si bien el foco del proyecto está puesto sobre ellos, también es necesario reconocer que la correcta implementación de las herramientas beneficiará a otros miembros del equipo de desarrollo, como los artistas y los diseñadores de niveles. Con esto en mente, se espera diseñar un producto que permita mejorar el proceso de desarrollo de cada uno de los distintos actores.

3.2. Composición de los Niveles

Es necesario comprender los elementos que componen los niveles isométricos para poder diseñar una solución que los soporte y los agrupe de manera óptima.

Al estudiar los niveles de distintos juegos isométricos se pudieron identificar cuatro elementos que los componen: las baldosas que construyen el piso; las murallas, los personajes y los objetos que van sobre las baldosas, a los que llamaremos "ítems". Cada uno de estos cuatro elementos se posiciona y ordena de manera distinta. Además, cumplen distintas funciones dentro de los niveles como se puede apreciar en la Figura 3.2, por lo que se tratarán como objetos completamente distintos en el código.

Figura 3.2: Captura del juego "Los Sims".

3.3. Arquitectura General

El proyecto se implementará siguiendo el patrón de Modelo, Vista y Controlador. Esto es con el objetivo de facilitar la extensibilidad y modularidad del código. En esta ocasión la vista y el controlador estarán más acoplados debido a que Unity trabaja de esta manera y las ventajas de forzar la separación de estos elementos no son suficientes para justificar el gran costo de hacerlo.

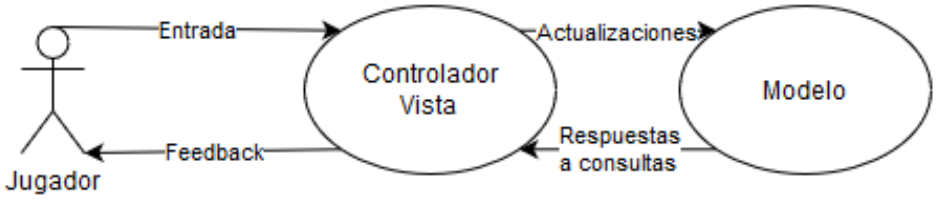

Figura 3.3: Separación entre Controlador-Vista y Modelo.

Se dejó una sola clase como punto de acceso para los desarrolladores que utilicen las herramientas. Esta clase representa un nivel y con ella se podrá interactuar mediante preguntas como ¿Qué hay en las coordenadas (x, y)? o ¿Puedo cruzar desde la baldosa x a su vecino y? Se le podrá pedir que modifique los contenidos del nivel y se podrán registrar callbacks que serán gatillados por eventos como cambios en baldosas o el movimiento de ítems.

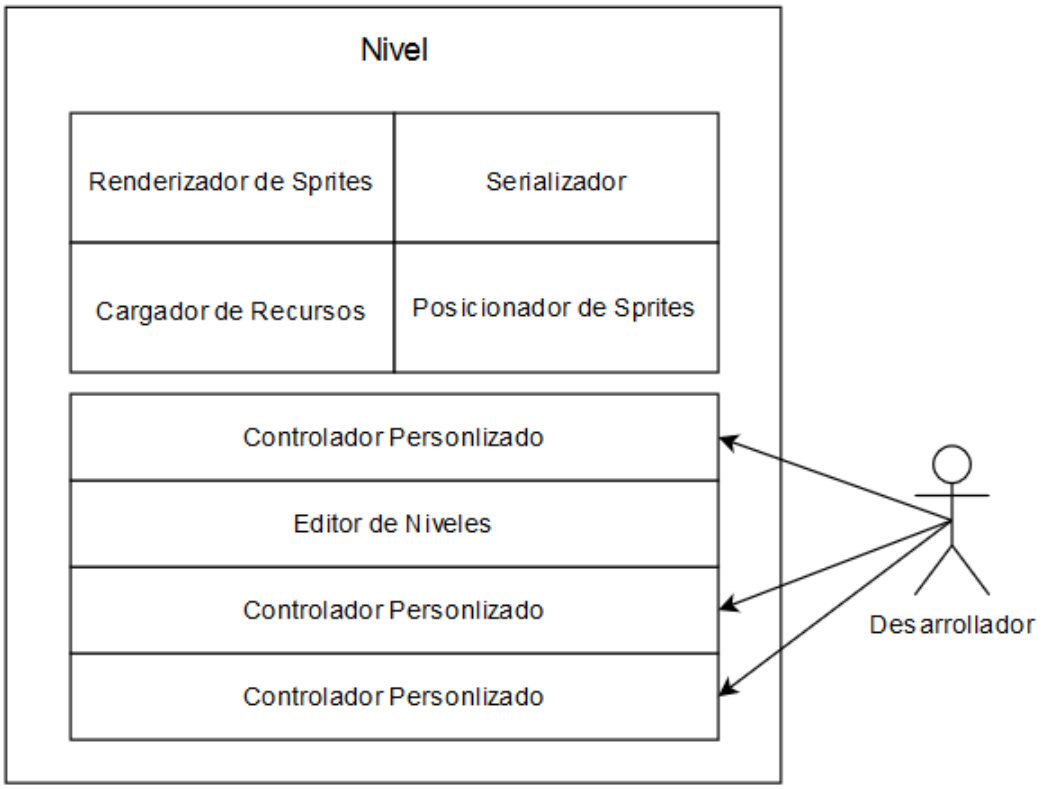

Figura 3.4: Arquitectura de los controladores.

Como se puede ver en la Figura 3.4, los desarrolladores solo tienen que conectar sus controladores al controlador principal y asignar la lógica de sus propios juegos a los callbacks correspondientes.

Todos los controladores que estén conectados al controlador del nivel estarán escuchando los eventos de este último. De esta forma todo el flujo será manejado por el controlador de nivel, mientras que los demás reaccionaran a los mensajes que este les envíe.

Capítulo 4

Implementación

4.1. Modelamiento de Niveles

La unidad fundamental de los niveles es la baldosa, por lo que es esencial modelarla al comienzo. Sus características más importantes son su posición, que será representada por dos enteros, y su tipo, que será representado por otro entero. El tipo sirve como identificador. En un juego sencillo, éste solamente se utilizará para asociar cada baldosa a su sprite correspondiente, pero en juegos más complejos, cada tipo de baldosa puede tener distintos efectos y lógica personalizada para cada uno de sus callbacks.

Entonces, los niveles se modelan como una clase con un arreglo bidimensional de baldosas.

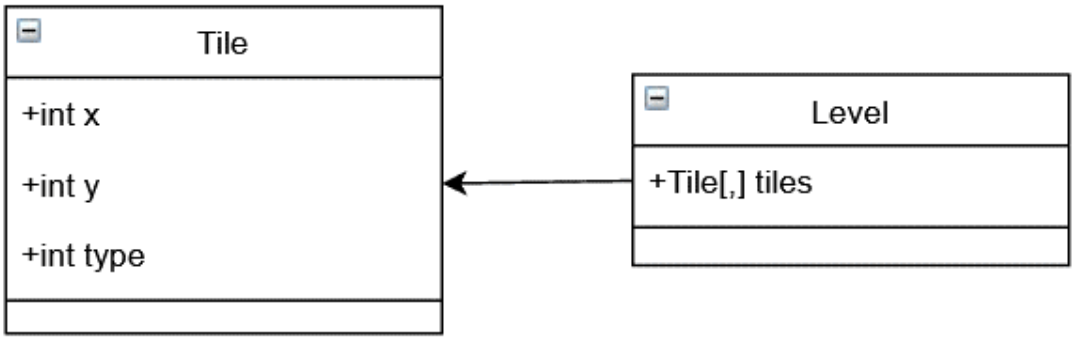

Figura 4.1: UML de la primera versión del modelo.

4.2. Posicionamiento de baldosas

Para posicionar las baldosas es necesario transformar de coordenadas isométricas a coordenadas cartesianas, como se muestra en la Figura 4.2. Para esto se creó una clase estática con todas las operaciones que pueden ser necesarias para transformar entre éstas.

Con la siguiente fórmula es posible encontrar el equivalente entre ambos tipos de coordenadas en tiempo constante, sin depender del tamaño del nivel ni la cantidad de baldosas

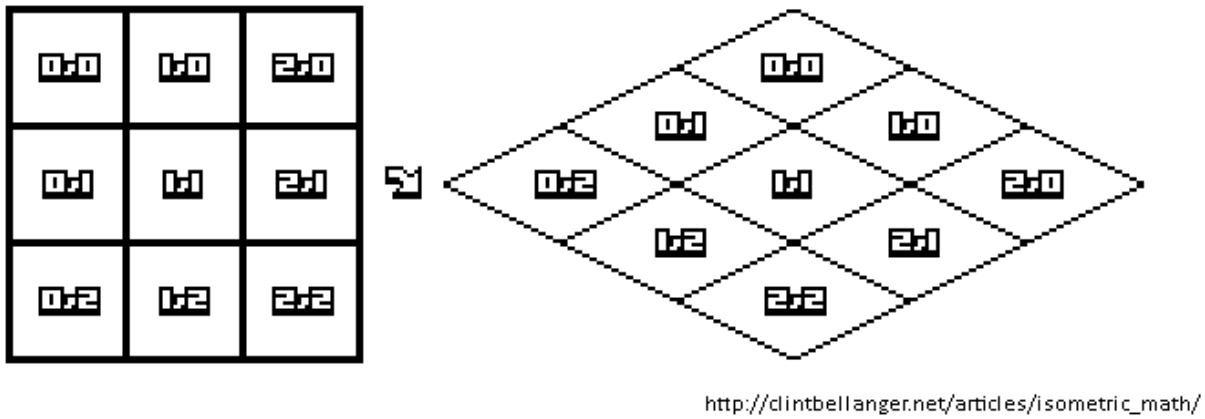

Figura 4.2: Transformación entre coordenadas cartesianas e isométricas.

existentes.

$$cartesian_x = \frac{\mathrm{i}sometric_x * tileWidth - \mathrm{i}sometric_y * tileWidth}{2}$$

$$cartesian_y = \frac{\mathrm{i}sometric_x * tileHeight + \mathrm{i}sometric_y * tileHeight}{2}$$

Unity trabaja también con otro sistema de coordenadas al que llamaremos coordenadas de pantalla. En este sistema cada par (x,y) representa un pixel en la pantalla. Ya que Unity entrega la posición del cursor en coordenadas de pantalla, se agregaron métodos para transformar este tipo de coordenadas a cartesianas e isométricas para permitir, por ejemplo, saber que baldosa está debajo del cursor.

4.3. Observador de sprites

Si bien tenemos una clase que nos permite encontrar la posición en la que deben ir los sprites, es necesario tener otra clase encargada de actualizarlos y posicionarlos todas las veces que sea necesario.

Para evitar llamados a funciones y sobrecarga innecesaria se utilizará el patrón de diseño Observer de manera que solo se realicen actualizaciones cuando una baldosa sea efectivamente modificada. Esto además permite mantener la separación entre modelo y controlador, pues el modelo implementa los callbacks, pero no está acoplado a ninguna funcionalidad específica del controlador.

El observador de sprites también se encarga de asignar el orden de renderizado a cada uno de los sprites. Como cada tipo de elemento que forma parte del nivel utiliza una manera levemente distinta de calcular su orden de renderizado, se creó una clase encargada de asignarlo a cada objeto.

En el caso de las baldosas, el orden de renderizado es simplemente el siguiente:

$$orden = \text{i}sometric_x + \text{i}sometric + y$$

4.4. Controlador de nivel

El último componente necesario para probar por primera vez la visualización de una escena es el controlador de nivel. Como se mencionó antes este es el punto de acceso único al nivel, pues todas las demás clases, propiedades y métodos fueron construidos con acceso restringido para que los desarrolladores no puedan interferir accidentalmente con el flujo de las herramientas.

Para esta primera etapa, el controlador del nivel se encargará de inicializar un nivel de 5x5 baldosas, con un tipo aleatorio entre 0 y 3. Además, se encargará de conectar el observador de sprites con el modelo del nivel.

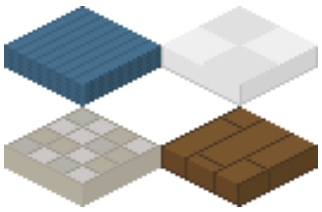

Figura 4.3: Sprites de baldosas.

Se crearon cuatro sprites de baldosas y se ingresó su tamaño en la clase encargada de las transformaciones entre coordenadas. Al cargarlos en el controlador de nivel se obtuvo el siguiente resultado:

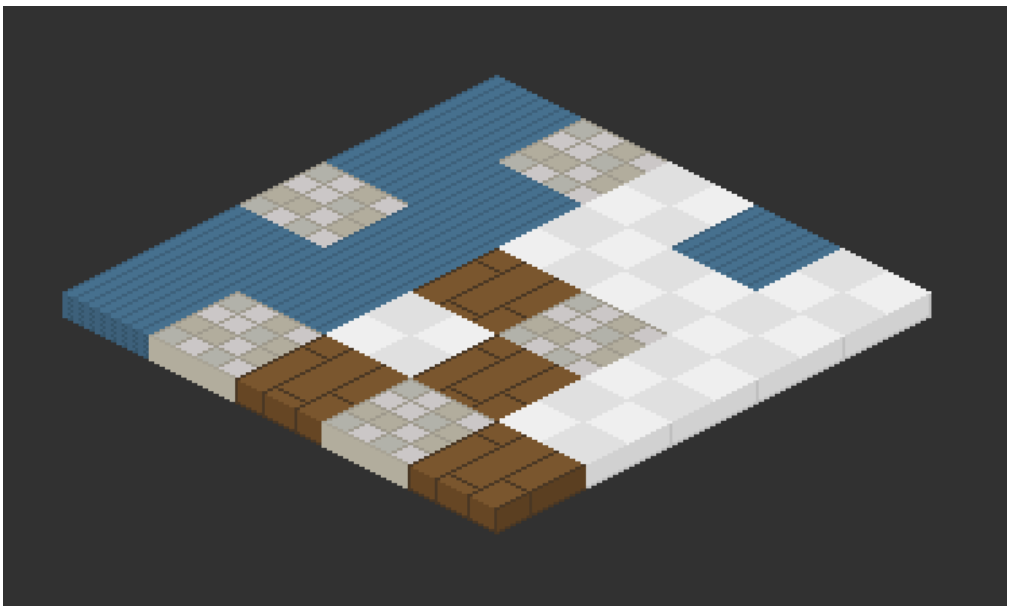

Figura 4.4: Primera visualización de una escena isométrica.

Cabe mencionar que, incluso en esta etapa temprana del proyecto, ya se presentan ventajas con respecto a las herramientas ya existentes. Principalmente la capacidad de poder modificar baldosas indicando solamente su posición en coordenadas isométricas es algo que no era posible realizar con otras herramientas.

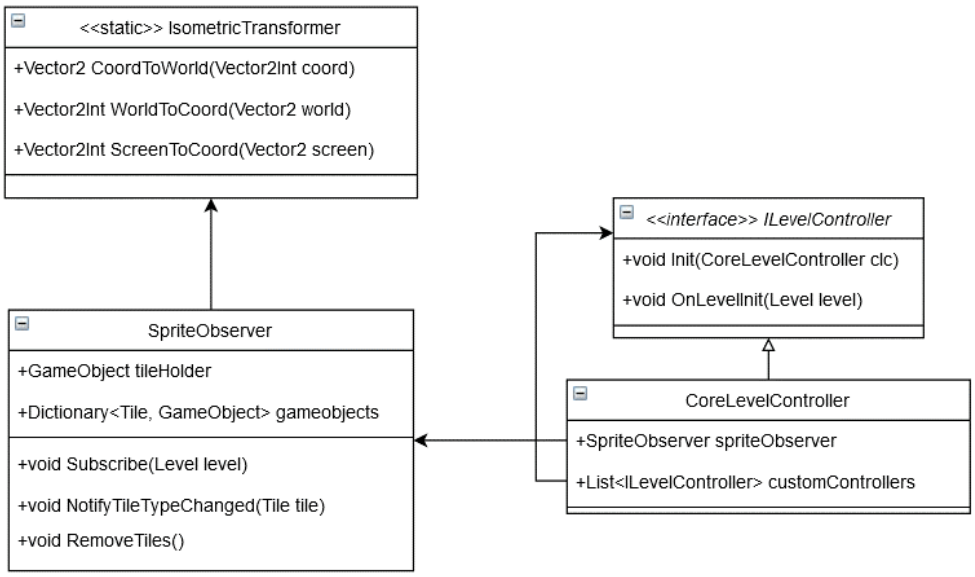

Figura 4.5: UML de los principales elementos relacionados con el renderizado y posicionamiento de sprites.

4.5. Carga de recursos

Hasta el momento, la carga de recursos, como los sprites de las baldosas, se ha realizado especificando la ruta en la que se encuentra cada uno de ellos de manera manual. Se creará un sistema que se encargara de automatizar la generación de rutas y la carga de recursos de tal manera que el desarrollador, y las otras herramientas, puedan pedir a una clase los recursos necesarios sin saber dónde se encuentran.

El agregar una capa de abstracción como esta entre el proceso de carga y sus invocaciones, nos permite asegurarnos de que no se cargará el mismo sprite más de una vez en diferentes posiciones de memoria, optimizando así el uso de esta y mejorando el rendimiento.

4.6. Cámara *Pixel Perfect*

Ya que una cantidad significativa de los juegos que se desarrollan utilizando proyección isométrica utilizan imágenes de baja resolución para obtener una estética retro, es necesario entregar la opción de no utilizar antialiasing.

En Unity, esto es un poco más complejo que simplemente desactivarlo, pues el motor no está diseñado para juegos con sprites de baja resolución. Unity trabaja con la analogía de "cámaras", donde el contenido renderizado en la ventana de juego es el contenido que está observando la cámara.

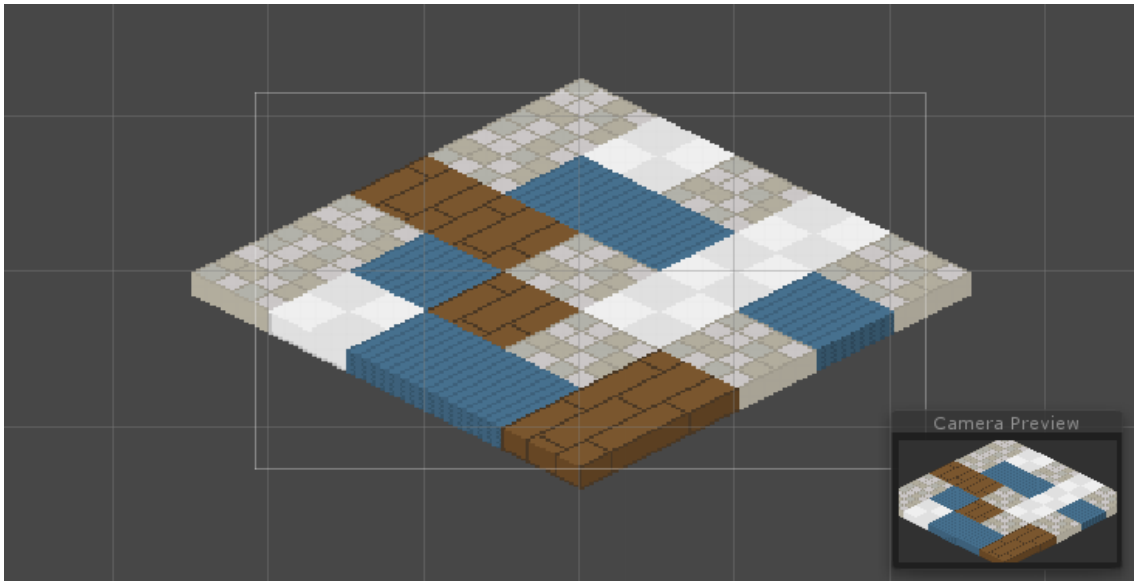

Figura 4.6: Visualización del alcance de una cámara y la previsualización de la ventana del juego.

En la Figura 4.6 se puede apreciar el funcionamiento de una cámara: el rectángulo blanco que rodea las baldosas es el área que está observando la cámara y en la esquina inferior derecha se puede observar el resultado que se renderizaría en la ventana del juego.

Como las cámaras pueden definir el tamaño del área que observarán, pueden renderizar una imagen de 50 pixeles de largo en una pantalla de 70 pixeles de largo, por ejemplo, obteniendo un resultado en el que algunos pixeles son escalados y otros no.

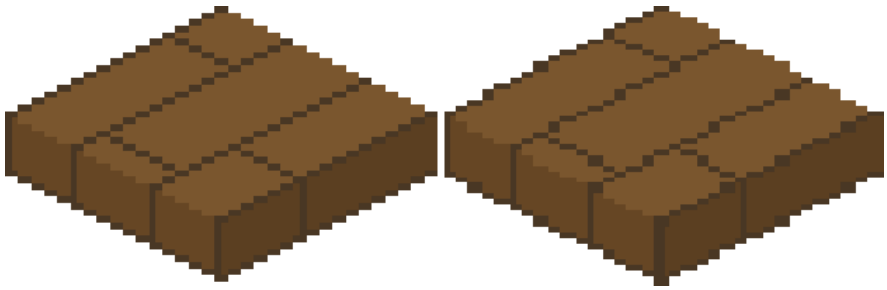

Figura 4.7: Comparación entre una imagen de 66x42 pixeles y la misma imagen escalada a 80x51 pixeles.

Con un buen entendimiento del problema no es difícil llegar a la solución: restringir el tamaño de la cámara a alguno que renderice una cantidad de pixeles igual a la del tamaño de la ventana de juego o esa misma cantidad dividida por una potencia de dos.

Al implementar esto se evitaron las fallas gráficas sin dejar de permitir distintos niveles de aumento.

4.7. Serialización

Un factor que afectará de manera importante la extensibilidad de las herramientas es la capacidad de cargar niveles en distintos formatos. Por esto se creó un sistema de serialización con una interfaz común, para así permitir a los desarrolladores crear sus propios serializadores de acuerdo con sus necesidades individuales. Se implementó, además, un serializador por defecto que permite cargar y guardar niveles en formato binario, permitiendo la persistencia de estos entre distintas ejecuciones del programa.

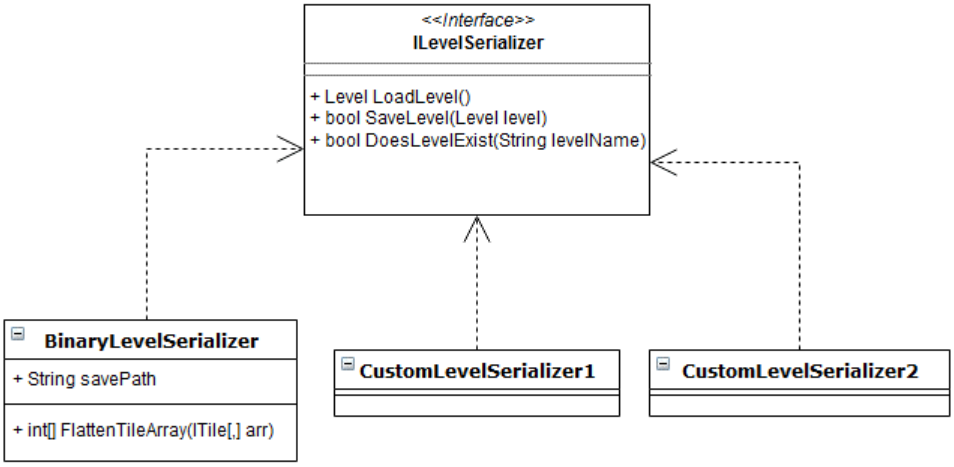

Figura 4.8: Diagrama de clases de los serializadores.

4.8. Rotación

Otro elemento que es necesario en varios juegos isométricos es la capacidad de rotar el nivel para verlo desde distintos lados. Se creo una clase estatica encargada de informar la orientación actual del controlador (norte, sur, este u oeste) y de proveer métodos para operar entre orientaciones. Luego, se hizo que la clase encargada de realizar las transformaciones isométricas, que fue creada anteriormente, considere la orientación para calcular las posiciones de las baldosas.

Para considerar la rotación se rotaron las coordenadas isométricas de la misma manera que aplican rotaciones en coordenadas cartesianas. Luego lo último que se debe hacer es notificar al observador de sprites cuando ocurra un cambio de orientación para que actualice todas las baldosas.

Al incluir estos nuevos elementos (rotación, serialización, carga centralizada de recursos y cámara "pixel perfect"), el diagrama de clases que se tiene hasta el momento se ve como se muestra en la Figura 4.10.

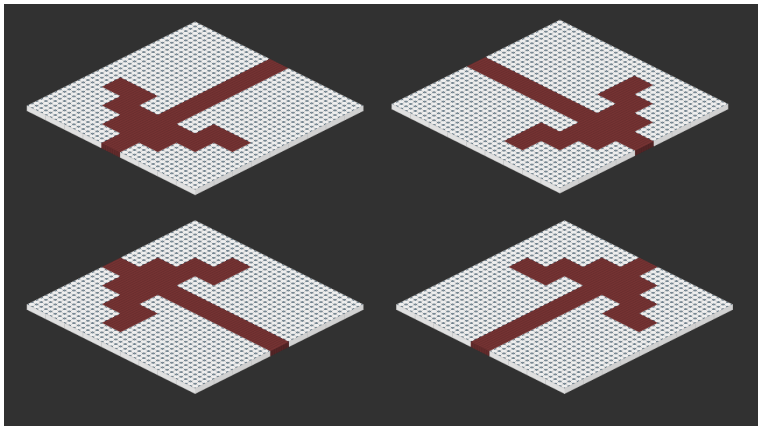

Figura 4.9: El mismo nivel visualizado desde cuatro orientaciones distintas.

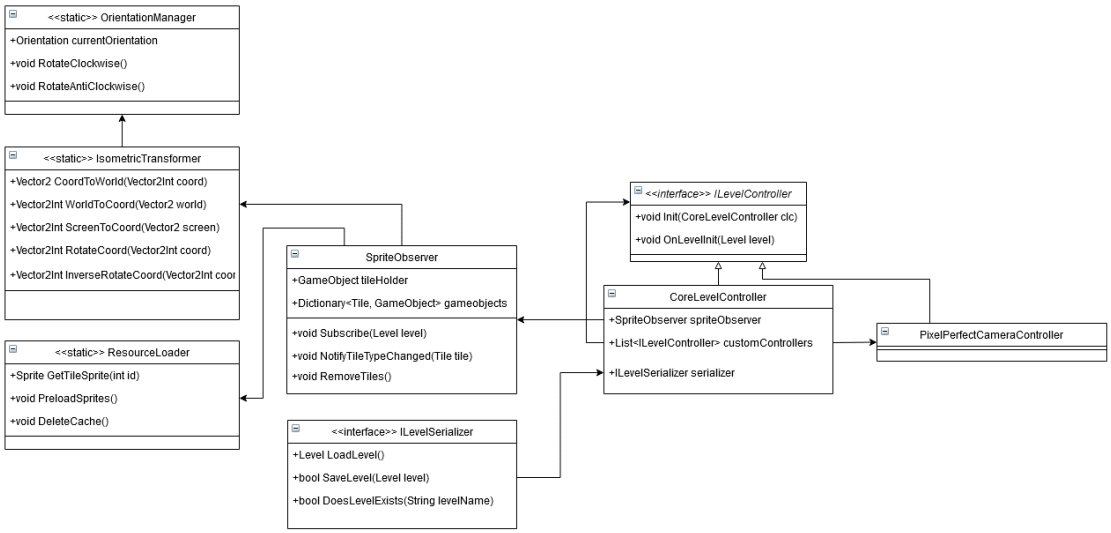

Figura 4.10: UML de los elementos de renderizado y posicionamiento de sprites incluyendo rotación, serializado de niveles y carga de recursos.

4.9. Posicionamiento de murallas

Las murallas son elementos que se posicionan en el borde de una baldosa o entre dos de ellas, por lo que no es posible reutilizar el sistema de posicionamiento de baldosas con las murallas.

Un mismo diseño de muralla debe usar dos sprites distintos, pues si se encuentra en el borde superior izquierdo de una baldosa o en el borde inferior derecho, se verá como en la Figura 4.11.a, mientras que si está en uno de los otros bordes se verá como en la Figura 4.11.b.

Para obtener el posicionamiento deseado lo primero fue agregar las murallas al modelo. Se decidió utilizar un arreglo tridimensional en el que a cada baldosa le corresponden las dos murallas que están en sus bordes superiores. De esta forma si necesito acceder a la muralla que se encuentra en el borde superior izquierdo de la baldosa (x,y) tengo que acceder al

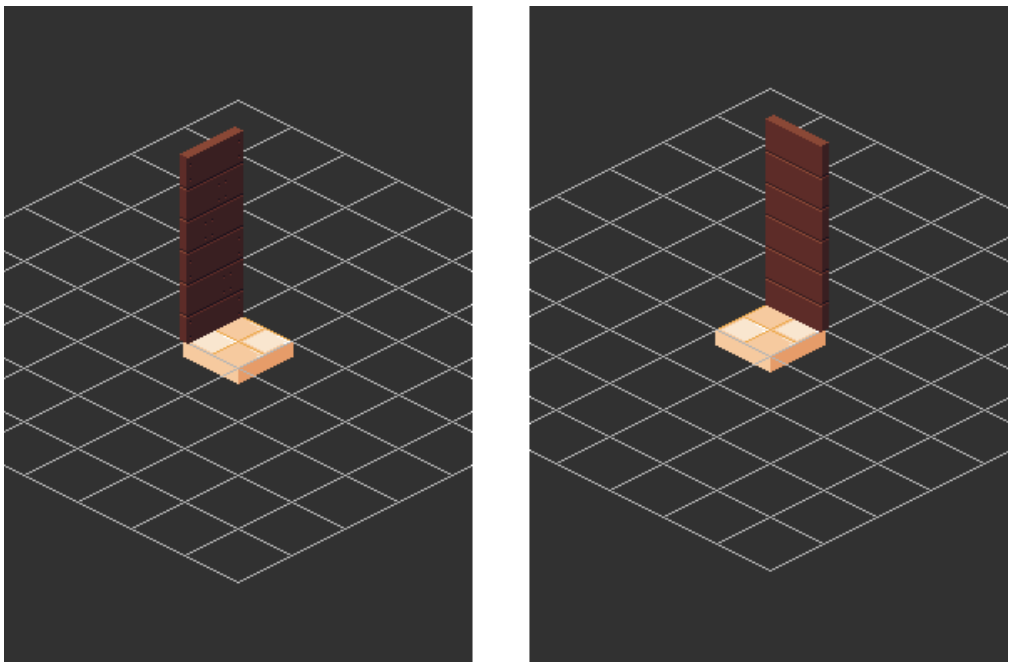

Figura 4.11: Uso de dos sprites distintos para una misma muralla. En el costado izquierdo (a) hay una muralla en el borde superior izquierdo de la baldosa y en el costado derecho de la Figura (b) hay una muralla en el borde superior derecho.

arreglo de murallas con los índices (x,y,0), y los índices (x,y,1) para la del borde superior derecho.

Pero esta forma de modelar las murallas tiene un problema: si las murallas del borde inferior pueden tener otra al lado, se podrán agregar murallas en lugares donde no debería. Como se muestra en la Figura 4.12, ninguna de las murallas blancas debería poderse agregar.

Para evitar esto se implementó un tipo de muralla siguiendo el patrón de diseño "Null Pattern". Esta muralla comparte la interfaz de las murallas regulares, pero siempre se mantiene vacía.

Con esto solucionado, se agregaron métodos para realizar operaciones entre coordenadas cartesianas, isométricas de murallas y de pantalla en una clase similar a la encargada de hacer esto mismo para las baldosas. También se implementó un observador de sprites y se hizo que el controlador del nivel lo conecte con el modelo de manera análoga a como se hace con las baldosas.

4.10. Sprites de bordes de murallas

Otra dificultad del manejo de murallas es que sus sprites no calzan perfectamente con los otros. Esto puede parecer un detalle insignificante, pero en juegos con imágenes de resoluciones muy bajas esto es importante, pues es muy notorio.

Como se muestra en el aumento realizado en la Figura 4.13, los sectores donde las murallas se intersecan presentan fallos gráficos que hacen parecer que las murallas se atraviesan.

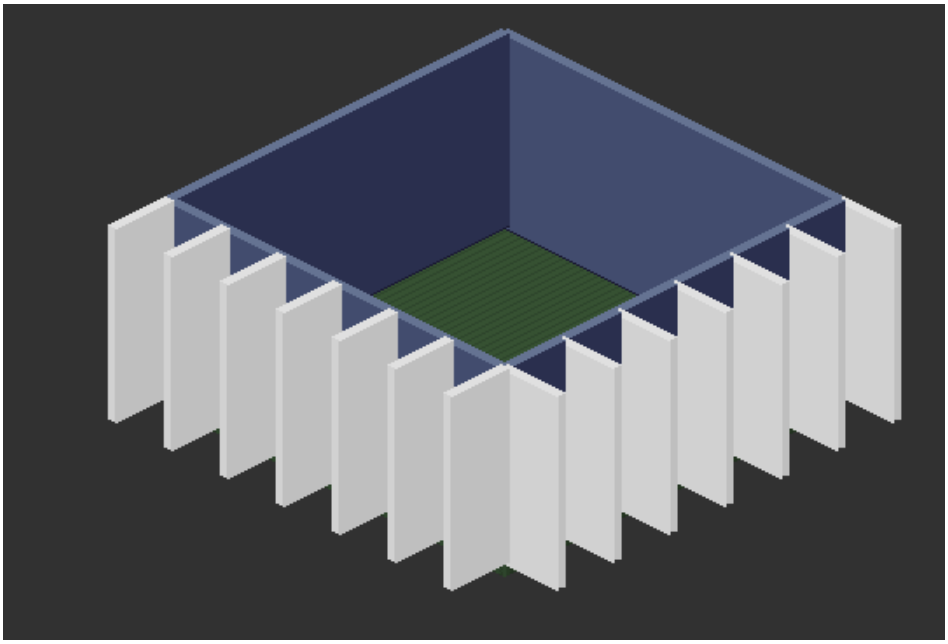

Figura 4.12: Problema del modelar las murallas respecto a su baldosa base.

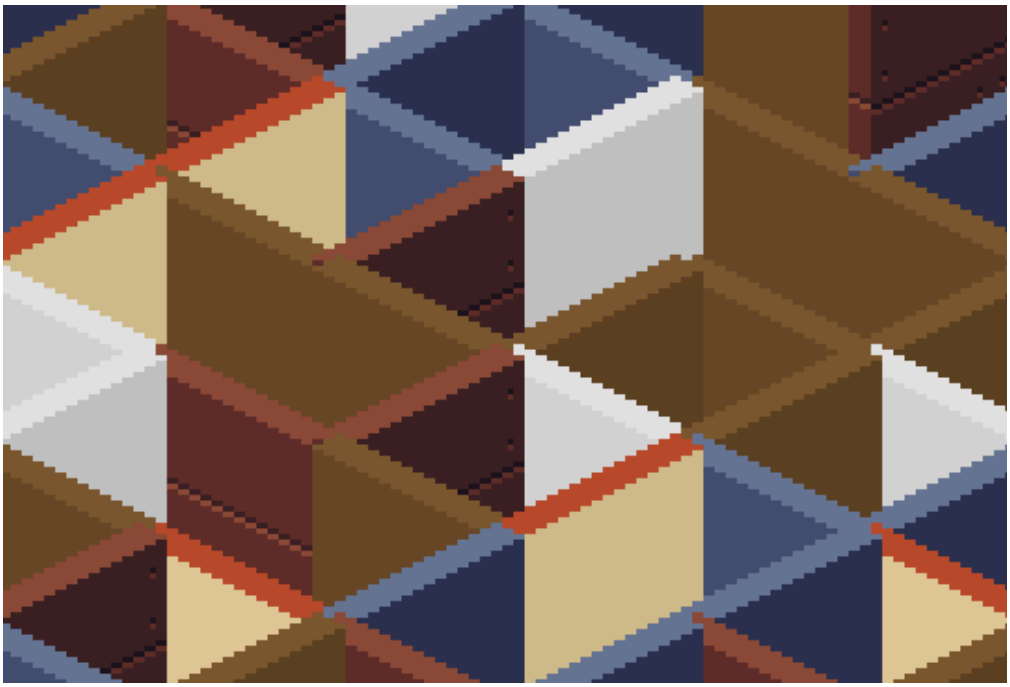

Figura 4.13: Errores gráficos en las intersecciones de murallas.

Como los pixeles que hay que agregar o quitar para obtener una visualización mas limpia en cada uno de los sprites dependen de los tres vecinos que tiene la muralla en cada extremo, se crearon mascaras para cada una de las $2 * 2^3$ combinaciones.

Para modificar el sprite utilizando la mascara se creó una capa intermedia entre los sprites y el observador, de manera que cuando el observador necesite un sprite, este se calcule utilizando la máscara y modificando la imagen a renderizar. Para saber qué máscara utilizar

se tuvieron que agregar métodos que permitan obtener de manera sencilla los vecinos de cada muralla.

En la Figura 4.14 se muestra la diferencia entre el renderizado de murallas original y el resultado luego de aplicar las máscaras. Se puede ver claramente que en la Figura 4.14.b los bordes tienen una apariencia mucho más limpia.

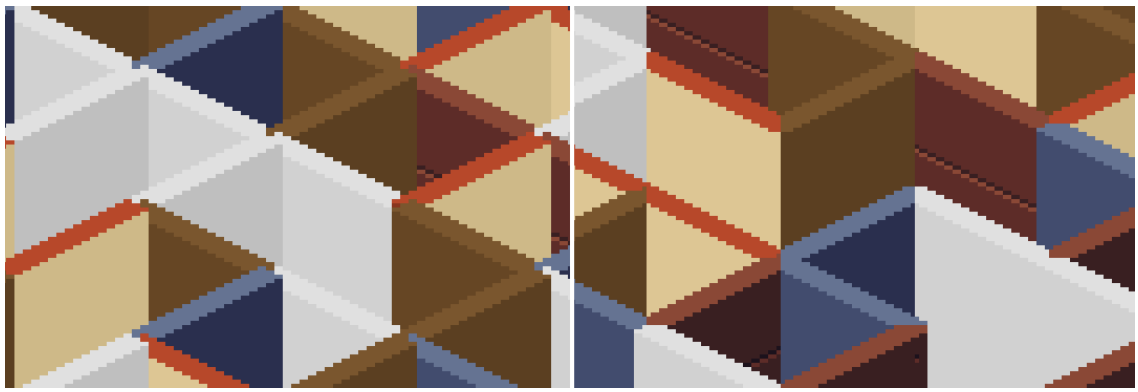

Figura 4.14: Diferencia entre intersecciones de murallas antes y después de aplicar las máscaras correctoras. Al lado izquierdo de la Figura (a) se muestra el antes y al lado derecho (b) el después.

En la Figura 4.15 se puede ver el proceso de carga de recursos y como los sprites de las murallas son modificados antes de llegar al observador.

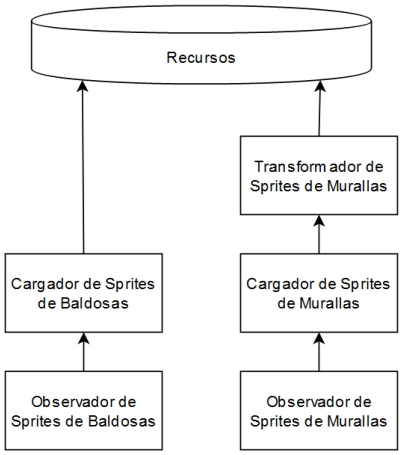

Figura 4.15: Proceso de carga y modificación de sprites.

4.11. Recorte de murallas

Como las murallas ocupan un espacio tan grande y no permiten ver con claridad lo que está detrás de ellas, muchos juegos optan por permitir al jugador recortar las murallas. De esta manera, se puede ver lo que hay detrás sin perder la información de donde están las murallas.

Para implementar esta funcionalidad, se hace que la misma clase que agrega la máscara a los sprites de las murallas pueda recortarlos. Esto es simplemente tomar una porción del área superior del sprite y moverlo hacia abajo.

La clase encargada de observar los sprites de las murallas mantendrá una variable de estado para saber si debe solicitar que los sprites sean recortados.

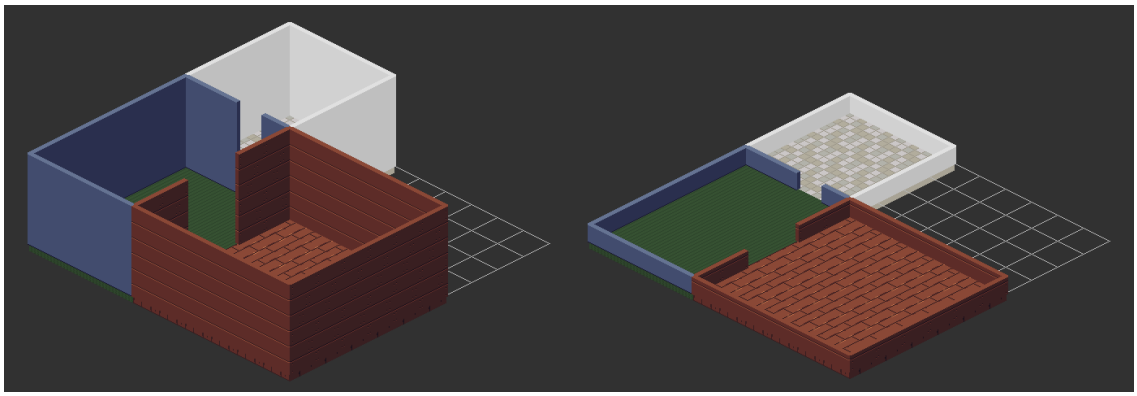

Figura 4.16: Un mismo nivel con las murallas normales y recortadas.

4.12. Rotación de murallas

Otro aspecto en el que las murallas son más complejas que las baldosas es la rotación. Pues la forma de modelarlas es respecto a la baldosa que está debajo suyo, pero al cambiar la orientación puede que su baldosa base no se vea debajo suyo. Además, el sprite de la muralla cambia dependiendo de la orientación desde la cual se observa.

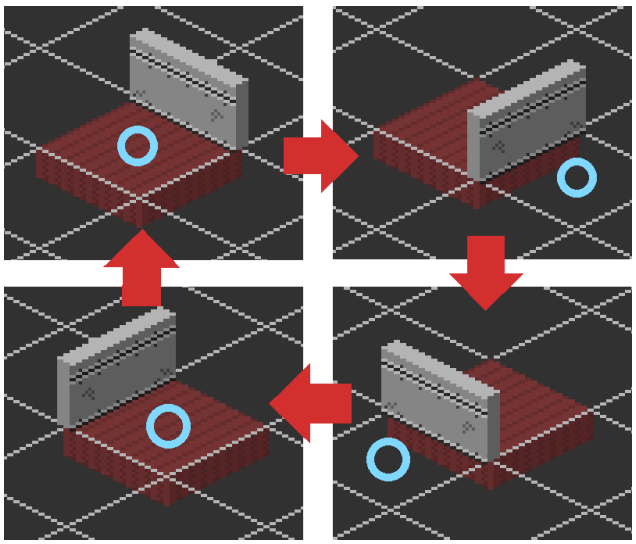

Figura 4.17: Una muralla, su baldosa base (en rojo) y la baldosa que se ve debajo suyo (en celeste).

Como se puede ver en la Figura 4.17, en solo dos de las cuatro orientaciones la baldosa base del modelo corresponde a la baldosa base de la visualización. Esto se tuvo que considerar

en la clase encargada de transformar coordenadas isométricas para las murallas, pues esta debe modificar la posición de su baldosa base dependiendo de la orientación actual y el lado que utiliza la muralla en el modelo.

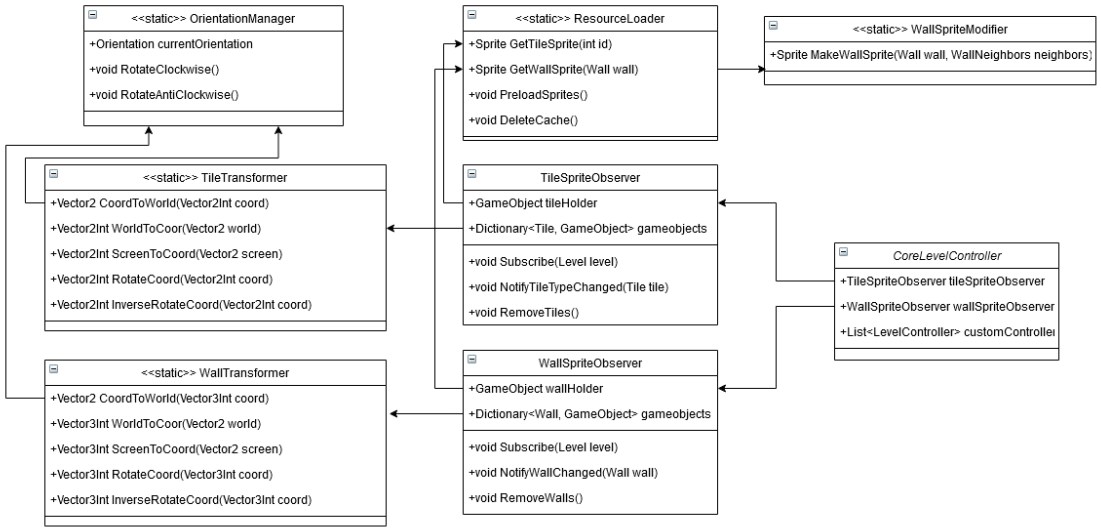

Figura 4.18: UML con la comparación entre las clases para baldosas y murallas.

Se puede ver en la Figura 4.18 que se mantiene el mismo patrón que con las baldosas, pero se implementan clases especiales para los requerimientos específicos de las murallas.

4.13. Posicionamiento de ítems

El tercer elemento que forma parte de los niveles isométricos son los ítems. Estos representan cualquier otro objeto que vaya sobre una baldosa. Ya que se posicionan de la misma forma que las baldosas, es posible reutilizar su clase encargada de transformar entre coordenadas isométricas y cartesianas.

Cabe mencionar que, aunque existan ítems con alturas diferentes, esto no produce ningún problema ya que los sprites se posicionan desde su esquina inferior izquierda. De esta forma siempre se puede utilizar la misma posición para cualquier ítem independiente de su altura.

4.14. Observador de sprites de ítems

La diferencia que tienen los ítems con respecto a los otros elementos con los que se ha trabajado, es que pueden existir ítems que necesiten un sprite diferente para cada uno de sus lados y otros que utilicen el mismo para todos ellos. Por ejemplo en la Figura 4.19 se puede ver que la planta utiliza el mismo sprite para sus cuatro orientaciones, mientras que la caja gris con azul necesita cuatro sprites diferentes.

Para permitir flexibilidad en la cantidad de sprites que utilizará cada uno de los ítems, se creó una interfaz que recibe la orientación del nivel junto a la orientación del ítem y entrega el sprite que debe mostrarse. De esta manera se pudieron crear clases contenedoras de sprites

que responden de manera adecuada a los requerimientos de cada ítem. Esto implica, que para la carga de sprites de ítems, el flujo es análogo al de la carga de sprites de las murallas: existe una clase intermedia que entrega las versiones específicas dependiendo del contexto.

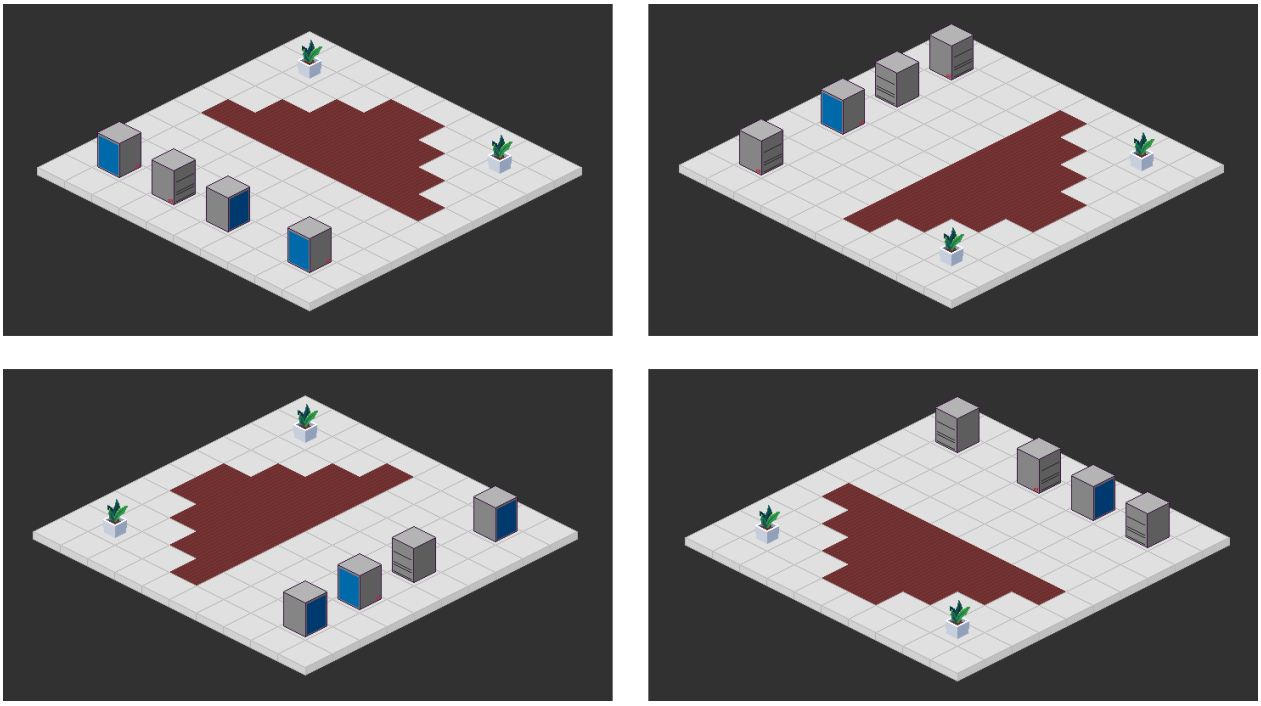

Figura 4.19: Ítems con un solo sprite para todas las orientaciones (plantas) junto a ítems con un sprite distinto para cada orientación (cajas grises) vistos desde distintos lados.

4.15. Editor de niveles

Luego de tener lista la visualización de niveles, se quiso dar un paso más para facilitar la interacción de los desarrolladores con el nivel. Para esto se creó un editor de niveles que funciona en tiempo de ejecución. Su objetivo es no solo ayudar a los desarrolladores a hacer niveles de manera más fácil, sino que también permitirles reutilizar algunos de sus componentes si desean permitir al jugador modificar los niveles.

4.15.1. Modificaciones al modelo

Para permitir al editor un comportamiento más inteligente, se decidió encapsular las modificaciones al nivel dentro de operaciones que siguen el patrón de diseño "Command". De esta manera se puede llevar un historial de modificaciones, el cual permitió implementar funcionalidades como "deshacer" y "rehacer".

Utilizar este patrón también permitió establecer la diferencia entre pintar elementos y construirlos. Para el sistema no hay diferencia entre estas dos operaciones, pues tanto el construir una baldosa como el pintarla es cambiar su tipo, el sistema solo asigna un sprite nulo a aquellos elementos cuyo tipo sea 0. Con el patrón "Command" es posible establecer reglas en las distintas operaciones de tal manera que el comando "Pintar" no permita cambiar el tipo de un elemento si este es 0 y tampoco permita asignarle un 0.

Luego, se implementaron interfaces para los comandos, de esta manera los desarrolladores podrán crear comandos que se ajusten a la lógica de su juego.

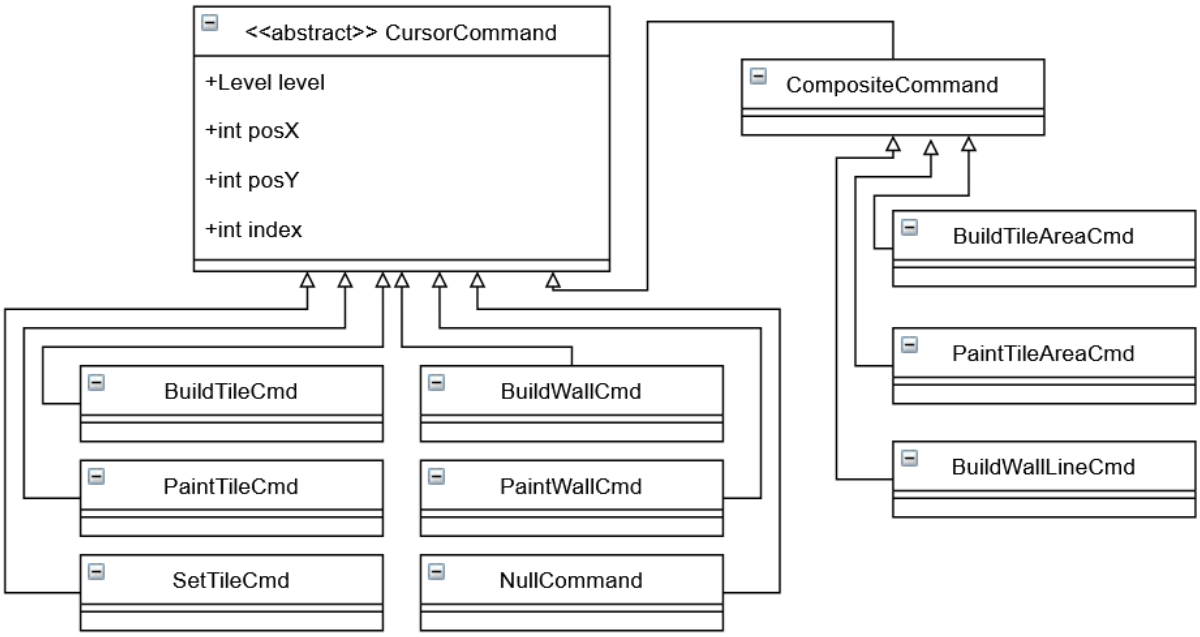

Figura 4.20: UML de los comandos de edición de niveles.

4.15.2. Modo de uso

Para separar de manera amigable las distintas operaciones que se pueden realizar, se agregó un panel que permite escoger entre cinco modos de edición del nivel: construir baldosas, construir murallas, pintar baldosas, pintar murallas y agregar ítems. Dependiendo del modo seleccionado cambiará el comportamiento del cursor.

Se creó un controlador que se conectará al controlador principal del nivel, de la forma ilustrada en la Figura 3.4. Este controlador ejecutará los comandos correspondientes al modo de edición activo y los guardará en un historial para poder deshacerlos.

4.15.3. Baldosas fantasma

Como el editor soporta operaciones sobre varios elementos a la vez, se decidió agregar baldosas "fantasma" cuando el usuario arrastre el cursor, para entregarle feedback de que es lo que está por hacer, como se muestra en la Figura 4.21.

Esto produce un problema: se están creando y destruyendo muchas baldosas fantasmas al modificar grandes porciones del mapa, y crear o destruir objetos es un proceso muy caro.

Para solucionar esto se utilizó un sistema de "pooling" que permite precargar una cantidad determinada de baldosas fantasma y luego encenderlas o apagarlas en vez de crearlas o destruirlas.

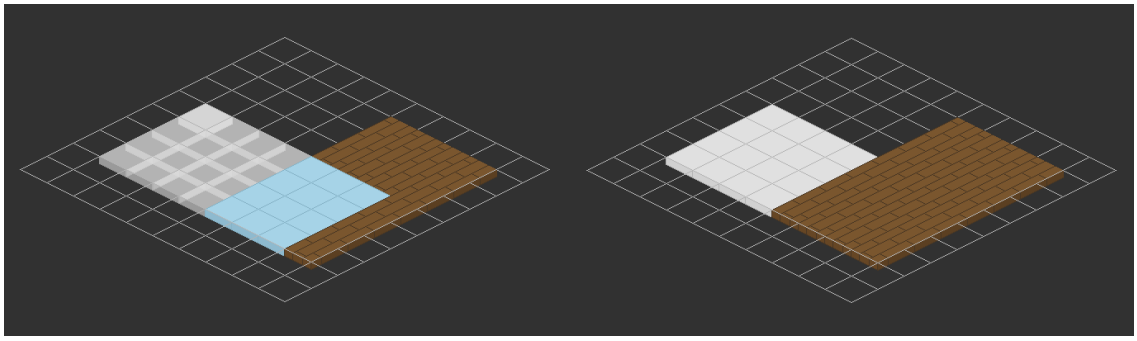

Figura 4.21: En el costado izquierdo de la Figura (a) el usuario arrastró el cursor creando baldosas fantasmas. Al costado derecho (b) el usuario soltó el botón del cursor construyendo las baldosas faltantes.

Los objetos en Unity pueden ser desactivados, con lo que dejan de recibir actualizaciones y su costo de procesamiento es muy cercano a cero.

Como se puede ver en la Figura 4.22, al crear y destruir constantemente 100 baldosas fantasmas, cada ciclo del juego tarda más de 40 milisegundos en computarse, mientras que con el sistema de pooling cada ciclo tarda menos de 2 milisegundos. Esto muestra un aumento de rendimiento importante.

Figura 4.22: Diferencia de rendimiento causada por el uso de pooling al crear y destruir 100 baldosas cada frame. La imagen superior (a) muestra los resultados sin pooling y la inferior (b) los resultados con pooling.

Cabe mencionar que utilizar este método requiere mantener más objetos en memoria, pero el espacio extra que se utiliza es insignificante. Además, de ser necesario en el futuro,

se pueden implementar sistemas que liberen esa memoria cuando sea poco probable que se requieran las baldosas fantasmas de nuevo (por ejemplo, al cambiar de modo de edición o cuando las baldosas no se hayan activado un determinado tiempo).

4.15.4. Construcción de murallas

La última dificultad para implementar el editor de niveles estuvo relacionada a la construcción de murallas, pues para los usuarios, la forma intuitiva de construirlas es seleccionando un vértice de una baldosa y arrastrando el cursor por otros vértices para dibujar la muralla deseada.

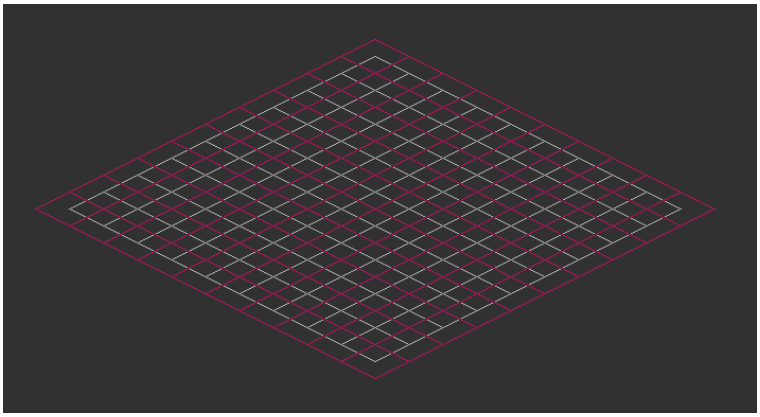

Figura 4.23: Áreas de selección de vértices de baldosas.

El problema es que no se tienen métodos para buscar el vértice más cercano al cursor. Pero para ello basta con utilizar el mismo algoritmo para encontrar baldosas desfasado en la mitad del ancho de una baldosa. Como se puede ver en la Figura 4.23, el espacio que ocupa cada baldosa desfazada corresponde a un vértice especifico.

Con esto queda finalizado el sistema de edición de niveles. Su arquitectura y las relaciones entre sus componentes se pueden ver en la Figura 4.24.

4.16. Personajes

Lo último que falta por incorporar al motor es el soporte para personajes en los niveles. Estos son muy similares a los ítems, pero tienen la diferencia que interactúan activamente con los otros elementos del nivel y, además, tienden a moverse de lugar mucho más seguido.

La principal dificultad para integrar el manejo de personajes en el motor es que cada desarrollador necesitará un comportamiento completamente distinto para sus personajes. Por esto se optó por implementar controladores con los comportamientos más comunes a modo de ejemplo para que los desarrolladores puedan implementar los suyos. Se implementaron tres controladores para personajes: uno que mueve al personaje una baldosa completa cada vez que se presiona una flecha en el teclado, uno que mueve gradualmente al personaje hacia los lugares donde se presiona el cursor y uno que actualiza su sprite según la dirección de su movimiento.

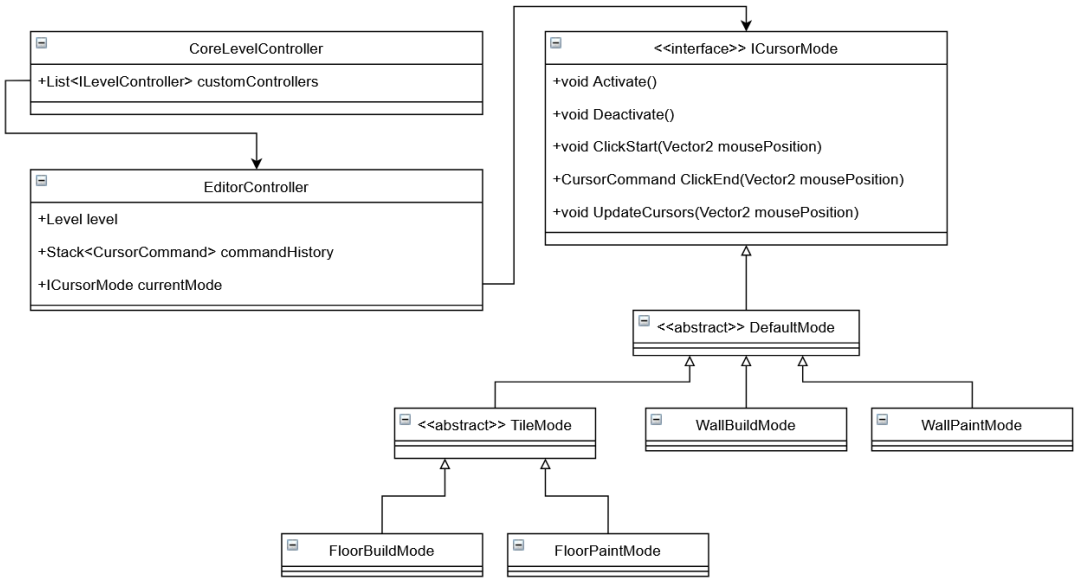

Figura 4.24: UML con la organización final del editor de niveles.

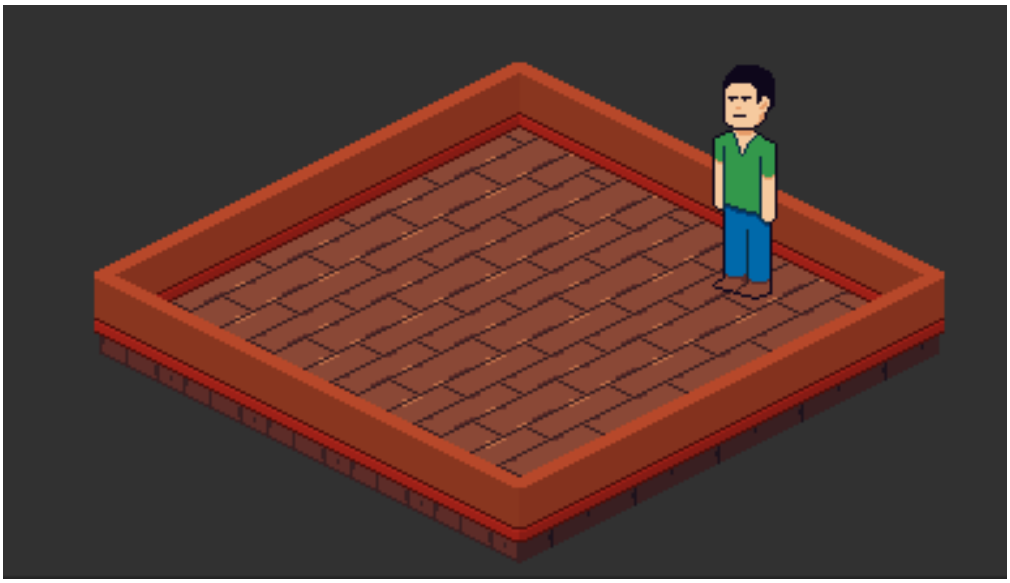

Figura 4.25: Un personaje parado entre dos baldosas.

Los personajes se modelaron de manera similar a los ítems, pero con dos diferencias clave: la primera es que su posición en coordenadas isométricas permite decimales. Esto es para permitir un movimiento más fluido entre las baldosas. La segunda es que, como el movimiento es una parte fundamental en los personajes, se agregaron métodos en el modelo para preguntar a las baldosas si es posible caminar sobre ellas. Para renderizar sus sprites se utilizó el patrón de diseño “Observer” de manera análoga a los ítems y baldosas.

4.17. Callbacks

La forma de conectarse al flujo del nivel es mediante callbacks. Se implementaron callbacks en los eventos que se consideraron más importantes. Así el desarrollador puede suscribirse a ellos implementando clases con las interfaces correspondientes y usarlos para organizar el flujo de los eventos de su juego.

Los callbacks se implementaron a nivel de modelo, pues los eventos más relevantes ocurren ahí (por ejemplo, el cambio de una baldosa, la construcción de una muralla, o el cambio de posición de un personaje). Por esto, fue necesario agregar interfaces a las que puedan suscribirse los elementos del controlador que lo requieran, aunque en varios casos fue posible reutilizar el patrón observador implementado en los observadores de sprite para suscribirse a los callback de cambios en el modelo.

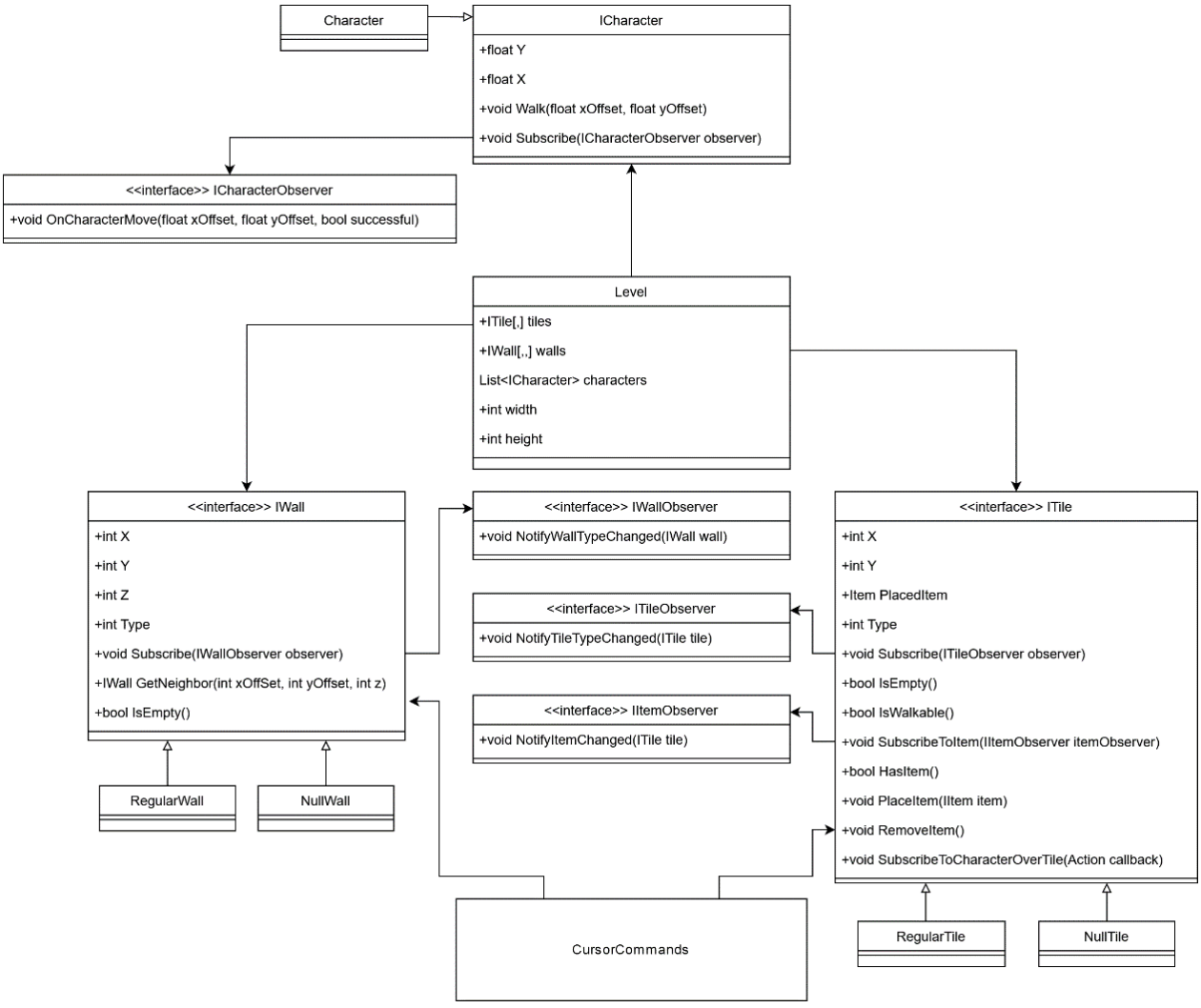

Figura 4.26: UML con los elementos más importantes del modelo al final de la primera iteración.

Capítulo 5

Prueba de concepto: Sokoban

Para evaluar el funcionamiento de las herramientas en un escenario más realista se decidió crear un juego pequeño de principio a fin. Este ejercicio se realizó desde la perspectiva de un desarrollador externo al proyecto, por lo que se estableció como regla el no poder modificar las herramientas ya existentes.

El juego que se desarrollará es Sokoban: un clásico juego de puzle japones en el que el jugador debe empujar cajas para dejarlas en posiciones específicas. La dificultad del juego es que como las cajas solo se pueden empujar y no tirar, es fácil llegar a situaciones en las que no hay forma de dejar las cajas en los lugares necesarios. Se escogió este juego porque puede ser implementado usando proyección isométrica y tiene una sola mecánica, por lo que es fácil de desarrollar.

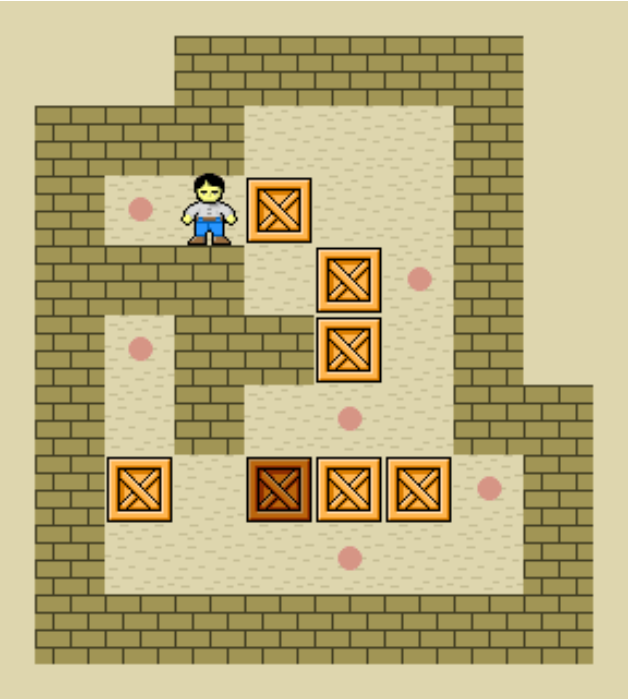

Figura 5.1: Un ejemplo de un jugo de Sokoban.

Ya que todas las herramientas se han probado y desarrollado en Windows, el juego se desarrollará para Android, pues como Unity es multiplataforma se quiere comprobar que las herramientas también funcionen en distintas plataformas.

5.1. Murallas

Las murallas en el juego de Sokoban utilizan una baldosa completa, por lo que, en vez de usar el sistema de murallas creado anteriormente, se utilizaron ítems que ocupan todo el espacio de una baldosa y en base a ellos se construyeron las murallas.

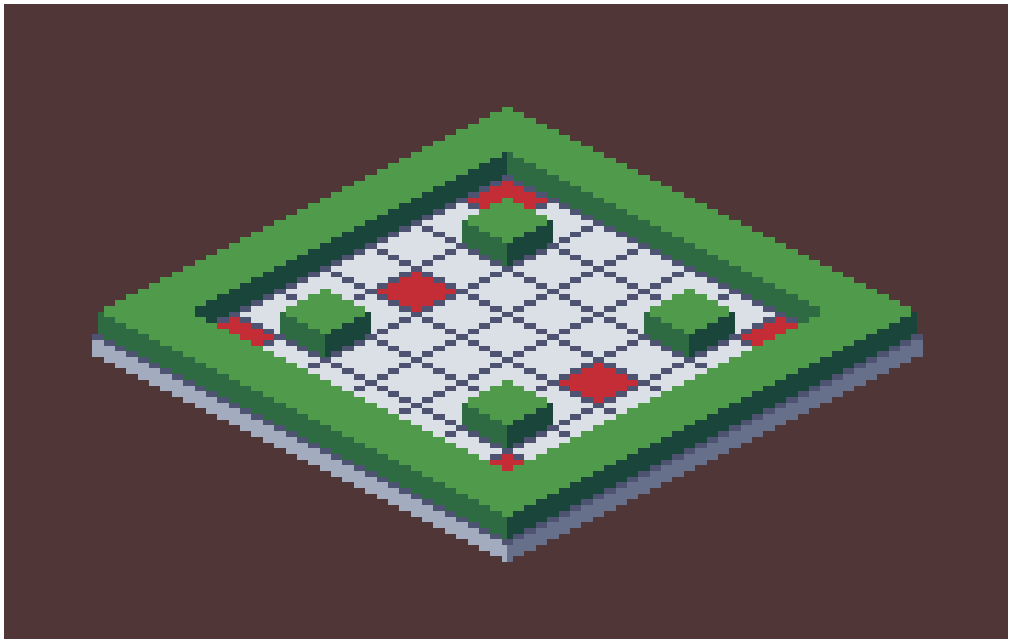

Figura 5.2: Murallas verdes compuestas por varios items.

5.2. Movimiento del Jugador

El controlador para personajes creado anteriormente no permite caminar sobre las baldosas que están marcadas como ocupadas, por esto no fue necesario implementar las colisiones con las murallas.

5.3. Movimiento de cajas

Para que el jugador pueda empujar las cajas se permitió caminar sobre ellas cuando no tengan obstaculos en frente. Luego, en el callback que se activa cuando un personaje se para sobre una baldosa, se revisa si la baldosa tiene una caja y, de ser así, esta se mueve en la dirección que el personaje iba caminando.

En caso de que la caja no pueda empujarse, porque hay una muralla u otra caja en frente, el personaje no puede caminar sobre la caja en primer lugar y nunca se activa el callback.

5.4. Carga de Niveles

Se encontró una implementación web de Sokoban [16] bajo una licencia permisiva [6]. De aquella implementación se pudieron extraer los niveles. Estos niveles se encuentran serializados en archivos de texto plano como se muestra en la Figura 37. En ellos cada carácter representa un componente distinto del nivel (muralla, caja, personaje o posición objetivo).

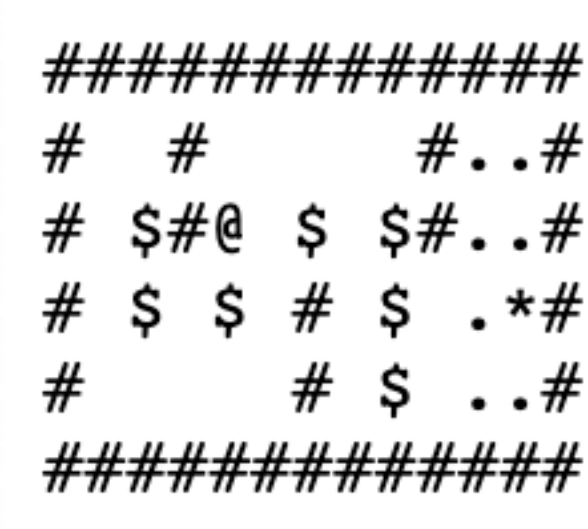

Figura 5.3: Texto plano que serializa información de los niveles de Sokoban.

Utilizando las interfaces de serialización fue posible implementar una clase que cargue estos niveles en texto plano dentro del modelo de nivel ya existente.

5.5. Condiciones de Victoria y Derrota

Cada vez que el jugador realiza un movimiento se revisa si todas las cajas están sobre una posición objetivo y, de ser así, el jugador ha ganado y se le presenta la opción para pasar al siguiente nivel.

El jugador pierde cuando llega a un escenario desde el cual ya es imposible ganar, pero como es parte del desafío del juego el descubrir cuando se ha perdido, no se notifica al jugador cuando ha perdido, sino que se agregó un botón que permite reiniciar el nivel y el jugador debe usarlo cuando estime conveniente.

5.6. Interfaz del Juego

La interfaz de menú y selección de niveles no es parte del alcance de las herramientas, por lo que se implementaron de la manera habitual. Un menú inicial, una pantalla de selección de niveles, y una pantalla de victoria.

Figura 5.4: Interfaz de la prueba de concepto de Sokoban.

5.7. Dificultades Específicas de Android

Alegremente, al usar las herramientas en Android hubo un solo problema: como la carga de recursos se realizaba indicando la ruta completa de los archivos y Android utiliza otro sistema para ordenarlos, no se pudieron cargar. Esto se solucionó llamando a una librería de Unity que detecta el sistema operativo para el cual se está compilando la aplicación y genera rutas compatibles con el sistema adecuado. Fuera de esto, todo funcionó sin la necesidad de modificaciones.

5.8. Conclusiones

La experiencia cumplió su objetivo de permitir ponerse en el lugar de un desarrollador que utiliza las herramientas y evaluar su utilidad y flexibilidad.

Los aprendizajes obtenidos al crear esta prueba de concepto serán explicados y analizados en la siguiente iteración. El juego está disponible en Google Play bajo el nombre de “Just Sokoban”.

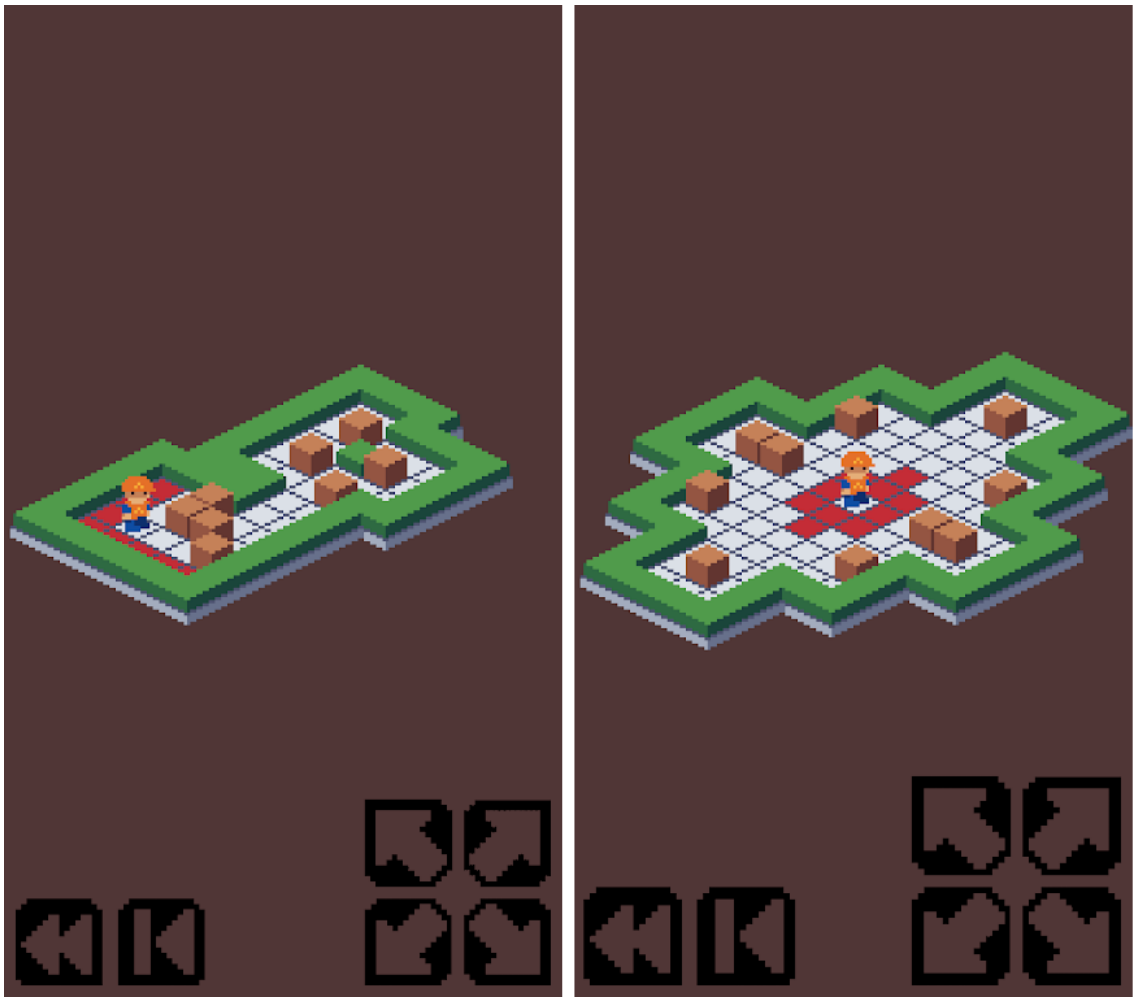

Figura 5.5: Capturas de dos niveles distintos del juego finalizado.

Capítulo 6

Optimizaciones para Unity

La realización de una prueba de concepto, efectivamente, sirvió para identificar problemas e inconsistencias en el diseño que van en contra de los objetivos de flexibilidad, extensibilidad, modularidad y coherencia con el motor para el que se desarrollaran las herramientas.

A continuación, se presentarán cada uno de los cambios realizados a la arquitectura del motor, junto al análisis y la evaluación que llevaron a su ejecución.

6.1. Serializadores como componentes

Al momento de crear la prueba de concepto utilizando las herramientas, se hizo contra intuitivo que ninguna de las herramientas utilice o sea un componente. Por esto, se realizó una evaluación para determinar qué elementos del proyecto deben convertirse en componentes. Entre todos ellos, el primero que se modificó fue el conjunto de serializadores por los motivos que se detallan a continuación:

6.1.1. Disminución de las variaciones de niveles

Como se explicó anteriormente, la clase Level posee una propiedad de tipo ILevelSerializer, la cual se utiliza para cargar y guardar los datos del nivel. En una situación más tradicional, para especificar el tipo específico de serializador que utilizará el nivel se pueden crear subclases de Level que en su constructor creen instancias distintas de serializadores, también se podrían utilizar patrones como Factory o Builder.

El convertir los serializadores en componentes nos permite utilizar una solución mas elegante: hacer que la instancia de Level busque el GameObject que lo contiene y que busque dentro de él algún componente que sea un serializador. Esto permite utilizar la misma clase Level con distintos tipos de serializadores solamente agregando un distinto componente mediante la interfaz gráfica de Unity, como se muestra en la Figura 6.1.

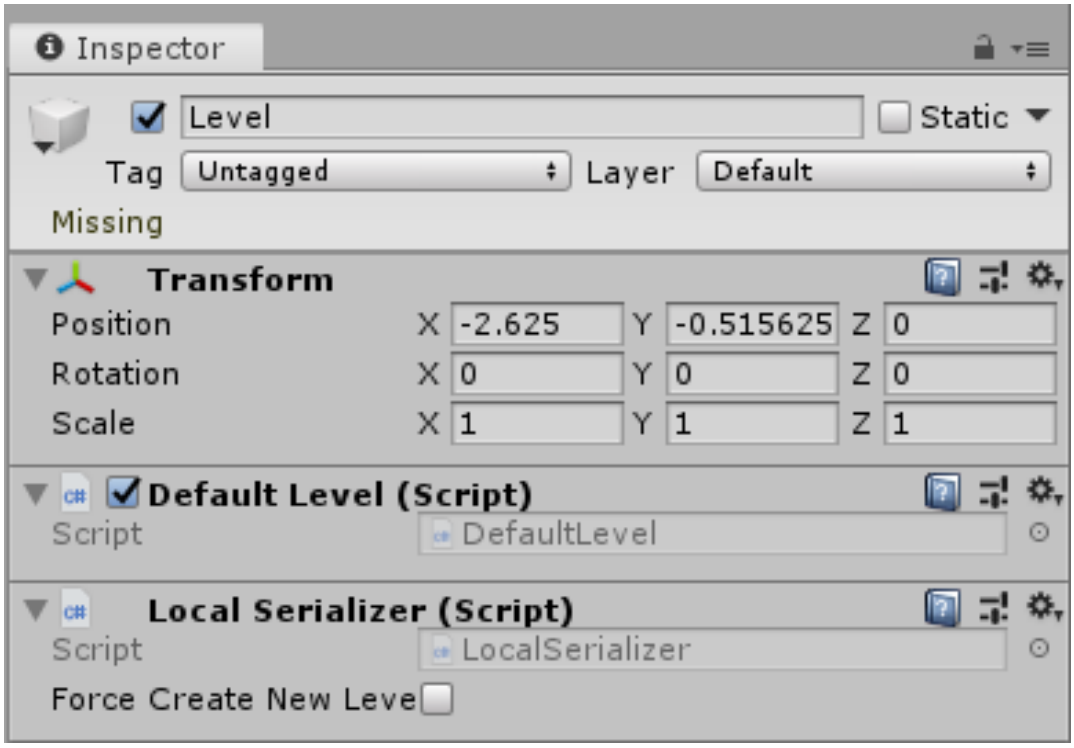

Figura 6.1: Visualización de un serializador como componente en la interfaz gráfica de Unity.

6.1.2. Exposición de propiedades en el editor de Unity

Otra de las ventajas que se mencionaron de los componentes es la capacidad de serializar sus propiedades. Esto significa mostrar su valor en el editor de Unity y permitir modificarlo mediante la interfaz gráfica. En la Figura 6.1 se muestra que el serializador expone su propiedad "Force Create New Level". Para lograr esto, en el código simplemente se creó una variable de clase de tipo booleano con el nombre forceCreateNewLevel y Unity se encargó del resto.

En el caso de los serializadores, esto abre la posibilidad de, por ejemplo, exponer una variable de texto que indique el nombre del nivel a cargar, o el url desde donde cargar el nivel para el caso de un serializador web, tambien se pueden indicar valores por defecto que indiquen como crear un nuevo nivel en el caso de que no exista uno, entre otras muchas posibilidades.

6.1.3. Solución implementada: Serializadores que heredan de MonoBehaviour

La solución para convertir los serializadores a componentes es simple: hacer que hereden de la clase MonoBehaviour. Con eso Unity comprende que estas clases son componentes y los trata como tal. De todas formas, hay que tener en cuenta que la flexibilidad que esto nos brinda trae consigo la responsabilidad de hacerse cargo de un nuevo escenario, pues ahora existe la posibilidad de que el desarrollador no agregue ningún serializador a su nivel.

Si bien siempre existe la posibilidad de arrojar un error al detectar que el nivel no tiene ningún serializador, esto es una solución poco elegante. El tener un componente que requiera

de otro para funcionar adecuadamente indica que estos no son completamente modulares y va en contra de la filosofía que se intenta adoptar.

Por este motivo se recurrió al patrón de diseño Null. Se creó un nuevo serializador que entrega un nivel vacío cuando se le solicita cargar un nivel e ignora las solicitudes de guardado. Este serializador nulo se instanciará cada vez que se detecte que el desarrollador no agregó ningún otro serializador.

De esta forma, el nivel se encargará de buscar el primer componente que implemente la interfaz ILevelSerializer y lo utilizará de la misma forma que lo hacía anteriormente.

6.2. Transformaciones dependientes del nivel

Inicialmente, todas las transformaciones necesarias para posicionar y obtener la rotación de los sprites estaba contenida en una clase estática. Esta decisión se tomó bajo el supuesto de que la transformación entre distintos sistemas de coordenadas no depende de factores externos y siempre se realizará de la misma manera. Aunque en estricto rigor el supuesto es correcto, al crear el pequeño juego que se utilizó como prueba de concepto se manifestaron distintas limitaciones y complicaciones introducidas por el uso de clases estáticas, las cuales se detallaran a continuación:

6.2.1. Existencia de un único nivel

La desventaja más obvia del enfoque que se estaba utilizando es que se vuelve muy complejo el tener más de un nivel al mismo tiempo, pues con un transformador de coordenadas estático se tienen que utilizar soluciones alternativas para trasladar todos los sprites a su posición equivalente en relación al origen de otro nivel. Si bien esto es técnicamente posible, no es una buena práctica el tener que agregar esta lógica en otra parte del código. La clase encargada de las transformaciones debería poder encapsular esta funcionalidad.

Si además se quisiera tener una rotación independiente para cada nivel, habría que utilizar aún más soluciones alternativas, pues como la rotación actual es una única variable estática que es utilizada por varios métodos distintos del transformador, habría que mantener un registro externo con la rotación de cada uno de los niveles y sobre escribir la variable estática de rotación antes de usar cualquier otro método. Esto resultaría en un flujo de ejecución demasiado confuso y propenso a errores.

6.2.2. Uso de coordenadas locales

Como se mencionó anteriormente, en Unity todos los GameObjects tienen un componente llamado Transform. Este componente define la posición, rotación y escala de cada uno de los objetos presentes en el juego.

El mantener el transformador de coordenadas como una clase estática no permite una incorporación más natural a Unity que utilice un componente Transform para definir la posición, rotación y escala del nivel isométrico. Tampoco permite diferenciar entre coordenadas globales y coordenadas locales (relativas al origen de un nivel especifico).

6.2.3. Poca transparencia en el uso de rotaciones

Por último, otra complicación acarreada con el uso de un transformador estático es que como cualquier parte del código puede acceder a la variable de rotación actual, no se muestra de manera explícita que elementos la están usando para efectuar rotaciones. Esto demostró ser confuso al momento de desarrollar ya que no era fácil saber si los resultados que se obtenían al utilizar el transformador consideraban la rotación o no. Incluso, en algunos casos al utilizar dos o más métodos del transformador en el mismo elemento las rotaciones realizaban dos veces.

Al no tener una rotación estática, los elementos que la utilicen tendrán que solicitarla explícitamente, esto tanto en el código de las herramientas que se están desarrollando, como en el código de los usuarios de las herramientas.

6.2.4. Solución implementada: transformador instanciable

Se modificó el transformador de coordenadas de tal manera que ahora sea instanciable. Al momento de crear un nuevo transformador se le debe entregar un nivel como parámetro, de esta forma el trasformador podrá realizar operaciones relativas a su propio nivel. Ya que todos los niveles tienen un GameObject, se pueden aprovechar los métodos del componente Transform de cada nivel para transformar entre la posición global y local de cada uno de los sprites. Esto permite posicionar los sprites de una manera coherente con las distintas propiedades del componente Transform.

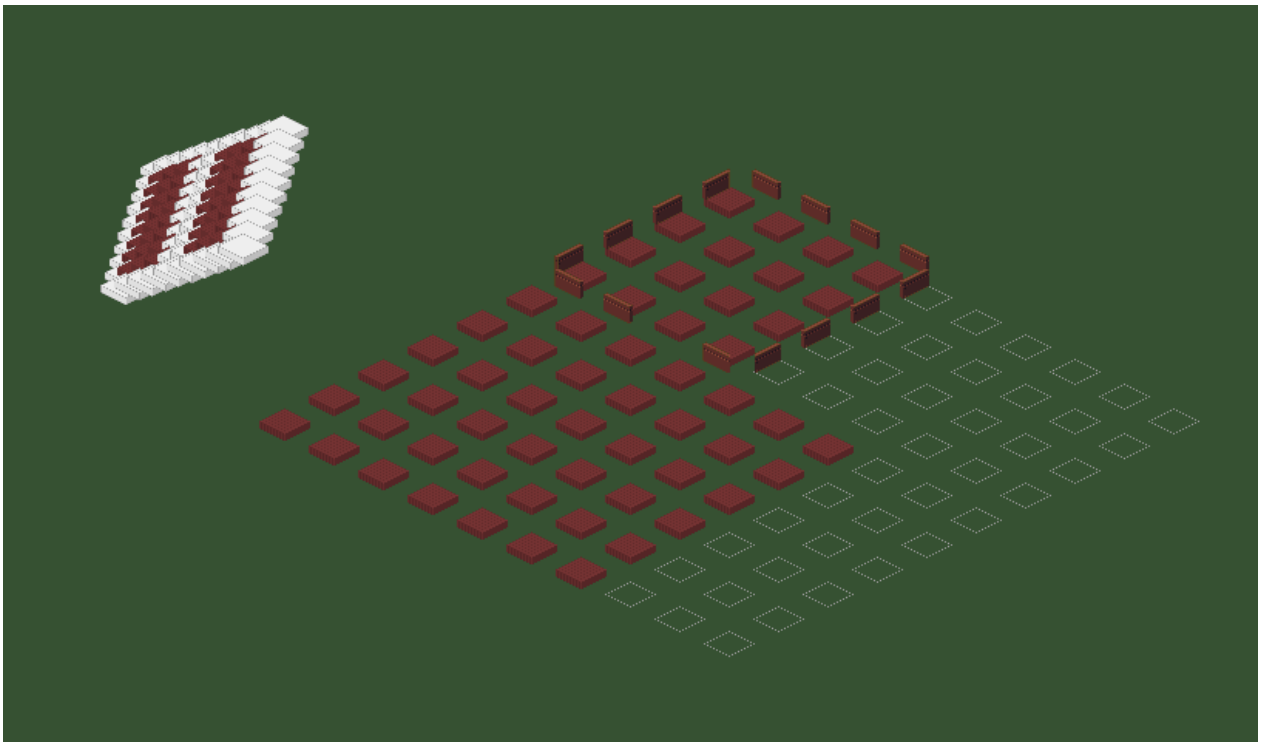

Figura 6.2: Dos niveles rotados y escalados mediante la interfaz de Unity en una misma escena.

Cada nivel creará su propio transformador al momento de ser instanciado. Este estará

disponible para que cualquiera con una referencia al nivel pueda acceder también a los métodos de su transformador. De esta forma se espera aliviar la única desventaja significativa de utilizar transformadores instanciables: la necesidad de una referencia cada vez se desee utilizar sus métodos. Pues como el nivel es el elemento principal con el que se trabaja, es común tener referencias que apunten a él en varias partes del código.

6.3. Componente de posicionamiento isométrico

Continuando con el uso de componentes para mejorar la modularidad del proyecto se decidió delegar la lógica del posicionamiento de sprites a un nuevo componente, pues la filosofía de Unity nos sugiere que las distintas funcionalidades presentes en un GameObject se vean reflejadas en sus componentes. De todas formas, se analizaron las consecuencias de delegar esta funcionalidad y, finalmente, se decidió que era será un cambio positivo por los siguientes motivos:

6.3.1. Funcionamiento independiente de otros componentes

Existirán situaciones en las que el desarrollador solamente estará interesado en posicionar sprites de manera isométrica. Con el modelo anterior era imposible posicionar elementos sin incluir todo el resto de las herramientas, lo cual significa crear un nivel, asignarle datos y organizar los sprites de manera que puedan ser cargados correctamente, como mínimo.

Al crear un componente que se encargue del posicionamiento de sprites, se abre la posibilidad de permitirle funcionar por su cuenta sin dejar de permitir que el resto de las herramientas interactúen con el cuándo estén presentes.

6.3.2. Mantener un registro de la posición isométrica del GameObject

Los observadores de sprites mantienen un registro que les permite encontrar un GameObject a partir de su posición isométrica. Si, por el contrario, se desea encontrar la posición isométrica de un GameObject, no hay registro que permita realizarlo efectivamente.

El componente de posicionamiento es un muy buen candidato para almacenar la posición isométrica de su GameObject correspondiente, pues este la estará utilizando constantemente para calcular su posición cartesiana.

Otra ventaja de este registro es que permite al desarrollador interactuar directamente con el GameObject sin tener que pasar por el nivel. Por ejemplo, se puede hacer que al modificar la posición isométrica de un GameObject este actualice su posición en la pantalla y se encargue de notificar al nivel del cambio.

6.3.3. Serialización de posición y tipo

Otro beneficio es la posibilidad de serializar y mostrar en la interfaz grafica de Unity toda la información relacionada con el posicionamiento del sprite. Tanto su posición isométrica,

como su tipo y el nivel que lo contiene es información útil y conviene poder leerla y modificarla de manera cómoda.

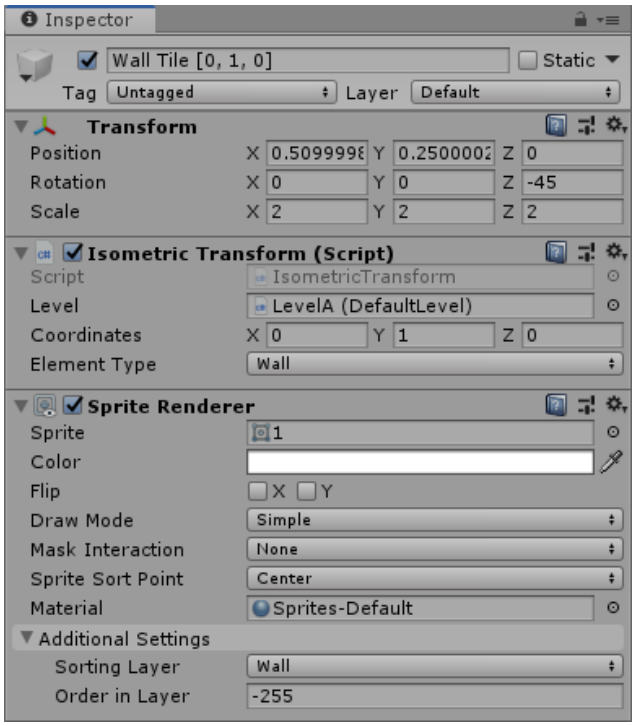

Figura 6.3: Visualización del componente de posicionamiento en la interfaz gráfica de Unity.

6.3.4. Solución implementada: Componente de posicionamiento isométrico

Se construyó un componente que tiene una referencia a Level para poder acceder a su transformador, al cual se le pedirán los cálculos de posiciones. De esta forma se mantienen todas las ventajas del transformador instanciable dentro del componente.

Para asignar la referencia del nivel se optó por implementar dos alternativas: serializar la variable de nivel para que pueda ser asignada manualmente mediante el editor de Unity y, la otra opción, crear un método que reciba un nivel como argumento y lo asigne al componente para poder realizar este proceso de manera procedural con varios componentes a la vez.

Finalmente, se utilizó el patrón Null para crear un nivel nulo con un transformador que funciona como si su nivel siempre estuviera en la posición (0,0,0) con una rotación de 0 grados y una escala de 1. Este nivel nulo será instanciado en el caso de que no se haya asignado la referencia necesaria en el nivel y permitirá la utilización del componente de posicionamiento sin la necesidad de crear un nivel explícitamente.

6.4. Componente de selección de sprites

Siguiendo un razonamiento similar al utilizado con el componente de posicionamiento, se decidió delegar la responsabilidad de asignar los sprites de cada elemento a un componente

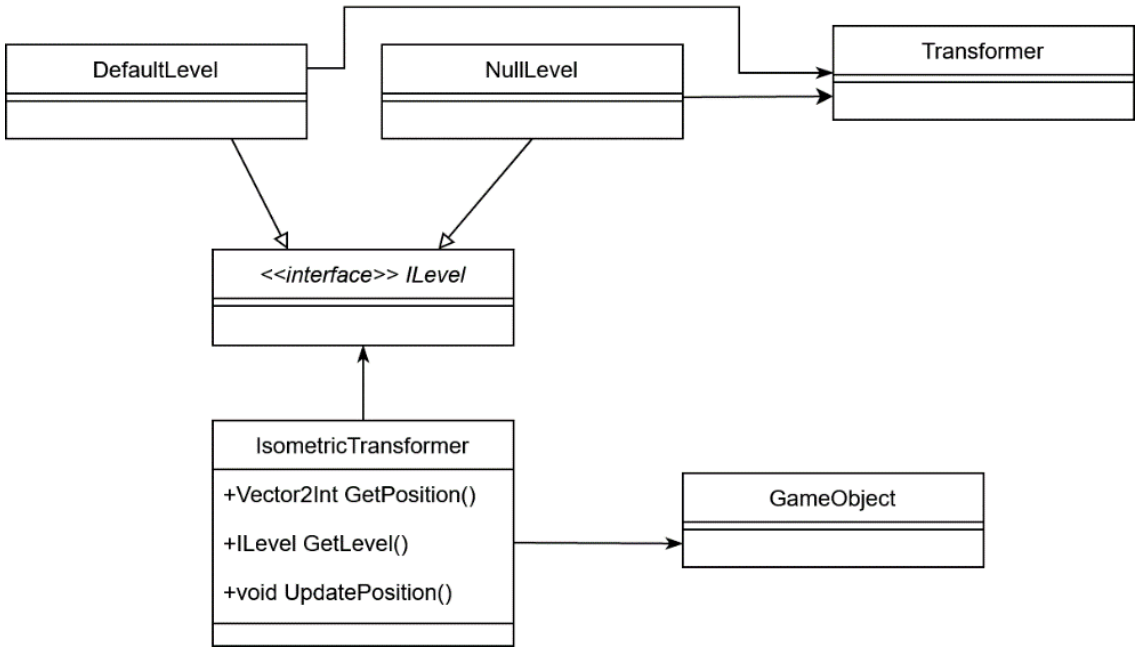

Figura 6.4: UML del componente de posicionamiento isométrico.

nuevo. Si bien todas las ventajas ya mencionadas en la sección anterior se conseguirán de nuevo para el componente de selección de sprites, este cambio entregará aún mas ventajas, por los motivos que se detallan a continuación.

6.4.1. Agrupación de sprites pertenecientes al mismo elemento

Como se vio anteriormente, existen elementos que deben usar diferentes sprites dependiendo de la orientación desde la que se observen, como la baldosa de la Figura 6.5, mientras que también existen elementos que pueden utilizar solo dos sprites para sus cuatro orientaciones o incluso solo uno, como los de la Figura 6.6.

Si bien ya se tiene soporte para estas tres posibilidades, dentro del código no existe ningún tipo de relación entre los distintos sprites pertenecientes a un mismo elemento. Por este motivo, si el desarrollador desea utilizar los sprites para alguna otra cosa, tendrá que obtener los cuatro sprites manualmente y revisar por su propia cuenta si alguno de ellos se repite.

Al utilizar un componente para almacenar los grupos de sprites relacionados, es posible solicitar a la clase encargada de cargar recursos el paquete completo de sprites para un mismo elemento. Luego, el componente puede incluir especificaciones de cuantos sprites diferentes incluye el paquete e incluso puede entregar una lista de los sprites únicos.

6.4.2. Opcionalidad de la funcionalidad

Si bien ya se mencionó que una de las principales ventajas de separar la funcionalidad es que da paso a la posibilidad de reutilizarla en otras situaciones, en este caso la separación de funcionalidad también permite desactivarla cuando sea necesario y con un alto nivel de granularidad. Esto es porque los componentes de Unity pueden ser desactivados tan solo

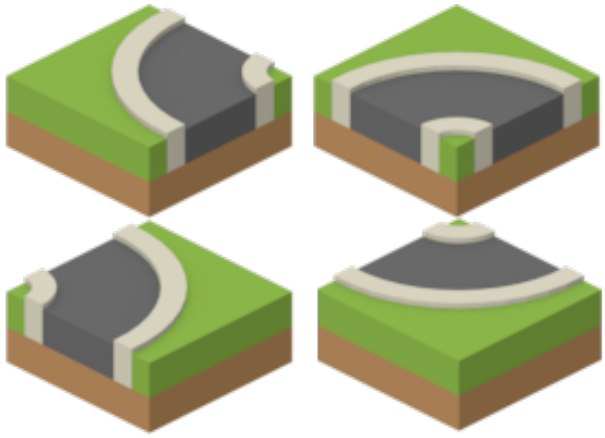

Figura 6.5: Ejemplo de una baldosa que utiliza cuatro sprites para sus distintas orientaciones.

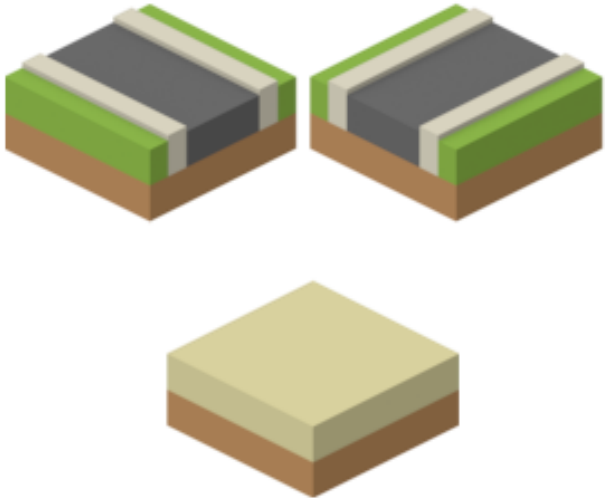

Figura 6.6: Ejemplo de baldosas que utilizan menos de cuatro sprites para sus orientaciones. En la mitad superior (a) una baldosa que utiliza dos sprites en total y en la mitad inferior (b) una baldosa que utiliza solo un sprite para sus cuatro orientaciones.

modificando una propiedad booleana del mismo componente. Esto permite desactivar la rotación tanto en tipos completos de elementos como en elementos individuales.

6.4.3. Solución implementada: Componente de selección de sprites

La solución que se implementó consiste en una clase que estará encargada de actualizar los sprites para cada GameObject que forme parte de un nivel isometrico. Esta extiende de MonoBehaviour y mantiene una lista estática de todas sus instancias, lo que permite un fácil acceso para cuando deban actualizarse todos los sprites (por ejemplo, en un cambio de orientación del nivel). Además, el componente tiene una referencia al nivel al cual pertenece. Esto permite que al rotar un nivel completo no se modifiquen los sprites pertenecientes a otros niveles.

Estos componentes son instanciados automáticamente por los observadores de sprites al momento de crear o modificar algún elemento del nivel, pero también podrán ser agregados a otros GameObjects mediante la interfaz gráfica de Unity.

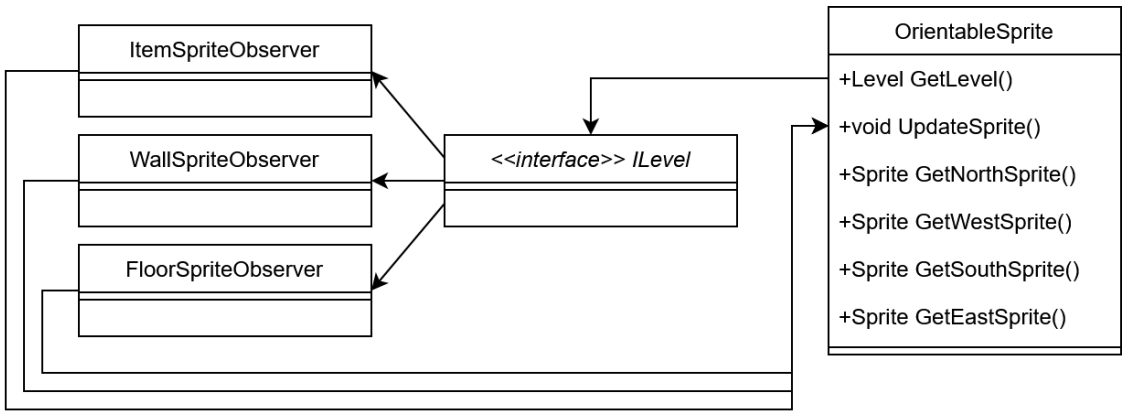

Figura 6.7: UML de la interacción entre el componente de selección de sprites y los observadores.

6.5. MonoBehaviour con callbacks

Al crear componentes durante esta segunda iteración de desarrollo, se pudo ver que la mayoría de ellos necesitan trabajar con elementos de su nivel que tardan en inicializarse. Hasta el momento, esto se ha manejado preguntando periódicamente al nivel si los componentes necesarios ya están listos para ser utilizados hasta que el componente pueda comenzar a trabajar.

Para hacer este proceso mas eficiente, se decidió crear una clase abstracta que pueda recibir callbacks de tal forma que pueda ser usada por los componentes que necesiten ser notificados de los cambios de estado en el nivel.

6.5.1. Solución implementada: IsoMonoBehaviour

Se creó una clase abstracta que hereda de MonoBehaviour, esto permitirá usar esta nueva clase para crear componentes manteniendo toda funcionalidad de MonoBehaviour, incluyendo la compatibilidad con la interfaz gráfica de Unity.

Para que el desarrollador pueda incluir su propia funcionalidad dentro de los callbacks, se crearon funciones vacías en la clase abstracta que serán llamadas por el nivel cuando ocurran eventos importantes.

La decisión de utilizar una clase abstracta en vez de una interfaz fue tomada porque en esta ultima es posible crear funciones con una implementación vacía. De haberse utilizado una interfaz el desarrollador habría tenido que escribir explícitamente cada una de las llamadas, aunque no vaya a utilizarlas.

Las llamadas que estarán disponibles para ser utilizadas por el desarrollador serán las siguientes:

- **OnInit:** Llamada al inicio del ciclo de vida de un nivel.
- **OnLevelLoad:** Llamada cada vez que se cargan los datos de un nuevo nivel.
- **OnLevelUnload:** Llamada antes de cargar los datos de un nuevo nivel, cuando los datos del nivel anterior aún siguen presentes.
- **OnOrientationChange:** Llamada cuando cambia la orientación desde la cual se debe observar el nivel.

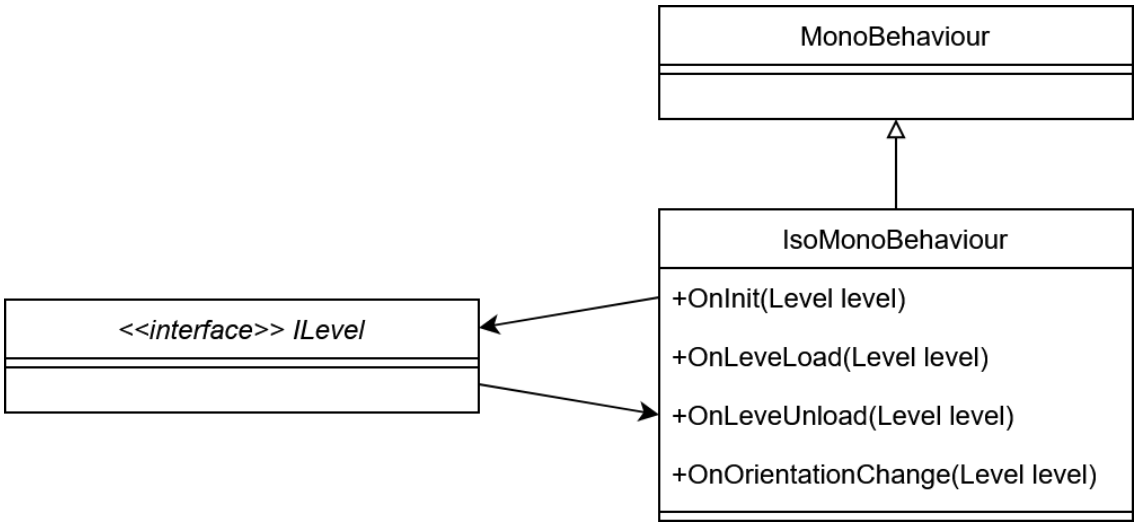

Figura 6.8: UML de la interacción entre IsoMonoBehaviour, MonoBehaviour y ILevel

6.6. Observadores de sprites como componentes

El paso final para terminar de modularizar completamente las funcionalidades de posicionamiento y selección de sprites es convertir a los observadores en componentes. Se detallarán a continuación los motivos por los cuales se decidió realizar este cambio:

6.6.1. Escenarios independientes del nivel

Si bien, con los componentes anteriores ya era posible agregar elementos que no dependieran de un nivel, seguía sin ser posible combinar el uso de estos nuevos componentes con un nivel. Si se intentaba hacer esto los observadores crearían y posicionarían sus propios sprites, interfiriendo con los que el desarrollador hubiera agregado manualmente.

El convertir a los observadores en componentes permite apagarlos y evitar que interfieran con los demás elementos que quiera agregar el desarrollador. Esto otorga la capacidad de aprovechar las otras funcionalidades que implementa el nivel, como el almacenamiento de sus datos y la serialización de ellos.

Además, el separar los tres tipos de observadores en componentes distintos permite ahorrar recursos a un juego que no utilice alguno de ellos. Cabe mencionar que al buscar ejemplos de juegos isométricos se encontró una cantidad considerable que no utilizaban murallas, los cuales aprovecharían esta funcionalidad

6.6.2. Observadores personalizados

Otra ventaja de realizar este cambio es que los componentes, por su naturaleza modular, son muy fáciles de reemplazar. Si el desarrollador quisiera utilizar un observador distinto que observe solo algunas partes del nivel o que espere a que se cumplan ciertas condiciones para actualizarlo, simplemente tendría que crear un componente propio que comparta la interfaz del que observador que desea reemplazar.

Por ejemplo, varios juegos isométricos de estrategia, como Age of Empires [13] y FreeCiv [12], implementan una mecánica conocida como *Fog of War*, lo que se traduce al español como *Niebla de la Guerra*. Esta mecánica consiste en solo revelar las baldosas en las que el jugador se encuentra presente de manera activa, como se muestra en la Figura 6.9. Mediante el uso de un observador personalizado se podría solo actualizar los sprites de las baldosas que se encuentren cercanas al jugador.

Figura 6.9: Captura del juego FreeCiv donde se pueden ver baldosas negras cuyo contenido no ha sido revelado, pues aún no han sido exploradas.

6.6.3. Solución implementada: Observadores como componentes independientes

Para mostrar con mayor claridad los métodos que deben implementar los observadores, se crearon tres interfaces distintas: una para cada tipo de observador. Se modificaron los observadores actuales para que extiendan de IsoMonoBehaviour e implementen la interfaz que les corresponda.

Los métodos OnLevelLoaded y OnLevelUnloaded de IsoMonoBehaviour se utilizaron para saber cuándo crear y eliminar todos los sprites del nivel. De esta manera el mismo nivel puede reutilizarse para cargar distintos datos.

El componente IsometricTransform, encargado de posicionar los sprites, fue modificado para que cuando se modifique la posición del sprite, se registre este cambio en el modelo del nivel. De esta forma se mantendrá la consistencia de los datos de manera independiente al modo en el que el desarrollador utilice las herramientas.

A diferencia de los componentes anteriores, los observadores no tendrán soporte para

funcionar sin nivel. En este caso no tiene sentido que puedan funcionar por su propia cuenta, pues si no hay nivel no hay nada que observar.

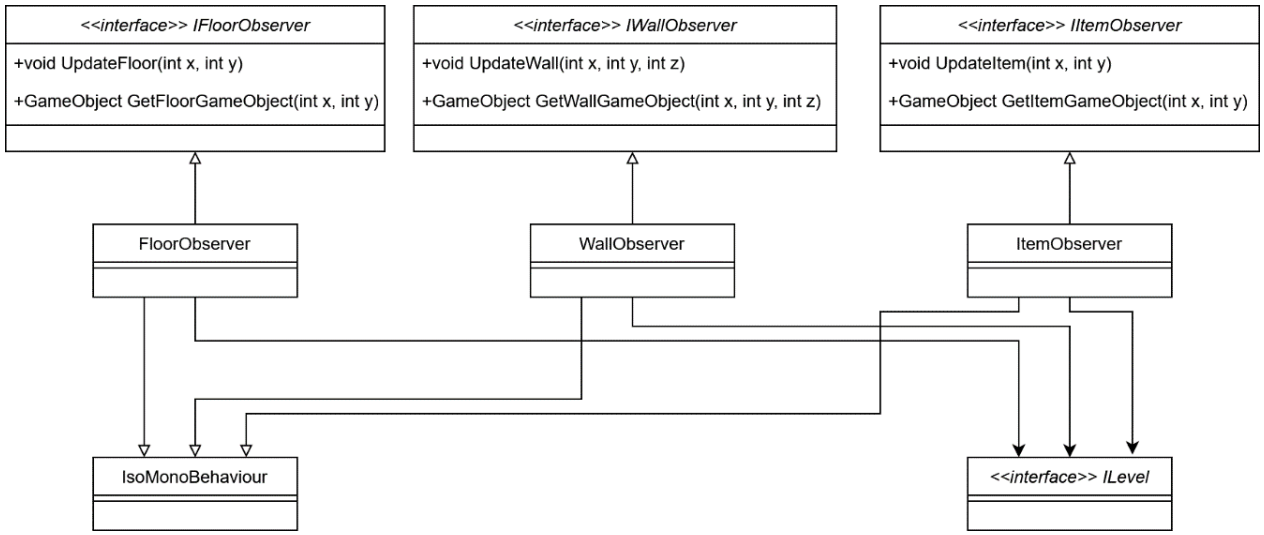

Figura 6.10: UML de los observadores de sprites como componentes.

6.7. Selección de componentes basales

Como se explicó anteriormente, la forma correcta de trabajar con Unity es agregando funcionalidad a los GameObjects mediante componentes. Por este motivo es necesario pensar que las herramientas isométricas serán utilizadas en conjunto a una gran cantidad de componentes distintos.

Como los desarrolladores requerirán poder agregar sus propios componentes a los GameObjects creados por las herramientas, se crearon dos mecanismos que permitirán realizar esto de la manera más sencilla. Para poder entender el funcionamiento de estos mecanismos, es necesario comprender el sistema que Unity ya incluye con este objetivo: los Prefabs.

6.7.1. Utilización de Prefabs como base para los elementos creados con las herramientas

Para permitir a los desarrolladores personalizar los elementos que serán instanciados mediante las herramientas isométricas, se crearon tres Prefabs a los que se les dará el nombre de Prefabs Basales.

Cada uno de estos Prefabs servirá de base a uno de los tres elementos que componen un nivel isométrico: baldosas, murallas e ítems. La idea es que los observadores creen instancias de estos Prefabs para mostrar los sprites.

6.7.2. Versatilidad de la utilización de Prefabs Basales

Este cambio es un avance considerable en la integración entre las herramientas y Unity, pues permite a los desarrolladores continuar con su ciclo de trabajo normal sin ser afectados

por el uso de lo que vendría a ser una librería externa.

Si antes de implementar este cambio, un desarrollador hubiera necesitado agregar algo tan común como un componente detecte si el ratón ha sido presionado sobre su GameObject, este habría tenido que crear un script que itere sobre todos los GameObjects agregando su componente y que repita el proceso cuando sean actualizados. Con este cambio solo se necesita buscar el Prefab Basal correspondiente y presionar "Agregar Componente".

La integración en el flujo del motor permite agregar a las herramientas todas las funcionalidades que incluye Unity. Por lo tanto, ahora los elementos isométricos pueden incluir animaciones, detectar eventos, detectar colisiones, incorporar físicas, e incluso ser parte de juegos de realidad aumentada o realidad virtual.

6.7.3. Solución Implementada: Carga de Prefabs en tiempo de ejecución

Anteriormente, los observadores de sprites creaban un GameObject y le agregaban el componente encargado de renderizar los sprites manualmente. Cuando se crearon los componentes de posicionamiento y de selección de sprites se tuvieron que modificar los observadores para que también agreguen estos nuevos componentes.

Para incorporar futuras modificaciones automaticamente, se cambió aquella parte del código para instanciar un GameObject en base a un Prefab sin conocer sus componentes. Este código también revisa que el Prefab tenga por lo menos los componentes esenciales para funcionar junto a las herramientas y los agrega si no los encuentra.

Para saber qué Prefab instanciar se utiliza el sistema de carga de recursos de Unity. Este permite acceder en tiempo de ejecución a cualquier carpeta de nombre Resources que se encuentre en el entorno del proyecto.

Figura 6.11: Visualización de Prefabs Basales dentro del explorador de recursos de Unity.

Como se puede ver en la Figura 6.11, se puede acceder a los Prefabs Basales desde dentro de Unity, lo que permite editarlos mediante la interfaz que se muestra en la Figura 2.3.

6.8. Modificación de Prefabs individuales

Actualmente, para crear un elemento isométrico se instancia su Prefab Basal, se busca en la carpeta correspondiente el sprite cuyo nombre es igual al índice del elemento y se asigna al GameObject. Con este método, la única personalización que puede haber entre dos GameObjects de un mismo tipo es su sprite.

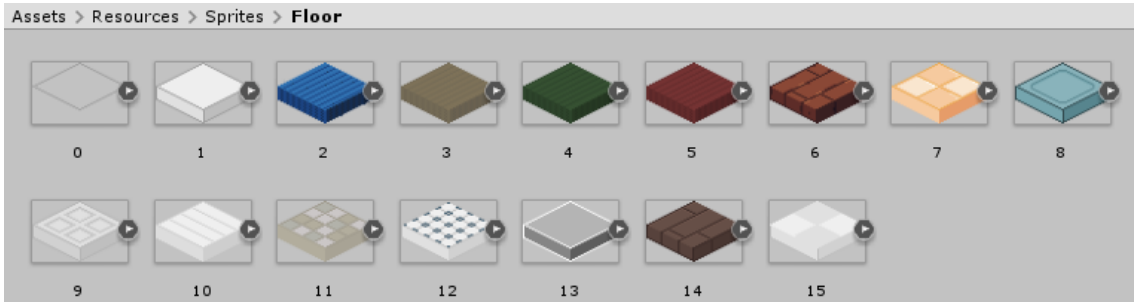

Figura 6.12: Visualización de sprites de las baldosas dentro del explorador de recursos de Unity.

Se implementó un sistema que permite sobrescribir todos los componentes del Prefab Basal de manera distinta para cada elemento que se quiera incluir. Con este cambio es posible tener Prefabs distintos para elementos del mismo tipo, lo cual significa que las todas las ventajas obtenidas con la incorporación de Prefabs Basales serán extendidas de manera granular a cada elemento individual.

Para conseguir esto se permite al usuario poner un Prefab en vez de un sprite en la carpeta de recursos, el cual es utilizado por la clase encargada de cargar recursos que ya existe actualmente.

Para aislar este cambio del resto de las herramientas se decidió hacer que la clase encargada de cargar recursos siempre entregue Prefabs. En el caso de que no exista un Prefab personalizado para el elemento particular que se quiere cargar, se entregara el Prefab Basal correspondiente y se le asignará su sprite como se hacía anteriormente.

Esto significa que los observadores dejan de tener la responsabilidad de asignar los sprites a los Prefabs, ya que el cargador de recursos los entrega listos para ser utilizados.

A continuación, se referirá a las ventajas de agregar esta personalización de este modo.

6.8.1. Encapsulación de diferencias entre sprites y Prefabs personalizados

Cada vez que un usuario de las herramientas quiera agregar un nuevo elemento isométrico, este deberá preguntarse si el elemento que quiere incorporar tendrá funcionalidades distintas al resto para saber si utilizar un Prefab personalizado o simplemente un sprite.

El cargador de recursos se encarga de entregar en ambos casos un Prefab listo para ser usado, lo que consigue que este cambio no produzca diferencia alguna en ninguna otra par-

te del código en el futuro y que el desarrollador que quiera usar el recurso no tenga que preocuparse de diferenciar si este es un Prefab o sprite.

6.8.2. Incorporación de GameObjects de manera transversal a las herramientas

Este cambio también abre el paso a la utilización de GameObjects en varias herramientas distintas. Al tener la seguridad de que siempre tendremos un GameObject para cada elemento, se pueden agregar métodos que permitan obtenerlo utilizando solo la posición isométrica del elemento.

Al permitir a los desarrolladores obtener con facilidad los GameObject que necesitan, se les da acceso a utilizar sus componentes e interactuar con Unity, lo cual permite aprovechar todas las distintas funcionalidades que ofrece el motor.

6.8.3. Solución implementada: Cargador de recursos con soporte para Prefabs

No se eliminaron las funciones del cargador de recursos que permiten obtener los sprites, pues pueden ser necesarias en el futuro. En el caso de que al cargador de recursos le soliciten un sprite, pero el recurso sea un Prefab, se entregara el sprite asignado al componente renderizador de sprites del Prefab. Esto también apoya la encapsulación del nuevo comportamiento, pues se podrá obtener un sprite de manera independiente al tipo de recurso que se esté usando.

El método para obtener Prefabs los carga de la misma manera que se hace con los sprites, y en caso de no existir un Prefab utilizará el Prefab Basal junto a su sprite correspondiente.

También se agregaron métodos en los observadores para obtener GameObjects de elementos en base a su posición. Como se puede acceder a los observadores desde el nivel, será posible obtener los GameObjects en la mayoría de los casos. En el caso de que no se hayan agregado observadores al nivel, no será posible obtener aquellos GameObjects, lo cual tiene sentido pues el desarrollador estaría creando los suyos con algún otro método.

6.9. Almacenamiento de variaciones de murallas en un caché

Anteriormente, se había creado un sistema de caché para evitar cargar en la memoria el mismo sprite más de una vez, el cual fue implementado en la clase encargada de la carga de sprites. Luego de esto se implementó el sistema de bordes dinámicos para las murallas, que se puede ver en la Figura 4.14, pero como se puede ver en la Figura 6.13, estos bordes son generados después de cargar los recursos, por lo que no son guardados en aquel caché.

Al realizar pruebas de rendimiento se pudo ver que generar estos bordes es costoso. Esto es particularmente notorio en las operaciones que requieren actualizar todos los sprites de un nivel, como la inicialización o los cambios de orientación. Por ejemplo, en la Figura 6.14

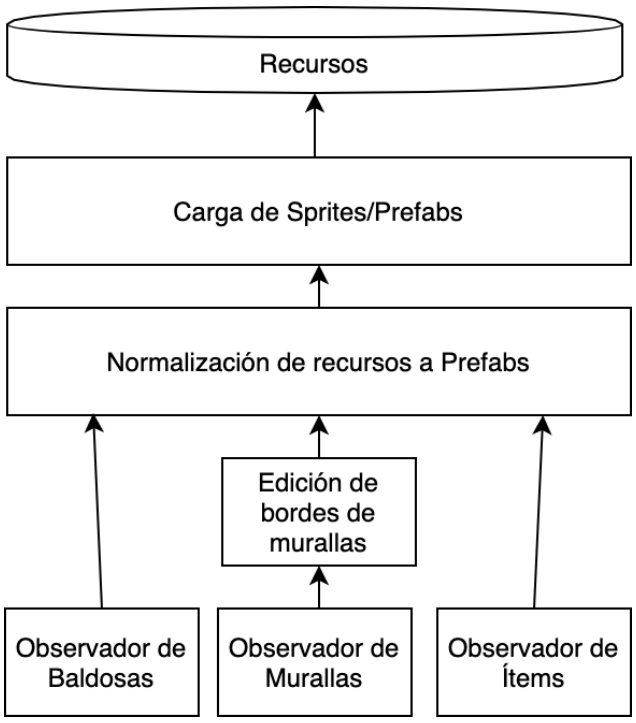

Figura 6.13: Flujo de la carga de recursos y su transformación a Prefabs.

se puede ver una espina que corresponde al cambio de orientación de un nivel, donde la generación de sprites toma por si sola más de un segundo.

Figura 6.14: Rendimiento de las herramientas al actualizar todos los sprites de un nivel de 10x10 al mismo tiempo, antes de implementar el caché de variaciones de murallas.

6.9.1. Impacto en el uso de memoria

Antes de realizar el cambio, se evaluó el impacto en el uso de memoria RAM que este puede producir.

Los bordes de la muralla dependen de los vecinos que tenga, entonces como cada muralla se conecta a 6 vecinos, para cada sprite se pueden generar 64 variaciones diferentes. Como el sprite de una muralla pesa 20KB antes de ser comprimido, en el peor de los casos se utilizarán 2.5MB de memoria por cada tipo de muralla presente en un nivel.

De todas formas, al hacer una medición del uso real de memoria antes y después del cambio se pudo ver que al guardar los sprites en un caché se utiliza menos memoria que al no hacerlo. Luego de investigar el fenómeno se comprendió que esto ocurre porque antes de

utilizar el caché, cada vez que se generaba un sprite dinámicamente este se guardaba en una nueva posición de memoria sin importar si este era duplicado o no.

6.9.2. Solución implementada: Diccionario de variaciones de murallas

Utilizando un simple diccionario de sprites, se haran consultas antes de generar uno nuevo. Si el sprite ya existe se evitará la generación y se devolverá el sprite del caché.

Para las llaves aquél diccionario se generó un identificador único para cada sprite que considera la existencia de cada uno de sus vecinos por separado, la orientación de la muralla y su estilo. De esta forma se evitarán las colisiones.

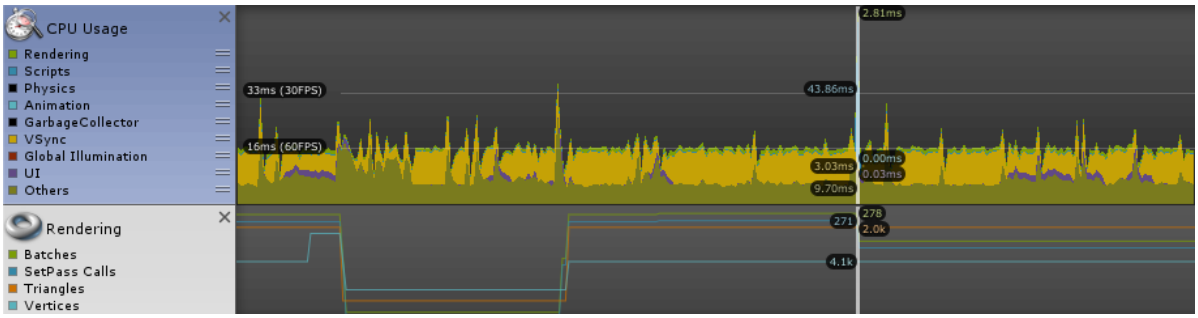

Figura 6.15: Rendimiento de las herramientas al actualizar todos los sprites de un nivel de 10x10 al mismo tiempo, después de implementar el caché de variaciones de murallas.

Como se puede ver en la Figura 6.15, el rendimiento mejoró de manera considerable, haciendo que el costo de una actualización pase de más de un segundo a 45 milisegundos.

6.10. Búsqueda de caminos

Una de las funcionalidades más utilizadas en los motores de videojuegos es la capacidad de buscar caminos y navegar dentro de los niveles evitando los obstáculos.

Para conseguir esta funcionalidad, se implementa un algoritmo de búsqueda de caminos de la misma manera que un desarrollador externo al proyecto incluiría cualquier otra funcionalidad que necesite. Esto permitirá comprobar que aquel proceso no requiera modificar el código de las herramientas, sino que solo usar los mecanismos de interacción ya establecidos.

6.10.1. Algoritmo de búsqueda A*

Para buscar caminos dentro del nivel es útil visualizarlo como un grafo donde cada baldosa corresponde a un nodo que está conectado a sus cuatro vecinos inmediatos. Las aristas pueden incluir un valor que represente la dificultad de moverse entre sus dos baldosas.

Modelando el problema de esta manera es sencillo entender que encontrar el mejor camino para navegar el nivel desde una baldosa a otra equivale a encontrar el camino más barato entre los dos nodos que le corresponden en el grafo.

Para realizar esto se utilizó el algoritmo A*. Este es una variación del algoritmo de Dijkstra que incorpora heurísticas para dar preferencia a los nodos más cercanos al nodo objetivo utilizando la distancia euclidiana. Este algoritmo requiere que a cada nodo le corresponda una posición, lo cual se cumple en este caso, pues cada nodo representa una baldosa la cual tiene una posición determinada.

Los nodos se almacenan en alguna cola de prioridad, donde su prioridad es la distancia euclidiana hasta el nodo objetivo. De esta forma el algoritmo busca primero los caminos más directos y solo se comienza a desviar al encontrar obstáculos que no le permitan moverse. Este es un algoritmo muy popular en el desarrollo de videojuegos, pues en el mejor caso su orden es de n (donde n corresponde al número de nodos que tiene el camino optimo). Este caso corresponde a cualquier nivel donde no existan obstáculos en el camino que se quiere encontrar, por lo que ocurre frecuentemente.

6.10.2. Componente con información del terreno

Para que un algoritmo de búsqueda de caminos funcione correctamente debe poder preguntarle a cada baldosa si se puede pasar por sobre ella y cuál es el costo de hacerlo. Para que cada tipo de baldosa pueda tener valores distintos en estas variables se decidió crear un componente nuevo, pues este puede ser agregado a las baldosas mediante el sistema de Prefabs basales.

Otra ventaja de utilizar un componente es que como este tiene acceso a todos los otros componentes de su propio GameObject puede acceder a otros elementos relevantes del nivel. Por ejemplo, puede utilizar el componente de posicionamiento para obtener la posición isométrica de la baldosa, la que le permitirá preguntar al modelo del nivel si la baldosa tiene un ítem sobre ella que dificulte o impida el tránsito.

6.10.3. Solución implementada: Extensión de búsqueda de caminos

La solución implementada funciona como una extensión a las herramientas ya existentes, pues no las modifica, pero depende de ellas.

El algoritmo de búsqueda de caminos se implementó en una clase estática que recibirá el nivel en el que se quiere trabajar, la baldosa inicial y la final. Esta clase buscará el componente que almacena el costo de tránsito en cada baldosa que lo necesite.

Fue incluida la opción de considerar las cuatro baldosas más cercanas como vecinos para poder caminar directamente a ellas y la opción de considerar las ocho más cercanas para permitir caminar en diagonal. No se incluyen dentro de los vecinos las baldosas que tienen una muralla entre medio.

Luego de esto, bastó con agregar el nuevo componente al Prefab basal de las baldosas y en algunos otros Prefabs individuales para sobrescribir sus valores en algunos tipos específicos de baldosas.

El poder agregar funcionalidad a las herramientas desde la perspectiva de un desarrollador externo y sin modificarlas indica que ya es posible pasar a la última iteración del proyecto y

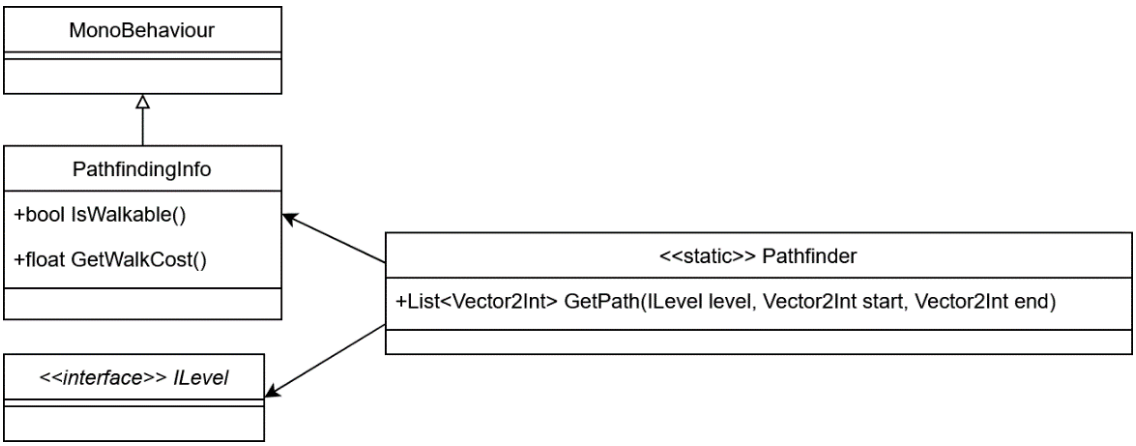

Figura 6.16: UML del sistema de búsqueda de caminos.

comenzar a construir pruebas de concepto.

6.10.4. Eliminación de callbacks desde el modelo

Una consecuencia de las optimizaciones implementadas en las secciones anteriores fue el cambio gradual desde una arquitectura donde toda la interacción del desarrollador con las herramientas pasaba por la clase *Level*, que era el único punte de acceso a la funcionalidad (Figura 3.4), a una arquitectura donde la modularidad de los componentes permite exponer interfaces propias de cada uno para acceder a sus funcionalidades de manera mas directa.

Esta nueva manera de interactuar con las herramientas dejó obsoletos a los callbacks que se efectuaban desde el modelo, pues ahora el desarrollador puede consultar el estado de los diferentes elementos de un nivel isométrico directamente y realizar cambios en ellos sin la necesidad de una clase intermedia. Si bien este nuevo enfoque es mas sencillo, además tiene la ventaja de ser mas flexible, pues el desarrollador no está limitado a utilizar elementos con callbacks explícitamente definidas.

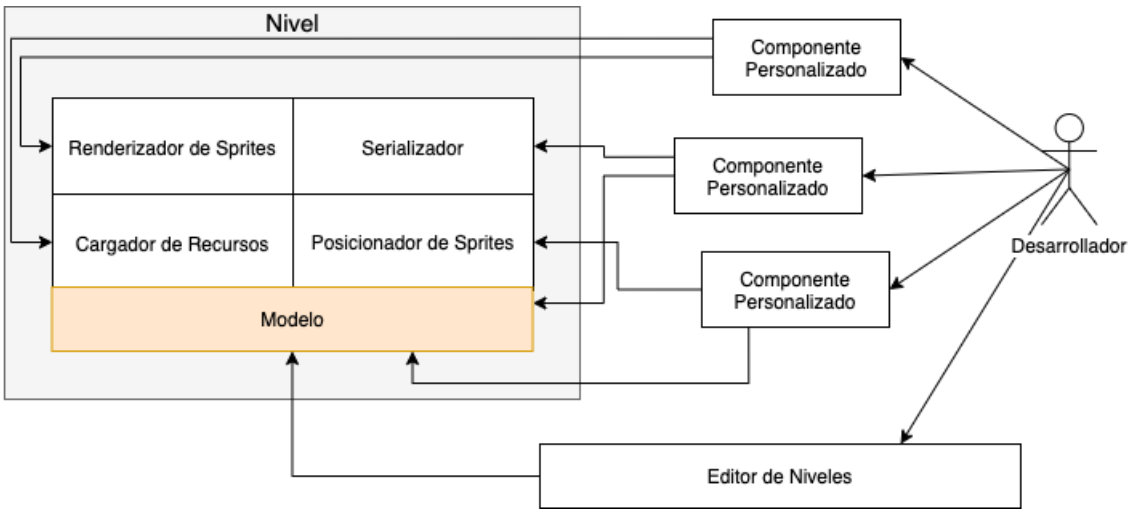

Figura 6.17: Arquitectura de las herramientas al no usar callbacks.

Como se ve en la Figura 6.17, los componentes creados por el desarrollador pueden interactuar libremente con cualquier elemento de las herramientas, incluso el modelo.

Capítulo 7

Validación

Para validar el cumplimiento de los objetivos propuestos se utilizaron las herramientas desarrolladas para construir prototipos de videojuegos isométricos. Estos prototipos implementaran mecánicas comunes utilizadas en varios videojuegos, lo cual permitirá demostrar la flexibilidad del resultado obtenido.

Con el objetivo de poder abarcar la mayor cantidad de funcionalidades distintas, solo se implementó la lógica de juego en los prototipos. Esto quiere decir que elementos como menús de selección de niveles, pantallas con tablas de puntajes máximos y otros componentes que no interactúan con las herramientas serán ignorados para usar ese tiempo en probar mas mecánicas distintas.

7.1. Juego de estrategia por turnos

La simplicidad y regularidad de la proyección isométrica la convierte en una de las favoritas a la hora de hacer juegos que requieran mostrar mucha información en la pantalla. Tal es el caso de los juegos de estrategia por turnos pues en este genero, como lo dice su nombre, el jugador debe planear complejas estrategias, para lo que necesita que la información le sea presentada con la mayor claridad posible.

En los juegos de estrategia por turnos, el nivel representa un tablero en el que se posicionan distintos elementos. Cada jugador usa su turno para mover estos elementos y hacerlos interactuar con otros. Dentro de esta categoría entran las implementaciones de juegos de mesa como el ajedrez y las damas, y los juegos de combate por turnos, como los que se muestran en la Figura 7.1.

Si bien estos juegos son muy distintos entre si, todos comparten la misma mecánica base ya mencionada: seleccionar personajes, moverlos y hacerlos interactuar con otros. Toda la complejidad extra que hace distinto a cada uno de estos juegos se construye sobre esta base y depende poco de las herramientas isométricas.

Para validar la capacidad de usar las herramientas para hacer un juego de estrategia por turnos, se construyó un juego de ajedrez donde los jugadores puedan tomar piezas, moverlas

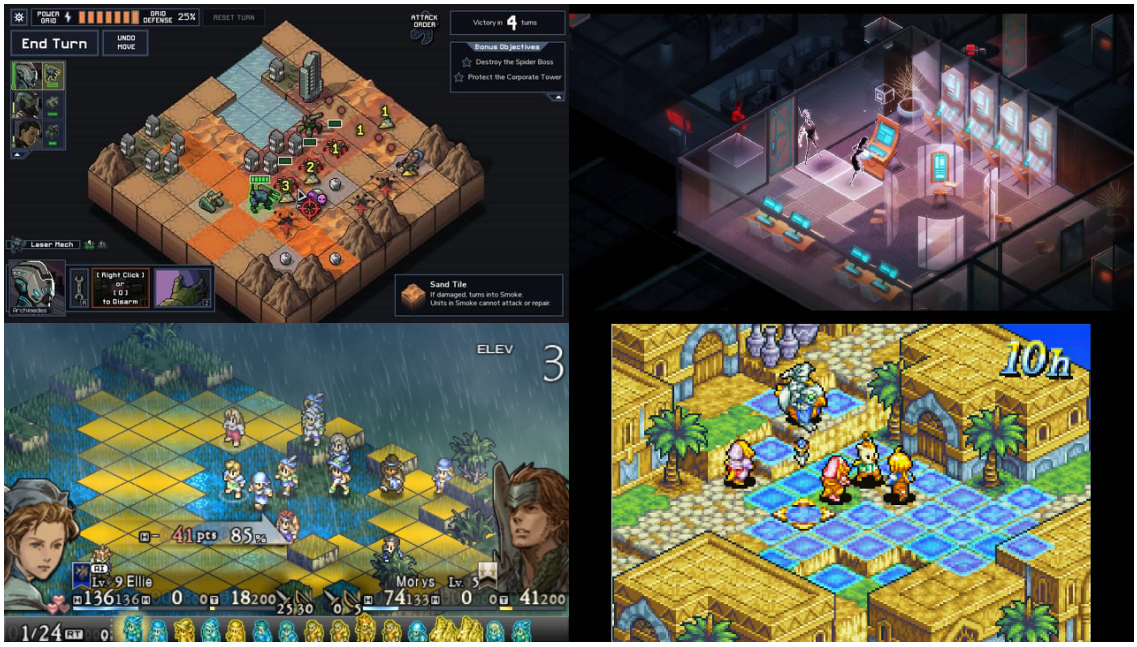

Figura 7.1: Juegos isométricos de combate por turnos. Into the Breach (2018), Invisible Inc (2015), Tactis Ogre (1995) y Final Fantasy Tactics Advance (2003) respectivamente.

y usarlas para capturar piezas del rival.

7.1.1. Diseño del juego de ajedrez

Para poder tener un prototipo funcional se requieren las siguientes funcionalidades: seleccionar piezas con el cursor, destacar las baldosas donde se puede mover la pieza seleccionada, mover la pieza al seleccionar una baldosa valida y eliminar las piezas que se encuentren en una baldosa que ocupará otra pieza.

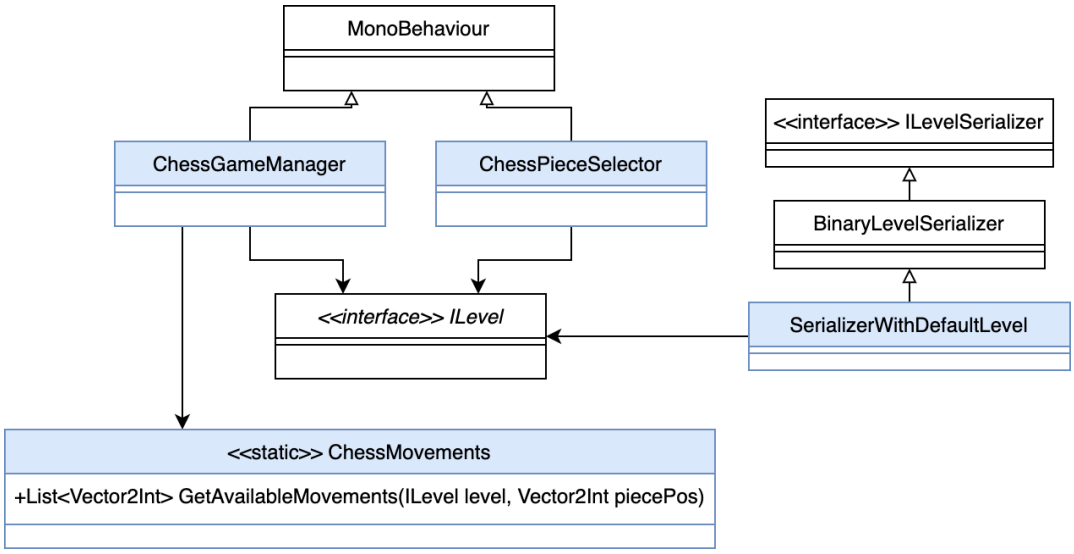

Figura 7.2: UML de las clases implementadas para el prototipo de ajedrez. Se muestran en azul las clases nuevas.

Para conseguir esto, el componente *ChessGameManager* interactuará directamente con el nivel para modificar la posición de las piezas y le pedirá a *ChessMovements* una lista con los movimientos validos para cada una. El componente *ChessPieceSelector* se encargara de que las piezas puedan ser seleccionadas con el cursor y *SerializerWithDefaultLevel* de preparar el tablero con las piezas en sus posiciones iniciales correspondientes.

7.1.2. Modelo del juego de ajedrez

Para comenzar la implementación del prototipo es necesario modelar los distintos elementos que lo componen. Se identificaron dos elementos distintos: las baldosas que componen el tablero y las piezas de ajedrez con las que se juega.

En el ajedrez cada pieza se mueve de manera diferente según su tipo y su color. Para poder modelar en el juego a cada una de las 12 combinaciones de tipo y color se les asignó un índice distinto. Además, para trabajar de manera mas cómoda, se creo una clase con métodos auxiliares que recibe el índice de una pieza y entrega su tipo o color.

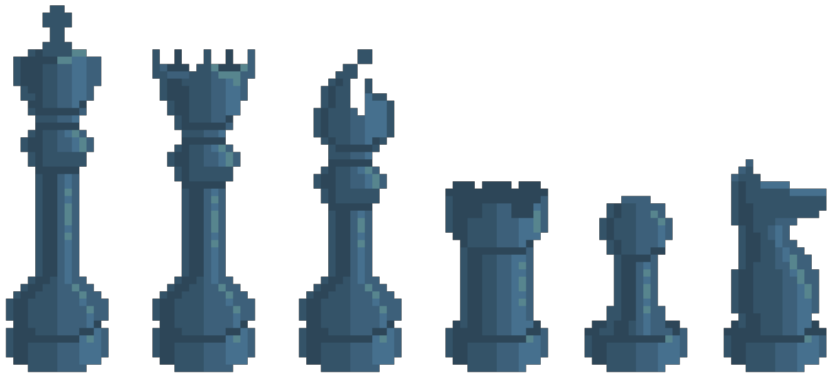

Figura 7.3: Sprites de piezas para el juego de ajedrez

Para calcular los posibles movimientos de cada una de las piezas dependiendo de su contexto se implementó la clase estática *ChessMovements*, que recibe una referencia al nivel y la posición en la que se encuentra la pieza que se quiere mover, con lo que obtiene el tipo y color de la pieza en aquella posición y utiliza esta información para calcular los lugares donde la pieza se puede mover o capturar piezas enemigas. Esta clase entrega como resultado una lista de coordenadas isométricas locales que representa todas las posibilidades de movimiento.

Para obtener la información necesaria para realizar este procedimiento, la clase realiza consultas al nivel modelado por las herramientas isométricas. Esto permite tomar en consideración la distribución de todas las piezas en el tablero para calcular los casos mas complejos, como por ejemplo el movimiento de un peón que solo puede ser en diagonal cuando tiene una pieza enemiga que pueda capturar.

Si bien el serializador ya incluido con las herramientas permite guardar niveles de ajedrez sin problemas, fue necesario implementar uno nuevo que crea un nivel con piezas distribuidas de la manera necesaria para iniciar una partida de ajedrez cuando se le pide cargar un nivel

que no existe. En todos los demás casos delega la solicitud al serializador binario incluido por defecto en las herramientas.

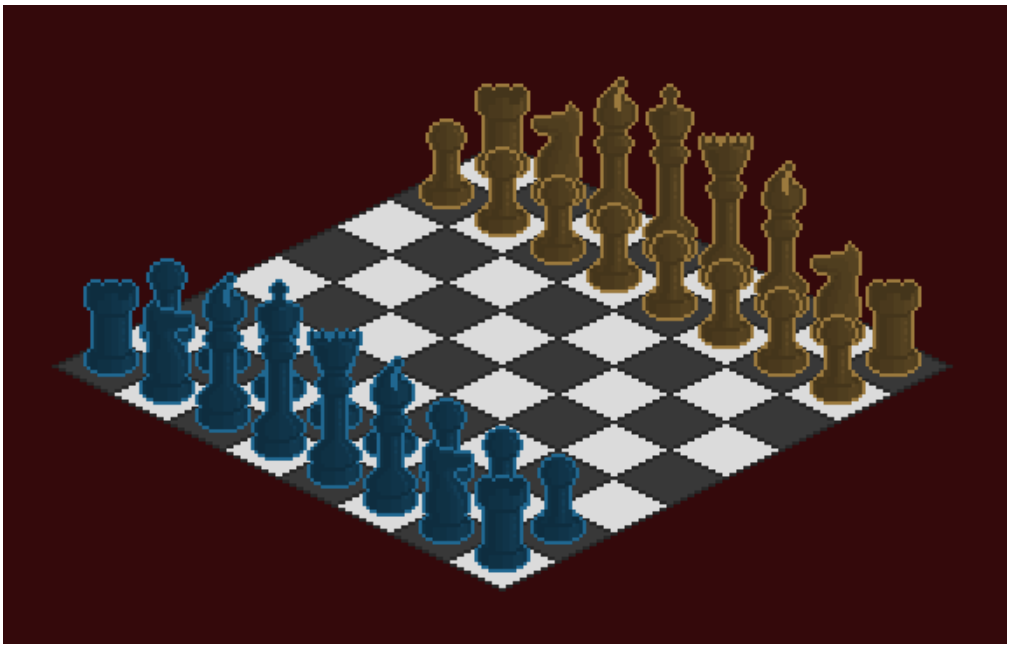

Figura 7.4: Distribución inicial de piezas en el tablero de ajedrez.

7.1.3. Visualización del juego de ajedrez

Para visualizar de manera correcta los sprites creados para el prototipo fue necesario indicarle a las herramientas el tamaño en pixeles del sprite de una baldosa. Esto es para asegurarse de que sistemas como el de cámara *Pixel Perfect* y el de conversión entre coordenadas isométricas, cartesianas y de pantalla sigan funcionando adecuadamente. Basta con escribir las dimensiones del sprite y las herramientas se encargan del resto.

Se creó un prefab distinto para cada pieza, lo que permitió reutilizar los sprites blancos y colorearlos utilizando mecanismos de Unity. Para hacer esto, al momento de crear el nivel se pregunta al modelo el color de la pieza que se está creando y se le asigna al SpriteRenderer. Si bien cambiar el color de un sprite es un caso de uso simple, este demuestra la integración entre las herramientas isomeétricas y los componentes nativos de Unity.

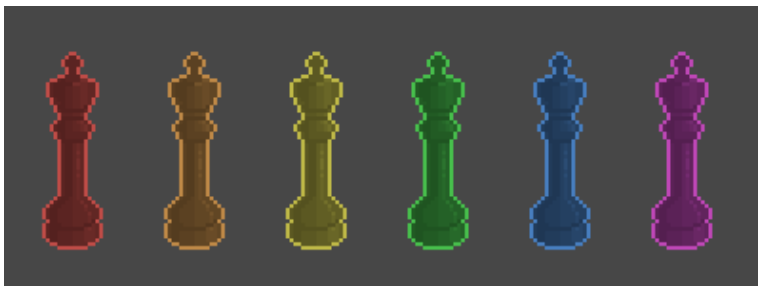

Figura 7.5: Re-coloración de un sprite mediante el SpriteRenderer de Unity.

7.1.4. Control de piezas de ajedrez

El nuevo componente *ChessPieceSelector* fue agregado a los Prefabs de todas las piezas de ajedrez. Este se encarga de utilizar callbacks de Unity para cambiar el color del sprite de la pieza cuando se pase el cursor sobre el y cuando se presione. También mantiene una variable estática con el registro de la pieza que está seleccionada actualmente para poder moverla si se selecciona una baldosa valida.

Para controlar el flujo de la partida, se implementó el componente *ChessGameManager*, el cual no se agregó a ningún elemento de las herramientas, sino que al nivel mismo de Unity. Este se encarga de destacar las baldosas a las que se puede mover la pieza seleccionada utilizando los métodos creados anteriormente.

En caso de presionar una baldosa donde la pieza se puede mover, el componente se comunicará con el modelo del nivel y le notificará del movimiento de la pieza. Con esto, los observadores de sprites modificaran automáticamente el estado del tablero.

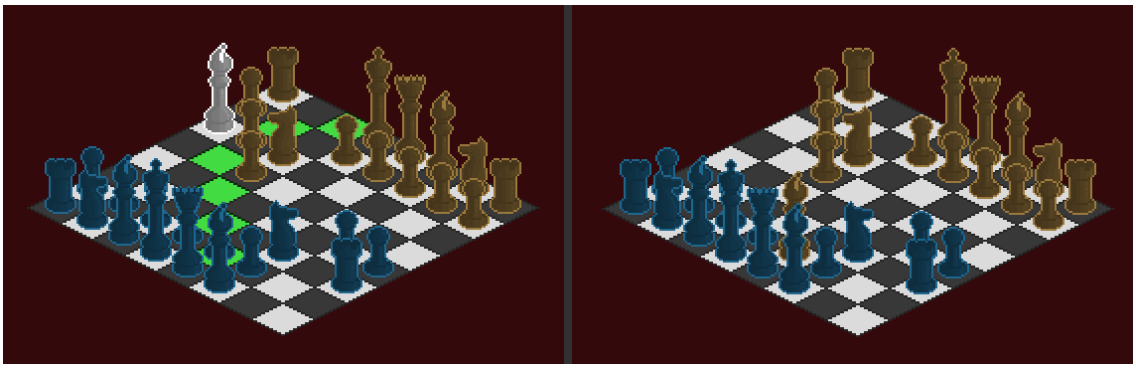

Figura 7.6: Muestra del prototipo de ajedrez. A la izquierda un alfil seleccionado mostrando en verde las casillas a las que se puede mover, a la derecha el mismo tablero luego de que el alfil haya capturado un peón.

De esta forma se puede ver que es posible la creación de un juego de estrategia por turnos utilizando las herramientas isométricas sin la necesidad de implementar funcionalidades especificas para la proyección isométrica.

7.2. Juego de Construcción de Niveles

Existe un grupo de juegos cuya mecánica principal se basa en la edición y construcción de los niveles. Este tipo de mecánicas es generalmente utilizada por juegos de simulación, donde el jugador se limita a construir un escenario y ver como interactúan los elementos que ha agregado. Por ejemplo, en el juego *SimCity 2000* el jugador construye una ciudad intentando mantener en equilibrio factores como su economía y población. Otro ejemplo es el juego *Theme Hospital*, en el que el jugador construye un hospital preocupándose de que sea rentable y entregue una buena experiencia a sus pacientes. Se pueden ver capturas de estos juegos en la Figura 7.7.

Otros juegos que se basan en la construcción del nivel son los de estrategia en tiempo real. En estos existe una interacción mas activa y directa del jugador sobre los distintos elementos

del nivel. Por ejemplo en el juego *Command and Conquer: Red Alert II* (Figura 7.7) el jugador modifica el nivel construyendo edificios para crear tropas que le permiten recolectar recursos para construir mas edificios. A diferencia de los juegos de simulación, aquí se tiene control directo de los personajes que habitan el nivel, pues se les asignan tareas individualmente.

Figura 7.7: Juegos cuya mecánica principal se basa en la construcción de niveles: Simcity 2000 (1993), Theme Hospital (1997), Command and Conquer: Red Alert 2 (2000) y They Are Billions (2017)

7.2.1. Prototipo a construir

El prototipo consiste en un pequeño juego en el que el jugador tendrá que crear y hacer crecer una aldea preocupándose de producir alimento y dinero para poder continuar expandiéndola. Para conseguir su objetivo el jugador podrá agregar tres elementos distintos al nivel: caminos, granjas y casas.

Para construir caminos el jugador tendrá que gastar 2 unidades de dinero. Las granjas costarán 15 unidades de dinero y las casas 5 unidades de alimento. Para construir casas también se tiene que cumplir un requisito especial, pues estas solo podrán ser construidas al lado de un camino. Además, ningún elemento podrá ser construido sobre un camino.

Los mismos edificios servirán para producir recursos, pues las casas producen una unidad de dinero cada cinco segundos y las granjas producen una unidad de alimento cada tres segundos.

Para construir elementos el jugador seleccionará con el cursor lo que quiere construir desde un menú situado en la parte inferior de la pantalla y presionará la parte del nivel donde quiera construirlo, como se puede ver en la Figura 7.8.

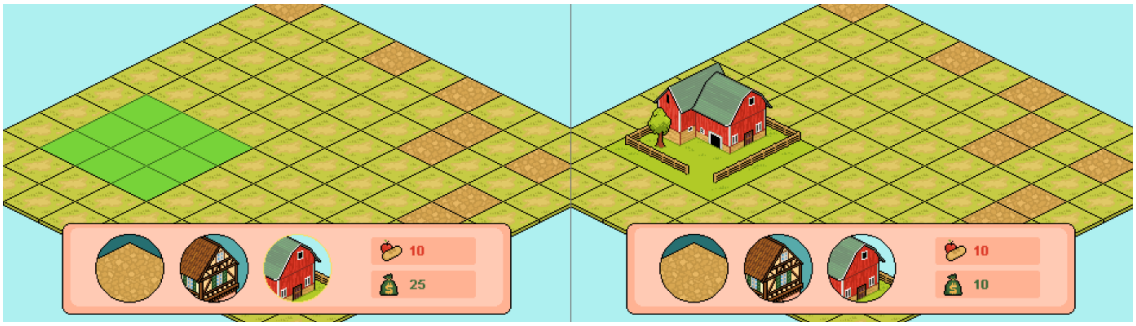

Figura 7.8: Proceso de construcción de edificios en el prototipo de creación de ciudades.

Como se muestra en la Figura 7.9, la mayor parte de la interacción con el nivel pasará por el componente *CityBuilderManager*, pues este se encargará de llevar el registro de los recursos del jugador y de comprobar que los edificios puedan ser construidos.

Los componentes *Farm* y *House* se incluyen en el Prefab de la granja y de la casa, respectivamente y le notifican a *CityBuilderManager* que estan produciendo recursos.

La clase *MapGenerator* genera un mapa aleatorio al iniciar el nivel y *ResourceIndicator* agrega una animación simple para entregar feedback al usuario.

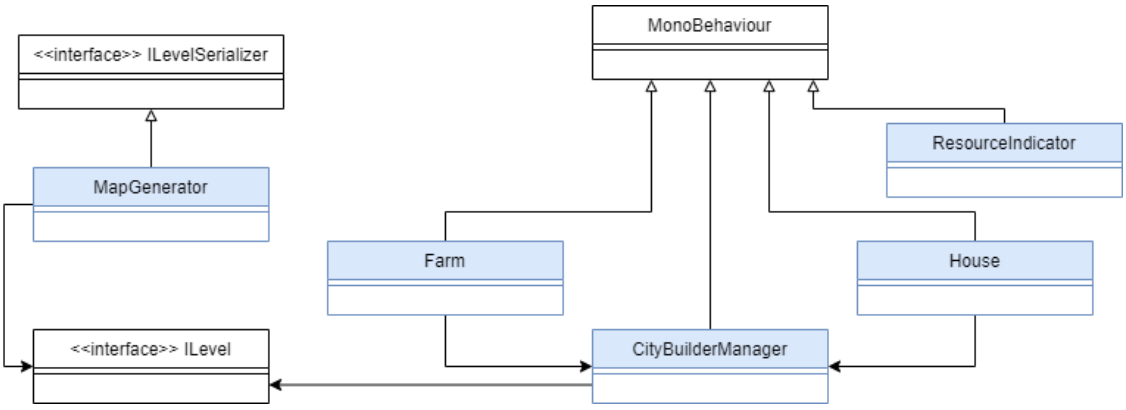

Figura 7.9: UML de las clases implementadas para el prototipo de construcción de niveles. Se muestran en azul las clases nuevas.

7.2.2. Modelo del prototipo de construcción de ciudades

Ya que los sprites que se utilizaron para las casas y la granja incluyen una baldosa debajo de ellos, se decidió modelar todo el prototipo utilizando solamente baldosas.

Para manejar el sprite de la granja que usa el espacio de nueve baldosas, se creó un nuevo tipo de baldosa que funcionará como *extensión de la granja* y no tendrá un sprite, pero impedirá construir cosas sobre ella. De esta forma, al construir una granja se asignará a una baldosa el tipo de granja y a las otras ocho el tipo de extensión.

Como se tienen dos sprites de casas, se optó por hacer que el controlador escoja uno de estos dos sprites al azar cuando encuentre una casa. Esto simplifica el modelo, pero tiene la

Figura 7.10: Sprites utilizados en el prototipo de creación de ciudades.

consecuencia de que al cargar un nivel los sprites de las casas no estarán distribuidos de la misma manera que cuando se guardó. Cabe mencionar que si esto es un problema, siempre se pueden incluir las distintas variaciones al modelo y escoger una al azar solo al momento de construir la casa.

Los niveles comienzan con una distribución aleatoria de baldosas de pasto y camino. Para generarlos se utilizó el mecanismo ya implementado de serializadores de forma que si el controlador de nivel le pide cargar un nuevo nivel, el serializador generará uno nuevo con la distribución especificada.

7.2.3. Prefabs de edificios

Se crearon Prefabs para las granjas y las casas, estos se agregarán al directorio de Prefabs personalizados para que las herramientas los carguen en las posiciones que indique el modelo.

Los componentes personalizados se agregaron a los Prefabs siguiendo el flujo normal de trabajo de Unity. Los componentes actualizan los recursos para cobrar el precio de construcción de los edificios al ser instanciados y agregan cada cierto tiempo los recursos que se van produciendo.

Para entregar feedback al usuario, los componentes también hacen que el color de los edificios cambie por una fracción de segundo al momento de producir un recurso y crean sprites indicadores con el símbolo del recurso que se está produciendo.

Estos indicadores se mueven hacia arriba y disminuyen su transparencia hasta desaparecer por completo gracias a un nuevo componente que se encarga de esto. En la Figura 7.11 se pueden ver los sprites indicadores sobre ambos edificios. Además se muestra el cambio de color en la granja que acaba de producir un recurso.

Durante todo el proceso de creación de Prefabs, el único momento en el que se tuvo que hacer algo distinto a lo que se haría al crear un juego normal en Unity fue cuando se pusieron los Prefabs en un directorio especial para poder ser instanciados por las herramientas isométricas.

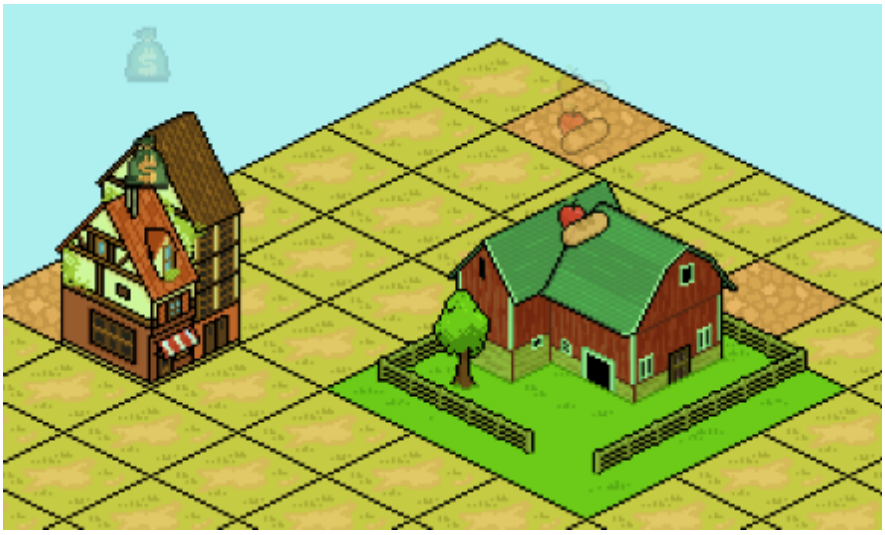

Figura 7.11: Edificios con sus sprites indicadores respectivos.

7.2.4. Construcción del nivel

Para permitir al jugador agregar edificios al nivel se creó una clase que toma una referencia al nivel y modifica directamente los datos de su modelo. Esta clase es un MonoBehaviour de Unity, lo cual le permite acceder al estado del ratón para saber donde y cuando agregar edificios.

Para seleccionar el tipo de edificio que se quiere construir se creó un panel con un botón para cada uno de los edificios y un indicador de los recursos actuales. Este panel se puede ver en la Figura 7.12.

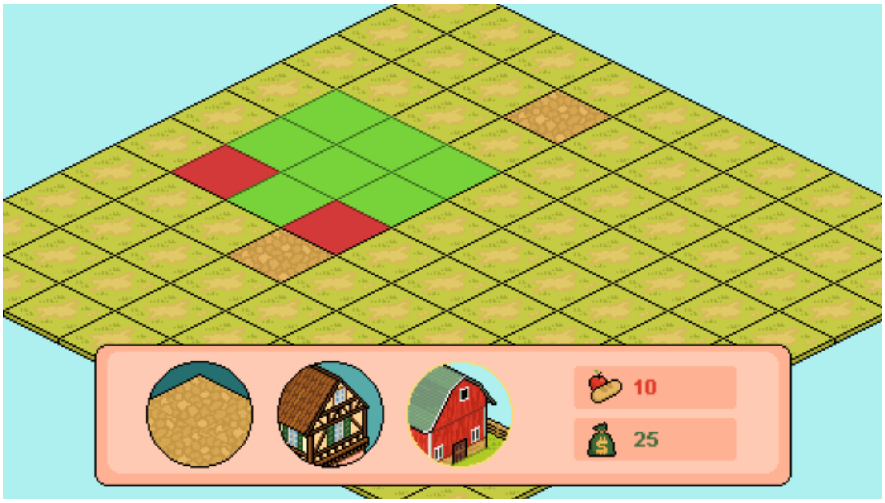

Figura 7.12: Indicador de factibilidad de construcción.

También es necesario entregar feedback al usuario antes de construir el edificio, para esto se implementó un sistema que destaca las baldosas en las que se está por construir. Estas se muestran verdes o rojas dependiendo de si es valida la construcción en aquel lugar o no. Por ejemplo, en la Figura 7.12 se muestra el indicador al querer construir una granja sobre dos

baldosas que son invalidas, pues tienen un camino.

De esta manera se pudo construir un prototipo funcional que implementa las mecánicas básicas de los juegos de construcción de niveles. Se puede ver el resultado final del prototipo en la Figura 7.13.

Figura 7.13: Muestra final del prototipo de construcción de niveles.

7.3. Movimiento complejo de personajes

Si bien se ha mostrado que el motor es flexible para distintos géneros, no se ha mostrado la capacidad de hacer uso intensivo de las características de Unity. Es por esto que este prototipo se concentra en crear un personaje complejo, controlado por el jugador, de apariencia personalizable, animado y que utilice el algoritmo de pathfinding implementado anteriormente.

Figura 7.14: Juegos donde el personaje principal tiene un movimiento complejo: Diablo (1997), Revenant (1999), Bastion (2011) y Hades (2018).

Existen varios juegos donde se necesitan personajes con comportamientos complejos, principalmente aquellos en los que el jugador controla a un único personaje. Un ejemplo de esto es el juego *Diablo*, el cual esta basado principalmente en caminar con un personaje derrotando enemigos y obteniendo mejor equipamiento, el cual modifica la apariencia del personaje. Tal fue el exito de *Diablo* que hasta el día de hoy se siguen creando juegos basados en el, como Basion y Hades, los cuales fueron publicados en el 2011 y 2018 respectivamente (Figura 7.14). La popularidad de estos juegos con personajes complejos demuestra la importancia de poder implementar estos sistemas utilizando las herramientas isométricas.

7.3.1. Diseño del prototipo

El prototipo consiste en un personaje que puede navegar por un nivel reproduciendo distintas animaciones para que parezca que camina en diferentes direcciones. Dentro del nivel, el personaje del jugador se encontrará con enemigos que caminan por el nivel con un sistema similar de animaciones. El jugador podrá presionar sobre una baldosa para caminar hacia ella y presionar sobre un enemigo para atacarlo, lo cual activará las animaciones correspondientes tanto en el personaje del jugador como en el enemigo.

El nivel tendrá obstáculos, los cuales serán esquivados automáticamente por el personaje del jugador utilizando el algoritmo de búsqueda de caminos implementado anteriormente. Además, el nivel también tendrá cofres que podrán ser presionados por el jugador para caminar hacia ellos y recibir mejoras que cambiaran el sprite de su personaje, por ejemplo, si el jugador recibe una espada podrá verla en la mano de su personaje.

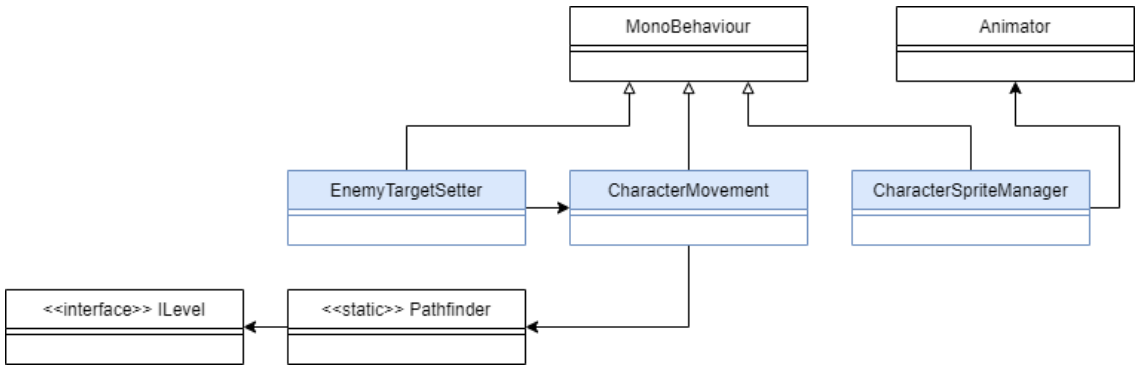

Figura 7.15: UML de las clases implementadas para el prototipo de movimiento complejo. Se muestran en azul las clases nuevas.

Este prototipo solo utiliza tres componentes nuevos: uno para mover al personaje por el nivel, otro para manejar sus animaciones y otro que se agrega a los enemigos para poder atacarlos al presionarlos con el ratón. No se requiere de mas componentes porque la mayor parte de las funcionalidades necesarias se implementan haciendo uso intensivo de componentes nativos de Unity, pues el objetivo principal de este prototipo es mostrar que las herramientas isométricas pueden interactuar con elementos complejos de Unity.

7.3.2. Creación de nivel

La generación de niveles se maneja de manera similar al prototipo anterior. Un generador implementa la interfaz del serializador de niveles y genera un nivel en el que todas las baldosas

son iguales. Luego, agrega aleatoriamente ítems que servirán como obstáculos para el jugador. El modelo para este prototipo es trivial, pues solo existe un tipo de baldosa y dos tipos de ítem: cofres y obstaculos.

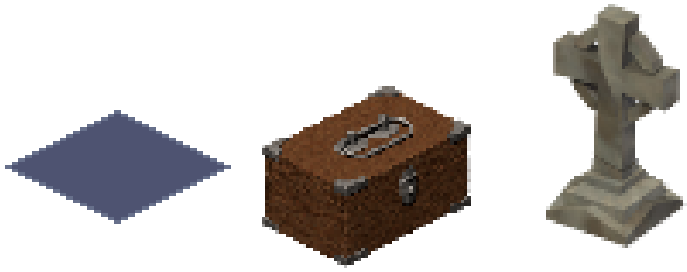

Figura 7.16: Sprites utilizados para el prototipo de movimiento complejo.

7.3.3. Animación del personaje

El personaje tiene la capacidad de caminar el ocho direcciones, por lo que para cada animación que se quiera incluir se necesitaran ocho versiones diferentes. Con el objetivo de poder moverse por el nivel e interactuar con los enemigos se utilizarán tres animaciones: mantenerse en el lugar, correr y golpear, esto significa que se necesitarán 24 variaciones de animaciones en total.

Figura 7.17: Personaje del jugador orientado en sus 8 direcciones diferentes.

Para manejar las animaciones se utilizó el sistema estándar de Unity: se agruparon los distintos sprites convirtiéndolos en clips de animación y se utilizó el componente *Animator* para decidir que clip que se debe reproducir. El componente *Animator* funciona de manera similar a un autómata, pues en el se definen estados que indican la animación que se reproducirá y transiciones que regulan los cambios entre estados. Este componente recibe un conjunto personalizable de parámetros y en base a ellos se activan las transiciones.

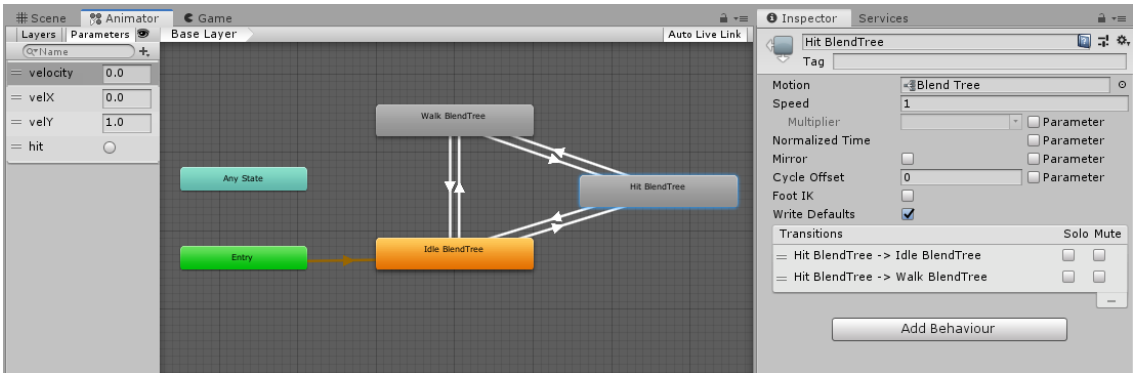

Figura 7.18: Visualización del *Animator* del personaje en la interfaz de Unity.

Como se puede ver en la Figura 7.18, el *Animator* solo tiene tres estados, esto es porque cada uno de estos tiene 8 sub-estados para las distintas direcciones. Para escoger los sub-estados se utilizó un *Blend Tree Bidimensional* que permite asignarle una dirección a cada una de las animaciones para luego escoger la mas cercana al movimiento que está realizando el personaje.

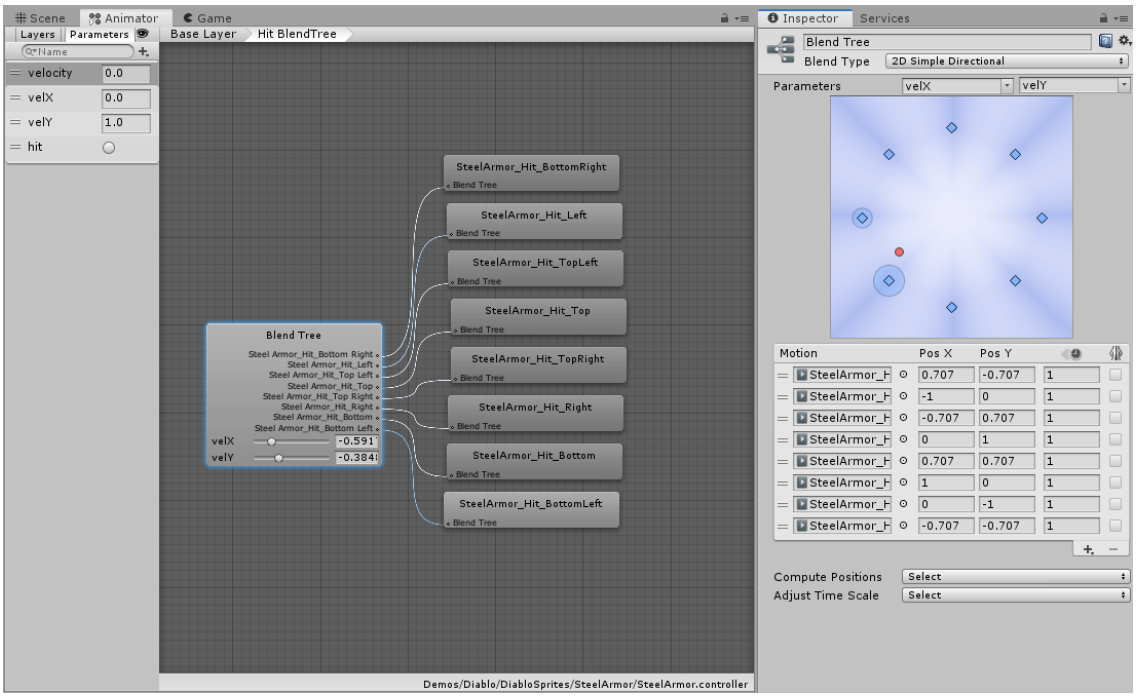

Figura 7.19: Visualización del *Blend Tree* del personaje en la interfaz de Unity.

Todos estos elementos trabajaran en conjunto para animar al personaje del jugador en base al los caminos que se encuentren para navegar el nivel isométrico.

7.3.4. Personalización del personaje

Otra característica de los juegos que se desea simular con este prototipo es la posibilidad de personalizar el personaje modificando su apariencia. Para incluir esto en el prototipo se permitirá al jugador cambiar su armadura y recoger una espada.

La forma de conseguir esto es separando los sprites en las distintas partes que se quieren poder modificar, para luego ponerlos uno encima del otro como si fueran distintas capas de la misma imagen. Así, es posible reemplazar por ejemplo, la capa de la armadura por una con otro diseño. La única desventaja de un sistema como este es que cada una de las partes necesitará su propio *Animator* con los clips que le corresponden.

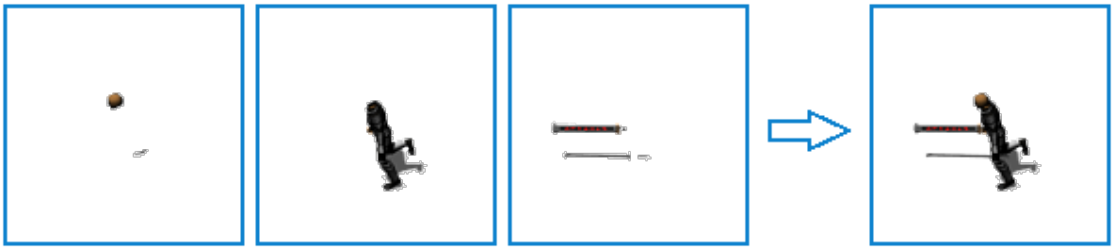

Figura 7.20: Construcción de un sprite modularmente mediante sus distintas capas.

Cada una de estas capas debe estar sincronizada con las otras para conseguir una animación final coherente. Para conseguirlo se creó un componente que calcula la velocidad de movimiento del personaje en base al camino que está siguiendo y lo comunica al mismo tiempo a todos los *Animators* que lo componen.

Este componente también permite reemplazar los *Animators*, pues esto permite cambiar en tiempo de ejecución las distintas capas de la animación. Esto será utilizado cada vez que el jugador abra un cofre, pues en uno de ellos encontrará una espada y en el otro una mejor armadura.

Figura 7.21: Distintas variaciones del mismo sprite del jugador.

7.3.5. Movimiento del personaje

La parte en la que este Prefab complejo interactúa con las herramientas isométricas en su movimiento, pues para calcular el camino que tomará el personaje cuando el jugador presiona una baldosa se utiliza el sistema de pathfinding implementado en el Capítulo anterior. Este sistema hace consultas al modelo del nivel para poder encontrar el camino mas corto hacia su objetivo evitando los obstáculos.

Tanto para obtener las coordenadas de la baldosa objetivo como para saber donde posicionar al personaje se utiliza el transformador de coordenadas del nivel. A este también se

le pregunta el orden de renderizado del personaje, lo que permite incorporarlo de manera coherente al nivel.

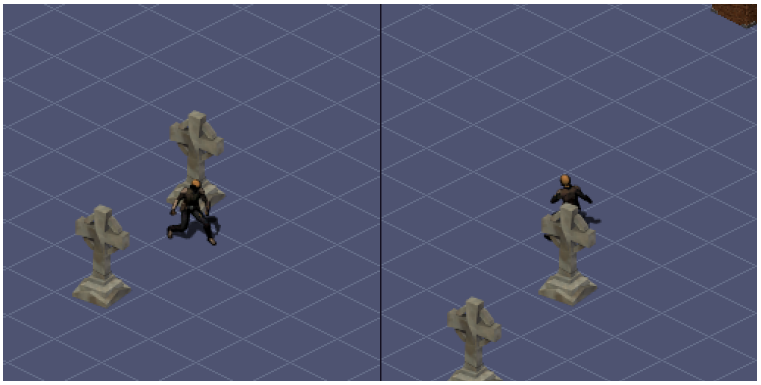

Figura 7.22: Ajuste del orden de renderizado del jugador dependiendo de su posición respecto al nivel isométrico.

Para hacer que el jugador camine hacia objetivos con posición dinámica se crearon enemigos. Para ellos se reutilizaron todos los componentes que se utilizaron para el personaje del jugador a excepción del encargado de asignar un objetivo al presionar una baldosa. Los objetivos de los enemigos se generan aleatoriamente cada cierto tiempo para simular que deambulan por el nivel.

Al presionar sobre un enemigo el personaje calculará su ruta en cada fotograma tomando como posición objetivo la posición isométrica que corresponde al enemigo. Además cuando el jugador alcance a un enemigo se activará la animación de golpear y el enemigo se eliminará luego de reproducir su animación de derrota.

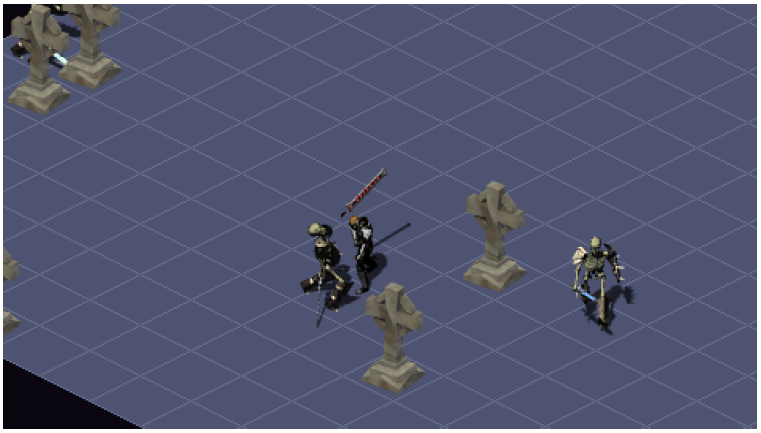

Figura 7.23: Personaje del jugador atacando a un enemigo.

Finalmente, se importó y agregó un componente externo que se encarga de mover la cámara suavemente hacia la posición del jugador para mantenerlo siempre dentro de la pantalla de juego. Este componente también se incorporó sin problemas al prototipo.

7.4. Juego en línea

Un aspecto de importancia significativa los videojuegos en general es la posibilidad de jugar con otras personas por internet. La mayoría de los juegos isométricos mencionados anteriormente incluyen un modo multijugador, además existen juegos cuyas principales mecánicas giran al rededor de la interacción en línea, conocidos como juegos sociales.

Un ejemplo de un juego social es Habbo, pues en este juego los jugadores construyen niveles isométricos que simulan ser habitaciones de un hotel, las cuales tienen como única función reunir jugadores para conversar. Otro ejemplo es FarmVille, un juego cuyo objetivo es construir y manejar una granja en el cual la forma mas eficiente de progresar es visitando las granjas de otros amigos y ayudándose mutuamente.

Figura 7.24: Ejemplos de juegos sociales: a la izquierda Habbo (2000) y a la derecha FarmVille (2009).

Para validar la posibilidad de crear juegos en línea con las herramientas isométricas se implementa un prototipo en el cual los jugadores podrán interactuar con otros en línea editando un mismo nivel isométrico, caminando por él y conversando mediante texto.

7.4.1. Diseño del prototipo

Las funcionalidades que cumplirá el prototipo para mostrar que es posible usar las herramientas isométricas para hacer videojuegos en línea son: mostrar un nivel y permitir editarlo localmente, guardar el nivel en un servidor y sincronizarlo entre varios clientes, agregar un personaje para cada jugador que pueda caminar por el nivel y ser observado por los demás clientes, y permitir a los jugadores conversar mediante mensajes de texto.

Un prototipo así requiere de un cliente que interactúa con varios servicios distintos, un servidor http simple que almacenará y entregará los datos del nivel a cargar, un servidor de tiempo real que se encargará de sincronizar a los jugadores caminando por el nivel y un servidor de chat que se encargara enviar los mensajes de los distintos clientes.

Además, el servidor http manejará cuentas de usuario, por lo que será usado tanto por el cliente como los otros servicios para administrar el proceso de autenticación.

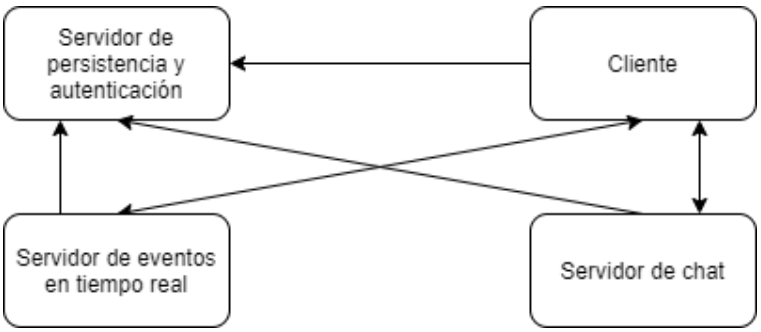

Figura 7.25: Diagrama de interacciones entre los distintos servicios para el prototipo de juego en línea.

Para esto, el prototipo necesita de varios elementos que trabajan en conjunto. Las clases *ChatWindow*, *ChatInterface* y *ChatInputField* se encargan de las conversaciones entre usuarios; para cargar los niveles desde el servidor web se usan las clases *WebLevelData*, *PersistentAPI* y *WebLevelSerilizer*; por último, la clase *CharacterMovement* administra el movimiento de los personajes y sincroniza sus posiciones en todos los clientes.

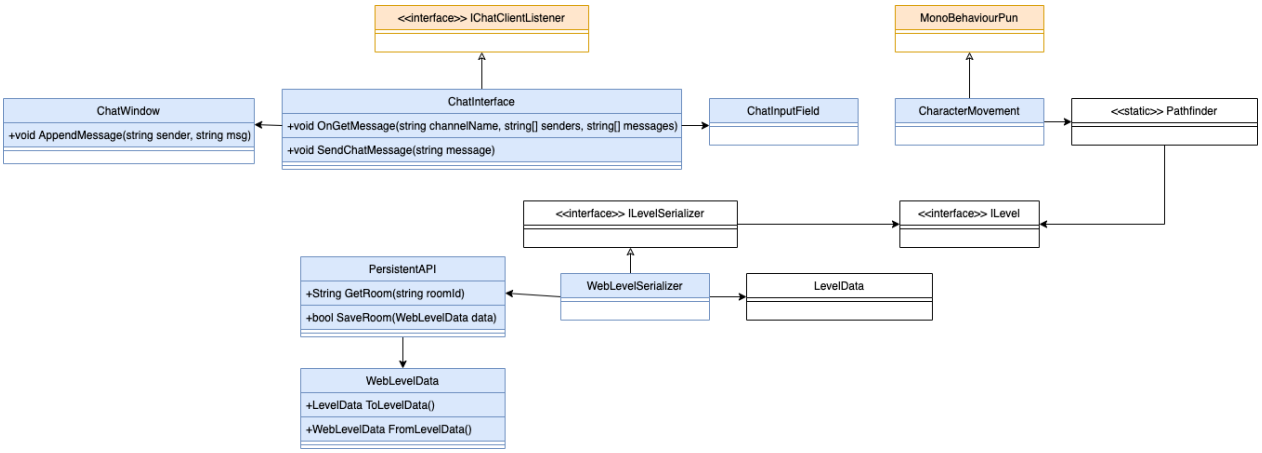

Figura 7.26: UML de las clases implementadas en el prototipo de conexión en línea. Se muestran en azul las clases nuevas y en naranjo las clases importadas desde librerías externas.

7.4.2. Persistencia del nivel

Para almacenar los niveles isométricos se utiliza un servidor básico construido sobre Django Rest. Este servidor maneja las cuentas de usuario y los niveles almacenándolos en una base de datos SQLite.

Los niveles se almacenan en el servidor de forma similar a como se hace en el cliente, pues se utilizan cuatro listas: una para guardar los índices de las baldosas, otra para índices de las murallas, una para los ítem y otra para sus orientaciones, además de dos enteros para el ancho y alto del nivel. La única diferencia entre ambos modelos es que estas listas se tuvieron que aplanar porque SQLite no soporta arreglos multidimensionales.

El servidor expone un endpoint que permite crear, modificar y consultar niveles mediante solicitudes GET, POST y PUT. Este entrega las respuestas en formato Json, las cuales se

convierten en el cliente a una instancia de la clase *WebLevelData*.

En el cliente, un serializador realiza las solicitudes al servidor y convierte los objetos *WebLevelData* a *LevelData*, los cuales se entregan al nivel para que los utilice de manera normal. Para guardar los cambios del nivel se realiza el proceso inverso.

Dentro del cliente se pueden editar los niveles mediante el uso de la funcionalidad implementada anteriormente en el editor de niveles (Capítulo 4.15). Estos cambios serán vistos por todos los clientes que carguen el nivel en el futuro.

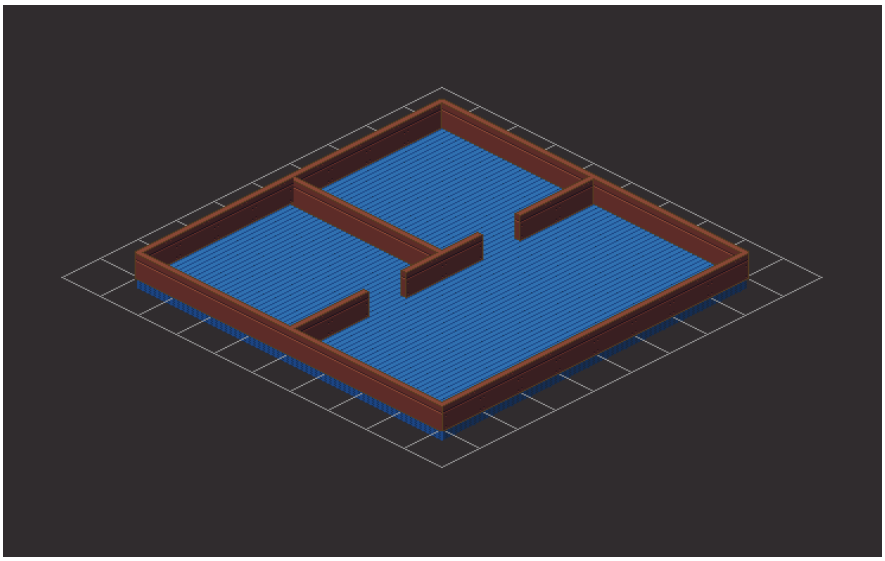

Figura 7.27: Nivel cargado desde un servidor externo.

La autenticación de los usuarios se maneja mediante tokens. En este sistema el usuario envía una solicitud indicando un correo y una contraseña, el servidor revisa que la cuenta sea valida y, en caso de que lo sea, responde con un token el cual podrá ser usado por el cliente para indicar que ya se encuentra autenticado.

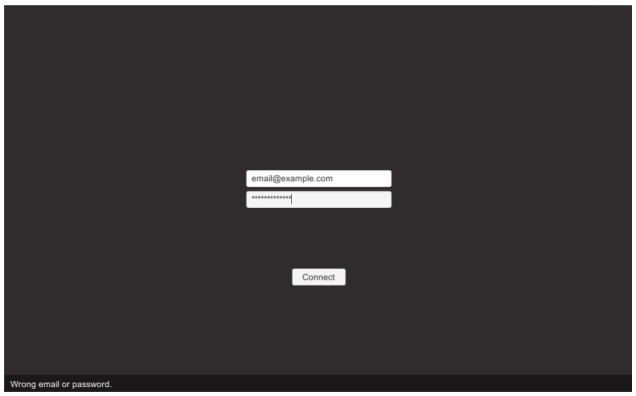

Figura 7.28: Pantalla de autenticación en el cliente del prototipo en línea.

En el cliente se agregó una pantalla simple de autenticación con campos de texto para el correo y la contraseña, además de un texto informativo que indica el estado del proceso de autenticación, como se muestra en la Figura 7.28.

7.4.3. Eventos en tiempo real

Para enviar eventos en tiempo real entre clientes se decidió utilizar un motor de conexión multijugador en línea diseñado especialmente para videojuegos. Esta decisión se tomó con el objetivo de simular de mejor manera un escenario real de uso de las herramientas isométricas, pues es común que los desarrolladores opten por este tipo de servicios.

El motor elegido es Photon Engine, pues tiene una versión diseñada específicamente para integrarse con Unity de manera sencilla. La forma de interactuar entre clientes utilizando Photon es mediante funciones especiales a las que se les da el nombre *RPC* que es la sigla de *Remote Procedure Call*, estas son funciones que se ejecutan en todos los clientes al mismo tiempo, lo que permite enviar información entre clientes mediante sus parámetros.

Para evitar la sobrecarga tanto de los clientes como del servidor, los jugadores que se conectan a Photon se agrupan en salas y cada cliente solo recibe los eventos de otros jugadores que estén en su misma sala. Ya que no se espera que el prototipo sea utilizado masivamente, se creó una sola sala a la que entrarán todos los clientes que inicien una conexión.

Para representar al jugador, se creó un personaje simple con un sprite sin animación que se mueve por el nivel utilizando el sistema de búsqueda de caminos implementado anteriormente. Para sincronizar el movimiento de los personajes a través de todos los clientes se utiliza una *RPC* que se activa cuando un cliente presiona con el cursor sobre una baldosa. Este evento se envía a todos los otros clientes indicando el cliente que la activó y la posición donde se presionó. Luego, cada uno de los clientes calcula el camino que el personaje debe tomar para llegar a su destino y lo mueve gradualmente.

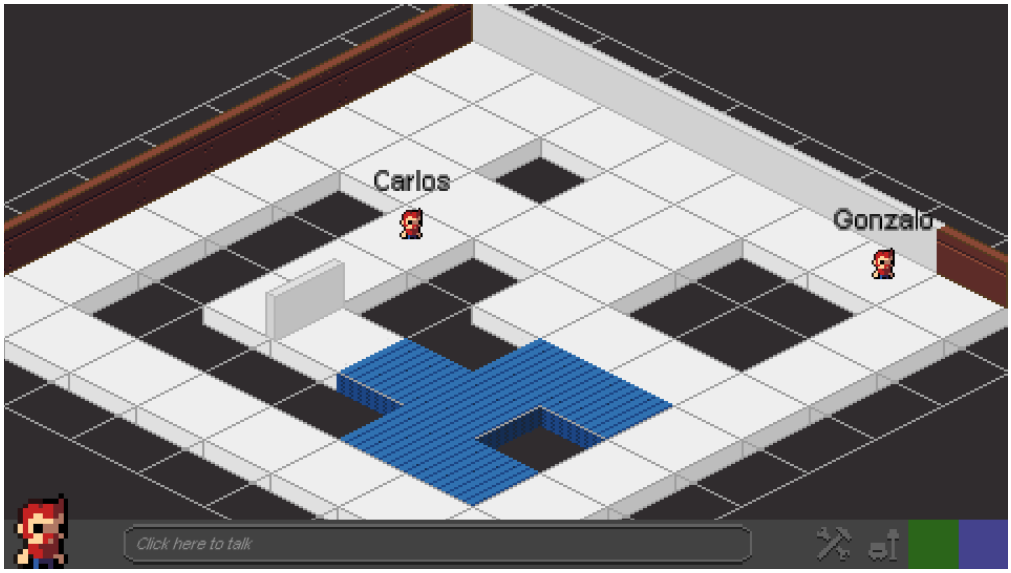

Figura 7.29: Dos jugadores conectados en un mismo nivel.

Finalmente, la integración con el sistema de autenticación fue realizada indicando a Photon la dirección del endpoint de autenticación creado anteriormente. De esta forma Photon revisará si el cliente tiene credenciales validas antes de conectarlo a una sala.

7.4.4. Conversaciones en tiempo real

Para las conversaciones entre jugadores se decidió por usar otro servicio externo: Photon Chat. Este servicio está pensado para integrarse fácilmente tanto a Unity como a Photon Engine. De hecho, utiliza el mismo método de autenticación que Photon Engine por defecto, esto significa que no es necesario agregar configuraciones extra para hacer que use el servidor de persistencia.

Photon Chat trabaja con un sistema similar al de Photon Engine para separar los mensajes de los jugadores en grupos, en este sistema los jugadores se unen a distintos canales de conversación para recibir sus mensajes. La diferencia entre este sistema y el de salas, es que un mismo jugador se puede unir a varios canales distintos para mantener conversaciones simultaneas con varios grupos de jugadores.

Como el prototipo espera tener una cantidad muy baja de trafico, todos los jugadores se suscribirán a un mismo canal al iniciar el cliente. Por el lado de la interfaz gráfica, se agregó un campo de texto donde los jugadores podrán enviar sus mensajes y un panel donde saldrán los mensajes de todos los jugadores. Para recibir los mensajes, se implementó en el cliente una clase que utiliza una interfaz provista por Photon Chat y agrega los mensajes a la interfaz gráfica.

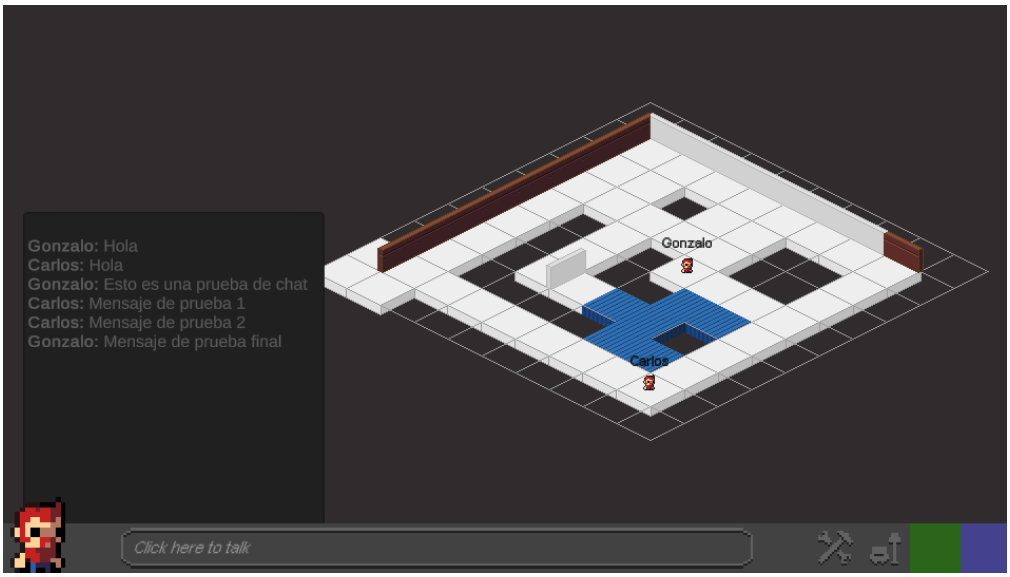

Figura 7.30: Dos jugadores enviando mensajes de texto en un mismo nivel.

Con esto se demuestra la factibilidad de integrar las herramientas isométricas a un juego con funcionalidades en línea.

Conclusión

El objetivo general de esta memoria era disminuir el trabajo de los desarrolladores evitando que implementen funcionalidades comunes que podían ser generalizadas y reutilizadas. Se implementaron sistemas que permiten simular un espacio tridimensional con imágenes bidimensionales utilizando la proyección isométrica de una manera natural y que permiten interactuar con los distintos elementos de los escenarios isométricos sin restricciones. Como se pudo comprobar al construir los prototipos de validación, las herramientas permitieron incorporar distintas funcionalidades comunes de los juegos isométricos sin reimplementarlas y con una cantidad mínima de modificaciones al código del software que las utiliza. Por lo tanto, es posible afirmar que se cumplió en su totalidad el objetivo general. Esto finalmente se traduce en una reducción de tiempo de desarrollo de los proyectos que la utilicen, lo cual es importante porque en los proyectos de desarrollo de software el tiempo es uno de los recursos mas escasos.

Los objetivos específicos ayudaron a cumplir el objetivo general de mejor manera. El objetivo de permitir la creación de niveles, su modificación y la interacción con sus elementos tuvo especial importancia por su parte de interacción con los elementos del nivel, pues esto es lo que diferencia las herramientas implementadas de las otras opciones disponibles. Esta interacción en tiempo de ejecución permite al desarrollador delegar las responsabilidades propias de la perspectiva isométrica en su completitud, mientras que las otras alternativas disponibles solo permiten previsualizar niveles e importarlos como si fueran una fotografía. El objetivo de diseñar una arquitectura genérica que permita implementar las herramientas en otros motores sugirió enfrentar el problema en primera instancia de una manera general, lo que permitió identificar los problemas principales con mayor facilidad para tenerlos en cuenta en el momento de diseñar e implementar la versión final, lo cual fue muy positivo. El objetivo de mantener la flexibilidad de las herramientas para que se adapten a distintos tipos de juegos benefició directamente al objetivo general, pues mientras más tipos de juegos sean soportados por las herramientas, más desarrolladores podrán disminuir su trabajo. El objetivo de permitir que las herramientas sean extensibles y se puedan integrar con otras tecnologías permitió disminuir la cantidad de trabajo de los desarrolladores, pues de no poder integrar las herramientas con otras tecnologías los desarrolladores tendrían que decidir entre reimplementar por su cuenta las herramientas o la librería externa que deseen utilizar, lo cual es un gasto de recursos importante. De manera similar, el objetivo de lograr que las herramientas sean independientes entre sí, permitió disminuir la cantidad de código escrito por los usuarios dado que elimina la necesidad de inicializar herramientas que no se van a usar. Finalmente, el objetivo de que las herramientas compartan la filosofía de diseño del motor aportó significativamente al objetivo general, pues disminuye la curva de aprendizaje

del uso de las herramientas y la cantidad de cambios que deben ser realizados a un proyecto ya existente que quiere incorporarlas, además de disminuir las complicaciones de diseño a las que se puede tener que enfrentar un desarrollador que quiera utilizar librerías con patrones fundamentales de diseño discordantes. Cabe mencionar que las modificaciones realizadas a la arquitectura general en la etapa de optimización jugaron un rol importante en el cumplimiento de este objetivo.

Respecto a donde se ven reflejados los resultados de los objetivos específicos, se mostró que las herramientas se incorporan con normalidad al flujo de desarrollo en Unity con los prototipos de ajedrez y construcción de ciudades, pues en estos se trabaja utilizando Prefabs y componentes de la misma manera que en un proyecto regular de Unity. La modularidad de las herramientas se pudo ver en el prototipo de movimiento complejo de personajes, pues en este se usa un pequeño subconjunto de las herramientas dejando de lado elementos como la visualización de murallas y la interacción con baldosas. La extensibilidad de las herramientas se muestra en el prototipo de videojuego en línea, pues para construirlo se tuvieron que modificar los serializadores de niveles para que respondan y realicen solicitudes http. Finalmente, la flexibilidad de las herramientas fue validada por todos los prototipos en conjunto, pues con ellos se abarcó un amplio rango de géneros y mecánicas, las cuales pudieron ser implementadas sin limitaciones impuestas por las herramientas.

La separación del trabajo en dos iteraciones fue de gran utilidad para obtener una evaluación temprana del trabajo realizado y permitió reconocer las falencias de las herramientas para poder mejorarlas en la segunda iteración. Además de servir como un claro limite entre el diseño de herramientas para ser usadas por cualquier motor y las optimizaciones especificas para Unity.

La separación de funcionalidades en componentes resultó ser exitosa para lograr el objetivo de modularidad, pues al seguir las guías de desarrollo de Unity se consiguió un conjunto de herramientas que no requieren incluir mas de lo necesario en el proyecto al que se incorporarán. Además, esto permitió una fácil interacción con Unity que se hace natural a los desarrolladores acostumbrados a ese ambiente.

Trabajo a Futuro

Actualmente las herramientas no soportan baldosas con distintas alturas ni ítems que utilicen mas de una baldosa, las cuales son funcionalidades utilizadas con relativa frecuencia en los videojuegos isométricos. Si bien, las herramientas se pueden extender para trabajar con este tipo de sistemas, realizar esto no es tan sencilla como si se hubieran implementado nativamente en las herramientas. Implementar esta funcionalidad dentro de las herramientas permitiría optimizar una mayor cantidad de casos de desarrollo.

Otra posibilidad interesante es la de implementar las herramientas en otro motor de videojuegos. Esto permitiría evaluar y refinar la arquitectura desarrollada en la primera iteración. Además, permitiría hacer llegar las herramientas a una mayor cantidad de desarrolladores que las necesiten.

Bibliografía

[1] Gamebase Co. Gamebryo, 01 2020. http://www.gamebryo.com/gamebryo.php.

[2] Itch Corp. Most used Engines, 01 2020. https://itch.io/game-development/engines/most-projects.

[3] Kadokawa Corporation. RPG Maker | Make Your Own Video Games!, 01 2020. https://www.rpgmakerweb.com/products/programs/rpg-maker-mv.

[4] Crytek. CRYENGINE | The complete solution for next generation game development by crytek, 01 2020. https://www.cryengine.com/.

[5] Cyberglads. Making Cyberglads 2: first steps in Godot, 02 2020. https://cyberglads.com/making-cyberglads-2-first-steps-in-godot.html.

[6] Free Software Foundation. The GNU Genral Public License, 03 2020. http://www.gnu.org/licenses/gpl-3.0.html.

[7] Interactive Fiction Technology Foundation. Twine / An open-source tool for telling interactive, nonlinear stories., 01 2020. https://twinery.org/.

[8] A. Hinton-Jones. Isometric 2D Environments with Tilemap, 02 2020. https://blogs.unity3d.com/2019/03/18/isometric-2d-environments-with-tilemap/.

[9] A. Ledoux. Bitsy, 02 2020. http://ledoux.io/bitsy/editor.html.

[10] Dbolical Pty Ltd. IW Engine, 01 2020. https://www.moddb.com/engines/iw-engine.

[11] Mento. AnvilNext Game Engine (Concept), 01 2020. https://www.giantbomb.com/anvilnext-game-engine/3015-3972/.

[12] The Freeciv Project. Freeciv.org - open source empire-building strategy game., 06 2020. http://www.freeciv.org/.

[13] XBOX Game Studios. Age of Empires Franchise - Official Website, 06 2020. https://www.ageofempires.com/.

[14] M. Toftedahl. Which are the most commonly used Game Engines?, 09 2019. https://www.gamasutra.com/blogs/MarcusToftedahl/20190930/350830/

Which_are_the_most_commonly_used_Game_Engines.php.

[15] Valve. Source, 01 2020. https://developer.valvesoftware.com/wiki/Source.

[16] B. Þorsteinsson. Sokoban: levels, 03 2020. http://borgar.net/programs/sokoban/#Intro.