UN CONJUNTO DE HERRAMIENTAS PARA UNITY ORIENTADO AL DESARROLLO DE VIDEOJUEGOS DE ACCIÓN-AVENTURA Y ESTILO RETRO CON GRÁFICOS ISOMÉTRICOS 3D

D. Iván José Pérez Colado
D. Víctor Manuel Pérez Colado

GRADO EN INGENIERÍA DEL SOFTWARE
FACULTAD DE INFORMÁTICA
UNIVERSIDAD COMPLUTENSE DE MADRID

TRABAJO DE FIN DE GRADO DE INGENIERÍA DEL SOFTWARE

Madrid, 20 junio de 2014

Director: Prof. Dr. D. Federico Peinado Gil

Autorización de difusión y utilización

Iván José Pérez Colado y Víctor Manuel Pérez Colado autorizamos a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, los contenidos audiovisuales incluso si incluyen imágenes de los autores, la documentación y/o el prototipo desarrollado.

Fdo. Iván José Pérez Colado Fdo. Víctor Manuel Pérez Colado

Para nuestro nono
Quien con su empeño e ilusión, fue un motivo en nuestras vidas para llegar hasta donde hemos llegado y no está hoy con nosotros para verlo

Agradecimientos

Queremos agradecer, en primer lugar, a nuestro director de proyecto Federico Peinado Gil, por darnos la oportunidad de participar en un proyecto del campo que más nos apasiona, los videojuegos, y por toda la ayuda y dedicación que nos ha prestado durante el curso para completarlo.

En segundo lugar, agradecemos a nuestros amigos y familia, su opinión y continua valoración de los resultados que íbamos obteniendo durante la realización del proyecto.

Por último, agradecer a la comunidad de usuarios de Unity, y a los usuarios de StackOverflow, la cesión de múltiples fragmentos de código que han servido de referencia para la realización del proyecto.

Gracias a todos por vuestra colaboración.

Índice

Índice de figuras

Resumen

El sector de los videojuegos ha sido desde los años 70 uno de los campos más importantes dentro de la Informática. A lo largo de su historia, sus complejos y costosos desarrollos hacían muy habitual que pequeños e independientes proyectos fracasaran. Los nuevos motores comerciales de videojuegos han abierto las puertas a nuevas formas de desarrollo de videojuegos destinadas a ser multiplataforma y explotar al máximo las capacidades de los diferentes dispositivos. Con el auge de los nuevos motores de videojuegos, los desarrollos independientes han florecido hasta ser actualmente una de las alternativas más relevantes de la industria para obtener el éxito. Con el objetivo de ayudar a estos desarrolladores y recuperar el estilo retro de obras memorables como La Abadía del Crimen o Final Fantasy Tactics Advance, así como potenciar con la videoaventura el ingenio y la curiosidad del jugador, se han desarrollado un conjunto de herramientas para el motor de videojuegos Unity que simplifican la creación de este tipo de juegos con gráficos isométricos 3D. Concretamente estas herramientas dan soporte a las labores de construcción, pintado y decoración de mapas así como a la creación de entidades y el desarrollo del argumento del juego a través de cinemáticas y secuencias interactivas. Todo ello, ha sido pensado hacia su máxima capacidad extensible para poder ser ampliado por miembros de la comunidad o futuros usuarios del proyecto.

Palabras clave: Videojuegos, Motores de videojuegos, Herramientas para desarrollo de videojuegos, Extensiones de motores de videojuegos, Desarrollo independiente de videojuegos, Videojuegos en perspectiva isométrica, Videojuegos de estilo retro, Desarrollo multiplataforma

Abstract

The gaming industry has been since the 70s one of the most important fields in computer science. Throughout its history, its complex and expensive developments made very common that small and independent projects fail. New commercial game engines have opened the door to new ways of game development designed to be cross-platform exploiting the capabilities of multiple devices. With the rise of the new game engines, independent developments have growed up to be one of the most relevant alternatives industry for success. In order to help these developers and recover the retro style of memorable works as La Abadía del Crimen or Final Fantasy Tactics Advance and enhance videoadventure with wit and curiosity of the player, have developed a set of tools for Unity game engine that simplify the creation of that kind of games with isometric 3D graphics. Specifically these tools support construction, painting and decorating maps as well as the creation of entities and plot development through play of the game through interactive cinematic sequences. All this has been thought to its maximum extensibility to be extended by members of the community or future users of the project.

Keywords: Games, Game engines, Tools for game developing, Game Engine Plugins, Indie game development, Isometric games, Retro style games, Cross-platform development

Capítulo 1. Introducción

El videojuego o software de entretenimiento es hoy en día uno de los sectores más importantes y valorados del mundo de la Informática. A lo largo de las últimas décadas ha experimentado un crecimiento exponencial, habiendo alcanzado prácticamente todos los hogares de Europa y Norteamérica, y gran parte del resto de los hogares del mundo. En términos económicos, factura anualmente más que el sector del cine y la música juntos, adquiriendo una fuerte importancia en el mercado, abriendo multitud de puestos de trabajo y revitalizando la economía.

Hoy en día este apasionante mundo abarca cientos de formas y géneros para todos los gustos y personalidades. Desde el clásico género Arcade, con sobresalientes títulos a lo largo de la historia como el comecocos (PacMan [1]) o los marcianitos (Space Invaders [2]) o recientes y famosos títulos como Angry Birds [3], pasando por los géneros de aventura con títulos como The Legend Of Zelda [4] que sigue progresando hoy en día y renovándose con nuevas mecánicas para explorar el género, hasta los géneros disparos en primera persona con títulos que incluyen exitosas sagas como Half Life [5] y Call Of Duty [6].

La tendencia durante las últimas décadas ha sido siempre valorar mejor los videojuegos que mejor aspecto gráfico lograban, y que explotaban al máximo las capacidades de los chips según se iban lanzando. Esto causaba, a los pequeños y nuevos estudios, grandes dificultades para adaptarse y generar los resultados que el público esperaba.

Los grandes estudios, por su parte, a la par que crecían, desarrollaron y ampliaron sus herramientas, siendo estas cada vez más potentes y capaces de generar videojuegos con un mejor acabado en menores tiempos y costes. De las pocas herramientas que llegaban a comercializarse, muchas menos se encontraban al alcance de la economía de los pequeños y recién nacidos estudios y desarrolladores independientes.

Con la comercialización de los motores de videojuegos, las compañías de desarrollo de videojuegos han sufrido una evolución en la que, estas, han comenzado a dedicar equipos enteros para desarrollar y mantener motores gráficos, o incluso, dejar el desarrollo de videojuegos, para centrarse únicamente en el desarrollo de su motor.

Esta evolución generó un nuevo producto que se comercializaba en el mercado, y por consiguiente, todas aquellas compañías que comercializan un motor, tendrían que participar en una batalla comercial, en la que, tanto la gran demanda de motores gráficos, como la gran cantidad de los nuevos motores gráficos que surgían, produjo que poco a poco el precio de las licencias de dichos motores fueran descendiendo hasta tener un precio bajo comparado con los precios iniciales.

Esto, junto a las nuevas alternativas gratuitas, e incluso libres y de código abierto, hicieron que un sector se viera beneficiado por esta fuerte competencia: El sector de los

desarrolladores independientes y de los pequeños equipos de desarrollo de videojuegos. Éstos, gracias a la bajada de los precios y a la llegada de motores gráficos gratuitos, pudieron acceder a plataformas de desarrollo, elevadamente costosas de desarrollar desde cero para un desarrollador independiente o un pequeño equipo, sin disponer de los recursos que antes eran necesarios.

Debido a este cambio en el que, ahora, el mercado ofrece infinidad de herramientas que permiten alcanzar excelentes resultados gráficos con pocos costes, y disminuyendo drásticamente los tiempos de desarrollo, nuevos estudios y desarrolladores han conseguido publicar videojuegos de éxito teniendo unos recursos muy escasos, especialmente económicos y de mano de obra.

Gracias a este cambio, han surgido títulos en estas últimas décadas que han tenido un abrumador éxito, siendo números uno de ventas y ganando infinidad de premios. Entre estos títulos encontramos videojuegos como: Terraria [7] de Re-Logic, o Magicka [8] de Paradox Interactive, desarrollados con el motor gráfico gratuito XNA de Microsoft [9]; AntiChamber [10] de Alexander Bruce, o Sanctum [11] de Coffee Stain Studios, desarrollados utilizando el motor de licencia muy barata Unreal Engine 3 [12]; Rust[13] de Facepunch Studios LTD, o Slender: The Arrival [14] de Blue Isle Studios, desarrollados utilizando Unity [15], motor de licencia de precio muy bajo; e infinidad de títulos adicionales [16] que, pese a no haber sido enumerados, también han tenido un éxito muy grande y han recaudado grandes cantidades de dinero.

Con la reciente llegada del Smartphone y las tabletas inteligentes, el mundo del videojuego ha sufrido además un impulso adicional. Estas nuevas plataformas no sólo permiten la creación de nuevos géneros de videojuegos, sino que además, permiten la recuperación de antiguos géneros de videojuegos que requieren menor capacidad computacional, y menor cantidad de recursos para los desarrolladores. Estos antiguos géneros aportan una experiencia diferente a las anteriores, y por otra parte, más adecuada a estas, pues los videojuegos más antiguos estaban caracterizados por estar centrados en la habilidad del jugador para superar los diferentes niveles y por los tiempos de juego cortos, que se adaptan al tiempo de uso de este tipo de dispositivos.

La mentalidad de los jugadores también ha cambiado. Antiguamente, los jugadores buscaban en los juegos grandes retos que pudieran llevarle difíciles y repetitivas tareas hasta que pudieran lograr superarlas. El mercado de los videojuegos era mucho más cerrado y los juegos que más se tardaban en completar acababan por tanto siendo los mejor valorados. Actualmente, los jugadores disponen de un amplio mercado de videojuegos de todos los tipos, géneros y estilos y pese a seguirse repitiendo el concepto de reto y repetición, esto no significa que dichos juegos tengan más probabilidades de atraer jugadores. La simplicidad en conjunto con el reto forman ahora la clave para ser atractivo.

El estilo de videojuegos retro pretende recoger todas aquellas características de los primeros juegos con limitadas capacidades gráficas y utilizarlas con las herramientas actuales para la creación de juegos innovadores. Tal es el caso de juegos como FEZ [17] de Polytron Corporation, que pese a contar con un estilo retro, ha sido uno de los juegos más exitosos del último año por el atractivo modo de juego que planteaba. Otro juego más reciente es Starbound [18] de Chucklefish, el cual extrayendo el concepto de Terraria [7] de un mundo abierto, expande el universo para ofrecer una experiencia prácticamente limitada en un entorno retro 2D. Esta sección de juegos retro no podría terminar sin mencionar Minecraft [19] de Mojang, el cual combinando gráficos 3D mínimos con texturas pixeladas consigue crear la perfecta armonía del estilo retro en la actualidad.

Dentro del mundo de los teléfonos y tabletas inteligentes, el estilo retro se ha hecho un hueco especial, albergando sagas completas con este estilo como Zenonia [20] de GAMEVIL, o las series de juegos-simulador de Kairosoft destacando su título más importante Game Dev Story [21], cuya representación se realiza en un estilo retro isométrico.

Y es este estilo el que se busca en este proyecto, pues plantea la recuperación de títulos como el clásico español La Abadía del Crimen [22].

Actualmente, dentro de los juegos en perspectiva isométrica podemos encontrar pequeños títulos como Towns [23] desarrollado por el español Xavi Canal o Monument Valley [24] de ustwo. Sin embargo, muy pocos mezclan el estilo retro con la perspectiva isométrica. Por ello, ya que es un estilo que ha sido muy recurrido a lo largo de la historia de los videojuegos, a causa del motor gráfico Filmnation [25] para Spectrum, pero que no ha sido continuado, en este proyecto se pretende dar una nueva vida al estilo.

Obteniendo referencias del exitoso motor editor de juegos de rol RPG Maker [26], cuyas claves del éxito radican en su simplicidad, y transformando el estilo al de perspectiva isométrica, se realizará un editor capaz de crear juegos de forma versátil, cómoda, simple y fácilmente extensible, capaz de llevar a todas las plataformas disponibles los juegos generados y cuyos uso no requiera elevados conocimientos de programación para que nuevos y novatos desarrolladores puedan comenzar a experimentar y desarrollar sus propios y novedosos títulos, que renueven tanto el género de aventuras como el estilo isométrico.

No será necesaria la realización de una herramienta independiente, sino que, aprovechando la versatilidad de los motores vigentes que proveen de desarrollo multiplataforma de forma nativa, podremos realizar herramientas o plugins que los extiendan y provean de las mecánicas necesarias para poder realizar el proyecto.

En el estado del arte en el capítulo número 1, estudiaremos las diferentes opciones que nos ofrecen los motores para realización del proyecto, así como los diferentes videojuegos en los que se referencia, y define sus estilos, este proyecto.

Capítulo 2. Revisión del Estado del Arte

Con respecto a lo que rodea a nuestro proyecto, y las necesidades que nos han llevado a la creación del mismo, establecemos que a día de hoy, las herramientas (tanto complementos como herramientas que se soportan por si mismas) que podemos utilizar para la creación de videojuegos isométricos del estilo que nosotros planteamos, son realmente escasas, con funcionalidad pobre, o nulas.

Nuestro estado del arte, entonces, podemos dividirlo en tres categorías, las cuales serían:

1. Videojuegos de referencia.
2. Motores de videojuegos.
3. Complementos de motores de videojuegos.

Establecemos estas tres categorías dado que este fue el orden que seguimos en la investigación cuando nos planteamos nuestro proyecto. Al comenzar, buscamos aplicaciones completas, capaces de generar un videojuego completo del estilo que buscábamos, y al fracasar, decidimos buscar complementos para las aplicaciones que habíamos encontrado. De todas estas aplicaciones que encontramos, Unity fui la más extensa, y dedicaremos una categoría a hablar de los *plugins* de dicho motor, que se utiliza en nuestro proyecto.

Sección 2.1. Videojuegos de referencia

El punto de partida de este proyecto no fueron las herramientas que íbamos a crear, ni el motor que íbamos a utilizar, sino el resultado final que queríamos obtener en los videojuegos creados con este proyecto.

Dado que todos los integrantes tenemos una gran experiencia en el mundo de los videojuegos, las ideas y referencias comenzaron a surgir con rapidez, delimitando qué queríamos de cada una de ellas, puntos a favor, puntos en contra, y cuales sencillamente debíamos desechar.

Por consiguiente, en las siguientes subsecciones se referencian todos aquellos videojuegos que nos han servido como referente para la creación de nuestro sistema.

Sección 2.1.1. La Abadía del Crimen

Este videojuego fue el punto de partida de nuestro trabajo, y por ello, se merece el ser mencionado en primer lugar de nuestra lista de referencias.

La Abadía del Crimen [22] [27] es un videojuego de la compañía Opera Soft, que fue lanzado al mercado en 1987 y que es considerado uno de los juegos más icónicos y mejor valorados de la Edad de Oro del Software Español. El videojuego sigue la trama de la novela En el nombre de la rosa [26] [27] de Umberto Eco.

En la novela se narra la historia de Fray Guillermo, un fraile que realiza una investigación en una abadía de los Apeninos Ligures acerca de unos misteriosos crímenes que suceden allí, y su pupilo Adso, el cual le ayuda a lo largo de su tarea.

En el videojuego, los nombres se han conservado, aunque modificado un poco (Fray Guillermo de Occam en el videojuego, Fray Guillermo de Baskerville en la novela). Por su parte, la organización de la trama es similar, desarrollando esta en siete días.

Por otra parte, y hablando más de los aspectos de género y jugabilidad, nos encontramos con una videoaventura que nos permite interactuar con el entorno moviéndonos por el mismo y recogiendo objetos, lo que desarrolla unos sucesos u otros en función de donde y cuando nos encontremos y que objetos hayamos obtenido.
Además con respecto a los gráficos, encontramos que están en perspectiva isométrica, con escenarios divididos en casillas (aunque estas casillas se comportan de manera muy diferente a las de este proyecto, lo cual se discutirá más adelante)

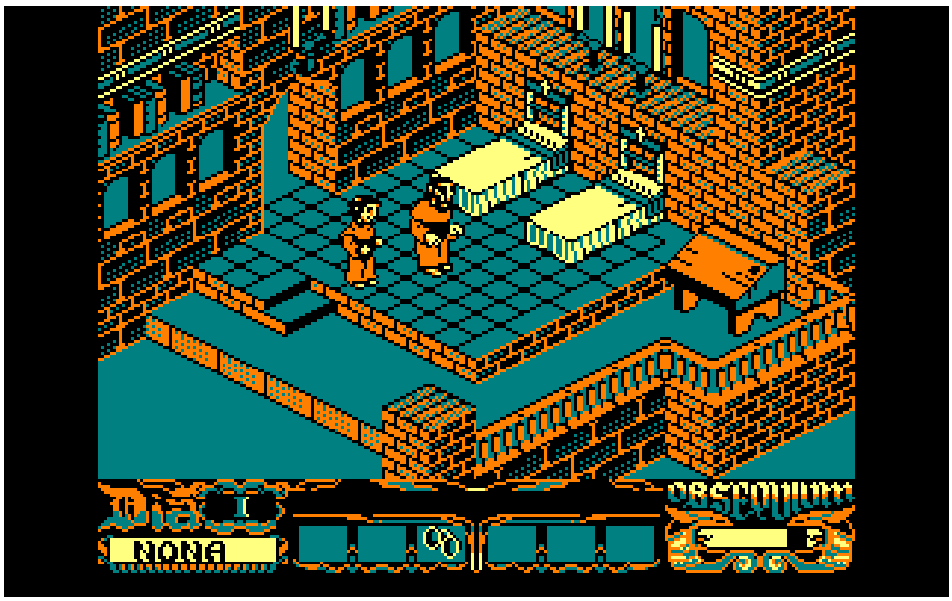

Figura 2.1 - Imagen tomada del videojuego La Abadía del Crimen

En la Figura 2.1, vemos a Guillermo (el hombre más alto y adulto) y a su pupilo en su dormitorio de la Abadía. Se aprecia perfectamente la perspectiva en isométrica, y, las

casillas de suelo en forma cuadrada que delimitan la zona por donde se mueven los personajes.

Por su parte, y al ver la imagen, encontramos uno de los aspectos que queríamos obtener en nuestro resultado final: el estilo "retro". Concretamente el hecho de que en la representación final del videojuego, se pudieran apreciar los gráficos "pixelados", consiguiendo un ambiente y unas sensaciones similares a las que produce jugar videojuegos clásicos.

En el videojuego, los diferentes personajes y sucesos se comunican con nosotros mediante un pequeño cuadro de texto que se encuentra en la parte inferior de la pantalla. Esta característica, también la queríamos incorporar a nuestro resultado final, pero con una serie de mejoras, como el poder decidir cuándo hemos terminado de leer el texto para que se nos muestre un párrafo nuevo o terminar, o la facultad de poder tomar decisiones y responder cuando los diferentes personajes se comuniquen con nosotros. Esta capacidad de diálogo que falta en la Abadía del crimen, nosotros la consideramos vital en una videoaventura, ya que, nos abre un mundo de posibilidades como desarrolladores para esconder nuevos hitos o logros en el juego al responder de una manera determinada a los personajes. Le da al jugador una mayor inmersión, dado que ahora puede tomar parte en la historia, y mejora la calidad de la experiencia abriendo un mundo de posibilidades para el jugador.

Por último, y aunque de manera poco desarrollada, encontramos una de las características más importantes que deseábamos incorporar a este proyecto: las secuencias. Estas son fragmentos de videojuego que realizan una pequeña interpretación de un suceso como si de una pequeña película se tratase. Estas secuencias, en La Abadía del Crimen, se producen sólo cuando nos hayamos en un lugar determinado en un momento concreto. En ellas observamos cómo suceden diferentes acciones, entre las cuales están: el movimiento de personajes de un lugar a otro, el cambio de escenario (mostrando la cocina, o los dormitorios) y la impresión de texto en la parte inferior de la interfaz.

Como dijimos antes, en dichas secuencias de La Abadía del Crimen no hay interacción por parte del jugador. Este simplemente se limita a observar lo que ocurre, sin tener oportunidad de tomar parte y, por ejemplo, responder cuando le preguntan. Por otra parte las acciones que se han enumerado anteriormente nos parecían demasiado limitadas, por lo que estas pequeñas y simples secuencias fueron sólo un punto de partida sobre el cual realizamos una investigación para obtener un conjunto de secuencias más rico y ajustado a nuestros criterios.

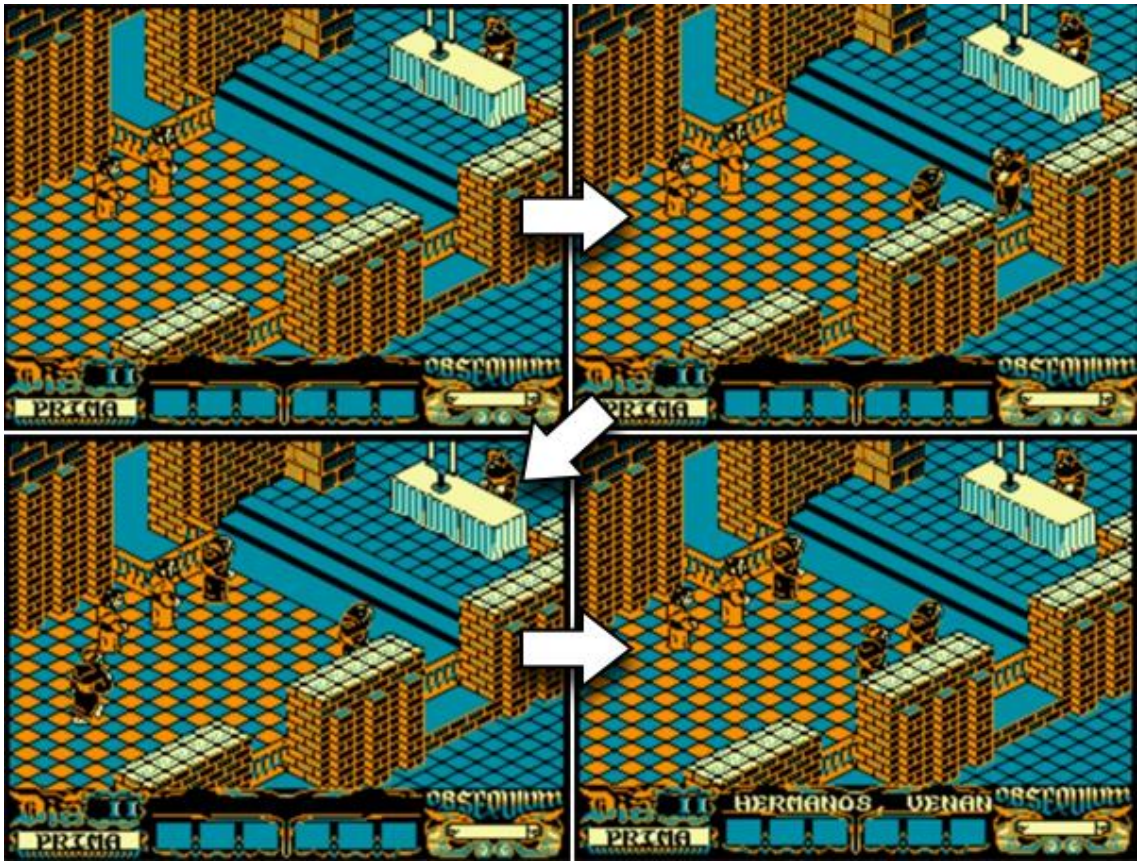

Figura 2.2 - Cuatro fragmentos de La Abadía del Crimen que forman una pequeña secuencia

En la figura 2.2 vemos como los frailes van llegando poco a poco a la reunión y, cuando todos se encuentran en sus posiciones, en la ranura de abajo, se empieza a mostrar el texto "Hermanos, Venancio ha sido asesinado" en una marquesina y de manera gradual, carácter a carácter. La secuencia se completa cuando el texto termina de mostrarse y cambia el escenario, colocando al protagonista y a su ayudante de nuevo en sus aposentos.

Sección 2.1.2. Final Fantasy Tactics

La saga de videojuegos Final Fantasy Tactics , compuesta por Final Fantasy Tactics[28], Final Fantasy Tactics Advance [29] y Final Fantasy Tactics A2 [30], nos ha servido en gran medida para obtener referencias acerca de lo que queríamos obtener del resultado final del motor, sobre todo, en la parte de desarrollo de mapas y entornos, dado que, como se mencionó anteriormente, hay un problema con las casillas de los mapas de la Abadía, y que, en el Final Fantasy Tactics, especialmente en su versión A2, se muestran multitud de detalles en los mapas.

La saga Final Fantasy Tactics, cuyo primer título fue desarrollado por SquareSoft en 1997 para la plataforma PlayStation, y el resto de sus títulos fueron desarrollados por Square-Enix, siendo FFT Advance en el año 2003 y para plataforma Nintendo Game Boy Advance, y FFTA2 en el año 2008 y para la plataforma Nintendo DS; son unos videojuegos del género RPG, que, aunque no forman parte del género de las

videoaventuras, sí que tiene muchas de las características que queríamos incorporar a nuestro resultado final.

La dinámica del videojuego es un poco más compleja de lo habitual, llegando a ser poco intuitiva en algunos momentos. La trama se desarrolla mediante diferentes misiones, las cuales podemos obtener de los taberneros de las ciudades.
Al hablar con ellos se nos ofrece una lista de misiones disponibles, las cuales van desde visitar un territorio hasta entablar una guerra con un clan enemigo que está causando estragos a los habitantes de dicho mundo, y que por ello, requieren de tu ayuda por medio de dicha misión.

En el videojuego nos encontramos con un mundo que vamos desbloqueando según avanzamos, compuesto por diferentes territorios donde, a su vez, encontramos una serie de zonas a las que podemos acceder. Estas zonas son las que nos hemos basado en mayor medida para definir el estilo de los mapas que deseábamos poder crear. En ellos (véase figura 2.3) encontramos la mayor parte de las características que al final decidimos incorporar.

Figura 2.3 - Mapa del videojuego Final Fantasy Tactics A2 remarcando las: Decoraciones, Rampas, Casillas y Alturas

Entre los elementos que destacamos en la anterior imagen se encuentran:

- **Celdas**: Forman la unidad mínima con la que se constituye el mapa. Sobre ellas se mueven los personajes, ocupando siempre una de ellas. En el videojuego Final Fantasy Tactics A2 en realidad existen personajes que pueden ocupar más de una

celda, sin embargo, decidimos no incorporar esta característica para simplificar nuestro sistema.

- **Alturas y medias alturas**: Las casillas a su vez están colocadas en el mapa a diferente altura del suelo. En nuestro análisis definimos que la altura por completo de lo que llamaremos una *celda* sería aquella que hiciera que esta tenga la forma de un cubo perfecto o varios cubos apilados, permitiendo también la medida de la "media altura", como un cubo cortado por la mitad.
- **Superficie Inclinada y Semi-Inclinada**: Existen un tipo especial de casilla que no se halla a un nivel determinado, sino entre dos de los mismos. Estas casillas que conectan un nivel con otro son las denominadas *superficies inclinadas y semi-inclinadas*, estas, a su vez tienen una orientación, un nivel de inicio, y un nivel de fin, lo que hace que conecten los niveles con diferente dirección, siendo las cuatro cardinales: norte, sur, este y oeste.
  A su vez, las rampas tienen diferentes alturas entre el nivel de comienzo y el nivel de fin, siendo similares a las anteriores alturas, es decir, una altura o media altura, creando así las rampas y las "medias rampas".
- **Decoraciones**: Por último destacamos un elemento del mapa llamado decoración, que aporta al mapa todos aquellos detalles y elementos que, sin ser interactivos, completan lo que un sería un primer pintado del mapa.
  A su vez, definimos que las decoraciones pueden ocupar una casilla, pero que esto no es necesario, dado que podemos encontrar decoraciones que se encuentren entre dos o más casilla, como la roca que se encuentra marcada en la Figura 2.3.

Todos estos detalles que se han explicado de manera muy breve en este capítulo, se detallarán con más extensión en capítulos posteriores.

Por último, y de una manera similar al apartado anterior, encontramos las secuencias de esta saga. Estas son bastante más complejas y elaboradas que las de La Abadía del Crimen, y, gran parte de sus características merecen la pena destacarse.

Si antes hablábamos de personajes que se movían por el mapa, y mensajes de texto que se mostraban en la parte inferior de la pantalla, ahora los personajes pueden también mostrar pequeñas animaciones con emociones, expresando visualmente si están alegres, tristes, o enfadados.
Los mensajes, por su parte, se muestran ahora en unos diálogos emergentes que incluyen una imagen a más alta resolución para representar al personaje que habla, junto con su nombre (cosa que no se podía diferenciar en La Abadía del Crimen, donde no se dice cuál es el emisor del mensaje).

Las capturas que se muestran a continuación están tomadas de la secuencia de introducción del videojuego Final Fantasy Tactics Advance. En dicha secuencia se ayuda al jugador a aprender los controles del videojuego.

Figura 2.4 - Dos ejemplos de diálogo en secuencia del Final Fantasy Tactics Advance tomados con varios segundos de diferencia.

Como vemos en las figuras 2.4, se observan dos bocadillos de texto en los que tenemos la imagen del personaje que habla con mucha mayor resolución, y pudiendo leer el mensaje con mayor claridad dado que dicho bocadillo es de mucho mayor tamaño que el pequeño recuadro donde se nos mostraba un texto que avanzaba independientemente de si nosotros éramos capaces de leerlo a tiempo o no.

Figura 2.5 - Dos ejemplos de expresiones en secuencia del Final Fantasy Tactics Advance tomados con una separación muy pequeña.

Por otra parte, en las figuras 2.5, vemos cómo el personaje mujer con el pelo de color rojo y largo, situado en el centro de las imágenes, inclina la cabeza expresando afirmación, y, a continuación, se nos muestra un cuadro de texto en el que se dice “Si, Sr. Lesli.”. Existen multitud de expresiones y posibilidades que se dan en el videojuego, como la negación, el susto, el miedo, etc. y consideramos que esta característica también debe de ser incorporada a nuestro proyecto.

Figura 2.6 - Ejemplo de elección en secuencia del Final Fantasy Tactics Advance

Por otra parte, en la figura 2.6, observamos cómo se le da al jugador la posibilidad de elegir y tomar decisiones dentro del videojuego. Esta capacidad abre bastante la libertad de tomar decisiones dentro de la linealidad del juego, sin embargo, únicamente podemos tomar decisiones sobre aspectos muy simples. De hecho en esta saga de videojuegos, las decisiones que tomemos, no variarán la trama de la historia, ni las reacciones de los personajes a posteriori, ni la obtención de diferentes objetos, únicamente nos permiten sentirnos un poco más dentro de la historia, creando una ilusión de libre albedrío.

En esta serie de secuencias fuimos capaces de abstraer distintas acciones, entre las cuales encontramos:

- **Mostrar diálogos**: Con un mayor nivel de detalle que en La Abadía del Crimen, pudiendo mostrar diálogos enfocados en un personaje, o enfocados en una roca encantada con la libertad de poder seleccionar la imagen y texto del cuadro de diálogo.
- **Realizar elecciones**: A un nivel muy básico y respondiendo "Si/No", le damos la oportunidad al jugador de elegir, lo que aumenta la inmersión dentro del videojuego.
- **Mostrar expresiones y animaciones**: Entre las cuales podemos hacer que un personaje mire a otro inclinando la cabeza, que diga "Si" o "No" gestualmente, o que se asuste en un momento determinado.
- **Ejecutar Comandos**: Comandos de cualquier tipo, como dar un objeto al jugador, mover un personaje de una casilla a otra, o cambiar el mapa actual por otro mapa distinto.

Sin embargo, las acciones de esta lista serían insuficientes, según el tipo de videoaventuras que queremos conseguir con nuestro proyecto y el tipo de experiencia de juego que buscamos (tan completa en elecciones como una aventura gráfica). Por lo tanto es necesario conocer mejor el estado del arte para así obtener información acerca de qué tipo de opciones se ofrecen actualmente en los videojuegos.

Sección 2.1.3. FEZ

Hubo un punto a lo largo de la definición del proyecto en el que surgió la pregunta: "¿2D o 3D?". Este juego fue el que nos dio la respuesta, aunque otros fueron, además de este, los que nos hicieron consolidar la decisión tomada.

FEZ [17] es un videojuego desarrollado por los desarrolladores independientes de videojuegos Polytron, lanzados al mercado en 2012. Forma parte del género que une dos mundos: los videojuegos de lógica y los de plataformas.

El protagonista del juego es Gómez, un pequeño personaje en dos dimensiones que vive en un mundo en tres dimensiones. Gómez, al principio del videojuego, únicamente es capaz de percibir dos de las tres dimensiones de su mundo. Llegado un momento, nuestro protagonista obtiene un pequeño *fez* (tocado masculino común en Turquía y el Norte de

África [31]) que le permite percibir aquella tercera dimensión que hasta el momento para él era inexistente.

Sin embargo, la capacidad que Gómez obtiene para percibir esta tercera dimensión es bastante limitada, y únicamente le permite rotar fragmentos de 90º sobre el eje vertical (véase figura 2.7), y, una vez la rotación se ha producido, Gómez vuelve a encontrarse nuevamente en un mundo en dos dimensiones, lo que abre un mundo de posibilidades y juegos de profundidad con la perspectiva.

Figura 2.7 - Secuencia de imágenes del videojuego Fez en el que se muestra la rotación de perspectiva que se permite hacer en dicho videojuego.

Gracias a este videojuego, y a su peculiar manera de utilizar las tres dimensiones, nos dimos cuenta de que no era necesario limitar nuestro proyecto a las dos dimensiones, sino que sería mucho más apropiado y mucho más potente que este proyecto utilizase las tres dimensiones, engañando a la perspectiva del jugador tal y como se hace en Fez.

Asimismo, el potencial del mundo en tres dimensiones, nos permitirá en un futuro, incorporar a nuestro resultado final, distintos elementos que únicamente podamos incorporar o nos faciliten los motores en tres dimensiones. Por ejemplo: con nuestro proyecto podrías incluir dentro de tu mapa un elemento en tres dimensiones y, gracias a su juego de cámara, y a su manejo de texturas, conseguir que este elemento parezca un elemento plano. Dentro de este caso encontramos: árboles, barriles, cofres, mesas, sillas, o elementos con mucho mayor grado de detalle como un reloj de cuco, o un complejo sistema de engranajes, los cuales, dibujarlos en dos dimensiones sería una gran carga de trabajo, y, a su vez, perderíamos capacidad de detalle.

Y, por otra parte, gracias a los sistemas de partículas y los efectos tridimensionales que pueden lograrse con los grandes motores de desarrollo de videojuegos (Unreal Engine [32], CryEngine [33], Unity [15], etc…) podríamos tener elementos como el agua realista

y con unas propiedades físicas muy similares a la realidad, o el fuego con un buen acabado, todo ello sin apenas esfuerzo para el desarrollador.

Como conclusión, de este videojuego obtuvimos la idea de que puedes tener un mundo, que aunque a simple vista parezca tener únicamente dos dimensiones, permite al desarrollador explotar el potencial gráfico de las tres dimensiones, conservando el estilo y la apariencia del primero.

Sección 2.1.4. Ragnarok Online

Como se ha comentado anteriormente, FEZ no ha sido el único videojuego que influyo al tomar la decisión de optar por un mundo en tres dimensiones, sino que fueron varios títulos, además de este, los que han conseguido consolidar esta decisión. Ragnarok Online es uno de ellos.

Ragnarok Online [34] es un videojuego de origen coreano, desarrollado por la compañía Gravity Corp., y publicado entre los años 2002-2006 (dependiendo del país donde se publicó). Este videojuego es un poco diferente a los anteriormente mencionados ya que pertenece al género de los videojuegos de rol multijugador masivo en línea, también llamados MMORPG [35] (del inglés *Massive Multiplayer Online Role Playing Game*).

En Ragnarok Online, como en muchos otros videojuegos de este género, encarnamos a un personaje personalizable en un mundo multijugador masivo con miles de usuarios jugando al mismo tiempo, en un mundo abierto lleno de misiones y objetivos que cumplir. Nuestro objetivo fundamental en este videojuego es alcanzar el nivel máximo, y con ello, desarrollar los atributos de nuestro personaje hasta su máximo. Tras ello podremos conseguir la mejor de las armaduras o derrotar al más fuerte de los enemigos en grandes grupos.

El videojuego está caracterizado por tener un mundo representado completamente en tres dimensiones, en el que los personajes (tanto avatares de jugadores, como aquellos controlados por la máquina) se representan gráficamente mediante imágenes bidimensionales (*sprites*).

Este contraste visual tan característico ha sido la marca de este videojuego y fue tan innovador y llamativo en su lanzamiento, que le propulsó directamente a la cumbre de este género, siendo muy popular entre los jugadores durante más de una década.

Figura 2.8 - Instantáneas del videojuego Ragnarok Online. En ellas se ve el estilo que une personajes en dos dimensiones, en un mundo completamente en tres dimensiones.

En la figura 2.8 se observa la característica que comentábamos anteriormente: un mundo en tres dimensiones con personajes en dos dimensiones. En este juego se nos permite girar la cámara (aunque sólo horizontalmente), característica que planteamos como posible futura ampliación.

Por último, y aunque no se observa en ningún lugar, el mapa de juego está dividido en casillas, lo que nos hizo remarcar aún más la idea de utilizar un mundo 3D.

Sección 2.1.5. The Secret of Monkey Island

Retomando lo comentado acerca de la importancia de la elección, existe un género de videojuego característico que da nombre a esa característica. Este es el género de las Aventuras Gráficas [36]. Pero como este género es muy amplio, se ha elegido The Secret of Monkey Island [37] como representante.

The Secret of Monkey Island es un videojuego desarrollado por LucasArts en 1990, perteneciente a la saga Monkey Island [38], y siendo el primer título de la misma. Este

videojuego es realmente diferente a los anteriores, ya que pertenece al género de las Aventuras Gráficas.

En este videojuego encarnamos a un joven llamado Guybrush Threepwood, el cual tiene como objetivo llegar a ser un pirata. Para ello, debemos de completar Las Tres Pruebas, las cuales deben realizar con éxito todos aquellos que deseen convertirse en un pirata. Sin embargo la trama del juego se ve afectada para embarcar al jugador en una gran historia.

El progreso en este género de videojuegos es diferente, ya que para poder avanzar en este videojuego, debemos de resolver multitud de acertijos que nos permitirán alcanzar nuevos hitos en la historia. Estos acertijos pueden ser cosas de cualquier tipo, como obtener información hablando con otro personaje, recoger y utilizar objetos con el entorno, o simplemente utilizar aquello que vemos.

La capacidad de interactuar, en Monkey Island, se abstrae en una serie de acciones que podemos realizar. Estas acciones van desde "abrir", "ir a", "coger" o "empujar" hasta "usar" o "hablar".
Esta serie de capacidades, tal y como se muestran en la aventura, resultan demasiado abiertas, permitiendo expresar acciones que no tienen sentido ni utilidad en la mayoría de los casos, como intentar "abrir una persona", o "hablar con un objeto", por lo que nosotros, pese a la libertad que da tener tantas opciones, decidimos limitarlo un poco, permitiendo ejecutar únicamente las acciones que tuvieran sentido con aquello que vamos a utilizar.
Una vez hubiéramos seleccionado aquellas capacidades que tienen sentido en dicho contexto, se mostrarán al usuario de la mejor manera posible, o incluso se ocultarán en el caso de que sólo podamos realizar una única acción en dicho contexto.

Figura 2.9 - Imagen del Monkey Island donde observamos en la parte inferior izquierda, las acciones que permiten al jugador interactuar con el entorno.

Por otra parte, Monkey Island (y aquellos juegos realizados con SCUMM[1] [39]), goza de un potente sistema de secuencias mucho más extenso que el que encontramos en Final Fantasy Tactics. A diferencia de Final Fantasy Tactics, aquí se le da mucha importancia a la elección y a la capacidad de elegir qué decir en cada momento (véase figura 2.9), por lo que, los diálogos incluidos en dichas secuencias, ahora, más que una secuencia lineal, se transforman en un nuevo acertijo en el que tendremos que elegir qué debemos decir en cada momento.

Gracias a este sistema de secuencias pueden suceder diferentes cosas en el videojuego, entre las cuales encontramos: El desbloqueo de punto importantes en la historia, la obtención de objetos que no pueden ser recogidos o tomados de otro modo, La obtención de cierta información que nos habilita más opciones en otros puntos del videojuego, etc.

Sin embargo, estas secuencias no tienen por qué surgir únicamente de un diálogo, sino que las puede provocar cualquier acción, como el caso de salir por la puerta de una tienda, habiendo tomado objetos de su interior; lo que provoca que el tendero de la tienda nos avise de que nos estamos marchando robando y salga a detenernos.

Asimismo, estas secuencias permiten mostrar animaciones, y no tienen por qué mostrar diálogos, siendo estas secuencias, en grandes rasgos, partes del videojuego en las que la capacidad de interacción normal se ve reducida a la elección de aquello que debemos decir en cada momento.

Figura 2.10 - Al hablar con el hombre de la izquierda nos hace una pregunta.

[1] Mezcla de lenguaje y motor de aventuras gráficas creado por LucasArts

En la secuencia que se inicia en la figura 2.10 se nos realiza una pregunta, sin embargo, esta pregunta puede extrañar al jugador, por lo que quizás quiera responder alguna otra cosa diferente de: Sí o No. Por ello, se nos facilitan muchas más opciones como se puede observar en la figura 2.11.

Figura 2.11 - Imagen donde se muestra, en la parte inferior, la lista de opciones entre la que podemos elegir lo que queremos decir.

Una vez elijamos la opción que deseamos decir, nuestro personaje la repetirá en voz alta, produciendo diferentes reacciones en la otra persona, lo que lleva al jugador a avanzar en el juego cuando elige la opción correcta (véase figura 2.12).

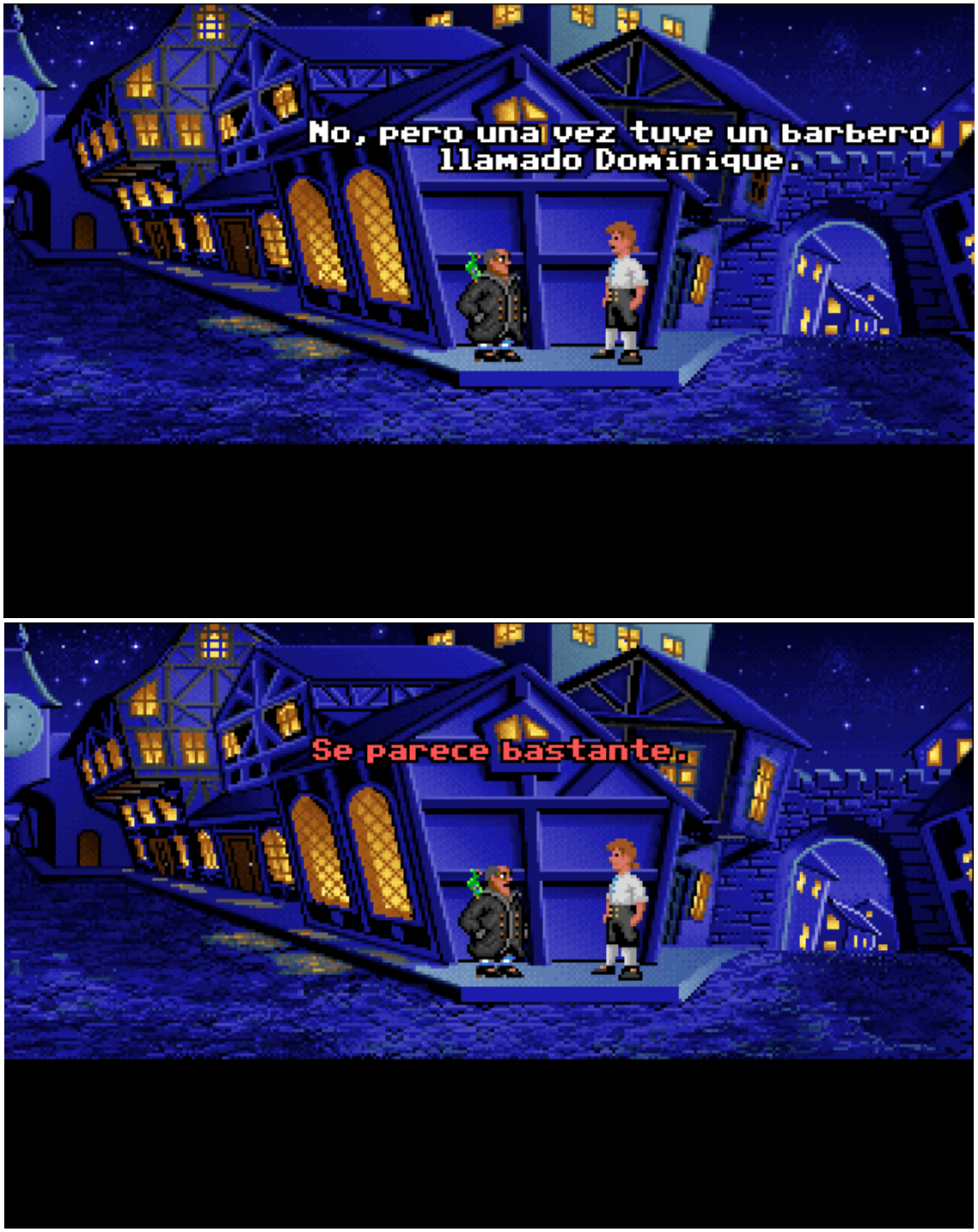

Figura 2.12 - El personaje dice en voz alta la opción elegida, a lo que el hombre responde "Se parece bastante."

Finalmente, merece la pena mencionar que estas opciones que surgieron en juegos de los años 80-90, se han ido adaptando a los videojuegos a lo largo de los años, viéndolas en juegos como: Star Wars: Caballeros de la Antigua República [40] (véase figura 2.13) o The Elder Scrolls: Skyrim [41] (véase figura 2.14).

Figura 2.13 - Instantánea del videojuego Stars Wars: Caballeros de la Antigua República donde se muestra en la zona inferior, la lista de opciones que podemos responder.

Figura 2.14 - Instantánea del videojuego The Elder Scrolls: Skyrim donde se observa en un cuadro flotante en la parte central-derecha de la pantalla, las opciones que podemos responder.

Sección 2.2. Motores de videojuegos

Tras un amplio análisis de los videojuegos en los que decidimos referenciarnos, se realizó otro análisis de las herramientas que existen para la creación de videojuegos, intentando que las mismas, completaran o abarcaran la mayor parte de requisitos que recopilamos.

Asimismo, además de los requisitos obtenidos de los distintos referentes, también decidimos incorporar nuevos requisitos procedentes de las herramientas, como, por ejemplo, que dicha herramienta permita la generación del videojuego en múltiples plataformas: PC, dispositivos móviles, videoconsolas, y más, o que minimice en mayor medida posible para el desarrollador la necesidad de conocer lenguajes de programación.

Las herramientas que más se utilizan para la creación de videojuegos son los llamados Motores de Videojuegos [42], y por ello, se ha realizado un análisis de muchos de los motores que existen actualmente. Se han encontrado motores que se acercaban mucho a los requisitos de este proyecto, pero que no los completaban. Sin embargo, todos ellos han servido en gran medida para encontrar una plataforma en la que construir esta herramienta si alguna de ellas lo permite.

Sección 2.2.1. RPG Maker

De entre todos los motores que se analizan en esta sección, RPG Maker [43] es el que ha servido en mayor medida para definir cómo íbamos a construir la herramienta, y para encontrar referencias en el diseño y construcción de las diferentes partes de nuestra herramienta.

RPG Maker son un conjunto de motores de origen japonés, creados por ASCII Corporation, parte de la empresa Enterbrain. Este conjunto de motores consta de diferentes versiones, siendo las más conocidas y exitosas: RPG Maker 2003 [44]. RPG Maker XP [45] y RPG Maker VX Ace [46], publicadas en diciembre de 2002, septiembre de 2005 y diciembre de 2011 respectivamente.

Los motores RPG Maker son conocidos por la sencillez con la que permiten la creación de un videojuego completo sin conocimientos de programación, arte, o música. Estos motores están enfocados a la creación de juegos del género RPG [47] (*Role Playing Game*)[2], aunque, no por ello sirven únicamente para la creación de los mismos.

En esta sección nos centraremos en la última versión del motor, siendo esta RPG Maker VX Ace, y refiriéndonos a ella como RPG Maker. Las herramientas, entre sí, contienen grandes diferencias, sin embargo, la última versión de la misma, contiene las herramientas de todas sus predecesoras, con ciertas mejoras y expansiones.

[2] Videojuegos inspirados en los juegos de rol tradicionales.

RPG Maker facilita al usuario un enorme conjunto de herramientas que permiten la creación y gestión de mapas, personajes, secuencias, objetos, armas, armaduras, enemigos, animaciones, etc. Y todo ello sin necesidad de conocer lenguajes de programación. Sin embargo, los conocimientos en lenguajes de programación permiten al usuario crear comportamientos más personalizados para su videojuego.

El editor de mapas de RPG Maker es la herramienta fundamental del motor y, por ello, es la primera herramienta que nos encontramos al ejecutar la aplicación. Este editor de mapas permite al usuario generar mapas divididos en casillas, pintar sobre los mismos utilizando unas paletas especiales llamadas *tilesets* (en los que profundizaremos más adelante), y añadir eventos[3] sobre el mismo.

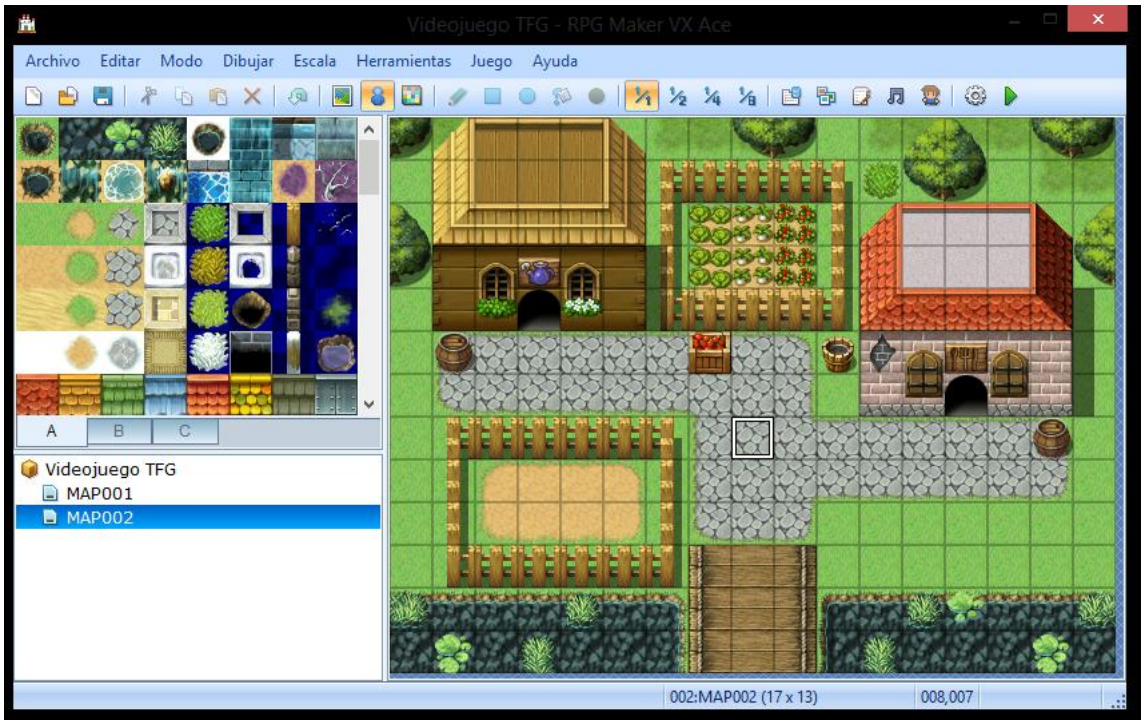

Figura 2.15 - Motor RPG Maker VX Ace mostrando un ejemplo de un mapa creado con el mismo en cuestión de minutos.

En la figura 2.15 se observa un mapa creado utilizando la herramienta. En la parte superior izquierda observamos la paleta de tilesets, entre la que podemos seleccionar uno de sus pequeños cuadrados y pintarlo sobre el mapa. Debajo de esta paleta se observan tres pestañas que nos permiten pintar contenido en tres capas diferentes.

[3] Estos eventos son diferentes a los referenciados en el resto de la memoria, ya que en RPG Maker los eventos pueden representar desde personajes con los que interactuar hasta iniciadores de secuencias, pasando por puertas que se abren y nos mueven a otro mapa, por ejemplo

Como hemos visto, los mapas creados con RPG Maker, a diferencia de nuestros requisitos, no están creados en perspectiva isométrica, por lo que, pese a ser muy buen referente, no se ajusta a nuestras necesidades. La perspectiva utilizada por RPG Maker es la perspectiva oblicua normal (véase figura 2.16), eliminando por completo la representación del eje restante, y dando así una sensación de tres dimensiones sobre dos dimensiones, y consiguiendo que aquellas zonas que representen techos, suelos o paredes, ocupen el mismo tamaño, siendo este el tamaño de una casilla.

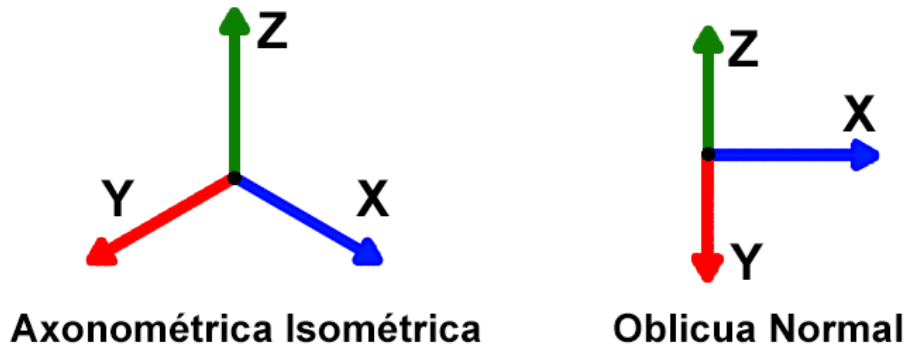

Figura 2.16 - Diferencia entre la perspectiva Isométrica y la Oblicua Normal.

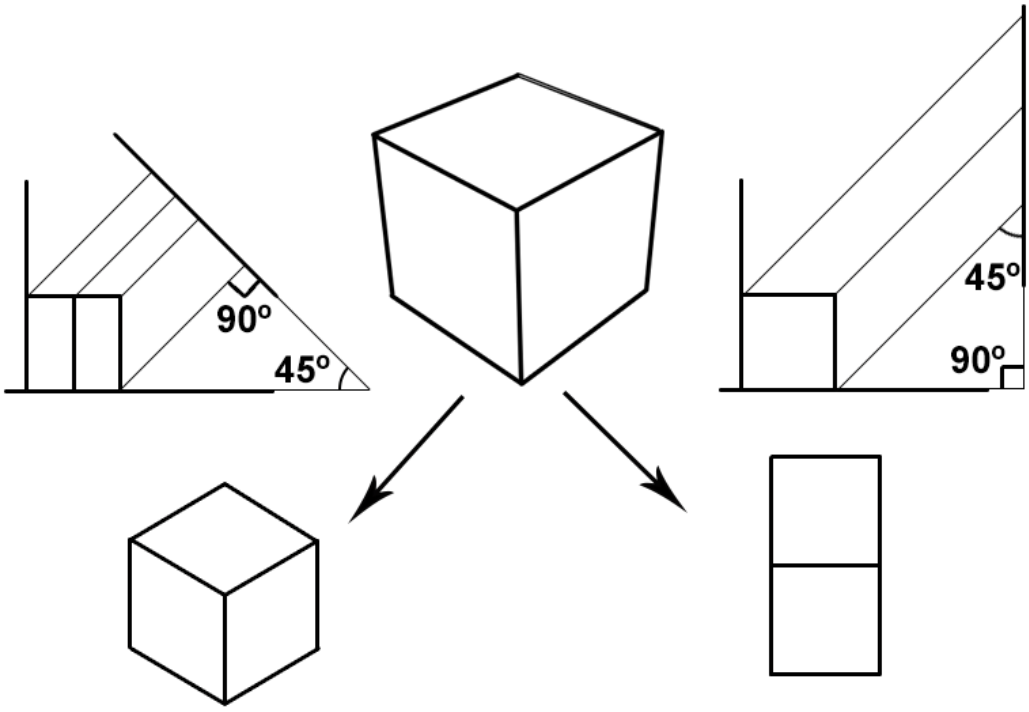

Figura 2.17 - Representación en perspectiva Isométrica (izquierda) y Oblicua Normal (derecha).

En la figura 2.17 se puede observar el funcionamiento de las transformaciones mencionadas.

Una vez se ha realizado el pintado del mapa, cambiaremos el modo utilizando los botones de la parte superior del editor (véase figura 2.18) y comenzaremos a colocar eventos sobre el mapa, ocupando estos una casilla y pudiendo tener una representación, o no.

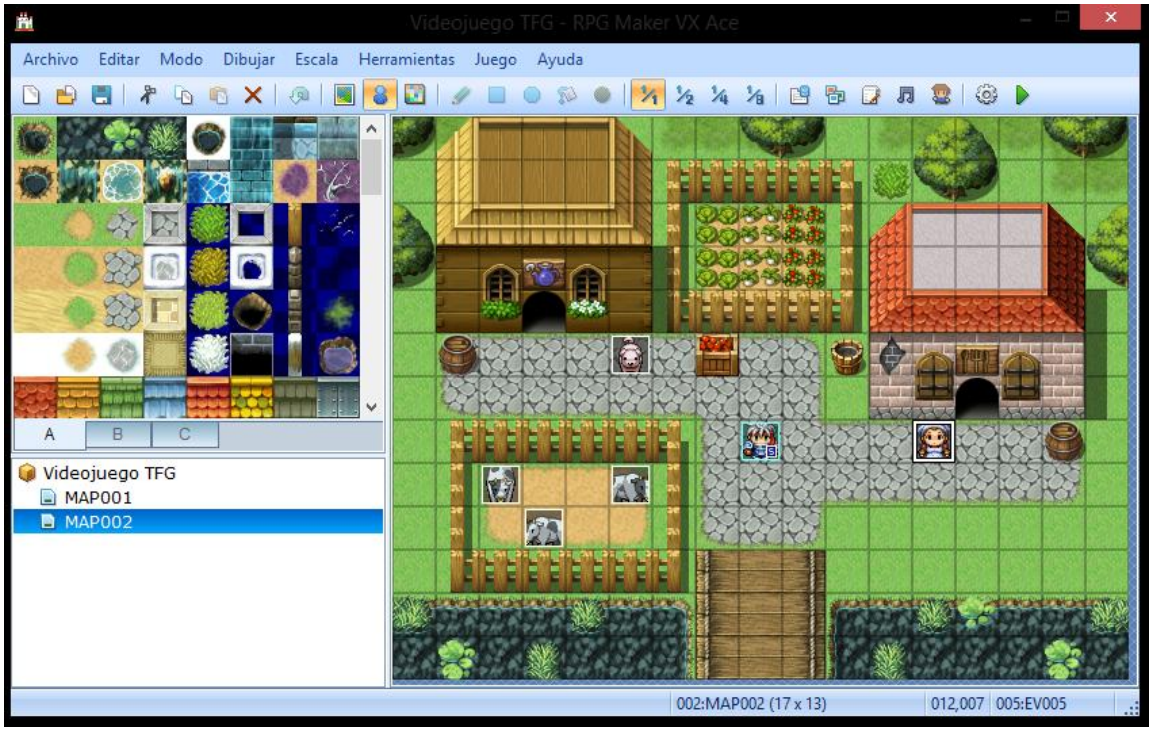

Figura 2.18 - Motor RPG Maker VX Ace mostrando un ejemplo de un mapa con eventos sobre el mismo.

Los eventos que se han colocado sobre el mapa pueden moverse, y además, se les puede configurar una inteligencia artificial para que se muevan o interaccionen con el protagonista en tiempo de ejecución. Todo esto, así como la secuencia que se ejecutará al interactuar con dicho evento, se configuran desde el panel de configuración de dicho evento.

Dichos eventos puede que no siempre se muestren, sino que puede que se encuentren inactivos y ocultos al jugador en distintas circunstancias, como haber alcanzado un punto en el juego, o haber adquirido un objeto a lo largo del mismo.

Asimismo, el editor de secuencias de RPG Maker es uno de sus puntos fuertes, donde podemos recrear pequeñas películas, en las que el jugador puede actuar, y que pueden tener una representación u otra dependiendo de las circunstancias en las que nos encontremos en el videojuego. Este editor es, básicamente, una lista de ejecución de comandos, que se sucederán al interactuar con el jugador.

Dichos comandos en RPG Maker son de muchos tipos diferentes, entre los cuales encontramos: Mostrar texto, Mostrar elecciones, Controlar interruptor, Cambiar oro, Cambiar objetos, Iniciar temporizador, Esperar, Mostrar animación, Transferir jugador a Mapa, Sacudir la Pantalla, Iniciar efecto de clima, Reproducir sonidos, etc… Siendo

tantos que prácticamente permiten hacer cualquier cosa al usuario. Todos y cada uno de estos comandos gozan de un editor personalizado para que añadirlos sea una tarea simple y, además, se minimizan así las posibilidades de producir errores.

Sin embargo, y pese a que estos comandos satisfacen prácticamente todas las necesidades, existe la posibilidad de no poder encontrar lo que buscamos, y por ello, en nuestro proyecto queremos dar la oportunidad al usuario de poder crear sus propios comandos e incluirlos al editor con sencillez.

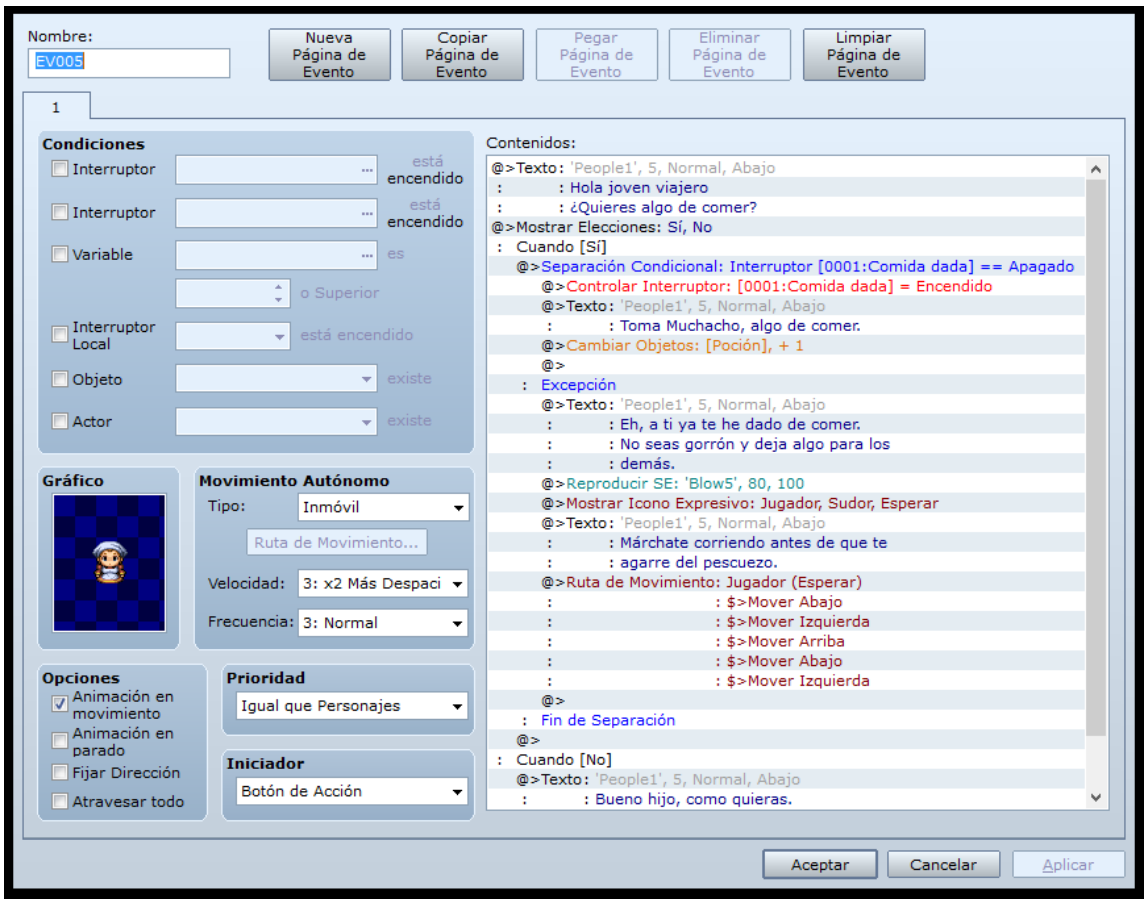

Figura 2.19 - Panel de edición de eventos.

En la figura 2.19 observamos: en la parte izquierda superior, las condiciones que hacen que dicho evento aparezca en el mapa, en la parte inferior, las características y comportamiento de dicho evento, y, a la derecha, la secuencia que se ejecutará al interactuar con dicho evento y, por consiguiente, iniciarlo.

Por último, RPG Maker tiene un gestor de bases de datos de contenido del videojuego, el cual comprende todo el contenido que los juegos del género RPG puede abarcar, sin embargo, el contenido de los videojuegos RPG, es más extenso de lo que se abarca en nuestro proyecto, por lo que, no se analizará ninguna de esas partes.

Asimismo, y gracias a las referencias obtenidas de RPG Maker, nuestro editor de mapas permitirá trabajar de una forma similar, pero adaptada a los mapas de nuestro género, y gozará de un editor de secuencias más abierto que el editor de RPG Maker.

Sección 2.2.2. Unreal Engine

Actualmente en el mercado, existen motores de videojuegos de un tamaño titánico, capaces de procesar y renderizar[4] elementos tridimensionales con un gran realismo (véase figura 2.20). Muchos de estos motores son privados, y pertenecen a empresas que los utilizan para la creación de juegos propios. Sin embargo, algunas compañías deciden comercializar licencias de sus motores, para que pequeñas empresas o desarrolladores, utilicen estas grandes plataformas, sin dedicar el gran esfuerzo que cuesta desarrollarlas.

Uno de estos grandes motores es Unreal Engine [32]. Este motor creado y desarrollado por la compañía Epic Games, tiene múltiples versiones, siendo la versión Unreal Engine 4, la más actual, y la cual fue publicada para aquellos que desearan adquirir licencia, en el año 2014.

Unreal Engine está caracterizado por tener una gran capacidad de renderización, así como un complejo sistema de materiales, iluminación y sombras, físicas, etc… Que consiguen que el resultado sea muy similar a la realidad. Asimismo, Unreal Engine permite crear videojuegos para múltiples plataformas, entre las cuales encontramos las plataformas móviles (Android [48] e iOS [49]).

Sin embargo, encontramos que modificar Unreal Engine 4, pese a que se han incorporado nuevos sistemas de gestión de *plugins*[5] [50] en esta última versión, es una tarea extremadamente laboriosa, sobre la que encontramos escasa documentación, y que todavía se encuentra en estado de desarrollo. Por lo que desarrollar un *plugin* para Unreal Engine fue descartado, y, por ello, generar nuestro proyecto sobre Unreal Engine sería inviable.

[4] Término usado en jerga de la Informática para referirse al proceso de generar una imagen o vídeo mediante el cálculo de iluminación partiendo de un modelo en 3D.
[5] Un plugin es un complemento para un software que permite añadir funcionalidad adicional al mismo.

Figura 2.20 - Imagen de la Elemental Demo [51] de Unreal Engine 4, demostrando las capacidades y el potencial del mismo. Esta imagen es propiedad de Epic Games.

Sección 2.2.3. Unity

De entre todas las alternativas que existen en el mundo de los motores de videojuegos, encontramos una opción, que desde que fue lanzada al mercado, ha ganado gran popularidad, sobre todo entre los desarrolladores independientes, gracias a las licencias gratuitas, a la facilidad para la multiplataforma, y a la facilidad para poder extender el mismo.

Unity [17] es un motor de videojuegos creado por la empresa Unity Technologies en 2005, que desde su lanzamiento ha ido evolucionando, para encontrarse actualmente en la versión 4.5. Unity está caracterizado por abarcar la gran mayoría de los dispositivos que existen actualmente en el mercado. Entre ellos encontramos: Windows, OS X, Linux, Xbox 360, PlayStation 3, Playstation Vita, Wii, Wii U, iPad, iPhone, Android, Windows Phone y, utilizando el *plugin* web, navegadores de internet. Asimismo, Unity está trabajando en compatibilidad para PlayStation 4 y Xbox One.

Unity presenta características similares a las de otros grandes motores de videojuegos, permitiendo la renderización de complejos elementos 3D, con un sistema de iluminación, materiales, físicas, etc… Para alcanzar el mayor realismo posible. Sin embargo, este no es su mayor punto fuerte, ya que el resto de sus rivales de mercado como, el anteriormente mencionado, Unreal Engine, son mucho más potentes y capaces de procesar con mayor nivel de detalle y acabado, aquellos elementos que se encuentran en su entorno.

Una de las grandes ventajas que Unity presenta frente a otros motores, es que tiene un mercado interno conocido como Asset Store [52] (véase figura 2.21), donde los

desarrolladores pueden publicar complementos para el motor. Pudiendo ser desde herramientas para facilitar tareas, hasta paquetes con recursos como modelos 3D, texturas, o bandas sonoras. En este proyecto, la parte más importante es la primera, ya que este proyecto es una herramienta para facilitar la creación de videoaventuras en perspectiva isométrica.

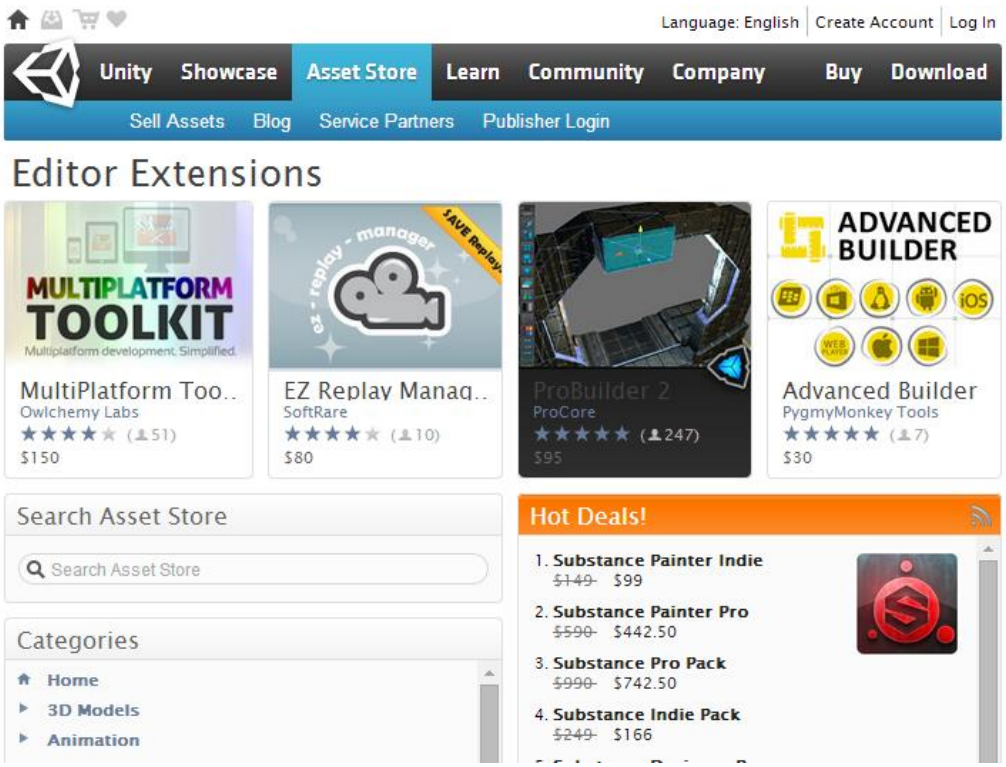

Figura 2.21 - Instantánea de la Asset Store de Unity donde los usuarios comparten sus complementos

Por otra parte, encontramos que Unity proporciona clases para realizar extensiones de su editor, pudiendo extender sin demasiado esfuerzo (una vez que se han adquirido los conocimientos), el mismo. Esto hace que podamos crear interfaces [6] para nuestras herramientas con facilidad, y esto hace de Unity una plataforma muy potente sobre la que basar nuestras herramientas.

Por todo ello, y por los elementos que Unity facilita a sus usuarios, se decidió utilizar este motor como plataforma sobre la que desarrollar nuestras herramientas. Realizar las herramientas desde cero, utilizando un motor gráfico como base, como OpenGL [53], tiene un coste demasiado elevado para realizarse en el tiempo que se proporciona para un Trabajo de Fin de Grado como este, por lo que, utilizar Unity como plataforma, reduce el tiempo de implementación y permite poder dedicar más tiempo a desarrollar las herramientas con la funcionalidad deseada.

[6] Una interfaz es el medio que utiliza un usuario para comunicarse con su equipo.

Sección 2.2.4. AlterEngine

Otro de los motores que se descartó, pero que también ha servido de referencia para la creación de mapas, es AlterEngine (véase figura 2.30). Sobre este motor hemos trabajado anteriormente y tenemos por tanto gran conocimiento y experiencia con él. Este motor fue cerrado y actualmente no cuenta con página web de referencia.

Nosotros, y nuestro grupo de desarrolladores AlterWorlds, desarrollamos AlterEngine como ampliación de Ramza Engine. Este desarrollo sucedió entre los años 2007-2008, y nunca llegó a publicarse una versión definitiva. El proyecto se alojaba en un repositorio abierto en SourceForge, por lo que el código fuente estaba abierto y disponible a quien lo deseara.

Este motor es capaz de generar entornos similares a los de RPG Maker, siendo estos en perspectiva oblicua normal, y divididos en capas. Como característica especial de este motor, encontramos que el género de videojuegos que nos permite crear es el de los MMORPG. Esto añade un nuevo nivel a la jugabilidad, el cual es, el modo de juego multijugador online.

El modo Multijugador Online ya ha sido referenciado a lo largo de este capítulo en varias ocasiones, y, pese a que en el diseño y especificación de este proyecto no se habla de modo multijugador online, no descartamos una posible ampliación, realizándose la misma tras la entrega del Proyecto de Fin de Grado.

Figura 2.30 - Instantánea tomada del motor AlterEngine en funcionamiento

Sección 2.3. Complementos de motores de videojuegos.

Dado que este proyecto es un complemento para un motor de videojuegos, además de los sectores analizados, es importante analizar algunos de los complementos que aportan una funcionalidad similar, o parcialmente similar a la nuestra. Sin embargo, son pocas las plataformas que dan soporte a complementos.

Por otra parte, los complementos, por lo general, no aportan una funcionalidad suficiente como para poder generar videojuegos utilizando únicamente los mismos, sino que intentan abarcar una funcionalidad concreta (como la creación de mapas, o, la representación de los mismos en una forma específica) y expandir la misma hasta su máximo exponente.

Por ello, se ha decidido dividir este subcapítulo en motores que dan soporte a complementos, en lugar de dividirse en complementos individualmente.

Sección 2.3.1. Complementos para RPG Maker

Los complementos para este motor, dado que este no admite complementos por sí mismos, se encuentran en forma de *Scripts*[7] que se ejecutan dentro del videojuego. Estos *Scripts* se ejecutan únicamente cuando el videojuego está en ejecución, por lo que no son concretamente una expansión del editor.

Existe un editor de mapas isométricos llamado Layy Meta Editor [54], desarrollado por GubiD[8], que permite crear mapas isométricos, utilizando una pequeña interfaz de usuario, y sin necesidad de tener conocimientos de programación. Cabe destacar que Layy Meta Editor es gratuito.

La creación de mapas con Layy Meta Editor es simple. Tras haber ejecutado el proyecto y haber inicializado los parámetros del mapa a crear (número de casillas a lo largo y ancho), se nos facilitará una superficie que podremos modificar. En dicha superficie podremos seleccionar casillas con el cursor y determinar la altura que tienen arrastrando con el botón derecho del ratón.

Tras esto, cambiaremos el modo del editor, seleccionando en la pestaña superior derecha la opción que deseemos, y podremos pintar sobre el mismo utilizando la paleta que se nos

[7] Programa usualmente simple, que por lo regular se almacena en un archivo de texto plano.
[8] GubiD es el nombre de un usuario de los foros de RPGMakerVXAce.net

facilita. Seleccionamos la imagen que deseamos pintar y determinamos con el cursor sobre las casillas que vamos a pintar.
Una vez finalizada la edición del mapa, se nos permite guardarlo para su posterior uso en un videojuego (véase figuras 2.22).

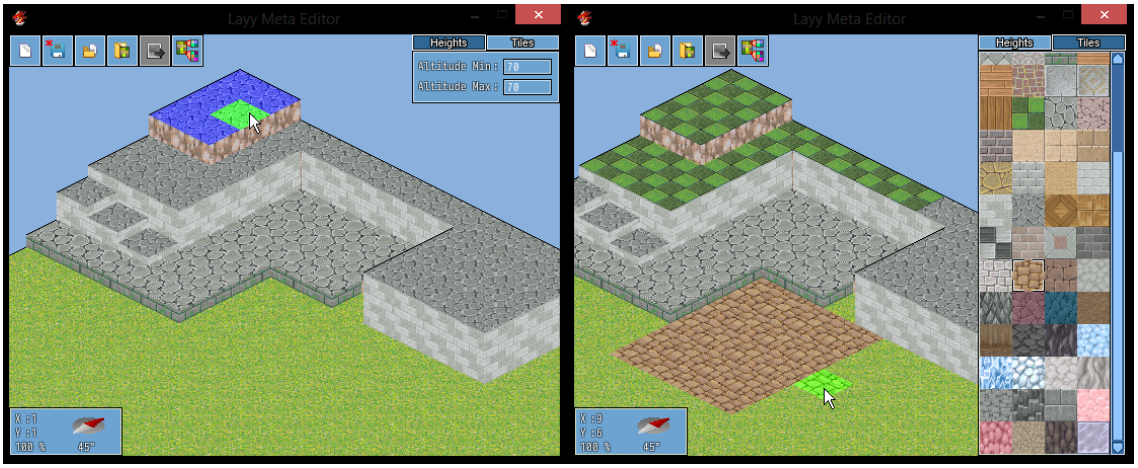

Figura 2.22 - Instantáneas tomadas del complemento para RPG Maker Layy Meta Editor.

Existen más complementos para este motor, sin embargo la mayor parte de estos, únicamente se dedican a la representación de mapas, y no a la creación de los mismos. Estos complementos utilizan los mapas creados con RPG Maker como base, y generan un mapa en perspectiva isométrica.

Sección 2.3.2. Complementos para Unity

Como ya se expuso anteriormente, Unity da multitud de facilidades a sus usuarios para la generación de complementos para su motor, por consiguiente, existen una gran cantidad de complementos desarrollados en esta plataforma.

En primer lugar, encontramos proTile Map Editor [55], un editor de mapas por casillas creado por proAndrius Productions (véase figura 2.23). Los mapas generados con este complemento, pese a compartir el hecho de estar formados por casillas, están destinados a conseguir otro resultado, ya que no se intenta mantener la perspectiva isométrica (salvo que se desee lo contrario) ni mantener un estilo retro[9], y además, se intenta explotar el potencial de Unity, utilizando modelos tridimensionales, y, por ejemplo, la capacidad de generar iluminación y sombras que provee el mismo.

Por su parte, la característica de incorporar elementos tridimensionales en un mapa, es un concepto que ya se presentó con anterioridad en el videojuego Ragnarok Online. La capacidad de añadir elementos tridimensionales a nuestro mapa, dado que nuestra

[9] Estilo caracterizado por la similitud con la apariencia de los primeros videojuegos. Presenta atributos como la presencia de grandes puntos cuadrados, característicos de los monitores antiguos.

plataforma de desarrollo será la misma que la de este complemento, consideramos que debe de ser un requisito funcional de nuestro proyecto.

Figura 2.23 - Imágenes del complemento proTile Map Editor. Estas imágenes son propiedad del proyecto proTile Map Editor.

Por otra parte, explorando la Asset Store de Unity, encontramos un desarrollador que mantiene la ideología de utilizar celdas para el posicionamiento de los elementos. Dicho desarrollador es ProCore, y ha publicado determinados proyectos que es importante mencionar.

El primero de estos es ProGrids [56]. Este complemento es bastante diferente a lo mostrado, ya que más que un editor, consiste en una herramienta que ayuda a alinear los elementos de un mapa. Dicho alineamiento se realiza utilizando como base una malla de líneas paralelas que se extienden en cada eje.

Una vez se han creado los objetos y posicionado sobre la malla, estos alinean su centro automáticamente a una posición válida para la malla, lo que hace que se mantengan proporciones y alineamientos.

Este concepto es muy relevante para nuestro proyecto, ya que las casillas que el mismo generará, han de mantenerse alineadas y parametrizadas en los ejes, para que así se mantengan las proporciones de los elementos de mapa, que se abstrajeron en el capítulo 2.1.2 (Siendo estos celdas, alturas, medias alturas, etc…). Sin el mismo, todos aquellos elementos que colocásemos sobre nuestro mapa se posicionarían sin orden.

Por último, merece la pena mencionar que, pese a que ProGrids es un proyecto que engloba el posicionamiento de cualquier cosa, la funcionalidad que se abstrae en este proyecto, es mucho más pequeña, siendo la misma la de posicionar únicamente casillas, colocándose correctamente dentro de un mapa.

Capítulo 3. Objetivos y especificación de requisitos

El objetivo principal a conseguir con el conjunto de herramientas es el de simplificar la creación de un juego utilizando el motor Unity. Dado que Unity ya de por sí plantea una plataforma bastante sencilla para la creación de un videojuego [57] nos planteamos en qué puntos, para un desarrollador, puede resultar más complicado realizar un juego de nuestras características.

El tipo de juego que se pretende poder realizar con las herramientas se caracterizará por tener mapas en perspectiva isométrica, en los cuales se podrán identificar casillas que estarán "habitadas" por entidades. Todo ello, tendrá la posibilidad de conservar un aspecto retro pixelado.

Además, los juegos creados con las herramientas, tendrán progresión y avances, permitiendo así al usuario generar una aventura con un contenido y un nivel de profundidad mayor al de un videojuego del estilo Arcade.

Sección 3.1. Objetivos

Cabría entonces destacar cinco objetivos principales:

- Creación de escenarios.
- Creación y gestión de entidades dentro del juego.
- Gestión de interfaces gráficas dentro del juego.
- Gestión de la interacción del jugador con el juego.
- Edición de secuencias.

En los próximos subcapítulos abordaremos la metodología o plan de trabajo que se ha seguido, el porqué de dichos objetivos y qué requisitos deberá realizar cada uno si queremos lograr simplificar el proceso de creación del juego.

Sección 3.2. Plan de trabajo

Para la realización del proyecto la organización y planificación es una de las partes fundamentales y primeras que cubrir. Dado que el proyecto pretende abarcar muchos aspectos muy complejos, y la experiencia sobre la plataforma Unity es prácticamente nula, se catalogará el proyecto como un proyecto de desarrollo de aplicación con destacables características de proyectos de investigación.

En este último tipo de proyectos predomina la necesidad de tener un modelo de desarrollo flexible, capaz de ser susceptible a muchos cambios provocados por los conocimientos adquiridos durante su implementación tanto como por los cambios provocados por los descubrimientos en fallos en la experiencia final. Por ello, pese a tratar de seguir un

modelo clásico en cascada [58] para la especificación, análisis e implementación, en la práctica, las dos primeras serán realizadas destacando el resultado final a conseguir, sin detallar en exceso la forma para realizarlo, ya que ésta se verá definida mediante pruebas, ensayos y prototipos según el método científico de los proyectos de investigación [59].

Sin embargo, en las partes con menor vinculación con la plataforma Unity se realizará un diseño más en profundidad, indicando posibles patrones y diagramas [60] para llevarlos a cabo. Estas partes serán los generadores de editores de forma dinámica, clases gestoras o intérpretes dinámicos.

Como recursos para la implementación se contará con los desarrolladores Iván José Pérez Colado y Víctor Manuel Pérez Colado, bajo la dirección de Federico Peinado Gil. Como recursos físicos, se requerirán ordenadores con Windows y con capacidades para ejecutar Unity. Como editor, se utilizará el editor proporcionado por Unity llamado MonoDevelop, con capacidades para edición y vinculación. Para el mantenimiento del proyecto se utilizará un sistema de control de versiones basado en Git utilizando para ello el servicio proporcionado por GitHub. Para el desarrollo de la memoria y planes de trabajo se utilizará el sistema Google Drive proporcionado por la Universidad Complutense de Madrid que permitirá el acceso y edición simultánea de archivos.

Como priorización de los objetivos, el primero y más complejo debido a la grandísima vinculación con la plataforma es el de la realización del editor de escenarios. Por ello, se estima una carga de trabajo muy superior a las demás, que podrá requerir, en conjunto con la especificación de requisitos la mitad del tiempo de desarrollo del proyecto.

Una vez se hayan sentado las bases, la lógica del juego será más sencilla ya que, pese a seguir bastante vinculada con la plataforma, se podrán construir sistemas sobre ella que logren los objetivos planteados. Por ello, el sistema del motor de juego, capaz de manejar entidades y desarrollarlas a lo largo del tiempo requerirá aproximadamente un tercio del tiempo restante.

Para implementar las posibles entidades, formas de control del juego y las interfaces que lo representen se estima un tiempo variable, en función de las características que se desearan implementar para esta primera versión del motor.

Por último, los sistemas editores de secuencias e intérpretes contarán con aproximadamente los últimos dos meses del desarrollo, dada la gran cantidad de posibilidades que deberán abarcar. En principio, se implementarán una serie de características básicas, pero se facilitará la extensibilidad.

El orden para analizar, diseñar e implementar los sistemas es fundamental ya que cada sistema se construye sobre lo anterior. El editor de mapas sienta las bases físicas para que las entidades empiecen a darle vida, los sistemas permitan las interacciones y representaciones y las secuencias permitan lograr el progreso en el juego.

Sección 3.3. Especificación de requisitos software

En los próximos subcapítulos se expondrán los requisitos requeridos para los distintos objetivos planteados.

Sección 3.3.1. Creación de escenarios

Dentro de un mapa isométrico, la unidad básica de terreno es la casilla. Dicha casilla deberá poder ser creada, editada y ocupada de una forma clara y sencilla.

Aunque uno de los objetivos es la creación de los mapas con aspecto retro, no se quieren cerrar las posibilidades a representar objetos y personajes tridimensionales. Por ello la casilla que se diseñe deberá tener una estructura tridimensional, y será la cámara la que le dé el aspecto bidimensional en perspectiva isométrica. El diseñador plasmará el resultado visual que quiere lograr y la casilla será capaz de mapear dicha textura para lograr este efecto 2D en un entorno 3D.

Unity facilita la creación de escenarios tridimensionales utilizando "Terrenos". Dichos terrenos forman una superficie moldeable orientada a crear escenarios realistas. El objetivo, sin embargo, es el de crear un escenario basado en casillas, no es importante que quede realista. Es más, cuanto más plano y cuadriculado sea el resultado, mejor. Por ello, los Terrenos incluidos por Unity no se ajustan a nuestras necesidades así que tendremos que desarrollar nuestro propio tipo de terrenos.

Dichos terrenos tendrán como requisitos ser capaces de añadir, quitar y editar las casillas. Cada casilla representará una coordenada dentro del terreno, por lo que no podrá haber dos casillas con la misma coordenada. Además, las casillas tendrán altura. Dicha altura modificará su aspecto visual, representando su tamaño desde el punto más bajo del mapa. La altura deberá ser un múltiplo de la anchura, con la excepción de la media altura.

Además, las casillas podrán tener diferentes tipos de superficies, a saber, planas, semi-inclinadas e inclinadas. Dichos tres tipos modificarán también su altura, pues una casilla inclinada tendrá una altura máxima de "altura + 1", una altura mínima de "altura" y una altura media de "altura + 0.5". Análogamente, la casilla semi-inclinada tendrá dichos valores, con el sumando dividido a la mitad.
Por último, dichas superficies podrán ser rotadas.

Una vez tenemos las casillas, la parte lógica la tendremos cubierta. Para cubrir la parte gráfica entonces, deberemos poder asignar texturas a dichas casillas, tanto a la parte superior, como a la lateral, para poder crear suelos y paredes. Estas texturas representarán el bloque mínimo, entendido como "baldosa" (*tile*) y deberán ocupar un cuadrado como el ancho de la casilla.
Si queremos asignar texturas a una casilla no habrá ningún problema, sin embargo, si queremos lograr ese aspecto retro pixelado, deberemos tratar las texturas de una forma diferente. Es ahí donde nace el concepto de "Textura Isométrica" o "IsoTextura". Esta

textura isométrica se caracterizará por pretender ser representada en el entorno tridimensional, de la misma forma que si fuera bidimensional, es decir, exactamente igual que la imagen de la textura original como se puede observar en la figura 3.1.

Por ello, será necesario realizar una herramienta capaz de mapear las texturas pixeladas, en forma de rombo por la vista isométrica, a una forma cuadrada, que finalmente será transformada a forma de rombo de nuevo por el mapeado de texturas[10] tridimensional.

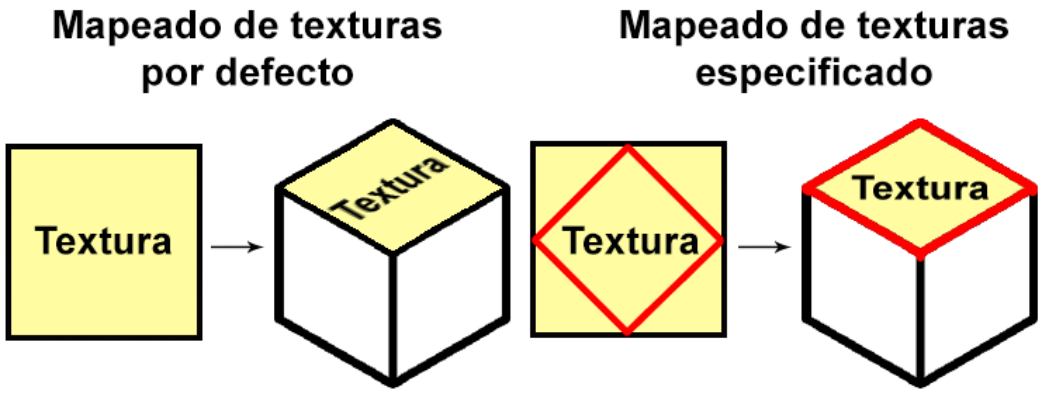

Figura 3.1 - Imagen que muestra las diferencias entre el mapeado de texturas por defecto y el mapeado de texturas que se especifica en este proyecto.

Una vez se han definido las casillas y sus texturas, el siguiente paso es darle detalles al mapa. Dichos detalles no interferirán en el curso del juego, pero permitirán personalizar los mapas. Algunos de los detalles podrían ser antorchas en las paredes, enredaderas, charcos de agua en el suelo, etc. Al no interferir en el curso del juego, no interactuarán con la lógica, pero estarán sujetas al paso del tiempo, por lo que podrán representar animaciones e incluso interactuar con algunas acciones del jugador.

Las decoraciones podrán ser de dos tipos, tridimensionales o bidimensionales. De ser tridimensionales serían tratadas todas por igual. Sin embargo, si fueran bidimensionales podrían ser colocadas de cuatro formas diferentes, paralelos al suelo, perpendiculares al suelo, paralelos al lateral o perpendiculares al lateral. Además, deberán ser mapeados de una forma análoga a como se hace en las texturas de la casilla.

El editor de escenarios entonces, permitirá colocar dichas decoraciones en las casillas, ya sea en la parte superior, como en los laterales. Cabe recordar que estas decoraciones son simplemente eso, decoraciones, y no tienen ninguna capacidad de interacción, ni de presencia en el mapa. Las decoraciones solo son objetos que el artista podrá utilizar para añadir detalles visuales a un mapa.

El resultado será un mapa físico que podrá ser utilizado de manera lógica. Dicha lógica deberá permitir el acceso libre a casillas mediante coordenadas y mediante vecindades.

[10] El mapeado de texturas establece cómo se sitúa la textura sobre el objeto al momento de proyectarse.

El mapa tendrá entonces el control de su representación, conociendo únicamente las casillas que lo forman y sus propiedades.

Los requisitos del editor de mapas pueden resumirse en la siguiente tabla:

Campo	**Requisitos**

Casilla	• Añadir
	• Quitar
	• Editar
	• Estructura en 3D
	• Modificar altura (Permitiendo medias alturas)
	• Modificar superficie
	  o Plana (inclinación de 0º)
	  o Semi-inclinada (inclinación de 22,5º)
	  o Inclinada (inclinación de 45º).
	• Colocación de texturas

Textura	• Mapeado básico de cualquier textura • Mapeado de texturas isométricas conservando el aspecto
Decoraciones	• Añadir • Quitar • Editar • Colocación de objetos 3D • Colocación de objetos 2D • Paralela a las diferentes superficies de la casilla. • Mapeado 2D a 3D conservando aspecto • Permitir animaciones

Tabla 3.1 - Requisitos del editor de mapas

Sección 3.3.2. Creación y gestión de entidades

Una entidad en el mundo de los videojuegos es la unidad "viva" mínima del juego. Dicha vida puede representarse de cualquier manera, movilidad, bloqueo, capacidad de hablar, etc. Esta vida consideramos que debe adquirirse de tres formas.

En primer lugar, se les dará la capacidad de representar animaciones. Las animaciones son aquellos progresos meramente visuales que representan un acto. Por ejemplo: moverse de una casilla a otra, estar en llamas, etc. Dichas animaciones interferirán en el curso de la lógica, pues mientras estén en proceso la entidad no podrá hacer otra cosa a menos que la cancele. Por tanto las entidades dispondrán de una forma de realizar animaciones con pleno control sobre ellas. De manera simple, un sistema de cambio de fotogramas puede ser suficiente.

En segundo lugar, se les dará la capacidad de pensar. Una entidad deberá tener al menos la oportunidad para decidir qué hacer, aun cuando no reciba ningún estímulo. Dicha capacidad puede aprovecharse para iniciar alguna animación, decidir si intercambiarse de casilla, etc. Este comportamiento personalizado dará cabida a la creación del jugador, teletransportadores y de entidades que ronden por una zona e incluso permitirá crear en requisitos futuros enemigos, entidades capaces de instanciar otras entidades, guardianes, etc.

En tercer y último lugar, se les dará la capacidad de reaccionar y comunicarse. Las entidades deberán ser capaces de recibir y enviar "mensajes". Dichos mensajes los entendemos como eventos, pues se producen eventualmente provocando un comportamiento en las entidades. De no suceder el evento, la entidad no realizará ese comportamiento a menos que decida realizarlo de forma autónoma. Gracias a esta comunicación podremos lograr reacciones mucho más precisas que si estuvieran programadas de forma autónoma, así como provocar el encendido y/o apagado de interruptores.

En este último campo, damos pie entonces a la necesidad de un sistema de envío y recepción de eventos. Dichos eventos deberán ser procesados tras cada actualización en el núcleo del juego. En cada fase de procesamiento de eventos sólo se procesarán los eventos producidos durante la anterior actualización con el objetivo de prevenir bloqueos.

Sin embargo, aunque pueda parecer que una entidad queda definida de una sola forma, puede ser interesante que una entidad esté formada por varios comportamientos o personalidades. Dichas comportamientos o personalidades añadirán o modificarán determinadas respuestas para las cuales están pensadas. Por ello, cada comportamiento por separado agrupa el conjunto de características para el cual está pensado. Estos comportamientos podrían almacenar la capacidad de moverse, que almacenaría la capacidad para responder a eventos de movimiento, la lógica para decidir si realizarlos o no y por último la mecánica para realizar dicho movimiento.

De esta forma las entidades podrían conformarse con muchas configuraciones de comportamientos distintas hasta lograr el conjunto de comportamientos deseado. Por ejemplo, una entidad "Javier el Mercader" podrá tener comportamientos para hablar, vender y además poseer un inventario. Dichos tres comportamientos definirán a "Javier" al completo. Este hecho, además de mejorar la flexibilidad a la hora de crear las entidades, permitirá reutilizar los comportamientos en distintas entidades.

Por otro lado los comportamientos podrían ser incluidos y retirados en tiempo de ejecución o activadas y desactivadas, permitiendo así crear sensaciones como envenenado, si realizáramos un comportamiento que vaya restando a la entidad salud y además mostrará visualmente un tono verdoso en el personaje, o paralizado, si se impidieran los eventos de movimiento (o se modificara la velocidad del comportamiento mover a 0).

Supongamos el siguiente caso de ejemplo para entender correctamente la diferenciación de casos:

La entidad habladora se caracteriza por tener en sus tres momentos de vida:

- **Animaciones**: Fundamentalmente para desplazarse de casillas. Dichas animaciones tendrán consecuencias lógicas. Por ejemplo, durante un desplazamiento, ambas casillas deberán contener a la entidad, aunque se esté reproduciendo la animación la casilla ya será accesible.
- **Comportamiento proactivo**: Una vez iniciada la conversación con él, será capaz de conocer su estado e irlo interpretando a lo largo del tiempo.
- **Reacción a eventos**: Cualquier otra entidad envía su petición de hablar con ella a través de un evento. En ese momento, la entidad programaría su diálogo y lo comenzará a reproducir.

Dentro de los requisitos que se quieren abarcar una vez implementada dicha plataforma, es el de proporcionar las personalidades básicas y sus editores. Los comportamientos que se implementarán serán:

- **Jugador**: que permitirá a una entidad poder ser controlada por la persona que juegue.
- **Mover**: que almacenará la capacidad para moverse y representar dicho movimiento.
- **Teletransportador**: que almacenará la capacidad de transportar a las entidades que se encuentren sobre ella a otras celdas o mapas.
- **Hablador**: que almacenará la capacidad para iniciar conversaciones o secuencias de acciones (como eventos, animaciones, etc.) y configurarlas.
- **Movedor aleatorio**: que almacenará la capacidad de generar cada cierto tiempo o bajo una probabilidad movimientos a celdas vecinas de forma aleatoria. Éste requerirá de la coexistencia con un mover para poder funcionar.

- **Inventario**[11]: cuya característica es la capacidad de almacenar objetos. Una entidad podría tener un inventario y entregar un objeto en un determinado momento al jugador. Un vendedor, de hecho, requerirá un inventario que se utilizará como fuente de objetos a vender. El inventario podrá tener cualquier cantidad de objetos, pero para su visualización, agrupará los objetos que sean iguales

Como idea o requisito futuro, se podrán implementar bajo el mismo sistema:

- **Salud**: que añadirá una cantidad de salud a las personalidades, permitiéndolas así que reciban daño o que se curen. Al llegar a 0 producirán un evento especial que, normalmente, será interpretado por el jugador y otras entidades como su muerte.
- **Vendedores**: cuya característica es la capacidad de mostrar una interfaz en la que se puedan comprar y/o vender los objetos del inventario.
- **Criaturas de ataque cuerpo a cuerpo**: Cuyo objetivo será el de perseguir al jugador una vez identificado y atacarle hasta matarlo.
- **Criaturas de ataque a distancia**: Similares a las criaturas cuerpo a cuerpo, pero lanzando proyectiles. Los proyectiles serán otro tipo de entidades/personalidades.

Esta forma de programar se conoce como programación orientada a componentes y suele ser mucho más flexible que una programación orientada a objetos clásica, por lo que ha tenido mucho éxito en el mundo de los videojuegos. En el motor Unity la programación orientada a componentes viene integrada de manera nativa.

Sección 3.3.3. Formas de interacción del jugador con el juego

La manera en la que los jugadores podrán interactuar con el juego es un aspecto clave para el éxito del juego. Siempre que dicha interacción sea simple y eficaz, el jugador no se agobiará con complicadas interfaces que le harán perderse en los primeros minutos de juego.

Para evitar que esto ocurra, se establecerá una forma sencilla de interacción con el juego. Dicha forma, deberá poder ser universal para todos los entornos que dispone Unity, entre los cuales se destacan los clásicos (ratón y teclado) y los táctiles.

Dentro del entorno de ratón y teclado, parece lógico proporcionar un mínimo entorno de controles con botones de movimiento con las cuatro direcciones y al menos un botón de acción, un botón de menú y un botón de salir. Dichos botones podrán ser extensibles fácilmente y permitirán al usuario final añadir botones adicionales. Para poder simular los botones en entornos táctiles, se proporcionará una interfaz en pantalla con los botones

[11] El inventario es un registro de bienes, en el que se almacenan los datos de un objeto y la cantidad de cada uno de ellos. Si uno de los objetos es sustraído, y se tiene una cantidad nula del mismo, no aparece en el inventario

que desee el usuario, en principio, la básica cruceta para las cuatro direcciones cardinales. Además si se utiliza el ratón se podrá interactuar de la misma forma que si utilizáramos el entorno táctil.

Por otro lado, el entorno táctil, además de la posibilidad de contar con botones en pantalla, contará con la funcionalidad básica de acciones. Dichas acciones se realizarán con la siguiente mecánica. Si el usuario pulsa una celda vacía, se avisará al personaje de que se ha realizado dicha pulsación. Por defecto el personaje interpretará esa pulsación como un intento de movimiento a dicha celda. Sin embargo, si la celda cuenta con alguna entidad sobre ella, se generará un menú con las posibles acciones que las entidades y el jugador proporcionen.

Para ejemplificarlo, si se pulsara sobre un hablador, dicha entidad proporcionará la acción de "hablar". Si se selecciona dicha acción, el jugador recibirá un evento de control con la entidad, la celda y la acción a realizar. De esta forma, cuando el jugador haya acabado el movimiento, enviará un evento "hablar" a dicha celda, provocando el que NPC inicie la conversación que tenga programada.

En el caso de existir una única opción como es el caso anterior, se realizará la acción por defecto, sin embargo, de existir más de una, se mostrará un menú contextual que permitirá seleccionar la acción mientras que el juego se quedará pausado. Dicha pausa podrá ser configurable en las configuraciones del juego.

Esta estructura abre la posibilidad de que si el programador decidiera ignorar los eventos en pantalla o deshabilitarlos podría hacerlo, generando otros tipos de juego.

En último lugar, la funcionalidad que proporcionarán los eventos de menú permitirá al jugador guardar y cargar partida, además de salir del juego y volver al menú principal. Por otra parte, dicho menú podrá ser ampliado por el programador utilizando las interfaces del apartado anterior. Sin embargo, las interfaces del apartado anterior no quedan restringidas a este aspecto. Podrían ser lanzadas desde eventos de entidades también, utilizando un sistema centralizado que se encargará de recoger los eventos del usuario y pasarlos a la interfaz.

Conviene señalar que la entidad jugador es una entidad también, y por tanto, también tendrá la opción de crear sus opciones, permitiendo, por ejemplo, abrir el inventario.

Con este sistema se permitirá al jugador interactuar con todo el juego, simplificando las tareas del diseñador del juego.

Sección 3.3.4. Menús e interfaces

Una parte fundamental de todo videojuego son los menús e interfaces gráficas que se utilizan para poder acceder a las distintas funcionalidades que se ofrecen. Estas interfaces están compuestas por algunos de los elementos que utilizan la mayor parte de las interfaces gráficas de usuario (botones, cuadros de texto, etiquetas de texto, etc.), y modificando los mismos para adaptarlos a las necesidades del proyecto.

Se facilitarán una serie de interfaces gráficas que permiten al usuario de los videojuegos creados con este proyecto, acceder a la funcionalidad del mismo, así como mostrar datos e información de lo que sucede. Dichas interfaces han de ser capaces de mostrar la información adecuada para su funcionalidad, y han de ser capaces de ser utilizadas desde plataformas táctiles.

Clasificando las diferentes funcionalidades de las interfaces, hemos establecido cuatro ámbitos para los componentes de las interfaces. Estos ámbitos son: la Interfaz de Conversaciones, el Menú Contextual de Acciones, la Interfaz de Inventario o Interfaz de Objetos, y los Controles en Pantalla

Sección 3.3.4.1. Interfaz de conversaciones

La Interfaz de Conversaciones se encarga de establecer una manera de representar los diálogos dentro del videojuego y, por otra parte, mostrar las opciones de respuesta entre las cuales el jugador puede elegir. Esta interfaz es altamente personalizable, ya que ha de mostrar diferente contenido dependiendo de una serie de parámetros variables.

Para la generación de la Interfaz de Conversaciones estableceremos cinco parámetros:

- **Emisor**: Establece quién está emitiendo el mensaje. Esto ayuda al diseñador a crear una representación más descriptiva gracias a poder mostrar detalles del emisor en la interfaz, o proyectar la propia Interfaz de Conversaciones en la posición del emisor. Gracias a este componente podremos: representar su nombre si lo deseamos, representar una imagen que identifique a dicho emisor, o posicionar la interfaz donde sea más apropiado.
- **Mensaje**: Es el texto que se ha de representar. Dicho texto ha de ser una cadena de caracteres alfanuméricos.
- **Caja**: Es la zona en la que se representará el mensaje. Tiene muchas opciones de representación, así como de animación del mensaje que se representa. Además, dentro de los requisitos opcionales, debe de ser capaz de fraccionar el mensaje en porciones para mostrar cada uno con una pausa y sin que el texto salga fuera del mismo, o sin dejar de mostrar alguna de las palabras del mensaje.
- **Bocadillo**: Es el recuadro que contendrá la información de todos los elementos anteriores. Se utiliza para englobar la Interfaz de Conversaciones al completo
- **Opciones**: Son las diferentes alternativas entre las que el jugador podrá seleccionar la respuesta que quiere emitir sobre la conversación. Han de ser

capaces de mostrarse en la pantalla ordenadamente y de comunicarse con el juego para indicar que el jugador ha elegido dicha opción.

Esta interfaz dispondrá de una serie de métodos que faciliten la funcionalidad de invocarse cuando se desee, y ha de ser capaz de identificar cuando el usuario ha finalizado la lectura del mensaje para continuar con la secuencia de mensajes y/u opciones

Sección 3.3.4.2. Interfaz de Inventario

La Interfaz de Inventario se encarga de mostrar todos los objetos que hay registrados en el Inventario. Además, ha de ser capaz de permitir una navegación entre los objetos, mostrar sus datos y permitir interactuar con ellos. Por último, esta interfaz ha de facilitar la opción de cerrarse cuando se desee, para poder ocultarla cuando deseemos.

Para la generación de la Interfaz de Inventario estableceremos los siguientes elementos:

- **Título**: Establece un nombre para el inventario. Este nombre no es necesario que pueda cambiarse en tiempo de ejecución[12].
- **Objeto**: Un objeto de interfaz muestra una representación de un objeto de inventario. Esta representación está formada por una imagen, un texto que define lo que son, y un valor numérico que representa la cantidad actual de cada uno de ellos. Además, este elemento permite al usuario comunicarse con el objeto representado, pudiendo acceder a su funcionalidad.
- **Lista de Objetos**: Recuadro cuyo contenido puede ser de mayor tamaño que el mismo. Ha de permitir una navegación entre todos los objetos que el jugador tiene, mostrando cada uno de ellos. Esta navegación ha de darse de forma vertical, y no horizontal, mostrando los objetos, cada uno en una posición inferior al anterior (siendo esta posición inferior debajo, y no siendo detrás, de tal manera que ambos elementos estén visibles).
- **Botón[13] de cierre**: Este botón permite acceder a la funcionalidad de cerrar la interfaz. Una vez pulsado, esta interfaz desaparecerá hasta que vuelva a ser mostrada.

Similarmente a la interfaz de conversaciones, esta interfaz dispondrá de un método que facilite la funcionalidad de invocarse cuando se desee. Por otra parte, esta interfaz no almacenará los objetos ni el inventario, solamente se encargará de facilitar al usuario las funcionalidades descritas para trabajar con el inventario.

[12] Mientras que el programa se halla en funcionamiento.
[13] Un botón es un elemento de Interfaz que tiene la funcionalidad de detectar pulsaciones sobre el mismo.

Sección 3.3.4.3. Menú contextual de acciones

El menú contextual de acciones se encarga de representar aquellas operaciones que podemos realizar al interactuar con un elemento del videojuego. Entre las acciones mostradas, se permite seleccionar una de ellas, realizándose al ser elegida.

Únicamente se muestra y representa las acciones cuando dicho elemento permite la interacción con él, no mostrándose cuando el elemento no permite una interacción, o cuando las posibilidades de interacción son una única opción, en cuyo caso, se determinará que dicha opción es la acción elegida.

Esta interfaz además, ha de cumplir con unos requisitos estéticos entre los cuales definimos que, esta interfaz no debe ocupar más de lo necesario en la pantalla, siendo esta del menor tamaño posible, y permitiendo la visualización de esta como un elemento más del videojuego. A su vez, también ha de ser capaz de funcionar y proporcionar una interacción cómoda en plataformas táctiles.

Dentro de esta interfaz encontramos tres elementos:

- **Acción**: Una acción representa una operación que se puede realizar. Se presenta como un texto que simboliza la acción. Estas acciones podrán ser seleccionadas cuando se deseen ejecutar.
- **Lista de Acciones**: Esta lista de acciones ha de representar todas las acciones que el elemento interactuado contiene, independientemente de la cantidad de las mismas. Por consiguiente, el tamaño de esta lista es variable, y permite almacenar en su interior un número variable de acciones. A su vez, estas acciones han de presentarse de la manera en que permiten ser fácilmente accesibles al jugador.
- **Contenedor**: Es el lugar donde se almacena la Lista de Acciones.

Por último, y similarmente a las demás interfaces, ha de presentar funcionalidad adicional que permita mostrar y ocultar esta interfaz cuando se deba. Esta interfaz no almacena las acciones, sino que recopilará las acciones disponibles en el elemento del videojuego con el que hemos interactuado y las mostrará.

Sección 3.3.4.4. Controles en pantalla

Los controles en pantalla son un tipo muy específico de interfaz que se utiliza con un propósito muy concreto: Facilitar funcionalidad perteneciente a dispositivos de entrada de datos (como un teclado o un ratón) sobre plataformas táctiles. En concreto, los controles en pantalla, simulan la funcionalidad procedente del teclado, al pulsar las teclas contenidas en la cruceta de direcciones[14].

[14] La cruceta de direcciones es un elemento que se haya en los teclados de ordenador (y en algunos mandos controladores de videoconsolas) que permite emitir pulsaciones con direcciones: arriba, abajo, izquierda y derecha.

Esta interfaz, además, ha de mostrarse únicamente cuando el usuario lo especifique, y, esta interfaz, no debe de ocultar ni total, ni parcialmente, a otras interfaces que se muestran delante de esta, estando siempre en un segundo plano.

Esta interfaz está compuesta por dos elementos:

- **Botón de dirección:** Representa una de las direcciones hacia las que el personaje es capaz de andar. Estas direcciones son: arriba, abajo, izquierda, y derecha. Este componente es capaz de detectar cuando está siendo pulsado, en cuyo caso debe activarse y hacer que el personaje se mueva en dicha dirección. Así mismo, es capaz de detectar cuando dicha pulsación se detiene y dejar de hacer que el personaje se mueva.
- **Contenedor:** Se encarga de posicionar los botones de dirección en su lugar, y de posicionarse en un lugar de la pantalla que no sea molesto para el jugador, y que sea fácilmente accesible cuando se utilice en plataformas táctiles (en las que, en la mayoría de los casos, hay que sujetar el dispositivo con las manos a la vez que se utiliza).

Por último, y como requisito estético, esta interfaz ha de presentarse de tal manera que el usuario sea capaz de utilizarla con comodidad, y de identificar su función sin necesidad de textos, siendo válida en cualquier contexto (independientemente del idioma o la cultura del jugador).

3.3.5 Sistema de secuencias

Una vez completas la parte gráfica, la parte lógica y la parte de interacción están sentadas las bases para la elaboración de un juego. Dado que tipo de juego que se pretende diseñar con estas herramientas es del tipo aventura gráfica, el siguiente paso a dar para especializarnos en dicho campo es permitir la progresión en el juego.

Para realizar dicha progresión existirán las secuencias. En ellas, el desarrollador organizará secuencialmente las acciones que realizarán las entidades durante dicha secuencia. Dichas acciones seguirán una exploración en forma de árbol, basando las decisiones en las circunstancias del juego o en las entradas que el usuario proporcione.

Un ejemplo esquemático de secuencia podría ser el de la figura 3.2 que incluye acciones y bifurcaciones.

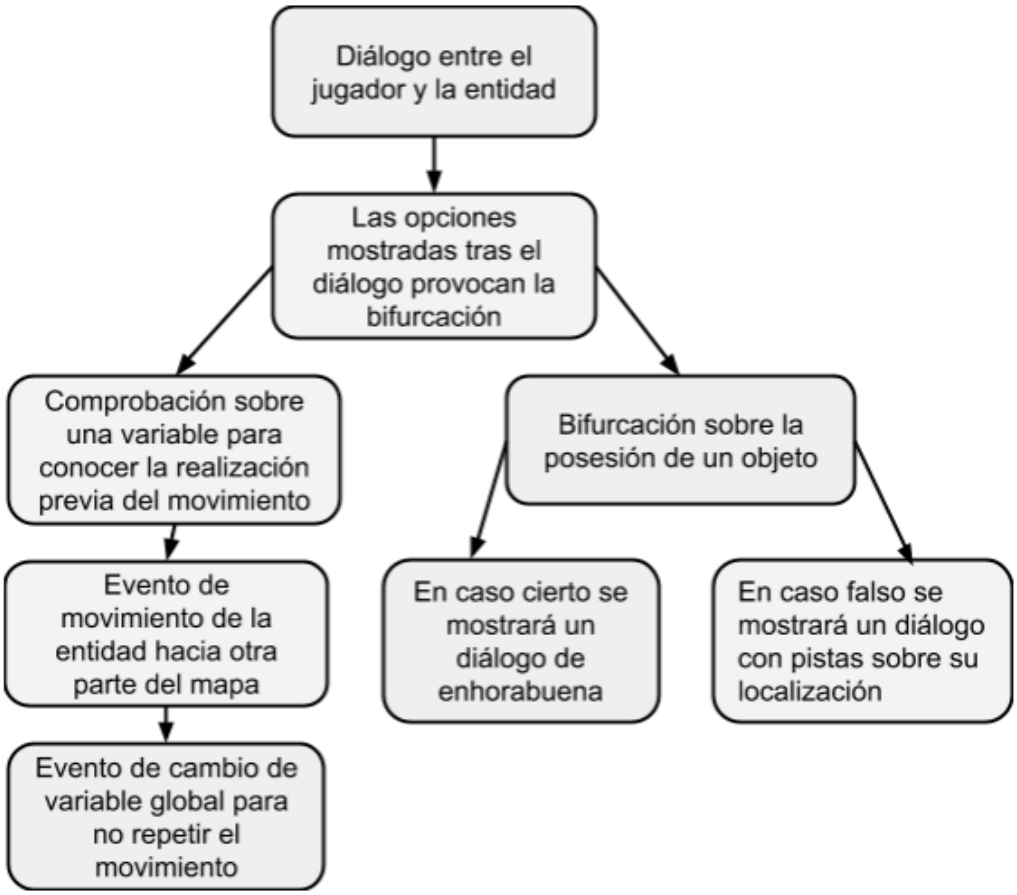

Figura 3.2 - Esquema de una posible secuencia para una entidad.

Por lo tanto, una secuencia será una sucesión de acciones y decisiones con sentido lógico que se sucederán una vez iniciadas de manera no interrumpible y que podrán tener consecuencias visuales.

Las acciones que formarán la secuencia incorporarán entre ellas mecánicas de este género de juegos como diálogos, movimientos de personajes, cambios de estados en el juego, creación o destrucción de entidades, etc. Dentro los objetivos se pretende crear un sistema fácilmente expandible por el usuario de tal forma que puedan ser implementadas nuevas mecánicas por los desarrolladores más experimentados.

Las mecánicas que se implementarán serán:

- **Diálogos**: Conformados por una secuencia de fragmentos de texto con imágenes, deberán representar un diálogo entre entidades en el juego. De esta forma, cuando se realice una pregunta o se deje decidir al usuario, los diálogos finalizarán con una lista de opciones.
- **Opciones**: Al finalizar el diálogo, de haberlas, permitirán al usuario decidir cómo continúa el diálogo, simulando el diálogo del juego con el jugador. (De ser un diálogo vacío se mostrarán estas directamente).

- **Eventos**: Dado que la lógica del juego está basada en el lanzamiento e interpretación de eventos, se podrán almacenar eventos en las secuencias para ser ejecutados en el momento oportuno. De esta forma se dará la posibilidad de realizar acciones como:
    - Movimientos de entidades utilizando el evento mover.
    - Incremento (o decremento) de objetos en el inventario, usando los eventos de objetos.
    - Apertura o cierre de puertas/compuertas/pasadizos/etc. usando eventos de disparadores.
    - Todo tipo de comportamientos usando scripts y eventos personalizados.

  Estos eventos además deberán poder producirse tanto de forma asíncrona como síncrona, lo que significa, que tras lanzar un evento se esperará según se configure o no a su finalización para proseguir con el curso de la secuencia.

Gracias a estas mecánicas, se podrán realizar todo tipo de tareas.

Utilizando comprobaciones se podrá dar sentido y progresión a las acciones. Dichas comprobaciones podrán realizarse sobre todo tipo de aspectos del juego, incluyendo, variables globales, estado del personaje, momento del juego, etc.

Basándonos en algunas de las comprobaciones que se pueden realizar en RPG Maker, las comprobaciones que se implementarán para las herramientas son:

- **Comprobaciones de variables globales o "*IsoSwitches*":** Dichas variables globales podrán ser comprobadas mediante una interfaz y modificadas mediante eventos y serán la forma más fácil de lograr la progresión en el juego.
- **Comprobaciones sobre el personaje**: Principalmente sobre los objetos del inventario. Si realizamos esta comprobación, podremos abrir puertas cuando el personaje tenga llaves u otros objetos. Opcionalmente se implementarán comprobaciones sobre otros aspectos como salud, efectos alterados u otras mecánicas de otros juegos.

Como requisitos para futuras versiones, se podrían implementar:

- **Comprobaciones sobre el tiempo o relojes**: Se realizarán comprobaciones de momentos del juego, permitiendo así realizar acciones a contra reloj u otras mecánicas. Además, se permitirán iniciar temporizadores, con un tiempo predefinido, que irán descontando el paso del tiempo en tiempo real y se podrán consultar, obligando al jugador a ser rápido al realizar algunas tareas.

A continuación se muestran algunos ejemplos de secuencias de acciones en el juego:
<u>Ejemplo 1:</u>
El protagonista habla con la entidad "Abad". El abad le saluda y le comunica que les acompañe a mirar un cartel en la pared. Ambos se mueven hacia celdas cercanas al cartel y hablan sobre el cartel. Al finalizar, se añade un cartel al inventario.

Ejemplo 2:
El protagonista se posiciona delante de un cartel de guardado y habla con él. Se muestra un diálogo que avisa de este hecho y se pregunta al usuario si desea continuar. Al finalizar se envía un evento para guardar la partida.

Ejemplo 3:
El protagonista se posiciona frente a una puerta e interactúa con ella. Este hecho envía un evento de teletransporte que cambia a dicha entidad de sala.

Como se puede observar, se puede realizar prácticamente cualquier cosa con una secuencia siempre y cuando las entidades estén preparadas para interpretar los eventos que se produzcan.

Uno de los objetivos del proyecto es que sea fácilmente ampliable. Debido a esto, los editores y los intérpretes serán fácilmente extensibles. Las nuevas mecánicas para acciones, comprobaciones y eventos serán sencillamente expansibles sobre el proyecto original. Entraremos más en detalle sobre este aspecto en el apartado de diseño y en el apartado de implementación.

Por último, la cuestión que queda por resolver es, quién deberá iniciar estas secuencias y cómo lo hará. Para iniciarlas existirán dos formas. La primera, la entidad *talker* o hablador, la cual, al recibir una acción iniciará la secuencia que tenga configurada. Y la segunda, el gestor de secuencias, que será el encargado de, al recibir un evento específico, iniciar una secuencia almacenada de manera global.

De esta forma en primer lugar las entidades cobrarán vida, y por otro lado, en determinados momentos podremos visualizar secuencias animadas (tal como si de un video se tratara) dentro del juego, al accionar un determinado disparador o *trigger*.

El objetivo principal, de acorde a nuestros principios de extensibilidad, es el de realizar un gestor global para ambos tipos, y que éstos, proporcionando la secuencia a reproducir, inicien su interpretación desentendiéndose de cómo interpretarlas.

Capítulo 4. Análisis y diseño

En este capítulo abordaremos el análisis y el diseño del sistema especificado en el capítulo anterior. Parte por parte, se abordarán las formas de interactuar por parte del usuario, los análisis y el diseño realizado para materializar la especificación.

Los apartados que se expondrán son los siguientes:

- Creación de escenarios.
- Creación y gestión de entidades.
- Gestión de interfaces gráficas
- Gestión de la interacción del jugador con el juego.
- Edición de secuencias.

En cada uno de los apartados se mostrarán las decisiones tomadas hasta llegar a la decisión final que se implementó, y los cambios sufridos en las labores de diseño.

Sección 4.1. Diseño de la creación de escenarios

Dentro de las posibilidades que nos brinda Unity para la creación de escenarios, podemos tener en cuenta la utilización de terrenos [61], la utilización de formas básicas y la utilización de nuestras propias estructuras.

En primer lugar, dado que se especificó la necesidad de establecer entornos 3D, con la capacidad de mantener la estructura 2D, se descartó la opción de utilizar los terrenos propios de Unity, dado la gran cantidad de funcionalidades que realizan ajenas al resultado que nosotros esperamos obtener.

En segundo lugar, se abordó la opción más básica, la utilización de cubos para la formación de escenarios. El primer objetivo a lograr utilizando cubos era el de materializar el hecho de que se debía poder establecer una textura a cada cara, de una forma sencilla para el usuario. Este simple hecho se convirtió en una tarea ardua y compleja de ingeniería inversa, pues se requería conocer el orden de los vértices, sus puntos de mapeado de textura (UVs) [62] y otros aspectos para lograr colocar cada textura.

Tras lograr establecer una textura en la capa superior, y otra distinta en el lateral, se decidió que este proceso iba a ser demasiado complejo de realizar y que probablemente no cubriera todas las necesidades que se tendrían en el futuro. Por ello, se decidió crear una estructura propia a la que se bautizó como celda.

Sección 4.1.1. La celda

La celda es la unidad mínima del mapa, capaz de generar su propia estructura visual y proporcionar las bases para la estructura lógica del juego. Si bien la celda tiene que ser capaz de poder moldearse a gusto del diseñador del juego, la celda deberá proporcionar la vía en la que el juego posiciona entidades en ella.

Sin embargo, dado que la lógica de la celda no es modificada por la apariencia de ella, la celda tendrá por un lado su forma y por otro lado su comportamiento. Las únicas excepciones a este hecho son las propiedades comunes de la celda. Dichas propiedades deberán ser accesibles por ambas partes, pues permitirán que la parte visual y la parte lógica coexistan y el resultado final tenga una apariencia realista.

Dentro de la celda, en primer lugar, la parte lógica constará de al menos, una altura, una altura de andado (sobre la cual se posicionarán las entidades), capacidad para almacenar entidades y comunicarse con ellas, capacidad para recibir eventos del juego y distribuirlos a las entidades dependientes y capacidad para conocer el mapa que la mantiene.

Por otro lado, la parte gráfica deberá ser capaz de materializar de la manera que el diseñador prefiera la celda, la altura y las texturas que se deseen. Para ello, la celda generará una malla procedimentalmente que facilitará las labores de mapeado finales de la textura, y será capaz de mantener las texturas que deberá colocar en cada cara. Para ello, cada celda utilizará una estructura de caras, capaces de conocer qué vértices les pertenecen, qué parte forman de la malla final y cómo realizar el mapeado de sus texturas. Si bien, cada cara recibirá un rectángulo con los puntos referentes a la textura que utilizarán, además deberán conocer el mapeado isométrico que la textura isométrica les proporcionará.

Para realizar el mapeado de las texturas a las caras, se utilizarán los métodos proporcionados por Unity para la unificación de texturas. Cabe destacar que será necesario para que esto funcione que todas las texturas que se utilicen tengan habilitada la propiedad de lectura y escritura (hecho que se explicará más adelante). De no ser así, dicha textura se ignorará pues causaría causar fallos.

Sumando todas estas partes, lo único para establecer la textura a una cara será indicarle a dicha cara que la textura le pertenece y regenerar la malla final para hacer que tome los puntos de la textura.

Para simplificar la creación de las celdas, se unificará la propiedad de la altura con la coordenada Y de la transformada. De esta forma, si visualmente tratamos de levantar la celda, esta crecerá en su número de caras. Dicho crecimiento o decrecimiento mantendrá a su vez las texturas propias de cada cara. Asimismo, podremos seleccionar múltiples celdas y realizar esta acción simultáneamente obteniendo el mismo resultado. Más adelante detallaremos las cualidades de las texturas isométricas.

Finalmente, para la edición de las celdas se creará un editor implementando las clases proporcionadas por Unity para ello, que se muestre a la hora de editar la celda, que permita cambiar la altura, cambiar la forma de la cara superior entre plana, inclinada (45º) y semi-inclinada (22,5º) y cambiar la rotación de dicho panel superior. Por último se añadirá un botón capaz de transportar al usuario al mapa que contiene la celda, para realizar más sencillas las labores de diseño, pues seleccionar el mapa requiere buscarlo en el inspector. De esta forma, con dos clics podremos ir al editor del mapa.

Este editor además se encarga de que las celdas permanezcan alineadas a la cuadrícula, basándose en el ancho de dichas celdas, permitiendo así, que las coordenadas X y Z de la celda sean a su vez su posición lógica.

Sección 4.1.2. Las texturas isométricas, *IsoTextures*

Las texturas isométricas que describimos en la especificación constarán de las siguientes partes:
- Una textura.
- Una configuración de puntos de mapeado.
- Un script capaz de interpretar ambos y almacenarlos.

Dentro de las características de la textura cabe destacar que se quiere conseguir el resultado pixelado retro, la configuración de éstas deberá evitar todo tipo de transformaciones como redondeos a potencias de dos y filtros bilineales. Pese a poder causar problemas de eficiencia, deshabilitar los redondeos, si se quiere lograr el efecto exacto será necesario forzar dicho coste. Para evitarlo, si las texturas fueran ya potencias de dos, no habría pérdidas de eficiencia.

La configuración de puntos de mapeado representarán las esquinas que visualmente se utilizarán como referencia para transformar la textura. En ella aparecerá el punto del eje horizontal que representa la esquina superior y el punto del eje vertical que representa la esquina lateral izquierda. Las otras dos esquinas restantes (inferior y lateral derecha) se calcularán por simetría, siendo estas la diferencia del punto contrario y la distancia del lateral.

Se añadirá un último valor a la textura, el de la rotación, que, debido a que existen tres representaciones distintas de una textura, será necesario en alguno de los casos.

Puesto que objetivo es lograr una representación isométrica, en la mayoría de los casos estos puntos serán el punto medio, a excepción de ser una textura para el lateral, por lo que, existirán tres botones para configurar los valores a los correspondientes a estas tres representaciones.

La representación de los conceptos explicados aparece en la figura 4.1.

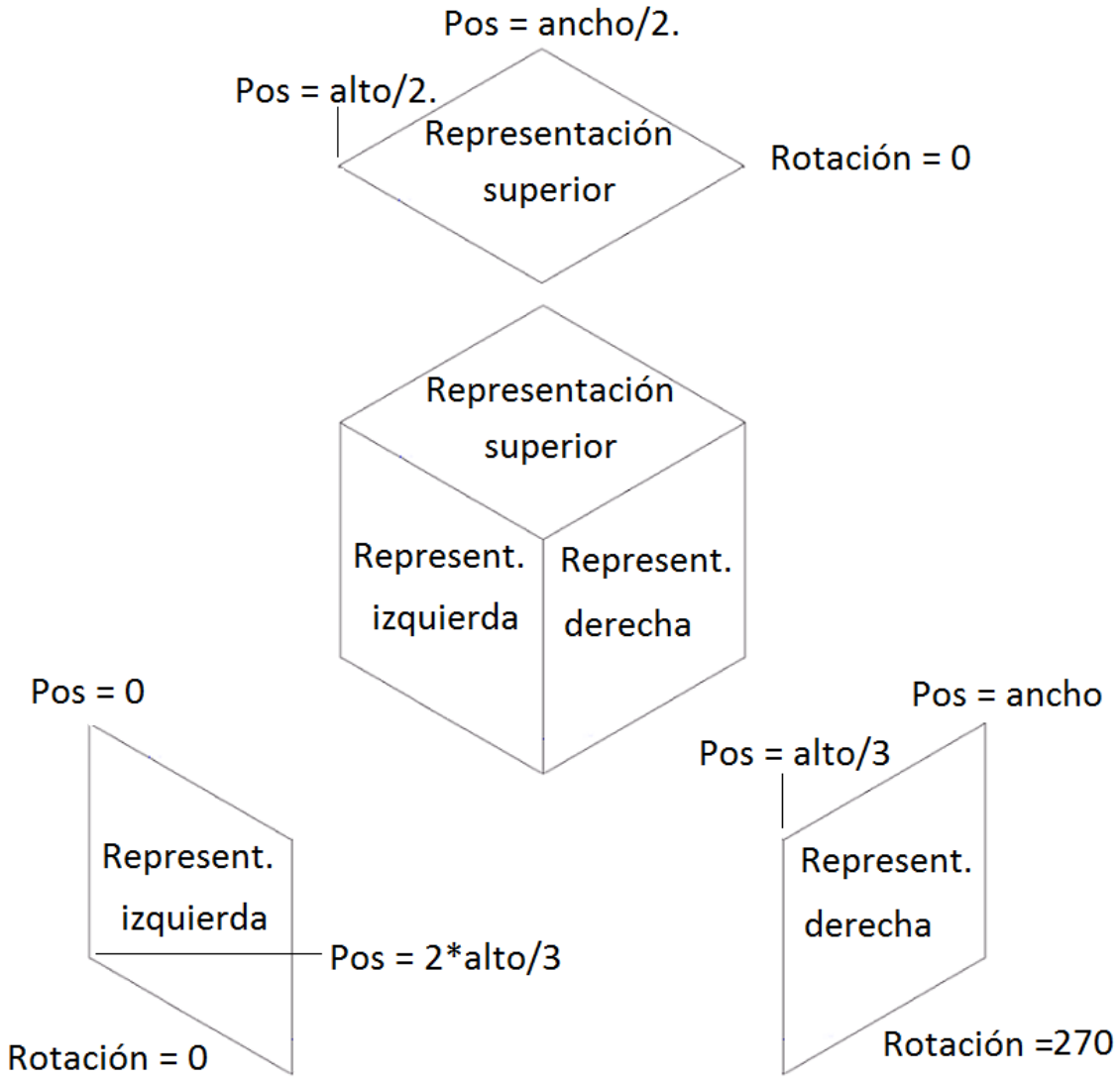

Figura 4.1 - Las configuraciones por defecto de los mapeados contarán con valores de la imagen.

Los puntos son evidentes visualmente, sin embargo, la rotación no es tan evidente debido a que surge como consecuencia de la creación de la celda, por el procedimiento de creación de caras, ya que, la esquina superior de la cara rotará dependiendo de la posición en la celda. Como podemos observar, se numeran las esquinas siguiendo el sentido de las agujas del reloj, la posición de la esquina superior varía en las dos representaciones laterales.

Sección 4.1.3. Las decoraciones, *IsoDecorations*

Las decoraciones son partes de la representación de un mapa, cuyo gráfico es una textura o un elemento tridimensional, que simbolizan elementos del mapa que se pueden identificar por sí mismos, que aportan un significado superior al de terrenos y superficies, y que no son celdas.

Surgen cuando determinados elementos del mapa no pueden representarse pintándose como una *IsoTexture* sobre una cara de una celda. Asimismo, las decoraciones representan ideas que se sostienen por sí mismas.

En primer lugar, se creará un contenedor que ha de ser capaz de escalar las texturas, que se deseen utilizar como decoraciones, para mantener la misma escala y proporciones que se mantiene entre una textura y su celda, y conseguir así, que el tamaño de los puntos de una textura que se encuentra pintada sobre una celda, y el tamaño de los puntos de una decoración, sea el mismo.

Por otra parte, la colocación de una decoración en una celda no es una tarea trivial, pues existen una extensa variedad de modos distintos para el posicionamiento de las mismas. Dichos modos cambian las características de una decoración, la distancia que una decoración mantiene de la celda en la que se halla, así como la forma del contenedor que las ha de representar.

Entre dichos modos de representación encontramos: una representación perpendicular al observador, y una representación paralela a la cara de la celda. Dichos modos, a su vez, se ven afectados por la cara de la celda sobre la que estemos posicionando la decoración, ya que, tanto su posicionamiento, como su deformación no serán la misma si se halla en la cara superior, izquierda o derecha.

Además, estas decoraciones han de poder variar su representación si se desea (de manera opcional para el usuario), pudiendo realizar animaciones y, ser capaces de volver a comenzar el ciclo, o de autodestruirse, al terminar el ciclo de representaciones de la animación. Dicha animación será fácilmente diseñable, y pudiendo establecer parámetros como una lista que contiene la secuencia de imágenes a mostrar, y la frecuencia con la que se cambia la imagen.

Sección 4.1.4. El mapa

Continuando con el desarrollo del editor, el mapa será el contenedor de las celdas. Dicho contenedor deberá ser capaz de instanciar celdas, pintarlas, colocar decoraciones sobre ellas y colocar entidades sobre ellas.

Estos mapas serán, por un lado un contenedor de celdas, y por otro lado encargados de manejar y distribuir los eventos del juego por el mapa. Por ello, serán invisibles, dado que su función es meramente lógica. Su colocación en la escena Unity es indiferente para el desarrollo, por lo que para el diseñador será sencillo administrar sus mapas dentro de la escena.

Edición:

Dentro de la parte gráfica, se proporcionará un editor ampliado al nativo de Unity, que constará de las cuatro herramientas editar, pintar, decorar y colocar entidades.

Para abordar la herramienta de edición se implementará un grupo de funciones CRUD (*Create, Read, Update and Delete*). Para crear celdas decidimos realizar un sistema basado en una rejilla imaginaria que permitiera conocer de antemano la posición de la celda final. La altura de la celda se basará entonces en la altura a la que se encuentre la rejilla.

Para manejar esta rejilla decidimos simplemente bloquear el comportamiento nativo de Unity al segundo botón del ratón, bloqueando el movimiento de la cámara, y moviendo la grilla una altura fija según el ratón se moviera hacia arriba o hacia abajo.

La forma en la que el usuario podrá ver el resultado final de la celda antes de hacer clic e instanciarla será la de mostrar una celda semitransparente en el lugar deseado antes de realizar el instanciamiento final, mostrando a su vez la cara superior iluminada. Esta celda semitransparente la bautizamos como la celda fantasma y el editor del mapa se encargará de destruirla en el momento que se abandone dicha herramienta.

Una vez instanciada una celda, dicha celda será establecida en una coordenada local dentro del mapa. Si otra celda se posicionara en la misma coordenada local, la anterior celda se destruiría, conservando la nueva celda generada. Si lo que quisiéramos fuera modificar la altura sin generar nuevas celdas, en su lugar, deberíamos seleccionar el conjunto de celdas y utilizar la herramienta de edición de transformada proporcionada por Unity o cambiar directamente la altura de ellas en el editor.

Pintado:

Para realizar las tareas de pintado del mapa básicamente tendremos que establecer en las caras de la celda la textura que deseamos que se visualice.

Si bien todas ellas permitirán establecer tanto texturas normales como IsoTexturas en las celdas, cabe destacar que por la característica de la celda las texturas deberán tener el atributo de lectura y escritura habilitado. Si se selecciona una textura sin esta propiedad habilitada, se mostrará una advertencia en el editor y se impedirá posicionar dicha textura en las celdas. Las formas de pintado son las siguientes.

La primera de ellas y más básica es la que consiste en seleccionar una textura e ir posicionándola donde deseemos haciendo clic. Para realizar esto, básicamente proyectaremos una línea desde el ratón y seleccionaremos la celda con la que corte dentro de las celdas de nuestro mapa. Para ello utilizaremos las mallas de colisión (*colliders*) proporcionados por Unity. Una vez obtenida la celda, se iluminará la cara que se encuentre bajo el ratón para indicar al diseñador qué celda se va a pintar. Al hacer clic, entonces, se indicará a la celda que debe posicionar la textura en dicha cara. Si pulsáramos *shift* antes de hacer clic, entonces en lugar de pintar, lo que haríamos sería extraer la textura de la celda, simplificando así las tareas de pintado de mapas con texturas que se repiten varias veces.

Por otro lado como requisito no prioritario, se tratará de desarrollar otro tipo de pintado más automatizado, en el cual se definirán las texturas y los roles que cumplirán cada una de ellas dentro de la celda (cara superior para celdas con altura mayor que X o inferior que X, cara lateral de altura menor que X, o superior...) y que tras aplicar dicha configuración a la celda, ella establecerá todas las texturas automáticamente. Además se proporcionará un botón que realizará dicha acción sobre el mapa completo automáticamente.

También como requisito no prioritario, se tratará de hacer una herramienta capaz de pintar por superficies, que, una vez detectada la cara de corte, se buscarán todas las caras del mismo rol que formen un plano, y se pintarán todas simultáneamente. Las caras superiores, además de compartir rol y vecindad deberán compartir la altura de la celda. Dado que el mapa conoce las vecindades, basándose en el rol buscará los planos laterales conectados y efectuará el pintado completo. Si un plano completo se encontrara visualmente tapado completamente por una celda de mayor altitud, dicho plano se considerará en dos planos.

Decoración:

En cuanto a las labores de decoración, si bien anteriormente hablamos de decoraciones 2D y 3D, dichas decoraciones serán tratadas de una manera similar en cuanto a su colocación. A su vez, se facilitan herramientas para poder poner a disposición del usuario, acceso a los diferentes modos de representación de decoraciones.

En primer lugar, se permitirá elegir al usuario si desea utilizar las decoraciones de este proyecto, o si desea realizar la instanciación de un objeto prefabricado (archivo con la extensión *prefab*) que contenga una decoración tridimensional. Una vez seleccionada la primera pestaña se mostrará una lista de aquellas decoraciones disponibles. De entre ellas, se permitirá al usuario seleccionar una, y tras ello, seleccionar: tanto los parámetros que

definirán tanto el modo de pintado deseado (perpendicular o paralelo), como si dicha decoración es una decoración animada.

Además de los modos que se describen en el Subcapítulo 4.1.3, los cuales son: El pintado perpendicular al observador y el pintado paralelo a la cara, encontramos dos modos adicionales de representación que modificarán la posición de la misma. Estos son: El posicionado centrado en la cara de una celda, o el posicionado centrado en el punto de corte producido por una línea lanzada desde el cursor, paralela a la dirección con la que se orienta el observador, y la cara de una celda.

Tras haber seleccionado tanto la decoración a pintar como el modo de pintado que deseamos, únicamente se debe posicionar en el mapa dicha decoración. Este posicionamiento se realiza mediante la ayuda de una *decoración fantasma*,

La *decoración fantasma* es un tipo especial de decoración que se encuentra entre una representación de una decoración y una decoración instanciada. Dicha decoración se muestra únicamente cuando el modo de edición de decoraciones está activo, además, tenemos una decoración seleccionada, y por último, el cursor del ordenador se halla sobre una celda del mapa. Por último, la decoración fantasma desaparece cuando cerramos este editor.

Finalmente, por defecto se establece que la decoración aparecerá centrada en la cara de la celda, y que al pulsar la tecla de Mayus (*shift*), se produzca el segundo modo en el que el posicionamiento se realizará según el punto de corte anteriormente explicado.

Sección 4.1.5. Diseño del editor de mapas

Una vez descritos y diseñado el comportamiento de los módulos del editor, el siguiente paso es describir la clase que se encargará de orquestarlo todo y darle sentido.

Esta clase será *MapEditor* y deberá implementar *Editor*, siendo *editor* de la clase *Map*. De esta forma, todo *map*, al ser seleccionado, podrá ser editado en la parte del inspector, de forma similar a como se realiza con cualquier terreno de Unity.

Para realizar el editor, las primeras pruebas que se realizaron propusieron un sistema simple, cuya funcionalidad sería almacenada en la propia clase. Sin embargo, al observarse la gran cantidad de código, variables y métodos débilmente acoplados que surgían y observarse la necesidad de que fuera fácilmente extensible, se optó por un diseño modular. Para ello, cada módulo deberá implementar *MapEditorModule* y el *MapEditor* se encargará de ponerlos en marcha cuando sea necesario.

El diseño final será similar al de la figura 4.2.

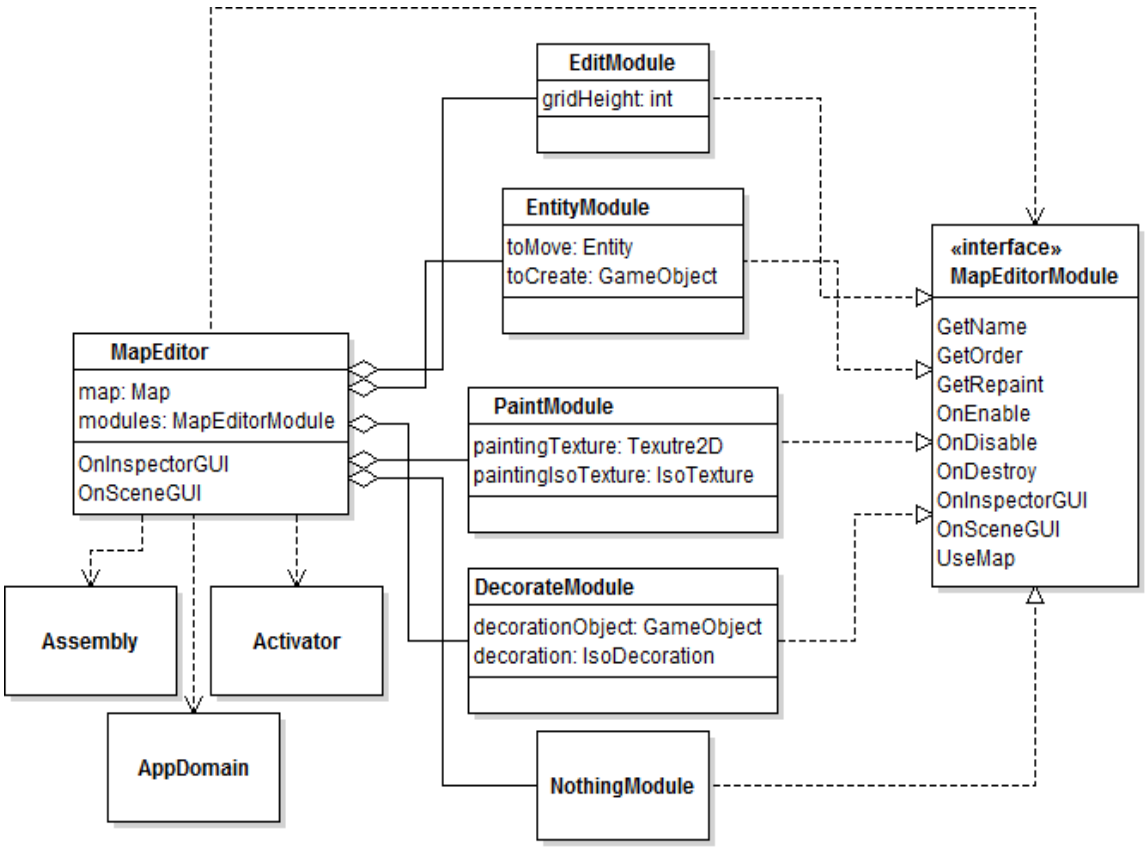

Figura 4.2 - Diagrama de clases del editor de mapas.

A simple vista, el diseño es simple y claro pero cabe matizar un hecho sobre éste. *MapEditor* nunca conoce sus módulos, sino que los crea dinámicamente.

El concepto de dinámicamente es un aspecto muy característico de los lenguajes interpretados (Java, JavaScript, C#...), y en este caso, se refiere a la capacidad de moverse a través de la meta información de las clases para hallar todas las clases que implementan *MapEditorModule* y crear en tiempo de ejecución una instancia de éstas.

Para ello, como guía para la implementación, serán necesarias las clases de *System*: *AppDomain*, *Assembly* y *Activator*. Se entrará en detalle sobre este aspecto en el próximo capítulo. [15]

El funcionamiento de forma general del lanzamiento de los distintos editores del mapa será el siguiente:

[15] http://stackoverflow.com/questions/26733/getting-all-types-that-implement-an-interface-with-c-sharp-3-0

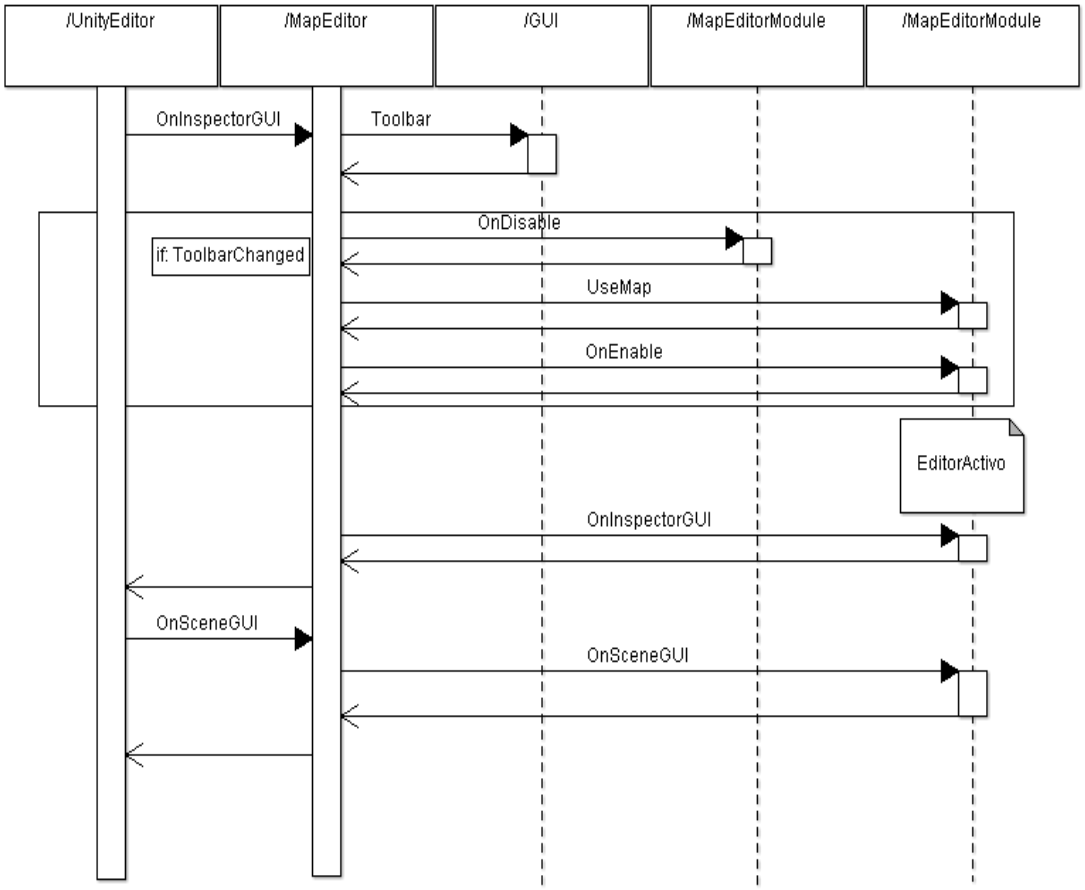

Figura 4.3 - Diagrama de interacción referente al ciclo de dibujado del editor de mapas

La secuencia describe el procedimiento a llevar a cabo en cada dibujado. En primer lugar, se creará una herramienta para seleccionar uno de los posibles módulos. Si se detectara el cambio de herramienta, el editor activo se desactivaría y se habilitaría el nuevo editor seleccionado, permitiendo así que cada editor pueda aprovechar este momento para adquirir datos sobre el mapa o crear objetos que ayuden como apoyo en su funcionamiento tales como la carga en memoria de las *IsoTextures* o *IsoDecorations*.

De no detectarse ningún cambio en la herramienta, se proseguirán haciendo las llamadas sobre el anterior editor activo, sin necesidad de llamar a los métodos de habilitación y deshabilitación.

Por otro lado, la implementación de cada editor se verá en profundidad en el próximo capítulo ya que debido a la gran vinculación con la plataforma es difícil conocer en este punto las vinculaciones exactas con las clases de Unity de cada uno de ellos. Las funcionalidades que deberán realizar los módulos, sin embargo, fueron descritas en el subcapítulo anterior, por lo que las definiciones y mecánicas a implementar se tomarán como referencia para llevar a cabo la implementación.

Sección 4.2. Creación y gestión de entidades

Este subcapítulo se dividirá en la creación de las entidades, por un lado, y la gestión de ellas en tiempo de ejecución por otro.

Sección 4.2.1. Creación de entidades

Para la creación de entidades trataremos de aprovechar al máximo el potencial de componentes de Unity. En el capítulo anterior de especificación hablábamos de la necesidad de que una entidad estuviera conformada por un conjunto de personalidades. Dichas personalidades, sumadas, harían que la entidad conformará un comportamiento total. Por ejemplo, un NPC tendrá capacidad para hablar, para bloquear y para moverse siguiendo algún patrón o aleatoriamente.

Unity facilita esta capacidad con el uso de componentes. Por ello, dejaremos esta capacidad de creación a Unity, el usuario simplemente deberá añadir sus componentes para dar la personalidad adecuada a su entidad.

Para poder anexar las entidades al juego en sí, será necesario que ellas se creen en una celda, cuya referencia cruzada permitirá saber a la celda, quien contiene y a la entidad, qué celda está ocupando.
Se podría haber optado por utilizar el sistema físico de Unity para estas cosas, pero se optó finalmente por un manejo lógico, dado que el sistema tendrá que pensar por el usuario en la mayoría de los casos, como puede ser al moverse, dado que tendrá que establecer rutas y saltos para conocer si se alcanza el destino finalmente.

Sección 4.2.2. Gestión de entidades en tiempo de ejecución

Los objetivos a lograr para cada entidad son ofrecer a sus personalidades, la recepción de eventos, la capacidad de tener autonomía y la capacidad para actualizarse en el tiempo.

Para lograr la primera Unity plantea el sistema de mensajes integrado, en el cual, un *GameObject* (todo objeto contenido en la escena) es capaz de pedir a otro que ejecute un método en cualquiera de sus componentes con la posibilidad de distribución a sus hijos. Visto así, si se quisiera distribuir un evento a todas las entidades simplemente se ejecutaría un mensaje sobre el mapa, que es también un *GameObject*, con el evento como argumento. Sin embargo, se ha descubierto que esto puede causar problemas de eficiencia en determinados dispositivos, por lo que se simulará el funcionamiento de este sistema implementando además la posibilidad de crear un cortafuegos del avance del evento

Para realizar este comportamiento, el objeto Game será el encargado de recibir los eventos en primera instancia, y tras ello, enviarlos al mapa o mapas correspondientes para que continúen con dicha propagación. Por otro lado, Game enviará a diversos módulos los eventos también, permitiendo así centralizar eventos globales como los mencionados anteriormente cambios de variables globales o *IsoSwitches*, inicio de secuencias, etc.

Para ello, Game contendrá un listado de módulos a los que debe distribuir los eventos. Dichos módulos, simplemente deberán implementar una interfaz que unifique la recepción de eventos (*IEventable* por ejemplo), que contenga el método *eventHappened* o evento ocurrido.

Una vez superada la capa de *Game*, el evento se distribuirá a través del mapa a las celdas y de éstas a las entidades que las ocupen. Finalmente, la entidad será la puerta de entrada de estos eventos, y los distribuirá a los comportamientos que la conformen.

Las entidades por su parte, gestionarán una parte de la lógica del juego. En ellas se almacenará su capacidad para permitir o no el paso de otras entidades a la celda que ocupan. Para ello, utilizarán un modelo de lista blanca o de lista negra, en cuya lista podrán almacenar entidades del juego y tipos herederos de *EntityScript*. De tomar una entidad se bloqueará (o permitirá) a esta entidad y de tomar un tipo se bloqueará (o permitirá) a todas las entidades que implementen dicho tipo.

Dado que para realizar el cálculo de rutas es necesario conocer si la entidad puede o no moverse a una celda y, el comportamiento deseado es que exista un límite de salto para acceder a la celda, la entidad también almacenará la diferencia máxima de altura que ésta admitirá.

Por otra parte, la entidad gestionará también la capacidad para crear una decoración en el caso de decidirse una representación de esta como *IsoDecoration*. Recordemos que, dado que la entidad es un Script, cualquier *GameObject* puede convertirse en entidad, incluidos los objetos tridimensionales.

Por último, configurará automáticamente una serie de *EntityScripts* útiles para cualquier entidad, pero cuyo código debe ser abstraído de la entidad en sí. Entre ellos, generará el script andador o *mover*.

El script mover almacena toda la funcionalidad relacionada con el movimiento, tipos de movimiento, saltos, animaciones, etc. Una vez recibe un evento de movimiento, esta deberá pedir una ruta hasta su destino e irla procesando a lo largo del tiempo. De encontrarse una diferencia de altura entre las dos celdas, se realizará una animación parabólica en lugar de la animación clásica lineal. Además, se encargará de animarla.

Para realizar estas animaciones utilizará el siguiente sistema.
De encontrarse un *animator* se ejecutará la animación que corresponda, a saber, andar o saltar a las cuatro direcciones norte, sur, este y oeste.
De no encontrarse, se buscará una decoración y se forzará al cambio de fotograma para realizar la animación.

Para realizar los cálculos de las rutas, se analizaron dos opciones.

El cálculo en el script de la ruta y el almacenamiento local de la misma.
El cálculo en un gestor de rutas y acceso a la ruta paso a paso.

La primera opción, por su parte, permite una gestión local de la ruta aunque obligaría a la entidad a mantener el control del complicado código de planificación de ruta. Al optar por la segunda opción, desvinculamos dicho código permitiendo el acceso a él por cualquier entidad, a la vez que permitimos un mejor desacoplamiento del código para futuros cambios o mejoras del algoritmo.

El planificador de las rutas funcionará de la siguiente manera.

Planificador de rutas

- Será establecido mediante un *singleton*. Este hecho permite:
    - Conocer todas las rutas activas simultáneamente.
    - Prevenir bloqueos, ya que se puede conocer el estado de la entidad en un determinado momento futuro.
    - Recalcular rutas de forma óptima, evitando realizar un cálculo que comprometa otra ruta para evitar futuros cálculos y bucles.
- Utilizará el algoritmo A estrella [63] para el cálculo óptimo de rutas.
    - Para el cálculo de la celda óptima se utilizará una cola con prioridades, permitiendo así un mejor tiempo en el algoritmo.
- Se comportará como un iterador.
    - Siempre que una entidad solicite una ruta, para consumirla irá pidiendo al planificador la siguiente celda a la que moverse, de tal forma que, si fuera necesaria recalcular la ruta la entidad no notaría este hecho.
    - Si se solicita voluntariamente un cambio de ruta podrá seguirse consumiendo la ruta sin apenas cambios.

Otros comportamientos

En el capítulo anterior se habló acerca de los distintos comportamientos que podrían tener las entidades y se dieron una serie de ejemplos o propuestas que se querían implementar. Sin embargo, la mayoría de ellos pretendían abarcar más comportamiento del que deberían para ser reutilizables.

Al realizar el análisis, se planteó que sería más sensato disgregar más aún los comportamientos y, agruparlos con ese sentido en forma de *prefab* u objeto prefabricado.

Continuando con el ejemplo de "Javier el Comerciante", éste comprenderá los comportamientos Vendedor e Inventario. Estos dos comportamientos darán lugar a la siguiente mecánica. El vendedor le dirá a la entidad que para comunicarse con ella debe lanzarle un evento Hablar con ella. Si recibiera dicho evento, consultaría su inventario y generaría una interfaz que representaría la venta de los objetos.

Entonces, ¿quién es "Javier el Comerciante"? Javier es una instancia de un *Prefab* llamado "Comerciante" que contiene en su interior un comportamiento Vendedor y un comportamiento Inventario. Sin embargo, Javier no es el *Prefab* "Comerciante", es un tipo de "Comerciante" que contiene una serie de objetos en su inventario y que se llama Javier.

Unity normalmente utiliza los *prefabs* para lograr que todos los objetos generados con ese p*refab* sean iguales a él. Sin embargo, esto no impide la personalización de pequeñas variables en la instancia del *prefab*, haciendo que éste sea único y cuyos cambios no afecten al *prefab* del que nació.

Si bien el "Comerciante" es uno de los *prefab* que proponemos, este sería un listado de los posibles *prefab* que se podrían generar agrupando distintos comportamientos:

1. Jugador: contiene los comportamientos Jugador, Salud e Inventario.
2. NPC: contiene los comportamientos Hablador, Salud y Patrullador.
3. Comerciante: contiene los comportamientos Vendedor e Inventario.
4. Teletransportador: contiene el comportamiento Teletransportador.
5. Guardador: contiene el comportamiento Hablador configurado para guardar la partida.

Si bien se han mencionado los Prefab para dar sentido a agrupaciones de comportamientos, la definición de los comportamientos sería la siguiente, siguiendo el esquema:

1. Qué comunicaciones proporciona.
2. A qué eventos reacciona.
3. Cómo realiza sus mecánicas.

Jugador

Ninguna
Ninguno
Se registra en el controlador del juego y crea eventos para simular ser controlado.

Inventario

Si se detecta un objeto player comunicará que puede examinarse. (Abrir el inventario).
Evento añadir objeto, evento quitar objeto y evento usar objeto.
Al añadir o quitar un objeto modifica su lista de objetos almacenados y al usar un objeto le proporcionará la entidad sobre la cual se está usando. Si éste se destruye lo eliminará de la lista.

Hablador

Comunicará que puede hablarse con él usando el evento hablar.
Evento hablar, evento siguiente fragmento.
Al recibir el evento hablar, este inicia la secuencia que tenga almacenada.

Teletransportador
No comunicará nada, será imperceptible.
Evento celda pisada, lanzado por cualquier "movedor".
Al recibir dicho evento se teletransportará a la entidad al mapa elegido en la celda seleccionada. De ser el *player*, se mostrará un efecto de oscurecimiento. Puede estar configurado para no teletransportar otras entidades que no sean el *player*.

Movedor
Ninguno.
Evento mover.
Al recibir el evento mover solicitará la ruta e irá procesándola según lo indicado anteriormente. Al avanzar irá enviando eventos celda pisada.

Dentro de los comportamientos que se describieron como requisitos futuros, la definición de ellos será la siguiente:

Vendedor
Comunicará que puede hablarse con él usando el evento hablar.
Evento hablar, evento comprar objetos, evento comprar objeto, evento vender objetos, evento vender objeto.
Al recibir el evento hablar iniciará una secuencia que preguntará si se desea comprar, vender o salir. Según la decisión se recibirá el evento adecuado (comprar o vender) y se mostrará una interfaz para representar el acto. Al comprar un objeto, se lanzará el evento añadir objeto a la entidad compradora y se restará el dinero. Al venderlo se producen los eventos de forma contraria.

Salud
Ninguna
Evento restar salud y evento sumar salud.
Al recibir el evento modifica sus variables de forma pertinente.

Sección 4.3. Creación de interfaces

Durante los capítulos y subcapítulos anteriores se ha mencionado que las entidades crearán interfaces para esquematizar acciones. Hablar, comprar, vender, examinar un inventario, elegir respuestas, etc., son algunas de las interfaces mencionadas.

Unity, por su parte, provee un sistema para la creación y pintado de interfaces [64]. Dicho sistema está integrado con el motor de Unity y puede ser utilizado de forma opcional por todos aquellos *MonoBehaviour* que se encuentren activados en la escena. Los *MonoBehaviour* reciben varios mensajes a lo largo de su ejecución, entre ellos, el mensaje "*OnGUI*" y es durante éste cuando se puede hacer uso de dichos componentes. La palabra GUI son las siglas de *Graphical User Interface*, en castellano, interfaz gráfica de usuario.

Las componentes de interfaz, se acceden mediante la clase GUI y contienen todo tipo de métodos prefijados para la realización de componentes tales como: botones, cajas, etiquetas de texto, todo tipo de campos, ventanas, etc.

Dentro de las componentes existen dos variantes.

La primera es GUI. Ésta permite la creación de elementos indicando siempre la posición donde se desean pintar y delimitando siempre el área que se desea abarcar. Excepcionalmente, utilizando los métodos *BeginArea* y *EndArea* podremos trasladar las coordenadas a un espacio local.

La segunda y más simple es *GUILayout*. Ésta, a diferencia de la primera solo necesita indicar el área que desea abarcar, pero no la posición o tamaño que sus contenidos abarcarán. Para ello, existen eventos de disposición o *layout* durante los cuales se definirán (automáticamente) estos tamaños. Estos eventos deben ser diferenciados de los normales, pues cualquier cambio en los valores a representar podría provocar fallos en el dibujado.

Para la definición del aspecto de dichas interfaces GUI provee de un sistema de estilos y pieles que almacenan preferencias de aspecto, borde, márgenes, fondos, etc. Las pieles están conformadas por un conjunto lógico de estilos, para ser utilizados de forma automática por los distintos componentes (botones, cajas, áreas, etiquetas, etc.), pudiendo almacenar estilos personalizados para indicar posibles variantes de estos anteriores.

Debido a todas las facilidades que estos sistemas y componentes proveen, se optó por utilizarlos ya que no requieren tiempo de desarrollo, están bien documentados en la documentación de Unity y serán mantenidos por Unity, pudiendo ser mejorados o revisados sin realizar ningún esfuerzo.

Una de las características que se pretenden en este proyecto es lograr que una persona con bajos conocimientos de Scripting o incluso de Unity pueda llevar a cabo un juego. La

simple tarea de elaborar una interfaz que realice tareas complejas puede resultar muy complicada si tenemos en cuenta que será necesario modificar el código de la entidad en cuestión para mostrarla.

Además, dada la desconexión entre los distintos *MonoBehaviour* que se encuentran vivos en la escena, es imposible evitar que las interacciones realizadas con la interfaz se filtren al juego. Cualquier botón pulsado podría conllevar una interacción con la celda o entidad que se encuentren visualmente bajo él.

Para ello, se decidió crear un sistema de gestión de interfaces.

En primer lugar**,** el gestor almacenará y dibujará las interfaces activadas. Dichas interfaces podrán ser dadas de alta y de baja a petición de las entidades, quienes, una vez dadas de alta no necesitarán gestionar las labores que éstas realicen. Para ello, las interfaces deberán tener un método (similar a *OnGUI*) en el cual se pintarán y detectarán las posibles acciones a realizar. Para realizar las acciones, simplemente indicarán al juego los posibles eventos que desean lanzar.

En segundo lugar, el sistema será el encargado de organizar la preferencia de las interfaces activas. Esta preferencia se define de dos formas, el orden en el que capturan los eventos y la capa en la que se pintan. El orden en el que capturan los eventos es una parte fundamental.

Si se desea evitar que una pulsación afecte a otras interfaces se deberá marcar este evento como usado. Las otras interfaces, al detectarlo como usado, ignorarán las acciones que conlleva dicho evento. El orden de estas llamadas es desconocido, pues depende del motor de Unity. Utilizando nuestro gestor, aseguramos que las interfaces sean llamadas en el orden de preferencia correcto.

Por otro lado, el orden de pintado de las interfaces no está relacionado con el orden de las llamadas. Para pintar la interfaz, en el orden correcto se utiliza la variable *depth* de GUI, indicando en ella qué interfaz está por encima. Utilizando esta herramienta, en nuestro gestor, aseguraremos que las GUIs que capturan eventos primero, sean además las que se dibujarán por encima de las otras.

En tercer lugar el gestor evitará que los eventos capturados por la interfaz se propaguen al mundo del juego. Este aspecto se explicará más en profundidad en el próximo subcapítulo de interacción del jugador con el juego. Cada interfaz deberá mantener la capacidad para saber si ha capturado un evento para poder comunicarlo al controlador en el momento que se le pregunte.

En último lugar el gestor permitirá una forma sencilla de comunicar opciones al jugador, tales como las que produciría una entidad, permitiendo que, en lugar de provocar que un evento maneje el personaje, el personaje controle el lanzamiento del evento. Este hecho

es muy importante para aspectos como el movimiento, que requieren el conocimiento de la posición actual para conocer las celdas contiguas, u otros casos en los que sea necesario acceder a referencias o variables de forma local.

Para dar de alta y de baja las interfaces se deberá avisar al gestor de este hecho. Sin embargo, el momento adecuado puede que no sea el momento de la petición, por lo que se esperará al evento correcto para cerrar la interfaz. Por ejemplo si se pidiera quitar una interfaz y el siguiente evento fuera un evento de *layout* se evitaría eliminarla, así como si se pidiera en mitad de un pintado.

Es importante que esto ocurra en un punto determinado, ya que los cuatro factores mencionados anteriormente no ocurrirán en el mismo instante temporal. Los eventos de pintado ocurrirán en *OnGUI*, mientras que los controles de usuario ocurrirán durante la actualización de Game. Si detectáramos en *OnGUI* una petición de cierre, ésta no se hará eficaz hasta que dicha interfaz haya cancelado la propagación del evento de pulsación hacia el mapa del juego.

La interfaz de conversaciones definida en el capítulo tres de especificación se desarrollará utilizando las herramientas propuestas de GUI. Esta se explicará en detalle el próximo capítulo de implementación. Este sistema de conversaciones trabajará con los eventos de siguiente fragmento para avisar a la entidad que esté comunicándose que el usuario ha pedido que la conversación avance. Esta simplemente deberá decirle a la interfaz que cambie sus variables de texto e imagen.

La interfaz del inventario permitirá al usuario inspeccionar sus objetos. Para poder realizar acciones sobre los objetos, al pulsar sobre éstos se creará una interfaz de menú contextual de acciones que incluirá las acciones de usar y tirar.

La interfaz del menú contextual de acciones se hará de una forma circular, para simular una interfaz táctil en la que el usuario pulsa el objetivo y sobre éste aparecen las opciones para que con un simple desplazamiento del dedo y levantamiento en la opción indicada se pueda lanzar la acción. Este menú será una de las interfaces con mayor preferencia, pues será utilizada en varios casos y no tiene sentido que sea ocultada por otras interfaces.

Los controles en pantalla serán la segunda interfaz con mayor preferencia, ya que permitirán simular las pulsaciones y siempre deberán pintarse sobre todo lo demás.

Todas estas interfaces se desarrollarán en profundidad en próximo capítulo, definiendo paso a paso, el orden en el que detectan y comunican sus acciones.

A continuación se explicará cómo se comunicará el jugador con el juego usando varios de los conceptos tratados en este subcapítulo e introduciendo varios conceptos de la componente *Input* y eventos fuera de métodos *OnGUI*.

Sección 4.4. Interacción del jugador con el juego

En el capítulo anterior se explicó la intención de unificar las formas en las que el jugador interactuaría con el juego. Esta unificación surge de la necesidad de tres aspectos distintos. El primero el de lograr un comportamiento uniforme para las interacciones táctiles y de ratón. El segundo el de lograr una interfaz común para las distintas formas de interacción por mando, teclado y teclado en pantalla. En último lugar lograr que las interfaces del juego puedan decidir qué eventos se propaguen hacia el juego y con qué acciones lo harán.

Para lograr el comportamiento unificado de táctil y ratón se decidió renunciar a la opción multitáctil que Unity ofrece, unificando de esta forma la primera pulsación táctil con el comportamiento natural del ratón (al pulsar con el dedo, se realizaría un clic de ratón en dicho lugar). Dicha interacción entonces será leída utilizando la componente que proporciona Unity llamada *Input*. Input proporciona una manera sencilla de leer la posición y estado del ratón, así como unificar las pulsaciones con este estado. Tomando como referencia estos valores, se encapsulará dicha información en una variable propia de este proyecto en la que se podrán modificar estos valores.

Esta variable propia será la encargada de encapsular toda la información que se pasará a todos los interesados en conocer las interacciones de la persona.

Para lograr el comportamiento unificado de los controles, Unity en la componente *Input* facilita unos ejes que ya unifican los controles de movimiento provocados por las teclas del teclado, ejes de un mando u otros periféricos. Dichos controles pueden ser modificados por el usuario que desarrolle el juego en las preferencias del motor de Unity. Para nuestra conveniencia, decidimos encapsular estos valores en la variable propia, permitiendo así, que pudieran ser modificados a posteriori por cualquier interfaz (dando lugar a la creación de controles táctiles).

Para lograr el último hecho, que las interfaces puedan decidir qué eventos se propaguen, durante el proceso de creación de la variable, se pasará por las interfaces para que éstas decidan si desean que se propague antes de que el mapa pueda decidir.

Por ello el ciclo de ejecución del controlador del juego será el siguiente:

En primer lugar, se creará una variable para almacenar datos.

Acto seguido, se completarán en ella el estado actual del ratón y teclado obtenidos del componente Input de Unity, habiendo unificado el control táctil.
En tercer lugar, se preguntará al gestor de las GUI si alguna de ellas captura ese estado. Si alguna de ellas respondiera de forma afirmativa, se le pediría que completara el evento.

En este punto, la GUI tiene dos opciones:

1. No realizar ninguna modificación, en cuyo caso el evento se extinguirá en ese punto.
2. Realizar una modificación en la variable del evento y decir que el evento se propague.

De manera paralela, la GUI podrá utilizar en este punto el evento para realizar modificaciones sobre ella misma.

En ningún punto se ha comentado este hecho, pero la GUI está preparada para funcionar y responder a eventos de ratón, y para reaccionar a determinados eventos de teclado, pero no existe ninguna conexión entre los eventos de GUI e *Input*. Al comunicarse estos valores de Input a la interfaz, estamos simplificando este hecho, sin cerrar ambas opciones, pero simplificando el acceso por parte de la GUI a los valores de *Input*.

En cuarto lugar, de no haber sido capturado el evento por la GUI, se preguntará al mapa si es capaz de capturar el evento. Por su parte, el mapa realizará los siguientes pasos:

1. Si es evento de ratón tratará de hallar el objeto pulsado.
    a. De ser una entidad, le preguntará a esta sus acciones.
    b. De ser celda, preguntará a sus entidades si tienen alguna opción que comunicar y de no haberlas, si se puede andar sobre la celda creará una acción en la que encapsulará un evento de movimiento.
2. Si es evento de teclado de tecla de acción tratará de preguntar a las entidades de la celda a la que se está mirando y rellenar las acciones que ellas le comuniquen.
3. Si es evento de teclado de tecla de movimiento, creará la acción que encapsulará el evento de movimiento.

Por último, si el evento está marcado para enviar será enviado a todos aquellos que se encuentren registrados al controlador. En principio, el *player* recibirá estos eventos, pero habrá momentos en los que el gestor de secuencias deberá tomar el control de las entradas de usuario. Dado que los oyentes se registrarán utilizando un delegado será posible revertir el delegado al estado anterior al acabar la secuencia.

Como hemos mencionado varias veces el concepto de acción, para evitar confusiones, la definición de acción, es toda aquella interacción que el *player* puede interpretar (así como cualquier script personalizado que lo desee), teniendo como claves, el origen de la acción, el evento necesario para llevarla a cabo e información extra como si es necesario estar contiguo a la entidad para realizar la acción.

Conociendo esta información, el *script* de *player* automáticamente interpretará las acciones, realizando los movimientos hasta el lugar correspondiente y lanzando el evento una vez llegado al destino. Por ejemplo, para lograr comunicarse con un *Talker* el proceso completo es el siguiente.

Primero, el controlador detecta el clic y pregunta a la interfaz si desea capturarlo. Al no capturarlo, el mapa recibe la interacción y la completa con las acciones que la entidad *talker* le ha proporcionado. En esta acción, el *talker* indicará que para comunicarse con él es necesario estar contiguo (a una distancia de una casilla de él) y que deberá lanzar el evento *tal"* tal y como se lo proporciona éste ya pre configurado.

Para ello, la entidad contaba, como se analizó en el capítulo de gestión de entidades, con un método en el que devolvería las acciones que ésta podía ofrecer.

Una vez el mapa ha rellenado el evento, éste marcará que el evento debe ser propagado a los oyentes.

Al terminar, *player* recibirá el conjunto de información e interpretará que debe pedir al planificador de rutas una ruta hasta la entidad, acortada una celda de distancia, y al finalizar la ruta, lanzar el evento que éste le proporcionó.

Otra de las cosas que pueden ocurrir, es que se reciban en el *player* varias acciones como se había comentado anteriormente. En ese momento, *player* creará una interfaz de selección de acciones y le facilitará el evento original que lo provocó.

A partir de ese momento, dicha interfaz capturará todos los eventos de controlador hasta detectar la acción seleccionada, pisando todos los valores del evento de controlador con los valores antiguos y resumiendo la cantidad de acciones del evento original. Así se simula volver a la pulsación original, pero con una sola acción posible.

Sección 4.5. Sistema de secuencias

Siguiendo el hilo del capítulo anterior, tras haber finalizado la forma en la que el jugador se comunicará con el juego, el siguiente paso a trabajar es el de cómo lograr que esas acciones se vean plasmadas en el progreso del juego.

Si bien cualquier persona con un conocimiento medio de Unity y scripting puede lograr este hecho fácilmente, ha sido más de una vez comentado que el objetivo de esta herramienta está orientada a personas con bajos conocimientos de Unity y casi nulos conocimientos de scripting. Por ello, queremos crear un sistema de secuencias que permita simplificar este hecho.

La complejidad de una secuencia puede variar en gran medida, pues como fue explicado en el capítulo anterior, una secuencia es una sucesión de eventos en el juego que pueden incluir movimientos, diálogos y todo tipo de eventos que el usuario quiera programar. Si bien una secuencia está conformada por eventos, también se definió que éstas seguirían una estructura de árbol, pudiéndose crear distintas ramas en función de distintas variables.

Para almacenar estas secuencias, parecerá lógica una estructura similar a la de un árbol, basada en nodos con referencias a otros nodos hijos. Cada nodo de la secuencia simplemente almacenará un objeto de Unity que, por una parte los editores deberán poder conocer y editar, y los reproductores de la secuencia sabrán interpretar.

Dentro de los objetivos que se pretenden destacamos tres, la capacidad de manejar eventos, la capacidad de tomar decisiones y la capacidad de reproducir diálogos. Sin embargo, el sistema no debe quedar cerrado a esto. Es por ello que existe la necesidad de realizar un diseño altamente extensible para lograr que, cualquier usuario que desee realizar funcionalidades más avanzadas, no le suponga un gran esfuerzo de codificación.

Para ello, basándonos en algunos de los patrones estudiados [60] como el patrón prototipo y el patrón cadena de responsabilidades se logró realizar un sistema altamente extensible. A continuación explicaremos ambos patrones y su importancia en este sistema.

El patrón prototipo logra la funcionalidad de crear clones de sí mismo para ser entregados a quien los solicite, siempre que sea pedido, sin que exista la necesidad de conocer la definición o configuración del contenido del prototipo. Es por ello que es muy común ver este patrón en muchas labores, donde un objeto es utilizado para realizar una labor que él se encarga de personalizar gracias a su configuración. Por ejemplo, un sistema de pinceles podría estar conformado con un conjunto de prototipos que son clonados para ser utilizados por la herramienta pincel para pintar de distintas formas.

El patrón cadena de responsabilidades no es menos importante. Este patrón está por encima del patrón prototipo y es el responsable de elegir el editor/intérprete adecuado para cada contenido. El proceso es el siguiente: tal como lo haría una cadena de responsabilidades, si el primer prototipo es incapaz de hacerse responsable del contenido, este comunicará al segundo que trate de hacerse responsable de ello, de no ser así, éste lo comunicará al tercero y así sucesivamente.

Por último, será necesario utilizar el patrón singleton y el patrón factoría para poder llevar a cabo la labor y asegurar que el código que utilice los editores se modifique dinámicamente al modificar estas cadenas.

El patrón singleton, asegura que en todas partes donde se quiera solicitar un editor, se solicite a través del mismo lugar, garantizando que los cambios se propagarán.

El patrón factoría, como su nombre simboliza, es una fábrica. Para fabricar sus editores utilizará los prototipos que le sean facilitados en su construcción y, utilizando la cadena de responsabilidades, hallará el prototipo adecuado a la petición de fabricación que se le haya dado.

Para realizar la fabricación, entonces, qué necesitaremos. Existen diferencias entre la parte de editores y de intérpretes por una sencilla razón. En el editor es lógico querer

cambiar el contenido del nodo pero en el intérprete no. De esta forma, la factoría de editores podrá conocer el editor más adecuado para un nodo, pero deberá crearlo utilizando una clave que el usuario seleccione. Dicha clave podrá facilitarla el propio editor, de tal forma que, si no se quiere cambiar, utilizando la primera clave devuelta por la factoría será suficiente para editar. Sin embargo, si se desea cambiar el contenido, con un simple cambio de clave se generará otro editor para el nodo, cambiando (o adaptando) este, el contenido del nodo a su gusto. El intérprete por el contrario, creará los intérpretes utilizando simplemente el nodo, ya que éste no cambiará.

Finalmente, cada editor deberá realizar las siguientes labores:

- Decir su nombre.
- Conocer sus capacidades de edición.
- Editar el contenido del nodo recibiendo una referencia a este.
- Poder dibujarse utilizando *GUILayout*.
- Materializar los cambios si fuera necesario.

El sistema final creado utilizando los distintos y las distintas clases puede verse representado en el diagrama de clases UML de la figura 4.4.

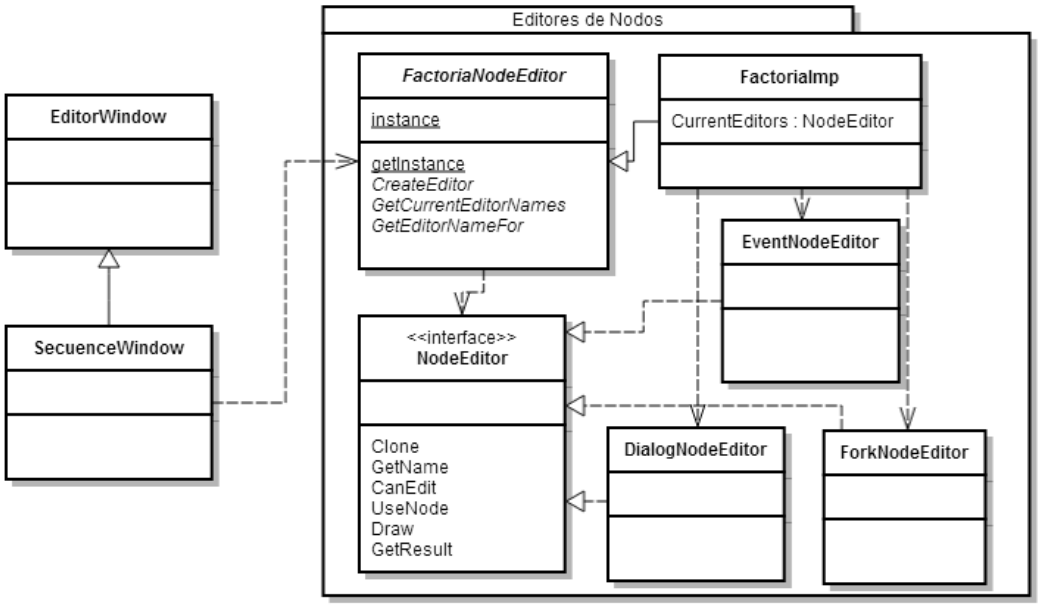

Figura 4.4 - Diagrama de clases combinando los patrones mencionados, factoría, singleton, prototipo y cadena de responsabilidades.

SecuenceWindow mantiene la referencia a una secuencia, siendo ésta explorada en forma de árbol, pasando cada nodo a los respectivos editores.

Una vez alcanzado este punto del diseño, queda clara la extensibilidad del editor de secuencias. Sin embargo, a lo largo de toda la especificación hemos hablado de los distintos tipos de eventos y de los distintos tipos de bifurcaciones o *forks*. Los diálogos,

en su medida, contienen una definición cerrada[16], por lo que no será necesario que sean extensibles.

Entonces, tanto los eventos como las bifurcaciones tendrán que ser extensibles. Para ello, ambos contarán con una estructura de clases muy similar a la anterior.

Sin embargo, contarán con algunas diferencias puntuales. En el caso del editor de eventos, el editor creado no deberá almacenarse, sino que deberá crearse en cada dibujado, ya que el editor no se crea en función del tipo de contenido, sino en función de cómo está configurado el contenido. Esta diferencia tan sutil causa que, por ejemplo, conforme se escriba el nombre del evento (*move* por ejemplo), el editor adecuado para el evento se cambiará automáticamente, creando la sensación de que forzosamente se debe configurar el evento de esa forma, y dando la sensación de que cada evento es un objeto cerrado, pese a ser una estructura altamente personalizable

El editor para las bifurcaciones seguirá un comportamiento similar que combinará una vez más los patrones anteriormente mencionados.
De la misma forma que se contó en el apartado 4.1.6 de este mismo capítulo, las clases que implementarán las interfaces para los distintos editores de bifurcaciones podrán ser cargadas dinámicamente, permitiendo la extensibilidad con la simple extensión de la clase.

Para realizar el procesamiento de las secuencias, existirá un intérprete que maneje la secuencia y el nodo actual. Dicho intérprete procesará la secuencia de forma dinámica, generando un subintérprete adecuado para el contenido del nodo. De la misma forma que existirán editores para diálogos, nodos y bifurcaciones, existirán subintérpretes encargados de procesar adecuadamente cada uno de estos tipos de nodos. El intérprete simplemente se encargará de transmitir actualizaciones y eventos a cada subintérprete para que pueda realizar su interpretación conforme se desarrollen los eventos o sucedan las entradas por parte del usuario. Para finalizar la interpretación, si el nodo actual no existiera o su contenido no existiera (fuera vacío o *null*), la secuencia se daría por terminada en ese momento.

A continuación se entrará en detalle sobre las distintas mecánicas, abarcando su edición e interpretación.

Sección 4.5.1. Diálogos

Como ya se ha mencionado anteriormente, un diálogo constará de una sucesión o lista de fragmentos de diálogo. Cada uno de estos contará a su vez con un nombre y un texto y, de forma opcional, con una foto y una entidad asociada. Dicha lista de fragmentos no tendrá límite de tamaño. Adicionalmente, el diálogo podrá contar con una lista de

[16] Un diálogo está formado por un conjunto de fragmentos, con imágenes y textos, y un conjunto de respuestas opcionales.

opciones. Dichas opciones contarán con un texto que mostrar y serán identificadas por su posición dentro de la lista.

Para realizar el editor de estos, se mostrarán los fragmentos de forma editable y se permitirá seleccionar una entidad, causando que se rellenen automáticamente los campos del nombre e imagen. Las opciones por su parte, conllevarán la creación de diversos hijos del nodo actual para representar las diferentes bifurcaciones en la elección de las opciones. Por defecto, de no haber opciones, existirá un nodo hijo que continuará tras el diálogo, pudiendo indicarse su contenido como vacío para terminar la ejecución.

Para realizar el intérprete de los diálogos, se procesarán los distintos fragmentos, creando una GUI que sepa presentar el fragmento y dándola de alta en el *GUIManager* para que reciba las pulsaciones del usuario que causarán su avance. Las opciones por su parte, contarán con su propia GUI, y al conocer la opción resultante deberemos comunicar al intérprete que el siguiente nodo a ejecutar no es él mismo (para que no avance), sino el hijo indicado. De no haber opciones, al llegar a esta parte se pasará el primero de los hijos.

Sección 4.5.2. Eventos de juego

Los eventos de juego o *GameEvents* son un recurso muy utilizado por el motor para la comunicación de intenciones de realizar acciones. Por ello, para realizar acciones durante una secuencia, los eventos serán la opción más adecuada como lanzador o *trigger*.

Para realizar el editor de eventos, como se mencionó anteriormente, se tratará de maximizar la extensibilidad, permitiendo así a futuros usuarios expertos poder extender el editor fácilmente para la configuración de sus propios eventos de una manera sencilla y eficaz. Sin embargo, si un usuario no quisiera centrarse en realizar un editor, se proporcionará un editor genérico para eventos, menos específico, pero suficientemente potente para una necesidad casual.

Para realizar la selección del editor, la arquitectura deberá ser muy similar a la anteriormente descrita con algunas pero muy sutiles diferencias ya mencionadas en este subcapítulo.

Dentro de los editores específicos para los *GameEvent*, al no ser identificables por tipo (herencias) se identificarán por el nombre. Según su nombre contaremos con los siguientes editores:

- ***Default*** (cualquier nombre): Editor capaz de editar cualquier evento de forma genérica. Permitirá crear nuevos atributos con un nombre. El valor asignado a cada atributo podrá ser cualquier tipo básico o cualquier objeto que herede de *UnityEngine.Object*. Para ello se utilizará un campo especial explicado más adelante.
- ***Move***: Que editará el evento mover, contando con los campos entidad y celda.

- ***ChangeSwitch***: Que editará el evento capaz de cambiar el estado de una variable global *IsoSwitch* al valor contenido por el evento, pudiendo contener cualquier tipo básico.

Para realizar el intérprete, es extremadamente simple. El intérprete de un *GameEvent* simplemente deberá lanzar el evento y pasar al siguiente nodo. Sin embargo, existe una pequeña salvedad en este comportamiento. Los eventos pueden ser marcados como síncronos, siempre que contengan un atributo *synchronous* y este esté configurado como el valor booleano verdadero. De ser así, el intérprete deberá esperar a que se reciba el evento e*vent finished*. Para asegurar que el evento finalizado es el adecuado, en el atributo *event* se deberá introducir el evento causante.

Sección 4.5.3. Bifurcaciones

Las bifurcaciones o *forks* implementan una decisión sencilla, evaluable como verdadera o falsa. Dicha bifurcación decidirá el nodo hijo a ejecutar.

El editor de las bifurcaciones será similar al editor principal, pues cada bifurcación es una clase distinta que implementará *Checkable* (o Comprobable en castellano). Será extensible de la misma forma que el editor principal (no existe la salvedad de los eventos). En principio, en la especificación se habló de hacer los siguientes:

- **Bifurcaciones sobre variables globales *IsoSwitch***: Para ello crearemos una clase *Checkable* que acceda al manager de las variables globales, para acceder al valor y realizar la comparación. Cualquier comparación de tipos distintos se entenderá como falsa. Para realizar la comparación, el editor proporcionará distintos tipos de comparaciones: menor y mayor (o igual) que, igual y distinto.
- **Bifurcaciones sobre personaje**: Principalmente, sobre si el inventario contiene o no un tipo de objeto específico. Para ello, el editor permitirá seleccionar el inventario y el objeto que deberá contener.

Dentro de las ideas o requisitos futuros, se mencionaron las siguientes:

- **Bifurcaciones sobre tiempo de juego o relojes**: En este apartado se mencionó la posibilidad de utilizar el tiempo de juego o un cronómetro/temporizador como objeto de la bifurcación. El primero, el tiempo de juego, es trivial, pero el segundo requiere el uso de un *TimeManager*. Dicha clase escuchará a los eventos de juego (por lo que se registrará en el bucle de Game para recibirlos) y responderá a los eventos de iniciar y detener temporizador. De esta forma, con un evento iniciaremos el temporizador, y con la bifurcación correspondiente podremos utilizarlo y detenerlo.

Todos ellos deberán asegurar la existencia de dos hijos en el nodo para que el intérprete pueda acceder según convenga.

Para el intérprete, es más o menos tan trivial como el intérprete de eventos. Su sencillez radica en la ejecución del método *Check* del *Checkable* (el método Comprobar del Comprobable). Si el método devuelve verdadero, se pasará al primer hijo, y de lo contrario, se pasará al segundo.

Sección 4.5.4. Variables globales, *IsoSwitches*

Los *IsoSwitches* son variables globales de tipo variable almacenadas en un archivo común al contexto del juego. Dichas variables, por tanto, estarán almacenados en un archivo *"IsoSwitches.asset"* de manera estática accesible a través de *Resources* de Unity.

Podrán contener prácticamente cualquier tipo básico. Para ello, dado que Unity no serializa[17] la clase *object* (*System.Object*), se creará una envoltura para los tipos básicos que implemente *ScriptableObject* (heredera de *UnityEngine.Object* y por lo tanto, serializable), que contenga un identificador del tipo y un valor para cada tipo disponible.

De esta forma, los *IsoSwitches* podrán contener cualquier valor a lo largo del juego, podrán almacenarse en un lugar para poder ser retomados en otros momentos del juego y podrán ser preconfigurados para tener unos valores al inicio del juego.

Sección 4.5.5. Editor para múltiples tipos básicos

La clase envoltorio antes mencionada podrá ser editada de manera sencilla utilizando un editor para múltiples tipos básicos. Este editor podrá ser reutilizable en diversos puntos de la interfaz, pues no dependerá del contenido a editar, sino del objeto que reciba como entrada. Por ello, si recibe un tipo entero generará un editor para enteros y si, por lo contrario recibiera un tipo *string*, generaría un editor compatible con *string*. Los tipos admisibles serán los mismos que los que se podrán almacenar en la clase envoltorio de las variables, permitiendo al editor ser utilizado en ese punto. Dado que un evento puede almacenar todo tipo de parámetros, también deberá utilizar este editor, con la salvedad de que podrá contener el tipo *UnityEngine.Object* como parámetro.

El editor funcionará utilizando la clase *Reflection* de C#, haciendo llamadas en tiempo de ejecución a los distintos métodos asociados. Para ello, se guardará una estructura de datos que almacenará, el tipo, el *string* que identifica el tipo y *string* del método que crea un editor para dicho tipo.

Utilizando *Reflection*, se extraerá el método con el nombre indicado, y se invoca en tiempo de ejecución, permitiendo que el editor pueda ser reutilizable para otros fines. Por defecto, el editor llevará una lista de tipos básicos con sus diferentes editores. Por ejemplo, para el tipo *string* se utilizará el método *TextField*.

[17] La serialización es el proceso mediante el cual la escena se convierte en un archivo de manera automatizada.

Sección 4.6. Cierre del análisis y diseño

En este capítulo han sido descritos en profundidad los distintos sistemas, mecánicas y funcionalidades que fueron especificados en el capítulo anterior. Basándose en el contenido de este capítulo, en el próximo capítulo de implementación y resultados se llevará a cabo todo lo aquí descrito, así como los casos particulares de módulos o componentes para cada sistema.

Capítulo 5. Implementación, pruebas y resultados

El trabajo comenzó con una especificación poco concreta y con el objetivo de realizar una herramienta que permita crear juegos isométricos como los descritos en el capítulo 2 (títulos como La Abadía del Crimen [27] o Final Fantasy Tactics [28]), pero con un ambiente más propio de aventura gráfica y puzles. En este capítulo se detallan la implementación, pruebas y resultados que nos permiten comprobar el potencial de esta herramienta.

Como no se conocía suficiente sobre el motor Unity, y no había experiencia en la extensión de este, se comenzaron realizando algunos prototipos sobre una posible forma de editar el mapa. Se ha utilizado la documentación de clases y recursos proporcionados por Unity en su página web como referencia para el desarrollo [65]. Las referencias para la extensión del editor también se encuentran en los manuales de Unity [66].

Sección 5.1. Prototipos

Toda primera toma de contacto comienza por estudiar. Por ello, no se podía comenzar a codificar directamente, sino que se debía comenzar realizando prototipos.

El primer prototipo pretendía crear un editor de mapas utilizando imágenes puras en 2D, sin estructuras 3D ni mapeados de texturas. Para ello, se utilizó un prototipo de sistema de rejilla en el cual, ésta indicaría la altura en la que se crearía la celda. Para instanciar la celda, se realizaba en ejecución la instanciación de un Prefab, por lo que al finalizar la ejecución todo lo realizado se perdía. El resultado de este primer prototipo puede observarse en la figura 5.1*:*

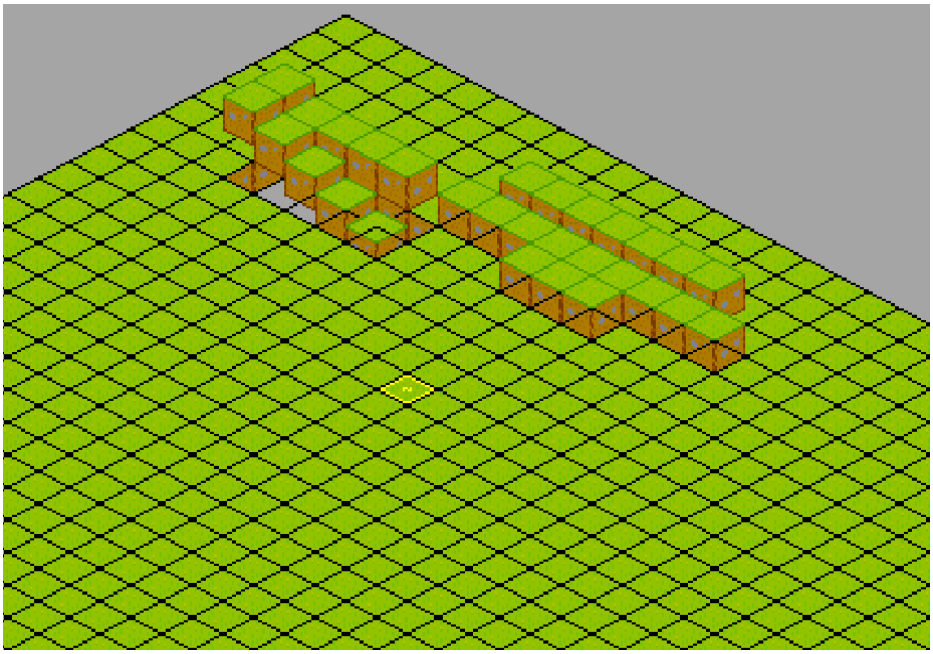

Figura 5.1 - Imagen del primer prototipo del proyecto - Prototipo de editor 2D

Como se puede observar en la Figura 5.1, la celda es una imagen simple, que simboliza un bloque de tierra y sobre el editor completo aparece la rejilla (de color negro).

La rejilla está en el nivel 0 y hay varias celdas a distintos niveles. En este primer prototipo, la rejilla se posicionaba utilizando la rueda del ratón, y no existía la posibilidad de hacer zoom para observar más en detalle. Al hacer clic izquierdo sobre la rejilla, se actualizaba la altura de la celda correspondiente.

Este modelo finalmente se descartó por no aprovechar las posibilidades 3D, por requerir estar en ejecución para ser utilizado y por las escasas y dificultosas posibilidades de pintado de las celdas. Sin embargo, como primera toma de contacto resultó una experiencia muy provechosa.

En el segundo prototipo se trataron de solventar estos dos puntos. Para ello, se comenzó a trabajar en 3D utilizando cubos. En este punto además se comenzaron a tomar contacto con los editores de Unity las ventanas, y las distintas herramientas que proporciona Unity para el manejo de la escena.

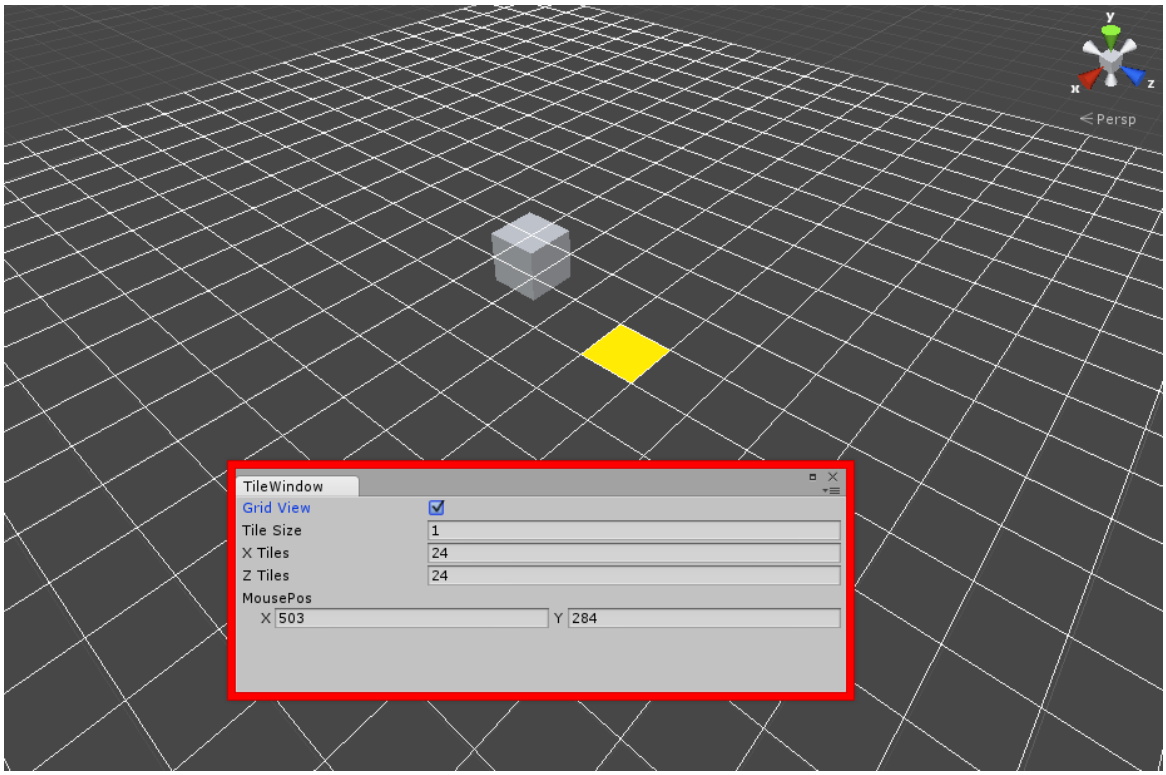

Figura 5.2 - Imagen del segundo prototipo del proyecto - Prototipo de rejilla 3D

Como se puede observar en la Figura 5.2, en el segundo prototipo se trabaja en 3D. Para pintar la rejilla, en lugar de utilizarse una imagen blanca, se utiliza una herramienta de Unity llamada *Handles*. Los *Handles* pueden ser llamados durante el evento *OnSceneGUI* de la GUI y permiten dibujar distintos tipos de formas. En el prototipo utilizamos las líneas y los rectángulos para simbolizar y manejar la rejilla.

En este punto se estudió la capacidad del cubo como celda, pues era muy sencillo instanciarlos con un simple clic, en la posición de la rejilla. Sin embargo, se desconocía si iba a ser sencillo posicionar texturas sobre él y, por otro lado, lograr el efecto del mapeado isométrico.

Por ello, el posterior prototipo que se realizó fue el prototipo de la celda. El Script Texturizador, utilizando como parámetros las distintas texturas debía mapearlas a las caras del cubo. Utilizando el valor "Es Isométrico" se asumía un mapeado isométrico simple, basando las esquinas en el centro del lateral.

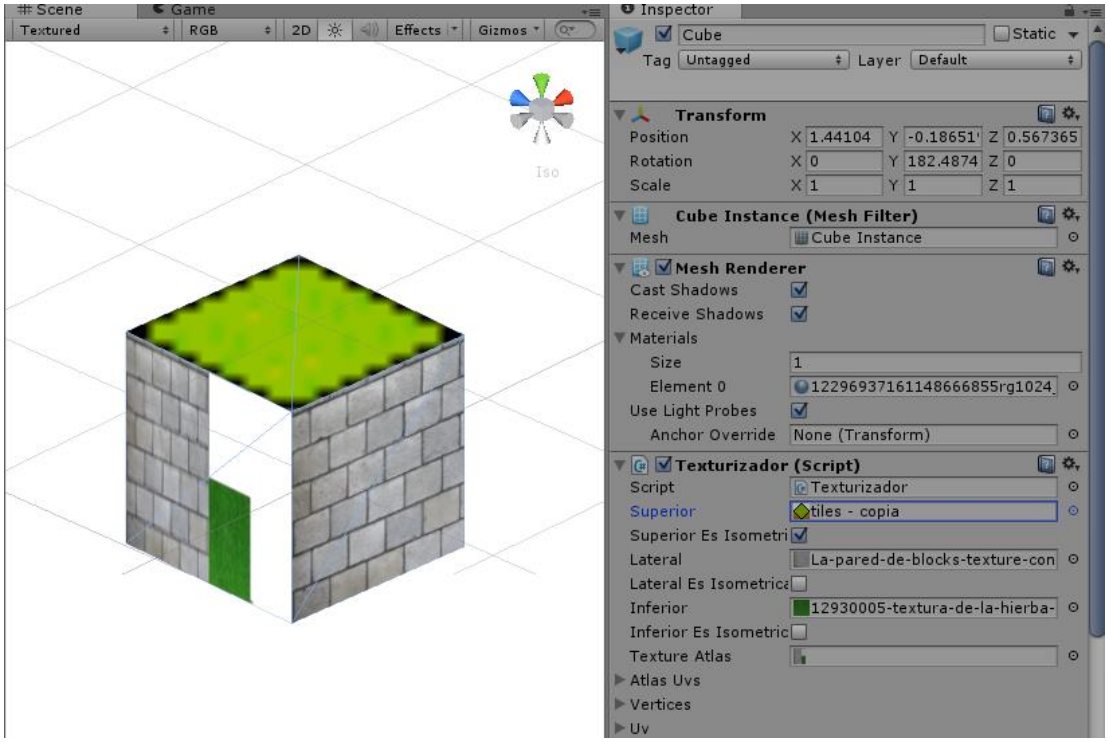

Figura 5.3 - Imagen del tercer prototipo del proyecto: celda basada en cubo

Satisfactoriamente se lograron mapear dos laterales como se observa en la figura 5.3 (incluso más en alguna versión perdida del mismo prototipo), hasta desistir finalmente por la complejidad de este modelo. Realizar la ingeniería inversa pareció simple al principio, pero al comenzar a mapear texturas al revés sin sentido, a crear formas extrañas y deformaciones desconocidas, se detuvo el proceso y se planteó la idea de la celda descrita en el análisis.

Sin embargo, algunas de las pruebas realizadas en este prototipo permitieron demostrar las capacidades de Unity para el agrupamiento de texturas, el mapeado de texturas en mallas 3D y la transformación teórica de textura plana a textura isométrica.

La textura utilizada en la cara superior como se puede observar en pequeño, es la misma textura del cubo de tierra del primer prototipo. En este punto, parece difusa, pero muy poco después se halló la forma para evitar los filtros bilineales de OpenGL y mostrar la textura tal cual es (proporcionada).

Sección 5.2. Implementando la celda

Después de analizar este segundo prototipo, el primer pequeño hito a lograr era crear la celda que se utilizaría para elaborar el editor de mapas.

Recapitulando, la celda es aquella unidad básica de mapa, sobre la que podrán situarse entidades y decoraciones. Dicha celda estará formada por un conjunto de caras que conocerán las texturas que crearán su representación visual.

Para implementarla, en primer lugar será necesario crear la malla tridimensional. Para ello, Unity provee un sistema de mallas (*meshes*) en la que la clase *Mesh* almacena toda la información necesaria para la representación de la malla. Utilizándose los conocimientos sobre Informática Gráfica se realizó un método iterativo que crearía los vértices de dicha malla. Una vez finalizados los cuatro vértices de la base y el primero de la primera línea de altura, de ahí en adelante todos los vértices generados hasta la altura indicada implicarán la generación de una nueva cara. Para ello, se facilitará a la cara la lista de los cuatro índices de los vértices que la conforman (el anterior y los vértices paralelos de la línea anterior), así como la lista compartida donde se almacenan todos ellos. Si la altura de la celda no fuera un número entero, se multiplicaría por dos, se redondearía a entero y se dividiría entre dos. De esta forma, se utilizará el sobrante como multiplicador de altura para la última fila, permitiéndose las medias alturas.

Una vez generada la lista de caras laterales, se procede a la generación de la parte superior. Según los distintos tipos establecidos en la fase de diseño, puede ser, plana, semi-inclinada (22,5º) e inclinada (45º). La generación de la cara plana es trivial, pero las otras dos se generan realizando dos nuevos puntos, generando a su vez dos caras triangulares en el proceso. En la figura 5.4 se representan seis celdas combinando las posibles representaciones de su cara superior y altura:

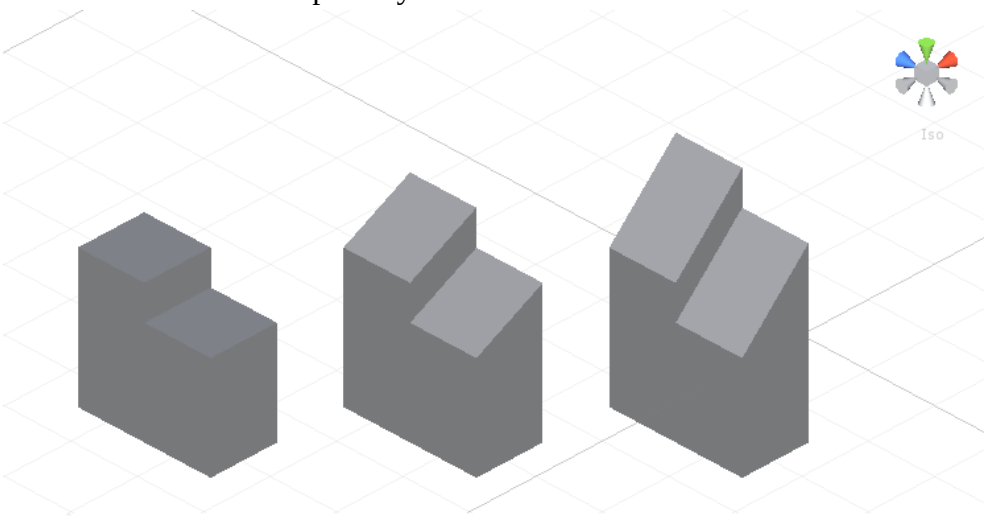

Figura 5.4 - De izquierda a derecha, celda plana, semi-inclinada e inclinada. Junto a ella, su variante de media altura.

Las caras de una celda, realizar el mapeado automático de las *IsoTextures*. Para ello, cada cara contará con una textura y una *IsoTexture* que almacenará las características del mapeado.

Finalmente, para la generación de la malla, se recorrerán todas las caras, solicitando los vértices finales (copias de los vértices originales), los triángulos que la conforman y los mapeados de las texturas, conocidos como UV. Es ahí donde entra la clave del mapeado de las texturas.

Durante el recorrido de las caras se recolectarán todas las texturas y se creará una nueva textura que empaquetará todas. Dicho paquete, en su creación devolverá un mapa que relacionará las texturas con las coordenadas que ocupan dentro del paquete. Estas coordenadas, de no existir un mapeado, serán las que se utilicen para el establecimiento de los UV. De existir el mapeado, se realizará un cálculo para posiciona los UV en las esquinas que establece el mapeado. De existir rotación, simplemente se devolverán los UV reordenados.

Los cálculos simplificados pueden verse en el siguiente fragmento de código de la figura 5.5, siendo los valores contenidos en el *array* de UVs, las cuatro esquinas correspondientes al plano, ya posicionadas en las esquinas de *textureRect* generadas en la agrupación de texturas.

```
uvs[BotLeft] += Vector2(0,textureRect.height*(1f-(mapping.YCorner/mapping.height)));
uvs[BotRight] -= Vector2(textureRect.width*(1f(mapping.OppositeXCorner/mapping.width),0);
uvs[TopRight] -= Vector2(0,textureRect.height*(mapping.OppositeYCorner/mapping.height)));
uvs[TopLeft] += Vector2(textureRect.width*(mapping.XCorner/mapping.width),0);
```

Figura 5.5 - Transformaciones para los UVs de las caras

Al completar vértices, triángulos y UV podemos utilizar la malla en un renderizador con normalidad.

Una vez implementada su capacidad generativa, para su edición se implementó un Editor. El editor, en primer lugar permite establecer si la celda puede utilizarse para andar sobre ella o no, su altura, la forma de su cara superior y la rotación de esta.

El editor cuenta con la característica de poder actuar sobre múltiples celdas al mismo tiempo. Para ello, se estableció la etiqueta *CanEditMultipleObjects* sobre el editor.

Utilizando *SerializedProperties* extraídas mediante *FindProperty* en el objeto a editar, se crearon los campos adecuados de forma automática mediante *PropertyField* para los tres valores, pudiendo mostrar si existen valores mezclados con la propiedad, *showMixedValue*. La rotación de la cara superior cuenta con un botón que la incrementa

90º (ya que solo existen cuatro posibles orientaciones reales). Por último un botón bien grande que indica "*Select Map*" permite seleccionar el mapa de la celda para poder acceder al editor del mapa.

La vista del inspector se puede ver en la figura 5.6.

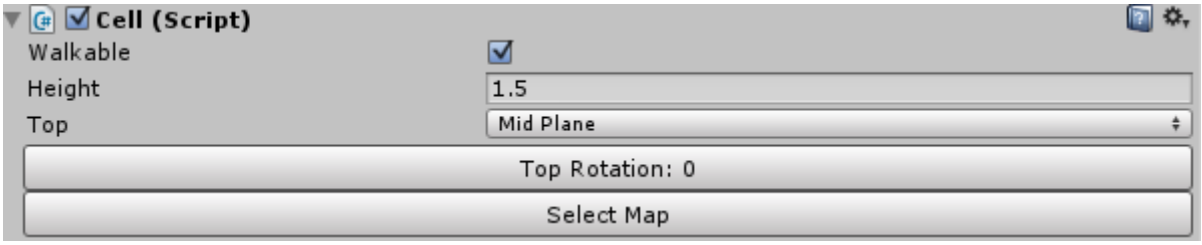

Figura 5.6 - Vista del inspector del editor de la celda.

Las dos últimas funcionalidades que permiten la edición de la celda ocurren durante la actualización de ésta. Para ello, la celda se ha marcado como RunOnEditor para que sea actualizada sin necesidad de ejecutar el juego. Dichas actualizaciones se encargarán de garantizar dos cosas:

La celda se construye siempre desde Y = 0 y cualquier cambio en su coordenada Y provocará que la altura de la celda se cambie como se puede observar en la figura 5.7.

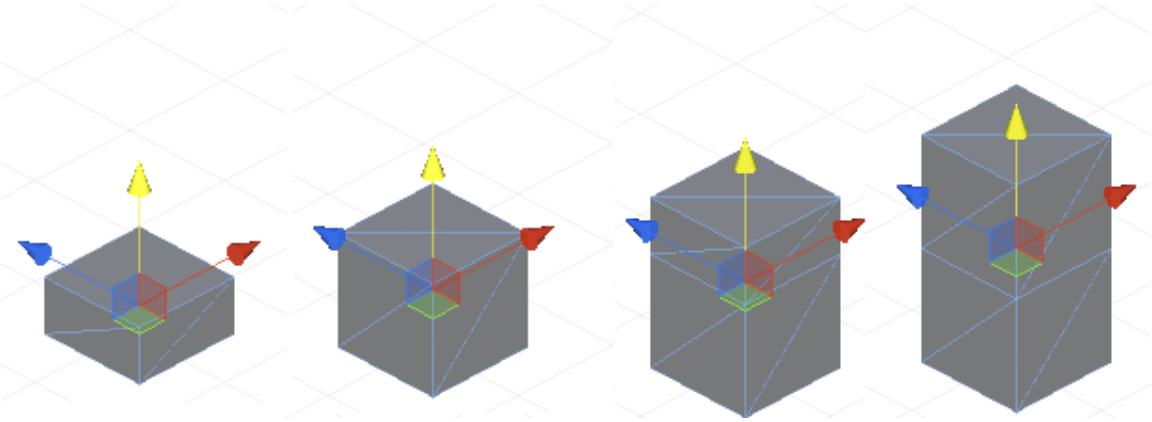

Figura 5.7 - Crecimiento de la celda al levantar la flecha superior.

La celda se alineará automáticamente a coordenadas del mapa. Esto, además de simplificar las búsquedas, crea el efecto de movimiento alineado a la rejilla de edición.

Ambas funcionalidades pueden utilizarse en múltiples celdas simultáneamente.

Sección 5.3. Las IsoTexturas

Una de las motivaciones del proyecto era la de lograr el aspecto retro en perspectiva isométrica. Por ello, y dada la elección de la característica 3D del editor era fundamental crear una forma para lograr fácilmente dicho aspecto. Según la especificación, una IsoTextura almacenaría todas las características de mapeado de una textura normal, que, una vez interpretadas de manera correcta, producirían el mapeado adecuado. A lo largo de la memoria, es común mencionar a las IsoTexturas como información de mapeado, pues suele ser más sencillo para el lector interpretar lo que quieren dar a entender.

Dichas características fundamentalmente eran cuatro: la textura a mapear, la situación de la esquina superior, la situación de la esquina lateral izquierda y la rotación de las esquinas. Las esquinas opuestas también son necesarias para el mapeado, pero al ser perspectiva isométrica, ambas se encuentran en la posición inversa sobre el lado opuesto. Adicionalmente almacenaremos un nombre para identificarla.

Según la especificación, las IsoTexturas serán recursos utilizados por el editor para la creación de los mapas. Al ser un recurso, deberán almacenarse en el directorio el proyecto para poder ser accesibles por los distintos editores.

Unity permite la creación de recursos en su directorio del proyecto conocidos como *assets*. Existen todo tipo de *assets* reconocibles por Unity como texturas, animaciones, terrenos, escenas, etc. El único requisito para poder ser almacenados y serializados automáticamente es el de heredar de Unity *Object*. Dado que *MonoBehaviour*[18] no era la opción correcta (pues una IsoTextura es un recurso y no una componente), se eligió *ScriptableObject*. A través de *AssetDatabase* se podrá almacenar en un *asset* el objeto en la ruta especificada. Para crearlos en la ruta actual, se utiliza *ProjectWindowUtil* que permite esta característica.

Finalmente, para poder acceder al método encargado de crear el archivo, Unity permite la vinculación de métodos estáticos a acciones de menú a través de la etiqueta *MenuItem*. Gracias a esto, se pueden crear IsoTexturas utilizando clic derecho en el explorador del proyecto. La figura 5.8 ilustra las IsoTexturas y su creación.

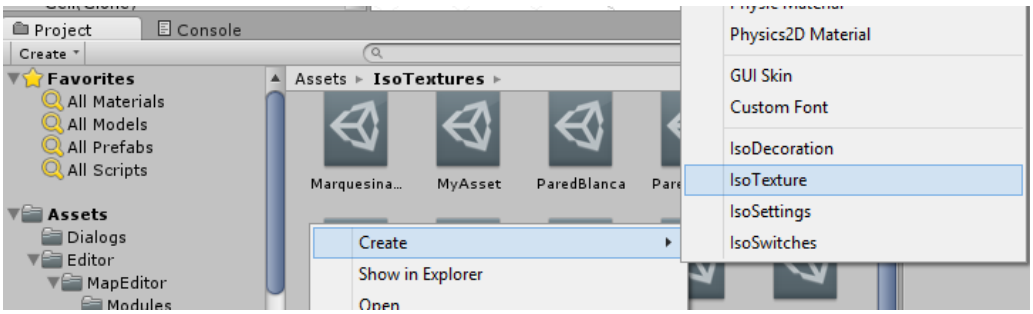

Figura 5.8 - Ventana de proyecto, creando una IsoTextura.

[18] Clase básica para la creación de componentes en Unity.

Sección 5.3.1. El asistente de IsoTexturas

Para el manejo de la configuración de las IsoTexturas, dado que introducir la posición de las esquinas puede resultar engorroso, se decidió crear un asistente visual para la configuración de las IsoTexturas.

Dicho asistente funciona de una forma similar a los editores pero las pruebas realizadas en el inspector impedían algunas características para el dibujado de líneas sobre éste. A causa de este aspecto, se creó una ventana capaz de realizar dicha asistencia. Dicha ventana implementa la clase *EditorWindow*. Para poder abrirla, se vincula un método estático con la etiqueta *MenuItem* a través de la ruta "*Window*".

Dado que es común al inspeccionar una IsoTextura querer editarla, se creó un Editor para la IsoTextura que mostrará un botón para abrir el asistente con la IsoTextura.

El asistente de texturas se implementó con la idea de facilitar de un vistazo la visualización del mapeado que se estaba realizando. Por ello, una vez seleccionadas las características y establecidas las esquinas, se realizan los cálculos sobre el rectángulo de la textura de las posiciones de las esquinas y se dibujan unas líneas sobre el contorno de lo que la cara finalmente mapeará.

Por otro lado, dado que se pretende ser simple, se muestran varios botones con las configuraciones más comunes y se permite el acceso a la configuración más fina en el apartado avanzado.

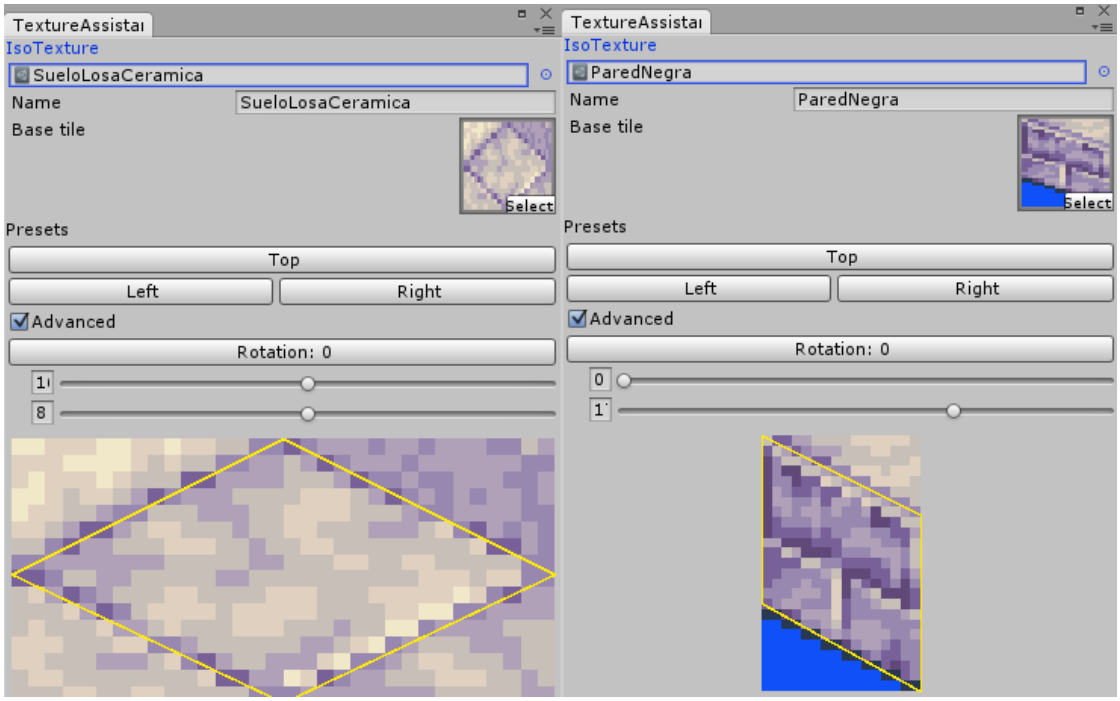

Figura 5.9 - A la izquierda, textura mapeada para cara superior, a la derecha, para la cara izquierda.

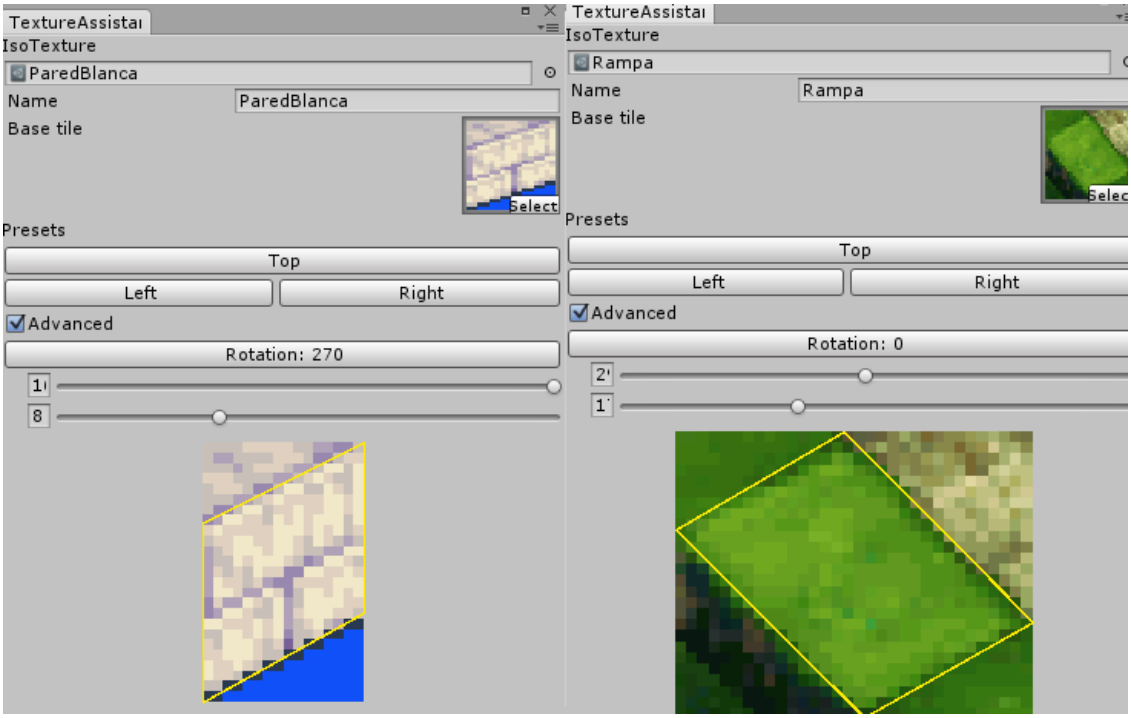

Figura 5.10 - A la izquierda, mapeado para el lado derecho, a la derecha, mapeado de una rampa semi-inclinada.

En las figuras 5.9 y 5.10 se pueden observar las posibilidades del asistente de IsoTexturas. Las tres primeras configuraciones muestran las configuraciones preestablecidas, mientras que la última muestra el mapeado para una rampa semi-inclinada generado por configuración avanzada.

Dada su naturaleza de recursos, las IsoTexturas junto con las texturas pueden ser exportadas e importadas por Unity facilitando la creación de paquetes de recursos para compartir por la comunidad o para distribuir el trabajo en equipos de un proyecto.

Sección 5.3.2. El manejador de texturas

Dada la necesidad de tener un acceso limpio y sencillo a los recursos activos del proyecto, así como de conocer los mapeados existentes para una textura se creó una fachada que simplificaría dichos accesos.

Dicha fachada conocida como *IsoTextureManager*, es un singleton capaz de buscar sobre la base de datos de *assets* las IsoTexturas y entregarlas al solicitante. Por otro lado, si se le facilita una Textura normal, será capaz de buscar todas aquellas IsoTexturas que lo mapeen, permitiendo así, múltiples configuraciones para una misma Textura.

Con esta pieza, las IsoTexturas quedan cerradas, permitiendo su creación, edición y manejo sin necesidad de programación alguna, utilizando puramente recursos de Unity.

Sección 5.3.3. Las Decoraciones

Dada la especificación de las Decoraciones, y el comportamiento deseado, se ha decidido implementar dos elementos para este componente. Estos dos elementos son, por una parte, las *IsoDecoration*, que almacenan datos referentes a las texturas, y el elemento *Decoration*, que se utiliza para representar una *IsoDecoration* dentro del entorno tridimensional.

En primer lugar, la clase *IsoDecoration*, que extiende de *ScriptableObject*, almacena en ella una referencia a la textura a representar y dos números enteros que indican el número de columnas y el número de filas en las que está dividida dicha textura; en el caso de que dicha textura esté dividida, siendo número de columnas 1 y número de filas 1 cuando no está dividida.

Asimismo, la clase *IsoDecoration* tiene la capacidad de cambiar su representación cuando se requiere. Para ello se ha implementado una función *Tile*, que recibe un valor entero, en el que se indica el número del recorte de textura que debe mostrar.

En segundo lugar, la clase *Decoration*, que extiende de *MonoBehaviour*, y que establece el comportamiento que tiene el elemento tridimensional. Por su parte, dicho elemento tridimensional es una clase de Unity llamada *Quad*. Esta clase es un *GameObject*, cuya representación es un cuadrilátero. Dado que las texturas que se desean pintar son de forma cuadrilátera, y únicamente utilizan una superficie para representarse, los *Quads* se adaptan a los requisitos.

Además, esta clase *Decoration* aporta el comportamiento a la clase *Quad* para que, determinando una serie de parámetros, sea capaz de posicionarse y deformarse dados los modos de representación definidos en el apartado de diseño. Estos parámetros están directamente relacionados con los modos, y son: el punto de corte exacto de la línea que, nace en la posición del cursor y mantiene la misma dirección del observador, y la celda donde se desea posicionar, el objeto Celda que será el padre de esta decoración, dos valores lógicos, verdadero o falso, que establecen si dicha decoración ha de permanecer centrada, o si ha de pintarse de forma paralela a la Cara de la Celda cortada.

Una vez parametrizada e inicializada la *Decoration*, se ejecutan diferentes métodos que se encargan de: actualizar la textura y deformar la estructura del *Quad*, y de posicionarse de forma relativa al padre.

La forma de posicionarse es diferente dependiendo de la cara en la que se encuentre el objeto, ya que las distancias que ha de tener de su padre son distintas. Partimos de que el punto donde la decoración se va a posicionar es, si es centrado en la celda, el centro de la superficie de la Celda, y si está centrado en el cursor, el punto de corte que se parametrizó en la inicialización; y que, el punto central de la superficie de la *Decoration*, es el que va a estar en contacto con alguno de los anteriores.

En primer lugar, cuando la decoración se desea colocar en la cara superior de la celda, se levantará sobre el eje Z una distancia equivalente a la mitad del alto de la *Decoration*; y, sólo cuando hagamos posicionamiento centrado en la cara, se posicionará en el vértice inferior de la cara, siendo este el vértice que se encuentra más cercano al observador.

Por otra parte, cuando se desea posicionar la decoración en una cara lateral de la Celda, esta distancia que debemos separar no es tan trivial de calcular cómo en el caso anterior, necesitando realizar complicados cálculos trigonométricos. En primer lugar, se desplaza sobre el eje perpendicular al lateral donde se posiciona la decoración, una distancia equivalente a: la mitad del ancho de la decoración, multiplicada por el coseno del ángulo de 45 grados. En segundo lugar, se desplaza en el eje perpendicular al anterior, una distancia similar. Por último, si se está posicionando centrado en la cara, se posiciona en la arista más cercana al observador. Con estos desplazamientos se consigue que el punto donde se posiciona la decoración sea la arista contraria al lateral de la Celda donde se posiciona la decoración.

Este cálculo puede verse representado en las Figuras 5.11y 5.12.

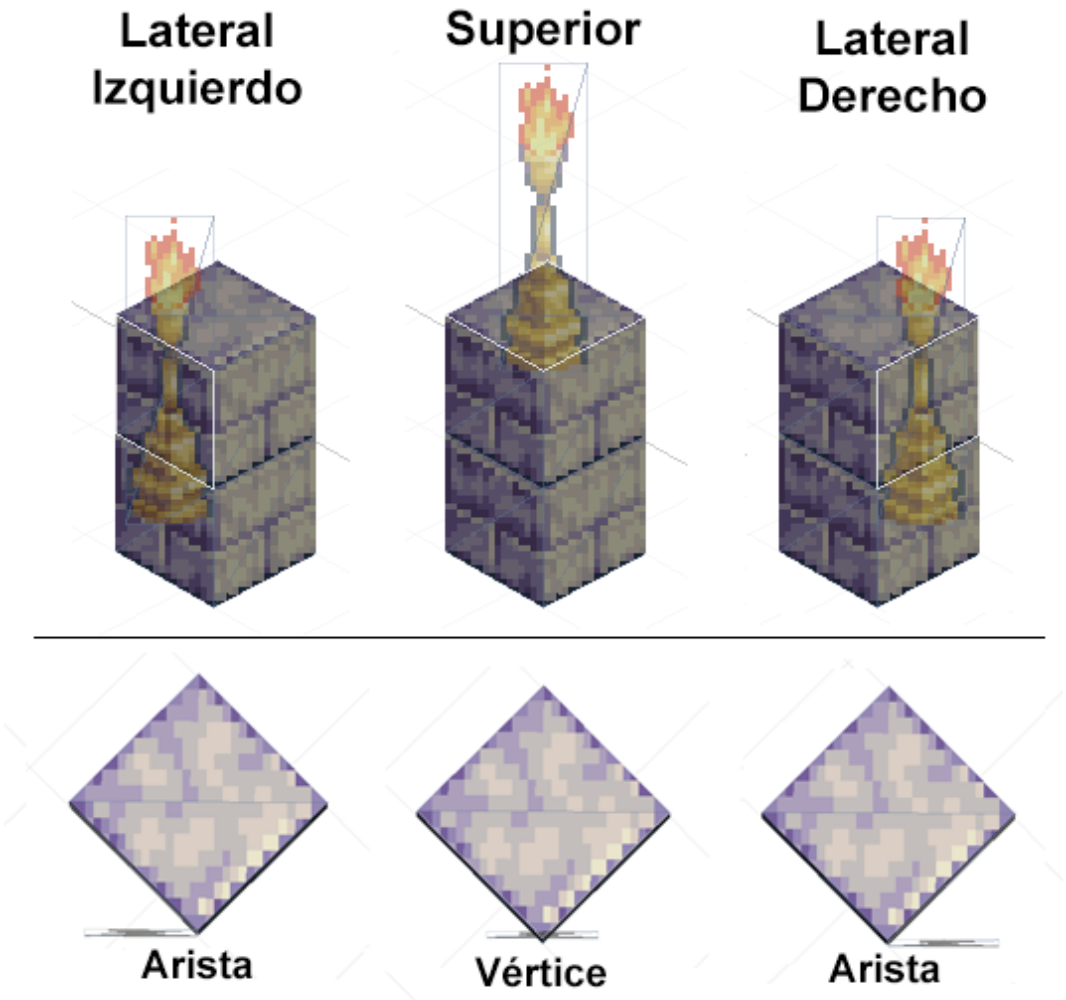

Figura 5.11 - Imagen de los diferentes posicionamientos que adquiere una Decoration al ser posicionada sobre una cara de una Celda.

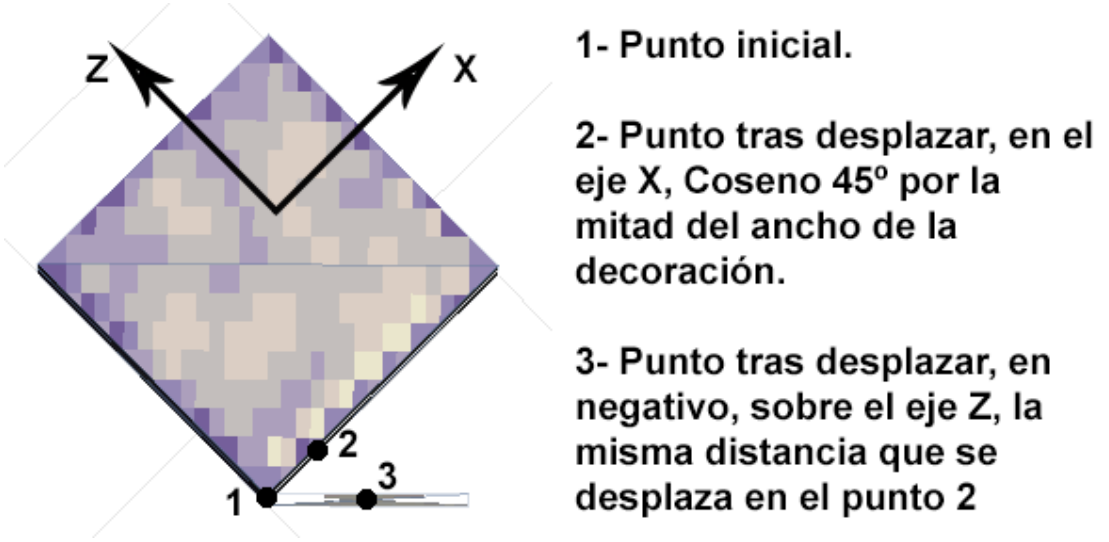

Figura 5.12 - Imagen que representa los puntos por los que pasa la decoración para alcanzar la posición final deseada.

Por último, como la *Decoration* se coloca paralelamente al eje vertical, es decir, el eje Y, y el observador está posicionado, observando hacia el suelo con un ángulo de 45 grados, debemos de realizar un estiramiento en ella. Esta deformación consiste en incrementar la altura de la decoración, la razón de: dos veces el seno de 45 grados.

En el caso del modo de colocación paralelo a la cara de la celda, el posicionamiento no es tan importante como la deformación, dado que cuando la colocación se realiza paralelamente, no se pueden producir cortes con la misma. Por consiguiente, la colocación se realiza utilizando los puntos medios de ambos elementos, y, se separa una pequeña cantidad de la cara para que no se produzcan fallos.

El objetivo de deformación es: deformar la *Decoration* para que, posicionada paralelamente a una de las caras de una Celda, mirada desde el punto de vista del observador, tenga el mismo aspecto que si se visualiza la textura plana bidimensional. Para ello, se realizan distintas transformaciones si la decoración se coloca sobre la cara superior o sobre los laterales de la Celda.

Si la decoración se va a colocar paralelamente sobre la cara superior, únicamente debemos realizar una transformación, la cual es duplicar el alto de la decoración. Con esto se consigue que la decoración se vea correctamente desde el punto de vista del observador.

Por otra parte, si la decoración se va a colocar paralelamente sobre uno de los laterales, necesitaremos realizar complejas deformaciones. En primer lugar, se parte de que la deformación de la decoración iniciar es la que conseguimos tras haber incrementado la altura en dos por el seno de 45 grados, consiguiendo una situación similar a la de las deformaciones no paralelas.

Tras esto se incrementa el ancho de la *Decoration*, siguiendo el Teorema de Pitágoras en:

$$Anchura = \sqrt{2 \cdot (Anchura_{Inicial})^2}$$

Una vez incrementado el ancho, dos de los vértices del cuadrilátero, siendo estos los más alejados al observador, descienden su posición la siguiente cantidad:

$$Bajada = \frac{Anchura \cdot Tan(45º)}{Altura \cdot \sqrt{2}}$$

Tras haber realizado dichas transformaciones puede verse el resultado en la figura 5.13.

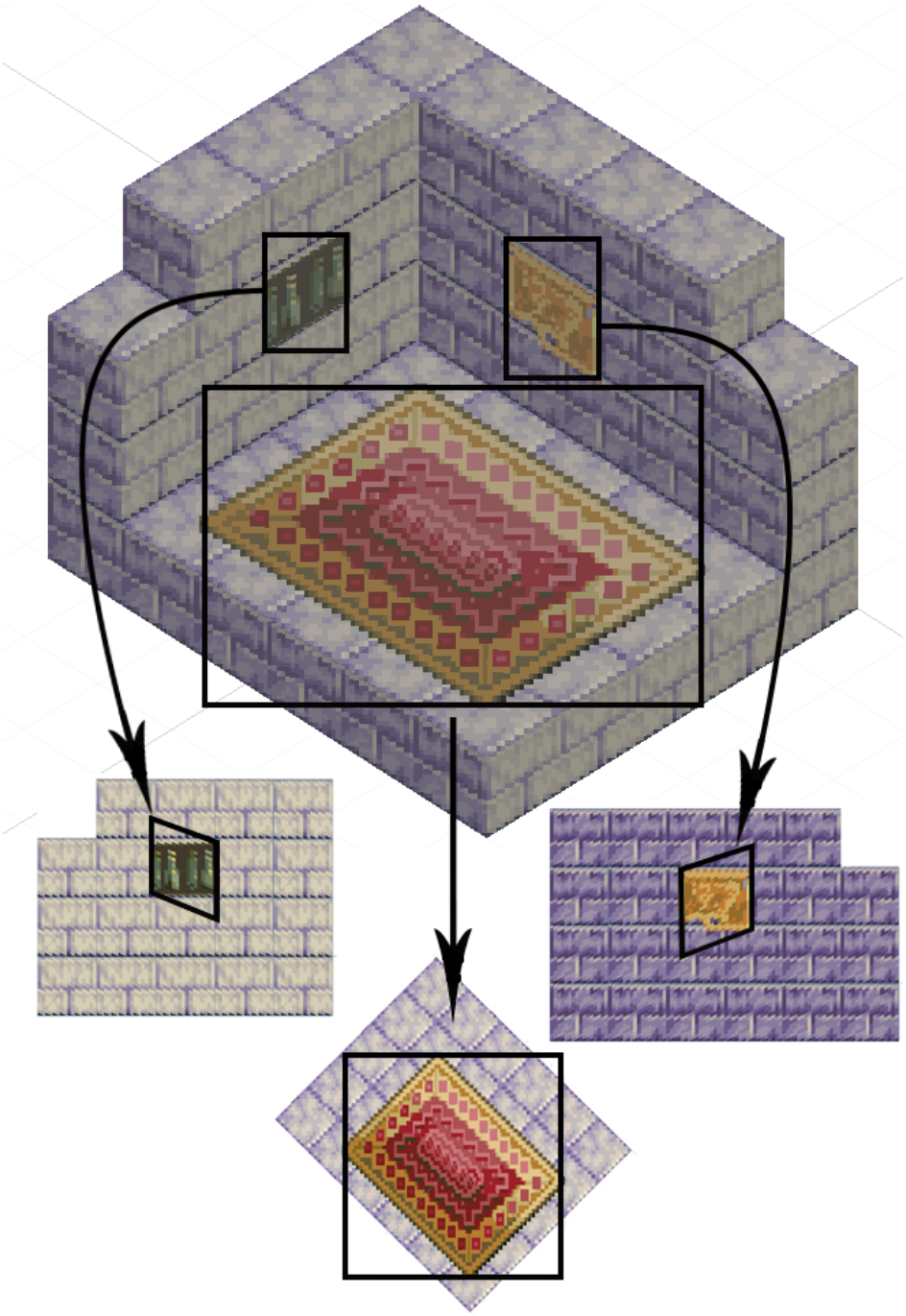

Figura 5.13- Resultado de las transformaciones realizadas al colocar paralelamente una Decoration sobre cada uno de los tipos de superficies.

Por otra parte, para las decoraciones paralelas, la rotación es más importante, pues una mala rotación no conseguiría que estas fuesen paralelas a las caras, y, haría que los planos se corten entre sí. Se definen tres tipos de planos, el plano superior, el lateral izquierdo, y el lateral derecho.

Para el plano superior se realizará una rotación de 90 grados sobre el eje X, y una rotación de 45 grados sobre el eje Y. Para el plano lateral izquierdo únicamente será necesaria una rotación de 90º sobre el eje Y, y, finalmente, para el plano restante, el plano lateral derecho, no será necesario realizar ninguna rotación, pues su posición inicial es paralela a este plano.

Se especificó que el sistema debe de poder colocar sobre el mapa decoraciones tridimensionales. Tras una investigación, se determinó que no era necesaria una modificación de la plataforma Unity para ello, pues esta, automáticamente coloca este tipo de elementos con un posicionamiento similar al que se realiza para posicionar las decoraciones no paralelas. Únicamente debemos de especificar, en la jerarquía de objetos, cual es la celda que deseamos que sea padre de nuestra decoración tridimensional. Esto puede verse en la Figura 5.14.

Figura 5.14 - Decoración tridimensional (Barril) colocada dentro de una celda. Este barril es propiedad de Tooncraft 3D y está disponible, de forma gratuita, en la Asset Store de Unity.

Para completar con la especificación y diseño, se especificó que las decoraciones podrían estar animadas. Estas animaciones, gracias a la capacidad de la clase *IsoDecoration* para poder cambiar su representación, se han implementado con facilidad.

La implementación de las animaciones se ha realizado creando un *Script* de comportamiento llamado *AutoAnimator*, que extiende de *MonoBehaviour*, y que se encarga de cambiar la representación de la IsoDecoración cuando se le establezca. El *Script AutoAnimator* tiene una serie de parámetros, los cuales son: La duración del fotograma, La lista de los fotogramas, el número de repeticiones que debe realizar de la secuencia, y la opción de destruir la decoración cuando termine su animación.

La duración del fotograma de la animación, en Inglés *frame rate*, establece el tiempo que permanece activa y mostrándose un fotograma. La lista de fotogramas establece el orden en el que se suceden los fotogramas, pudiendo estar vacía si se desea realizar una secuencia de fotogramas por defecto, siendo esta aquella que comienza en el primero, se sucede hasta el último, y vuelve a comenzar.

La opción de destruir la decoración es muy importante, ya que, si deseamos tener una animación con un principio y un fin, ha de destruirse al terminar.

Finalmente, uno de los requisitos abstraídos en el primer análisis realizado al estado del arte especificaba que se debería implementar de alguna manera la posibilidad de representar emociones por parte de las entidades. Dichas emociones se crean utilizando *AutoAnimator* y, necesitan un último modo de posicionamiento: El modo de posicionamiento que se produce al establecer como padre a una decoración. Este modo es muy simple, colocando esta decoración hija en la parte superior de la decoración padre, pudiendo crear así pequeños bocadillos con diferentes expresiones animadas. Esto puede verse en la figura 5.14.

Figura 5.15 - El personaje de la derecha muestra una emoción animada de indignación.

Sección 5.4. Implementando el mapa

Conociéndose el objetivo y con unas bases sentadas se comenzó con la implementación del editor de mapas, pues esta era la primera capa a asentar del desarrollo. Un mapa, es una estructura conformada por casillas, decoraciones y entidades.

El mapa, en sí mismo, no tiene representación, sólo es el encargado de dar un sentido lógico al conjunto de objetos que lo conforman y proporcionar una interfaz de comunicaciones al juego. Para su construcción, se utilizará un *MonoBehaviour*, que se añadirá a un *GameObject* vacío y contendrá toda la funcionalidad lógica. Dado que todos los mapas se basan en esta estructura, se creará un objeto prefabricado (*Prefab*) que facilitará las labores de creación del mapa.

La funcionalidad que proporciona contiene métodos para la búsqueda de celdas, de celdas adyacentes y para añadir y quitar celdas al conjunto de celdas del mapa. Todos estos proporcionan una capa de abstracción sobre la capa de funcionamiento nativo de Unity. Dicha abstracción se basa en la capa física (posiciones de los elementos y situación en la jerarquía de objetos) para realizar sus labores.

Para añadir o quitar una celda a un mapa, el mapa proporcionará un método que simplifique la labor de creación del objeto, vinculación como hijo del mapa en la jerarquía y refresco de la lógica del mapa. El mapa sólo deberá conocer el punto global donde se desea crear o borrar la celda. El punto global se deberá pasar a coordenadas locales y tras la transformación las coordenadas X y Z representarán la posición y la Y la altura de la celda.

Para realizar los accesos a celdas singulares se proporcionarán dos vías, por coordenadas globales (útil para el editor) y por coordenadas locales (útil para el juego). En coordenadas globales, se transformará a coordenadas locales y se utilizará el otro método de acceso. En coordenadas locales, si se solicita el acceso a una casilla, se buscará la celda, cuyas coordenadas corresponden con éstas. Para ello, durante la carga del mapa, éste realizará una tabla hash que realice el mapeado instantáneo de todas las celdas, descargando así los tiempos de acceso. El acceso a celdas vecinas se realizará buscando las celdas que se ubiquen en la coordenada X ± 1 y Z ± 1, siendo, por ejemplo, las celdas vecinas a (2,3) -> (1,3), (3,3), (2,2) y (2,4). Si no existe una celda vecina, se respetará su espacio en la lista, pero se almacenará el valor vacío *null*. De esta forma, garantizamos el acceso por direcciones cardinales (norte, sur, este y oeste) de forma sistemática.

Por último, el mapa contará con varias funcionalidades más únicamente útiles para el desarrollo del juego.
En primer lugar, el mapa será responsivo a actualizaciones del juego. Durante actualizaciones o *ticks* las entidades realizarán las decisiones en función de variables y de los eventos que hayan recibido. Para ello, se recorrerán todas las entidades que estén dentro de la jerarquía del mapa (normalmente serán nietas de éste, al ser hijas de las celdas en las que se encuentran), informando de dicha actualización. Por contra, para desactivar la funcionalidad de un mapa simplemente no deberemos actualizarlo.

En segundo lugar, el mapa transmitirá eventos de juego, mediante el método *broadcastEvent* o retransmitir evento, que iterativamente notificará a todas las entidades de este hecho.

Por último, el mapa podrá ser ocultado y mostrado, siendo implementado aprovechando la característica de los renderizadores de Unity para ser habilitados y deshabilitados. Una vez más, se recorrerán iterativamente todos éstos disponiendo su estado de la forma que convenga.

Sección 5.4.1. El gestor de mapas

El gestor de mapas o *MapManager* es el encargado de abstraer la comunicación del juego con los mapas. Esta clase hace de fachada para la clase gestora *Game* como puerta de comunicación hacia los mapas a través de actualizaciones y distribución de eventos.

Los mapas podrán estar activos o inactivos y ser visibles o invisibles. La distribución de actualizaciones y eventos se produce solamente hacia los mapas activos, mientras que los inactivos mantienen su lógica paralizada. Por otra parte, el estado visible e invisible afecta a la visualización del mapa. Normalmente solamente habrá un mapa visible pudiendo haber múltiples mapas activos simultáneamente. Para conocer los mapas activos se mantendrá una lista con éstos mientras que para conocer si son visibles o invisibles, cada mapa almacena una variable.

Además de la distribución de eventos (explicada en profundidad en el subcapítulo 5.10), el gestor de mapas es el encargado de personalizar el evento de controlador. Durante la gestión de eventos de controlador, una parte fundamental ocurre en el gestor de mapas, pues, si ninguna GUI atrapa el evento, el mapa deberá rellenarlo. Para ello, el gestor de mapas facilitará a los mapas activos el evento para que éstos lo completen con la información de celda, entidad y posibles acciones. Esto se realiza de manera iterativa en cada mapa. El proceso es el siguiente, si el botón izquierdo del ratón acaba de ser pulsado, se proyectará un rayo utilizando el sistema de físicas de Unity que nos devolverá la lista de todos los objetos con los que éste ha colisionado. Es importante que los objetos tengan un *collider* pues sino no se producirá tal colisión. Tras esto, se ordenan por cercanía y se realiza una búsqueda de la primera celda o entidad que pertenezca al mapa.

Sección 5.4.2. El editor de mapas

Para extender el editor, Unity facilita un espacio de nombres *UnityEditor*. En dicho espacio de nombres se encuentra la clase Editor. Mediante la etiqueta *CustomEditor*, se podrá indicar que la clase a continuación deberá ser la encargada de editar el script que se encuentre seleccionado en el inspector de Unity. Dado que el mapa es un *MonoBehaviour* y ésta es un script de Unity, se podrá realizar un Editor personalizado para él.

Como se ha comentado anteriormente, el editor de mapas contendrá cuatro herramientas o módulos: edición, pintado, decoración y entidades. Cada módulo dispone de una interfaz genérica que será utilizada según sea seleccionada en el editor.

Dentro de la clase Editor existen varios métodos importantes en la implementación, entre los que se destacan *OnInspectorGUI* (durante las llamadas a GUI del inspector) y *OnSceneGUI* (durante las llamadas a GUI de la escena).

Dentro de la parte del inspector, la clase *GUILayout* proporciona la herramienta *Toolbar*, capaz de mostrar una barra de herramientas. Utilizándola, se implementa el cambio entre módulos de una manera clara y sencilla como se puede observar en la Figura 5.16.

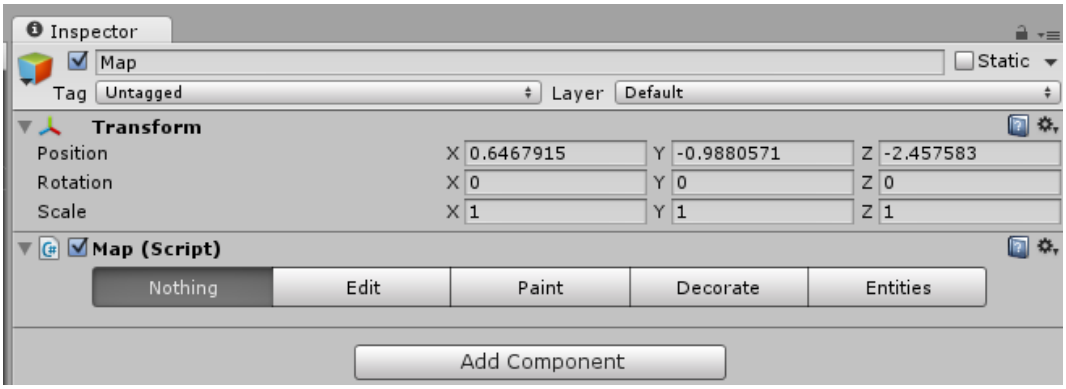

Figura 5.16 - Editor de mapas sin módulo seleccionado.

Al pulsar en cada módulo se realiza *OnDisable*, al deshabilitar, sobre el módulo anteriormente seleccionado y *OnEnable*, al habilitar, sobre el nuevo módulo seleccionado. Cambie o no cambie el módulo, tras esto se enviará la orden *OnInspectorGUI* al módulo activado para que prosiga con el desarrollo del editor.

Por su parte, dado que el editor trabaja sobre la escena, *OnSceneGUI* es vital para poder interactuar con el mapa de forma visual. Durante éste se llamará a *OnSceneGUI* del módulo activo.

Sección 5.4.2.1. Edición

Para realizar la edición, se trató de investigar la forma más simple para el usuario de control. Tras hacer pruebas con la rueda del ratón y el botón derecho del ratón se decantó por esta segunda opción. El hecho era simple, durante las pruebas se echó más en falta el comportamiento de zoom que el de reorientación de la cámara.

Dentro de la vista del inspector, ya que Unity no posee ninguna opción para la configuración de la cámara en vista isométrica, se añadió un botón que la situara de esta forma. Para bloquear el botón derecho, la opción bloquear perspectiva impide a Unity mover la cámara en la escena.

Durante el modo de edición se verá el panel de la Figura 5.17.

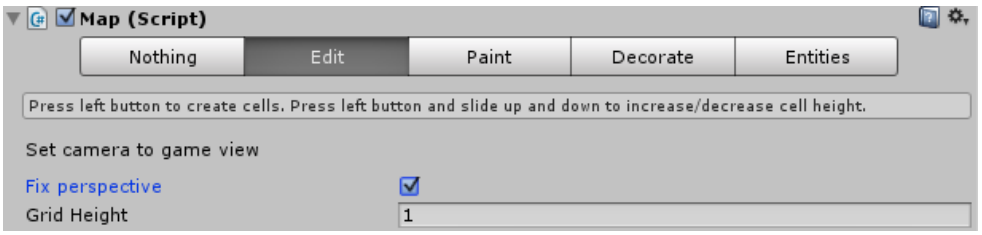

Figura 5.17 - Vista del inspector del módulo de edición.

Por otra parte, el comportamiento durante la escena permitirá la creación de las celdas de manera visual y sencilla. Tomando el modelo de grilla en el que se basan los prototipos, se observó que la grilla era en ocasiones molesta y podía ensuciar la vista, por lo que se decantó por un modelo más sutil, en el que solamente era mostrada la cara superior de la celda a construir. Al ser este modelo excesivamente confuso y poco claro, se realizó el esfuerzo de crear el modelo de celda fantasma.

La celda fantasma representa la celda que se crearía si se hiciera clic en el punto donde se encuentra el ratón. Dicha celda proporciona al usuario una vista previa de su acción en tiempo real, siendo el perfecto modelo híbrido entre limpieza y extensibilidad. Esto puede verse en las Figuras 5.18.

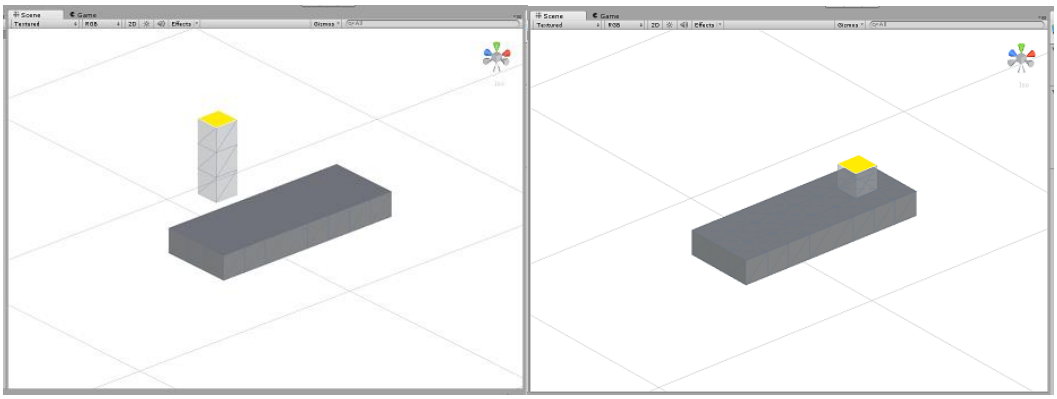

Figura 5.18 - La celda fantasma con tono semitransparente muestra la ce crear.

Como se puede observar en la figura 5.18 a la izquierda, se crea una celda nueva de altura 3 junto al mapa actual. A la derecha, en cambio, se crea una celda sobre una celda ya existente, de tamaño superior. Siempre que se cree una celda en la misma posición de otra, la primera se destruirá.

Al presionar el botón izquierdo del ratón, se observa que, durante el desplazamiento, se crean nuevas celdas hasta que se levante el botón, a modo de pincel. El comportamiento nativo de Unity ante esta acción es realizar una selección con un área rectangular. Para bloquear este comportamiento nativo, mediante la instrucción de *HandleUtility: AddDefaultControl*, lograremos bloquearlo.

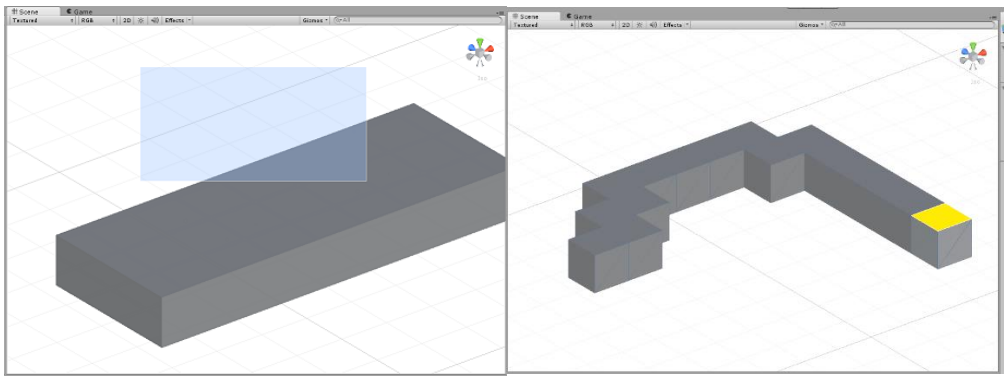

Figura 5.19 - A la izquierda el comportamiento nativo, a la derecha el logrado.

En la Figura 5.19 a la izquierda se observa el comportamiento sin bloquear, y a la derecha, el comportamiento tras haberse bloqueado.

Sección 5.4.2.2. Pintado

Para realizar el pintado, se usan las diferentes texturas e IsoTexturas disponibles. Estas se posicionan en las caras de las celdas del mapa cuando el usuario pulsa sobre ellas.

Al inicializar el módulo, con el método *OnEnable*, éste se encargará de utilizar *AssetDatabase* e ir explorando todos los *assets* del proyecto que sean IsoTexturas, para cargarlas en memoria y poder trabajar con ellas en un panel visual. Aun así, no sólo se podrán seleccionar de manera visual, sino que si se tiene en mente la textura exacta a pintar, podremos seleccionarla y elegir el mapeado disponible de una manera más eficaz. Para ello, la clase *TextureManager* será la encargada de analizar las IsoTexturas en memoria en busca del conjunto de IsoTexturas que compartan una misma textura. Además se añadirá un mapeado vacío, que no establecerá un mapeado, sino simplemente una textura.

Para realizar la selección visual, se solicitará a *GUILayout* un rectángulo donde dibujar las texturas y en su interior, se dibujarán las texturas utilizando *GUI.DrawTexture*. En el caso de estar seleccionada, la textura se pintará con un reborde amarillo.

El resultado final del inspector de la parte de pintado se puede ver en la Figura 5.21.

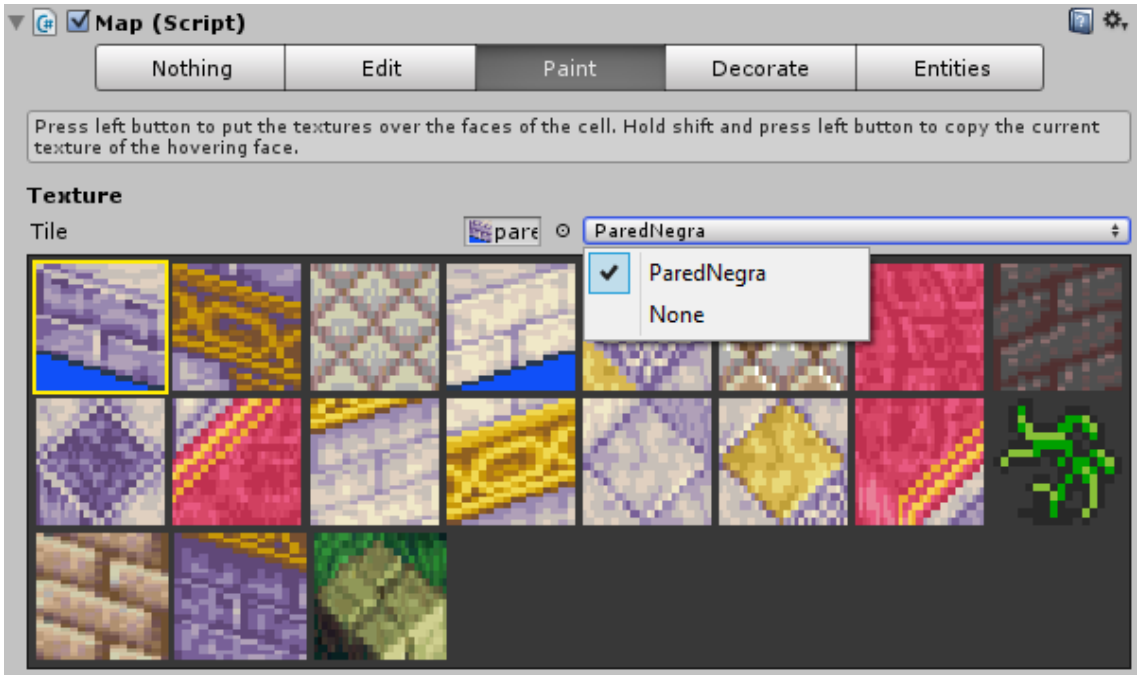

Figura 5.21 - Inspector del módulo de pintado. Texturas extraídas de Final Fantasy Tactics, de la compañía Square Enix.

Al pulsar sobre las texturas de la parte inferior dicha textura se seleccionará.

Por otra parte, en la parte de la vista de escena el editor realizará el pintado de las celdas. Para realizarlo utilizando una vez más el sistema de físicas, proyectaremos un rayo para detectar la colisión de éste con la celda. Una vez encontrada la celda, se le pedirá a ésta que nos diga la cara en la que estamos colisionando.

Para realizar esto, Unity proporciona una clase *Bounds*, que simboliza los límites de una malla, para conocer cuando un objeto colisiona con otro a nivel físico. Aprovechando esta característica, se crea una malla temporal con los puntos que conforman la cara y se generan los límites, *bounds*, de dicha cara. Se extraen los límites y se almacenan. Para comprobar si el punto pertenece a la cara, se realiza el método *contains* de *Bounds* pasando el punto.

Una vez conocida la cara, en ella podremos colocar o extraer información de textura y mapeado y tras realizar la actualización pertinente de la celda, ésta regenerará las texturas, mostrando el cambio.

Para indicar la cara seleccionada, se utiliza el método *DrawRect* de *Handles* que dibuja un rectángulo del color deseado. Dicho rectángulo varía su color en función de la acción que realiza el botón izquierdo del ratón. De ser amarilla, establecerá las texturas, pero de ser azul, las extraerá, permitiendo así simplificar el trabajo del usuario. Esta función estará activa mientras esté pulsada la tecla Mayús, en inglés: *shift*. El estado de *shift* se conoce a través de *Event.current.shift*. Dicho comportamiento se representa en la Figura 5.20.

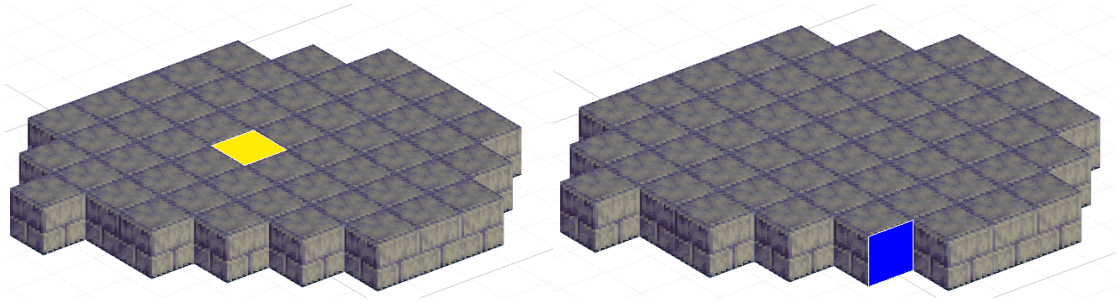

Figura 5.20 - A la izquierda, la cara a pintar, a la derecha, la cara a extraer.

En el momento de seleccionar la cara, como podemos observar, el editor tratará por igual a caras laterales como a caras superiores.

Por último, en ambos modos, el comportamiento nativo de Unity de selección también ha sido bloqueado, permitiendo el mismo comportamiento de pincel del módulo de edición.

Sección 5.4.2.3. Decorado

El funcionamiento del decorado es similar al de pintado, utilizando mecánicas similares a la hora de interactuar con el mapa para colocar decoraciones en él. El proceso a seguir para colocar una textura en el mapa es muy similar al de pintar una casilla, manteniendo editores con estructura semejante.

En primer lugar, tras un breve texto que explica el funcionamiento del módulo, se muestran dos opciones que permiten establecer: Si la decoración se va a colocar de forma paralela a la cara o no, y si se desea que la decoración sea animada o no.

En la parte inferior de estas opciones, se muestra un selector visual similar al selector de IsoTexturas que se encuentra en el módulo de pintado del mapa, con la diferencia de que este selector, en lugar de mostrar la texura de una IsoTextura, muestra la textura de una IsoDecoración. El resultado de este inspector puede verse en la figura 5.21

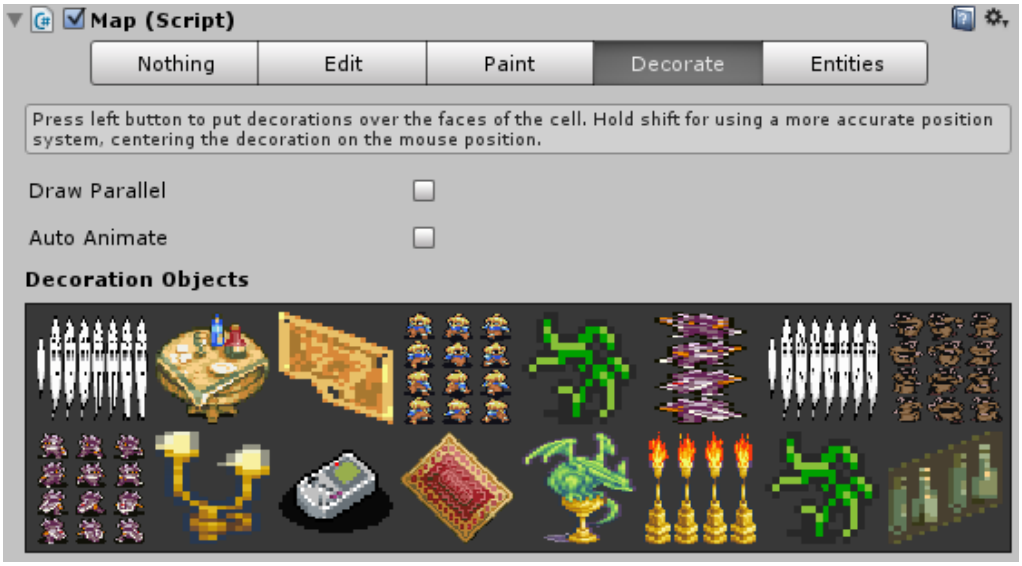

Figura 5.21 - Inspector del módulo de decoración. Las decoraciones mostradas han sido extraídas de: Final Fantasy Tactics Advance y son propiedad de Square Enix, La Abadía del Crimen y fuentes libres.

La segunda de las opciones, la opción que permite establecer si una decoración está animada, al activarse, muestra una nueva sección en la interfaz. En dicha sección se permite personalizar el comportamiento de dicha animación estableciendo: El tiempo de duración de cada fotograma en segundos, en Inglés *frame rate*, y una lista de fotogramas, en el orden en el que se deben producir para realizar la animación. Esta lista, si se mantiene vacía, producirá una secuencia de fotogramas por defecto, comenzando por el primero, y, al terminar, volviendo a éste. Este panel puede verse en la figura 5.22.

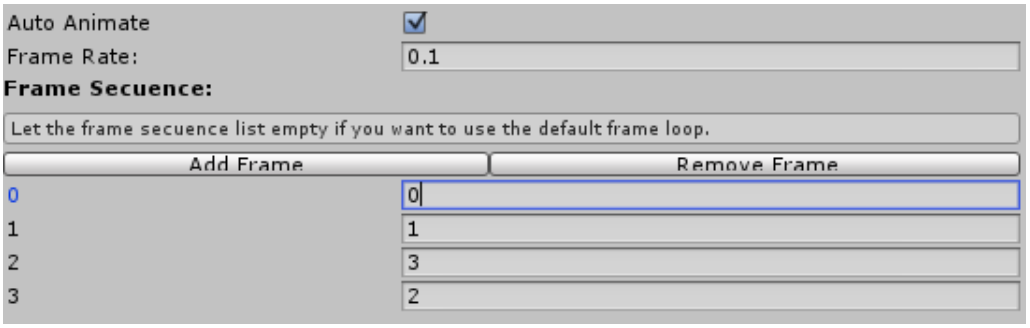

Figura 5.22 - Panel que se muestra al activar la opción de animado.

Este apartado de animado, hace que, en el momento de instanciar una decoración, se instancie con un componente adicional *AutoAnimator* que se encarga de realizar dicha animación.

Una vez se encuentre parametrizada y seleccionada la decoración que se desea colocar en el mapa, al colocar el cursor sobre el mismo, se muestra la Decoración Fantasma. Esta Decoración Fantasma se diseñó en el capítulo 4, y se ha implementado su funcionalidad por completo, estableciendo un comportamiento similar al de la Celda Fantasma. Esta decoración es especial, pues su representación es parcialmente transparente, y se elimina automáticamente cuando se abandona este editor, o se inicia el juego. La función de esta decoración es la de mostrar, antes de instanciar dicha decoración en el mapa, una representación del resultado que se va a obtener, y por consiguiente, facilitar la tarea de decorado al usuario.

Como mecánicas adicionales, los modos de posicionamiento de decoraciones: centrado en la celda, o centrado en la posición del cursor, son accesibles pulsando o no la tecla Mayús., en Inglés *Shift*, del teclado. Cuando dicha tecla no está pulsada, se produce un posicionamiento centrado en la cara de la celda donde se va a posicionar, y, cuando sí está pulsada, este posicionamiento se realiza centrando en la posición del cursor.

Finalmente, para remarcar exactamente la cara en la que se va a posicionar la decoración, se dibuja un borde blanco en la misma. Esta característica se muestra en cualquiera de los modos de posicionamiento de decoración.

Sección 5.4.2.4. Entidades

El módulo de entidades es el más simple de todos. Su funcionalidad no dispone de creación de entidades, pues es complejo de implementar y Unity ya provee herramientas suficientes para llevarlo a cabo.

Dentro de la parte de inspector, para realizar su funcionamiento simplemente contará con un campo en el que podremos seleccionar una entidad, siendo ésta la que se posiciona.

Dentro de la parte de escena, de una forma similar a como se hace en el módulo de pintado, seleccionando la cara superior de la celda podremos situar la entidad en ella.

De esta forma, facilitamos la capacidad para disponer las entidades en el mapa haciéndolo levemente más simple.

Para crear las entidades, el método más simple consiste en la creación de una decoración con la apariencia que se desea dar a la entidad, en la celda que se desee, seleccionarla con el cursor y añadirle la componente *Entity*.

Sección 5.5. La clase gestora Game

Durante la especificación del proyecto, se mencionó la necesidad de una clase gestora del funcionamiento del juego. Dicha clase, deberá mantener el bucle principal de la lógica del juego. Unity ya proporciona un bucle principal, sin embargo, la necesidad de *Game* radica en controlar dicho desarrollo, permitiendo la distribución de actualizaciones y eventos a las distintas partes que conforman el proyecto.

En primer lugar, *Game* en su inicio será el encargado de configurar la cámara en su perspectiva, de ocultar y desactivar los mapas marcados como tal y de centrar la cámara en el jugador. Por último, creará los gestores de eventos que se necesiten. Al finalizar, comenzará el bucle principal que pasará por las siguientes partes:

1. **Actualización del controlador**: Dado que la entrada de datos por parte de una persona está centralizada en el controlador, se le manda a éste que realice su actualización. Durante ésta, el controlador recolectará datos de entrada y los distribuirá entre los distintos oyentes. En próximos capítulos se detalla más el funcionamiento.
2. **Distribución de eventos a los manejadores**: Dado que las partes del proyecto se comunican a través de Eventos, se distribuirán los eventos generados desde el último *tick* a los distintos manejadores. Algunos manejadores son el de animaciones, el de secuencias y el de mapas.
3. **Notificación de actualización a los distintos gestores**: Tal y como en la distribución de eventos, iterativamente se avisa a todos los gestores que se está produciendo una actualización.

Dentro del bucle principal de Unity, al finalizar la actualización de Game descrita en el bucle anterior, se ejecutará varias veces la actualización de la GUI. Por ello, durante el método *OnGUI* de *Game* se notificará al *GUIManager* que realice su actualización y dibujado.

El orden y la creación de los manejadores puede ser expandido creando nuevas posibilidades.

Para la distribución de eventos, la clase *Game* mantendrá un acceso estático a la instancia principal de *Game*, que permitirá encolar los nuevos eventos ocurridos, para ser notificados en la próxima iteración. Por ello, todos los eventos serán almacenados en una cola dentro de *Game*, que será vaciada mediante la distribución de cada evento, a lo largo de cada iteración del juego.

Dado que una de las piezas clave de este proyecto es la comunicación por eventos, en el siguiente subcapítulo trataremos su implementación.

Sección 5.6. Implementando eventos del juego

Un evento del juego es la mínima expresión de comunicación en este proyecto. Garantiza que su distribución será conocida por todos los gestores y entidades del juego, permitiendo que cualquiera pueda generar una respuesta a dicho evento. No necesariamente todas las acciones deben ocurrir a través de eventos, pero utilizarlos garantiza el desacoplamiento de las distintas clases.
Para crear un evento, debido a su necesidad de poder ser almacenado en caso de pausa del juego, necesitaremos que sea serializable. Como ya se comentó anteriormente, dado que no se trata de una componente la clase adecuada para crear un evento será *ScriptableObject*.

Dentro de los eventos se almacena un identificador o nombre que sirva como filtro principal para los encargados de manejarlos y un mapa de claves y valores con los distintos atributos que caracterizan el evento. Para su implementación, un mapa hash es suficiente. Sin embargo, Unity no garantiza la serialización de mapas, por lo que se mantienen dos listas con las claves y valores, para poder ser reconstruido el mapa en el momento de su deserialización. Por otro lado, se fuerza a almacenar nombres y claves en minúsculas, pese a que se reciban en cualquier formato, evitando así posibles errores en el futuro a los usuarios de la herramienta.

Uno de los problemas de serialización mencionados en el análisis reside en que Unity no provee soporte a la serialización de *System Object*. Solamente serializa *Unity Object* y los tipos básicos. Sin embargo, para almacenar en el mapa de claves y valores ambos tipos simultáneamente, es necesario utilizar *System Object*. Por ello, se ha creado un envoltorio que cataloga los tipos serializables y los almacena en las variables adecuadas. Dicho envoltorio almacena una variable para cada tipo básico. Cada vez que se escribe un valor en ella, vacía todas las variables excepto la nueva y guarda el nombre del tipo para poder acceder a él.

Pese a que, aparentemente, los eventos trabajan con *System Object*, realizan esta transformación internamente y se han dado casos aislados en los que los envoltorios no se han serializado correctamente por lo que, en revisiones futuras, será conveniente mejorar el sistema.

Sección 5.7. Implementando las entidades

Se entiende por entidad todo aquel agente perteneciente a un mapa, capaz de ser reactivo a eventos, ser actualizable y desarrollar acciones a lo largo del tiempo. Para implementar su apariencia, se podrán utilizar decoraciones como base para su renderización.

La entidad entonces, es una componente añadida a un objeto perteneciente a un mapa, que permite que éste la identifique como tal, y permita el flujo de información del juego hacia ella. Una vez identificada su característica de componente, conocemos que Unity

provee la clase *MonoBehaviour* para tal fin. Al ser una componente, durante la edición del mapa se encargará de mantener las entidades centradas en sus celdas y de posicionarlas en la cara superior de éstas. Fuera del modo de editor, este soporte no es forzado, por lo que se permiten los movimientos. Para identificar esto, Unity provee la clase estática *Application* que mantiene información sobre la ejecución actual y sus características.

La entidad posee unas características que la personalizan, a saber, nombre, textura para su rostro, su posición en el mapa y su capacidad para bloquear a otras entidades. Esta última se ha implementado de una manera simple, indicando si debe bloquear o no el paso a las entidades. En futuras revisiones, debería ampliarse el sistema para filtrar los tipos de entidades que permitiría que la atravesaran.

Durante la especificación, se establecieron personalidades o comportamientos, en la fase de diseño, se materializaron como componentes, y en su implementación, se han desarrollado como *EntityScript*. Cada *EntityScript* es capaz de recibir eventos y actualizaciones del juego y proponer acciones de interacción con ella. Para ello, la clase *Entity* es la puerta de entrada desde el mapa, propagando todos los eventos y actualizaciones y agrupando toda la información de acciones de cada *EntityScript* que la conforme. De esta forma, con añadir nuevas componentes *EntityScript* automáticamente éstas están vinculadas al mapa y se sumarán al comportamiento de la entidad.

El *EntityScript*, dada su naturaleza, es una componente en su más pura definición y, por ello, hereda de *MonoBehaviour*. Dentro de su inicialización, buscará la *Entity* y la pondrá a disposición como variable protegida al heredero, disminuyendo así los tiempos de acceso a las características de la entidad.

Dentro de los métodos de cada *EntityScript* destacan cuatro: *EventHappened* para la recepción de eventos, *Tick* para las actualizaciones de *Game*, *Update*, para las actualizaciones temporales y *GetOptions*, para la comunicación de interacciones posibles con él. En los siguientes subcapítulos se expondrán los distintos *EntityScript* implementados destacando estos tres métodos.

Sección 5.7.1. La componente jugador, *Player*

De entre todas las entidades, *player* destaca por ser aquella entidad controlable por el usuario, que le representa y responde a sus interacciones. Por ello, esta entidad cuenta con un método más, el método que recibe los eventos del controlador del juego.

Durante dicho método *player* se encarga de interpretar la información recibida por el controlador de la siguiente forma:

1. Si existe información de acciones:
    a. Si es sólo una, y ésta requiere que me mueva, se genera un evento *Move*, de la entidad que contiene a *Player*, hacia la celda donde se encuentra la

entidad que generó la acción, a la distancia que él provea e indicando que se desea la respuesta de finalización del evento, para garantizar lanzar el evento contenido por la acción en el momento apropiado. De no requerir movimiento, se lanza instantáneamente.

   b. De ser múltiples acciones se crea una nueva *OptionsGUI* con la información de las distintas acciones y se da de alta en el *GUIManager*. Cuando *OptionsGUI* selecciona una correctamente, devuelve el evento de controlador con una única acción, y es procesada en el anterior caso.

2. Si existe información de celda se generará un evento *Move* hacia la celda indicada de la entidad que contiene el *player*, y se encolará.
3. Si hay datos sobre las teclas arriba, abajo, izquierda o derecha se genera un evento *Move* hacia la celda vecina que corresponda, accediendo a éstas a través de la información que el mapa provee.

En su método *EventHappened* analiza si ha ocurrido *EventFinished* del evento *Move* y durante el *Tick* se encargará de encolar el evento contenido en la acción tras haber finalizado *Move*. Antes de encolar el evento, añade el parámetro *executer* al evento, para que el receptor pueda identificar quién ejecutó la acción, permitiendo respuestas dinámicas.

Sección 5.7.2. La componente movedora, *Mover*

Mover es también un caso especial de entidad. Dado que aparece prácticamente en todas las entidades, no es necesario añadirla, pues *entity*, de no detectarla, la crearía y configuraría automáticamente.

Durante el método *EventHappened*, mover es susceptible a dos eventos, *Move* y *Teleport*. El segundo, al ser más prioritario, cancelará cualquier movimiento actual. Para realizarlos, dado que no podemos conocer el orden de llegada pero el segundo es más prioritario, se almacenan localmente y se procesan al llegar la actualización *Tick*.

Durante el método *Tick* de deberse teletransportar, si lo hace a una celda que esté en el mismo mapa, lo hará instantáneamente, en el caso de estar en otro mapa se pondrá visible o invisible en función del estado de mapa al que se dirija. De ser el *player*, además, marcará como activo y visible el mapa de destino. En el caso de tener que moverse, se solicitará al planificador de rutas una ruta, y de encontrarse, se comenzará a procesar dicho movimiento en *Update*.

Durante el método *Update* se solicitan las celdas al planificador de rutas y se definen los movimientos a lo largo del tiempo. Para ello existen dos tipos de movimientos: lineales y parabólicos. Los primeros, están orientados a movimientos a celdas de la misma altura o hacia rampas. Los segundos, están orientados a movimientos a celdas de distintas alturas, simbolizando forma de salto.

Para implementar esto, la clase *mover* cuenta con una clase abstracta interna llamada *Movement* que almacena información del origen y destino y permite acceder a la posición actual en función de un valor entre 0 y 1, que indica el progreso del movimiento. Es clase abstracta ya que el método que proporciona la posición actual es abstracto y depende de cada implementación. En el caso lineal, se devolverá un valor proporcional a la distancia recorrida en línea recta, pero en el caso parabólico, se utilizará una función especial que dibuja dicha forma entre ambos puntos. Para generar las distintas implementaciones, *Movement* cuenta con un método estático al cual se le especifica el tipo de movimiento a generar y autoconfigura el movimiento.

Por último, de detectarse la componente decoración en la entidad y tener ésta 3 columnas y 4 filas, se identificará como un *tileset* de animación de movimiento. Durante el movimiento en este caso, se muestra la animación intercambiando el fotograma activo.

Sección 5.7.3. La componente habladora, *Talker*

La capacidad de interactuar con una entidad nace de *Talker*. *Talker*, no sólo proporciona la capacidad de hablar y mostrar diálogos, sino que permite el desarrollo de secuencias completas. Para ello, sus métodos se han descrito de la siguiente forma.

Durante el método *GetOptions*, *Talker* devolverá un *array* con una acción, en cuyo interior se indica, que la entidad deberá acercarse a una distancia de 1 casilla para interactuar y que el evento es *talk* y cuenta con el parámetro *talker* que le contiene para posteriormente identificarse al recibirlo.

Durante el método *EventHappened*, se esperará a la recepción de *talk* y, al recibirlo, se guardará en una variable que en el próximo *Tick* se debe iniciar la secuencia almacenada.

Durante el método *Tick*, de haber recibido el evento, se crea un nuevo evento *start secuence* indicando como atributo *secuence* la secuencia almacenada. Dicho evento, es posteriormente capturado por el intérprete de secuencias iniciando así la interpretación.

La entidad *talker* cuenta con un Editor especial, capaz de abrir el editor de secuencias con la secuencia actual como parámetro. Dicho editor cuenta con dos botones para abrir y cerrarlo. Esto se muestra en la figura 5.23.

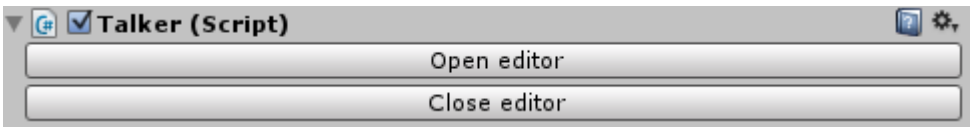

Figura 5.23 - Vista del inspector de Talker.

Sección 5.7.4. La componente almacén, *Inventory*

Para implementar el almacenamiento de objetos en una Entidad, *Inventory* cuenta con una lista de objetos y es capaz de responder a distintos eventos para añadirlos, quitarlos o utilizarlos.

Durante su *GetOptions*, se buscará una componente *player* en la misma entidad y de encontrarlo, genera una acción con el evento *open inventory*, que no requerirá moverse para ejecutarla.

Durante su *EventHappened*, interpreta los eventos *add item*, *use item* y *remove item*. Dado que el orden de llegada puede ser no homogéneo, se almacenan los eventos en listas para ser tratados a posteriori durante *Tick*. Por otro lado, de recibir el evento *open iventory* se crea una *InventoryGUI* con el inventario como argumento y se da de alta en el *GUIManager*.

Durante su *Tick*, todos los eventos recibidos conllevan operaciones sobre la lista. El orden de interpretación realiza primero las adiciones, después los usos y por último las eliminaciones. Dado que las secuencias pueden requerir del aviso síncrono para garantizar la correcta interpretación, al interpretar los eventos, se solicita a *Game* que envíe el evento *event finished* sobre cada uno de ellos.

Para implementar los objetos, se ha implementado una clase abstracta *Item* que proporciona una interfaz genérica para el tratamiento y utilización de los objetos, permitiendo a su vez su serialización automatizada. Los objetos cuentan con un nombre, una imagen que les represente y una representación en forma de *prefab* para poder instanciarse en el mapa. Cada objeto debe implementar el método *Use* que se ejecuta al ser utilizado y el método *IsEqualThan* que permite comparar los objetos entre sí, para poder agruparlos en posibles representaciones, o garantizar que no se repitan objetos catalogados como únicos.

Sección 5.7.5. La componente objeto, *ItemScript*

Dado que los objetos como tal, no son entidades, es necesario que para su representación en el mapa y su interacción con el mundo, se cree un *EntityScript* que haga de envoltorio de éstos. Para ello, la clase *ItemScript* permite que los objetos puedan ser recogidos del suelo y tengan una representación.

Durante su *GetOptions*, devuelve una acción con un evento *pick*, que se ejecuta a la distancia de una casilla.

Durante su *EventHappened*, si se recibe un evento *pick*, se comprueba que el objeto es él mismo a través del parámetro *item* y, se busca un ejecutor. En unión a lo implementado en la componente jugador, al ejecutar un evento, el jugador añadirá al evento el parámetro *executer*. Una vez recibido el evento en el inventario, el parámetro *executer* realiza un

enlace para la búsqueda del inventario utilizando *GetComponent*. Si se encuentra, se almacena el inventario para ser procesado durante el próximo *tick*. Se realizará un control de los eventos *event finished.* Al añadir el objeto al inventario, se verifica si dicho *event finished* ha ocurrido antes de eliminar la entidad del mapa, garantizando el traspaso del objeto del mapa a la entidad.

Durante su *Tick*, si se encuentra el inventario, se crea un evento *add item* colocando como parámetros el objeto en *item* y el inventario en *inventory*. Por otra parte, durante su *Update*, si se recibió el evento event finished correctamente, se solicita a Unity que destruya el *GameObject* que le contiene, eliminando así el envoltorio del objeto del mapa.

Sección 5.7.6. La componente movimiento aleatorio, *RandomMover*

Una de las formas más comunes de dar vida a una entidad es la de dar la sensación de autonomía al moverse hacia algún lugar cada cierto tiempo. Para realizar esto, si a una entidad se añade *RandomMover*, ésta puede generar aleatoriamente algún movimiento en cualquier dirección.

Para ello, solamente se hace uso del *Tick*, durante el cual, utilizando una variable de probabilidad y la variable *Time.deltaTime*, para conocer el tiempo desde la última ejecución, realizaremos un producto para calcular la probabilidad de moverse en ese instante. Dicha probabilidad se prueba generando un número aleatorio entre 0 y 1, que de ser menor que el número calculado, entra dentro de la probabilidad. Al darse como cierto, se genera un evento *move* a una de las cuatro celdas adyacentes seleccionándolas una vez más utilizando un número aleatorio.

Sección 5.7.7. La componente teletransportadora, *Teleporter*

La forma de comunicar varios mapas consiste en la teletransportación de entidades entre los mapas. Para ello la componente *Teleporter* se encargará de teletransportar una entidad siempre que esté activado. Para su activación, podrá programarse siempre activo, activación al recibir un evento o utilizar las mismas bifurcaciones que las especificadas para las escenas.

Durante su *GetOptions*, creará una acción con un evento *teleport* que se ejecutará a distancia 0, para provocar que la entidad se desplace hasta él, pero sin indicar el evento.

Durante su *Tick*, solicitará a su celda que le proporcione las entidades sobre ella. Iterará sobre ellas y de detectarse cualquier entidad que no sea el propio *teleporter*, se creará un evento *teleport* con la entidad y la celda de destino como argumento. Sin embargo, esto sólo se producirá si el *teleporter* está activo. Para detectarlo, si se recibiera el evento disparador o *trigger* durante *EventHappened* se marcaría como activado, o en el caso de

haber seleccionado como método de activación una bifurcación, se ejecutará la bifurcación y se teletransportará a la entidad en caso de ser cierta.

Para su edición, el *teleporter* cuenta con un Editor especial, que permite seleccionar la celda de destino que estará vinculada con el *teleporter*, así como indicar el método de activación.

El modelo más simple para implementar esto fue tomar el código del editor de secuencias, y replicar la generación de editor de eventos y de editor de bifurcaciones en esta parte (se darán detalles de la implementación más adelante). Para seleccionar el tipo, se utilizará *Toolbar* de *GUILayout*. Esto se observa en la figura 5.24.

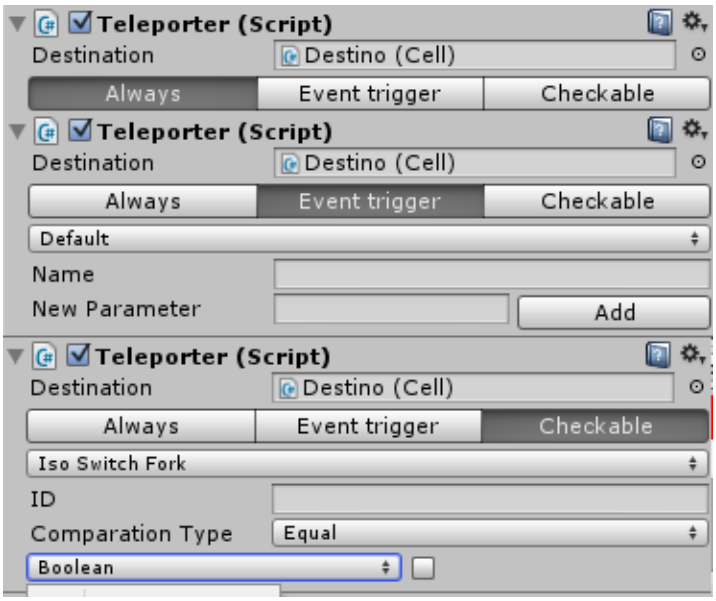

Figura 5.24 - Distintas opciones del teletransportador en la vista del inspector.

Sección 5.8. Implementando los manejadores de eventos

Dado que no sólo las entidades funcionarán con eventos, existirán sistemas a un nivel mayor capaces de manejar eventos. Durante el capítulo de *Game* se explicó que serían capaces de recibir eventos y actualizaciones de *Game*.

Por ello, los manejadores de eventos implementarán la interfaz *EventManager*, que permitirá ambas funcionalidades. Un manejador en sí, es prácticamente un módulo para *Game*, un sistema que podría crear cualquier tipo de mecánica introducido en el *Game* a través de la comunicación por eventos.

El manejador más importante, que se implementó en primer lugar, es el manejador de mapas. Dicho manejador se encarga de realizar un puente entre *Game* y *MapManager* para realizar la distribución de eventos y actualizaciones.

El segundo manejador surge de la necesidad de crear el intérprete para las secuencias. Dado que, una secuencia no es una entidad, y durante su ejecución *Game* se comportará de otra forma, el manejador de secuencias interpretará los eventos *start secuence* creando los intérpretes para las secuencias, el resto de eventos que reciba, si tuviera una secuencia iniciada, los remitiría a ella. Más adelante se explicará la implementación de la interpretación de las secuencias. Una de sus características, es que bloqueará el controlador del juego hasta la finalización de la secuencia. Para esto, tomará el delegado del controlador, dejándolo vacío provocando que todos los objetos que estuvieran registrados a las actualizaciones de controlador dejaran de recibirlas. Para realizar su interpretación, irá realizándola paso por paso durante método de actualización. Al finalizar se restablecerá el contenido del delegado.

El tercer manejador existe para gestionar la creación de emociones. Durante los capítulos anteriores apenas se mencionaron estas, pero viendo la sencillez de su implementación y la riqueza que agregaban al juego se decidió implementarlas. Para ello, el manejador interpretará el evento *show emotion* que contendrá, la entidad sobre la que se instanciará la emoción, y el *prefab* de la emoción a crear. En el momento que se detecte que la emoción se ha destruido, se enviará el *event finished* que avisará al manejador de secuencias que puede continuar con la ejecución.

El último manejador será el encargado de recibir los eventos de cambio de variables globales llamadas *IsoSwitches* y materializarlos en el archivo físico almacenado en la carpeta de recursos del proyecto. Será necesario que esté almacenado en dicha carpeta para poder ser accedido en tiempo de ejecución. Para realizar el cambio, utilizando *Resources.Load* lo cargará en memoria y tras esto realizará el cambio con los métodos proporcionados por la clase *IsoSwitches*.

Sección 5.9. Implementando el controlador

Para implementar la unificación de los controles mencionada en el subcapítulo 4.4 se decidió crear una clase estática muy vinculada al ciclo de vida de la clase *Game*. Durante su bucle principal, *Game* hará llamadas explícitas para provocar eventos de controlador. Un evento de controlador no es igual que un evento de juego, pues cuenta con una estructura prefijada en la que se almacenará el estado de las entradas por parte del jugador hacia el juego.

Dicha clase estática realizará el bucle principal descrito en el diseño. Para su implementación describiremos punto por punto los pasos diseñados de la siguiente forma.

El primer paso consistirá en crear un objeto de la clase *ControllerEventArgs* en el que almacenaremos todos los valores del ratón y teclado recibidos de la clase Input de Unity. Para asegurar que recibiremos la posición del ratón pese a estarse utilizando controles de ratón, pondremos a cierta la variable *Input.simulateMouseWithTouches*.

Tras esto, solicitaremos a *GUIManager* que compruebe si alguna interfaz capturara el evento de controlador. Para ello, todas las interfaces contarán con un método *CaptureEvent* al que se le proporcionará la variable con el estado del controlador para que la interfaz pueda identificar si debe capturarlo o no, por ejemplo, si se encuentra el ratón en su espacio.

Si alguna interfaz lo capturara, se le solicitaría que lo rellenara utilizando *FillControllerEvent* y pasándole como argumento la variable anterior con el estado de la entrada del usuario. Si cualquiera interfaz quisiera controlar el lanzamiento final del evento a los objetos que se encontraran escuchando al controlador, con la variable *send* de los *ControllerEventArgs* podrá modificar este comportamiento.

Por otro lado, si ninguna interfaz lo capturara, se le solicitaría a *MapManager* que rellenara el evento. *MapManager*, en este punto, si el evento de controlador indicara que se acaba de pulsar el botón izquierdo lanzaría un rayo utilizando las físicas de Unity para conocer la entidad o celda que se está tocando. Es necesario este proceso, pues el ratón se encuentra sobre la pantalla en coordenadas bidimensionales y el mapa se encuentra en coordenadas tridimensionales. De encontrarse una entidad, se le solicitarán las acciones con el método *GetOptions* y se rellenarán los campos *entity*, *cell* y *actions* del evento de controlador. De no encontrarse una entidad, pero sí una celda, se rellenará simplemente el campo *cell*. Si se diera cualquiera de estos casos, se marcará el evento para enviar.

Finalmente el controlador contará con un delegado público al que podrá registrarse cualquier objeto indicando el método que se llamará. Si se hubiera marcado el evento para enviar, se utilizará el delegado para avisar a todos aquellos interesados.

Sección 5.10. Implementando las interfaces

Como se describió en el capítulo de diseño para el desarrollo de las interfaces utilizaremos la clase GUI y *GUILayout*. Todas las interfaces serán gestionadas por un gestor de interfaces que se encargue de realizar las funciones descritas en el subcapítulo 4.3.

El gestor de interfaces cuenta con un mapa hash que almacena las interfaces activas y su prioridad. Basándonos en este mapa, se implementó una clase comparadora que permitía ordenar las interfaces para su tratado en función de la prioridad. Para darlas de alta, de encontrarse cualquier método iterativo se impedirá, almacenando las nuevas en una lista temporal para su posterior agregación.

Para el correcto funcionamiento con el controlador, cuenta con un método que busca por orden de prioridad la interfaz que captura el evento. Sin embargo, para su dibujado, y dado que el funcionamiento de *GUI.depth* es distinto al esperado, se reordenarán las interfaces para que las menos prioritarias sean dibujadas en último lugar, apareciendo así sobre las demás.

Dado que será una clase estática, las interfaces se registrarán mediante la llamada a *addGUI* y se darán de baja mediante *removeGUI*. Para lograr el funcionamiento uniforme del bucle, todas las interfaces que se utilicen deberán de heredar de *IsoGUI* que contiene los métodos *CaptureEvent*, para conocer si capturará el evento y *Draw* para solicitar el dibujado.

Sección 5.10.1. Interfaz de diálogos

Para implementar la interfaz de diálogos, en su constructor recibiremos el fragmento del diálogo a representar. En el fragmento van contenidos el nombre, la imagen y el texto a mostrar.

Durante el método *draw* se dibuja la interfaz haciendo uso de GUI.*Box* para delimitar el rectángulo del área del diálogo. En el interior de este rectángulo dibujaremos otra GUI.*Box* modificando la imagen de fondo del estilo para que utilice la proporcionada en el fragmento. Para dibujar el texto se utilizan dos GUI.*Label*, una para el nombre del emisor y otra para el mensaje.

Por defecto, a menos que se sobrescriba, toda *IsoGUI* capturará el evento de controlador. Durante el método *FillControllerEvent* que es llamado tras verificar que se captura el evento, se mira si el evento de controlador indica que se acaba de pulsar el botón izquierdo, y de ser así, se crea un evento *ended fragment* con el parámetro *launcher* que se obtiene en el momento de creación de la GUI se indicará quien creó la GUI. Tras encolar el evento, se solicita a *GUIManager* que destruya la GUI. Esta interfaz se observa en la figura 5.25.

Figura 5.25 - Interfaz de diálogos para un fragmento.

El texto de la figura 5.25 y títulos será proporcional a orientaciones y resoluciones de la pantalla.

Por otro lado, la GUI de diálogos cuenta con otro constructor para opciones. Para ello, se recibirá en el constructor un *array* de *DialogOption*. Durante el método *draw*, en este caso se dividirá la altura de la pantalla entre el número de opciones y se crearán tantos GUI.*Button* como opciones existan abarcando toda la pantalla de forma vertical. De detectarse la pulsación, se generará un evento *chosen option* indicando en el parámetro *option* el índice de la opción seleccionada. Además como en el caso anterior, incluiremos el parámetro *launcher*. Esta interfaz se ve en la figura 5.26.

Figura 5.26 - Interfaz de selección de opciones.

De esta forma, el intérprete de diálogos solamente tendrá que crear la GUI de diálogos, añadirla a *GUIManager* y esperar a recibir los eventos para continuar con la interpretación.

Sección 5.10.2. Interfaz de inventario

La interfaz de inventario permitirá al jugador inspeccionar el inventario y realizar acciones sobre sus objetos. Para ello, en su constructor deberá recibir el inventario a representar.

Para representarlo, en primer lugar se creará una GUI.*Box* que oscurecerá la pantalla entera utilizando como referencia los valores del tamaño de la pantalla proporcionados en *Screen.width/height*. A continuación, con una GUI.*Label* se pondrá el título. En la parte inferior utilizando GUI.*Button* dibujaremos un gran botón que permita cerrar la interfaz.

Para representar los objetos, en primer lugar unificaremos los objetos iguales del inventario. Utilizaremos el método *IsEqualThan* del objeto para conocer si son iguales. Si fueran iguales, almacenaríamos en un contador cuantos objetos como él existen. Tras ello utilizando *BeginScrollView* crearemos un espacio con barra de desplazamiento vertical en el que comenzaremos a dibujar los *item* utilizando diversas GUI.*Box* para fondos y la imagen y GUI.*Label* para el nombre y la descripción. Si existiera más de un objeto de ese tipo, se dibujaría un contador con un *Label* en el interior de una *Box*.

Para lograr que se puedan realizar acciones sobre los objetos sobre cada uno y basándonos en el rectángulo que abarcan crearemos un botón invisible utilizando GUI.*Button* y la propiedad de estilo *GUIStyle.none*. De esta forma, al pulsar sobre cada objeto podremos detectarlo y en ese momento crear una interfaz de selección de acciones capaz de representar las dos acciones posibles. En su interior, las acciones almacenarán un evento *use item* para utilizarlo y otro *remove item* para tirarlo.

La representación de la interfaz y del comportamiento que se realiza al interactuar con la misma se muestra en las figuras 5.27 y 5.28.

Figura 5.27 - Interfaz de inventario.

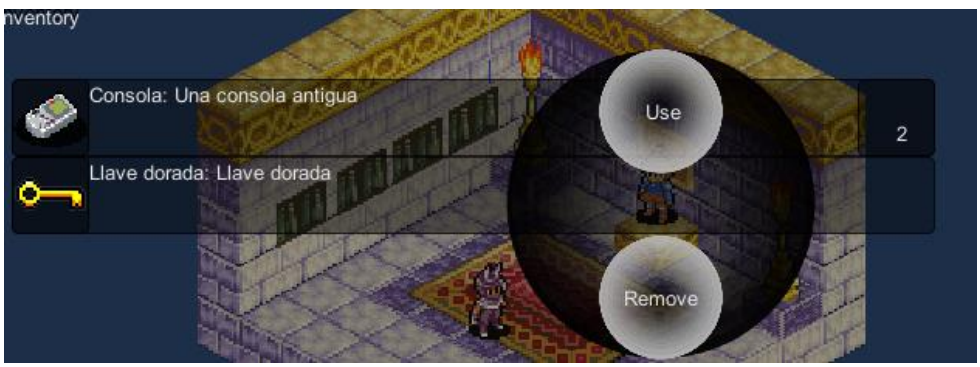

Figura 5.28 - Al pulsar sobre el objeto podremos realizar acciones sobre él.

Gracias a su representación permite un control táctil amigable y sencillo para el usuario.

Sección 5.10.3. Interfaz de selección de acciones

Se ha mencionado en la componente *player* que ésta, al detectar múltiples acciones del controlador deberá crear una interfaz para que el usuario decida sobre la opción que desea elegir. También, esta misma mecánica será utilizada por la interfaz del inventario para realizar acciones sobre los objetos. Dado el entorno táctil del proyecto, para facilitar la selección de la acción se realizó un diseño circular en el cual las opciones se sitúan en forma radial automáticamente distribuida en función de la cantidad de acciones.

Para implementarlo, dado que GUI no provee la capacidad para dibujar círculos, se crearon varias texturas circulares que simbolizarían el fondo y las distintas acciones a elegir. Para crear la GUI, se pasarán como parámetro la lista de acciones y el punto del mundo donde se originan.

Para dibujar el menú, se dividirán los 360º de la circunferencia entre el número de opciones. Tras esto, se realizarán los cálculos para conocer si el espacio es suficiente para dibujar las opciones, y, de no ser suficiente se expandirá el radio hasta poder contenerlas sin que se superpongan. Una vez calculado el radio, se recorrerá la circunferencia utilizando la porción de ángulo para cada opción, calculando la altura y la anchura donde colocar los botones utilizando las funciones seno y coseno. Finalmente para conocer si una acción ha sido seleccionada, si se levanta el ratón en el interior de su rectángulo se almacenará como seleccionada. Durante el próximo *FillControllerEvent*, se modificarán los atributos del *controller event* indicando la acción seleccionada como la lista de acciones, y poniendo la variable "*send"* del evento a verdadero el controlador la enviará a sus delegados.

La figura 5.29 muestra la interfaz de selección de acciones con varias opciones.

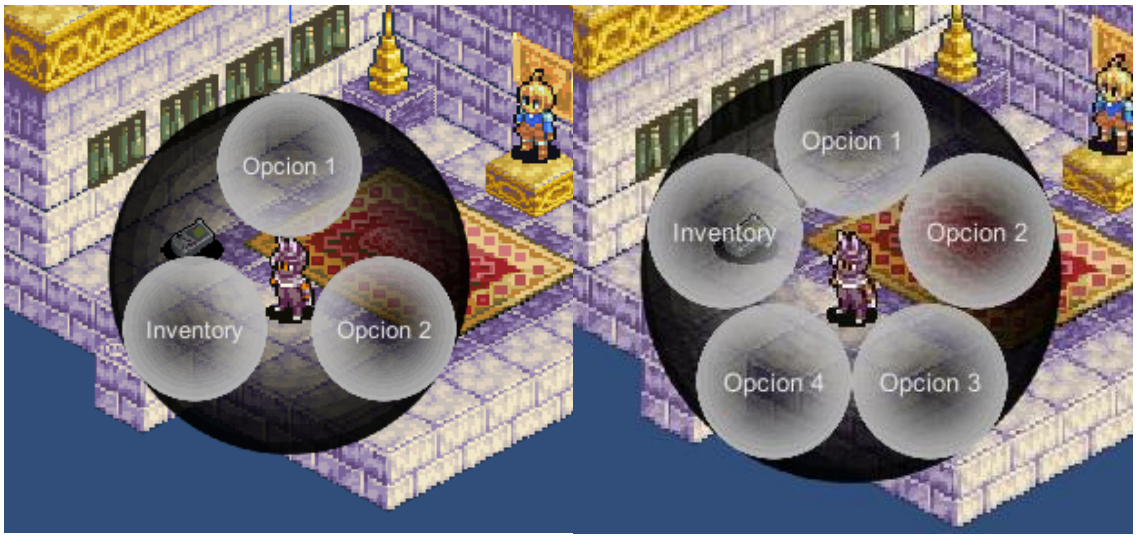

Figura 5.29 - A la izquierda interfaz con 3 acciones, a la derecha con 5 acciones.

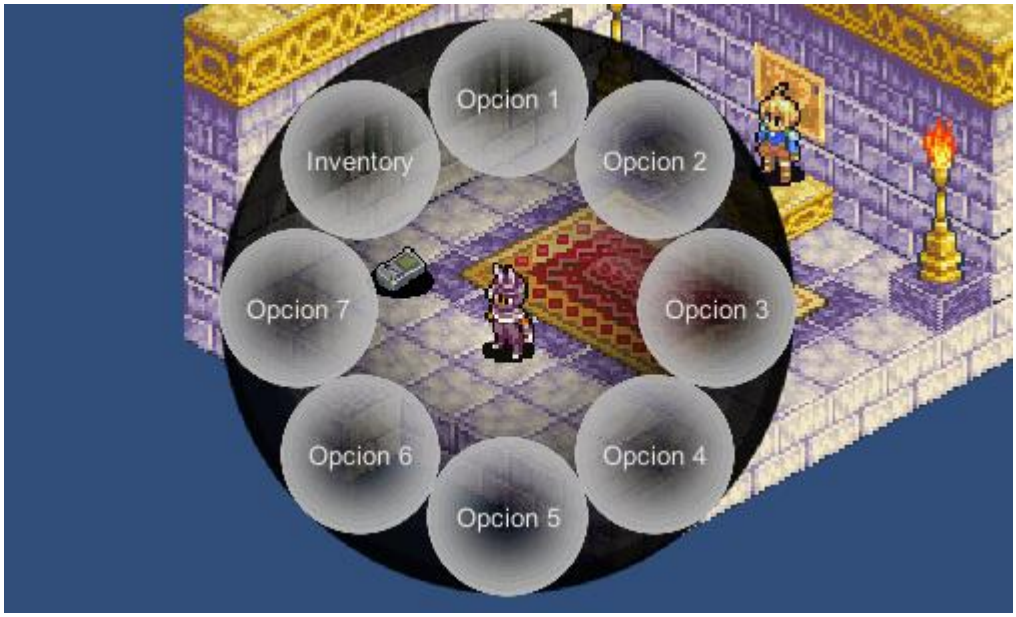

Figura 5.30 - Interfaz de selección de acciones con múltiples acciones.

Como se puede observar en las figuras 5.29 y 5.30, el menú de opciones crecerá en función de las acciones que reciba en su constructor.

En este caso, la interfaz de selección de acciones sí re implementa *CaptureEvent*, pues solamente lo capturará si el ratón se encuentra en el interior de la circunferencia. Si en ese punto se detectara el ratón fuera, se marcaría la GUI para destruir y al inicio de su próximo dibujado se solicitaría a *GUIManager* que la destruya.

Sección 5.10.4. Interfaz de controles en pantalla

La interfaz para controles en pantalla facilitará al usuario controlar el juego simulando un teclado para entornos táctiles. Dicha interfaz, aprovechará el método *FillControllerEvent* para reconfigurarlo indicando los botones que se han pulsado. En principio, la implementación solamente abarcará los cuatro botones de movimiento para ejemplificar la creación de dicha interfaz.

Para ello, en su método *draw* se dibujarán cuatro botones utilizando para ello una técnica distinta a la utilizada por GUI.*Button*. Dicha técnica se basa en el uso del método *draw* de un estilo *GUIStyle*. Tomando como referencia un estilo personalizado se realizará la llamada a este método. Es necesaria la utilización de este método para poder marcar como activos los botones cuando se pulsen los botones de teclado correspondientes. Por ejemplo, al pulsar la tecla hacia arriba se iluminará el botón pese a no estar siendo tocado físicamente. Durante este método, además, almacenaremos en variables si se ha pulsado cualquiera de los botones.

Durante el método *CaptureEvent*, si se hubiera pulsado cualquiera de los botones táctiles se retornaría que sí, para capturar el evento y recibir posteriormente la llamada de

FillControllerEvent. Además, aprovecharemos este momento para almacenar el estado inicial de los botones del controlador y poder pintarlos en el próximo dibujado.

En el método *FillControllerEvent* se sobrescribirá el valor de los botones de teclado que no estuvieran ya pulsados con aquellos que hayan sido pulsados durante el dibujado.

El resultado final puede observarse en la figura 5.31.

Figura 5.31 - Interfaz de controles en pantalla provocando el movimiento en el jugador.

Sección 5.11. Implementando las secuencias

Para implementar las secuencias serán necesarias varias piezas encargadas de distribuir el almacenamiento, interpretación y edición. Para almacenarlas, se implementó una estructura en forma de árbol en la que la clase secuencia almacenaría el nodo inicial. En cada nodo tendremos una variable para almacenar el contenido del nodo y un listado de nodos hijos al nodo actual.

Dentro de los posibles contenidos que se especificaron y diseñaron, contamos con la posibilidad de almacenar eventos, diálogos y bifurcaciones. La primera es trivial, pues en el contenido se almacenará el evento en cuestión.

Para la segunda se implementó una clase diálogo que hereda de *ScriptableObject* para poder serializarla automáticamente capaz de almacenar en su interior una lista de fragmentos de diálogo y una lista de opciones. Cada fragmento podrá contener en su interior una referencia a una entidad, un nombre y un texto y de forma opcional una imagen. Las opciones por su parte contendrán un texto.

Para los últimos se implementó una clase abstracta *Checkable* que hereda de *ScriptableObject* y que contiene un método comprobar. Durante dicho método deberán realizar las comprobaciones y se retornará un valor binario. Dentro de las clases que implementan *Checkable* encontramos las mencionadas en el apartado de análisis y diseño.

Para implementar las bifurcaciones sobre variables globales se almacenará una variable para conocer el nombre de la variable global a comprobar, un tipo de comparación definido por la clase *ComparationType*, y un valor con el que comparar. Este último, utilizará la misma clase que utilizan los eventos para envolver los múltiples tipos básicos, limitando la funcionalidad para almacenar solamente tipos básicos y excluir los *UnityObject*, se utiliza la clase *IsoSwitchesManager*, que hace las funciones de singleton para el acceso al recurso donde se encuentran almacenados los *IsoSwitches*. Una vez accedido a *IsoSwitches* se accederá al valor de la variable y se comparará utilizando una distinción de casos en función del tipo de comparación seleccionado.

Para implementar las comprobaciones sobre los objetos que contiene el personaje, se almacenará una referencia al objeto a buscar y otra referencia al inventario donde buscarlo. Además se incluirá una variable binaria para indicar si se considerará cierto encontrar el objeto o no encontrarlo. Utilizando la variable de objetos del inventario y el método *IsEqualThan* de la clase *Item* se realizará la búsqueda de forma iterativa.

Dado que se anotó como requisito futuro, finalmente no se implementaron las comprobaciones sobre el tiempo. Sin embargo su especificación quedará como parte de esta memoria para poder servir de base para futuras ampliaciones del proyecto.
Una vez cerradas las implementaciones, el siguiente elemento necesario será el intérprete.

Sección 5.11.1. Intérprete de secuencias

El intérprete de secuencias es uno de los sistemas más complejos dentro de la aplicación.

En primer lugar, la puerta de entrada para el inicio de las secuencias comenzará en el manejador de eventos llamado *SecuenceManager* o manejador de secuencias. Dicho manejador, como se ha mencionado en el apartado de manejadores, será el encargado de recibir el evento de inicio de secuencia y crear el intérprete para interpretarla. Para más información, consultar el subcapítulo 5.9.

Una vez creado un intérprete, el manejador le transmitirá todos los eventos y actualizaciones. El intérprete realizará el siguiente proceso para cada actualización. En primer lugar, comprobará que la secuencia no haya terminado. Para ello, si no tiene nodo actual, o el contenido del nodo actual es vacío se considerará por terminada. A continuación se creará un subintérprete para el contenido actual de la forma descrita más adelante. Dado que el subintérprete puede requerir varias actualizaciones o eventos para su interpretación, dicha creación sólo se producirá si no hay subintérprete asignado. Tras ello, se propagará la actualización al subintérprete para que realice su labor. Finalmente,

tras su actualización se le preguntará con el método *HasFinishedInterpretation* si ha finalizado su interpretación y de ser así se solicitará al subintérprete el próximo nodo a interpretar con el método *NextNode*. Gracias a este hecho, si el subintérprete interpretara bifurcaciones u otras mecánicas podría retornar uno u otro hijo de forma transparente al intérprete. Tras adquirir el siguiente nodo se destruirá el subintérprete. Dado que algunos subintérpretes pueden ser *UnityObject*, se tratarán de destruir.

Por otro lado, el manejador de eventos transmitirá los eventos al intérprete, quien los transmitirá a su vez al subintérprete activo.

Para la creación del subintérprete adecuado para el contenido, existirá una factoría singleton con un método *createSecuenceInterpreterFor* en el cual se facilitará el nodo a interpretar. La factoría contará con un prototipo de cada subintérprete que indicará si puede interpretarlo a través del método *CanHandle*. Una vez encontrado el prototipo adecuado, se creará una copia utilizando *Clone*.

Cada subintérprete deberá implementa la interfaz *ISecuenceInterpreter* que unifica todos los métodos anteriormente mencionados para su utilización.

Dentro de los subintérpretes contaremos con tres tipos ordenados por su complejidad, el intérprete de bifurcaciones, el intérprete de eventos y el intérprete de diálogos.

El primero y más sencillo, el intérprete de bifurcaciones, simplemente ejecutará el método *Check* de la bifurcación para conocer el hijo al que acceder, y en el método *NextNode* retornará el hijo adecuado.

El segundo y ligeramente más complejo, el intérprete de eventos, encolará el evento en *Game* tan pronto reciba su primera actualización. Sin embargo, cuenta con la complejidad extra de que deberá controlar los eventos marcados como síncronos. Para ello, si se detectara el evento síncrono se esperaría a la recepción del evento *event finished* apropiado para finalizar su interpretación. Dado que los eventos no conllevan una respuesta, siempre se devolverá el primer hijo del nodo actual.

El tercero y más complejo deberá interpretar el diálogo esperando a recibir las actualizaciones de las interfaces. Para ello, comenzará creando una cola con todos los fragmentos que contenga el diálogo. Tras ello y mientras queden fragmentos en la cola extraerá un fragmento y creará una interfaz de diálogos facilitándole la información del fragmento. Además, comunicará con la clase *CameraManager* para solicitarla que enfoque a la entidad almacenada en el fragmento. La interfaz de diálogos, en el momento en el que reciba una pulsación del usuario creará un evento f*ragment ended* que indicará que dicho fragmento ya ha terminado de ser representado. Este evento será la vía para decidir cuándo extraer el próximo fragmento de la cola. Una vez se haya vaciado la cola se retornará la cámara a su objetivo original y si el diálogo tuviera al menos dos opciones, se creará una interfaz de diálogo para la selección de la opción. Ésta al detectar la

pulsación de la acción enviará un evento *option selected* con el atributo *option* indicando la opción que se seleccionó. De no haber opciones, se omitirá este paso, marcando como opción seleccionada la 0. Una vez finalizada la interpretación, el nodo a retornar se elegirá en función del valor de la opción elegida.

Con esta pieza el intérprete de secuencias podrá reproducir todo tipo de secuencias y además será fácilmente expandible y reutilizable debido a su bajo acoplamiento y a la facilidad para incluir nuevos subintérpretes en la factoría.

Sección 5.11.2. Editor de secuencias

El editor de secuencias permitirá visualizar las secuencias modificando su contenido en función de los editores disponibles. En el apartado de análisis y diseño se dieron las pautas a seguir para lograr una implementación extensible y reutilizable que se seguirán durante este subcapítulo.

En primer lugar, el editor de secuencias implementará *EditorWindow*. Esto permitirá que sea una ventana del editor, permitiendo así que pueda ser de gran tamaño y permita visualizar grandes secuencias. Para la visualización de la secuencia, basándose en el ejemplo propuesto por Linusmartensson en los foros de Unity y utilizando su librería proporcionada se creó un sistema de ventanas dentro de la ventana del editor. Para ello, utilizando el método GUI.*Window* podremos crear sub-ventanas en el interior del editor actual.

De forma recursiva se accede a todos los nodos y se asigna un id a cada nodo. Dicho id será utilizado en el momento de dibujado de la ventana para poder identificar el nodo que lo contiene. Por otro lado, se creará un rectángulo para indicar el espacio inicial que abarcará dicha ventana. Una vez dibujada la ventana, se creará una línea entre el la ventana anterior a la llamada recursiva y la ventana actual utilizando los rectángulos de las ventanas como valores para situar el origen y destino de la recta.

En el interior del dibujado de cada ventana, como se describió en el análisis y diseño, se utilizará la factoría singleton *NodeEditorFactory* para crear un editor para el nodo actual. Para ello, en primer lugar se solicitará el índice del editor adecuado para el nodo actual y se mostrará un menú desplegable utilizando EditorGUILayout.Popup para mostrar los distintos editores disponibles para el nodo. De detectarse un cambio en la selección, se eliminaría el editor actual y se crearía un nuevo editor basándose en el editor seleccionado.

Una vez creado el editor se le solicitará el dibujado y finalmente al resultado modificado del nodo.

Contaremos con tres tipos de editores de nodos en función de su contenido, a saber, editor de diálogos, editor de eventos y editor de bifurcaciones. Además de éstos, existirá un

editor de nodo vacío que se mostrará cuando el nodo no tenga contenido, indicando así un final de la secuencia.

En este orden se especificará la implementación de cada uno de ellos en los siguientes subcapítulos.

Sección 5.11.2.1. Editor de nodo de diálogo

Será creado siempre que el contenido del nodo sea un diálogo. Para su representación, utilizando la clase *GUILayout* mostrará un nombre para el diálogo, el listado de fragmentos del diálogo y al final el listado de opciones.

Dentro del listado de fragmentos, se recorrerán en forma de bucle, creando para cada uno un campo para la textura de la imagen utilizando *ObjectField* indicando *Texture2D* en el tipo, un campo de texto para el nombre utilizando *TextField* y un área de texto para el contenido utilizando *TextArea.* Además se incluirá un campo ObjectField para poder seleccionar la entidad que lanzaría el fragmento. Automáticamente, de no asignarse ningún valor al nombre o a la textura, cobrarán el nombre y textura cobrarán el valor predeterminado para la entidad. Si el listado de fragmentos contara con al menos cuatro fragmentos automáticamente se convertiría en una vista con barra lateral desplazable utilizando BeginScrollView y EndScrollView. A la derecha de cada fragmento, se mostrará un botón para poder eliminarlo.

Bajo todos los fragmentos, utilizando *Button* se creará un botón que permitirá añadir un nuevo fragmento al diálogo.

Tras los fragmentos, se mostrará el listado de opciones, indicando a su derecha un botón para eliminarlas y bajo ellas otro botón para agregarlas.

Al finalizar el dibujado, el editor comprobará el que nodo que almacena el diálogo cuenta con tantos hijos como opciones tiene el diálogo, creando o eliminando las posibles diferencias hasta igualar el número de nodos con el número de opciones, manteniendo siempre al menos un nodo hijo.

Puede verse el resultado final del editor de diálogos en la figura 5.32.

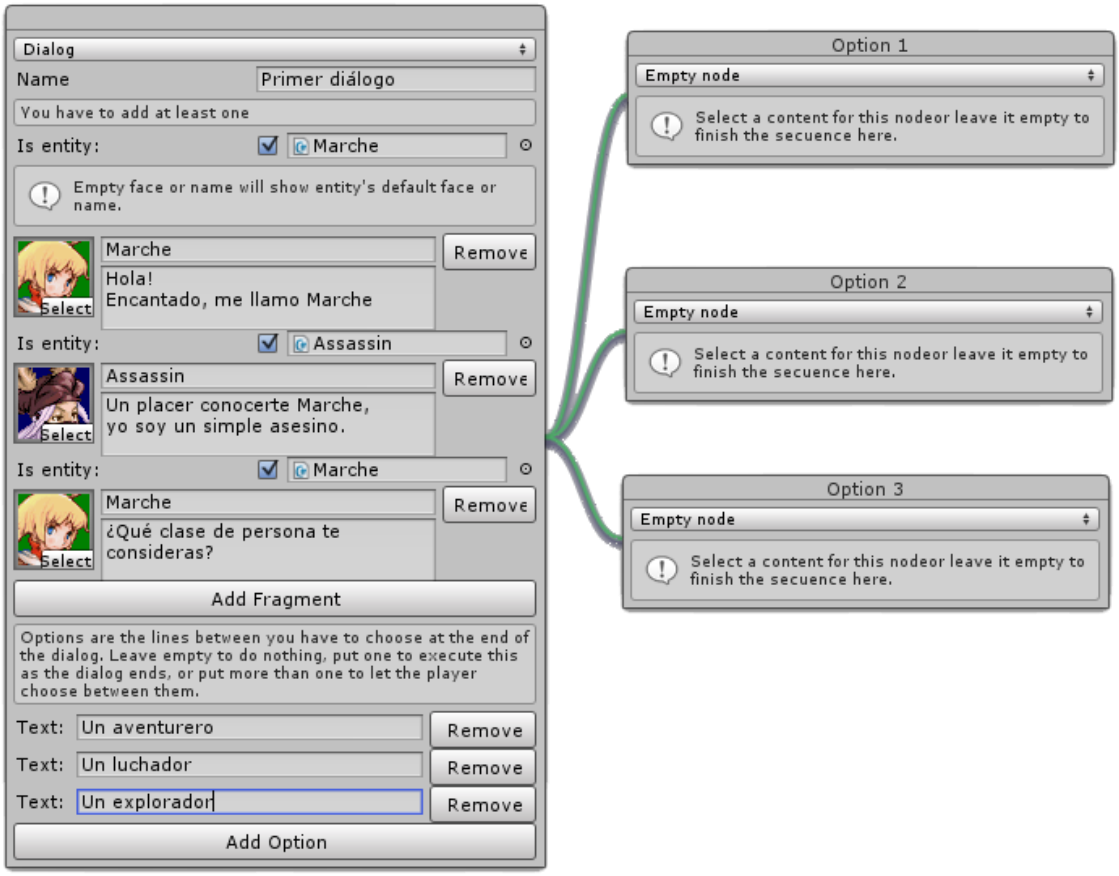

Figura 5.32 - Editor de secuencias mostrando a la izquierda un editor de nodo de diálogo con varios fragmentos y distintas opciones y a la derecha tres editores para nodos vacíos.

En la figura, el editor de nodo de diálogo aparece a la izquierda, y a la derecha, los nodos están editados por el editor de nodo vacío.

Sección 5.11.2.2. Editor de nodo de evento

Al detectarse como contenido del nodo, un *GameEvent*, este editor será seleccionado. En su interior, como se especificó en el análisis y diseño, deberá realizar un proceso similar al del editor de secuencias para crear un editor acorde al evento que contiene.

Para ello, utilizando la factoría singleton *EventEditorFactory* solicitará los nombres de los editores actuales y buscará en ellos el nombre del evento actual. De no encontrarse, se dejará la opción 0 por defecto. A continuación se mostrarán los distintos editores utilizando un *Popup* y se creará un editor cada iteración en función del texto seleccionado. Si se detectara un cambio, se borraría el nombre del evento para evitar volver a crear el editor anterior en la siguiente iteración. Por otro lado, en el momento en el que se cree un editor, se le pasará el evento con *UseEvent* y en este punto el editor deberá cambiar el nombre del evento al que le convenga y crear en él los campos adecuados.

Finalmente, se llamará al método draw para dibujar el editor y se recogerá el resultado del editor para introducirlo en el contenedor del nodo.

Dentro de los editores que la factoría de editores de evento puede crear, se buscarán todos aquellos que implementen EventEditor. Dentro de este proyecto, se han implementado los siguientes editores.

Editor por defecto:

Cuenta con una interfaz compleja para la creación de un evento, en la que se puede editar el nombre y cada parámetro así como definir los tipos de los parámetros. Es un editor bastante complejo para el usuario y su implementación requirió de la creación de un campo personalizado que permitía la selección del tipo a almacenar. Este campo, funciona en conjunto con la clase envoltorio ya mencionada para almacenar los distintos tipos básicos y crear un editor en función del tipo.

En la figura 5.33 se puede observar el resultado del editor por defecto para un evento.

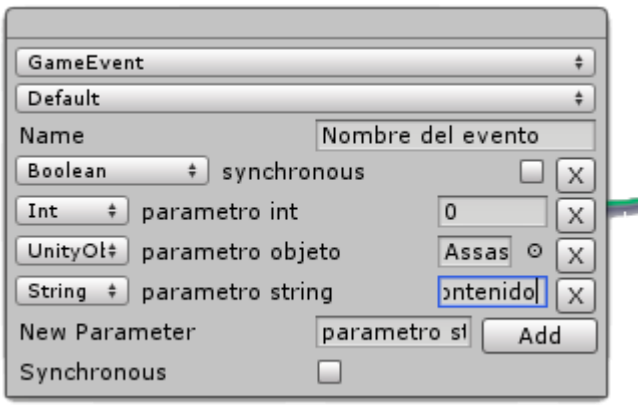

Figura 5.33 - Editor por defecto para un evento.

Como se puede observar es un editor bastante potente, pero que puede no ser suficiente en algunos casos, pues no permite seleccionar *assets* u objetos de un tipo específico.

Editor de eventos *move*:

Dada la frecuencia en la que aparecerán se creó un editor para eventos *move*. Dicho editor cuenta con dos campos *ObjectField*, uno para seleccionar la celda y otro para seleccionar la entidad.

Como se puede observar, el editor personalizado es bastante más simple de manejar y el esfuerzo para su creación es mínimo, ya que consiste en rellenar una clase de apenas 20 líneas, de las cuales, sólo 2 realizan el dibujado del editor, es decir, la parte más "compleja".

En la figura 5.34 se puede observar el resultado.

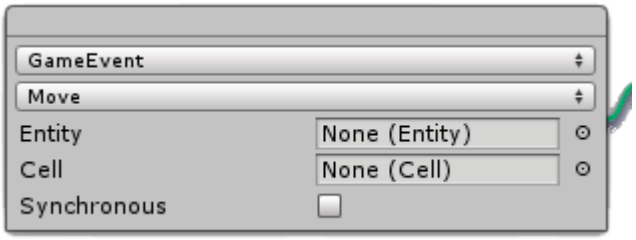

Figura 5.34 - Editor para un evento move.

Editor de eventos *add item*:

Siguiendo el ejemplo del caso anterior y considerando la gran capacidad de ocasiones en la que necesitaremos entregar objetos, se implementó el editor para eventos *add item*. Una vez más con dos campos *ObjectField* simplificamos la creación de estos eventos.

En la figura 5.35 se puede observar el resultado.

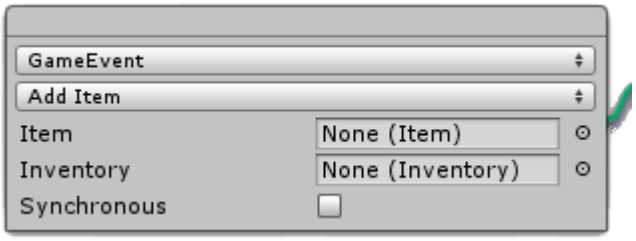

Figura 5.35 - Editor para un evento add item.

Editor de eventos *change switch*:

Dado que en múltiples ocasiones los usuarios querrán cambiar los valores de sus variables globales, este editor simplificará la labor de la creación del evento apropiado.

Su resultado es aparentemente más laborioso. Sin embargo, dado que se había implementado el campo para tipos genéricos desacoplado, se pudo reutilizar con una simple llamada, haciendo este editor no mucho más complejo que los demás.

Entre sus campos cuenta con un *TextField* y el ya mencionado editor de tipos genérico *ParamEditor*.

El resultado puede observarse en la figura 5.36.

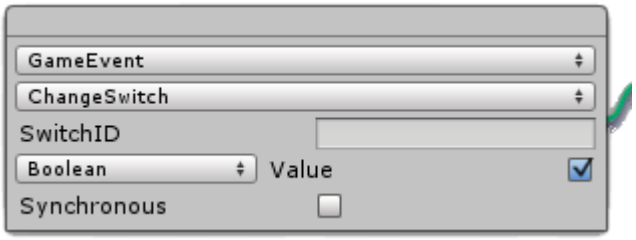

Figura 5.36 - Editor para un evento change switch.

Con este último editor se finalizan los editores para nodos de *GameEvent*.

Sección 5.11.2.3. Editor de nodo de bifurcación

Los nodos de bifurcaciones son aquellos que en su contenido tienen un objeto que herede de *Checkable*.

Dentro de las funciones que deberá realizar de forma genérica consistirá en asegurarse de que el nodo actual contiene dos hijos para poder elegir cada uno en función del resultado de la comprobación.

Para su creación, de manera casi idéntica a como lo realiza el editor de nodos principal, el editor de nodos de bifurcación utilizará una factoría singleton que creará un editor acorde al tipo de bifurcación a editar. Ya que esta estructura se ha explicado en apartados anteriores, pasaremos directamente a la implementación de los editores de cada tipo de bifurcación.

Editor de bifurcaciones basadas en variables globales:

Dado que editará la clase capaz de resolver una comparación entre una variable global, deberemos presentar las variables anteriormente mencionadas al inicio del subcapítulo 5.12, consistentes en el nombre de la variable a comparar, el tipo de comparación, en la variable con la que compararla.

Para ello, en tres simples líneas de código, se implementa el editor capaz de establecer estos tres campos utilizando para el primero *TextField*, para el segundo *EnumPopup* que permite mostrar los valores posibles del enumerado *ComparationTypes* y para el último el ya mencionado *ParamEditor* capaz de editar tipos genéricos.

El resultado de este editor puede observarse en la figura 5.37.

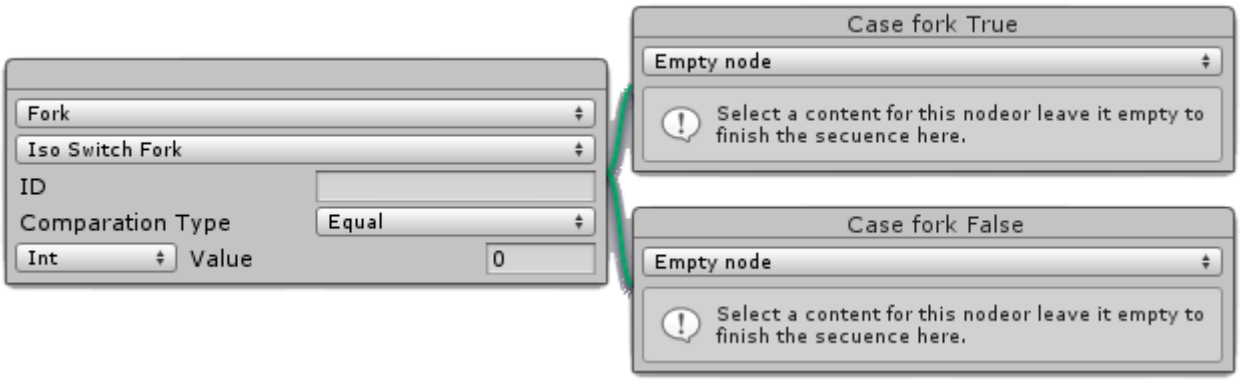

Figura 5.37 - Editor para una bifurcación de variable global.

Editor de bifurcaciones basadas en objetos de un inventario:

Para configurar el objeto y el inventario que se utilizarán para realizar la comprobación, se configurarán utilizando los campos *ObjectField* con el tipo *Item* y *ObjectField* con el tipo *Inventory*.

El resultado se observa en la figura 5.38.

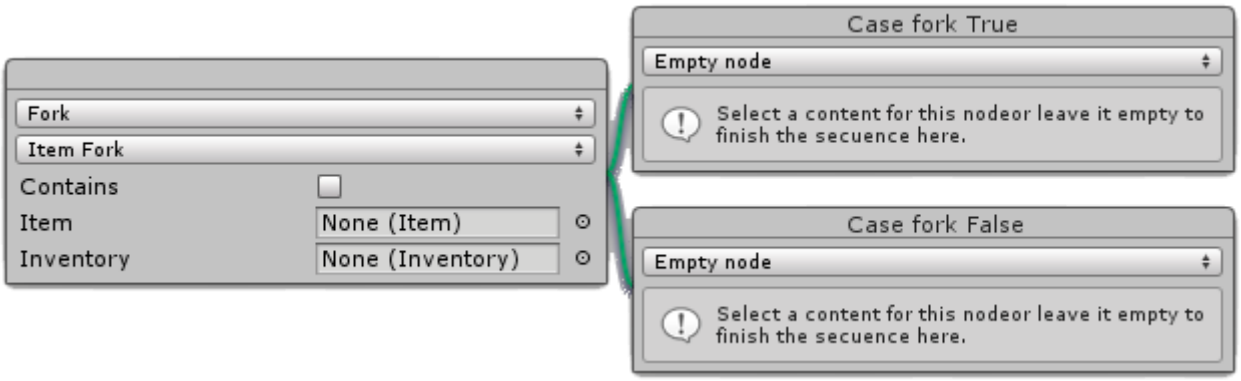

Figura 5.38 - Editor para una bifurcación basada en un objeto.

Para cerrar el subcapítulo de secuencias, se quiere destacar la excepcional capacidad extensible del editor de secuencias, que pese a comenzar siendo una característica deseable acabó por ser una de las características más interesantes del proyecto, decisivo para facilitar a cualquier persona el desarrollo de un juego sin apenas tener conocimientos de programación.

Capítulo 6. Discusión

Habiendo finalizado ya todos los requisitos fundamentales de la aplicación y viéndola con la perspectiva de una herramienta completa para el desarrollo simple de un videojuego, nos hemos dado cuenta de múltiples mejoras y ampliaciones necesarias para hacer que el proyecto esté a la altura de una herramienta de desarrollo completa.

Sin embargo, con la perspectiva de una herramienta novedosa en el campo y con sus múltiples posibilidades de fácil extensibilidad e incorporación de posibles ampliaciones, la herramienta podría llegar a convertirse en una pieza altamente valiosa para el mercado del desarrollo independiente. Si bien herramientas complejas con los mismos ideales de simplicidad y facilidad de uso como RPG Maker [25], en sus comienzos contaban también con posibilidades limitadas, tardaron años en alcanzar las posibilidades que hoy ofrecen tras pasar por múltiples revisiones y mejoras. Y es que hoy en día RPG Maker permite una edición muy avanzada de sus escenarios, con diferentes comportamientos de sus texturas en función de las celdas vecinas. Un estilo de textura inteligente que en el proyecto no se ha investigado por ser demasiado complejas en el nacimiento. Otras herramientas para el pintado y la construcción permiten en RPG Maker una creación mucho más rápida de escenarios. Copiar y pegar, botón de retroceso, pinceles avanzados, son algunas de las facilidades que RPG Maker permite para la edición de mapas y en las cuales el proyecto carece. En cuanto a sus entidades y secuencias, dicha herramienta cuenta con diferentes páginas de secuencias según el estado de la entidad, distintos disparadores y muy avanzados eventos en sus secuencias. Sin embargo, donde este proyecto supera con creces a RPG Maker es en su extensibilidad y en la libertad que proporciona al desarrollador. Donde RPG Maker contiene un conjunto limitado de acciones, métodos de edición, métodos de configuración y limitados conjuntos de acciones para sus secuencias, este proyecto supera a RPG Maker, por su condición de libre modificación y sus altas prestaciones en cuanto a modularidad y extensibilidad. Si cualquier usuario decidiera reimplementar el sistema de pintado, ampliar el conjunto de instrucciones o eventos, ampliar editores, ampliar módulos del editor… podría hacerlo con un pequeño estudio sobre la arquitectura del proyecto. Esto, en RPG Maker es prácticamente impensable salvo algunos valientes desarrolladores que utilizan su ingenio para introducir sus herramientas como es el caso de Layy Meta Editor [54].

Visto con perspectiva del desarrollo acontecido, es cierto que costó mucho alcanzar una especificación clara de los objetivos a conseguir. Uno de los primeros objetivos pretendía era poder realizar un *remake* del clásico La Abadía Del Crimen [27]. Por ello, se estudiaron entornos isométricos similares y su posibilidad de editarlos. Una de las grandes fuentes de referencia que se tomaron fue el videojuego Final Fantay Tactics Advance [29], cuyo estilo recoge un limitado conjunto de acciones (alturas escalonadas, planos inclinados…) pero a la vez recoge los fundamentos limitados del estilo retro, pues en dicho estilo, las limitaciones de los motores impedían muchas más acciones. Tras haber definido la celda y un método para crear mapas se realizó el descubrimiento de que en La Abadía del Crimen, el personaje no sólo se desplazaba de casilla en casilla, sino que

también podía situarse entre casillas, y que la lógica del mapa le afectaba de una forma totalmente distinta en este caso. El error fue claro, haber investigado representaciones pero no haber probado en tiempo real su funcionamiento. En aquel punto hubo que decidir, si rediseñar todo lo que se tenía (un editor y una breve parte de la lógica) o continuar con la exploración. Habiendo perdido brevemente la moral, se observó que las posibilidades del proyecto no habían quedado limitadas sino que simplemente serviría para generar otros juegos. Es cierto que la herramienta puede parecer limitada, pero es en las limitaciones donde ofrece grandes posibilidades de personalización, pues permite una fácil creación de escenarios y cientos de posibilidades de pintado y configuración de decoraciones. Todo esto, sin contar con el amplio marco que deja al descubierto de exploración y explotación el sistema de eventos y secuencias que se ha construido por encima.

En sus inicios, el motor pretendía solamente realizar un editor de mapas, que pudiera, si los plazos lo permitían, albergar entidades que realizaran pequeñas y limitadas labores. Durante esta fase, se mencionó quizás la realización de algún tipo de secuencias, pero en principio, no se estimó que el proyecto fuera capaz de llegar a implementarlas y menos, a diseñar el potente sistema que finalmente se diseñó. Mezclando el estilo retro antes mencionado, pero no queriendo limitar las posibilidades del editor, se tomó como referencia el editor proTile Map Editor [55] que permitía el uso de objetos tridimensionales para la decoración. Este hecho, hizo reflexionar sobre si era realmente importante abarcar el estilo retro exclusivamente o esto se convertiría a la larga en una limitación para el proyecto. Por ello se amplió dicho funcionamiento, permitiendo que las celdas pudieran utilizar todo tipo de texturas más o menos realista, o incluso, que el editor no tuviera por qué utilizar la celda que habíamos creado, sino que se pudiera configurar en función de lo que el usuario decidiera siempre que se utilizara una interfaz común para configurar ambas.

A partir de esa decisión, en el proyecto se tomó la política de, si se estaba diseñando algo y se podía ampliar fácilmente y con poco esfuerzo para lograr requisitos atractivos, merecería la pena realizarlo por la riqueza y la versatilidad que proporcionaría. En el momento, la idea parecía que implicaría mucho más trabajo, pero tras las múltiples ocasiones en las se dio este caso, se comprobó que, comparando con el beneficio que se obtenía, no implicaba tanto trabajo. Por ejemplo, es el caso de utilizar las componentes proporcionadas por Unity para lograr la extensión de las entidades o utilizar arquitecturas más avanzadas para el desarrollo de editores e intérpretes, características que realmente enriquecían mucho el proyecto a costa de un mínimo de esfuerzo.

Habiendo finalizado el editor de mapas antes de lo previsto, en la especificación se pretendía continuar generando múltiples tipos de comportamientos para entidades que permitieran un diseño variable del juego. Sin embargo, tras valorar con perspectiva el objetivo y teniendo en mente los juegos analizados en el estado del arte, se observó que el proyecto no sólo debía dar menos peso a entidades como los enemigos, sino que se

debía dedicar la mayoría del tiempo disponible a la realización de secuencias que permitirían dar vida al género de aventura conversacional y aventura gráfica.

Por ello, se retiraron del planteamiento inicial los enemigos, patrulleros y vendedores, pese a que estos últimos pudieran ser interesantes. Utilizando el resto se podrían desarrollar entidades lo suficientemente ricas como para poder dar ambiente a la escena y permitir que los personajes pudieran explorar las distintas escenas del mapa. A cambio, se extendió muchísimo el sistema de secuencias, inicialmente ideado como un simple sistema de diálogos, capaz de comunicarse mediante eventos con las entidades del juego, y de variar su interpretación siguiendo una estructura de árbol utilizando bifurcaciones. Pese a ser un sistema que tomó como referencia el utilizado en RPG Maker, cuenta con grandes diferencias, pues el sistema incorpora eventos, que son ajenos a como son finalmente representados por el juego, a diferencia del sistema de RPG Maker, cuyos comandos vienen prefijados para una funcionalidad específica.

La clave para el éxito del editor de secuencias radica en su forma altamente extensible en varios grados independientes. En primer lugar, la forma más extensible es la creación de eventos que serán manejados por entidades personalizadas o por manejadores de eventos. La creación de componentes para entidades es uno de los procesos por los cuales los desarrolladores optarían para ampliar las secuencias, siendo éste quizás el más simple y desvinculado con la arquitectura propia del proyecto, y el más divulgado entre los propios usuarios de Unity, y programadores de videojuegos, acostumbrados a la programación por componentes. El segundo proceso, podría constar de ampliar el conjunto de bifurcaciones disponibles. El último y más potente, pero a la vez más complejo por requerir estudiar a fondo la arquitectura, consistiría en crear nuevos tipos de contenidos para los nodos de la secuencia, que darán cabida a nuevas mecánicas en ellas. Será el más complejo, ya que requerirá de la adición de editores en el editor de secuencias y de intérpretes para la interpretación del contenido. En conjunto, las posibilidades permiten a las secuencias ser personalizadas para crear todo tipo de comportamientos.

Sin embargo, a cambio de haber realizado un sistema muy potente, estos cambios en el diseño de la idea del proyecto causaron que, en primer lugar, se proporcionaran pocos comportamientos a las entidades, y por consiguiente, pocos eventos disponibles a la hora de crear secuencias y aprovechar todo el potencial que éstas ofrecen. Sin embargo, dado que se podrá ampliar con poco esfuerzo, permitiendo que comunidades o nuevos desarrolladores ajenos al proyecto implementen nuevas mecánicas, consideramos que se optó por la decisión correcta si se quiere garantizar el futuro del proyecto.

Por otra parte, destacando los aspectos que pudieron ir mejor en el proyecto, el primero y más importante es su tendencia a fallos en la serialización producidos por el motor Unity, causando la pérdida total o parcial de los datos no guardados. A lo largo del desarrollo se puso especial énfasis y cuidado en evitar este hecho que trágicamente a pesar del duro esfuerzo no ha sido completamente solventado. Si bien un desarrollador cuidadoso no tendrá por qué perder datos si realiza guardados con suficiente frecuencia, un

desarrollador novato que no haya adquirido esta tendencia muy probablemente sufrirá las consecuencias. Otro aspecto quizás menos importante, es la capacidad en la celda para crear huecos en ella, dando posibilidad a la creación de puertas, pasadizos, etc. No es un aspecto que se crea excesivamente limitante, pero en versiones futuras sería muy interesante para lograr un aspecto visual mucho mejor, mejorar las capacidades de la celda. En último lugar, las capacidades para pintar o editar son muy limitadas actualmente. Sería muy interesante acabar implementando las herramientas que se mencionan en la memoria acerca del pintado por patrones y el pintado de superficie, o incluso otros modelos de pintado más parecidos a los que ofrece RPG Maker que permiten la selección de varias texturas que se posicionarán en conjunto sobre diversas caras.

De cara al futuro inmediato del proyecto se estima que será necesario ampliar contenidos, así como testar y comprobar el funcionamiento real de las herramientas en la realización de un juego completo. Durante la realización de dicho juego, se podrá obtener la información necesaria para conocer, especialmente dónde fallan y son más complicadas de utilizar las herramientas. Por el contrario, también se conocerán los puntos fuertes del proyecto permitiendo así que sean mejorados y ampliados para ser aún más atractivos. Por otro lado, se conocerán posibles errores que no se previeron en origen y los cuales no se pueden detectar hasta que no se prueban. Una vez se hayan diseñado varios escenarios completos con sus decoraciones y entidades, éstos servirán de apoyo para trabajar de lleno con el equipo de desarrollo del juego, proporcionando correcciones a dichas partes.

Con la finalización de esta primera fase de nacimiento del proyecto los usuarios más novatos podrían utilizar las herramientas para narrar elaboradas historias, y los más entusiastas podrían tomar el proyecto y ampliarlo para generar juegos complejos. Sin embargo, la herramienta no debería utilizarse aún para la realización profesional de un juego completo, a menos que se asumieran los costes de las posibles ampliaciones que requeriría la utilización de la herramienta. Pese a esto, en su estado la herramienta facilita muchísimas tareas, y da soporte a la creación de nuevas de manera sencilla, lo que podría llevar a considerar su uso para proyectos que no quieran partir de cero.

En el panorama actual, es la primera herramienta para un motor de videojuegos profesional, como es Unity, capaz de desarrollar juegos retro con aspecto 2D en perspectiva isométrica utilizando técnicas puramente 3D, un concepto nada trivial conformado por numerosas transformaciones y juegos de cámara que permiten al usuario integrar la potencia de un editor tridimensional con los resultados retro isométricos que se pretendían conseguir y abre las puertas a la recuperación de otro de los estilos como el de Ragnarok Online [34] que mezclan objetos bidimensionales y tridimensionales.

Queremos cerrar este apartado de discusión destacando el gran futuro del proyecto, debido a su innovación en funciones y a su potente capacidad extensible, que permitirá a comunidades de usuarios o desarrolladores ampliar la herramienta a su medida y aprovechar al máximo la arquitectura que se ha diseñado.

Capítulo 7. Conclusiones

En primer lugar, se pueden crear videojuegos utilizando este proyecto, sin necesidad de tener conocimientos de programación. Únicamente, el usuario deberá aprender a utilizar a grandes rasgos el motor Unity y aprender a manejarse con este proyecto.

Estos videojuegos pueden tener un progreso dentro de su contenido, estableciendo hitos, que, al alcanzarse, permiten al jugador avanzar en las distintas fases del videojuego, o, alcanzar el final del mismo. Por consiguiente, dicho contenido puede establecer una trama argumental, pudiendo desarrollar historias interactivas en las que el jugador tomará parte.

Estas fases, o mapas, podrán ser creadas con facilidad utilizando el editor de mapas de este proyecto, generando nuevas fases con comodidad y utilizando herramientas para la construcción, pintado, decoración y posicionamiento de entidades. Estas herramientas presentan cómodos menús e interfaces que ayudan al usuario a realizar la tarea de edición de mapas con sencillez. En conjunto, estos mapas generan un entorno lógico en el que las entidades realizan sus operaciones.

En dichos mapas, se puede realizar utilizando los componentes de este proyecto, un efecto óptico que consigue que, para las texturas que se posicionan sobre él, su visualización en perspectiva isométrica, sea la misma que la visualización del archivo original de las mismas.

La herramienta de construcción ayuda a crear y posicionar celdas en el mapa. Dichas celdas mantienen un posicionamiento ordenado, alineándose con respecto a una rejilla sobre el plano horizontal. Por último, las celdas se comportan como unidad, no solapándose entre sí.

Estas celdas mantienen una estructura de Ortoedro, manteniendo un ancho similar entre todas ellas, pero permitiendo variar su altura en fragmentos de la mitad de su ancho. Finalmente, estas celdas pueden cambiar su estructura por la de un paralelepípedo para representar, en su cara superior, una superficie inclinada, o semi-inclinada, dependiendo de si dicha inclinación se eleva una cara completa o media,

La herramienta de pintado ayuda a pintar las caras de las celdas creadas, permitiendo colocar diferentes texturas sobre las múltiples caras de una celda, utilizando el cursor del ordenador, e imitando el comportamiento de una brocha de pintado.

Dichas texturas pueden ser texturas estándar, o texturas isométricas, un tipo especial de textura que realiza un mapeado de texturas específico en la celda, pudiendo delimitar en dicho mapeado de texturas, el área a mostrar, utilizando un editor. Dicha área, es el área concreta que deseamos colocar al pintar dicha textura isométrica sobre la cara de una celda.

La herramienta de decorado ayuda a colocar decoraciones sobre las celdas, permitiendo colocar dichas decoraciones en multitud de formas diferentes, dependiendo de una serie de parámetros modificables, y de la cara de la celda donde se desee posicionar.

Estas formas diferentes de colocación permiten: colocar decoraciones en los tres laterales visibles de una celda en perspectiva isométrica, y además, el colocado de dichas decoraciones de manera perpendicular al plano horizontal o paralela a la cara de la celda donde se posiciona. Estos modos permiten evitar solapamiento de elementos. Como modo adicional, se puede establecer que una decoración se posicione relativamente a otra.

Además, se pueden crear decoraciones animadas utilizando la herramienta, pudiendo establecer el tiempo de cada fotograma y la rotación que dichos fotogramas debe realizar. Adicionalmente a esto, se puede especificar que la decoración se destruya al terminar, y que realice un número determinado de rotaciones antes de destruirse.

Se pueden crear emociones y efectos utilizando las funcionalidades de animación, y de posicionamiento relativo de una decoración con respecto a otra. Dichas emociones no cuentan con asistente, y deben almacenarse como objetos prefabricados para su posterior instanciación.

La creación de entidades se verá simplificada por la agrupación de comportamientos previamente implementados, compatibles a través de una interfaz de acciones, reacciones y actualizaciones con el resto del sistema. Los comportamientos implementados permiten realizar funciones de inicio y procesado de secuencias, almacenamiento y utilización de objetos, representación de objetos, teletransportadores, etc.

La gestión de los distintos sistemas del juego es gestionada por una clase central Game que proporciona a sus diferentes gestores actualizaciones y eventos según se vayan produciendo. El gestor más importante es el de mapas, capaz de distribuir eventos y actualizaciones a entidades. Otros gestores que forman parte de su ciclo de vida son el de secuencias y de emociones. El diseño de Game permite la fácil ampliación con nuevos gestores.

El sistema de eventos de juego proporciona a los distintos partícipes del sistema una interfaz común de mensajería transparente al origen o el destino, que garantiza la reutilización y el procesamiento de forma dinámica de los acontecimientos.

Las secuencias del juego permiten avanzar en el juego de una forma simple pues se proporciona con ellas un editor de secuencias avanzado y muy extensible capaz de modificar variables globales, distribuir eventos, discernir sobre diversas situaciones y mostrar secuencias de diálogo entre distintas entidades. Para ello, un intérprete de secuencias se encarga de materializar en el juego lo que el editor ha almacenado en la secuencia.

El controlador del juego proporciona a los objetos participes del sistema (entidades o gestores) la capacidad de recibir una entrada unificada entre los posibles controles físicos o táctiles. Trabaja en conjunto con el gestor de interfaces para asegurar la independencia de las entradas del usuario entre ellas y el mapa.

Aportaciones de los integrantes

1. Iván José Pérez Colado

Para comenzar, me gustaría hacer alusión al gran trabajo en equipo que hemos realizado Víctor y yo, diseñando prácticamente todos y cada uno de los elementos que conforman este proyecto en equipo, y repartiendo de una manera equitativa cada una de las tareas que iban surgiendo.

En general, he participado participé en la definición y desarrollo de todos los conceptos, habiendo participado de forma activa en todas las reuniones realizadas, sin faltar a ninguna de ellas.

En primer lugar, dediqué gran parte de la fase inicial del proyecto, a realizar una gran investigación del estado del arte, permitiéndonos obtener buenos y concretos referentes a la hora de diseñar y tomar grandes decisiones en el proyecto. Adicionalmente, realicé la investigación de los títulos propuestos por Víctor.

Además de los videojuegos, en mi investigación encontré herramientas como el Layy Meta Editor, o ProGrids. Herramientas que han servido de conceptos fundamentales para el desarrollo del proyecto.

Personalmente, me siento especialmente autor del sistema de decoraciones, en el que he invertido infinidad de horas, hasta conseguir que fuera capaz de realizar todos aquellos requisitos que fueron surgiendo durante la vida del proyecto, ideando y definiendo los diferentes modos de posicionamiento de decoraciones para que permitieran al usuario realizar dicha colocación con total libertad, contrastando todas estas ideas con Víctor para diseñar juntos el sistema que deseábamos, así como el diseño e implementación sistema de emociones y animaciones, que permiten que los personajes se acerquen un poco más a tener vida y comportamientos humanos.

Además, he realizado el cálculo matemático, con pequeñas aportaciones de Víctor, de la deformación necesaria para los diferentes modos de pintado de las decoraciones, así como la creación de casi todas las figuras que contienen dibujos técnicos, que representan las deformaciones, así como los conceptos de perspectiva y conceptos visuales.

Por otra parte, me siento precursor y diseñador del sistema de secuencias, pues desde el comienzo del diseño del proyecto, siempre quise que con él se pudieran generar pequeñas películas interactivas, en las que el jugador tuviera que tomar parte para descifrar enigmas, o simplemente observar como los personajes del videojuego cobraban vida propia, y desde el momento en el que compartí dicha idea con Víctor, el también compartió mi deseo por alcanzar este objetivo.

Asimismo, cree la primera versión del sistema de secuencias, siendo este en sus comienzos únicamente un sistema de diálogos, así como la interfaz, tanto de editor, como la representada en el videojuego, y un intérprete, llamado *Talker*, que era capaz de recibir los diferentes fragmentos de diálogo y mostrarlos.

Poco a poco dicho sistema fue evolucionando gracias a los aportes que creamos Víctor y yo en conjunto, generando un intérprete de secuencias extensible, formado por diferentes nodos que se suceden en la ejecución. Dentro de este sistema, además, implemente el sistema de hitos, compuesto por un conjunto de interruptores, llamados *IsoSwitches*. Asimismo, diseñé e implemente las bifurcaciones, que podían producirse por diferentes cosas. Estas, las implemente creando la clase nodo de editor *Fork*, y utilizando las posibilidades de las opciones que nos proporcionan los diálogos. Todas estas implementaciones, no se realizaron hasta alcanzar un diseño y especificación aceptado tanto por Víctor, como por mí, discutiendo los nuevos conceptos que iban surgiendo.

Este sistema de secuencias no hubiera sido posible de no haber sido por la extensa investigación que realicé sobre el motor RPG Maker, el cual me ha servido como base para afianzar muchos de los conceptos que, en el diseño inicial, estaban mal planteados, y que pudieron ser rediseñados y reinterpretados. Dichos conceptos se aplicaron tanto al editor de mapas como al editor de secuencias.

Asimismo, dedique gran cantidad de tiempo a recortar y obtener texturas y gráficos de videojuegos reales, absolutamente necesarios para el desarrollo de este proyecto, pudiendo realizar pruebas y depuración, que nos permitieron contrastar los objetivos con los resultados.

Por otra parte, y tras el diseño en conjunto de un sistema de eventos, diseñé e implemente el concepto de *EvenManager*, cuyo objetivo es manejar todos aquellos eventos que no están destinados a ser interpretados por entidades, ya que su implementación como componente de una entidad no tenía sentido. Además, he implementado concretamente el *IsoSwitchesEventManager* y el *AnimationManager*.

Por último, durante el desarrollo del proyecto, seguimos diariamente una planificación creada a grandes rasgos, planteada con hitos, incluyendo en ella, la planificación aproximada del volumen de la memoria que debíamos alcanzar. Particularmente he escrito todo el apartado del estado del arte, así como gran parte del apartado 3, especialmente la especificación de las interfaces, partes del apartado 4 y del apartado 5, relacionadas con todos aquellos elementos que yo había implementado y que se han explicado en esta sección de aportaciones.

Adicionalmente he revisado gran parte del texto escrito por Víctor, así como regenerado y editado muchas de las figuras que se encontraban en el documento. Finalmente, realizamos en equipo la edición y maquetado del documento, así como los capítulos 1, 6, y 7 de este documento.

2. Víctor Manuel Pérez Colado

En primer lugar, participé en la definición del concepto del proyecto. Además de haber realizado pequeñas aportaciones a cada apartado, me siento especialmente autor de la mayoría de ideas sobre la edición de celdas, imaginado el concepto de rejilla que más tarde evolucionó en el concepto de celda fantasma y habiendo ideado sistemas fáciles para la selección de texturas y la realización de un asistente que ayudaría visualmente a realizar el mapeado de texturas.

Otras ideas que aporté fue la creación del sistema de eventos de juego y la caracterización de las entidades mediante el uso de componentes *EntityScript*. Tras haber definido los comportamientos, aporté la idea de que una de las entidades fuera el jugador, creando el *player* y surgiendo a causa de ello, la idea del controlador unificado y la separación entre eventos de juego y eventos de controlador.

En general, participé en la definición y pulido de todas las ideas, habiendo participado de forma activa en todas las reuniones que se realizaron sin haber faltado ni un día.

Pese a haber aportado muchas ideas, siempre fueron discutidas en equipo por lo que no quiero quitar peso a la labor de Iván, quien me ayudó siempre a definir todas las ideas que se me iban ocurriendo.

Entre mis investigaciones, dediqué parte del tiempo del principio del proyecto a investigar el funcionamiento del editor de Unity, habiendo realizado varios tutoriales completos para la realización de juegos en el ámbito de funcionamiento normal de Unity. Tras ello, y haber tomado contacto con el motor, me propuse investigar a fondo las posibilidades de modificación y extensión del editor que Unity ofrecía. Esto último se lo comuniqué a Iván y puse especial énfasis en enseñarle a utilizarlo, disminuyendo así su tiempo de aprendizaje y permitiéndole que él realizara investigaciones de otro tipo, como referencias de juegos y otras bases necesarias para conformar el concepto.

Otra de mis ramas de investigación se centró en la búsqueda de otros editores similares para Unity encontrando el mencionado en el estado del arte proTile Map Editor y algún otro editor isométrico ajeno a Unity que nos ayudó a definir la idea a realizar.

La última investigación que realicé fue referente al mapeado de texturas en el cubo, como conté en el prototipo en el que se trató de utilizar un cubo como celda para el mapa.

Una vez acabamos de definir las primeras partes del concepto y los objetivos, el resto del tiempo lo dediqué a realizar diseños e implementaciones junto a Iván.

Entre otras cosas, desarrollé en solitario el primer prototipo del proyecto en el que se utilizaban casillas en 2D sobre el editor en ejecución. Tras acabar este prototipo, en

conjunto con Iván desarrollamos el segundo prototipo con la rejilla en tres dimensiones que posteriormente serviría de base para comenzar el editor.

En solitario, tras haber desistido el mapeado del cubo me propuse implementar la celda que necesitaríamos para el editor de mapas. Diseñé y mejoré en solitario dicha celda siempre contrastando la opinión de Iván con antes de tomar decisiones importantes como su superficie y su crecimiento escalonado.

Tras dividir el trabajo del resto del editor, me propuse a realizar el pintado y mapeado de las texturas e Iván se dedicó a las labores de decorado. Por mi parte diseñé los tres estilos de mapeados y me encargué de realizar el asistente de texturas que permitiría configurar las IsoTexturas de una forma sencilla. Aquella primera versión fallaba muchísimo porque la celda que había implementado no se serializaba correctamente, aspecto que no conocíamos de Unity.

Por ello, dediqué las semanas siguientes a investigar temas de serialización y garantizar que el mapa no destruiría todas sus caras cada vez que cerráramos el programa. Es entonces cuando, investigando, averiguamos el uso de ScriptableObject para clases que no serían MonoBehaviour pero que serían objetos de nuestro diseño.

Una vez realizadas las reparaciones y teniendo un mapa que se podía pintar y editar correctamente, dediqué los siguientes días a diseñar las entidades y sus componentes. Con la ayuda de Iván, definimos tres de los cuatro métodos que definen hoy a las entidades, su EventHappened, su Tick y su Update, que servirían de vía para la comunicación con otras entidades. Tras haber hecho entidades que se movían y realizaban algunas funcionalidade básicas, mientras que me dedicaba a la implementación del controlador del juego, descubrí que las entidades necesitarían un cuarto método GetOptions, que las permitiría decirle a la persona que está jugando, qué acciones puede hacer sobre ellas.

Habiendo cerrado ya el capítulo de entidades, Iván comenzó a explorar el mundo de las secuencias y yo me dediqué a investigar el mundo de las interfaces gráficas. Implementé el sistema de GUIManager para la gestión de preferencias y unificación del dibujado y tras ello, implementé la interfaz de selección de acciones que pretendía maximizar las posibilidades táctiles.

Por último cuando Iván había realizado una primera versión del sistema de secuencias, lo retomé y en conjunto con Iván lo rediseñamos para maximizar sus capacidades de extensibilidad. Habiéndolo diseñado, me dediqué a su implementación y realicé todos los editores para eventos.

Durante todo el proceso, llevamos al día la creación de los documentos conforme íbamos realizando diseños e implementaciones. Dentro de la memoria escribí gran parte de los capítulos 3, 4 y 5 y ayudé a Iván a revisar el estado del arte que él había escrito. Juntos, realizamos todo el maquetado de la memoria y revisamos el texto.

Bibliografía

[1] Pac-Man - Videogame by Midway Manufacturing Co. [Online]. Available: http://www.arcade-museum.com/game_detail.php?game_id=10816. [Accessed: 20-Jun-2014].

[2] Space Invaders - Videogame by Midway Manufacturing Co. [Online]. Available: http://www.arcade-museum.com/game_detail.php?game_id=9662. [Accessed: 20-Jun-2014].

[3] Angry Birds - Home. [Online]. Available: https://www.angrybirds.com/. [Accessed: 20-Jun-2014].

[4] Zelda Universe - The Legend of Zelda series for Nintendo Systems. [Online]. Available: http://www.zelda.com/universe/. [Accessed: 20-Jun-2014].

[5] Half-Life en Steam. [Online]. Available: http://store.steampowered.com/app/70/. [Accessed: 20-Jun-2014].

[6] Call of Duty®. [Online]. Available: http://www.callofduty.com/es. [Accessed: 20-Jun-2014].

[7] Terraria. [Online]. Available: http://www.terraria.org/. [Accessed: 20-Jun-2014].

[8] Magicka | Paradox Interactive. [Online]. Available: http://www.magickagame.com/. [Accessed: 20-Jun-2014].

[9] XNA Game Studio - Home. [Online]. Available: https://msxna.codeplex.com/. [Accessed: 20-Jun-2014].

[10] Antichamber - A Mind-Bending Psychological Exploration Game. [Online]. Available: http://www.antichamber-game.com/. [Accessed: 20-Jun-2014].

[11] Sanctum | Coffee Stain Studios. [Online]. Available: http://www.coffeestainstudios.com/games/sanctum. [Accessed: 20-Jun-2014].

[12] Unreal Engine Technology | Home. [Online]. Available: https://www.unrealengine.com/. [Accessed: 20-Jun-2014].

[13] Rust «. [Online]. Available: http://playrust.com/. [Accessed: 20-Jun-2014].

[14] Slender: The Arrival - Now Available on Steam. [Online]. Available: http://www.slenderarrival.com/. [Accessed: 20-Jun-2014].

[15] Unity - Game Engine. [Online]. Available: http://unity3d.com/es. [Accessed: 20-Jun-2014].

[16] Rockin' Indie Games one pixel at a time - Indie DB. [Online]. Available: http://www.indiedb.com/. [Accessed: 20-Jun-2014].

[17] FEZ | Polytron Corporation. [Online]. Available: http://fezgame.com/. [Accessed: 20-Jun-2014].

[18] Starbound. [Online]. Available: http://playstarbound.com/. [Accessed: 20-Jun-2014].

[19] Minecraft. [Online]. Available: http://minecraft.net/. [Accessed: 20-Jun-2014].

[20] ZENONIA® - Aplicaciones de Android en Google Play. [Online]. Available: https://play.google.com/store/apps/details?id=com.gamevil.zenonia&hl=es. [Accessed: 20-Jun-2014].

[21] Game Dev Story - Aplicaciones de Android en Google Play. [Online]. Available: https://play.google.com/store/apps/details?id=net.kairosoft.android.gamedev3en&hl=es. [Accessed: 20-Jun-2014].
[22] Towns Game. [Online]. Available: http://www.townsgame.com/. [Accessed: 20-Jun-2014].
[23] Monument Valley: an iOS and Android game by ustwo. [Online]. Available: http://www.monumentvalleygame.com/. [Accessed: 20-Jun-2014].
[24] Filmation (game engine) - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Filmation_(game_engine). [Accessed: 20-Jun-2014].
[25] Make Your Own Game with RPG Maker. [Online]. Available: http://www.rpgmakerweb.com/. [Accessed: 20-Jun-2014].
[26] Umberto Eco, *En el nombre de la rosa*.
[27] José Manuel Braña Álvarez, Enrique Colinet, José Manuel Fernández, Antonio Giner, José Antonio Morales, Juan Manuel Moreno, Manuel Pazos y José Luís Sanz, *OBSEQUIUM: Un relato cultural, tecnológico y emocional de La Abadía del Crimen*. Ocho Quilates.
[28] Final Fantasy Tactics - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Final_Fantasy_Tactics. [Accessed: 20-Jun-2014].
[29] Final Fantasy Tactics Advance - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Final_Fantasy_Tactics_Advance. [Accessed: 20-Jun-2014].
[30] Final Fantasy Tactics Advance 2: Grimoire of the Rift (Nintendo DS) | MeriStation.com. [Online]. Available: http://www.meristation.com/nintendo-ds/final-fantasy-tactics-advance-2-grimoire-of-the-rift/juego/1523386. [Accessed: 20-Jun-2014].
[31] Fez (ropa) - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Fez_(ropa). [Accessed: 20-Jun-2014].
[32] About Unreal Engine 4. [Online]. Available: https://www.unrealengine.com/products/unreal-engine-4. [Accessed: 20-Jun-2014].
[33] CRYENGINE Features. [Online]. Available: http://cryengine.com/features. [Accessed: 20-Jun-2014].
[34] Ragnarok Online - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Ragnarok_Online. [Accessed: 20-Jun-2014].
[35] Videojuego de rol multijugador masivo en línea - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Videojuego_de_rol_multijugador_masivo_en_l%C3%ADnea. [Accessed: 20-Jun-2014].
[36] Aventura gráfica - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Aventura_gr%C3%A1fica. [Accessed: 20-Jun-2014].

[37] The Secret of Monkey Island - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/The_Secret_of_Monkey_Island. [Accessed: 20-Jun-2014].
[38] Monkey Island - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Monkey_Island. [Accessed: 20-Jun-2014].
[39] SCUMM - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/SCUMM. [Accessed: 20-Jun-2014].
[40] LucasArts.com | Welcome to LucasArts.com. [Online]. Available: http://www.lucasarts.com/index.htmls. [Accessed: 20-Jun-2014].
[41] The Elder Scrolls Official Site. [Online]. Available: http://www.elderscrolls.com/skyrim. [Accessed: 20-Jun-2014].
[42] Motor de videojuego - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Motor_de_videojuego. [Accessed: 20-Jun-2014].
[43] RPG Maker - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/RPG_Maker. [Accessed: 20-Jun-2014].
[44] RPG Maker 2003 - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/RPG_Maker_2003. [Accessed: 20-Jun-2014].
[45] RPG Maker XP | RPG Maker | Make Your Own Game! [Online]. Available: http://www.rpgmakerweb.com/products/programs/rpg-maker-xp. [Accessed: 20-Jun-2014].
[46] RPG Maker VX Ace | RPG Maker | Create A Game! [Online]. Available: http://www.rpgmakerweb.com/products/programs/rpg-maker-vx-ace. [Accessed: 20-Jun-2014].
[47] Videojuego de rol - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Videojuego_de_rol. [Accessed: 20-Jun-2014].
[48] Android - Meet Android. [Online]. Available: http://www.android.com/meet-android/. [Accessed: 20-Jun-2014].
[49] Apple - iOS 7. [Online]. Available: http://www.apple.com/es/ios/. [Accessed: 20-Jun-2014].
[50] Unreal Engine | Plugins. [Online]. Available: https://docs.unrealengine.com/latest/INT/Programming/Plugins/index.html. [Accessed: 20-Jun-2014].
[51] Download and run the Unreal Engine 4 Elemental benchmark demo on your PC | ExtremeTech. [Online]. Available: http://www.extremetech.com/gaming/181608-download-and-run-the-unreal-engine-4-elemental-benchmark-demo-on-your-pc. [Accessed: 20-Jun-2014].
[52] Asset Store. [Online]. Available: https://www.assetstore.unity3d.com/en/. [Accessed: 20-Jun-2014].
[53] OpenGL - The Industry Standard for High Performance Graphics. [Online]. Available: http://www.opengl.org/. [Accessed: 20-Jun-2014].
[54] Layy Meta Editor Cheat Sheet - GubiD's Tactical Battle System v2(RMVXAce) - Redmine. [Online]. Available: http://gubi.us:8001/redmine/projects/0_010/wiki/Layy_Meta_Editor_Cheat_Sheet. [Accessed: 20-Jun-2014].

[55] proTile Map Editor | Tile Map Editor for Unity. [Online]. Available: http://protilemapeditor.com/. [Accessed: 20-Jun-2014].
[56] PROGRIDS | ProTools. [Online]. Available: http://www.protoolsforunity3d.com/progrids/. [Accessed: 20-Jun-2014].
[57] Unity - Manual: Unity Manual. [Online]. Available: http://docs.unity3d.com/Manual/index.html. [Accessed: 20-Jun-2014].
[58] Desarrollo en cascada S. Pressman, Roger. Ingeniería del Software: Un enfoque práctico, 3.ª Edición, Pag. 26-30.
[59] Método cientifico "Rules for the study of natural philosophy", Newton 1999, pp 794-6, libro 3, The System of the World.
[60] Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. .
[61] Unity - Manual: Creating and Editing Terrains. [Online]. Available: http://docs.unity3d.com/Manual/terrain-UsingTerrains.html. [Accessed: 20-Jun-2014].
[62] UV mapping - Wikipedia, the free encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/UV_mapping. [Accessed: 20-Jun-2014].
[63] Algoritmo de búsqueda A* - Wikipedia, la enciclopedia libre. [Online]. Available: http://es.wikipedia.org/wiki/Algoritmo_de_b%C3%BAsqueda_A*. [Accessed: 20-Jun-2014].
[64] Unity - Manual: GUI Scripting Guide. [Online]. Available: http://docs.unity3d.com/Manual/GUIScriptingGuide.html. [Accessed: 20-Jun-2014].
[65] Unity - Learn - Documentation. [Online]. Available: http://unity3d.com/learn/documentation. [Accessed: 20-Jun-2014].
[66] Unity - Editor. [Online]. Available: http://unity3d.com/learn/tutorials/modules/beginner/editor. [Accessed: 20-Jun-2014].

Anexo 1. Title (in English)

A Toolset for the Development of Retro-Style Action-Adventure Video Games with 3D Isometric Graphics

Anexo 2. Introduction (in English)

The video game or entertainment software is today one of the most important and valued part in the world of computing sectors. Over the past few decades has experienced an exponential growth, having reached almost all households in Europe and North America, and much of the rest of the world's households. In economic terms, it generates annually more money than the film industry and music together, acquiring a strong importance in the market, opening a multitude of jobs and revitalizing the economy.

Today this exciting world takes hundreds of forms and genres for all likes and personalities. From the classical genre Arcade, with outstanding titles throughout history as the Pacman [1] or Space Invaders [2] or recent and popular titles like Angry Birds [3], through the gender adventures with titles like The Legend Of Zelda [4] that continues progressing today and renewing itself with new mechanics to explore the genre, to the first person shooter genre with titles that include successful franchises such as Half Life [5] and Call Of Duty [6].

The trend in recent decades has always been better evaluate video games that succeeded best visuals and exploits the full potential of the chips as they were throwing. This caused, small and new studies great difficulties to adapt itself and produce the results that the public was expecting.

Large studies, meanwhile, at the same time grew, developed and expanded their tools, being increasingly powerful and capable of generating video games with a better result in less time and costs. Some few tools that came to the market, far fewer were available to the economy of small and newborns studios and developers.

With the commercialization of the game engines, the game development companies have suffered an evolution in which these have begun to put entire teams to develop and maintain graphics, or even engines, leaving the game development, to focus entirely in the development of their engine.

This change created a new product that was commercialized in the market, and therefore, all companies that were commercializing engines would have to participate in a commertial battle, in which both the high demand for graphic engines, as the large number of new graphics engine that were being created, caused gradually the price of licenses for those engines were growing down to have a low price compared to the start.

This, with the addition of new free and even free (as in freedom) and open source alternatives, did see a sector benefited from this strong competition: The sector of independent developers and small development teams games. These, taking advantage of the falling prices and the arrival of free graphics engines, gained access to multiple development platforms, highly costly to develop from the start for an independent developer or a small team, without having the resources that were previously needed.

Because of this change in that now the market offers plenty of tools to achieve excellent results at low cost graphics, and dramatically reducing development time, new studies and game developers have achieved publishing successful games having very few resources, especially economically.

Taking advantage of this change, new titles and franchises have emerged in recent decades that had an overwhelming success, being top selling games and winning countless awards. We can find titles like: Terraria [7] by Re-Logic or Magicka [8] by Paradox Interactive, developed with the free Microsoft XNA graphics engine [9]; AntiChamber [10] by Alexander Bruce, or Sanctum [11] by Coffee Stain Studios developed using very cheap license of the engine Unreal Engine 3 [12]; Rust [13] by Facepunch Studios LTD or Slender: The Arrival [14] by Blue Isle Studios developed using Unity [15], motor license free for little developments; and a lots of additional titles [16] that, despite not being listed, they have also had a big success and have raised large amounts of money.

With the arrival of smartphones and smart tablets, the video game world had also an extra boost. These new platforms, not only allow the creation of new genres of video games, also allow the retrieval of older video game genres that require less computational power and fewer resources for developers.These older genres bring a different experience to the above, and moreover, more suitable to these, because the older games were characterized by being focused on the player's ability to pass different levels and short game times, that fits the time of use of such devices.

The player’s mentality has also changed. Previously, players were looking for games, that could bring great difficult challenges and repetitive tasks, until they could pass them. The video game market was much tighter, and the games that were the longest and hardest to finish, were the top rated games Currently, players have a wide video game market of all types, genres and styles, and despite repeating the concept of challenge and repeat, this does not mean that these games will have more chances of getting more players. Simplicity, along the challenge, form now the key for being attractive.

The style of retro video game aims to collect all the features of the first games with limited graphics capabilities, and build new games using these features, on the new tools and platforms, bringing the possibility of innovation by mixing the older and new concepts. Such is the case of games like FEZ [17] by Polytron Corporation, which despite having a retro style, has been one of the most successful games of the last year because of its attractive gameplay. Another more recent game is Starbound [18] by Chucklefish which expands the open world concept from Terraria [7], to the immensity of the universe, to offer a practically unlimited experience in a retro 2D environment. This section of retro games could not finish without alluding Minecraft [19] Mojang, which creating a fusion of: 3D graphics, and pixelated minimal textures, manages to create, the perfect harmony of retro style today.

Inside the world of smart phones and tablets, the retro style has carved itself a special place, with complete game sagas as Zenonia [20] by GAMEVIL, or simulator-game series by Kairosoft highlighting its most important title Game Dev Story [21], whose representation is carried out with an isometric retro style.

And is this style, the one that is wanted in this project, because it raises the recovery of the Spanish classic titles like La Abadia del Crimen [22].

Currently, within the isometric games we can find small titles like Towns [23] by the Spanish developer Xavi Canal or Monument Valley [24] by ustwo. However, just a little few games mix the retro style with the isometric perspective. Therefore, as it is a style that has been highly contested throughout the history of video games, because of the games made with the graphics engine Filmnation [25] for Spectrum, but has been discontinued, this project is made to give a new life to this particular style.

Getting references of successful game engine RPG Maker RPG [26], whose success keys to success are present in its simplicity, and transforming the style to the isometric perspective, an editor capable of creating games for versatile, comfortable, simple and easily extensible, will be made, able to bring the developed games to all available platforms, and whose utilization, does not require high programming skills, for making it available for new and novice developers, who could start developing their own and new games, that renew both the adventure genre and isometric style.

It will not be necessary to develop a separate tool, but, taking advantage of the versatility of existing engines that provide cross-platform development of native applications, we can make tools or plugins that extend and provide the necessary mechanics to perform the project.

In the state of art in Spanish: estado del arte, in the chapter number 1, we will study the different options that the engines provides for the project, also the multiple games where the project has defined its style, and has taken references.

Anexo 3. Conclusions (in English)

First of all, videogames can be created using the project, having no knowledge about programming. The user must only learn, roughly, how to use Unity, and learn to deal with the project.

These videogames can have a progress inside their content, setting up milestones, that, when reached, allowing the player to move at different phases of the videogame, or, reach the end of it. Therefore, that content can establish a plot into the game, allowing to develop interactive stories where the player can take part

These phases, or maps, can be created easily using the map editor of the project, generating new phases comfortably and using tools for the build, paint, decoration, and entity positioning. These tools have comfortable menus and interfaces that helps the user to do simply the editing a map task. Overall, these maps generate a logical environment where the entities make their operations.

In these maps, can be performed using the components of this project, an optical effect that gets that, for the textures placed on the map, their isometric view will be the same as the original file view of them.

The building tool helps to create and position cells on the map. These cells maintain an orderly positioning, aligning using a grid on the horizontal plane. Finally, the cells behave as a unit, not overlapping each other.

These cells maintain a Cuboid structure, maintaining a similar with among them, but allowing to modify their height in fragments half its width. Finally, these cells can change their structure becoming a parallelepiped to represent on its upper face, a semi-tilted or tilted surface, depending on whether this tilt rises a full face or half a face.

The painting tool helps to paint the faces of the created cells, allowing to put different textures on the multiple faces of a cell, using the computer cursor, and copying the behavior of a painting brush.

These textures can be standard textures, or isometric textures, a special type of texture that performs a specific texture mapping on the cell, being able to define in this specific texture mapping, the area to be represented, using an editor. This area, is the particular area we want to paint when placing this isometric texture on a cell’s face.

The decorating tool helps to put decorations on the cells, allowing to place these decorations in many different ways, depending on a number of adjustable parameters, and the face of the cell where is being placed.

These different ways of placement allows to: placing decorations on the three visible sides of a cell in isometric perspective, and also, the placement of these decorations to be perpendicular to the horizontal surface, or parallel to the face of the cell where is being placed. These modes allow avoid overlapping elements. As an additional way of placing, it can be established that a decoration can position himself if it has been placed relative to another decoration.

In addition, can be created animated decorations using this tool, allowing the user to set the time of each frame, and the frame sequence that the animation must perform. In addition to this, it can be specified that the decoration must be destroyed when the perform ends, and that the animation must loop the sequence a certain number of loops before being destroyed.

Emotions and effects can be created using the animation functionality, and also, the relative placement of a decoration over the other decoration. These emotions have no assistant or editor, and must be stored as prefabricated object for later instantiation.

Creating entities will be simplified by grouping the previously implemented behaviors, compatibles through an interface of actions, reactions and updates with the rest of the system. The implemented behaviors allows to start sequences, storage and use items, representing these items, teleporting other entities, etc…

The management of the different game system is made by a central class called Game that provides updates and events to its managers as they occur. The most important manager is the map manager, who is able to share events and updates to entities. Other managers that take part on the Game's life cycle are the sequences manager and the emotion manager. Game design allows developers to easily expand itself with new managers.

The event system provides to each participant on the system, a universal interface for inner communication. This communication is made using messages with no origin and destination, and secures the reutilization and dynamic processing of the events.

The game sequences allow progress through the game in a simple way, as it provides with them, a highly extensible advanced sequence editor, able to modify global variables, distribute events, discern different situations, and showing dialog sequences using multiple entities for the conversation. For this, a sequence interpreter is used for showing inside the game, the things that the editor has stored in the sequence.

Game Controller provides to the partaker system objects, entities or managers, the ability to receive a unified input between physical controls or touch controls. Works altogether with the interface manager to secure the user inputs independency between them and the map.