UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Babel: desarrollo de un videojuego de plataformas en Unity

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Salvador Llobet, Ignacio

Tutor: Lluch Crespo, Javier

2014-2015

A Carlos Gómez Morillas,

por su ayuda, su apoyo,

y su eterna paciencia.

Resumen

En este documento tratamos el desarrollo de un videojuego de plataformas y acción, partiendo desde cero, utilizando para ello el motor de juego Unity3D y el lenguaje C# para programar los distintos *scripts*. La generación de los distintos niveles del juego se lleva a cabo mediante un algoritmo de generación procedural. De esta forma, cada partida es completamente diferente a la anterior, dotando al videojuego de una gran rejugabilidad.

Palabras clave: videojuego, plataformas, Unity3D, generación procedural.

Abstract

In this document we deal with the development of a platform-action videogame, from scratch, using Unity3D as game engine and C# language to program the different scripts. Generating different levels of the game is performed by a procedural generation algorithm. This way, each run is completely different from the previous one, giving the videogame a great replayability.

Keywords : videogame, platform, Unity3D, procedural generation.

Tabla de contenidos

1. INTRODUCCIÓN 7
1.1 MOTIVACIÓN 7
1.2 OBJETIVOS 7
2. ESTADO DEL ARTE 9
3. ANÁLISIS DE LAS HERRAMIENTAS 11
3.1 MOTOR DE JUEGO 11
3.1.1 Unreal Engine *11*
3.1.2 CryEngine *11*
3.1.3 GameMaker *12*
3.1.4 Unity *12*
3.2 LENGUAJE DE PROGRAMACIÓN 13
3.3 IDE 14
4. UNITY 16
4.1 GAMEOBJECTS Y PREFABS 16
4.2 SCENE 17
4.3 MONOBEHAVIOUR 18
4.4 GUI 19
5. ANÁLISIS 21
5.1 LA INTERFAZ Y LOS MENÚS 21
5.2 LA JUGABILIDAD 21
5.3 EL UNIVERSO DEL JUEGO 22
5.4 ALGORITMOS DE GENERACIÓN DE NIVEL PROCEDURAL 22
5.4.1 Basado en agentes *22*
5.4.2 Autómata celular *23*
5.4.3 ORE *25*
5.5 ALGORITMO DE POSICIONAMIENTO DE LOS ENEMIGOS 25
5.5.1 Distribución aleatoria *25*
5.5.2 Mediante una función fitness *26*
6. DISEÑO 27
6.1 LA INTERFAZ Y LOS MENÚS 27
6.2 LOS PERSONAJES 28
6.3 GENERACIÓN DE NIVEL 31
7. IMPLEMENTACIÓN 36
7.1 UTILIDADES 36
7.2 MANAGERS 36
7.3 CONTROLADORES 38
7.4 INTELIGENCIA ARTIFICIAL 40
7.5 INTERFAZ DE USUARIO 41
7.6 GENERACIÓN DE NIVEL 43
7.6.1 LevelGenerator *43*
7.6.2 RoomGenerator *43*
7.6.3 EnemyPlace *44*

8. RESULTADOS **46**
9. CONCLUSIONES **48**
9.1 RELACIÓN DEL PROYECTO CON LA CARRERA 48
9.2 TRABAJO FUTURO 49
BIBLIOGRAFÍA **50**
ANEXOS **51**

1. Introducción

1.1 Motivación

Los videojuegos son un mercado en alza; a lo largo del pasado año, sólo en nuestro país, este sector generó unos beneficios de 996 millones de euros [1]. Este boom del mercado de los videojuegos se ha visto acrecentado los últimos años por la explosión de nuevos motores de juego gratuitos, así como la posibilidad de distribuir tus propios videojuegos en multitud de plataformas de manera simple y sencilla: como *Steam* en PC, Linux y Mac OS X; o *Play Store* y *App Store* para el mercado móvil, en Android y iOS respectivamente.

A esta industria prometedora, se suma el hecho que los videojuegos sean el paradigma perfecto de la informática: necesidad de planificación mediante técnicas de ingeniería del software, inteligencia artificial, diseño e implementación de interfaces, algoritmos complejos de generación de terreno o niveles, etc.

No obstante, aun teniendo un gran mercado, el diseño de videojuegos es altamente costoso. La necesidad de diseñar e implementar todo un mundo explorable, así como cada uno de sus niveles, es una tarea que puede alargar cuantiosamente el desarrollo del mismo. Por ello, los estudios de desarrollo llevan años utilizando algoritmos de generación procedural para suplir el trabajo de diseñadores y artistas. De esta forma, los videojuegos se dotan de una aleatoriedad que hace cada partida diferente, así como ahorrarse recursos en su desarrollo.

Así pues, y con el fin de poder incluir estos algoritmos en la generación de nivel del juego, se ha decidido desarrollar un videojuego de plataformas. Donde la generación pseudoaleatoria dota al juego de gran dinamismo, ya que el jugador depende únicamente de su habilidad para conseguir llegar a su objetivo esquivando obstáculos y enemigos por el camino.

Son todos estos elementos, sumados al interés por dedicarse en un futuro a la industria, los que han instigado al autor a escoger este tema como proyecto de final de grado. Buscando acercarse y profundizar en las técnicas de desarrollo de videojuegos, así como en los distintos algoritmos de generación de contenido existentes y aplicables a este campo.

1.2 Objetivos

Como no era de extrañar, el principal objetivo que se perseguía con este proyecto es el desarrollo de un videojuego, así como el aprendizaje de las técnicas y buenas prácticas en la producción del mismo. Para ello, y a fin de comprender realmente todas las etapas del desarrollo, uno de los objetivos secundarios del proyecto fue no utilizar ningún *plugin* [1] de la tienda de *Unity* –exceptuando el que permite integrar Visual Studio 2010 como IDE [2] de *Unity*.

Así mismo, la segunda finalidad del proyecto era el aprendizaje de los distintos algoritmos de generación de contenido procedural o PCG [3] en su aplicación a la generación de niveles en videojuegos. Así como implementar uno de estos algoritmos que sea capaz de:

[1] Extensión que añade alguna funcionalidad al motor
[2] Siglas en inglés de *Integrated Development Environment* (Entorno Integrado de Desarrollo).
[3] Siglas en inglés de *Procedural Content Generation* (Generación de Contenido Procedural).

- Generar un nivel compuesto por salas o habitaciones, interconectadas, o no, entre ellas.
- Generar una ruta de conexiones garantizada entra la sala de entrada y la de salida.
- Colocar cofres, pinchos, escaleras, enemigos o tiendas a lo largo de los niveles.

El producto resultante de este proyecto debía ser un juego escalable. Donde, tanto el número de niveles, como diversidad de enemigos y objetos, fueran fácilmente ampliables en un futuro próximo. Así como poder añadir nuevas habilidades al jugador, etc.

De la misma forma, el producto debía ser coherente con el Documento de Diseño del mismo, redactado previamente, donde se indican todas las características del juego con detalle. Dicho documento se encuentra adjunto en los anexos de esta misma memoria.

2. Estado del arte

Como se ha mencionado en la introducción, el diseño de niveles en los videojuegos es una tarea altamente costosa, más aún en el caso que nos atañe, los videojuegos de plataformas. En este género, el jugador debe llegar de un punto del mapa a otro, valiéndose de distintas habilidades para ello, como: saltar, acabar con enemigos, esquivar proyectiles, recolectar monedas, etc. Todos estos elementos deben ser tenidos en cuenta por el diseñador a la hora de crear cada nivel, pues el jugador debe ser capaz de acabar el nivel, sin que ello le suponga frustrante, pero tampoco se aburra por un exceso de facilidad. En definitiva, es tarea del diseñador conseguir que el jugador experimente la sensación de flujo [2]:

- La tarea puede ser completada.
- Habilidad para concentrarse en la tarea.
- La concentración es posible porque la tarea tiene metas concretas.
- La concentración es posible porque la tarea proporciona retroalimentación inmediata.
- Habilidad para ejercer control sobre las acciones.
- Implicación profunda pero sin esfuerzo, que aleja la conciencia de las frustraciones de la vida cuotidiana.
- La preocupación por uno mismo desaparece.
- El sentido de la duración del tiempo se ve alterado.

Tomemos como caso de estudio dos videojuegos de plataformas: *Super Mario Bros*, desarrollado y publicado por Nintendo [4] en 1985; y *Spelunky*, desarrollado por Derek Yu y publicado en 2008. El primero, está compuesto por ocho mundos, que constan de cuatro niveles cada uno. El jugador debe atravesar cada nivel, de izquierda a derecha, para conseguir llegar al castillo del extremo opuesto del nivel. Cada nivel se compone de una serie de bloques, plataformas móviles, enemigos, monedas y objetos varios que siempre aparecen en el mismo punto del mapa. Toda esta tarea recae sobre el diseñador, que debe estudiar dónde colocar cada ítem y enemigo, así como medir las distancias entre plataformas para que el jugador pueda llegar de una a otra, etc.

Además de estos inconvenientes, el producto final es un videojuego que sólo puede sorprender e inducir al flujo al jugador una sola vez por nivel. Puesto que si se repite el mismo, vuelve a ser exactamente igual a la anterior ocasión que se jugó. En estos juegos, con lo que denominaremos Generación de nivel clásica[5], este hecho se suele compensar añadiendo una ingente cantidad de mundos y niveles. En este juego concreto, un total de treinta y dos niveles de juego.

Por otra parte, el segundo juego de nuestro caso de estudio, *Spelunky*, es un juego que se compone de cuatro mundos, cada uno formado también por cuatro niveles individuales. Cada nivel consta de una cuadrícula de cuatro por cuatro habitaciones interconectadas entre ellas. El

[4] Empresa de desarrollo y publicación de videojuegos japonesa.
[5] Entendiéndose como aquellos juegos en los que la generación de niveles se lleva a cabo por una persona y no mediante un algoritmo.

jugador deberá recorrer el nivel desde una de las columnas de la fila superior de la cuadrícula, hasta la fila inferior, donde se encuentra la salida. Estas habitaciones, se generan de manera pseudoaleatoria, de la misma forma, los ítems y enemigos se colocan en el nivel mediante sencillos algoritmos.

Mediante estas técnicas, se consigue reducir el tiempo de desarrollo, el esfuerzo del diseñador –el cual se reduce a concretar los ítems, enemigos y habilidades, pero no su posición exacta en el nivel–, y la cantidad de niveles totales del juego. Dieciséis niveles totales frente a los treinta y dos del *Super Mario*. Pues una vez acabado, el jugador puede volver a empezar el juego, experimentando una aventura totalmente nueva.

Todas las ventajas y desventajas de uno y otro tipo de generación de nivel se enumeran en la Tabla 1.

	Generación de nivel clásica	Generación de nivel procedural
Ventajas	• Implementación sencilla para el programador. • Entorno determinista fácilmente predecible. • Mayor tolerancia a fallos de programación.	• Tarea simple para el diseñador. • Permite una gran rejugabilidad. • Tiempo de desarrollo reducido. • Aporta dinamismo al juego.
Desventajas	• Dura tarea para el diseñador. • Tiempo y costes de desarrollo elevados. • Nula rejugabilidad.	• Implementación más compleja para el programador. • Sistema indeterminista complejo. • Un error de programación puede generar grandes consecuencias.

Tabla 1: Comparación generación de nivel clásica – procedural.

Así pues, debido al poco tiempo del que se disponía para realizar una tarea tan compleja como es desarrollar un videojuego, situamos nuestro proyecto en el grupo de los juegos de generación de nivel procedural, evitando tener que gastar una gran cantidad de tiempo en el diseño exhaustivo de los niveles que conformaran nuestro juego. Escogiendo además el género de las plataformas puesto que, en este tipo de juegos, un buen diseño de nivel es una de las características más importantes.

3. Análisis de las herramientas

3.1 Motor de juego

Una vez se definidas las características del juego a desarrollar, el siguiente punto era la elección del motor de juego. Este, debe escogerse siempre atendiendo a las necesidades que requerirá el juego, por lo que su elección debe llevarse a un estudio minucioso. A lo largo del presente punto indagaremos entre las características, puntos fuertes y débiles de los cuatro motores de juego más populares del mercado en la actualidad.

3.1.1 Unreal Engine

Desarrollado por *Epic Games*, este motor es el más usado entre los estudios de desarrollo más prestigiosos y conocidos, especialmente para grandes producciones en 3D. Debido en gran medida a sus potentes gráficos, sistema de iluminación dinámica y sistema de partículas. Todo esto ha llevado a los desarrolladores que lo han escogido como motor para sus juegos a desarrollar videojuegos con un nivel de detalle gráfico impresionante, como puedan ser *Bioshock Infinite* o *Gears of War*, desarrollado también por *Epic Games*.

Utilizando C++ como lenguaje de programación, este motor permite desarrollar contenido para multitud de plataformas, Windows, Mac OS X, Android, iOS, etc. Además con su descarga incluye acceso al código fuente del motor, permitiendo personalizar las funcionalidades que más convengan al desarrollador a su proyecto concreto. No obstante, esto mismo requiere un amplio conocimiento, tanto del lenguaje como del propio motor. Y es esa una de sus grandes pegas, la complejidad del motor. Con un editor poco intuitivo, se necesita invertir una cantidad considerable de horas en familiarizarse con el funcionamiento del mismo.

Otra de las grandes pegas que *Unreal Engine* arrastraba hasta ahora era su accesibilidad, pues era necesario pagar veinte dólares al mes además de un 5% de las ventas del juego producido. No obstante, con la última versión, *Unreal Engine 4*, se ha eliminado el pago mensual para tener acceso al motor y su código fuente, de manera que, actualmente, sólo es necesario pagar el 5% de las ventas.

Una vez vistas todas sus características se observa la nula necesidad de un motor tan complejo y profesional, pues el juego a producir es una pequeña producción 2D sin necesidad de grandes gráficos. A esto se suma la complejidad del mismo, pues se disponía de poco tiempo para elaborar el proyecto, que ya de por sí conlleva una cierta dificultad.

3.1.2 CryEngine

Motor con unas capacidades gráficas y de iluminación de última generación, capaces de igualar la potencia de *Unreal Engine*, desarrollado por *Crytek*. Otro motor dedicado a grandes producciones como pueda ser la franquicia *Crysis* o *Star Citizen*, juego todavía en desarrollo. Pese a sus increíbles capacidades gráficas, no hay muchos juegos que hayan sido desarrollados con él. En parte se debe a su falta de versatilidad, ya que está muy enfocado al desarrollo de *shooters* [6], así como su pronunciada curva de aprendizaje.

[6] Videojuegos de disparos y acción, generalmente en primera persona.

Aunque el editor es enteramente gráfico, lo cual permite poder crear contenido a desarrolladores sin conocimiento de programación. Si se quiere incorporar cualquier otra funcionalidad que no proporcione el editor, será necesario tener conocimientos de C++ y LUA.

No obstante, su gran inconveniente frente a su principal competidor, *Unreal Engine*, es su precio. Pese a costar 9.90 dólares al mes, no ofrece el código fuente del motor. De esta forma, la mayoría de estudios de desarrollo se decantan por el primero, pues ofrece características muy similares, pero de manera gratuita hasta el momento de comercializar el juego, además de proporcionar la posibilidad de personalizar el motor al gusto del consumidor.

Nuevamente, *CryEngine* resulta ser un motor demasiado elaborado para un proyecto de nuestras características. Además de la difícil curva de aprendizaje, no cuenta con una comunidad activa demasiado grande, y el hecho de tener que pagar una mensualidad pesa gravemente. Más aún cuando hay otros muchos motores de distribución gratuita.

3.1.3 GameMaker

Desarrollado por *YoYo Games*, éste es el motor más básico y enfocado a un público con escasos conocimientos de programación, con un corto periodo de aprendizaje. Especializado en videojuegos 2D, la mayoría de estudios que escogen este motor para sus juegos, suelen ser proyectos destinados a plataformas móviles, como Android o iOS. *GameMaker* es también uno de los motores más populares, junto con *Unity*, entre los estudios *indie* [7]. Algunos de los juegos producidos con este motor pueden ser *Hotline Miami*, *Gunpoint*, o el propio *Spelunky* visto en el punto anterior.

Es un motor *cross-platform*, es decir, el mismo código utilizado para desarrollar los juegos sirve para varias plataformas. No obstante, su versión gratuita sólo ofrece soporte para desarrollar para *Windows*. Si se desea crear contenido para todas las plataformas es necesario pagar los 559.99 dólares de su versión completa. Además su versión gratuita es poco más que una demo del potencial del motor, pues será necesario pasar por caja si se quiere realizar cualquier proyecto con un mínimo de complejidad.

No obstante, uno de sus puntos fuertes es la gran comunidad de desarrolladores que lo utilizan y lo que ello implica: numerosos tutoriales por internet, así como documentación y foros muy activos para resolver dudas. Además, ofrece un sistema de conexión directa con los principales servicios de monetización, anuncios y analíticas del mercado, por lo que puede ser una buena opción para desarrolladores noveles que quieran empezar a generar ingresos con sus primeros juegos y aplicaciones.

Pese a ser un motor enfocado al tipo de juego desarrollado en el proyecto, las titánicas limitaciones de su versión gratuita nos hicieron plantearnos otras posibilidades que también se adaptaran al trabajo en cuestión sin necesidad de pagar versiones pro. Además del lenguaje para los *scripts* [8], *Game Maker Language*, que pese a ser similar a Java, implicaba la necesidad de aprender un nuevo lenguaje concreto para este propósito.

3.1.4 Unity

[7] Estudios independientes con personal muy reducido, escaso presupuesto y que no cuentan con la ayuda de empresas de publicación.
[8] Programa usualmente simple, generalmente utilizado para interactuar con el sistema operativo o el usuario.

Anteriormente llamado *Unity3D*, el renombrado *Unity*, desarrollado por *Unity Technologies*, es uno de los motores más utilizados entre los nuevos desarrolladores en la industria de los videojuegos, ya que está enfocado a juegos sencillos que no requieran de una gran complejidad. Aun así, también es el motor escogido por grandes empresas para sus pequeñas producciones y juegos que no necesitan de un gran potencial gráfico. Ejemplos de esto último podrían ser *Grown Home* de *Ubisoft* o *Cities: Skylines* desarrollado por *Colossal Order*.

Unity dispone de un editor gráfico con una interfaz muy intuitiva que permite crear contenido incluso sin dotes de programación, no obstante este contenido se verá muy limitado, y habrá que tener unas dotes mínimas de programación. *Unity* permite programar sus *scripts* en *Boo*, Unityscript, un lenguaje muy similar a *Javascript*, y *C#*. Algo más complejo que *GameMaker*, y siendo también *cross-platform*, permite crear juegos más elaborados, por lo que dispone de un amplio abanico de desarrolladores, que van desde juegos para dispositivos móviles, hasta producciones destinadas a consolas como *PlayStation* o *Xbox*. Además cuenta con una grandísima comunidad dispuesta a resolver dudas y publicar tutoriales.

También dispone de una tienda de *plugins* integrada que permite descargar mejoras, tanto gratuitas como de pago, creadas por la comunidad que proporcionan facilidades y funcionalidades concretas para partes concretas del desarrollo.

Además, la versión gratuita ofrece casi todas las funcionalidades completas del motor, reservando para la versión Pro, que se puede adquirir por 75 dólares al mes, o 1500 dólares si se prefiere pagar una única vez, algunas funcionalidades específicas como personalización de la pantalla de inicio, *Unity Analytics* [9], etc.

Por todas estas razones, y debido a los inconvenientes concretos ya mencionados de los demás motores para con el proyecto, se decidió utilizar *Unity* como motor de juego para el trabajo.

3.2 Lenguaje de programación

Como hemos visto, *Unity* nos ofrece la posibilidad de programar los *scripts* del juego en tres lenguajes, es por ello que éste debía ser también un aspecto de análisis para determinar las características de cada uno y poder escoger el que mejor se adaptara a las necesidades del proyecto.

Boo es un lenguaje orientado a objetos con sintaxis similar a *Phyton*. Es un lenguaje muy simple de usar, no obstante, pierde bastante en robustez. Otro de sus defectos es la falta de contenido en internet, habiéndose quedado completamente obsoleto, puesto que en la última versión de *Unity*, *Unity 5*, ya ni siquiera ofrecen soporte al mismo en la documentación. Por todos estos motivos, se descartó el lenguaje rápidamente.

Unityscript es un lenguaje interpretado de sintaxis similar al conocido *Javascript* que, al igual que *Boo*, es uno de los lenguajes más simples que ofrece *Unity* –pues no necesita indicar tipos ni hacer *castings* [10]–. A diferencia de este último, *Unity* sí que ofrece tutoriales y documentación para este lenguaje.

C# es el último lenguaje que nos ofrece *Unity*. Al igual que *Boo*, es un lenguaje orientado a objetos, lo cual ofrece grandes ventajas a la hora de programar videojuegos. Con

[9] Sistema de estadísticas que permite conocer el comportamiento de los jugadores en el videojuego.
[10] Sentencia utilizada para cambiar el tipo de dato del valor de una expresión.

una sintaxis similar a la de *Java*, supone una ventaja considerable, pues este lenguaje se ha ido viendo a lo largo de toda la carrera. Pese a ser el lenguaje más difícil entre los ofertados por el motor, no presenta ninguna complejidad para programadores familiarizados con *C++* o el ya mencionado *Java*. Además es el lenguaje más utilizado en la actualidad por la comunidad de desarrolladores de *Unity*, siendo el más común entre los tutoriales y el principal lenguaje de la documentación del motor. A todo esto se suma el hecho que, mientras *Unityscript* es un lenguaje propio de *Unity* que sólo se utiliza en el motor, *C#* es un lenguaje externo al motor, utilizado por otros motores como *Paradox*.

Por todos estos motivos se decidió escoger *C#* como el lenguaje de *scripting* para el proyecto, dada la familiaridad del autor con lenguajes similares, la potencia del mismo y las recomendaciones recogidas por la comunidad.

3.3 IDE

Unity viene con el IDE integrado *MonoDevelop*. Este IDE es bastante simple de utilizar, con una interfaz intuitiva, además, al venir integrado con el propio motor, nos permite poder utilizar todas sus funcionalidades sin necesidad de preocuparnos por terceros. Con él podremos abrir directamente nuestros *scripts* desde el propio motor clicando dos veces sobre ellos, depurar los *scripts* paso a paso mediante *breakpoints* [11] y conseguir una interacción entre el motor y el IDE mucho mayor. Aun así, cualquier programador que haya trabajado previamente con *Visual Studio* echará en falta muchas funcionalidades de éste. No obstante, es una muy buena opción para proyectos que no tengan una gran extensión, o para estudios que no dispongan de los fondos necesarios para adquirir las versiones de pago de *Visual Studio*.

Visual Studio es un IDE desarrollado por *Microsoft*. Considerado uno de los mejores IDE del mercado, ofrece un sinfín de funcionalidades para los desarrolladores de cualquier tipo de software. No obstante, en lo que a integración con el motor que nos atañe se refiere, ofrece varios conflictos importantes. El primer detalle que ahuyenta a los desarrolladores que lo quieren utilizar para sus juegos es la imposibilidad de abrir un *script* en *Visual Studio* desde el propio motor –siempre que sea la versión gratuita de *Visual Studio*, es decir, su versión *Express*, pues las versiones *Pro* sí que ofrecen esta integración–, teniendo que abrir el proyecto desde *Visual Studio* «a mano». Otra imposibilidad es la de depurar los *scripts* desde el IDE paso a paso, lo que supone una gran desventaja a la hora de solucionar errores en nuestro código. No obstante, *Visual Studio* detecta mejor los errores, resaltándolos en el código, y ofrece un mejor sistema de autocompletado, permitiendo que la programación sea más fluida y rápida.

A pesar de todas estas desventajas, *Visual Studio* ofrece una serie de *plugins* de manera gratuita para integrar el IDE con *Unity*. Con estos *plugins*, podremos realizar todas las funcionalidades que ofrece *MonoDevelop* desde *Visual Studio* sin ningún tipo de problema. Podremos abrir los *scripts*, depurar nuestro código desde el IDE, generar los ficheros del proyecto desde *Unity*, así como disfrutar de todas las funcionalidades que ya ofrece de por sí éste programa. Además, puesto que la UPV dispone de un acuerdo *DreamSpark* con *Microsoft* mediante el cual se puede descargar su software de manera gratuita para la comunidad estudiantil, tenemos la posibilidad de trabajar con este IDE en su versión *Premium*. Es por ello que se ha escogido este software como IDE, pues disponiendo de la versión *Pro* gratuitamente,

[11] Punto en el cual se detiene el flujo habitual del programa para verificar valores, el correcto funcionamiento del flujo, etc.

y añadiendo el *plugin* para integrarlo con el motor de juego, tenemos un entorno realmente potente para la programación del proyecto.

4. Unity

Hasta ahora hemos visto las distintas herramientas más utilizadas para desarrollar videojuegos, comparándolas entre ellas para escoger la que más se adecúe a nuestras necesidades. A lo largo de este punto pasaremos a explicar en profundidad las características que ofrece el motor de juego escogido para el proyecto.

4.1 GameObjects y Prefabs

Los *GameObjects* son la unidad básica en la que se fundamenta *Unity*. Estos objetos representan todo el contenido de una escena, desde objetos tangibles del mundo como personajes, fondos e ítems varios, hasta objetos intangibles como la propia cámara, o los botones de la interfaz de juego. Esto quiere decir que cualquier objeto que esté en escena –tanto activo como inactivo–, será un *GameObject* necesariamente. Como los objetos que son, cada *GameObject* puede tener un único objeto padre, funcionalidad de gran utilidad que puede servir para agrupar diversos objetos en una única referencia, posición, etc.

No obstante, un *GameObject* no deja de ser un mero objeto en escena sin comportamiento alguno. El comportamiento de cada objeto, dependerá de sus componentes, por lo que dos *GameObjects* pueden tener funcionalidades completamente distintas. Los componentes son clases proporcionadas por *Unity* que añaden una funcionalidad concreta al objeto que las contiene. Aun así, si queremos añadir una funcionalidad que no venga dada por *Unity* siempre podremos crear un *script* y añadirlo al *GameObject* para incorporarla. A continuación detallamos los componentes básicos que se han utilizado en el proyecto:

- *Transform*: Por defecto, y puesto que los *GameObjects* son objetos en escena, todos los nuevos *GameObjects* que se crean vienen con este componente añadido al objeto. El componente *transform* determina la posición, rotación y escala del objeto en cuestión. Esta clase nos ofrecerá, entre otros, métodos para desplazar objetos –muy útiles para el movimiento del personaje y los enemigos–, girarlos con respecto a un eje concreto, o comprobar posiciones. Cabe destacar que, si el objeto tiene otro objeto padre, la posición del *transform* será una posición relativa. Mientras que si no hay ningún objeto por encima en su jerarquía, serán posiciones globales de la escena.
- *Collider*: Los *colliders* pueden ser de distintos tipos. Cada uno, generará un área de inclusión con la forma del tipo que sea el componente. En los *colliders* en 2D –que son el tipo que utilizamos en el proyecto, pues es un juego en dos dimensiones–, pueden ser: línea, cuadrado, círculo o polígono –donde el usuario define la cantidad de vértices y geometría–. Estos componentes, dotan al objeto de la posibilidad de colisionar con otros *GameObjects* –siempre que tengan *colliders* también–. También se puede utilizar como *trigger*, evitando las colisiones, pero disparando eventos cuando dos *colliders* se superpongan.
- *Rigidbody*: El *rigidbody* hace que un objeto se vea afectado por el motor de físicas de *Unity*. Este componente otorga la capacidad al objeto de recibir fuerzas que le propulsen, verse afectado por la gravedad, etc. Esto es necesario para todos aquellos objetos que quieran simular un comportamiento natural del mundo real en el juego, como pueden ser los personajes, o los proyectiles que se disparan. Es importante destacar la diferencia entre mover un objeto mediante el desplazamiento de su

transform y la aplicación de fuerzas al mismo. Mientras que el primero sirve para movimientos precisos como podría ser el del control de los personajes, el segundo es especialmente útil para proyectiles, donde ir calculando el desplazamiento paso a paso sería del todo ineficiente. En estos casos es recomendable añadir un *rigidbody* al objeto, indicarle una masa y una fuerza, y dejar que el motor de físicas haga el resto.

- *Sprite renderer*: Este componente permite asignar una imagen al objeto. Dicha imagen puede cambiarse a lo largo de la ejecución. También ofrece la posibilidad de asignarle una *sorting layer* a la misma. Es decir, una capa que sufre una jerarquía. Esto es especialmente útil para juegos en 2D, puesto que cuando dos objetos se superponen es importante saber qué imagen debe mostrarse por encima de la otra.
- *Animator*: Hay que hacer una distinción entre el *animator* componente, y el *animator* controlador. El primero es el componente que se añade al *GameObject*, que contiene una referencia al *animator controller* que le corresponda. El segundo es una estructura, similar a las máquinas de estados, en la que se crea una máquina donde cada estado corresponderá a una animación. Con la máquina hecha, el controlador se podrá asignar al componente *animator* de un objeto, permitiendo a dicho objeto variar su animación dependiendo de las circunstancias especificadas en la máquina. Estos componentes serán necesarios para todos aquellos objetos que vayan a tener una o varias animaciones entre las que iterar.
- *Audio source*: Dota al objeto de la capacidad de reproducir un clip de audio. Ya sea música o efectos de sonido, este objeto será necesario para emitir cualquier audio. No obstante, para poder percibirlo, deberá existir un objeto en la escena que cuente con el componente *audio listener*. Por defecto, cuando se crea una cámara, ya viene con el componente *audio listener* incorporado, de manera que no deberemos preocuparnos por ello.

Cabe decir que todos los componentes de los *GameObjects* no dejan de ser clases, cuyas variables pueden ser modificadas en cualquier momento. Tanto desde un *script* como desde el propio editor de *Unity* [12].

Por otro lado, los *prefabs* son *GameObjects* ya configurados previamente, que pueden estar, o no, en una escena. Los *prefabs* sirven para, una vez creado un *GameObject* que vaya a ser utilizado en varias ocasiones, guardar su configuración. De esta manera podremos instanciar un objeto complejo con una sola instrucción, sin necesidad de ir instanciando sus componentes una a una. Un ejemplo práctico del uso de los *prefabs* en un juego sería con los enemigos. Una vez creado el objeto de un enemigo con todas sus características, se asigna al *prefab*, de esta manera podremos instanciarlo en los niveles que corresponda, tantas veces como sea necesario, creando X *GameObjects* que actuarán según se haya configurado el original, pero de manera independiente.

4.2 Scene

A lo largo de la explicación de los *GameObjects* se han ido haciendo referencias a las escenas del juego. Las escenas, o *scenes* en el motor, son un entorno compuesto por *GameObjects* que representa un espacio del juego. Es decir, un juego en *Unity* se compone de una sucesión de *scenes*; cada una con sus *GameObjects*.

[12] En el editor sólo podrán ser modificados aquellos valores que sean públicos.

Los objetos que contenga una escena serán cargados en el momento de cargar la escena, y destruidos al cargar otra. No obstante, se puede indicar en el *script* de cualquier objeto que no se destruya al cargar otra escena. Aunque esto supone tener que llevar un seguimiento de aquellos objetos que permanecerán siempre cargados para evitar volver a cargarlos y duplicarlos, también conlleva menos accesos a memoria, lo que se traduce en mayor rapidez de carga. Cargar una escena supone un gasto computacional importante, puesto que hay que cargar todos los objetos y componentes que la constituyen. Por ello es importante estudiar bien cómo dividir el juego en escenas, evitando estar constantemente cargando una u otra.

Volviendo al caso de estudio del punto 2, en *Super Mario Bros*, el juego constaría de una escena para el menú principal, y una escena para cada nivel del juego. Por lo que el juego constaría, como mínimo, de treinta y tres escenas. Por el contrario, en el otro juego del caso de estudio, *Spelunky*, al ser un juego que, igual que el nuestro, utiliza una generación de nivel procedural, sólo necesita de tres escenas: una para el menú principal, otra para el nivel –pues cada vez que se cargue el algoritmo generará un nivel diferente–, y otra para la transición entre niveles, dado que en este juego, hay una pequeña habitación entre nivel y nivel.

4.3 MonoBehaviour

Ésta es la clase base de la que heredan todos los *scripts* de *Unity*. Similar a la clase *Object* de *Java*, cuando creamos un nuevo *script*, hereda de *MonoBehaviour* por defecto. No obstante, podemos cambiar esto si no queremos dicha herencia. Aunque aquellos *scripts* que quieran acceder a las funciones básicas del motor, deberá heredar de esta clase o, al menos, heredar de una clase que herede de *MonoBehaviour*.

Como ya se ha dicho, *MonoBehaviour* ofrece los métodos más básicos de *Unity*, a continuación pasamos a explicar brevemente aquellos que influyen en el flujo de ejecución y se han utilizado en el proyecto. Todos estos métodos se encuentran perfectamente documentados en la API[13] de *Unity*, por lo que no ahondaremos demasiado en ellos:

- *Awake*: Este método es llamado cuando el objeto al que pertenece el *script* es instanciado o cargado en memoria. Tanto si el objeto está activo, como si no. Esto es importante, pues este método viene especialmente bien para inicializar variables dentro del objeto. De esta forma, aseguramos que ciertos valores estén disponibles para otros *scripts* que pudieran necesitarlos desde su inicialización. *Awake* sólo se ejecuta una única vez.
- *Start*: Método parecido a *Awake*, con la diferencia que sólo se le llama cuando el objeto es instanciado, siempre que el objeto esté activo. Es decir, si el objeto instanciado está inactivo, no se llamará a *Start* hasta que se active. Al igual que *Awake*, sólo se ejecuta una vez. En este caso, la primera vez que después de ser instanciado el objeto está activo.
- *OnEnable*: Como su nombre indica, este método se ejecuta cada vez que el objeto se activa. A diferencia de *Start*, no se ejecuta una única vez cuando el *GameObject* se activa después de su instancia, sino que se llama cada vez que el objeto pasa de estar inactivo a activo.

[13] Interfaz de programación de aplicaciones (del inglés *Application Programming Interface*). Conjunto de métodos que ofrece una biblioteca para ser utilizados por otro software.

- *Update*: Mientras el *GameObject* esté activo, el método *Update* es llamado constantemente. En concreto, una vez por *frame* [14]. Es por ello que se debe tener cuidado con las operaciones que se introducen en el método, pues serán ejecutadas constantemente. Este método es una buena opción para controlar los valores de la entrada del jugador, desplazamiento de personajes, etc.

Obviamente, existen muchos más métodos que *MonoBehaviour* nos ofrece y se han utilizado a lo largo del proyecto, como: *OnTriggerEnter/Exit/Stay2D*, *OnCollisionEnter2D*, etc. No obstante, como no influyen tan decisivamente en el flujo de ejecución de los *scripts*, no los veremos; pues todos estos métodos están perfectamente explicados en la documentación de *Unity* [3] y sería contraproducente explicar cada método utilizado. Si observamos el diagrama 1, podremos observar el flujo de ejecución de los distintos métodos de un *script* que herede de *MonoBehaviour*.

También habría que destacar las corutinas. Una corutina es un método que puede suspender su ejecución durante unos segundos determinados, o hasta que ocurra la sentencia indicada. Como estos métodos se ejecutan en paralelo al resto del *script*, podemos utilizarlas para implementar tiempos de espera entre ataques o habilidades, para evitar que el jugador o un enemigo puedan repetir un ataque o habilidad constantemente sin una cierta espera entre ambos. No obstante, las corutinas pueden ser interrumpidas mediante la función *StopCoroutine*, de esta forma tenemos un control mayor sobre las mismas, evitando que alguna termine de ejecutarse si ocurre alguna situación concreta. Utilizar corutinas es altamente recomendado, ya que tienen un coste computacional bastante bajo y, por lo general, pueden sustituir a bastantes instrucciones que irían en el método *Update* [4].

4.4 GUI[15]

Hasta hace un año, hacer la interfaz de usuario era una de las tareas más engorrosas de *Unity*. Careciendo de un sistema gráfico de diseño de interfaces, era necesario programar desde los *scripts* cada elemento de la interfaz –con la excepción de los textos–, incluyéndolos en el método *OnGUI*, que es el encargado de dibujar las interfaces –el cual destruye toda la interfaz para volver a dibujarla cada vez que se ejecuta el método–. Esto presentaba el inconveniente de no poder ver cómo quedaba la interfaz que se estaba implementando hasta ejecutar el programa y verla dentro del juego. Aunque existían multitud de *plugins* que aportaban esta funcionalidad, no ofrecían una integración del todo correcta, pues eran programas externos creados por la comunidad. Un ejemplo de estos *plugins* podría ser *NGUI*, con un coste de 95 dólares y desarrollado por *Tasharen Entertainment*.

Con la llegada de la versión 4.6 del motor, todo esto cambió. Se incorporó un sistema gráfico de diseño y ahora es mucho más simple crear interfaces gráficas. Actualmente, los componentes de las interfaces se pueden crear desde el propio editor de *Unity*, con un sistema *drag-n-drop*[16] así como indicar su posición en pantalla, escala, colores, funcionalidad, etc.

---

[14] Unidad mínima, en forma de imagen estática, en la que se puede descomponer un vídeo. En el caso de los videojuegos, compone cada refresco de pantalla.
[15] Interfaz gráfica de usuario (del inglés *Graphical User Interface*). Sistema que utiliza componentes gráficos para representar información en la interfaz. Ejemplo: en un juego, se indica el dinero, la vida del jugador, etc.
[16] Arrastrar y soltar.

Esto se consigue gracias a la figura del *Canvas*. Esta estructura no es otra cosa que el espacio en el que la interfaz se coloca en la escena. Este objeto también actúa como padre de todos los elementos de la UI[17]. Podemos ajustar el *Canvas* para que se ajuste a la cámara del juego y la vaya siguiendo, o para que sea estático acorde a unas coordenadas de la escena. En este último caso habrá que tener en cuenta que los objetos de la UI se renderizarán aunque no haya una cámara en la escena.

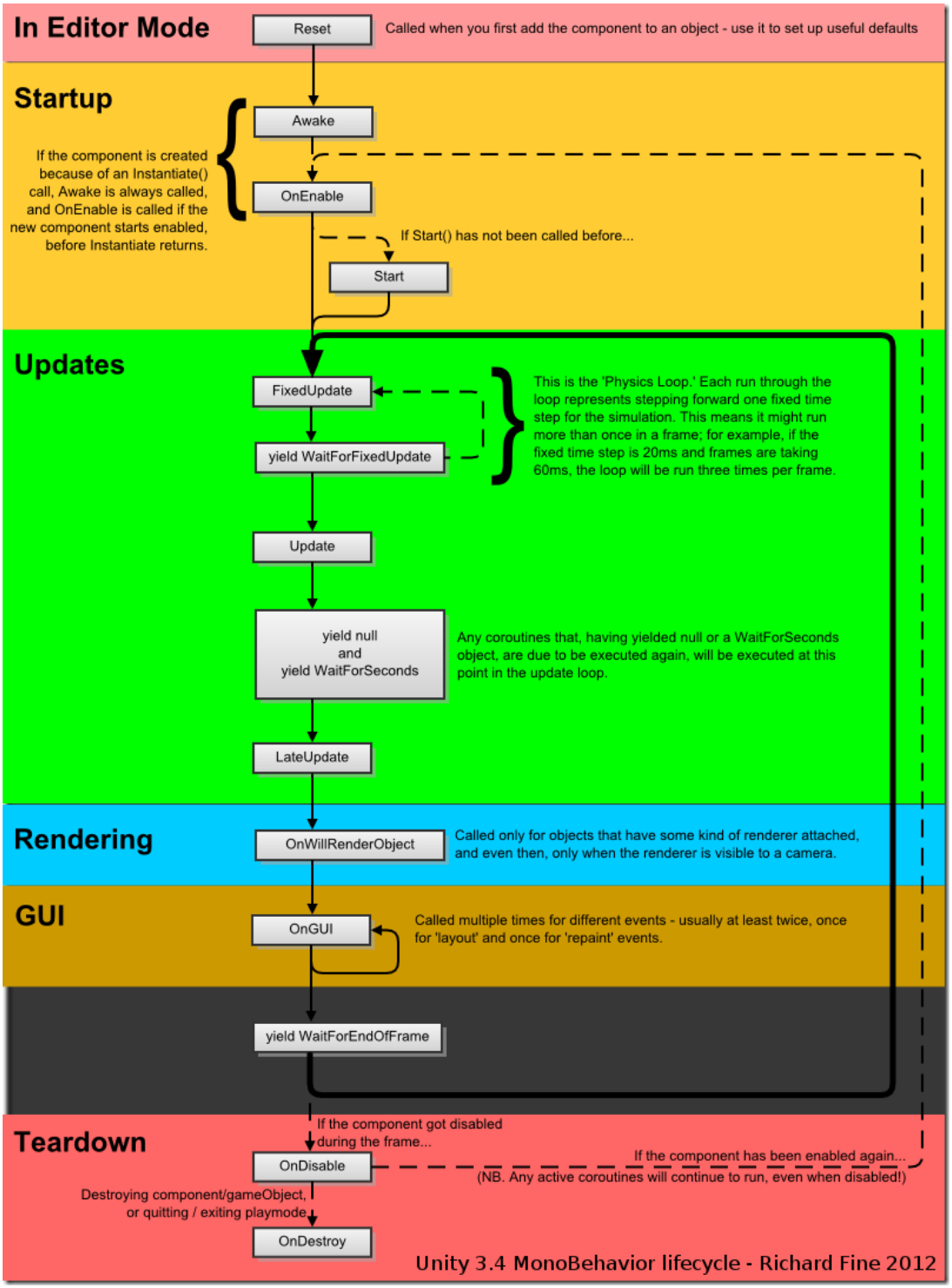


Diagrama 1: Ciclo de vida de *MonoBehaviour*

[17] Interfaz de usuario (del inglés *User Interface*).

5. Análisis

Puesto que un videojuego es un producto software de gran complejidad, su desarrollo debe ser motivo de un estudio exhaustivo. Una vez redactado el GDD [18], se deben analizar todas las características del juego, dejándolas claras, para su posterior diseño e implementación de la solución diseñada.

Si bien hasta ahora hemos ido estudiando las diferentes herramientas que ofrece el mercado actual para desarrollar videojuegos, así como un análisis en profundidad del funcionamiento del motor a utilizar, a continuación analizaremos las necesidades básicas de nuestro juego; sin entrar en detalles de diseño, que veremos en el punto 6.

5.1 La interfaz y los menús

Dado que Babel es un juego simple en lo que a jugabilidad se refiere, la interfaz no requiere mostrar demasiada información en pantalla. La única información que el jugador necesitará constará de su número de vidas, el número de cuerdas de las que dispone, y el dinero actual. No obstante, como el jugador podrá comprar objetos que modificarán sus habilidades, sería una buena práctica incluir un recordatorio de los objetos que se han comprado como muestra de las habilidades de las que se dispone.

Por lo que respecta a los menús de juego, no son extensos. Al ser un juego con *permadeath* [19], no habrá ningún tipo de persistencia, por lo que podemos ahorrarnos perfiles de juego, accesos a continuar una partida previa, etc. De esta forma las únicas necesidades del usuario en los distintos menús serán empezar una partida; acceder a las opciones del juego para configurar la resolución, *framerate* [20], activar o desactivar la *vsync* [21], o cambiar el volumen del audio; ver los controles del juego –los cuales sólo se podrán ver desde el menú de pausa–; visualizar los créditos; o salir del juego. En el menú de pausa –el cual se activa dentro de la partida cuando el jugador decide pausar el juego–, se añadirán las opciones de reanudar la partida, y volver al menú principal; sacrificando la opción de ver los créditos.

5.2 La jugabilidad

Dentro de una partida, el jugador debe ser capaz de poder interactuar libremente con todos los elementos del escenario. Así pues, podrá caminar, saltar, atacar a los enemigos, lanzar cuerdas, trepar por cuerdas o escaleras, realizar un pequeño sprint o *dash*, abrir cofres, recolectar monedas y dinero, y comprar ítems en las tiendas. De la misma forma, el jugador también tendrá capacidad para pausar el juego e interactuar con el menú de pausa. Cabe decir que estas son las capacidades básicas por defecto, dado que los ítems que el jugador podrá comprar modifican algunas habilidades, éstas pueden verse ampliadas a medida que avanza la partida.

Los enemigos, al igual que el jugador, también deben tener la capacidad de realizar algunas de estas acciones. No obstante, su capacidad de interactuar con el entorno debe ser

[18] Documento de diseño de juego (del inglés *Game Design Document*).
[19] Muerte permanente.
[20] Tasa de refresco de la pantalla en hercios.
[21] Sincronización vertical. Limita los fotogramas por segundo para evitar *tearing*.

mucho más limitada. No colisionar con otros enemigos, no poder acabar con otros enemigos, etc.

Tanto los enemigos, como el propio personaje controlado por el jugador tendrán unas animaciones concretas y diferenciadas, así como unos sonidos específicos para cada una de sus habilidades.

5.3 El universo del juego

El universo de Babel consta de un mundo compuesto por cinco niveles, todos con una misma ambientación. Cada nivel debe generarse proceduralmente al inicio del mismo, de manera que siempre sea diferente.

Al igual que ocurre en *Spelunky*, los niveles constan de una cuadrícula de cuatro por cuatro habitaciones interconectadas, o no, entre ellas. Así pues el algoritmo deberá encargarse de trazar una ruta segura, desde la entrada hasta la salida, que garantice que todas las habitaciones de la ruta están interconectadas –la partida tiene que poder acabarse–. Una vez trazada la ruta garantizada, se deberán generar las habitaciones de manera pseudoaleatoria.

También habrá habitaciones especiales que no se vean modificadas nunca –la tienda y la sala del tesoro–, las cuales tendrán un porcentaje de probabilidad de aparición concreto. Estas habitaciones nunca podrán formar parte de la ruta de salida garantizada, por lo que existirá la posibilidad que, en caso de aparecer en un nivel, no sean accesibles, pues haya habitaciones de por medio que no estén conectadas con estas.

De igual manera, tanto el número de enemigos, como su posición en cada habitación, se calcularán de manera pseudoaleatoria. Tarea que llevará a cabo otro algoritmo diseñado específicamente para tal fin.

5.4 Algoritmos de generación de nivel procedural

Pasemos ahora a analizar los distintos algoritmos existentes para generar los niveles en tiempo de ejecución. Hay que indicar que existen infinidad de algoritmos para este propósito, pero nosotros hemos analizado aquellos que más se adecuaban a nuestro propósito. Recordemos que buscamos un algoritmo que genere habitaciones individuales para luego juntarlas mediante otro algoritmo mayor.

5.4.1 Basado en agentes

Los algoritmos de generación procedural basados en agentes suelen utilizarse para generar mazmorras. La forma de la mazmorra, dependerá en gran medida del comportamiento del agente. Un agente «ciego», es decir, completamente aleatorio, generará mazmorras caóticas, llenas de cruces y salas solapadas. Por el contrario, un agente con una cierta capacidad de visión, evitará hacer cruces.

Estos algoritmos se basan en una cuadrícula de bloques en la que se libera uno o varios agentes en una posición que puede ser aleatoria, o no –consideraremos que se libera un único agente–. Dicho agente, llamado agente «excavador», o *digger* en inglés, tiene la capacidad de moverse en cualquier dirección y sentido –arriba, abajo, izquierda y derecha–, siempre que no supere un número máximo de desplazamientos que se le asigna previamente. Cada vez que el agente se desplaza una posición, se cambia el bloque que ocupa el agente por un espacio hueco. Después de cada desplazamiento, el agente tiene una cierta probabilidad de modificar su trayectoria, haciendo que gire hacia otra dirección, enredando la mazmorra.

En cada desplazamiento, el agente tiene una probabilidad de colocar una sala, de un tamaño aleatorio, en dicha posición. A partir de aquí interviene el comportamiento con el que se haya dotado al agente, pues un agente ciego seguirá desplazándose y colocando salas mientras que no agote sus movimientos permitidos (ver la Figura 1). Mientras que un agente con inteligencia comprobará si la nueva sala va a colisionar con un pasillo o sala anterior antes de colocarla (ver la Figura 2) [5].

Figura 1: Ejecución de un agente ciego

Figura 2: Ejecución de un agente inteligente

Como se puede apreciar en las imágenes, este algoritmo está enfocado a generar las plantas de las mazmorras. Es decir, no se tiene en cuenta la gravedad, pues el terreno generado es el suelo del nivel, no su alzado. Sin embargo, en un juego como el nuestro, que contempla los niveles desde su alzado, donde para llegar de una sala inferior a una superior habrá que llegar saltando, ésta no sería la mejor opción. Dado que nuestro personaje tiene una capacidad de salto limitada, no podría alcanzar las habitaciones superiores, lo que conllevaría que el nivel no sería superable de ninguna de las maneras.

Es esta falta de control sobre las habitaciones generadas las que hicieron descartar este método, pues necesitábamos un algoritmo que nos permita generar niveles de manera procedural, pero comprobando que eran practicables por el jugador, teniendo en cuenta sus limitaciones de salto y movimiento.

5.4.2 Autómata celular

Un autómata celular es un modelo matemático para un sistema dinámico, compuesto por un conjunto de celdas o células que adquieren distintos estados o valores. Estos estados son alterados de un instante a otro en unidades de tiempo discreto. De esta forma, este conjunto de células logran una evolución según una determinada expresión matemática, sensible a los valores de las células vecinas.

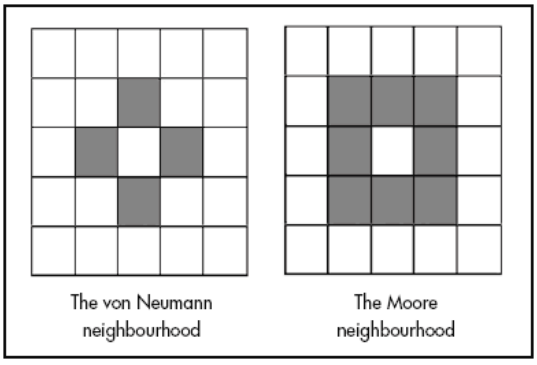


Figura 3: Tipos de vecindarios de los autómatas celulares.

Los vecindarios definen qué celdas, alrededor de una celda concreta, afectan al futuro estado de la misma. Como se puede apreciar en la figura 3, estos son los dos tipos más comunes: el Von Neumann y el vecindario de Moore.

Utilizando cualquiera de estos vecindarios –para el ejemplo de la figura 4 se utilizó el vecindario de Moore–, dependiendo de las necesidades del proyecto, sólo se necesitan cuatro parámetros de control:

- Porcentaje de celdas que sean muros.
- Número de generaciones del autómata celular.
- Valor de vecindario por el cual una celda se convierte en muro.
- Número de celdas del vecindario[22].

Una vez fijados los parámetros de control, el algoritmo empieza su ejecución. Con la cuadrícula vacía, se aplica a todas las celdas una probabilidad de convertirse en muros –el primer parámetro de control–. Con esto se consigue una distribución uniforme de muros en la cuadrícula. Una vez distribuidas, se libera el autómata celular en una posición cualquiera del nivel. El autómata irá convirtiendo en muro todas las celdas en cuyo vecindario haya un número mayor de muros que del valor umbral indicado en el tercer parámetro de control. De la misma forma, destruirá los muros, convirtiéndolos en celdas vacías, de todas aquellas celdas en cuyos vecindarios haya menos muros que el valor umbral [6].

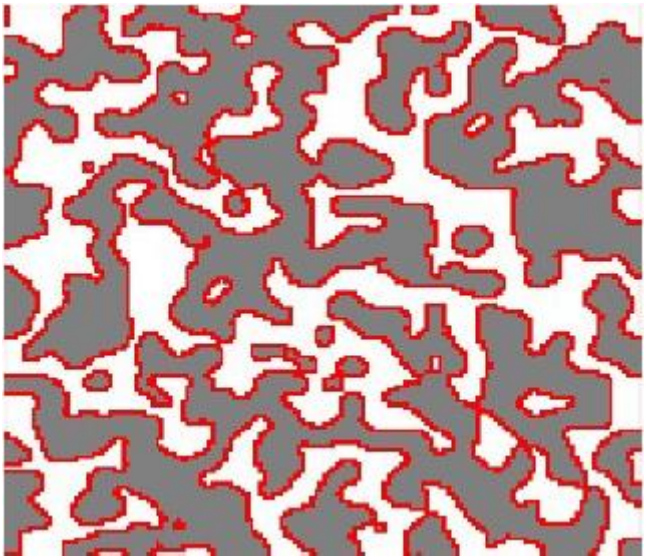


Figura 4: Resultado del algoritmo en una cuadrícula de 150 x 150 celdas. Los muros se representan mediante rojo y blanco, mientras que el gris representa celdas vacías.

Este algoritmo genera unos niveles muy similares a cuevas. La mayoría de salas suelen estar interconectadas, dado que las celdas se transforman en uno u otro valor en función de sus vecinos. No obstante, el algoritmo no garantiza ningún tipo de control sobre la conexión con las habitaciones vecinas. Recordemos que este algoritmo se aplicaría a cada habitación o sala de la cuadrícula del nivel, por lo que sería necesario controlar las conexiones entre habitaciones.

Además, nuevamente sufrimos el mismo problema que con el algoritmo basado en agentes: este algoritmo está enfocado a generar la planta de un nivel, y no su alzado. La

[22] Es decir, el tipo de vecindario que utiliza el autómata.

capacidad de salto de nuestro personaje podría no cubrir las paredes entre las diferentes grutas, por lo que sería imposible ascender hasta la salida, que se sitúa en las posiciones más elevadas del nivel.

5.4.3 ORE[23]

Este es el algoritmo más simple de generación procedural. Si bien también es el que más depende de un diseñador, es el que nos ofrece mayor control sobre el terreno generado.

Diseñado por Mawhorter y Mateas [7], el algoritmo se basa en utilizar *chunks*[24] prediseñados de una biblioteca e ir ensamblándolos para conformar el nivel. La ejecución del algoritmo es simple: se escoge una posición aleatoria, o no, del nivel y se instancia un *chunk* cualquiera; una vez se pasa a la siguiente posición vacía, se escoge un *chunk* compatible –pues hay que tener en cuenta el tamaño del mismo acorde al de la posición, y los parámetros de control–; si el *chunk* es compatible, se encaja en la posición correspondiente, y el algoritmo repite esta iteración para todas las posiciones hasta que el nivel quede completado [5]. No obstante, el algoritmo no define ningún parámetro de control por defecto, siendo la única restricción el tamaño del *chunk*, aunque el algoritmo es compatible con estos sistemas de control.

Aunque sigue requiriendo de un diseñador que cree los *chunks* previamente, este algoritmo es el que mayor control ofrece sobre el nivel generado. Es este control sobre el diseño previo de los *chunks* el que nos permite adaptar el nivel generado a nuestras necesidades. Pues, a diferencia de los algoritmos anteriormente vistos, nos permitirá decidir si el terreno generado será la planta o el alzado del nivel.

Por todo el control que nos ofrece sobre el nivel, se seleccionó este algoritmo para que fuera el encargado de generar los niveles. No obstante, y para hacerlo más aleatorio y dinámico, se le hicieron algunas modificaciones de diseño que veremos en profundidad en el punto 6.

5.5 Algoritmo de posicionamiento de los enemigos

Puesto que cada nivel es generado de manera pseudoaleatoria en tiempo de ejecución, y no podemos saber la forma que tendrá el terreno generado, será imprescindible utilizar un algoritmo capaz de colocar los enemigos a lo largo del nivel por nosotros. Las necesidades que debe cubrir este algoritmo son simples: seleccionar uno o varios enemigos y distribuirlos a lo largo de las distintas habitaciones que conforman el nivel.

5.5.1 Distribución aleatoria

Como su nombre indica, este algoritmo es el más aleatorio. El algoritmo se limita a escoger posiciones al azar de un nivel dado –en nuestro caso, una habitación dada–, y siempre que sea una posición vacía y su posición inmediatamente inferior sea un muro, se coloca al enemigo. El tipo de enemigo que se coloca en cada posición, en caso de disponer de varios, también se escoge de manera aleatoria. Como sistema de control se puede indicar al algoritmo cuántos enemigos se desean para cada nivel, dar una cierta probabilidad de aparición para las posiciones escogidas, o delegar todo el control en el algoritmo, dejando que sea él el que lo determine.

---

[23] *Occupancy-regulated extension*.

[24] En inglés «trozo». Hace referencia a una parte de un nivel o trozo del mismo ya prediseñado. Puede contener muros, enemigos, objetos, etc.

Si bien éste sería el método más simple de implementar y que menos trabajo requiere, cabe la posibilidad que el algoritmo ponga enemigos en posiciones remotas por las que el jugador tiene pocas probabilidades de pasar, consiguiendo que el enemigo de dicha posición sea un gasto computacional inútil. Llevando un breve estudio del comportamiento de los jugadores dentro de los niveles, y atendiendo a los ítems y objetos de que dispondrá nuestro nivel, se podría llegar a un mejor algoritmo para colocar los enemigos en posiciones más relevantes.

5.5.2 Mediante una función fitness[25]

Otra solución a este problema es posicionar a los enemigos en el nivel en función de su valor *fitness* o valor de idoneidad. En estos algoritmos, al igual que en el anterior, se estudian las posiciones donde se podría colocar un enemigo –celda vacía con celda de muro debajo de la misma–. La diferencia radica en que a cada posible posición de los enemigos, se le asigna un valor *fitness*, el cual se extrae de una fórmula concreta formulada por el diseñador o el programador. El valor de esta función indica el nivel de idoneidad de dicha posición para contener un enemigo en el nivel. Una vez se han recorrido todas las posibles posiciones y asignado un valor *fitness* a cada una de ellas, sólo queda escoger el número de enemigos del nivel. Con el número de enemigos, se escogen ese mismo número de las posiciones con el valor *fitness* más elevado, es decir, las posiciones más idóneas para contener un enemigo, y se colocan los mismos en ellas.

De esta forma podemos conseguir distribuir los enemigos a lo largo del nivel conforme a unos parámetros dados, aun a pesar de no saber qué forma tendrá el nivel generado. Con esta distribución se pueden plantear retos mayores al jugador que distribuyendo los enemigos aleatoriamente por el mapa, consiguiendo una mayor inmersión del jugador en la sensación de flujo [2], puesto que la posición dependerá de los parámetros que conformen la función *fitness*. Por todo ello escogimos este algoritmo para colocar los enemigos en nuestro proyecto.

[25] Idónea.

6. Diseño

Una vez analizadas todas las necesidades del juego, debemos pasar a pensar y diseñar las soluciones a las mismas que luego implementaremos. Esta tarea es de una complejidad importante, y debe dedicársele suficiente tiempo a la misma, pues un diseño ineficiente llevará a una posterior implementación deficiente y un producto mediocre.

6.1 La interfaz y los menús

El diseño de la interfaz es uno de los aspectos más importantes del juego. El jugador deberá interactuar con ella constantemente, tanto para iniciar una partida, como para cambiar la configuración de la misma. La interfaz debe ser intuitiva, sencilla y clara; por ello se siguieron las normas que definió Shneiderman para el diseño de una buena interfaz [9]:

- Diseño consistente y con coherencia: El diseño entre los distintos menús debe ser similar, utilizando la misma terminología y con una estructura común.
- Accesos directos o atajos: Los *shortcuts* [26] permiten a los usuarios experimentados evitar pasos intermedios, interactuando con la interfaz de manera rápida y fluida. De esta forma evitaremos que un usuario que ya conoce el sistema se frustre viéndose obligado a pasar por las mismas etapas que un usuario novel.
- Toda acción debe tener un *feedback*[27]: Toda acción llevada a cabo por el usuario debe ofrecer una reacción del sistema que indique que se ha llevado, o se está llevando, a cabo la petición del usuario.
- Acciones secuenciales: Si una acción requiere de varios pasos, se deben indicar los mismos; así como el paso actual en el que se encuentra el usuario. Las secuencias deben tener un comienzo, un intermedio, y un final.
- Gestión de errores sencilla: En la medida de lo posible, diseñar el sistema para que el usuario no pueda cometer un error grave. Si se comete uno, el sistema debe detectarlo y ofrecer una solución sencilla y comprensible.
- Retroceso de las acciones: Toda acción debe poder deshacerse, aliviando con ello la ansiedad del usuario, que sabrá que todo error puede ser corregido a posteriori. Con esto también se induce a la exploración de las opciones y la interfaz, pues el usuario no teme provocar errores permanentes.
- Control total: El usuario experimentado querrá tener control del sistema. Por ello hay que diseñar el sistema para que responda a las acciones del usuario, y no sea el usuario el que responde al sistema. No obstante se debe diseñar cuidadosamente, pues un exceso de control del usuario inexperto puede dar lugar a errores.
- Reducir la carga de memoria a corto plazo: Las pantallas deben ser simples, evitando cambios de ventanas frecuentes, y ofreciendo un periodo de aprendizaje adecuado a la interfaz. Con ello, y combinado con las acciones secuenciales, evitaremos que el usuario tenga que memorizar en qué punto se encuentra de un proceso dado.

[26] Atajos. Combinación de teclas o acceso directo que acciona una función concreta sin necesidad de pasar por elementos intermedios.
[27] Retroalimentación. Reacción o respuesta del sistema con respecto a una acción del usuario.

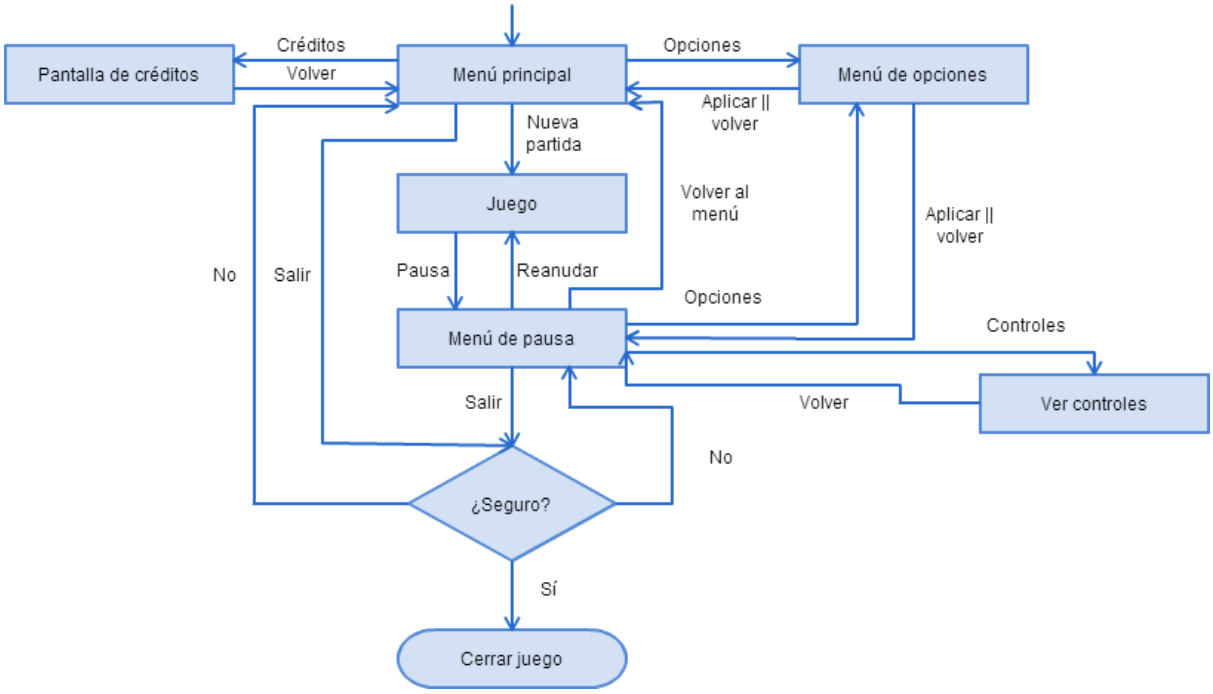


Diagrama 2: Diagrama de flujo de los menús

Como se puede apreciar en el diagrama 2, los menús garantizan el retroceso de las acciones, así como el control del usuario y la reducción de la carga de memoria del usuario. No obstante, obviamos las reglas de los *shortcuts* y las acciones secuenciales, pues en una interfaz tan simple no caben los atajos; debido a la poca interacción requerida por el usuario. Así mismo, observando los diseños de las interfaces que contiene el GDD, podemos apreciar la consistencia del diseño entre los distintos menús. Siguiendo todos la misma estructura. Así mismo, los botones cambian de color cuando están seleccionados, ofreciendo *feedback* al usuario. De la misma forma, las opciones exclusivas se gestionan mediante botones exclusivos. De esta forma el usuario no podrá seleccionar dos opciones que no sean compatibles, evitando errores.

6.2 Los personajes

Para gestionar a los personajes, diseñamos dos estructuras básicas que los componen –tres en el caso de los enemigos–: un controlador de personaje, que contendrá todas las funciones que dicho personaje puede realizar; y un controlador de las animaciones, el cual se encargará de ir cambiando entre las distintas animaciones del personaje conforme al estado en el que se encuentre. Además, en el caso de los enemigos, también contienen una máquina de estados que gestiona la inteligencia artificial de los mismos.

Los controladores de personaje están formados por un número concreto de funciones que les permiten realizar sus acciones básicas: caminar, saltar, atacar, etc. Por ello, dada su condición de código, se explicará en el siguiente apartado, en el punto 7 de Implementación.

El controlador de las animaciones es, como se ha mencionado, el encargado de ir oscilando entre las distintas animaciones de cada personaje dependiendo de la situación. No obstante, necesitamos una máquina de estados para cada personaje donde cada estado es una animación. El controlador de animaciones se encarga pues de ir cambiando el estado actual entre los estados de esta máquina.

Puesto que todos los personajes son capaces de realizar sus acciones en cualquier momento sin restricciones (por ejemplo: el jugador no necesita estar en el suelo para atacar), esto simplifica mucho el diseño de las máquinas de estados de animaciones.

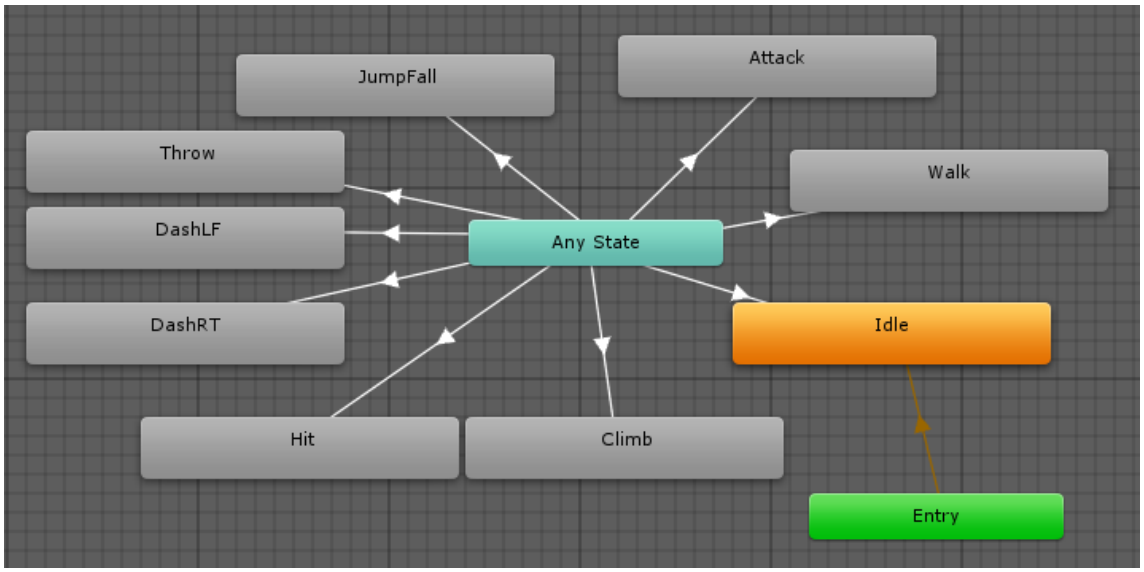


Diagrama 3: Máquina de estados de las animaciones del jugador.

Como vemos en el diagrama 3, la animación inicial es la de *idle*[28], no obstante, después se puede dar paso a cualquiera de las otras animaciones. El funcionamiento de la máquina es simple: cada máquina de estados de animaciones tiene un atributo llamado *controlAnimState*, dicho atributo es un valor entero que indica qué animación debe reproducirse. Una vez el jugador presiona alguna tecla de acción, el controlador de animación lee la acción y cambia el valor de *controlAnimState* para que se adecúe a la animación correspondiente. Con este valor cambiado, la máquina cambia el estado actual y empieza a reproducir la animación del estado correspondiente.

Las máquinas de los enemigos –diagramas 4 y 5–, funcionan de igual manera. Con la única diferencia que su animación principal es la de *walk*, pues lo enemigos no deben esperar las órdenes del jugador como el protagonista. Por ello empiezan caminando desde el primer momento.

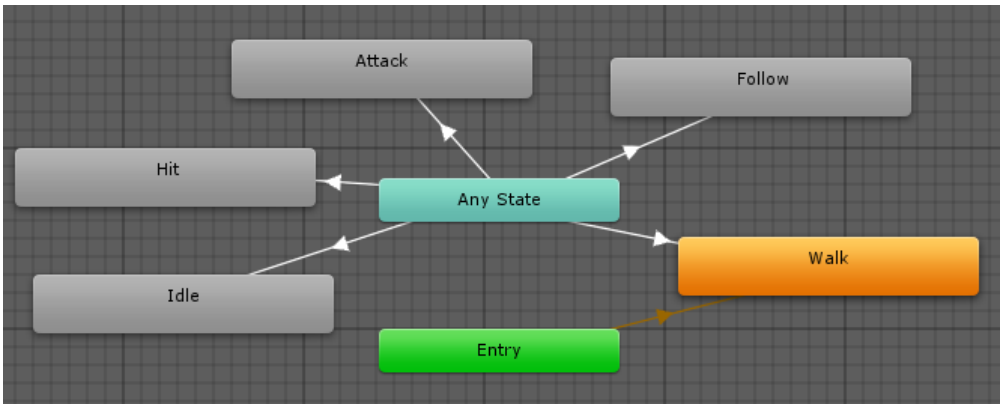


Diagrama 4: Máquina de estados de las animaciones de los enemigos «Capataz» y «Trabajador».

[28] Animación básica de reposo. Se utiliza cuando el jugador no interactúa con el personaje.

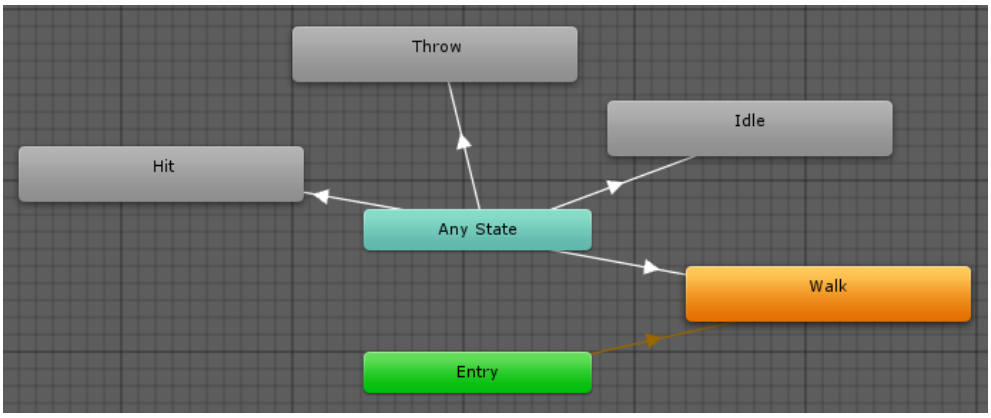


Diagrama 5: Máquina de estados de las animaciones del enemigo «Soldado».

Resaltar que las definiciones tanto de los enemigos, como del propio protagonista, se encuentran en el GDD, anexo a este mismo documento. Por ello no se han visto los tipos de enemigos, ni las características de cada uno, dado que estaríamos repitiendo el contenido del Documento de Diseño.

En lo que respecta a los enemigos, puesto que no reciben unas órdenes de entrada por parte del jugador, es necesario que cada uno cuente con su propia máquina de estados –la cual es independiente de la máquina de estados de animaciones–, que gestiona el comportamiento de los enemigos. Cada estado de la máquina representa una acción que el enemigo debe realizar mientras se encuentre en el mismo. El estado indica al controlador de personaje la acción para que éste la ejecute, así como avisar al controlador de animación para que reproduzca la animación correspondiente.

Los enemigos «capataz» y «trabajador» comparten una misma máquina de estados, pues su comportamiento es el mismo, variando sólo atributos de cada enemigo, como velocidad, rango de ataque, etc. Estos enemigos patrullan una superficie hasta chocar contra un obstáculo o detectar una cornisa, momento en el que se giran sobre sí mismos y siguen patrullando. Si detectan al jugador, empiezan a perseguirle, aumentando su velocidad temporalmente. Una vez tienen al jugador a tiro, atacan al mismo. Si por el contrario, aún no están lo suficientemente cerca, vuelven a perseguirle. Si el jugador escapa de su rango de visión, los enemigos vuelven a patrullar.

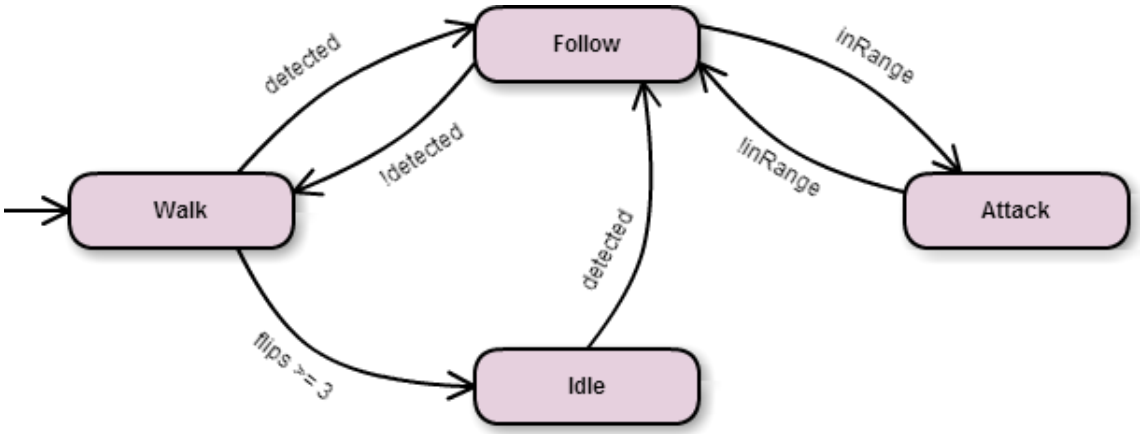


Diagrama 6: Máquina de estados de los enemigos capataz y trabajador.

No obstante, y como se observa en el diagrama 6, se ha añadido al diseño original un estado *idle*. Dicho estado actúa igual que el del protagonista, manteniendo al personaje en reposo sin realizar ninguna acción. Este estado se añadió a posteriori al diseño para solucionar un error del comportamiento de los enemigos. Cuando un enemigo se encuentra en una superficie pequeña, entre dos muros o dos cornisas, el enemigo sigue patrullando, girándose cada vez que va a caer o a colisionar con el muro. Al ser una superficie pequeña, el enemigo gira muchas veces en pocos segundos, dando lugar a parpadeos en el *sprite*. Es por esto que se añadió el estado *idle*. Si el enemigo gira más de tres veces en menos de un segundo, se pasa al estado *idle*, pasando a quedarse estático; y dado que la superficie es pequeña, no requiere patrullar. De esta forma, se soluciona el problema.

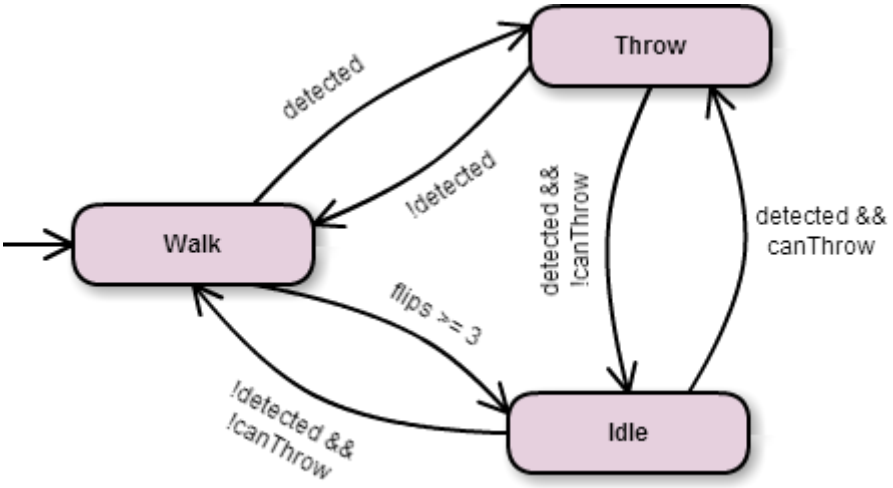


Diagrama 7: Máquina de estados del enemigo soldado.

Por el contrario, el funcionamiento del soldado difiere del de estos dos enemigos. Siendo el soldado un enemigo que lanza proyectiles, no necesita un estado de persecución. Al igual que los otros enemigos, patrulla una superficie, pero si detecta al jugador, pasa a lanzarle el proyectil directamente. El estado de *idle*, sigue siendo necesario para solucionar el mismo error que en los otros enemigos, pero aquí cumple una segunda funcionalidad. Teniendo el ataque un *cooldown* [29], una vez reproducida la animación de *throw* y haber lanzado el proyectil, no podemos dejar al soldado en el estado de lanzar, pues se quedaría reproduciendo la animación pero no lanzaría nada. Por ello, mientras no pueda lanzar –una vez ha lanzado un proyectil, no se le deja volver a lanzar hasta que haya pasado el tiempo del *cooldown*–, se cambia a estado *idle*, manteniendo la animación de reposo y quedándose quieto. Una vez ha pasado el tiempo, si el jugador sigue en su campo de visión, volverá a atacar. En caso contrario, volverá a patrullar la superficie.

6.3 Generación de nivel

Como ya se ha visto en el análisis, el algoritmo ORE era el que más se adaptaba a nuestras necesidades a la hora de crear los niveles en tiempo de ejecución. Aun así, modificamos su diseño ligeramente para hacerlo más aleatorio y darle un toque de personalidad propia al proyecto.

Cada nivel del juego se compone de una cuadrícula de 4 x 4. Cada una de estas celdas, es, como ya se ha visto, una habitación que podrá estar conectada o no a sus adyacentes. No

[29] Tiempo de espera hasta que se puede volver a realizar una misma acción.

obstante, las únicas conexiones que nos interesan son las de las salas que conforman la ruta de salida; pues esta ruta debe garantizar el camino desde la entrada hasta la salida. Para indicar esta información, representaremos cada tipo de habitación con un número entero:

- 0: Esta habitación no pertenece a la ruta de salida.
- 1: Pertenece a la ruta y tiene aberturas por sus dos laterales.
- 2: Pertenece a la ruta y tiene aberturas en sus laterales y la parte superior.
- 3: Pertenece a la ruta y tiene aberturas en sus laterales y la parte inferior.
- 4: Pertenece a la ruta y tiene aberturas en los cuatro puntos cardinales.
- 5: Tienda. Habitación especial. No puede pertenecer a la ruta de salida.
- 6: Sala del tesoro. Habitación especial. No puede pertenecer a la ruta de salida.

Como la entrada se coloca siempre en una habitación aleatoria de la fila inferior de la cuadrícula, sólo podrá ser de tipo 1 o tipo 2. Por otro lado, y puesto que la salida siempre se sitúa en la fila superior de la cuadrícula, la habitación que la contenga sólo podrá ser de tipo 1 o tipo 3. De esta manera terminamos de definir las representaciones de las habitaciones:

- 7: Habitación de tipo 1 con la entrada al nivel.
- 8: Habitación de tipo 2 con la entrada al nivel.
- 9: Habitación de tipo 1 con la salida del nivel.
- 10: Habitación de tipo 3 con la salida del nivel.

Hay que destacar que todos los tipos de habitaciones pueden tener aberturas en otras direcciones además de las indicadas. Los tipos de habitaciones sólo garantizan que siempre habrá aberturas en las direcciones que su tipo indique, no obstante, podría haber más o no. Esto quiere decir que podría darse el caso de una habitación de tipo 1 que tenga, además de sus aberturas laterales, una abertura en su parte superior; no obstante, nunca podría darse el caso de una habitación de tipo 1 que no tuviera una de sus aberturas laterales.

Las habitaciones de tipo 0 no garantizan ninguna abertura. Puede darse el caso que no se pueda acceder a una determinada habitación fuera de la ruta de salida, puesto que su acceso no está garantizado.

Así pues, una vez se inicia el algoritmo, éste genera una cuadrícula donde todas las habitaciones son de tipo 0. Libera a un agente, en una posición aleatoria de la fila inferior, capaz de desplazarse a izquierda, derecha, y arriba con una probabilidad del 40, 40 y 20% respectivamente. Este agente va transformando todas las habitaciones por las que pasa en habitaciones de la ruta garantizada de salida, adjudicándoles el valor que les corresponda de acuerdo al tipo de la anterior habitación visitada por el agente. Si el agente intenta desplazarse a izquierda o derecha, saliéndose con dicho desplazamiento de la cuadrícula, su movimiento se cambia por un desplazamiento superior. Una vez llega a la fila superior, tiene un 50% de probabilidades de acabar ahí. En caso que no acabe, sólo podrá desplazarse a izquierda o derecha, pues está en la fila superior y no puede continuar hacia arriba, hasta que termine. La habitación de inicio del agente se cambia al tipo 7 u 8, dependiendo de su situación, y la sala donde muere el agente, a 9 o 10, acorde al mismo criterio.

Una vez trazada la ruta de salida, el algoritmo puede cambiar alguna de las habitaciones de tipo 0 por una tienda o una sala del tesoro. La tienda tiene una probabilidad de aparecer de un 25% y la sala del tesoro una probabilidad del 5%. No obstante sólo puede aparecer una sala

especial por nivel. Dado que actualmente se dispone de tan solo cuatro ítems a vender en las tiendas, si el jugador compra todos los ítems, no se vuelve a generar ninguna tienda en los niveles restantes. Con esto se evita el absurdo de poner una tienda que no tenga objetos que vender. Una vez finalizado este proceso, ya disponemos de la ruta de salida así como de las posibles habitaciones especiales del nivel. Podemos observar un ejemplo de la representación de un nivel, en este punto de la ejecución del algoritmo, en la tabla 1.

0	9	3	0
5	0	4	0
3	1	2	0
2	7	0	0

Tabla 1: Representación de un nivel.

Las salas se han representado mediante una cuadrícula de 10 x 10 celdas. A diferencia de la representación del nivel, las salas se componen de una cuadrícula de caracteres, en vez de números enteros. Estos caracteres representan elementos que encontraremos en esa posición dada de la habitación. Así pues, los caracteres pueden ser:

- 0: Celda vacía.
- 1: Muro.
- 2: 33% de probabilidades de muro.
- 3: 66% de probabilidades de muro.
- S: 50% de probabilidades de pinchos, siempre que en la casilla inferior haya un muro.
- C: 40% de probabilidades de cofre, siempre que en la casilla inferior haya un muro.
- L: Escalera.
- E: Puerta de entrada al nivel.
- N: Puerta de salida del nivel.

Como hemos visto en el análisis, el algoritmo ORE posee un banco de *chunks* –en nuestro caso cada *chunk* es una habitación–, del cual escoge uno y lo indexa en el nivel. En nuestro caso, y como se indicó que variaríamos el algoritmo, disponemos de varios bancos de salas. En concreto, un banco por cada tipo de sala. Una vez el algoritmo ha generado la ruta y dispone de la representación del nivel, va recorriendo todas sus salas y escogiendo para cada tipo, una sala del banco de su mismo tipo. Una vez escogida la sala del banco –ver tabla 2–, se calculan las probabilidades de los elementos de la sala –muros probabilísticos[30], pinchos, y cofres–, y se representa la sala final –ver tabla 3–.

1	1	1	1	1	1	2	2	2	1
1	1	1	0	0	0	C	0	0	1
1	1	1	0	3	3	3	3	0	1
1	0	0	0	0	3	2	2	0	1
0	0	0	2	0	0	0	0	0	0
0	0	2	2	0	0	0	0	0	1
0	2	2	2	S	S	S	0	3	3
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Tabla 2: Representación de una habitación de tipo 1 en el banco de salas.

[30] Aquellos muros que tienen una probabilidad de aparecer.

1	1	1	1	1	1	0	0	0	1
1	1	1	0	0	0	0	0	0	1
1	1	1	0	1	1	1	1	0	1
1	0	0	0	0	1	0	0	0	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	1
0	0	0	0	S	S	S	0	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1

Tabla 3: Representación de una habitación de tipo 1 después de calcular las probabilidades de sus componentes probabilísticos.

Sobra decir que la tabla 3 sólo es una posible representación, pues después de calcular sus componentes probabilísticos la representación podría ser otra.

Por lo que respecta a los enemigos, se utiliza un algoritmo de posicionamiento independiente; basando la posición de los enemigos dependiendo del valor de la función *fitness*. Puesto que lo más importante para decidir la posición de un enemigo es su entorno, la función *fitness* debe evaluar y cuantificar la idoneidad del entorno de la posición que a evaluar. Por ello nos preguntamos «¿qué requisitos tiene un enemigo?» y «¿dónde ofrece un reto mayor para el jugador?». La respuesta a la primera pregunta es simple: en Babel, los enemigos necesitan superficies amplias sobre las que poder patrullar y defenderlas. Si el jugador cae en la superficie, con toda seguridad el enemigo lo detectará e irá a por él. Así pues, los rincones poco espaciosos no son una buena posición para los enemigos. Respecto a la segunda pregunta, los enemigos resultan molestos en dos posiciones: cerca de pinchos –pues el jugador tendrá que tener en cuenta tanto los pinchos como el enemigo a la hora de esquivar–, y cerca de cofres. Esta última posición hará que el jugador se debata entre ir a por el cofre o no. Pues el cofre ofrece una recompensa, pero si se decide a ir, deberá arriesgarse a perder puntos de salud o incluso morir. Así pues, la función *fitness* queda de la siguiente forma:

$$fitness(p) = chest\big(\mathrm{A(p)}\big) + flat(\mathrm{A(p)}) * 0.5 + spikes\big(\mathrm{A(p)}\big) - walls\big(\mathrm{A(p)}\big) * 0.5$$

Función 1: Función *fitness*.

- p: Posición actual.
- A(p): Celdas adyacentes a la posición actual. Para su mayor comprensión, se considera adyacentes a todas las celdas colindantes, como en el vecindario de Moore en los autómatas celulares.
- chest(A(p)): Número de cofres en las celdas adyacentes.
- flat(A(p)): Longitud de la superficie en la que se encuentra la posición.
- spikes(A(p)): Número de pinchos en las posiciones adyacentes.
- walls(A(p)): Número de muros en las posiciones adyacentes. Si este valor es menor o igual que cuatro, se cambia al valor nulo.

Con esta función valoraremos positivamente las posiciones cercanas a cofres y pinchos, aquellas que se encuentren en superficies de al menos tres celdas de longitud, y penalizaremos las posiciones demasiado rodeadas de muros, es decir, evitando los rincones. Cabe destacar que el valor de los muros se anula cuando su valor es menor o igual que cuatro puesto que el objetivo de este parámetro es penalizar los rincones. Si el valor es mejor que cuatro, indica que

no está en un rincón, por lo que no hay que penalizar dicha posición. Tanto los valores de la superficie, como el de los muros se multiplican por 0.5 por su posible alto valor. Dado que los parámetros *chest* y *spikes* tendrán un valor de 0 o 1, los otros dos parámetros alcanzarán valores bastante superiores; rebajándolos a la mitad conseguimos una distribución más equitativa, pues en caso contrario las posiciones más valoradas serían las grandes superficies, sin que los cofres o pinchos influyeran significativamente.

Una vez definida la función, el algoritmo simplemente va calculando el valor *fitness* para cada posible posición y escoge el número de mayores valores que corresponda al número de enemigos de la habitación.

7. Implementación

Con el diseño de las soluciones a los problemas planteados en el análisis realizado, la implementación de las mismas se antojaba fácil y sencilla. No obstante, surgieron algunos problemas debidos a malos diseños, como las máquinas de estado de los enemigos sin un estado de *idle*, que tuvieron que ser rediseñados en esta etapa del desarrollo.

Todos los *scripts* que componen el proyecto se pueden agrupar en seis grandes conjuntos: managers, controladores, IA[31], UI, generación de nivel, y utilidades. Cabe decir que tanto la UI como la generación de nivel podrían ser considerados como controladores, no obstante se les da un grupo propio por su peculiaridad y extensión.

7.1 Utilidades

Las clases de este tipo son *scripts* con utilidades puntuales que sólo se utilizan en alguna situación concreta. Así, de esta forma, tenemos como ejemplo la clase *Chest*, que se encarga de cambiar el *sprite* de un cofre cuando el jugador lo abre, así como generar un número aleatorio de dinero e instanciarlo en el juego. Como estas clases ofrecen puntualidades concretas, no entraremos a explicarlas todas ellas. No obstante, haremos un inciso en la clase *Pools*.

Las *Pools* son una estructura de la programación de videojuegos que constan de una lista de objetos ya instanciados. Puesto que las funciones *Instantiate()* y *Destroy()* de *Unity* tienen un coste computacional elevado, las *pools* son listas que nos permiten instanciar en ellas todos los objetos que vamos a necesitar, en una única fase de carga, y tener todos los objetos cargados pero inactivos. Cuando los objetos son necesarios, se activan, y cuando ya no lo son, se vuelven a desactivar. Con ello evitamos tener que estar instanciando y destruyendo objetos constantemente, pues esto sería una carga computacional para el *gameplay*. Veamos un ejemplo:

Puesto que nuestro juego genera cada nivel en la misma escena, y los materiales a utilizar son los mismos –muros, escaleras, cofres, pinchos, enemigos–, sería del todo absurdo ir instanciando al principio de cada nivel los objetos y destruirlos al acabar para volver a instanciarlos en el siguiente nivel. Por ello creamos un *pool* para cada tipo de objeto. Cuando el algoritmo tiene la representación del nivel, simplemente va accediendo a los distintos *pools* y va activando los objetos que necesite, y los va colocando en la posición pertinente. Cuando un enemigo muere, o el nivel se acaba, los objetos se desactivan, y vuelven a estar listos para utilizarse en el siguiente nivel o partida.

Obviamente esta estructura resulta altamente útil puesto que las funciones de activar y desactivar objetos son mucho más eficientes y rápidas que las de instanciar y destruir objetos. Así, con una única pantalla de carga al principio del juego, e indicando a las *pools* que no se destruyan en la carga de escenas mediante la función *DontDestroyOnLoad()*, ganaremos tiempo de cómputo final a costa de tener los objetos cargados en memoria.

7.2 Managers

Constituyen la dirección del juego. Los managers son los encargados de gestionar uno o varios recursos concretos para su correcto funcionamiento. Por ello, se ha declarado que los managers sean *singletons*. Haciendo que los managers sean patrones *singleton* conseguimos

[31] Inteligencia Artificial.

poder llamarlos desde cualquier clase del juego. Además, crearemos un único objeto para cada mánager, pues el *singleton* garantiza la instancia única. Con ello evitaremos problemas que pudieran originarse si en un caso concreto tuviéramos dos managers de un mismo tipo –pues los datos entre uno y otro podrían diferir, generando controversias en el juego–.

El proyecto consta sólo de dos managers: el *GameManager* o manager del juego, y el *SoundManager*. El primero es el encargado de gestionar toda la lógica del juego: instancia los elementos necesarios en cada escena, va cambiando entre niveles y escenas según el jugador vaya avanzando, y destruye los objetos innecesarios al acabar los niveles. Por el contrario, el *SoundManager* se encarga, como su nombre indica, de la gestión del volumen de juego. Tanto de la música como del volumen de los efectos, y el volumen general. Así mismo, es el encargado de reproducir todos los sonidos del juego. Veámoslos en más profundidad:

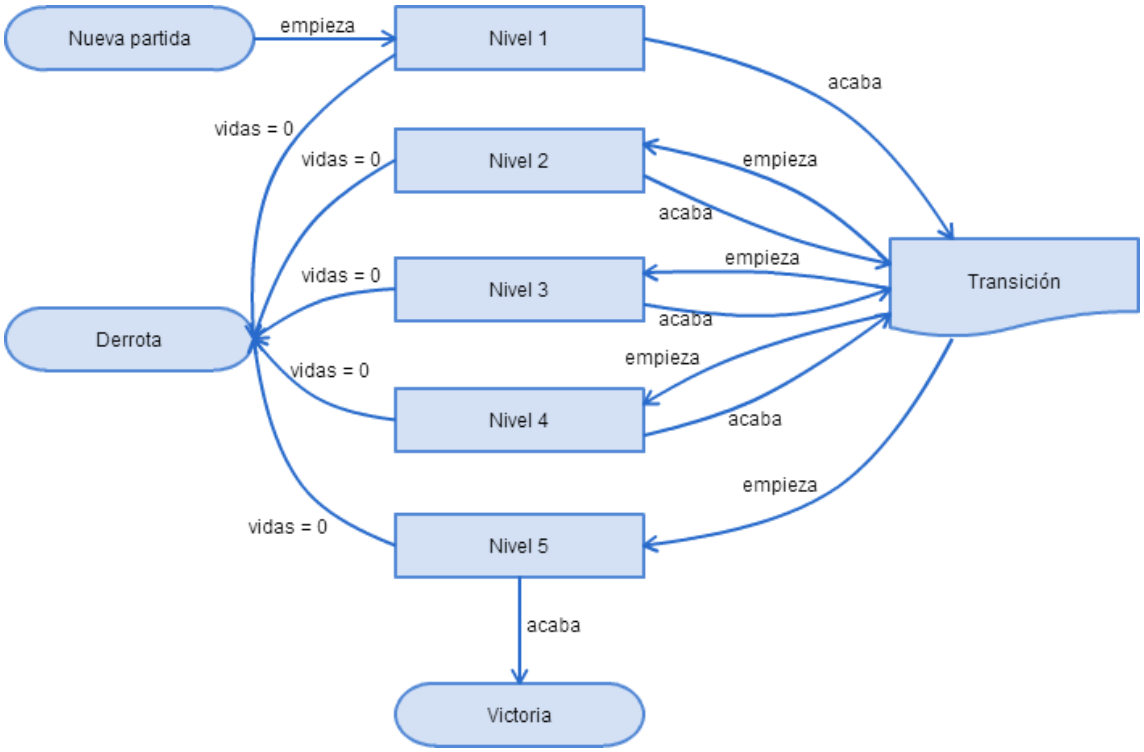


Diagrama 8: Diagrama de flujo del *GameManager* a lo largo de la partida.

Una vez se ejecuta el juego, el *GameManager* es instanciado por una clase *Loader* cuya única función es asegurarse que siempre hay un *GameManager*. El *GameManager* instancia el *SoundManager* para que empiece a reproducirse la música y se activen los sonidos. Una vez el jugador empieza una nueva partida, el *GameManager* instancia todas las *pools* y los objetos necesarios –el protagonista y el generador de nivel–, los cuales no volverán a ser instanciados pues son persistentes a la carga de escenas. De esta forma, antes de generar el primer nivel, ya tenemos todos los objetos que necesitaremos para el mismo. Una vez instanciados todos los requisitos, y como vemos en el diagrama 8, se carga la escena de nivel y el generador crea la primera pantalla. Cuando el jugador supera el nivel con éxito, todos los objetos se desactivan –que no destruir–, y se da paso al nivel de transición. Este nivel consta de una pequeña habitación con dos puertas, una de entrada y la otra de salida. El jugador simplemente debe ir desde la puerta de entrada a la de salida, muy cerca ambas. El objetivo original de este nivel era sustituir

a las pantallas de carga mientras se generaba el siguiente nivel. No obstante, en las pruebas realizadas se vio que la generación de nivel era casi instantánea debido a que no era necesario hacer instanciaciones, pues ya estaban hechas de antemano. Aun así, se dejó el nivel como descanso para el jugador entre niveles, ya que cargar directamente el siguiente nivel al acabar el primero puede resultar agotador para el jugador, que tomará el nivel de transición como un respiro. Además, y puesto que el juego se ambienta en la Torre de Babel, este mini-nivel de transición son sólo unos pocos escalones ascendentes hasta la siguiente puerta, por lo que encaja en el *lore*[32] del juego, siendo el tramo de subida de una planta a otra de la torre.

Una vez superado el quinto nivel, no se desplaza al jugador al nivel de transición, sino que se carga la pantalla de victoria y después se vuelve al menú principal. Todas estas cargas y gestiones las realiza el *GameManager* como encargado que es del correcto funcionamiento del juego.

Por otra parte, la clase *SoundManager*, como ya se ha dicho, se encarga de reproducir todos los audios del juego, tanto la música como los efectos de sonido; así como también de gestionar los niveles de audio. Esta clase está basada en el *SoundManager* del tutorial oficial de *Unity*: *2D roguelike tutorial* [10]. No obstante se le han añadido modificaciones para que también gestione los niveles de audio, cosa que no hace la clase original del tutorial.

Para los efectos de sonido, esta clase ofrece la posibilidad de asignar más de un efecto a una misma acción, y elegir qué efecto reproducir de manera aleatoria; también ofrece modificar ligeramente el *pitch*[33] de cada efecto de manera aleatoria, dando una mayor sensación de dinamismo con los sonidos.

7.3 Controladores

Los controladores son clases que se encargan de gestionar las distintas funcionalidades de los personajes. Dentro de los controladores tenemos dos tipos: los controladores de personaje y los controladores de animaciones. Como ya se vio en el diseño su estructura y sus responsabilidades, pasaremos a ver su implementación y estructura interna.

Puesto que todo juego debe ser escalable, para añadir en un futuro nuevo contenido o actualizar el actual, los controladores de personajes se basan en una estructura de herencias que podemos apreciar en el diagrama 9. Como vemos, todos los controladores de personaje heredan de *GroundCharacters* y esta clase de *Characters*.

Characters es una clase que recoge todas las características comunes a todos los personajes, ya sean enemigos o el propio jugador, junto a sus capacidades básicas. En esta clase podemos encontrar, por ejemplo, los atributos con referencia a la *transform* y el *rigidbody* de cada personaje, o los métodos que permiten caminar, girarse, lanzar un proyectil, o atacar. De esta manera, y puesto que todos los enemigos heredarán en última instancia de esta clase, definimos las propiedades comunes a todos ellos.

La clase que hereda de *Characters*, *GroundCharacters*, recoge, al igual que su clase padre, las características de todos los personajes, pero con la peculiaridad que son las de los terrestres. Esta clase existe pues en un principio se pensó añadir también un enemigo volador, por lo que se crearía otra clase *AirCharacters* que heredara de *Characters*. No obstante, debido

[32] Trasfondo de la historia, ambientación.
[33] Percepción de la frecuencia de un sonido.

a la falta de tiempo, finalmente no se implementó ningún enemigo volador. Aun así, se dejó la clase *GroundCharacters* para definir las características de los enemigos terrestres y que, si en un futuro se añadían enemigos de otros tipos, fuera más simple su implementación; habiendo estructurado y distinguido correctamente las características de unos y otros enemigos.

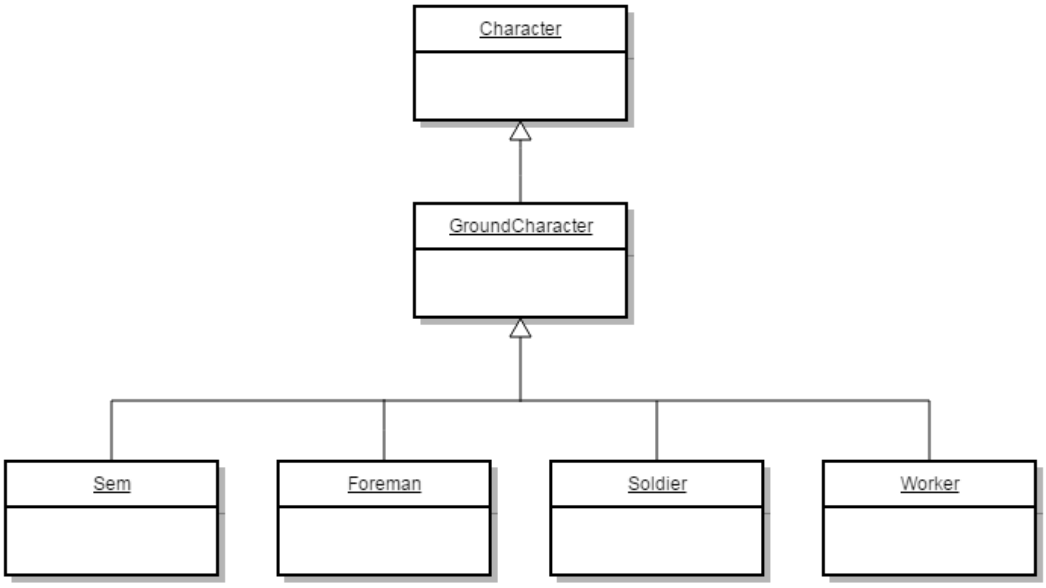


Diagrama 9: Diagrama de clases de los controladores de personaje.

Así pues, *GroundCharacters* contiene las funciones propias de personajes terrestres como saltar o realizar el *dash*. De la misma forma, esta clase se encarga de actualizar el control y las físicas de los personajes, cambiando su estado actual dependiendo de la acción que realicen. También llama al *SoundManager* con cada acción para reproducir el efecto de sonido correspondiente a la misma.

Por otro lado, las clases con el nombre de cada personaje son sus controladores particulares que, en este caso, heredan de *GroundCharacters*, pues todos son personajes terrestres. Los controladores de los enemigos son muy similares entre ellos. En ellos se definen sus capacidades, si pueden saltar, realizar un *dash*, etc. Así como su salud, y una llamada a los métodos *UpdateControl()* y *UpdatePhysics()* de su clase padre *GroundCharacters* en el método *Update()* para actualizar su estado constantemente. También constan de la función propia *Damage()* que les resta salud cuando reciben un ataque o interactúan con algún elemento que aplique daño. Los enemigos además, tienen una función que llama al *GameManager* para acceder a las *pools* del dinero. Dicha función se activa cuando el enemigo muere y escoge una suma aleatoria de dinero y activa el número pertinente de monedas, bolsas de dinero y barras de oro de las *pools* colocándolas en su posición actual. Este es el método *Drop*.

En cambio, el controlador del personaje, además de estas características, incluye las variables de los objetos comprados, el dinero actual, las cuerdas de las que se disponen, así como métodos para subir por ellas y por las escaleras –pues el jugador es el único personaje que puede subir por ellas–. No obstante, el método más importante del controlador del jugador es *DetectInput()*. Este método detecta los valores de entrada introducidos por el jugador, y dependiendo de la acción que sea, llama a una u otra función de *GroundCharacters* para que actualice el estado del personaje y realice la acción ordenada.

Por lo que respecta al controlador de animaciones, que podemos observar su estructura en el diagrama número 10, consta tan solo de dos clases: *Animations* y *GroundCharactersAnim*. No obstante, si se añadieran tipos de enemigos más adelante, se debería crear otra clase que herede de *Animations*, por ejemplo, en caso de implementar enemigos voladores, debería ser una clase llamada *AirCharactersAnim*.

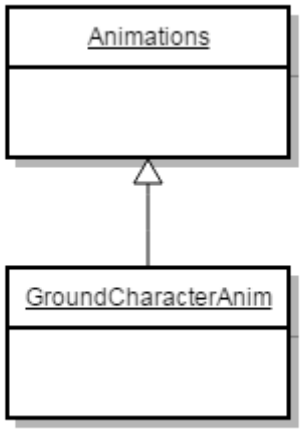


Diagrama 10: Diagrama de clases del controlador de animaciones.

La clase *Animations* simplemente se encarga de recoger en un atributo el componente *Animator* del personaje en cuestión, así como del estado actual en el que se encuentra. Por otro lado, la clase *GroundCharactersAnim*, se encarga de ir cambiando el atributo del *Animator*, *controlAnimState* –el cual recordemos servía para cambiar entre los distintos estados de las máquinas de estados de animaciones de cada personaje–, en función del estado actual. Así pues, existe una relación de dependencia entre la clase *GroundCharacters* y *GroundCharactersAnim*, pues cuando en la primera se actualiza el estado para realizar una acción, la segunda lo utiliza para actualizar la máquina de estados de animaciones y que cambie la animación del personaje.

Ilustremos esto con un ejemplo para aclararlo de manera más concisa: Cuando el jugador aparece en el nivel, mientras que no se interactúe con él, se mantiene en el estado *idle*. Es decir, su estado actual es igual a *NONE*. Cuando el jugador desplaza el personaje, su estado actual cambia a *WALK*, el controlador *Sem*, actualiza el estado, llamando a la clase *GroundCharacters* que ejecuta el método *Walk()* de *Characters* –pues es una acción común a todos los enemigos indistintamente de su tipo–, y el personaje se desplaza. Paralelamente, *GroundCharactersAnim* detecta el cambio de estado actual a *WALK*, por lo que cambia el valor de *controlAnimState* de 0 a 1. Este cambio le indica a la máquina de estados de animaciones, que debe cambiar al estado *Walk*, cuya animación es la de movimiento. Así pues, el jugador percibe que el personaje se desplaza, viendo como su animación es la de movimiento.

7.4 Inteligencia Artificial

La inteligencia artificial se basa, como se ha visto en el diseño, en máquinas de estados. Como a diferencia de las máquinas de estados de animaciones, *Unity* no ofrece un sistema de máquinas de estados para la inteligencia artificial, creamos nuestra máquina de estados basándonos en el diseño de Ignacio Díaz, presidente de la Asociación de Estudiantes de Videojuegos [11].

Siguiendo este diseño pues, las máquinas de estados se componen de dos clases: la máquina propiamente dicha –clase *StateMachine*–, y la clase *State*. La clase *State* sirve como método padre para cada uno de los estados que vayamos a crear. Esta clase contiene la referencia a *StateMachine*, es decir, a la máquina de estados. Así como el método *CheckExit()* el cual se llama para comprobar si hay que cambiar de estado.

La clase *StateMachine* tan solo necesita que le indiquemos el estado inicial y el tiempo en segundos cada cuanto debe comprobar el método *CheckExit* del estado actual para ver si debe cambiar de estado. Así pues, llama la atención que simplemente se le indique el estado actual; esto se explica indicando que es en el método *CheckExit* de cada estado que implementamos, donde se indican las condiciones y los posibles siguientes estados. De esta forma, la clase *StateMachine* se encarga de activar el estado actual y, cuando deba cambiar de estado, lo desactiva, cambia el estado actual al siguiente, y lo vuelve a activar. Podemos observar este funcionamiento en el diagrama 11.

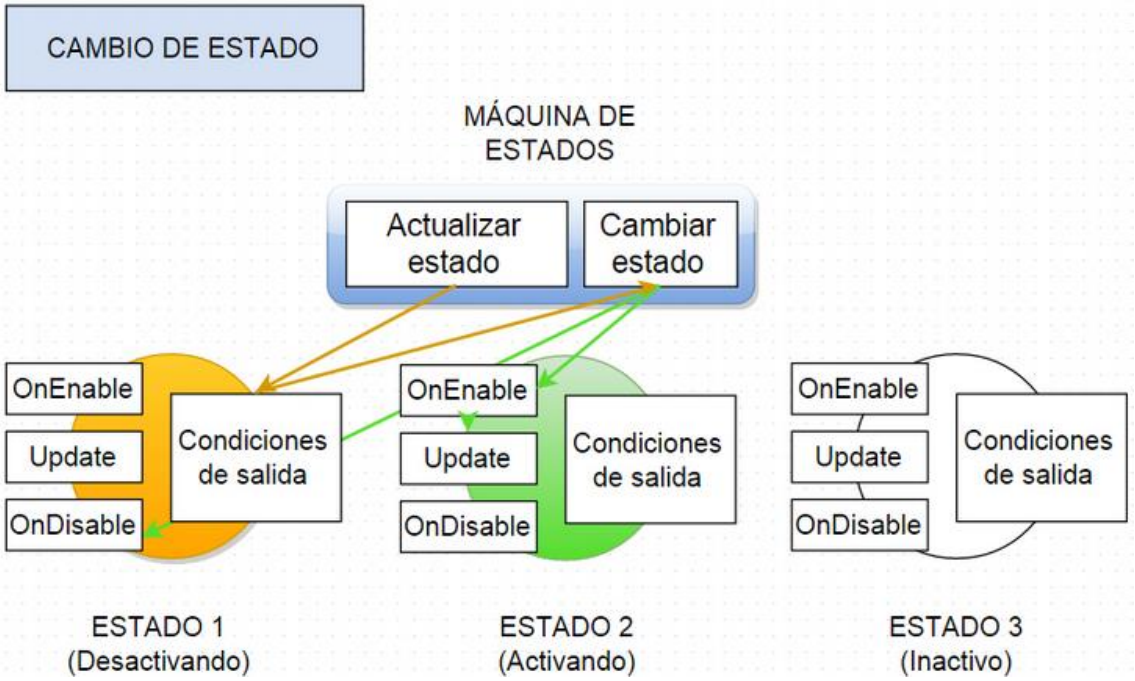


Diagrama 11: Diagrama de flujo de cambio de estado en la máquina de estados. Diagrama por Ignacio Díaz [11].

Con esta estructura compuesta por las dos clases ya mencionadas, la implementación de la IA simplemente consistió en crear una clase para cada estado e indicar las condiciones de salida, especificadas en la fase de diseño, dentro de la función *CheckExit*.

7.5 Interfaz de usuario

Puesto que desde *Unity* 4.6 la interfaz de usuario se puede crear de manera gráfica, los diferentes *scripts* que componen esta categoría han sido muy simples de implementar. Los *scripts* únicamente tienen dos propósitos en este apartado: implementar la navegación por los menús con *gamepad*[34], y aportar los métodos que se ejecutarán cuando se pulsen los determinados botones.

[34] Mando para jugar.

Puesto que, como se indica en el GDD, Babel se puede jugar tanto con teclado como con *gamepad*, fue necesaria una implementación explícita de la navegación con *gamepad* por los menús del juego; ya que *Unity* sólo ofrece por defecto la interacción mediante el ratón. Esta implementación fue simple: creamos una lista donde colocamos los botones del menú en cuestión –cada *script* representa un menú, por lo que tendrá unos botones propios–, junto con un índice que apunta al primer botón. Cada vez que el usuario mueve el *joystick*[35] del mando, aumenta o disminuye el índice, cambiando el botón al que está apuntando el jugador. Cuando presiona el botón de acción, se llama a la función del botón que esté siendo apuntado por el índice. Indicar que el botón apuntado por el índice cambia de color, de manera que el jugador siempre sabe dónde está apuntando –cumpliendo así la tercera norma de Shneiderman–.

La implementación de las distintas funciones de los botones se explica a continuación:

- Nueva partida: Se llama al *GameManager* para que instancie todas las *pools* –pues es la primera partida– y cargue la escena del nivel.
- Opciones: Puesto que este menú se superpone al menú principal o al menú de juego, dependiendo desde donde se le llame, lo primero que se hace al pulsar este botón es inhabilitar todos los botones del menú principal o del de pausa para evitar posibles errores por parte del usuario. Una vez hecho esto, activamos el menú de opciones –el cual ya estaba cargado pero inactivo–, una vez el jugador ha cambiado la configuración, si pulsa en «Aplicar cambios» se lee la nueva configuración, comprobando qué *toggle*[36] están marcados, y se aplica. Salvo para el volumen, que se le pasa al *SoundManager* para que él aplique los cambios. Una vez se sale del menú, se vuelven a habilitar los botones del menú anterior y se desactiva el de opciones.
- Créditos: Se llama al *GameManager* para que cargue la escena de créditos. En esta escena no hay interactuación posible, cuando los créditos acaban, se vuelve automáticamente al menú principal.
- Salir: Cuando se pulsa, se carga un mensaje que se superpone al menú principal o de pausa. Como pasa con el menú de opciones, se inhabilitan los botones del menú anterior para evitar errores. El mensaje cargado es una pregunta al usuario indicando si realmente quiere salir de la aplicación. Si el jugador pulsa sobre el botón «Sí» se cierra la aplicación mediante la instrucción `Application.Quit()` que ofrece *Unity*. En caso contrario, se desactiva el mensaje y se vuelve al menú anterior, habilitando sus botones.
- Pausar el juego: Cuando el jugador presiona el botón de pausa durante la partida, además de cargarse el menú pertinente, se congela el juego. Esto se consigue poniendo el atributo `Time.timeScale` a 0. De esta forma, todos los elementos que utilicen el tiempo –como los enemigos o el personaje, ya que lo utilizan para calcular la distancia a desplazarse–, se congelan.
- Reanudar: Desde el menú de pausa, se desactiva el menú y se vuelve a poner `Time.timeScale` a 1 para descongelar a los personajes.
- Controles: Nuevamente, al ser otro menú superpuesto al menú de pausa, se inhabilitan los botones de este y se activa el menú de controles. Este menú no ofrece ninguna interacción, pues *Unity* no permite remapear los controles, por lo que es meramente

[35] Pequeña palanca que incorporan los mandos para controlar el movimiento.
[36] Botón que es una caja de selección con dos únicas posiciones, seleccionado o deseleccionado.

informativo para indicar al usuario cómo jugar. Cuando se presiona el botón de atrás el menú se inactiva y se vuelven a habilitar los botones del menú de pausa.

- Vuelta al menú: Se destruyen todos los objetos de la escena y se vuelve al menú principal del juego mediante el *GameManager*, que realiza ambas acciones.

7.6 Generación de nivel

La generación de nivel se divide en tres clases: *LevelGenerator*, *RoomGenerator*, y *EnemyPlace*. Cada una de estas clases se encarga de una funcionalidad del algoritmo diseñado en el punto 6.

7.6.1 LevelGenerator

Podríamos decir que este es el controlador de la generación de nivel. Esta clase es la encargada de llamar a las otras dos y procesar la información que recibe de ellas. *LevelGenerator* es la clase encargada de crear la ruta de salida del nivel. Una vez generada, recorre todas las habitaciones del nivel y va llamando en cada una a *RoomGenerator*, de donde recibe la sala, en forma de un *array* bidimensional, ya con sus bloques definitivos –es decir, una vez se han escogido qué bloques probabilísticos se mantienen y cuáles no–. Las salas definitivas que va recibiendo de la llamada a *RoomGenerator* se guardan en una lista de habitaciones para que el algoritmo de posicionamiento de enemigos pueda trabajar posteriormente.

Después, el algoritmo vuelve a llamar a *RoomGenerator*, en este caso al método *GenerateRoom*, que se encarga de leer la sala que recibe como argumento e ir activando los objetos que la componen en los *pools* y colocándolos en la posición pertinente. El propio *LevelGenerator* añade las *backgrounds*[37] después y los muros de contención. Estos muros se colocan alrededor de la cuadrícula del nivel para impedir que los enemigos o el propio jugador se puedan salir del mismo; ya que no sabemos qué salas estarán en los laterales y no podemos confiar en que ofrezcan garantía de estar cerradas por ese lateral concreto. Si observamos la tabla 2 del punto 6 podemos observar que dicha sala tiene aberturas por ambos laterales –pues es una sala de tipo 1–, y si esta sala estuviera en una de las celdas de los extremos del nivel, el jugador y los enemigos podrían salirse. Esta es la necesidad que cubren estos muros.

Una vez colocados, se recorre la lista de las habitaciones finales escogidas por *RoomGenerator* y se le asigna una dificultad a cada una, pudiendo ser: 30% de probabilidades de que sea fácil, 50% de probabilidades que sea media, 20% de probabilidades que sea difícil. Cada habitación se pasa a *EnemyPlace* junto con su dificultad asignada para que seleccione los enemigos y los coloque en la misma.

7.6.2 RoomGenerator

Esta clase contiene los bancos de habitaciones prediseñadas. Un banco por cada tipo de habitación posible. Los bancos no son otra cosa que *arrays* de *strings*, donde cada componente de los *arrays* es una habitación. Esto permite una alta escalabilidad, ya que si se quieren añadir nuevas salas a algún tipo, simplemente hay que introducir un nuevo *string* con el diseño en su banco correspondiente. Además de los bancos, esta clase actúa como *parser*[38], pues lee los *strings* de las habitaciones y los traduce en *GameObjects*. El proceso es simple: una vez recibe del *LevelGenerator* un tipo de habitación, escoge aleatoriamente entre las habitaciones del banco de su tipo correspondiente. Una vez elegida la habitación, mira sus bloques probabilísticos y se decide si, para esa habitación, se colocarán o no. Una vez decidido, se

[37] Imágenes de fondo.
[38] Software capaz de traducir un determinado lenguaje.

vuelca su contenido en un *array* bidimensional de caracteres y se devuelve al *LevelGenerator*. Este punto del proceso correspondería con la tabla 3 del punto 6. Todo esto lo realizan los métodos *ChooseRoom* –para elegir habitación entre el banco de habitaciones–, y *ChooseTiles* –para escoger los bloques probabilísticos–.

El último método de la clase es *GenerateRoom*. Este método recibe un *array* bidimensional de caracteres como argumento y lo traduce en *GameObjects*. Así pues, va leyendo carácter a carácter y los va interpretando acordes al diseño explicado en el subpunto 6.3 por los objetos pertinentes. Llama al *GameManager* para tener acceso a los *pools* y va activando los objetos que sean necesarios, y colocándolos en su posición precisa.

7.6.3 EnemyPlace

Esta clase es la encargada de calcular los valores *fitness* para todas las posiciones factibles de la sala que recibe como argumento. Junto a la sala, el método *PutEnemies()* recibe también la dificultad de la sala. La dificultad se utiliza para definir los rangos entre los que estará el número de enemigos de la misma. Aunque cabe decir que el número de enemigos se elige de manera aleatoria, la dificultad influye en los límites de esa aleatoriedad de la siguiente forma –todos los valores se incluyen–:

- Dificultad fácil: Entre 0 y 2 enemigos.
- Dificultad media: Entre 1 y 3 enemigos.
- Dificultad difícil: Entre 3 y 4 enemigos.

Una vez escogido el número de enemigos de la sala, el algoritmo sigue conforme a su diseño. Se recorren todas las celdas vacías cuya celda inmediatamente inferior sea un muro, y se le asigna un valor de *fitness*. Se escoge entre ellas el número de posiciones máximas igual al de enemigos y se llama al *GameManager* para tener acceso a las *pools* de los enemigos. Para decidir qué enemigo colocar en cada posición se mira la altura de la posición concreta.

Para toda celda por encima de la altura 6, inclusive –altura relativa a la sala–:

- 30% de probabilidades de Capataz.
- 50% de probabilidades de Soldado.
- 20% de probabilidades de Trabajador.

Para toda celda por debajo de la posición 6 –altura relativa a la sala–:

- 40% de probabilidades de Capataz.
- 20% de probabilidades de Soldado.
- 40% de probabilidades de Trabajador.

El motivo de esta decisión de implementación es simple. Por los diseños de las salas, las posiciones elevadas suelen contener cofres, debido a la dificultad para el jugador de alcanzarlas; por esto, es posible que se coloque a un enemigo en estas posiciones, dado que los cofres cercanos influyen en la función *fitness*. Una mayor dificultad en estas posiciones sería incluir enemigos que disparen proyectiles, pues al tener que saltar el jugador para esquivarlos podría provocar que cayera al vacío. Es por ello que se dio una mayor probabilidad de aparición a los soldados en posiciones elevadas, para intentar inducir al jugador a este reto.

Una vez decididas las posiciones y los enemigos que se colocarán en cada una, el algoritmo va activando los mismos en las *pools* de enemigos y colocándolos en las posiciones pertinentes, acabando con ello la ejecución del mismo y dando por finalizada la generación del nivel.

8. Resultados

El proyecto, una vez finalizado, fue sometido a dos pruebas. La primera consistía en probar todas y cada una de las funcionalidades del juego, para asegurarnos de su correcto funcionamiento. Esto iba desde la interacción de los menús, hasta la IA, la generación de nivel, la carga de escenas, etc.

Por otra parte, la segunda prueba consistía en medir los tiempos del algoritmo de generación de nivel.

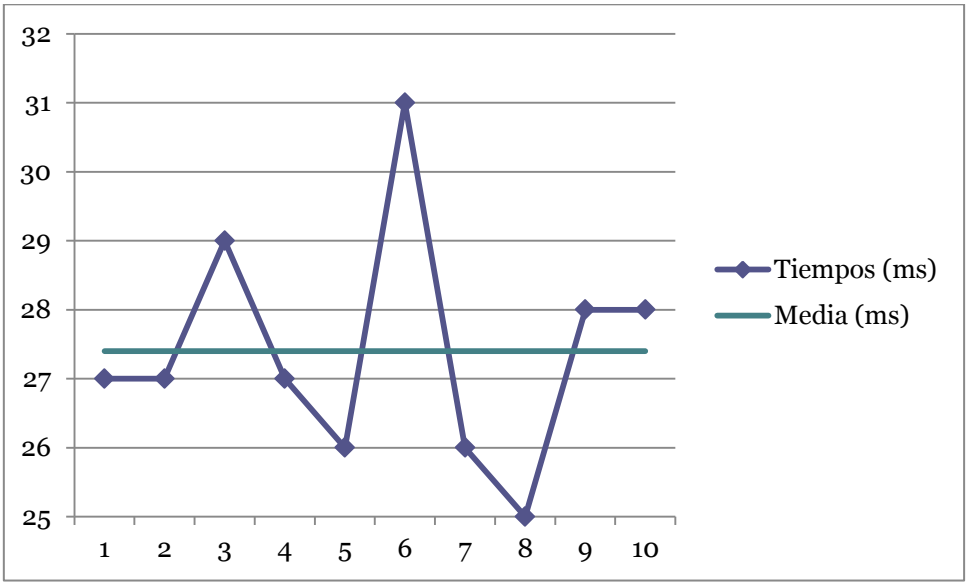


Tabla 4: Tiempos de ejecución del algoritmo de generación de nivel y su media.

Como se observa en la tabla 4, los tiempos de ejecución medidos para el algoritmo de generación de nivel, van entre 25 y 31 milisegundos; con la mayoría de tiempos más cercanos a los 25 que a los 31 ms. Estos tiempos van desde que se ejecuta el método de generación de nivel en *LevelGenerator* hasta que acaba el mismo; es decir, hasta que se han colocado todos los enemigos. De estos valores pudimos extraer la media, 27.4 milisegundos, confirmando la rapidez del algoritmo. Estos datos no son de extrañar, pues como ya se explicó, el uso de las *pools* ha servido para aumentar la rapidez del algoritmo a costa de aumentar levemente la carga de la memoria.

Estas pruebas se han realizado en tres ordenadores para comprobar distintas resoluciones, y hardware. A continuación pasamos a detallar los componentes de cada uno de ellos:

- **Ordenador 1:**
  - Procesador: *Intel Core i7-3610QM* a 2.3GHz
  - Memoria RAM: 6GB
  - Tarjeta gráfica: *Nvidia GeForce 610M*
  - Sistema operativo: *Windows 7* de 64bits
- **Ordenador 2:**

    - Procesador: *Intel Core i5-4460* a 3.2GHz
    - Memoria RAM: 8GB
    - Tarjeta gráfica: *Nvidia GeForce GTX 960*
    - Sistema operativo: *Windows 10* de 64 bits
- **Ordenador 3:**
    - Procesador: *Intel Core i5-M430* a 2.3GHz
    - Memoria RAM: 4GB
    - Tarjeta gráfica: *Nvidia GeForce GT330M*
    - Sistema operativo: *Windows 7* de 64bits

En todas las máquinas el juego ha demostrado su fluidez y era perfectamente jugable. Siendo el ordenador 3 el más obsoleto en cuanto a sus componentes, ha demostrado desenvolverse con soltura en este hardware. El ordenador 1 es un portátil en el cual fue desarrollado íntegramente todo el proyecto, por ello en este dispositivo se probó el juego tanto desde el propio motor de juego, como el ejecutable que genera al construir el proyecto. En los otros dos dispositivos sólo se probó el ejecutable, pues ninguno disponía de *Unity*. No obstante, y puesto que el ejecutable es el producto final, esto no supuso ningún problema.

Los requisitos mínimos del juego se han fijado por tanto en la configuración del tercer dispositivo, por ser el de gama más baja. No obstante, a falta de poder probarlo en dispositivos peores, seguramente los requisitos mínimos sean inferiores a la configuración de este ordenador; sobre todo en lo que a procesador se refiere.

9. Conclusiones

Durante el desarrollo del proyecto se ha ido adquiriendo una visión global de las capacidades que ofrece *Unity* a la hora de desarrollar un videojuego. Este motor ha sido todo un descubrimiento, pues ofrece la posibilidad de crear proyectos de una envergadura media, sin renunciar por ello a una curva de aprendizaje poco pronunciada y una sencillez únicas. De la misma forma, la comunidad de usuarios de *Unity* ha sido de gran ayuda a lo largo de todo el proyecto; ofreciendo tutoriales, tanto generales como específicos, y resolviendo dudas en sus foros rápidamente.

Así mismo, se ha aprendido sobre el funcionamiento de los algoritmos de generación de nivel procedural, mundo que se desconocía por completo –a nivel de programación–, y ha sorprendido por su sencillez y eficiencia. Se ha demostrado la eficacia de estos algoritmos para proyectos en los que no se dispone de mucho tiempo para su desarrollo, como es nuestro caso, evitando tener que gastar recursos y tiempo en un largo y tedioso diseño de nivel.

Podemos concluir pues el proyecto confirmando el cumplimiento de los objetivos planteados al principio del mismo, así como de su utilidad para el aprendizaje de cualquier desarrollador.

9.1 Relación del proyecto con la carrera

No obstante, y pese a todas las facilidades y ventajas que ofrece *Unity*, sería injusto afirmar que cualquier persona puede desarrollar una tarea tan compleja como un videojuego con sólo utilizar este motor. Un videojuego es un producto software muy complejo, para el desarrollo del cual se requieren unas capacidades y aptitudes concretas que ningún motor puede cubrir por simple que sea. A lo largo de la carrera se han ido aprendiendo conceptos y estructuras, así como adquiriendo habilidades, que han sido imprescindibles para la correcta realización del proyecto.

Desde el primer año de carrera se ha ido aprendiendo a programar, requisito obvio pero indispensable para todo desarrollo de un videojuego. Así pues, asignaturas como Introducción a la Informática y la Programación o Programación, han constituido la base del proyecto. Si bien es cierto que enseñan a programar en *Java*, dada la similitud de sintaxis entre este lenguaje y el utilizado, *C#*, las competencias adquiridas son escalables al proyecto. Otras asignaturas como Estructuras de Datos y Algoritmos o Algorítmica nos han enseñado estructuras de datos y algoritmos eficientes para solucionar problemas genéricos, que han servido para poder desarrollar un algoritmo de generación procedural eficiente.

También fueron útiles las nociones aprendidas en Interfaces Persona-Computador, permitiéndonos aprender las técnicas de diseño aplicadas en las interfaces del juego. Así mismo, todos los conceptos de Ingeniería del Software han sido altamente útiles, desde las fases de desarrollo del software seguidas para elaborar el proyecto, hasta las estructuras y herramientas utilizadas para crear los diagramas de clases y flujo necesarios para el diseño del proyecto.

Por lo que respecta a la inteligencia artificial, las competencias adquiridas en Agentes Inteligentes fueron de gran utilidad para los enemigos, así como para los agentes que se utilizan a lo largo del algoritmo de generación de nivel. Para la inteligencia artificial también fueron

útiles las explicaciones sobre las máquinas de estados recibidas en la optativa Introducción a la Programación de Videojuegos. Así como la explicación de managers y controladores que se explican en la estructura de un videojuego en esta asignatura. También sirvieron para el propósito de las máquinas de estados todos los conceptos aprendidos sobre grafos en asignaturas como Teoría de Autómatas y Lenguajes Formales.

Por último, si bien los conocimientos adquiridos en la asignatura Arquitectura y Entornos de desarrollo para Videoconsolas no se han podido aplicar directamente, pues estamos trabajando con un motor, han servido para valorar la utilidad de los motores de juego. Ya que en esta asignatura se trabajaba a bajo nivel sin la ayuda de ningún motor, se han visto las facilidades que ofrecen estas herramientas, permitiéndonos desarrollar videojuegos de manera mucho más rápida y simple.

9.2 Trabajo futuro

Como se indica en el GDD, Babel es un juego mucho mayor que el presentado en el proyecto. Debido a la falta de un artista que pudiera crear los *sprites* y texturas, y al poco tiempo del que se disponía, sólo se han podido hacer las mínimas para poder crear el primer mundo del juego. Así pues, se delega para un futuro la búsqueda de un colaborador que cree de cero todos los *sprites*, animaciones y texturas y añada más. De esta forma se podrán crear más mundos, cada uno con una temática y apariencia gráfica concreta, así como más enemigos característicos de cada mundo e ítems que el jugador pueda comprar.

Puesto que la generación de nivel la lleva el algoritmo desarrollado, añadir más mundos será sumamente fácil, simplemente se necesitará indicar qué texturas colocar a cada nivel, dependiendo del mundo en el que se encuentre. De esta forma, el mayor trabajo para el futuro recae sobre el artista que acepte el proyecto. No obstante, como ampliación a nivel de programación, habrá que diseñar e implementar los nuevos enemigos y efectos de los nuevos ítems. Aunque como se ha pensado en la escalabilidad desde un principio, estas novedades pueden ser implementadas fácilmente.

A pesar de ello, el mayor objetivo futuro a conseguir es seguir aprendiendo sobre el desarrollo de videojuegos, para completar Babel y, una vez acabado, poder comercializarlo en plataformas y tiendas online como *Steam*, para poder dedicarse profesionalmente a la industria.

Bibliografía

[1] Asociación Española de Videojuegos, «Noticias», [En línea]. Disponible: http://www.aevi.org.es/aevi/noticias/233-el-consumo-global-de-videojuegos-en-espana-fue-de-996-millones-de-euros-en-2014. [Último acceso: 10 de Agosto 2015].

[2] Csikszentmihalyi, M., Flow: The Psychology of Optimal Experience, Harper Perennial, 1990.

[3] Unity Documentation, «Scripting API», [En línea]. Disponible: http://docs.unity3d.com/ScriptReference/index.html. [Último acceso: 21 de Agosto 2015]

[4] Unity Community, «General performance tips», [En línea]. Available: http://wiki.unity3d.com/index.php/General_Performance_Tips. [Último acceso: 21 de Agosto 2015]

[5] Noor Shaker, Antonios Liapis, Julian Togelius, Ricardo Lopes, Rafael Bidarra, Constructive generation methods for dungeons and levels (DRAFT).

[6] Lawrence Johson, Georgios N. Yannakakis, Julian Togelius, Cellular automata for real-time generation of infinite cave levels.

[7] Mawhorter, P., Mateas, M., Procedural level generation using occupancy-regulated extension. In: IEEE Symposium on Computational Intelligence and Games (CIG), 2010.

[8] Walaa Baghdadi, Fawzya Shams Eddin, Rawan Al-Omari, Zeina Alhalawani, Mohammad Shaker, Noor Shaker, A procedural method for automatic generation of Spelunky levels.

[9] Ben Shneiderman, Catherine Plaisant, Diseño de interfaces de usuario, Pearson Educacion, 2005.

[10] Unity, «2D Roguelike Tutorial», [En línea]. Disponible: http://unity3d.com/es/learn/tutorials/projects/2d-roguelike-tutorial. [Último acceso: 23 de Agosto 2015]

[11] Ignacio Díaz, «Unity: desarrollo de una sencilla máquina de estados», [En línea]. Disponible: http://aev.org.es/unity-desarrollo-de-una-sencilla-maquina-de-estados/. [Último acceso: 23 de Agosto 2015]

[12] Microsoft Developer Network, «C# Reference», [En línea]. Disponible: http://msdn.microsoft.com/en-us/library/618ayhy6.aspx. [Último acceso: 24 de Agosto 2015]

[13] Unity, «Optimizations», [En línea]. Disponible: http://docs.unity3d.com/Manual/MobileOptimisation.html. [Último acceso: 24 de Agosto 2015]

[14] Darius Kazemi, «Spelunky generator lessons», [En línea]. Disponible: http://tinysubversions.com/spelunkyGen/. [Último acceso: 24 de Agosto 2015]

Anexos

A. Babel, documento de diseño……52

Documento de diseño

Índice

1. INTRODUCCIÓN **54**

1.1 CONCEPTO DEL JUEGO 54
1.2 CARACTERÍSTICAS PRINCIPALES 54
1.3 GÉNERO 54
1.4 PÚBLICO OBJETIVO 55
1.5 JUGABILIDAD 55
1.6 ASPECTO VISUAL 55
1.7 ALCANCE 55

2. MECÁNICAS DE JUEGO **56**

2.1 JUGABILIDAD 56
2.2 FLUJO DE JUEGO 56
2.3 PERSONAJES 57
2.3.1 Sem 57
2.3.2 Enemigos 57
2.4 OBJETOS 58
2.5 CONTROLES 58

3. INTERFACES **59**

3.1 MENÚ PRINCIPAL 59
3.2 MENÚ DE OPCIONES 59
3.3 MENÚ DE PAUSA 60
3.4 MENÚ DE CONTROLES 60
3.5 INTERFAZ DE JUEGO 61

4. RECURSOS **62**

4.1 ARTE 2D 62
4.2 ARTE DE INTERFACES 62
4.3 MÚSICA Y EFECTOS DE SONIDO 62

1. Introducción

1.1 Concepto del juego

Babel es un videojuego en el que controlaremos a un joven repartidor de pizza cuyo objetivo no es otro que cumplir con su trabajo, entregando la pizza al cliente. Este joven, llamado Sem, deberá buscar a su cliente, el arquitecto de la Torre de Babel, todavía en construcción. El arquitecto se encuentra en el nivel superior de la torre, por lo que Sem deberá subir todos los niveles de la torre para poder entregarle su pedido. No obstante, la torre está llena de peligros y enemigos, con los que Sem deberá acabar.

1.2 Características principales

El juego se basa en los siguientes pilares:

- **Planteamiento sencillo:** La historia no juega ningún papel importante en el juego, siendo algo trivial para completar el mismo. El jugador entenderá rápidamente su objetivo sin necesidad de información explícita por parte del juego; alcanzar la cima de la torre.
- **Dinamismo:** El juego debe ser dinámico y provocar sensación de tensión en el jugador. Tanto por dificultad como por necesidad de movimiento constante para no morir.
- **Ampliación:** El juego debe ser fácilmente ampliable, añadiendo nuevos niveles o «mundos» de manera sencilla, así como enemigos e ítems.
- **Combinación de géneros:** Combinará tanto plataformas como acción, con un marcado estilo *roguelike*.
- **Rejugabilidad:** Al tratarse de un juego con generación de niveles procedural, cada partida se presenta totalmente distinta a la anterior. El entorno resultará conocido al jugador, pero los niveles serán completamente diferentes.

1.3 Género

El juego supone una unión de varios géneros. A continuación se listan los géneros de los que toma elementos y sus motivos:

- **Plataformas:** Se caracterizan por tener que caminar, correr, saltar o escalar sobre una serie de plataformas y acantilados, con enemigos, mientras se recogen objetos para poder completar el juego. En Babel, este es el género principal. Nuestro personaje deberá ir desde la entrada hasta la salida de cada nivel esquivando, o acabando, con los enemigos y recogiendo la mayor cantidad de oro posible.
- **Acción:** En este género, el jugador debe usar su velocidad, destreza y tiempo de reacción para acabar con los enemigos sufriendo el menor daño posible. En el juego, el jugador podrá acabar con los enemigos, necesitando de su rapidez para acabar con ellos antes de recibir daño por su parte.
- **Roguelike:** Juegos con un énfasis en el contenido aleatorio: mazmorras generadas aleatoriamente, con enemigos, objetos, etc. En este género la jugabilidad es el aspecto primario, por encima de la estética o de la accesibilidad al jugador. Se caracterizan por su premisa sencilla con muy poca narrativa, así como por su alto nivel de dificultad; pues incorporan el concepto de «*permadeath*» o muerte permanente. Este elemento supone que cada vez que el jugador muera, deba empezar el juego desde el principio, perdiendo todas las posibles mejoras y oro que haya recaudado hasta el momento. No

obstante, puesto que los niveles se generarán de manera procedural, cada partida será diferente, consiguiendo evitar que el jugador se aburra repitiendo los mismos niveles si muere.

1.4 Público objetivo

El juego está dirigido a jugadores con una cierta experiencia en este tipo de videojuegos. Siendo un *roguelike*, puede ser frustrante debido a sus altas probabilidades de perder una y otra vez. Además de ser un juego de plataformas en el que no siempre se llegará de un punto a otro del mapa sin sufrir ningún daño, salvo que el jugador disponga de cierta habilidad. No obstante, la dificultad será asequible para que se pueda superar el juego, ya sea porque el jugador es hábil, o porque se han gastado horas en el juego aprendiendo las mecánicas y priorizando objetivos.

Por todo ello, Babel puede resultar frustrante para jugadores casuales o poco experimentados en estos géneros.

1.5 Jugabilidad

El escenario donde transcurre el juego es la Torre de Babel. El objetivo de cada nivel siempre será el mismo, llegar hasta la puerta de salida. Por el camino podremos recoger oro así como acabar con los enemigos del nivel. Para cumplir nuestro objetivo nos valdremos de los siguientes elementos:

- **Movilidad:** El jugador tendrá libertad de movimiento por el escenario en todo momento.
- **Arma:** Para enfrentarnos a los diversos enemigos de cada nivel, podremos usar nuestra espada.
- **Mejoras:** El jugador podrá comprar mejoras en unas tiendas habilitadas en los distintos niveles de manera aleatoria.

1.6 Aspecto visual

Babel tendrá un estilo simple y retro, con un *pixelart* estilo 8 bits. Los personajes, tanto el protagonista como los enemigos, serán amigables y satíricos; teniendo siempre presente un aspecto humorístico y caricaturesco que definirá al juego.

Las texturas seguirán el mismo patrón. Siendo sencillas y con un estilo *pixel* característico de juegos clásicos de los 80 y principios de los 90.

1.7 Alcance

El objetivo principal es desarrollar un sistema de juego sólido al que se le pueda introducir nuevo contenido sin dificultad. En primera instancia se desarrollará un pack de contenidos básicos que podrá ser ampliado en un futuro. Este pack inicial consistirá en: un mundo formado por cinco niveles, tres enemigos y cuatro ítems.

2. Mecánicas de juego

En esta sección se comentarán todos los pilares que fundamentan la jugabilidad del juego y se detallarán las acciones que podrá llevar a cabo el jugador dentro de una partida típica. Además se ofrecerá una lista con los personajes del juego –tanto protagonista como enemigos–, habilidades, objetos, etc.

2.1 Jugabilidad

- **Mundos:** La Torre de Babel estará dividida en mundos, cada uno con una ambientación diferente. Para superar el juego, el jugador deberá superar estos mundos. Cada mundo estará dividido en cinco niveles. En la primera versión del juego, sólo se implementará el primer mundo de juego.
- **Niveles:** Los niveles serán generados de manera procedural. En los distintos niveles podremos encontrar distintos enemigos, cofres que saquear, o tiendas donde comprar mejoras. Cada nivel contará obligatoriamente con una puerta de entrada y otra de salida. El jugador empezará siempre en la puerta de entrada, situada en la parte inferior del nivel, y deberá alcanzar la parte superior del mismo para llegar a la puerta de salida.
- **Intensidad:** La dificultad del juego viene dada tanto por la cantidad de enemigos, como la de obstáculos del nivel. Al ser un juego de generación de nivel procedural, no se puede medir con demasiada exactitud la dificultad entre dos niveles de un mismo mundo. Pero sí que se irá aumentando la dificultad entre los distintos mundos, siendo cada mundo más complejo y difícil que el anterior. Tanto por presentación de nuevos enemigos, como de mayor concentración de obstáculos a esquivar.

2.2 Flujo de juego

A lo largo de esta sección se detallará el transcurso de una partida típica del juego. Se comentarán los pasos que ha de seguir el jugador desde el comienzo del juego hasta completar un nivel.

El jugador inicia el juego y se le presenta la pantalla de título, con el menú principal. El jugador podrá: iniciar un nuevo juego, cambiar la configuración del juego accediendo al menú opciones, visualizar los créditos del juego, o salir del mismo.

Cada nivel comienza con Sem delante de una puerta cerrada, la puerta de entrada al nivel. No se le indicará al jugador en ningún momento cuál es su objetivo, deduciendo el propio jugador, de manera intuitiva, cuál es el mismo. El jugador deberá avanzar por la zona hasta encontrar la salida de la misma. Mientras avanza se enfrentará a diferentes obstáculos y enemigos. Será decisión del jugador esquivarlos o enfrentarse a los mismos. También podrá ir recogiendo el oro que vaya encontrando a lo largo de cada nivel, el cual se irá acumulando.

Las diferentes herramientas o ítems se obtendrán durante el transcurso de los niveles. Ya sea comprándolos en las tiendas que podrá haber en cada nivel –recordemos que se generan de manera pseudoaleatoria– u obteniéndolas en cofres.

El jugador será derrotado si sus puntos de vida se reducen a cero. Esto puede ocurrir por diversos motivos: ser tocado, o recibir un ataque, por parte de un enemigo; ser golpeado por algún proyectil; o caer o chocar contra pinchos.

Para poder curarse algún punto de salud, el jugador deberá comprar una porción de pizza en alguna tienda.

Si el jugador muere, deberá empezar el juego desde el principio. Además perderá todas las mejoras y oro que haya conseguido durante la partida. Exceptuando las armas y utensilios básicos que se dan desde el principio del juego. Las armas y objetos se detallarán más adelante.

Cuando se supere un nivel, el jugador aparece en una sala con dos puertas, una de las cuales estará cerrada. Será la transición de nivel. El jugador sólo podrá avanzar hasta la puerta abierta y entrar por la misma, dando comienzo al siguiente nivel.

En todo momento el jugador podrá detener el juego accediendo al menú de pausa. Desde el mismo podrá cambiar la configuración del juego desde opciones, ver los controles, volver al menú principal, o salir del juego. En estos dos últimos casos se perderá todo el progreso.

Salud: Nuestro personaje contará inicialmente con cinco puntos de salud. Cada vez que un enemigo le toque o le ataque, sea alcanzado por un proyectil, o choque contra algún elemento dañino, perderá un punto de salud. La salud es permanente entre niveles y mundos. Es decir, no se recuperará la salud de un nivel a otro, manteniéndose la salud con la que se haya acabado el nivel anterior. La salud no afecta al jugador en nada, mientras se tenga salud se podrá jugar. Cuando llegue a cero, se acabará la partida.

2.3 Personajes

En esta sección procederemos a enumerar y describir a todos los personajes de Babel, tanto al protagonista como a los enemigos. También se explicarán sus habilidades.

2.3.1 Sem

Joven de unos 20 años. Viste el uniforme de los repartidores de pizza: toga árabe simple atada a la altura de la cintura y gorra con muelle y un trozo de pizza en el extremo del mismo. Paleta de colores: rojizos, marrones.

Salud inicial: 5 puntos de salud.

Arma: Espada.

Capacidades físicas: Sem podrá saltar, trepar por cuerdas o escaleras, realizar un *dash*, atacar con la espada y lanzar cuerdas.

2.3.2 Enemigos

A continuación se presenta una tabla con todos los enemigos del juego, así como sus puntos de salud y de daño:

Enemigo	Puntos de salud	Puntos de daño	Características
Capataz	1	1	Enemigo terrestre. Rango de ataque aumentado.
Obrero	2	1	Enemigo terrestre. Velocidad aumentada.
Soldado	1	1	Enemigo terrestre. Dispara proyectiles.

2.4 Objetos

En este apartado listaremos todos los objetos del juego. Así como explicar su funcionalidad y precio.

Los objetos se compran en las tiendas repartidas por los niveles de los distintos mundos del juego. Aunque el jugador no puede utilizarlos, le aportan una serie de mejoras pasivas. Una vez el jugador muere, pierde todos los objetos.

Objeto	**Precio**	**Descripción**
Cohete	3000	Permite hacer doble salto.
Porción de pizza	5000	Recupera un punto de salud.
Sandalias con muelle	1000	Salto aumentado.
Túnica acorazada	1500	Cuando el jugador impacta con un enemigo recibe un punto de daño y se hace un punto de daño.

2.5Controles

El juego se podrá jugar con teclado y ratón o con *gamepad* si el jugador dispone de él. La lista de controles se detalla a continuación:

Movimiento:

Controles	**Teclado y ratón**	**Xbox 360 *Controller Pad***	**Descripción**
Desplazarse Izquierda/Derecha	A/D	*Stick* izquierdo	Mover al personaje.
Saltar	Barra espaciadora	A	Si se pulsa mientras se apunta una dirección, saltará en esa dirección. En caso contrario, saltará en vertical.
Dash	O/P	LT/RT	El personaje realizará un pequeño sprint hacia la izquierda o la derecha.

Acciones:

Controles	**Teclado y ratón**	**Xbox 360 *Controller Pad***	**Descripción**
Pausa	Esc	*Start*	Pausa el juego presentando el menú de pausa.
Interaccionar	W	Apuntar hacia arriba con el *stick* izquierdo.	Entra en la puerta del final del nivel, o compra algún objeto.
Lanzar gancho	U	Y	Dispara el gancho con cuerda hacia

			arriba.

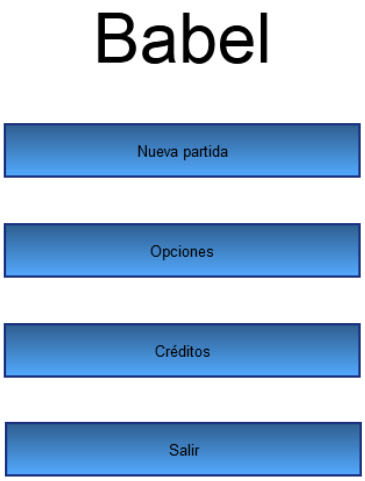
Combate:

Controles	**Teclado y ratón**	**Xbox 360 *Controller Pad***	**Descripción**
Atacar	F	X	El personaje ataca con la espada.

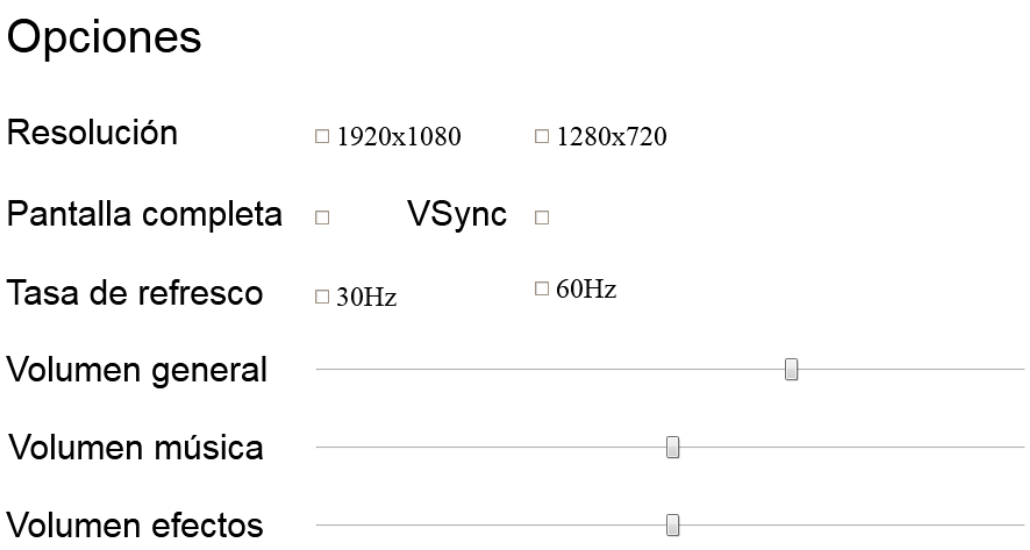
3. Interfaces

A continuación enseñaremos la estructura de cada menú e interfaz del juego.

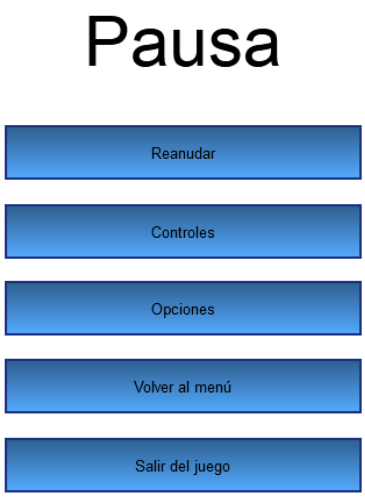
3.1 Menú principal



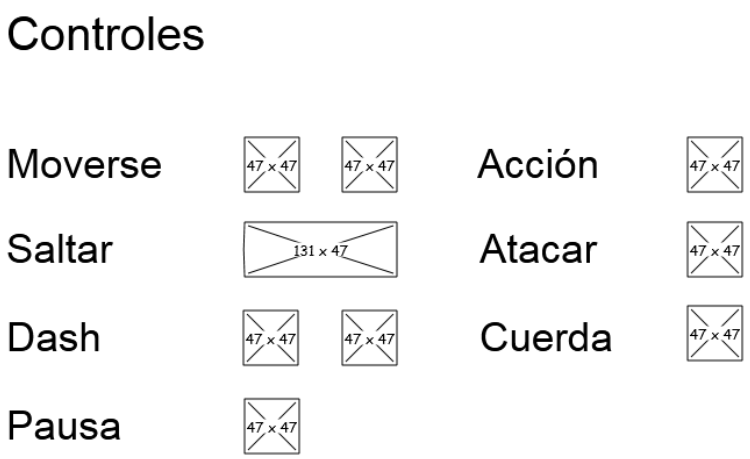
3.2 Menú de opciones



3.3 Menú de pausa

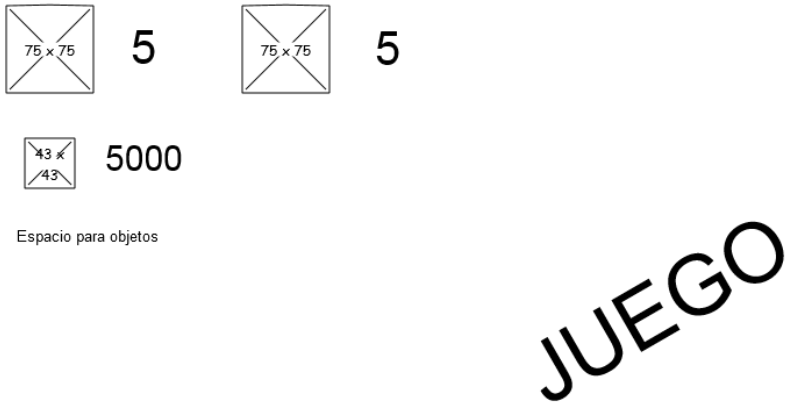


3.4 Menú de controles



Las imágenes de este menú serán iconos de las teclas para realizar las acciones correspondientes.

3.5Interfaz de juego



En la interfaz de juego la primera imagen será una imagen de una porción de pizza, representando las vidas del jugador. La segunda imagen –a la derecha de la anterior–, será la imagen de una cuerda con un gancho, siendo estas las cuerdas que le queden al jugador por lanzar. La imagen pequeña inferior, será una moneda, indicando el dinero del que dispone el jugador.

4. Recursos

4.1 Arte 2D

Todos los *sprites*, animaciones y texturas serán creadas por el autor, dado que no se dispone de la colaboración de un artista, y descargarlas por internet limitaría el arte del juego al que se encontrara con licencias abiertas.

4.2 Arte de interfaces

www.opengameart.org

4.3 Música y efectos de sonido

www.opengameart.org

www.freesound.org