UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escuela Técnica Superior de Ingeniería Informática

Universitat Politècnica de València

Steel Soldier: Un juego de plataformas-shooter desarrollado en Godot

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: David Morales Cortés

Tutor: Dr. Ramón Pascual Mollá Vayá

2015 - 2016

Resumen

El proyecto consiste en desarrollar desde cero un videojuego en dos dimensiones haciendo uso del motor gráfico Godot, de código abierto. El juego está dividido en diferentes fases, en todas ellas el jugador debe avanzar saltando por el escenario para superar obstáculos y enemigos. Además, se distinguen fases de acción con disparos y fases de infiltración en las que el jugador debe pasar desapercibido ante los enemigos hasta llegar al final de la fase. Se hará uso de una adaptación de la API de Inteligencia Artificial usada en el UPV Game Kernel. El objetivo final es experimentar todas las etapas de desarrollo de un videojuego generando una demostración práctica de la aplicación de la API ya mencionada y del motor de juegos Godot

Palabras clave: Videojuego, 2D, Plataformas, Shooter, Godot, Inteligencia Artificial, PC, UGK

Abstract

The project consists in developing a two-dimensional game from scratch using the Godot game engine, which is open source. The game is divided in different phases, in all of them the player must advance jumping through the stage in order to evade obstacles and enemies. Furthermore, there is a distinction between action phases which involve shooting and infiltration ones where the player must remain undetected by the enemies until the end of the level. An adaptation of the Artificial Intelligence API used in UPV Game Kernel will be used. The final goal is to experiment every stage of a videogame development process and thus generating a practical demonstration of usage of the previously mentioned API and the Godot game engine.

Keywords: Videogame, 2D, Platforms, Shooter, Godot, Artificial Intelligence, PC, UGK

Tabla de contenidos

1. Introducción

1.1. Motivación

La principal motivación tras el desarrollo de este proyecto, así como de cursar esta carrera, ha sido realizar un acercamiento al mundo del desarrollo de videojuegos de forma profesional. Es un hecho constatado que los videojuegos han ganado mayor peso no solo como productos de ocio, sino también como un fenómeno sociocultural. Poco a poco, la sociedad ha dejado de ver el videojuego como un producto nocivo y comienza a tratarse también con mayor seriedad en el ámbito laboral.

Hoy día es cada vez más frecuente ver la aparición de pequeños estudios independientes que, gracias a las facilidades que otorgan herramientas como *Godot*, *Unity* y otros motores gráficos, junto con la aparición de plataformas de publicación digital como *Steam* o *Google Play Store*, son capaces de desarrollar y hacer llegar sus proyectos al público de forma mucho más sencilla.

Si bien es cierto que resulta más sencillo desarrollar y publicar videojuegos de forma independiente, esto también conlleva un mayor número de productos disponibles[1] y, por tanto, la competencia es mucho mayor[2]. Para lograr crear un videojuego que se diferencie del resto es necesario que el diseño del mismo explore nuevas direcciones tanto en la jugabilidad como en la narrativa del videojuego. Dotar al videojuego de esa personalidad propia que lo haga destacar del resto requiere un conocimiento profundo de las herramientas de trabajo, así como ser capaz de encontrar soluciones a los nuevos retos de programación que la visión de los diseñadores puede plantear.

Este proyecto pretende ser la base para conocer la metodología propia del desarrollo de un videojuego. Para ello se ha decidido trabajar con el motor *Godot*, dada su versatilidad y el grado de personalización que admite, con el objetivo de crear un primer videojuego que aporte experiencia al equipo de desarrollo y que resulte divertido desde el punto de vista jugable, basado en juegos clásicos cuyo

[1] Presentación de la *Game Developers Conference*, donde se pueden ver datos relativos al crecimiento de *Steam* como plataforma de distribución en las diapositivas 33 y 34: http://www.eedar.com/Pres/EEDAR%20-%20GDC2016%20-Awesome%20Video%20Game%20Data%20Distribute%20%5bGeoffrey%20Zatkin%5d%20v2.7.pdf

[2] Artículo de *PCGamer* sobre la plataforma *Steam*, incluye el punto de vista de algunos desarrolladores independientes y reflexiones sobre el lanzamiento masivo de juegos y sus consecuencias: http://www.pcgamer.com/steam-games-number-of-too-many/

funcionamiento resulta familiar y a la vez supone un grado de dificultad apropiado para la realización de un Trabajo de Fin de Grado.

Cabe destacar que la realización de un videojuego implica el uso de conocimientos adquiridos en distintas áreas del Grado, tales como ingeniería del software, gestión de proyectos, inteligencia artificial, algorítmica, etc. Es por tanto una forma adecuada de englobar los distintos ámbitos cubiertos a lo largo del Grado.

1.2. Objetivos

El objetivo principal de este proyecto es diseñar y desarrollar un prototipo de un juego de plataformas y disparos en un entorno bidimensional. El juego contará con una versión de demostración para PC.

El prototipo estará centrado en una de las diferentes fases que se detallan en el documento de diseño del juego, la cual hace énfasis en el aspecto de las plataformas e infiltración. Dicho prototipo incluye toda la funcionalidad del personaje principal, incluyendo la interacción con el escenario del juego, además tres de los personajes enemigos. Así se logra ofrecer una experiencia de juego pulida y se establece la base necesaria para ampliar y completar el resto del proyecto.

Se busca conocer mejor el motor de videojuegos *Godot* como herramienta de desarrollo, familiarizarse con el flujo de trabajo y las posibilidades que este ofrece. Esta experiencia sirve también como una preparación al mundo laboral, en el que a menudo se requiere una adaptación rápida a herramientas que no se han usado previamente.

Este proyecto también proporcionará un ejemplo extenso del uso del motor. Se espera que este ejemplo pueda ayudar a otros compañeros a conocer la herramienta y fomentar su uso en proyectos futuros, haciendo crecer la comunidad de desarrolladores.

El otro gran objetivo del trabajo es emplear en un caso práctico la Interfaz de Programación de Aplicaciones (API, *Application Programming Interface*) de Inteligencia Artificial (IA) del UPV Game Kernel (UGK), adaptándola al motor Godot para suplir la carencia de herramientas de gestión de la IA del mismo. Mediante esta herramienta se integrarán máquinas de estado para controlar el comportamiento de los personajes enemigos y comprobar el funcionamiento de la misma fuera del UGK.

Por último, mirando al futuro, uno de los objetivos de este proyecto es también contactar con otros compañeros de la Universidad interesados en formar parte de un

pequeño equipo de desarrollo para continuar trabajando juntos una vez realizado el trabajo que ahora nos ocupa.

1.3. Estructura de la obra

Tras haber pasado el apartado de introducción y comprender la motivación y los objetivos del trabajo, se pasa a enumerar los distintos apartados que componen esta obra.

En el apartado 2 se trata el estado del arte, comenzando con una pequeña introducción. En el punto 2.1. se da un repaso a diferentes motores de videojuego y herramientas de desarrollo disponibles para realizar el proyecto. El punto 2.2. repasa las características de algunos videojuegos existentes que han servido como inspiración al proyecto. En el punto 2.3. se hace una crítica al estado del arte y se justifica la elección de herramientas realizadas. El punto 2.4. realiza una propuesta para mejorar el estado del arte, y el 2.5. es una pequeña introducción al motor UPV Game Kernel.

El apartado 3 trata sobre el análisis del problema. En el punto 3.1. se realiza un análisis DAFO (Debilidades, Amenazas, Fortalezas y Oportunidades), a partir del que se obtienen estrategias a seguir para realizar el proyecto. El punto 3.2. trata de la planificación temporal del proyecto. En el punto 3.3. se hace un análisis de la internacionalización del juego y finalmente el punto 3.4. habla de la colaboración realizada con otros compañeros durante el desarrollo del trabajo.

El apartado 4 abarca el diseño del juego desde el punto de vista del software. El punto 4.1. es una pequeña introducción a conceptos básicos del motor Godot. El punto 4.2. explica el concepto del juego a nivel general. El punto 4.3. abarca el diseño de los componentes principales del juego incluyendo diagramas de clase del personaje principal, enemigos, elementos interactivos, mapa, etc. El punto 4.4. trata sobre el diseño de la interfaz y cuenta con diagramas de casos de uso. Finalmente, el punto 4.5. da una visión global de la estructura del juego ofreciendo un esquema detallado de la conexión entre las escenas que lo componen.

El apartado 5 trata de la implementación del juego. Siguiendo la planificación temporal establecida en el punto 3.2., se detalla en cada uno de sus puntos todo lo relativo a la implementación de cada elemento del juego.

El apartado 6 está dedicado a los resultados obtenidos. En el punto 6.1. se analizan dichos resultados viendo qué objetivos se han cumplido, cuál ha sido la diferencia entre la planificación inicial y el tiempo real dedicado al proyecto, etc.

El apartado 7 desarrolla las conclusiones obtenidas tras realizar el trabajo. En el apartado 7.1. se habla de la relación del trabajo con los estudios cursados y en el apartado 7.2. se hace una reflexión acerca de posibles trabajos futuros con los que ampliar el videojuego.

En el apartado 8 se incluye una serie de agradecimientos. Finalmente, existen cuatro anexos. El anexo 1 contiene la bibliografía comentada del trabajo. El anexo 2 es un glosario de términos recurrentes. El anexo 3 es una guía detallada de cómo empezar a desarrollar videojuegos usando el motor Godot. Por último, el anexo 4 es el documento de diseño del juego, donde se amplía la información referente al diseño del juego.

2. Estado del arte

En este apartado va a realizarse un análisis de distintos motores y herramientas disponibles para desarrollar videojuegos. Tras esto se analizará el estado del arte mediante un repaso a videojuegos ya existentes. A continuación, se proporcionará una visión global del estado actual del motor de desarrollo escogido para el proyecto, *Godot.* Se expondrán algunas de las debilidades del motor para luego realizar una propuesta tratando de mejorarlo. También se incluye una pequeña introducción al UGK, así como un apartado en el que se revisan diferentes juegos existentes que han servido como inspiración para el diseño del juego.

2.1. Herramientas disponibles

Existe una gran variedad de motores y *frameworks* orientados a la creación de videojuegos, por lo que en principio puede resultar complicado decantarse por uno de ellos a la hora de realizar un primer proyecto. Por ello es conveniente examinar los distintos aspectos de diferentes alternativas a fin de poder seleccionar el más adecuado, acorde con la experiencia del equipo y las necesidades del proyecto.

Estos son los criterios que se tendrán en cuenta para realizar el análisis:

- Orientación a la creación de juegos 2D o 3D
- Código abierto
- Precio de la licencia
- Documentación disponible
- Editor gráfico
- Capacidad de desarrollo multiplataforma
- Lenguaje empleado para la programación o *scripting*

Partiendo de estos criterios, se da paso a una pequeña presentación de las cinco herramientas distintas analizadas en detalle. Tras esta sección se encuentra una tabla comparativa que resume los resultados del análisis realizado.

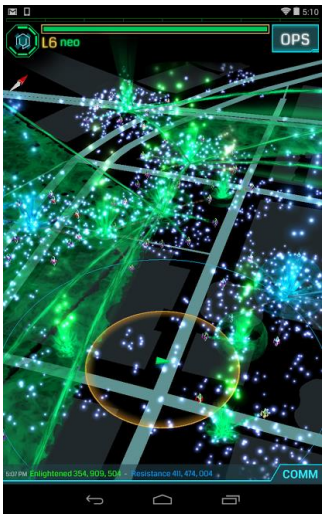


Ilustración 1: Imagen de Ingress, del estudio Niantic, desarrollado con libGDX

libGDX[3], de *Badlogic Games*, es un f*ramework* orientado a la creación de juegos en 2D y destaca por su gran eficiencia. El proyecto es de código abierto bajo licencia Apache 2.0 y su uso es completamente gratuito. La documentación del proyecto es muy completa y se ve complementada por una gran comunidad de desarrollo encargada de mantener el proyecto. Carece de editor gráfico. Puede funcionar con un IDE como Eclipse, pero esto dificulta su uso por miembros del equipo sin experiencia programando. Permite desarrollar juegos para las principales plataformas de escritorio y móvil. El lenguaje de programación empleado es Java, el cual se ha estudiado en profundidad a lo largo del Grado, lo cual facilita un primer acercamiento a su uso. Es en definitiva una propuesta muy sólida, lastrada únicamente por la carencia de un editor gráfico que permita trabajar conjuntamente a todo el equipo de desarrollo.

Ilustración 2: Captura de Crypt of the NecroDancer, de Brace Yourself Games, desarrollado con Monkey X

[3] Página oficial de libGDX: https://libgdx.badlogicgames.com/

Monkey X[4], desarrollado por *Blitz Research Ltd.*, es una plataforma de creación de videojuegos enfocada a títulos en 2D. El compilador y un gran número de módulos son de código abierto, pero no lo es todo el proyecto. Hay versiones gratuitas, pero estas no permiten exportar los proyectos a todas las plataformas disponibles en la versión de pago, disponible a partir de 99 dólares. Cuenta con documentación suficiente para aprender los fundamentos básicos y empezar a desarrollar. Carece de editor gráfico, requiere el uso de un IDE propio llamado *Jungle*. La versión gratuita incluye capacidad de desarrollo para plataformas de escritorio y HTML 5, pero no plataformas móviles. El lenguaje empleado es *Monkey*, un lenguaje propio. En conjunto, no es una herramienta atractiva para el desarrollo de este proyecto.

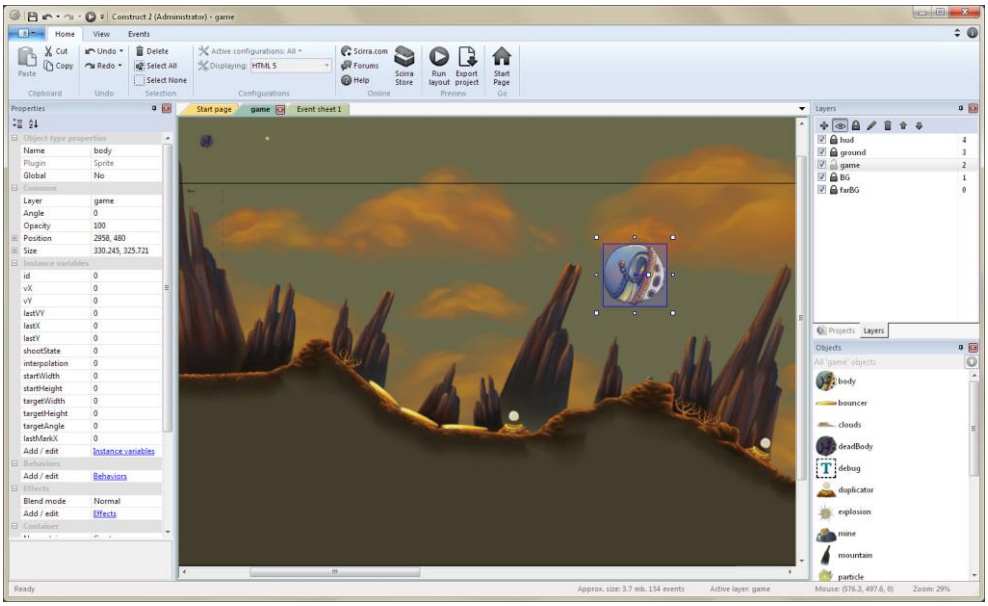


Ilustración 3: Editor gráfico de Construct2

Construct 2[5], de *Scirra*, es un motor de creación de juegos en 2D. Esta versión del motor no es de código abierto, y la licencia de desarrollo cuesta 99,99 dólares. La documentación está formada por un manual de uso, tutoriales y guías para principiantes y resulta muy completa. El editor gráfico es la gran fortaleza de este motor, ya que es muy potente y minimiza los conocimientos de programación necesarios para crear un videojuego. Es un motor basado en HTML5, por lo que es posible desarrollar juegos para entornos web, de escritorio y móviles. Es posible escribir módulos adicionales con el lenguaje Javascript para ampliar las capacidades del motor. Es una alternativa sólida para diseñadores y otros profesionales del sector que quieran trabajar sin tener amplios conocimientos de programación. Sin embargo,

[4] Página oficial de Monkey X: http://www.monkey-x.com/
[5] Página oficial de Construct 2: https://www.scirra.com/construct2

no es una alternativa que se ajuste a los requisitos de este TFG, dado el escaso control a nivel de código que ofrece.

Ilustración 4: The Next Penelope, de Aurelien Regard, desarrollado con Construct2

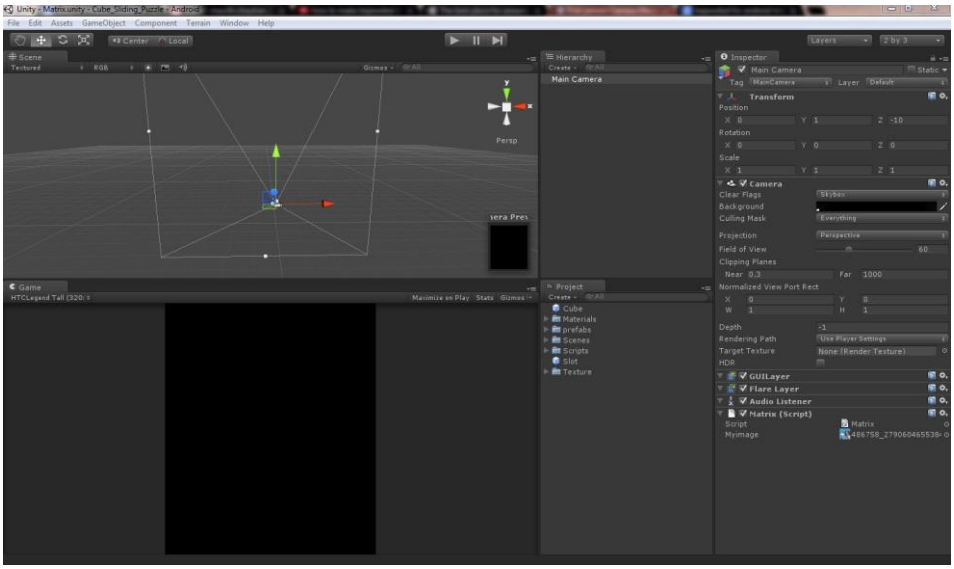


Ilustración 5: Editor gráfico de Unity

Unity[6], de *Unity Technologies*, es un motor de juegos en 3D y 2D, siendo el módulo de edición 2D un añadido de las últimas versiones. Existen algunos componentes de código abierto, pero gran parte del código del motor no es accesible. Su uso gratuito no cuenta con grandes limitaciones y permite desarrollar pequeños proyectos personales y académicos. Para juegos comerciales, es necesario pagar una suscripción si el equipo responsable factura más de 100.000 dólares anuales. La documentación disponible es muy extensa, y además cuenta con un gran número de tutoriales, tanto básicos como avanzados, para aprender a utilizar el motor. Ofrece una interfaz gráfica muy potente que permite editar las escenas del juego y organizar el proyecto. El desarrollo multiplataforma es uno de los estandartes de Unity, que permite

[6] Página oficial de Unity en español: https://unity3d.com/es/

soporte para plataformas de escritorio, móviles, web y también es compatible con todas las consolas que hay en el mercado. La programación puede hacerse empleando el lenguaje C# o Javascript. Teniendo en cuenta el gran parecido de C# y Java, el lenguaje más estudiado en el Grado, esto supone una gran ventaja a su favor. Es probablemente la alternativa más sólida de todas, aunque está orientado principalmente a juegos en 3D. También cuenta con todo tipo de módulos y extensiones, entre ellos la versión original de la API de gestión de IA, desarrollada por José Alapont Luján como TFM en la UPV. Por tanto, este proyecto no podría aportar nada nuevo.

Ilustración 6: Ori and the Blind Forest, de Moon Studios, desarrollado en Unity

Ilustración 7: Editor gráfico de Godot

Godot[7], de *Okam Studio*, es un motor centrado en el desarrollo de juegos en 2D, con posibilidad de crear juegos en 3D. El proyecto es de código abierto y fácilmente

[7] Página oficial de Godot: https://godotengine.org/

ampliable. Su uso es completamente gratuito, no requiere ningún tipo de licencia ni porcentaje de beneficios para publicar juegos comerciales. La documentación, como se ha mencionado en el apartado de estado del arte, resulta escasa. El editor gráfico es similar al ofrecido por Unity, pero cuenta con funcionalidades adicionales enfocadas al desarrollo en 2D. El editor está disponible en Windows, Max y Linux, y permite desarrollar juegos para estas plataformas junto con Android y HTML5 entre otras. Los módulos adicionales para ampliar el motor pueden escribirse utilizando el lenguaje C++. El *scripting* emplea un lenguaje propio, GDScript, basado en Python. Es la herramienta elegida para el desarrollo por los motivos que se han especificado en el apartado anterior de la memoria.

Ilustración 8: The Interactive Adventures of Dog Mendoça and Pizza Boy, de Okam Studio, desarrollado en Godot

Así pues, esta es la tabla comparativa de las herramientas analizadas:

	libGDX	Monkey X	Construct 2	Unity	Godot
Orientación 2D	Sí	Sí	Sí	No	Sí
Código abierto	Sí	Parcial	No	Parcial	Sí
Precio	Gratuito	Versión gratuita limitada	99,99$	Gratuito para proyectos académicos	Gratuito
Documentación	Extensa	Escasa	Extensa	Extensa	Escasa
Editor gráfico	No	No	Sí	Sí	Sí
Multiplataforma	Sí	Sí	Sí	Sí	Sí
Lenguaje	Java	Monkey	JavaScript	C# y JavaScript	GDScript y C++

Ilustración 9: Tabla comparativa de herramientas de desarrollo

Como puede verse, Godot cumple prácticamente todos los requisitos, fallando únicamente en el aspecto de la documentación. Aunque esto suponga inicialmente un

obstáculo ya que requerirá un mayor tiempo de adaptación a las herramientas, justifica una vez más la realización del proyecto como apoyo al crecimiento y la implantación de la tecnología empleada.

2.2. Videojuegos similares

En el mercado existen algunos videojuegos que han servido como inspiración al proyecto. El objetivo no es tanto desarrollar un concepto especialmente innovador sino llevar a cabo un proyecto didáctico que refuerce los conocimientos adquiridos durante el Grado y pueda ser empleado por otros compañeros como base para futuros trabajos.

Actualmente existen dos tendencias bien diferenciadas: por una parte, los grandes desarrollos con gráficos hiperrealistas y grandes presupuestos. Por otra parte, están los desarrollos más humildes llevados a cabo por estudios independientes, que tratan de ofrecer juegos muy distintos a las grandes superproducciones.

A la hora de tratar de poner nuevas ideas sobre la mesa es una práctica habitual mirar a los juegos clásicos y tratar de ofrecer un nuevo enfoque posibilitado por las herramientas actuales. Este ejercicio de retrospección también se ha realizado de forma previa al diseño del juego, poniendo especial interés en los siguientes videojuegos:

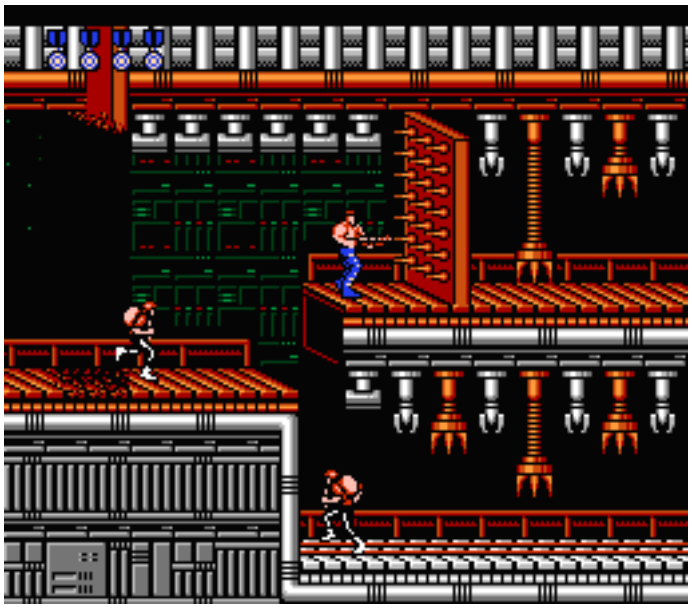


Ilustración 10: Captura del videojuego Contra

Contra, un juego clásico publicado en 1987 por la compañía japonesa *Konami*[8] en la consola *Nintendo Enternainment System* (NES). Es un juego en dos dimensiones en el que uno o dos jugadores avanzan por escenarios repletos de enemigos. Los jugadores cuentan con munición infinita para sus armas, por lo que el juego está más enfocado en disparar a los enemigos y esquivar sus ataques, primando la acción sobre la estrategia. Al contrario que otros juegos de disparos de la época, el avance no era automático, sino que eran los jugadores los que se movían a voluntad por el escenario y este contaba con algunos elementos muy básicos de plataformas. Estos elementos básicos se mantuvieron en los posteriores juegos de la franquicia, recordada por muchos jugadores por su alta dificultad. Fue uno de los primeros representantes de un subgénero conocido como *Run and Gun*, traducidos a menudo como “Corre y dispara”.

Ilustración 11: Captura del videojuego Metal Gear

Metal Gear, una de las primeras sagas de infiltración aparecida por primera vez en 1987 para la consola MSX, también de la mano de *Konami*. En los juegos de la saga prima el sigilo y la capacidad de pasar desapercibido dentro de una base enemiga. El protagonista emplea diversas herramientas para completar su misión con éxito, evitando en todo momento la confrontación directa con los enemigos. Pese a que las entregas originales utilizaban gráficos en 2D, la perspectiva cenital que empleaban ayudaba a dar la sensación de estar en un espacio tridimensional. Con el salto a las 3D en el título *Metal Gear Solid*, la trama ganó importancia y los juegos han dejado para el recuerdo algunos de los personajes más carismáticos de la historia del medio, así como una historia de guerra y espionaje a la altura de otros medios como el cine o la literatura.

[8] Sitio web de Konami: https://www.konami.com/en/

Ilustración 12: Captura del videojuego Mark of the Ninja

Mark of the Ninja, desarrollado por *Klei Entertainment*[9] y publicado en el año 2012 por *Microsoft Studios*, es uno de los mejores y más recientes exponentes del género de sigilo. Cuenta con escenarios en 2D en los que nuevamente es importante pasar desapercibido ante los enemigos. Para ello el juego pone a disposición del jugador diversos medios, como puedan ser provocar ruidos para despistar a un vigilante de seguridad, apagar las luces de un edificio, o simplemente usar las capacidades acrobáticas del protagonista para saltar por plataformas y aprovechar escondites. La mezcla de estas habilidades da como resultado una jugabilidad sólida y divertida que abre la puerta a superar las distintas fases del juego utilizando distintas estrategias. Llama la atención la forma en la que los enemigos reaccionan a las estrategias del jugador, signo de una IA bien planteada, que ayuda a hacer el universo del juego mucho más creíble. Destaca también su apartado artístico, que recuerda al del cómic en el diseño de los personajes y juega de forma muy acertada con el contraste entre las luces y sombras dentro de los escenarios.

[9] Sitio web de Klei Entertainment: https://www.kleientertainment.com/

Ilustración 13: Captura del videojuego Super Mario Bros.

Super Mario Bros., lanzado por *Nintendo*[10] en el año 1985. Es uno de los iconos culturales del mundo del videojuego y sentó las bases de los juegos de plataformas. Sus mecánicas sencillas y su diseño de niveles fueron la puerta de entrada de millones de jugadores al mundo de los videojuegos en general y los juegos de plataformas en particular. A día de hoy sigue siendo uno de los juegos más influyentes y una referencia clave con la que aprender los fundamentos del diseño de niveles para decenas de juegos que siguen sus pasos.

A continuación, se muestra una tabla resumen de los elementos característicos e ideas de cada videojuego que ha recogido Steel Soldier:

	Contra	Metal Gear	Mark of the Ninja	Super Mario Bros.
Elementos de juego	Enemigos agresivos en las fases de acción	Énfasis en la IA de los enemigos, trabajo coordinado de los mismos	Escondites, zonas de luz y sombra que alteran la visibilidad enemiga	Diferentes tipos de plataforma
Mecánicas de juego	Disparos como medio de defensa ante los enemigos	Estado de calma y alerta de los enemigos	Interacción con los escenarios para alterar la conducta de los enemigos	Saltos y uso de plataformas para evitar obstáculos y avanzar
Objetivos	Avanzar hacia el final de la fase eliminando a los atacantes	Infiltrarse en terreno enemigo, liberar rehenes	Llegar al final de la fase sin ser descubierto	Emplear la habilidad del protagonista para avanzar por los niveles

Ilustración 14: Tabla de mecánicas de juego

[10] Sitio web de Nintendo Europa: https://www.nintendo-europe.com/

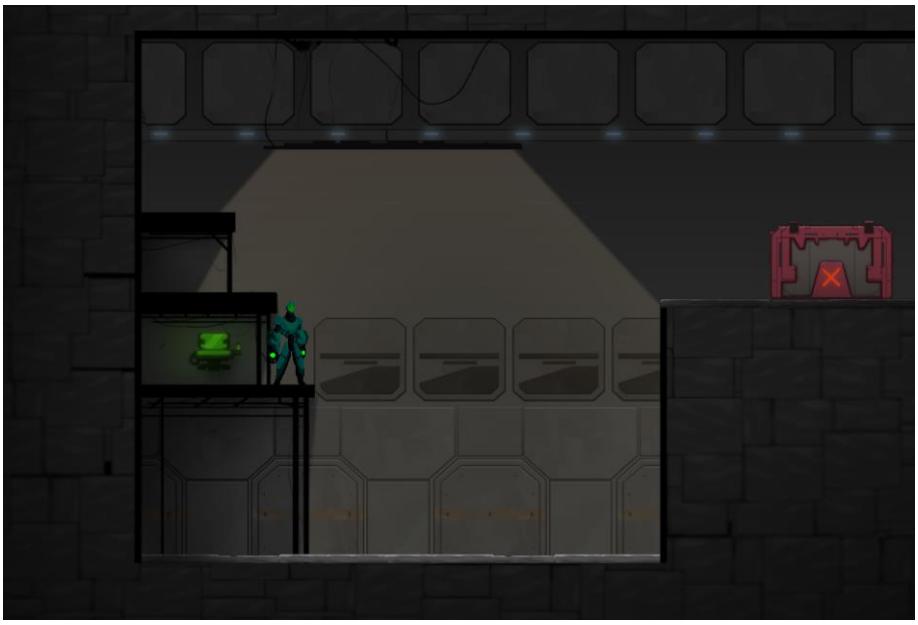


Ilustración 15: Captura del videojuego Steel Soldier

Steel Soldier recoge estas ideas y las mezcla ofreciendo dos tipos distintos de nivel: de acción e infiltración. Las fases de acción son más similares a un videojuego del estilo *Contra*, frenéticas y decenas de enemigos, con énfasis en los disparos. Las fases de infiltración hacen mayor hincapié en los elementos de plataformas e interactuar con el escenario para pasar desapercibido. Además, se incluye una serie de conceptos nuevos como, por ejemplo:

- Enemigos mecánicos indestructibles
- Sistema de terminales para alterar la seguridad del complejo
- Distintos tipos de armas con diferentes propiedades

En definitiva, el uso de conceptos previamente existentes junto con nuevas propuestas busca dar sensación de familiaridad al jugador con experiencia y proponer también una experiencia interesante para los nuevos jugadores. La mezcla de conceptos tiene como resultado un videojuego con carácter propio, diferenciado de las otras alternativas ya existentes.

2.3. Crítica al estado del arte

El motor de juego *Godot* es una herramienta de desarrollo potente a pesar de ser una tecnología joven, cuya primera versión estable abierta al público apareció en diciembre de 2014. Al contrario que otras alternativas, este motor está enfocado principalmente al desarrollo de juegos en 2D. Por esta razón incluye una serie de utilidades que

facilitan la tarea de creación de este tipo de juegos. Ofrece además una interfaz funcional y amigable que facilita su uso no solo por programadores, sino también por artistas u otros miembros del equipo de desarrollo.

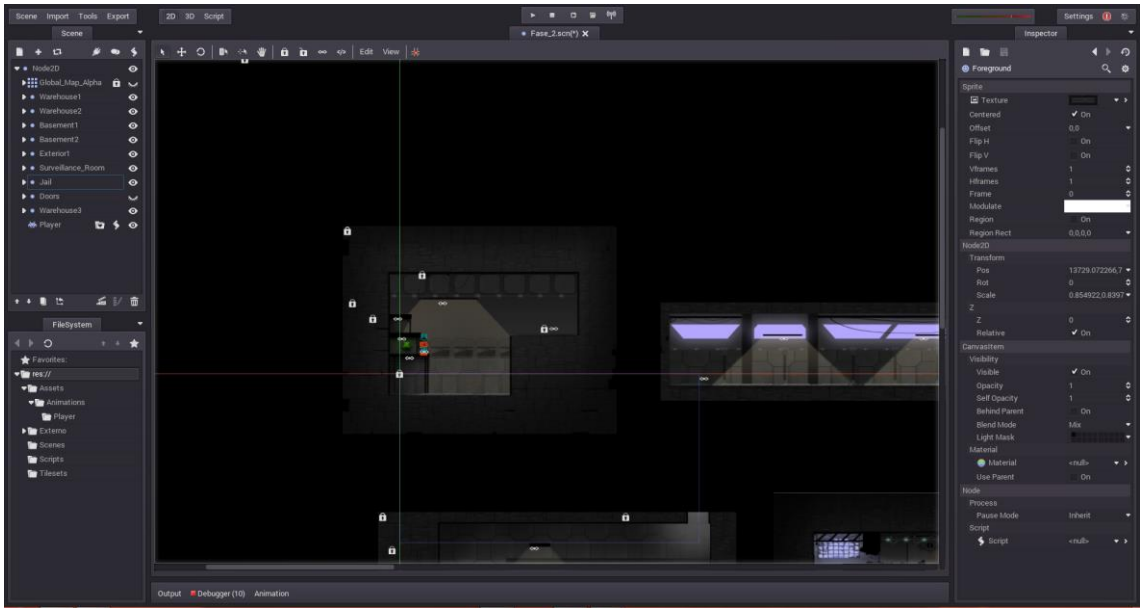


Ilustración 16: Captura del editor gráfico del motor Godot

No obstante, a pesar de sus bondades, *Godot* no está exento de algunas carencias derivadas de su reciente aparición. Actualmente el motor cuenta con un gran número de APIs para gestionar la física, colisiones, carga de imágenes y otros aspectos necesarios para la creación del videojuego. Sin embargo, todavía necesita un API que permita gestionar de manera eficiente la Inteligencia Artificial dentro del motor.

Además de esto, la documentación existente todavía resulta algo pobre. Además de la referencia del lenguaje propio de scripting, *GDScript*, es cierto que existe una serie de tutoriales y ejemplos básicos. Sin embargo, estos ejemplos son demasiado sencillos y no logran aclarar algunos aspectos clave. Por ejemplo: la detección de colisiones avanzada, interconexión y acceso a comportamientos y funciones entre diversos objetos, alternativas para crear el movimiento de un personaje, etc.

Esto supone una mayor dificultad a la hora de comenzar a trabajar con el motor y explorar sus posibilidades. La falta de documentación oficial y un mayor número de ejemplos desarrollados en detalle hace necesario tener que buscar información adicional y consultar otros tutoriales. Esto tampoco resulta sencillo, ya que la comunidad en torno al motor todavía es joven y su uso no está muy extendido.

Para obtener un mayor conocimiento de las capacidades y el funcionamiento del motor puede consultarse el anexo 3, en el que se detallan estos aspectos y se proporciona además una guía práctica con ayuda y consejos para empezar a trabajar con el motor.

2.4. Propuesta

Este proyecto busca ampliar la capacidad del motor para gestionar la IA del juego. Para ello se utilizará la API UGK_AI, desarrollada para el motor UPV Game Kernel. En concreto, se empleará una adaptación a C++ realizada por el compañero Xavier Mahiques Sifres[11].

Esta API permite la creación de máquinas de estados[12] y contiene además todas las herramientas necesarias para el paso de mensajes entre las distintas entidades del juego y su gestor de IA. Al disponer de la adaptación a C++, es posible integrar el código de la API como un módulo adicional del motor de juego. Como resultado de este proceso, la funcionalidad quedará completamente integrada en el motor y será posible utilizar los métodos definidos por esta herramienta desde el propio lenguaje de scripting empleado por el motor.

Resulta una buena herramienta, aparte de por los motivos ya mencionados, por emplear el modelo de máquinas de estado que ha sido estudiado en diversas asignaturas del Grado tales como Teoría de Autómatas y Lenguajes o Computabilidad y Complejidad.

2.5. UPV Game Kernel

El UGK es un proyecto que surgió para responder a la necesidad de implementar las prácticas de las asignaturas dedicadas al ámbito de los videojuegos dentro de la ETSINF, tales como Introducción a la Programación de Videojuegos (IPV) y Motores de Videojuegos (MOV). Para ello era necesario disponer de un videojuego sobre el que hacer pruebas, así como de un motor para ejecutarlo.

UGK es un API genérica que cubre los aspectos necesarios de la creación de un videojuego: soporte de ventanas, interfaz, visualización de los gráficos, gestión de entrada, inteligencia artificial, etc.

[11] El trabajo de este compañero aparece referenciado en el anexo de bibliografía

[12] Las máquinas de estados son un paradigma de IA en videojuegos muy extendido. Este artículo incluye una explicación teórica del concepto acompañada de un ejemplo de implementación: http://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867

Por ello supone una herramienta ideal para desarrollar trabajos de fin de grado o tesinas de fin de máster, dada su naturaleza abierta y la posibilidad de colaborar en ampliar la funcionalidad del mismo colaborando en tareas de investigación y docentes, tanto en las titulaciones relacionadas con la Ingeniería Informática como en otras.

3. Análisis del problema

Una vez visto el estado del arte, es necesario realizar una serie de preguntas o análisis para profundizar en distintos aspectos de la propuesta. En este apartado se va a realizar una comparativa de diversas herramientas para justificar la elección de Godot como la herramienta de desarrollo principal. También se va a realizar un análisis DAFO de cara a evaluar la viabilidad y los riesgos del proyecto. Se incluye además una planificación temporal del mismo, así como un análisis de la internacionalización del proyecto y un apartado donde se expone la colaboración realizada junto a otros compañeros y miembros del equipo.

3.1. Análisis DAFO

Es importante que los proyectos tengan una clara perspectiva tanto de las capacidades del equipo de desarrollo como de los factores externos que puedan afectar al éxito del proyecto. Una de los procedimientos que suelen emplearse para valorar estos puntos es el análisis DAFO, cuyas siglas se corresponden a: Debilidades, Amenazas, Fortalezas y Oportunidades. Las fuerzas y debilidades hacen referencia a aspectos internos del equipo de desarrollo y las amenazas y oportunidades son los factores externos a tener en cuenta. A continuación, se muestra el análisis DAFO de este proyecto:

Debilidades	**Amenazas**
• Equipo pequeño • Falta de experiencia en el desarrollo de videojuegos • Presupuesto prácticamente nulo • Compatibilizar el tiempo de desarrollo con las tareas de clase	• Plazo de entrega limitado sin posibilidad de prórroga • Mercado con competencia muy fuerte
Fortalezas	**Oportunidades**
• Un miembro del equipo tiene experiencia en programación e IA • Dos miembros del equipo tienen experiencia produciendo material gráfico • Un miembro del equipo tiene experiencia para creando música y efectos de sonido • El equipo conoce el medio	• Aprendizaje de una nueva tecnología de desarrollo • Oportunidad de colaborar en un equipo de desarrollo multidisciplinar • Posibilidad de ampliar el proyecto para emplearlo con fines comerciales

Ilustración 17: Matriz DAFO del proyecto

Uno de los objetivos de realizar este análisis es poder obtener estrategias para potenciar los aspectos positivos y neutralizar las debilidades y amenazas. Así pues, se muestran a continuación dichas estrategias:

Estrategia ofensiva	Estrategia adaptativa
• Desarrollar un producto original, derivado de esquemas jugables clásicos • Aprovechar la experiencia del equipo para crear un juego con un estilo artístico personal y diferenciador • Crear un sistema escalable que permita la reutilización del proyecto en ampliaciones o futuros proyectos similares.	• La realización del proyecto implica adquirir experiencia con una tecnología interesante para el equipo de desarrollo. • Colaborar en un equipo multidisciplinar ofrece una experiencia más cercana al desarrollo real de videojuegos en un estudio profesional. • El equipo formado puede seguir trabajando en el futuro para constituir un estudio independiente y adquirir mayor experiencia.
Estrategia defensiva	**Estrategia de supervivencia**
• Contar con diferentes miembros especializados permite repartir tareas y agilizar el desarrollo. • La creación de material gráfico y sonoro dedicados al proyecto lo dotarán de un mayor atractivo de cara al posible público comercial.	• Se desarrollará una demostración sencilla del proyecto. • Se tomarán como referencia juegos documentados y conocidos para establecer las bases jugables. • Se marcará una serie de hitos para completar un prototipo sencillo y sin errores.

Ilustración 18: Tabla de estrategias obtenidas a partir de la matriz DAFO

3.2. Planificación temporal

Para el desarrollo del proyecto se ha establecido una planificación por iteraciones que tienen como objetivo alcanzar determinados hitos. Esto permite dividir el desarrollo del trabajo en diversas etapas enfocadas en objetivos diversos que acercan el proyecto al resultado final.

Es una metodología adecuada dado que la creación de un videojuego conlleva un amplio abanico de aspectos a implementar. Por tanto, sin una planificación previa, la

falta de objetivos concretos puede llegar a abrumar al equipo de desarrollo, provocando una falta de motivación.

Cabe destacar que esta planificación no incluye el tiempo adicional empleado para realizar la primera versión del GDD y reunir a los cuatro miembros que forman el equipo de desarrollo. Estas tareas, incluyendo reuniones para establecer el reparto de funciones dentro del equipo y demás procedimientos previos a la fase de implementación, fueron realizadas en los meses anteriores al inicio del desarrollo del proyecto. Cada iteración supone también llevar a cabo un conjunto de pruebas para comprobar el funcionamiento de los sistemas implementados.

Se presentará a continuación la serie de iteraciones y las tareas asociadas a las mismas. Además, se incluye al final de este punto un diagrama de Gantt para ilustrar de forma gráfica la planificación

1ª iteración – 3 semanas

Alcanzar un primer prototipo jugable que cuente con el movimiento del personaje principal, además de las siguientes características:

- **Generación de colisiones y movimiento**. Empleando el motor de físicas de Godot, crear una primera aproximación que permita el movimiento del personaje.
- **Script de gestión del movimiento**. Necesario para recoger la entrada por teclado.
- **Personalización del esquema de control del juego**. Asignando acciones a las teclas siguiendo las especificaciones del GDD.
- **Arte provisional para el protagonista y los escenarios**. Sin realizar animaciones y con un conjunto reducido de objetos necesarios para probar el movimiento.
- **Movimiento de la cámara**, que debe seguir al jugador por los escenarios.
- **Creación del escenario empleando *tilemaps*[13]**.

2ª iteración – 2 semanas

Actualizar los controles del juego y habilitar algunos de los elementos interactivos de los escenarios. Las tareas a completar son:

- **Adaptar el comportamiento de las colisiones con plataformas.**

[13] Consultar el glosario para ver una definición del término *tilemap*

- **Añadir una función de transporte al jugador**, con el objetivo de que pueda cambiar de habitación empleando puertas.
- **Script para la gestión de puertas**, realizando las conexiones entre las distintas salas que componen el nivel del prototipo.
- **Añadir animaciones provisionales al personaje principal**, acercándolo al aspecto definitivo.
- **Modificar el script de control del personaje**, añadiendo las estructuras necesarias para la gestión de las animaciones.
- **Refinar el control del personaje**, ajustando la altura y velocidad del salto y añadiendo la posibilidad de agacharse.
- **Crear un sistema de colisiones dinámicas para el personaje principal**, en conjunto con la posibilidad de agacharse.

3ª iteración – 3 semanas

Incluir la funcionalidad de la API UGK_AI e implementar el comportamiento de los enemigos, estas son las tareas:

- **Recompilar el motor Godot**, para que incluya la funcionalidad de la API.
- **Implementar las máquinas de estados**, para gestionar la IA de los enemigos.
- **Adaptación del script de movimiento**, para que el movimiento de los enemigos esté condicionado por el estado de la IA.
- **Creación de un *Manager* de IA**, encargado de la gestión de los estados de cada enemigo.
- **Añadir las animaciones de los enemigos**, y el correspondiente control atendiendo a la máquina de estados.
- **Finalizar el scripting necesario para el resto de elementos interactivos**, basándose en la implementación de las puertas, programar otros elementos con los que el personaje puede interactuar.

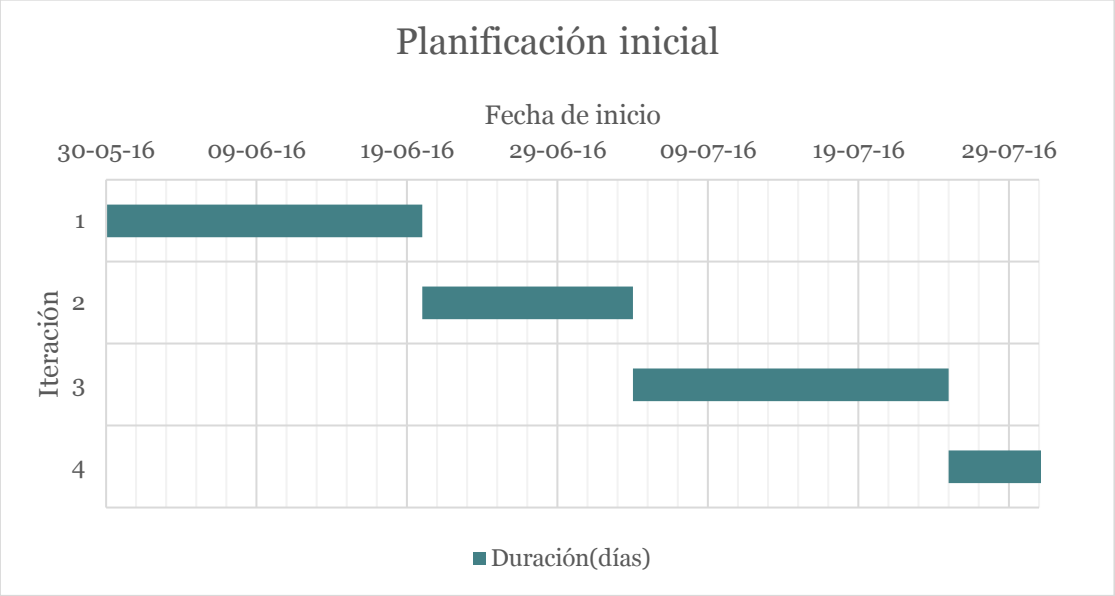
4ª iteración – 1 semana

Añadir los menús e interfaz junto con la siguiente lista de detalles para pulir el juego:

- **Añadir los menús del juego e implementar su funcionalidad**.
- **Añadir elementos visuales**, tales como la barra de vida del protagonista, información contextual de acciones disponibles, etc.
- **Añadir los elementos sonoros**, que incluyen la música de los niveles y los efectos de sonido.
- **Sustituir los gráficos de prueba por los definitivos de los escenarios**

Tras esta última iteración, se llevará a cabo un conjunto definitivo de pruebas para comprobar el correcto funcionamiento del sistema y analizar los resultados. Durante este periodo de pruebas se mostrará el trabajo realizado a jugadores de prueba fuera del equipo de desarrollo, a fin de recoger *feedback* y poder realizar ajustes finales de cara a la presentación.

A continuación, se muestra el diagrama de Gantt correspondiente a la planificación inicial del proyecto:



Iteración	Fecha de inicio	Duración (días)	Fecha de fin
1	30/05/2016	21	20/06/2016
2	20/06/2016	14	04/07/2016
3	04/07/2016	21	25/07/2016
4	25/07/2016	7	01/08/2016

Ilustración 19: Diagrama de Gantt Inicial

3.3. Análisis de la internacionalización

La existencia de las plataformas digitales de distribución de videojuegos hace posible que los videojuegos se publiquen en todo el mundo de forma simultánea. A raíz de esto aumenta la importancia de ofrecer el videojuego en el mayor número de idiomas posible. Esto puede suponer un problema para estudios pequeños, ya que la traducción de juegos con una gran carga de texto resulta complicada y cara, puesto que suele requerir de los servicios de profesionales externos al equipo de desarrollo.

Este proyecto sin embargo es un juego muy pequeño, por lo que la cantidad de texto a traducir resulta muy asequible. De cara a una posible continuación del proyecto como juego comercial, se ha decidido adaptar la demostración a los idiomas español e inglés. Esto permitirá que el juego pueda llegar a muchos usuarios alrededor de todo el mundo.

Para ello, Godot cuenta con herramientas capaces de gestionar traducciones a distintos idiomas a partir de textos almacenados en archivos de tipo csv. Estos archivos pueden ser generados por programas de edición de hojas de cálculo. También cabría en este punto considerar la utilización del módulo de internacionalización del UPV Game Kernel, para comparar el funcionamiento de ambos.

Como nota final, cabe destacar que el código desarrollado estará comentado en inglés, a fin de que pueda servir como ejemplo no solo en el ámbito de la Universidad, sino también a nivel global. Si se dispone de suficiente tiempo, se tratará de desarrollar la documentación en ambos idiomas.

3.4. Colaboración

Es importante comentar con más detalle quiénes son los compañeros con los que se ha realizado el desarrollo, así como el autor de la adaptación de la API UGK_AI empleada.

Empezando por el equipo de desarrollo, Mauro Beltrán[14] es el autor de la música del juego y los efectos de sonido. Es un músico independiente interesado en el mundo del desarrollo de videojuegos que forma parte del equipo. Carlos Mercé Vila[15], estudiante de Bellas Artes en la UPV, es el autor de los gráficos y animaciones del personaje protagonista y los enemigos del juego. Por último, Germán Reina Carmona[16], también estudiante de Bellas Artes de la UPV, es la persona responsable de los gráficos de los escenarios.

Este proyecto trata de ofrecer una aplicación práctica que sigue el trabajo del compañero Xavier Mahiques Sifres, alumno de la ETSINF que presentó su TFG el

[14] Página de Mauro Beltrán en Bandcamp, donde puede escucharse uno de sus trabajos: https://maurobeltran.bandcamp.com/
[15] Blog de Carlos Mercé, donde pueden verse algunos de sus trabajos: https://merccarlos.wordpress.com/
[16] Página de Germán Reina en Artstation, donde pueden verse algunos de sus trabajos: https://www.artstation.com/artist/germanreina

pasado mes de julio. Fue la persona encargada de adaptar el funcionamiento del módulo UGK_AI al lenguaje C++.

4. Diseño

Si bien el diseño es importante en cualquier proyecto de software para establecer unas bases sólidas sobre las que hacer el desarrollo, en un videojuego resulta especialmente determinante. De un mal diseño de conceptos y mecánicas jugables no puede obtenerse nunca un buen videojuego, por muchos medios o talentos de los que disponga el equipo de desarrollo. Además, sin un sistema bien diseñado resulta muy complicado hacer que un proyecto logre finalizar con éxito, dado que la ausencia del diseño implica no tener una visión global con la que poder desarrollar el proyecto. Es preferible pues invertir todo el tiempo necesario en esta etapa para evitar la aparición de problemas debidos a la toma de decisiones sobre la marcha.

Dentro del mundo del videojuego tiene especial importancia el GDD. Este documento desarrolla todos los aspectos del videojuego. En él se encuentran las directrices a seguir para realizar los escenarios, música, apartado gráfico, mecánicas jugables, interfaz, etc. Es por tanto el documento clave que todos los miembros del equipo de desarrollo han de tomar como referencia para realizar su correspondiente trabajo.

Fijando el punto de vista en la tarea del programador, es posible tomar este documento de diseño como la base sobre la que diseñar el software. Para ello se parte de los requisitos que se especifican en dicho documento. Combinar el contenido del GDD con una estructuración apropiada del software permite en definitiva obtener un producto sólido y con calidad.

En esta sección se incluye una pequeña introducción al motor Godot, para dar a conocer algunos términos clave y su metodología en términos generales. Tras esta introducción se mostrará una descripción del concepto general del juego. En base a este concepto, se dará paso al diseño estructural de los componentes del juego ilustrados con diagramas de clase. En un apartado dedicado a la interfaz se mostrarán también los casos de uso de la misma. El último apartado incluye un diagrama que muestra todas las escenas del juego, para poder observar la estructura global del proyecto.

4.1. Introducción al motor de juegos Godot

Godot es un motor desarrollado desde cero con la intención de facilitar la creación de videojuegos en 2D. Esto quiere decir que las librerías de física, colisiones, etc., han sido diseñadas e implementadas por el equipo de desarrollo del motor. Esto no implica

únicamente la búsqueda de una mayor eficiencia, sino también plantear una nueva metodología para desarrollar videojuegos.

Los videojuegos cuentan con un grafo de escena que contiene los elementos de un determinado nivel del juego, incluyendo su posición y estado de variables entre otros datos. Godot no es una excepción, y basa el esquema organizativo del proyecto en dos conceptos clave: nodos y escenas.

Los *nodos* son cada uno de los componentes que conforman el mundo del videojuego. Hay distintos tipos de nodos como, por ejemplo:

- **Node2D**, es el tipo de nodo básico del cual heredan sus atributos el resto de nodos.
- **Sprite**, que almacena la textura asociada a un objeto
- **KinematicBody2D**, empleado para crear el cuerpo de personajes 2D que actúan de forma independiente al motor de físicas.
- **CollisionShape2D**, que contiene la forma “sólida” de los objetos en 2D
- **Area2D**, sirve para definir una zona y detectar eventos como la entrada o salida de un objeto respecto a la zona, alterar las propiedades físicas dentro de un área, etc.
- **Tilemap**, permite crear un nivel de juego empleando elementos que se colocan en una cuadrícula.

Estos son solo algunos de los tipos de nodos disponibles. Haciendo una analogía a los lenguajes orientados a objetos, cada nodo puede verse como una clase. De hecho, cada tipo de nodo cuenta con una página dentro de la especificación del lenguaje de scripting GDScript, donde se pueden consultar sus funciones y variables. Es posible además crear nuevos tipos de nodo mediante extensiones de forma que quedan completamente integrados en el motor. Este será el procedimiento empleado para integrar la funcionalidad del módulo UGK_AI.

Estos nodos se organizan a su vez en *escenas*, formando lo que se conoce en Godot como un *árbol de escena*. Para ilustrar este concepto, se muestra a continuación el árbol de escena que corresponde al personaje principal:

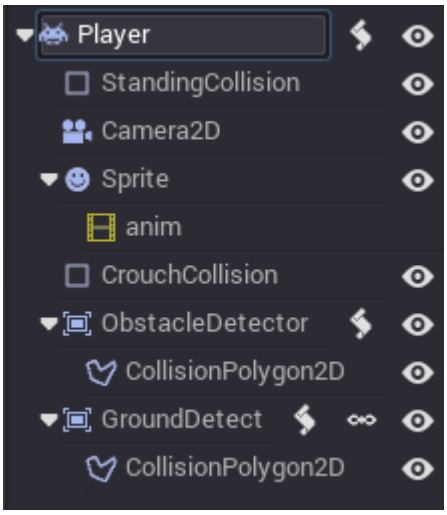


Ilustración 20: Árbol de escena del personaje principal

El árbol de escena del personaje principal tiene como nodo raíz un nodo de tipo *KinematicBody2D*, el cuál ve sus funcionalidades ampliadas por sus nodos "hijos". Los nodos de tipo *CollisionShape2D*, *StandingCollision* y *CrouchCollision*, sirven para dar forma al cuerpo cuando el personaje está de pie o agachado. *Camera2D*, cuando se asocia a un nodo y se define como la cámara principal, permite que la cámara siga en todo momento a dicho nodo. *Sprite* almacena las texturas del personaje, contenidas en una única hoja de texturas. De este nodo cuelga a su vez un nodo *AnimationPlayer*, para gestionar y reproducir distintas animaciones. Por último, se puede ver la existencia de dos nodos de tipo *Area2D: ObstacleDetector* y *GroundDetect*. Estas áreas auxiliares se emplean para detectar obstáculos sobre la cabeza del personaje cuando este se agacha y detectar cuando está apoyado sobre el suelo respectivamente. De cada nodo cuelga un nodo *CollisionPolygon2D* que delimita el área de detección.

Además de los nombres de los nodos se puede ver una serie de iconos que aportan información adicional. Cada nodo tiene asociado un icono característico que ilustra su tipo en caso de que se cambie el nombre por defecto. El icono del pergamino indica que hay un *script* asociado a dicho nodo. El icono del ojo abierto muestra los objetos visibles en el editor, se puede cambiar la visibilidad pulsándolo. El icono que parece un lazo indica que al hacer *click* sobre el editor de la escena, solo se puede seleccionar el nodo padre, y los hijos van ligados a este manteniendo así su posición relativa. Por último, al pulsar en las flechas se puede plegar o desplegar la lista de hijos de un nodo.

La escena que se ha mostrado, ilustrando la estructura de nodos que dan forma al personaje principal, puede integrarse dentro de otras escenas. Esto amplía

enormemente el abanico de posibilidades a la hora de organizar y desarrollar el proyecto. En un equipo de trabajo grande, sería posible dividir el trabajo en varios grupos y que estos desarrollasen escenas independientes por separado.

Supongamos por ejemplo que tenemos tres grupos. El primero de ellos se dedica a implementar el personaje principal. El segundo se dedica a trabajar en un enemigo. Por último, el tercer grupo se encarga de crear un escenario. Una vez acabadas las escenas independientes, existe la posibilidad de integrarlas todas en una escena global. Esto permite que cada grupo avance en su trabajo sin riesgo de modificar el desarrollo de los otros grupos. También incrementa la escalabilidad del proyecto, puesto que siempre sería posible desarrollar nuevos elementos e integrarlos dentro de las escenas existentes.

Ya se ha establecido cuál es la filosofía de diseño, pero todavía queda por revisar cómo funcionan los juegos desarrollados con Godot. Al igual que en la mayoría de motores, Godot cuenta con un *bucle de juego*. Este es básicamente un bucle infinito que se recoge el estado del juego en un momento concreto y lo actualiza atendiendo a la entrada recibida, la simulación de físicas, etc.

La gestión de este bucle la realiza el propio motor internamente, pero es posible definir el comportamiento de los nodos haciendo uso de *scripts*. Existen una serie de funciones que pueden reescribirse, por ejemplo:

- **_ready()**, la función que se ejecuta cuando el nodo asociado al script y todos sus nodos internos se han cargado. Suele emplearse para la inicialización de variables que dependen de nodos internos.
- **_fixed_process(delta)**, en esta función se puede especificar el comportamiento del nodo. Se ejecuta de acuerdo al *deltaTime*[17], definido por la variable *delta*.
- **_input(event)**, función destinada a la gestión de la entrada.

Algunas de estas funciones, como *fixed_process* o *input* pueden activarse o desactivarse mediante scripting. También existe la posibilidad de crear funciones asociadas a eventos y otras funciones propias.

Los tipos de nodos introducidos en este apartado se emplearán para realizar los diagramas de clase de los apartados siguientes. Si se quiere profundizar en las características y el funcionamiento del motor, se recomienda la lectura del anexo 3.

[17] Tiempo en segundos que tarda en ejecutarse un *frame* del juego.

4.2. Concepto general

Steel Soldier es un juego en dos dimensiones en el que se combinan elementos de acción y sigilo. Se pone especial énfasis en el movimiento de salto para explorar los escenarios y sortear obstáculos y enemigos. El juego está dividido en varias fases en las que se propone al jugador alcanzar diversos objetivos a fin de crear situaciones variadas.

El jugador encarna el papel de un soldado anónimo que se infiltra en la base de una organización militar para rescatar a un rehén. Basándose en este contexto, el juego distingue dos tipos de fases distintas: de acción e infiltración. En las fases de acción el jugador debe atacar a los enemigos que encuentre a su paso y tratar de hacer que estos no vacíen su barra de vida. En las fases de infiltración el objetivo es aprovechar la estructura del entorno para pasar desapercibido ante los enemigos. Si estos detectan al jugador disparan un estado de alarma y tratan de acabar con él.

La mezcla de situaciones pretende dar pie a una situación de tensión que permita al jugador sentirse dentro de una aventura de espías, aumentando la inmersión en el mundo del juego y haciéndolo más atractivo y disfrutable.

4.3. Diseño de los componentes principales

En este apartado se especifica la estructura de nodos que conforma cada uno de los elementos principales del juego, los cuales son: personaje principal, enemigos, elementos interactivos, plataformas y mapa del escenario. Siguiendo los principios de diseño modular explicados en el apartado 4.1, cada uno de estos elementos constituirá una escena aislada. Si se desea conocer más detalles acerca del diseño se recomienda la lectura del apéndice 4, que contiene el GDD del juego.

4.3.1. Personaje principal

Este es el diagrama de clases que conforma la estructura del personaje principal:

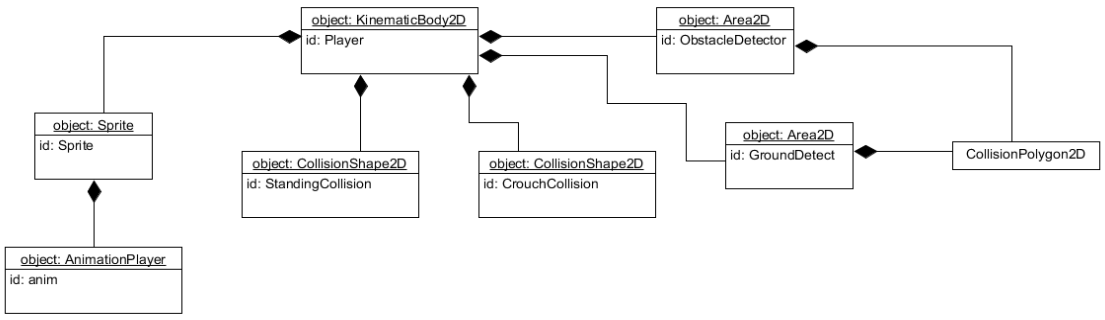


Ilustración 21: Diagrama de clases del personaje principal

A continuación, se incluye una breve explicación de la función de cada nodo dentro del conjunto:

- **KinematicBody2D – Player,** es el componente principal y el "cuerpo" del protagonista. Este tipo de cuerpo ignora las fuerzas del motor de físicas, lo cual reduce el coste de procesamiento empleado por los cálculos de movimiento del personaje.
- **Sprite** almacena la hoja de texturas del personaje.
- **AnimationPlayer – anim**, es empleado para crear las distintas animaciones
- **CollisionShape2D – StandingCollision**, almacena la caja de colisión activa cuando el personaje está de pie.
- **CollisionShape2D – CrouchCollision**, almacena la caja de colisión activa cuando el personaje está agachado.
- **Area2D – ObstacleDetector**, se activa cuando el personaje está agachado, detecta si hay algún objeto que impida que el personaje se levante.
- **Area2D – GroundDetector**, detecta la colisión con el suelo.

Se incluyen también funciones adicionales implementadas mediante *scripts*. Se detallará su funcionamiento en el apartado de implementación.

4.3.2. Enemigos

El prototipo a desarrollar cuenta con tres tipos de enemigos distintos. Aunque su aspecto y comportamiento varían, la estructura de nodos desplegada es similar. Este es el diagrama de clases correspondiente a los enemigos:

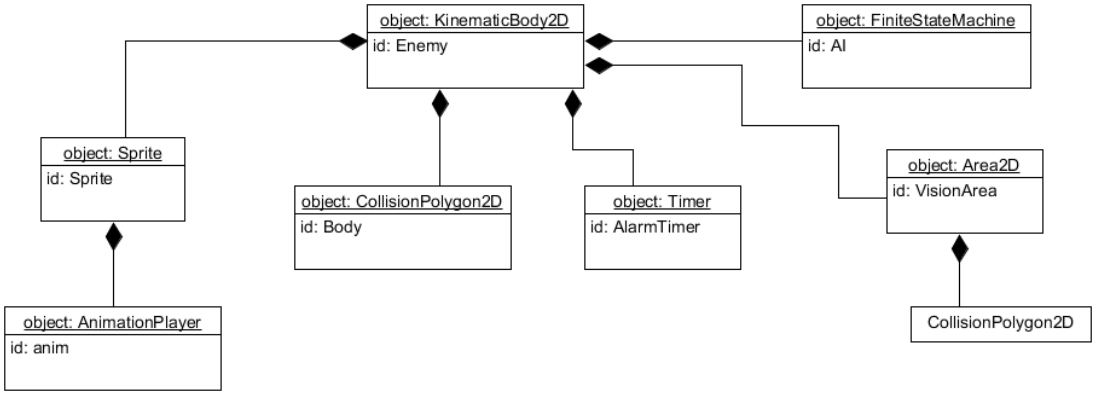


Ilustración 22: Diagrama de clases de los enemigos

Este es el cometido de cada tipo de nodo:

- **KinematicBody2D – Enemy**, componente principal. Su movimiento está controlado por la IA del juego.
- **Sprite** almacena la hoja de texturas.
- **AnimationPlayer – EnemyAnim**, empleado para crear y almacenar las distintas animaciones.
- **CollisionPolygon2D – Body**, es la forma sólida de cada enemigo, empleada para detectar colisiones con el mapa y otros objetos.
- **Area2D – VisionArea**, área empleada para detectar al personaje principal.
- **Timer – AlarmTimer**, nodo que habilita un cronómetro con el que medir el tiempo de espera para finalizar el estado de alerta
- **FiniteStateMachine – AI**, nodo que almacena los estados de la IA de cada enemigo y se comunica con el gestor de IA del juego.

Se ampliará la información de este apartado en la sección de implementación, incluyendo los detalles característicos de cada enemigo. Se puede consultar una descripción de su comportamiento en el GDD del anexo 4.

4.3.3. Elementos interactivos

Estos son los elementos del escenario con los que el personaje principal puede interactuar. Dentro de este grupo encontramos objetos como puertas, escondites y terminales. Aunque su comportamiento interno varía, comparten la estructura de nodos, que se puede ver en el siguiente diagrama de clases:

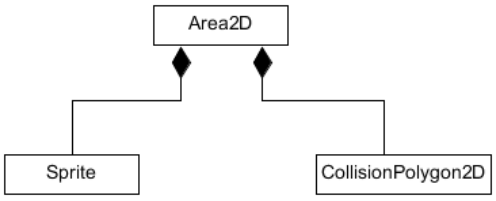


Ilustración 23: Diagrama de clases de los elementos interactivos

Esta es la función que cumple cada nodo dentro de la estructura:

- **Area2D** es el nodo principal de estos elementos. Se necesita su uso ya que se pretende detectar la presencia del jugador, pero estos elementos no son obstáculos, sino que forman parte del fondo.
- **CollisionPolygon2D** define la forma del objeto. Complementa al nodo *Area2D* ya que este no tiene capacidad de detectar objetos por sí mismo
- **Sprite** muestra el aspecto de estos elementos.

Las particularidades del comportamiento de cada elemento se especifican dentro de los apartados 5.6., 5.14. y 5.15.

4.3.4. Plataformas

Las plataformas constituyen un elemento aislado debido a las limitaciones del nodo tipo *Tilemap*. Este nodo permite definir un comportamiento de colisión muy básico para cada elemento que compone el mapa del juego. Sin embargo, para crear una detección de colisiones compleja es necesario que los elementos sean independientes. Esta es la estructura de las plataformas:

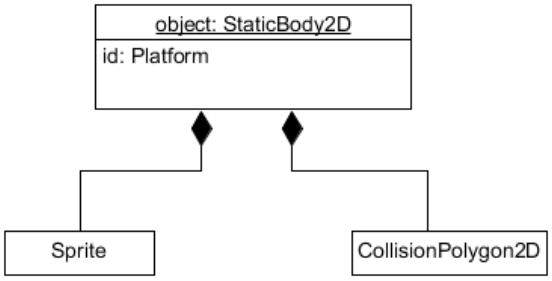


Ilustración 24: Diagrama de clases de las plataformas

- **StaticBody2D – Platform**, es un cuerpo físico estático, que no puede ser movido por fuerzas externas.
- **CollisionShape2D** contiene la forma sólida del objeto.
- **Sprite** almacena la textura de cada plataforma.

En el anexo 3 se hace una descripción más profunda del problema de los *tilemaps* y las colisiones avanzadas.

4.3.5. Mapa del escenario

Las distintas habitaciones que componen el mundo del juego están implementadas haciendo uso de *Tilemaps*. Un *tilemap* es un conjunto de texturas, normalmente de forma cuadrada o hexagonal, que son posicionadas en una cuadrícula para dar forma a los escenarios de algunos juegos en 2D.

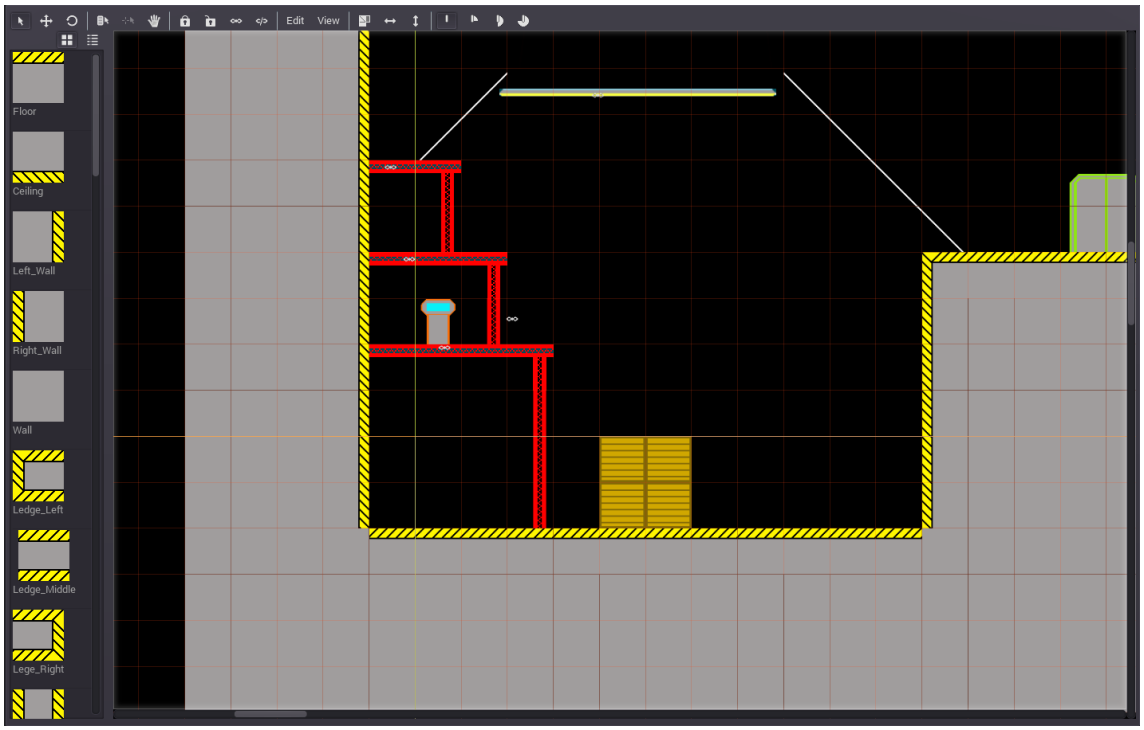


Ilustración 25: Pantalla de edición de tilemaps en Godot

Godot cuenta con la posibilidad de crear *tilemaps* propios y emplearlos para crear los niveles del juego. Esto evita tener que colocar decenas de nodos idénticos duplicados, lo cual ayuda a la organización del proyecto y a realizar pequeños cambios puntuales en el escenario de forma rápida y sencilla. Es posible emplear varios *tilemaps* en una misma escena. Así se obtienen distintas capas de profundidad visual, pueden organizarse mejor los distintos elementos, etc.

En el punto 5.1. del apartado de implementación se detalla el proceso de creación de *tilemaps* en Godot como parte del proceso de implementación.

4.4. Diseño de la interfaz

De cara a la versión definitiva del juego, el GDD incluye una serie de menús y prestaciones para ofrecer un producto más pulido. Este apartado recoge el diseño de dichos menús mediante diagramas de casos de uso, así como capturas de su aspecto definitivo dentro del videojuego. Cada uno de los menús que se presentan en los siguientes puntos quedará implementado como una escena independiente.

Dadas las características del prototipo, no se ha considerado necesario implementar algunos de los menús presentados en el anexo 4. El prototipo solo cuenta con un nivel, por lo que resulta innecesario dar opción a seleccionar niveles, guardar o cargar la partida. Aunque el proyecto está preparado para ser ampliado con estas funcionalidades, se ha optado por implementar el menú de inicio, opciones y pausa.

4.4.1. Menú inicial

Es el menú que se abre al iniciar el juego. Da acceso al resto de menús, así como al inicio de una nueva partida. Este es el diagrama de casos de uso correspondiente a este menú:

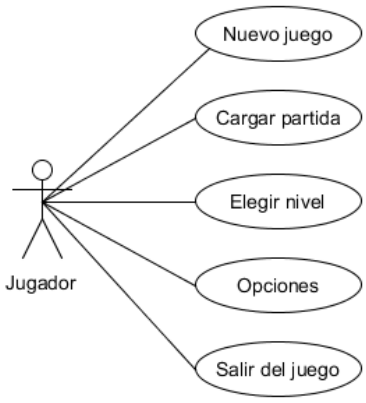


Ilustración 26: Diagrama de casos de uso del menú inicial

Los casos de uso implementados son: nueva partida, acceder al menú de opciones y salir del juego. Los menús de cargar partida y elegir nivel no se han implementado en el prototipo pues este cuenta con una única fase.

4.4.2. Menú de opciones

Este menú permite al usuario modificar algunos ajustes del juego. Es accesible desde el menú inicial y también desde el menú de pausa. Este es su diagrama de casos de uso:

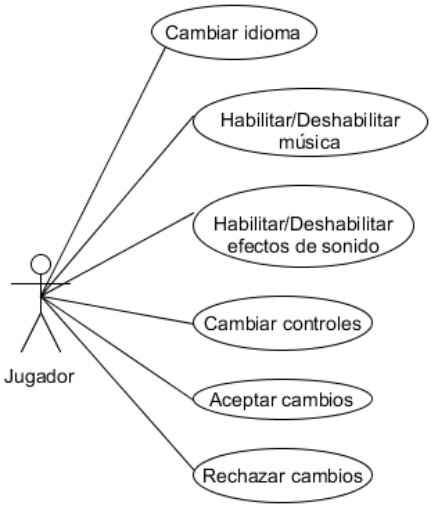


Ilustración 27: Diagrama de casos de uso del menú de opciones

En este caso, el menú es completamente operativo

4.4.3. Menú de pausa

Este menú permite al jugador tomar un descanso durante la partida. Sus otras funcionalidades están recogidas en el siguiente diagrama de casos de uso:

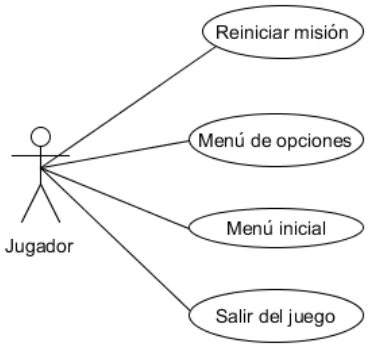


Ilustración 28: Diagrama de casos de uso del menú de pausa

Este menú también es completamente operativo en el prototipo.

4.5. Estructura global de las escenas

Tras haber repasado el diseño y los contenidos de cada escena por separado, se muestra a continuación un diagrama que recoge los distintos tipos de escena y su organización:

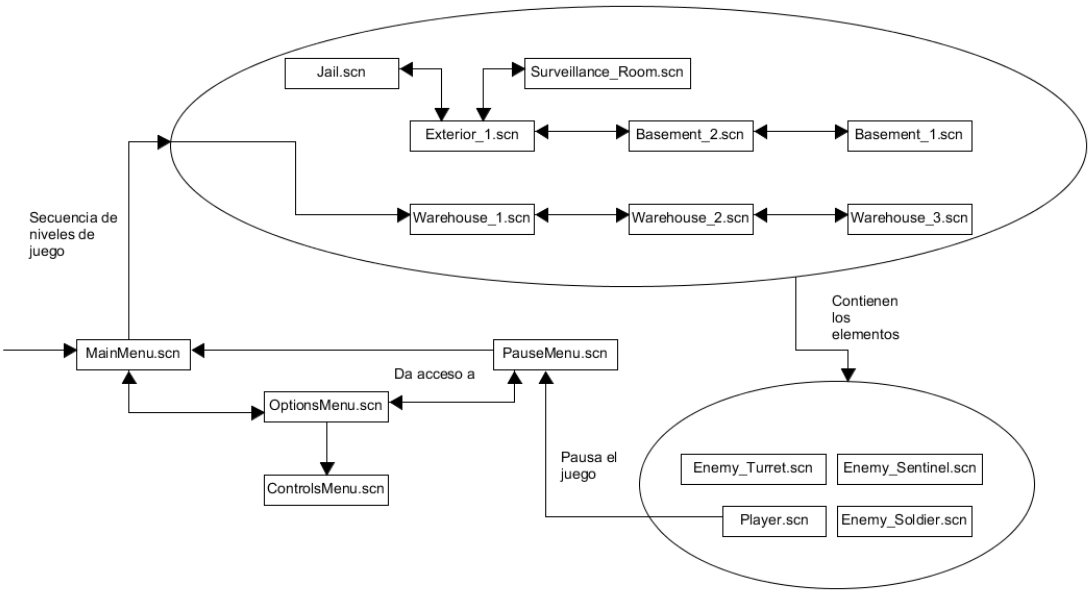


Ilustración 29: Diagrama global de escenas

Este diagrama puede ser empleado como base para futuras ampliaciones y da una visión general de la estructura del proyecto. En el anexo 4 se incluye además un diagrama de flujo del juego, así como menús adicionales y un esquema del *HUD*[18] del juego.

[18] *Heads-Up Display*, conjunto de elementos gráficos superpuestos que proporcionan información del estado de la aplicación

5. Implementación

Tras analizar el problema en profundidad y realizar el proceso de diseño, llega el momento de pasar a la fase de implementación del proyecto. Esta ha sido la fase más larga y costosa. Las primeras etapas del desarrollo resultaron especialmente complicadas debido a la falta de experiencia previa con Godot y su lenguaje de programación. Sin embargo, una vez superado el periodo de adaptación, el resto de la implementación se benefició de las posibilidades que brindan las herramientas seleccionadas.

El punto 5.9., que habla de la integración del módulo UGK_AI dentro de Godot, es la principal novedad que aporta este proyecto. El resto de los puntos pueden verse como una demostración práctica del motor Godot en este proyecto en particular.

Siguiendo el orden establecido por la planificación temporal del apartado 3.3, se muestran a continuación todos los pasos seguidos para lograr el resultado final.

5.1. Creación del mapa provisional

La primera tarea que se llevó a cabo durante esta etapa fue la creación de un escenario en el que poder probar el movimiento del protagonista. Para ello, se creó un conjunto de gráficos provisionales que permitieron dar forma a los niveles. Godot cuenta con la posibilidad de crear un *tileset*[19] a partir de una escena. Para ello es necesario disponer de todos los elementos del *tileset* en una cuadrícula, dentro de un nodo de tipo *Sprite*. Adicionalmente se puede asignar un polígono de colisión a los elementos sólidos como el suelo y las paredes. Este es el aspecto de la escena de edición del *tileset* al principio del desarrollo:

[19] Un *tileset* es el conjunto de elementos que conforman un *tilemap*. Definiciones en el glosario de términos, Anexo 2.

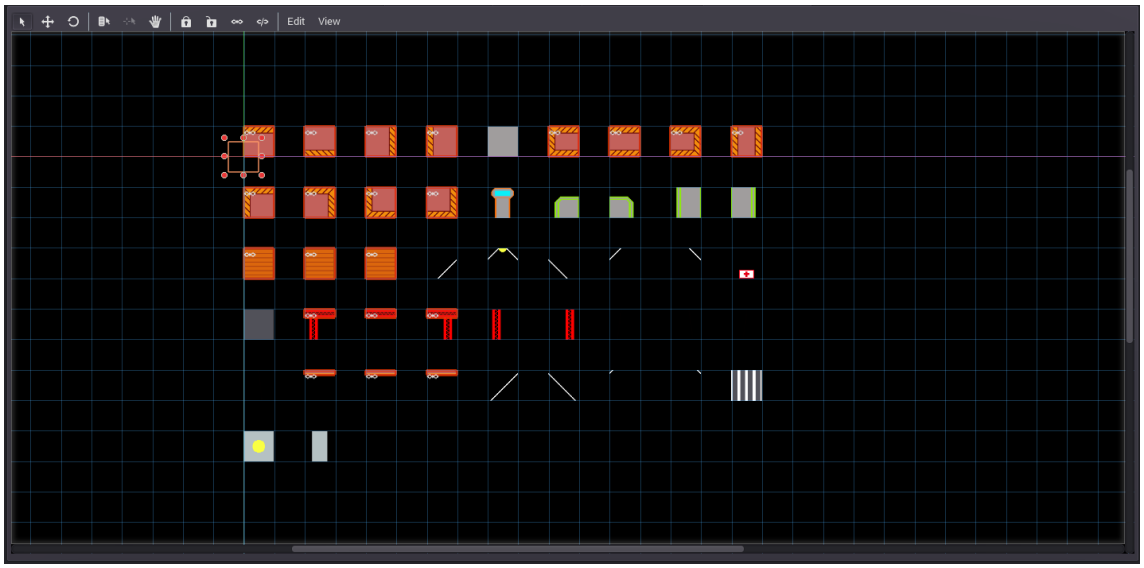


Ilustración 30: Escena de edición del tileset

Una vez creada esta escena, es posible exportarla a un recurso del proyecto que puede emplearse en otras escenas como base para la creación de los nodos de tipo *TileMap*. Tras crear el diseño definitivo del nivel este se trasladó a Godot haciendo uso de este procedimiento. La siguiente ilustración muestra el editor de mapas y parte del escenario diseñado:

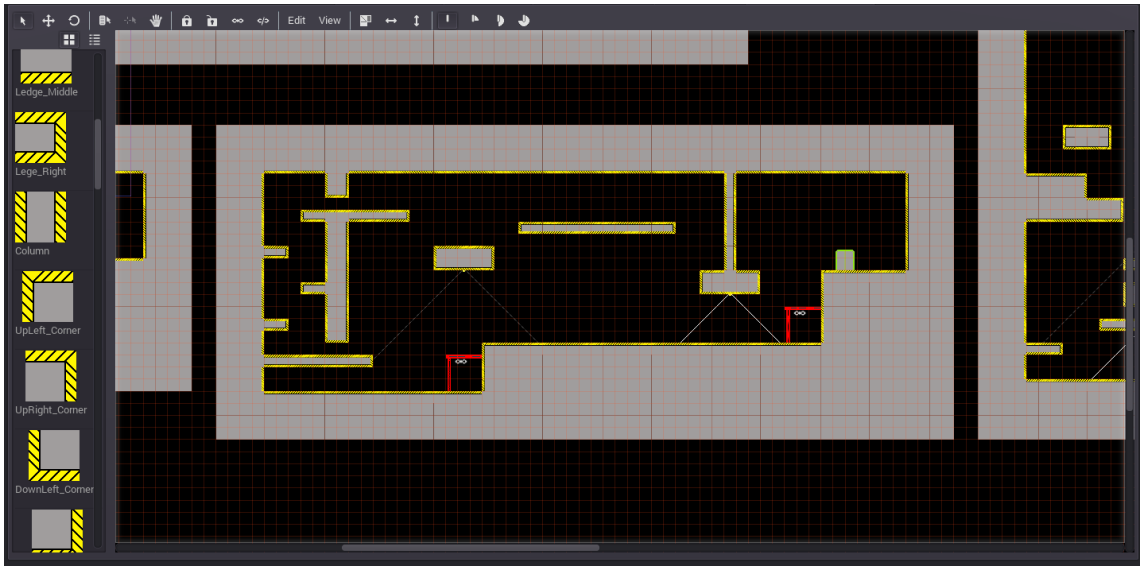


Ilustración 31: Escena de edición del tilemap durante la fase inicial del proyecto

Si se desea conocer el proceso en profundidad, se recomienda la lectura del anexo 3.

5.2. Creación del personaje principal y sus colisiones

Partiendo del esquema de diseño visto en el punto 4.3.1., la primera aproximación al personaje estaba orientada a responder a la entrada y colisionar con los elementos del

mapa. Para cumplir este propósito, tan solo era necesario incluir los nodos *KinematicBody2D* y *CollisionShape2D*, además de un *Sprite* para mostrar una imagen provisional.

Los cuerpos *kinemáticos*, como el del protagonista, ignoran algunos parámetros físicos como la fuerza de gravedad y la inercia. Esto hace que sea necesaria menos potencia de cálculo. Además, este tipo de cuerpos permite crear un control que responde mejor que los cuerpos controlados por físicas. Esto se debe a que este tipo de juegos requiere de precisión en el movimiento y los saltos, e introducir movimientos inerciales da la sensación de falta de control.

Por su parte, el polígono de colisión hace posible que el personaje no traspase los elementos sólidos del *tilemap*. Godot gestiona estas colisiones de forma interna, lo cual simplifica el proceso de trabajo.

Esta es la estructura de nodos y el aspecto inicial del personaje protagonista durante esta etapa:

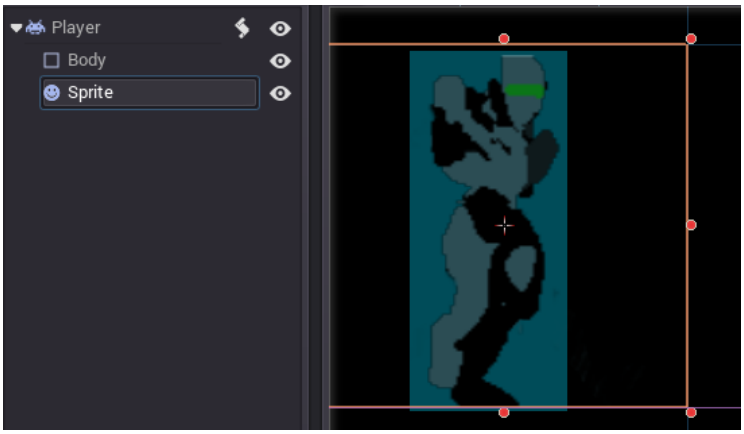


Ilustración 32: Estructura y aspecto del jugador en la primera iteración

5.3. Gestión de la entrada y movimiento del personaje

Una vez creada la estructura básica del personaje, es necesario dotarlo de movimiento para poder comprobar el correcto funcionamiento de las colisiones integradas en el *tilemap*. Para ello es necesario por una parte ajustar algunos parámetros dentro del motor y por otra parte crear un script básico asociado al nodo *KinematicBody2D* del personaje.

El primer punto se consigue abriendo la configuración del proyecto, dentro de la pestaña *Scene* del editor. Dentro de estas propiedades hay una sección que permite gestionar el mapeado de la entrada. Esta sección da la posibilidad de agregar

acciones a una lista básica predefinida, y asignar todas las teclas o botones de un *joystick* que lancen un evento de entrada con dicha acción. Las acciones añadidas fueron:

- ***Left*** **y** ***Right***, asignadas a las flechas de dirección correspondientes. Se emplean para caminar por el escenario.
- ***Jump***, asignada a la flecha de dirección superior. Como su nombre indica, es la tecla de salto.

Estas acciones básicas eran las necesarias para hacer una prueba de movimiento inicial. La lista fue ampliada durante la segunda iteración del proyecto.

Una vez creada la serie de acciones básicas, se continuó con la creación del *script* para controlar el movimiento del personaje. Ya que, como se ha explicado en el punto 5.2., los cuerpos *kinemáticos* son independientes de las fuerzas físicas, se definieron una serie de variables de velocidad, gravedad, fuerza del salto, etc., destinadas a hacer una simulación simplificada. También fueron definidas variables para comprobar el estado de cada acción.

El funcionamiento es sencillo, redefiniendo la función *fixed_process(delta)* asociada al nodo, cada paso del bucle de ejecución del juego establece inicialmente el vector de velocidad a (0,0), es decir, sin movimiento. Dependiendo de la entrada de movimiento, se comprueba primero la fuerza horizontal a añadir, almacenada en la variable *WALK_FORCE*.

Después, se comprueba si el personaje está en el suelo y, en caso afirmativo, si se ha pulsado la acción de salto. De ser así, se añade una fuerza vertical cuyo valor está almacenado en la variable *JUMP_FORCE*. En caso contrario, se entiende que el personaje está saltando en el aire. A cada paso del bucle se actualiza el tiempo que lleva en el aire, y cuando este alcanza un máximo, deja de aplicarse la fuerza de salto y el personaje cae empleando la fuerza de gravedad artificial.

Siguiendo estos procesos, se obtiene un vector de velocidad definitivo que es el empleado en cada pasada del bucle de juego para mover al personaje.

5.4. Cámara y relación de aspecto

Dado que los escenarios del juego son habitaciones grandes, es necesario dotar a la cámara de movimiento con tal de que pueda seguir al personaje principal. El procedimiento habitual es dotar a la cámara principal que muestra la escena de un *script* propio para seguir al personaje. Sin embargo, otra solución más sencilla es

hacer que el nodo *Camera2D* esté incluido dentro de la estructura del personaje al que tiene que seguir.

Esta solución se adapta a las necesidades del prototipo, que no necesita de cambios ni desplazamientos adicionales de la cámara. Dentro del nodo *Camera2D* puede definirse una propiedad adicional que indica cuál es la cámara principal. Marcando esta opción, se asegura que la imagen mostrada por esta cámara tendrá preferencia sobre el resto.

La resolución de imagen del proyecto es *FullHD*, 1920 x 1080 píxeles. Sin embargo, es posible que este también se ejecute en otros equipos que no alcancen tal resolución o en plataformas móviles. Por ello Godot da la opción de realizar un ajuste automático a distintas resoluciones. Tras comprobar las distintas opciones disponibles para este ajuste, se ha decidido emplear la opción *keep_height*, que mantiene la altura de los gráficos originales aprovechando el máximo espacio posible de la pantalla.

5.5. Animaciones del personaje principal

Tras hacer las pruebas pertinentes para comprobar el comportamiento de los apartados desarrollados hasta este punto, se comenzó el desarrollo de la segunda iteración introduciendo animaciones para el personaje principal. En este apartado se actualizó también el aspecto del protagonista, mostrando su aspecto final.

Ilustración 33: Aspecto final del personaje principal

Existen diferentes alternativas para crear las animaciones dentro de Godot. En este trabajo se ha empleado el nodo *AnimationPlayer*. Este nodo permite editar las distintas animaciones de un personaje desde el propio editor. Para ello es necesario partir de una hoja de animaciones, que contiene todos los fotogramas de cada animación. Las animaciones funcionan mediante el uso de fotogramas clave. Los fotogramas clave sirven para definir el valor de distintas variables. Aplicando esta posibilidad al cuadro visible dentro de la hoja de animaciones del personaje, se han creado las animaciones del mismo. Cada animación recibe un nombre descriptivo y puede ser reproducida mediante *scripts.*

Para obtener una visión más amplia del funcionamiento de esta herramienta, se recomienda la lectura del anexo 3.

5.6. Puertas y transporte entre habitaciones

Las puertas son el primer elemento interactivo a implementar dentro del escenario. Estas son la forma de moverse entre las distintas habitaciones que componen el nivel de demostración del juego.

Ilustración 34: Aspecto de las puertas del juego

Su funcionalidad ha sido implementada aprovechando el sistema de señales de Godot. Este sistema permite a los nodos reaccionar ante ciertos eventos, como pueda ser una colisión o el cambio del valor de una variable interna. Emplear el sistema basado en eventos permite ahorrar comprobaciones constantes en el bucle del juego, haciéndolo más eficiente.

El componente principal de las puertas es un nodo de tipo *Area2D*. Las puertas cuentan con un *script* que recoge las instrucciones necesarias para comprobar cuándo el jugador entra en contacto con ellas. Si el jugador entra en el área de detección de una puerta, se activa su función *_input(event)*. Esta función comprueba si el jugador presiona la tecla de interacción, X. Cuando esto sucede, se invoca a una función de transporte implementada en el script del jugador. Esta función recibe como parámetro las coordenadas de la puerta de destino y mueve al personaje a la nueva posición. Si el jugador sale del área de detección, se desactiva la detección de entrada de la puerta para consumir menos recursos.

5.7. Colisiones unidireccionales para las plataformas

Las plataformas son elementos del escenario que sirven como punto de apoyo para que el personaje protagonista alcance lugares altos y pueda sortear enemigos. Como tal, deben contar con un cuerpo físico y un polígono de colisiones, tal y como el resto de elementos físicos del escenario.

Sin embargo, necesitan un tratamiento especial que escapa a las posibilidades de los *tilemaps*. Es necesario que el jugador pueda acceder a ellas sin chocar cuando salta

desde abajo, pero también que pueda utilizarlas como punto de aterrizaje. Las colisiones unidireccionales cumplen este objetivo. Un cuerpo cuenta con colisiones unidireccionales cuando solo resulta sólido al aproximarse desde una determinada dirección. En el caso de las plataformas, son sólidas al colisionar desde arriba, pero dejan atravesar los objetos desde cualquier otra dirección.

Esta idea, que puede sonar algo extraña en principio, es un problema recurrente en los videojuegos en 2D. Por ello, Godot incluye una forma sencilla de definir este tipo de colisiones utilizando nodos de tipo *StaticBody2D*.

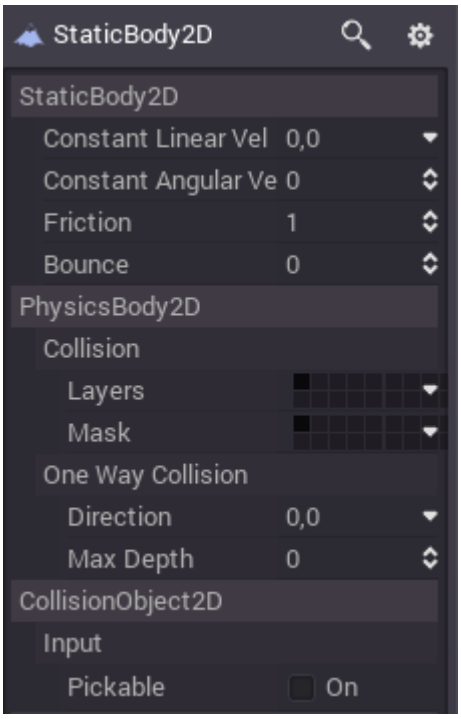


Ilustración 35: Parte de las propiedades del nodo StaticBody2D

Como se puede observar en la ilustración, dentro del apartado *One Way Collision* es posible definir el vector de dirección de la colisión. La variable *Max Depth* indica la profundidad en píxeles que otros cuerpos pueden atravesar en dicha dirección. Por tanto, haciendo uso de estos nodos se puede lograr que Godot gestione este tipo de colisiones sin necesidad de código adicional. Existe la posibilidad de crear estas colisiones utilizando también un cuerpo de tipo *KinematicBody2D*, pero en este caso no tiene sentido ya que las plataformas son un elemento estático.

5.8. Colisiones dinámicas del personaje principal

Para completar el conjunto de movimientos del personaje principal, es necesario hacer que este se agache, que avance mientras está agachado, etc. Con estos movimientos se facilita al jugador pasar por pasadizos estrechos y buscar refugios para huir de los

enemigos. Para ello, debe cambiar la postura del personaje y su polígono de colisión debería reflejar este cambio.

Ilustración 36: Aspecto del personaje de pie y agachado

Una solución sencilla es asignar un polígono de colisión adicional, y alternar su activación conforme el personaje se agache o se ponga en pie. Tras el pertinente cambio en las estructuras de control del estado del personaje dentro de su *script*, se observaron comportamientos que no estaban contemplados inicialmente.

Concretamente, si el jugador suelta la tecla de acción asignada a la acción de agacharse mientras atraviesa una sección del escenario estrecha, el personaje se queda atrapado en la pared y no puede moverse.

Este problema fue solucionado introduciendo un nodo *Area2D* adicional que se activa mientras el personaje está agachado, y fija este estado al detectar el techo. También se adaptó esta solución para la detección del suelo, pues el proceso de pruebas con las animaciones mostró algunos problemas con el cambio entre la animación de salto y caída del personaje.

5.9. Integración del módulo UGK_AI dentro de Godot

A fin de integrar la Inteligencia Artificial del videojuego se ha optado por adaptar el módulo UGK_AI dentro del motor Godot. Para ello, es necesario descargar el código fuente del motor e implementar un módulo personalizado que contenga el código de UGK_AI adaptado, teniendo en cuenta las siguientes consideraciones:

- En los archivos de cabecera es necesario incluir una referencia a “reference.h”, para que la clase pueda formar parte del motor.
- Si se incluye la cabecera “node_2d.h” cada nueva clase será representada como un nodo seleccionable en el editor.

- En los archivos de código es necesario añadir un nuevo método "_bind_methods()". En este método se crea un enlace a la base de datos de objetos del motor, conectando los distintos métodos de la clase al nuevo tipo de objeto. Así se pueden emplear los métodos de la clase dentro del entorno de scripting de Godot.
- Además de las cabeceras y el código del UGK_AI, es necesario añadir dos archivos "register_types.h" y "register_types.cpp" para registrar los nuevos tipos de objeto en la base de datos del motor.
- Adicionalmente, es necesaria la creación de un archivo de configuración básico, "config.py" y un archivo llamado "SCsub", para que la herramienta *Scons* pueda compilar el módulo.

En el anexo 3 se incluyen referencias adicionales a la documentación del motor para conocer el proceso en mayor profundidad.

Cabe destacar que, aun siguiendo todos los pasos necesarios para implementar el nuevo módulo, se ha experimentado una serie de problemas de dependencias entre archivos y errores de compilación de Godot al tratar de introducir el módulo UGK_AI.

Concretamente, el módulo hacía uso de un lector personalizado de archivos HTML para cargar las máquinas de estado. En la versión actual del UGK, este lector hace además uso de la librería externa *LiteHTML*. Al incluir los archivos de esta librería externa, junto con las tres cabeceras adicionales *FSMParser, HTMLParser* y *UGKDefaultHTMLLabels,* aparecen los de compilación mencionados.

Es de suponer que, al ser el UGK una herramienta probada en otras aplicaciones sin ningún tipo de problema, estos errores son derivados de cambios en la especificación del lenguaje C++. La herramienta desarrollada por Xavier Mahiques, tal y como se puede ver en su TFG, fue compilada utilizando la versión de Visual Studio 2015. Sin embargo, al tratar de compilar Godot empleando el compilador de Visual Studio 2015 este da errores de compilación en archivos del código fuente del propio Godot, ya que no está completamente adaptado a dicho compilador. Esto solo deja la posibilidad de compilar el motor empleando el compilador de Visual Studio 2013, el cuál presenta los errores ya mencionados.

Tras tres semanas tratando de salvar estas dificultades, y debido al excesivo retraso que estaban suponiendo en la planificación temporal, finalmente no ha sido posible realizar la adaptación del módulo de forma exitosa.

Por este motivo se ha decidido finalmente adaptar el modelo de máquinas de estados combinando las herramientas de Godot con los procedimientos seguidos dentro del UGK. En el siguiente apartado se explica en detalle cómo se ha solventado este problema.

5.10. Máquinas de estados finitos en Godot

Tal y como se anticipaba en el apartado anterior, finalmente ha resultado imposible implementar el módulo UGK_AI dentro del motor. Sin embargo, Godot sigue ofreciendo algunas utilidades que pueden ser de ayuda a la hora de simular el comportamiento de una máquina de estados finitos.

Los componentes principales de una máquina de estados finitos son tres:

- Los **estados**, a los cuales se puede asociar un comportamiento determinado.
- Las **transiciones** entre estados, que permiten cambiar el comportamiento en función de las necesidades del juego.
- Los **eventos**, que son los disparadores que activan el cambio de estado.

Siguiendo esta estructura, cada enemigo cuenta en el juego con un *script* de Inteligencia Artificial. En dicho *script* hay una lista de estados almacenada como un vector de *Strings*. Cada uno de estos estados tiene una función asociada. Los eventos serán modelados mediante señales que el propio motor puede mandar a los nodos. Cada señal tiene asociado una función que se ejecuta cuando esta señal es emitida. Las transiciones serán, por tanto, las comprobaciones de estado actual y condiciones adicionales dentro de estas funciones, creadas siguiendo el diseño original de las máquinas de estados.

5.11. Implementación del enemigo Soldado

Debido a los problemas al adaptar el módulo UGK_AI al entorno de Godot, finalmente el nodo FiniteStateMachine no aparece en la implementación final. Esta ausencia se suple empleando señales y funciones específicos dentro del script de comportamiento del enemigo.

Ante la ausencia de un gestor dedicado a comprobar los cambios en la máquina de estados, se ha optado por hacer que cada función de IA provoque el cambio de estado necesario al final de su ejecución. Por otra parte, hay transiciones activadas por eventos externos como la aparición del jugador, etc. En la función *fixed_process(delta)*, que actúa como bucle principal del comportamiento, se comprueban los cambios de estado, el estado actual, y se hace que el enemigo actúe

en consecuencia. A continuación, se explicará en detalle cómo se ha abordado el comportamiento de cada uno de los estados del enemigo soldado.

Estado “Patrolling”: *AI_Move()*

Tomando como modelo la implementación del personaje principal, se ha dotado al soldado de una serie de variables que simulan el entorno físico en el que se mueve el cuerpo *kinemático*. A diferencia del script del jugador, en lugar de recoger datos de la entrada por teclado, el soldado cuenta con una ruta predefinida por una serie de nodos distribuidos por la escena. Esta lista de nodos está disponible en la variable *route_points*. Además, se han incluido dos variables: *last_point* y *next_point*, para que el enemigo pueda controlar en qué punto de la ruta se encuentra. Al alcanzar cada punto de esta ruta, el enemigo espera 5 segundos antes de continuar, y pasa al estado “Standing”. Cuando recibe la señal del final de la cuenta atrás, vuelve al estado “Patrolling” si no ha visto al jugador.

Según estos datos, el enemigo calcula si debe moverse a izquierda o derecha. Después se llama a una función, *velocity_calculations(force)*. Esta función complementa los cálculos en caso de tener que saltar, si el enemigo se encuentra sobre una plataforma móvil, etc.

Estado “Alarm”: *AI_Chase()*

Es el estado de alarma del soldado. El comportamiento dentro de este estado varía en función de si puede ver al jugador o no. En caso de ver al jugador, hay un 85% de probabilidad de que el soldado dispare, en cuyo caso el estado actual pasa a ser “Shooting”. En caso de no verlo, empieza la cuenta atrás de 10 segundos para volver al estado normal. Si el soldado no ha disparado, entonces persigue al jugador. Para ello, se ejecuta el código de forma similar a la función AI_Move(), consultando en este caso la posición del jugador en vez de un punto de ruta predeterminado para saber si el movimiento es a izquierda o derecha. Al dispararse este estado, el soldado manda también una señal para cerrar todas las puertas de la sala.

Estado “Shooting”: *AI_Shoot()*

En este estado, el enemigo instancia un nuevo nodo “SoldierShoot”. Se hace uso de la función *yield()* para que la ejecución se sincronice con la animación del personaje. Los disparos viajan en línea recta hasta impactar con un objeto. Si es el jugador, este recibe daño y el disparo desaparece de la escena. En otro caso, el disparo

simplemente desaparece. Tras la ejecución de esta función, el enemigo vuelve por defecto al estado “Alarm”

Estado “Dying”: *AI_Die()*

En este estado, el enemigo reproduce la animación de muerte y desaparece de la escena.

El soldado es el más complejo de los tres enemigos, dado que tiene la capacidad de perseguir al jugador. Para que pueda hacerlo sin problemas, es necesario que este tenga información sobre la posición relativa del jugador respecto a sí mismo, y que pueda detectar obstáculos para sortearlos. Dadas estas características, se ha añadido un componente de tipo Area2D adicional, que activa la condición de salto cuando hay un objeto cercano que impide el paso.

Todo el código del comportamiento de este enemigo se encuentra en el archivo **enemySoldier.gd**, dentro de la carpeta Scripts del juego. Se recomienda su consulta para conocer más detalles sobre la implementación

5.12. Implementación del enemigo Centinela

Este es el enemigo más sencillo ya que tiene menos estados y animaciones, pues su funcionalidad es más limitada en la versión de demostración. Cabe destacar la conexión que existe con el enemigo torreta, ya que cada centinela tiene una variable *myTurret*, que contiene una referencia a una torreta en concreto. Cuando este enemigo entra en estado de alarma, activa dicha torreta y permanece en ese estado hasta que pierde al jugador de vista durante un tiempo. Como en el punto anterior, se va a realizar un breve repaso por la implementación de los distintos estados:

Estado “Patrolling”: AI_Move()

El funcionamiento es similar al de la función *AI_Move()* descrito en el punto 5.13., el enemigo basa la dirección del movimiento en su posición relativa dentro de una ruta predeterminada. Los cálculos, no obstante, son más sencillos ya que este enemigo en concreto está flotando en el aire y no requiere de tantas constantes ni de integrar fuerzas, como si lo hacen el soldado y el personaje principal. Los desplazamientos son siempre en línea recta horizontal y vertical, dependiendo de cuál es la menor distancia a cubrir en línea recta para llegar al siguiente punto. En esta función también se comprueba la dirección en la que está mirando el enemigo para rotar su *Sprite* y su área de visión si fuese necesario.

Estado “Alarm”: AI_Alarm()

En este estado el enemigo invoca la función de activación de la torreta que tiene asociada. También comprueba si está viendo o no al personaje, para poder activar la cuenta atrás antes de volver al estado “Patrolling”

Las transiciones entre ambos estados dependen de dos señales asociadas a nodos auxiliares. Por un lado, si el jugador entra dentro del área definida por el nodo VisionArea, se emite una señal que activa el estado “Alarm” y resetea el cronómetro “AlarmTimer” en caso de estar activo. Este cronómetro, cuando llega a 0, es el encargado de realizar la transición al estado “Patrolling”

5.13. Implementación del enemigo Torreta

La torreta posee una estructura similar a la vista en los dos enemigos anteriores. Cuenta con dos formas de colisión distintas, ya que su forma cambia cuando pasa de activa a inactiva y viceversa. Como se ha adelantado en el punto anterior, su activación no se produce hasta que el centinela asociado a ella detecta al jugador. Estos son sus estados y las funciones asociadas a los mismos:

Estado “Inactive”

Es el estado por defecto, en el que la torreta permanece a la espera de instrucciones. Como tal, no incluye una implementación de código. Cuando el contador AlarmTimer llega a 0 estando en el estado de alarma, la torreta vuelve a este estado.

Estado “Activating” – AI_Activate()

Esta función sencilla emplea la función yield() para sincronizar la animación con la ejecución del código. También se encarga de cambiar entre las dos formas de colisión empleando la propiedad *Trigger* de los nodos *CollisionPolygon2D*.

Estado “Standing” – AI_Alarm()

La torreta permanece activa a la espera de detectar al jugador. Si puede ver al jugador, pasa al estado “Shooting”. En otro caso, comprueba la posición de este para ver si necesita girar al lado contrario, pasando al estado “Turning”. Tras esto inicia la cuenta atrás para volver al estado “Inactive”.

Estado “Shooting” – AI_Shoot()

Similar a la función del soldado, descrita en el punto 5.13., la torreta instancia un nodo de tipo *turretShoot*, un tipo de disparo que hace el doble de daño que los del soldado y se mueve más rápido. Tras el disparo, regresa al estado “Standing”.

Estado “Deactivating” – AI_Deactivate()

Dependiendo de la dirección hacia la que esté apuntando la torreta, esta función reproduce una animación distinta. Después devuelve la torreta al estado “Inactive” y cambia de nuevo el polígono de colisión activo usando los *triggers*.

Como nota adicional, para mejorar el control de las animaciones y no duplicar estados, se incluye en una variable separada, *direction*, si la torreta está apuntando hacia la izquierda o la derecha.

5.14. Implementación de los escondites

Los escondites funcionan de manera muy similar a las puertas. Cuentan con un área capaz de detectar al jugador, y se activan cuando este entra en dicha área. Una vez dentro del área, el jugador puede pulsar la tecla de acción para esconderse. Esto crea una señal que altera una nueva variable booleana dentro del *script* del personaje principal, llamada *hiding*. Mientras esta variable sea cierta, ningún enemigo puede detectar al jugador. Este, por su parte, permanece estático hasta que se vuelva a pulsar la tecla de acción y salga del escondite.

5.15. Implementación de los terminales

Los terminales son el elemento interactivo más complejo. Cada uno de ellos tiene una funcionalidad distinta, por lo que partiendo de la estructura básica compuesta por un Area2D y un Sprite, se ha añadido un nodo de tipo *ProgressBar* para poder mostrar en tiempo real el progreso del *hackeo*. Cada terminal emite una señal distinta cuando la barra de progreso llega al 100%, y es en estas funciones donde se realizan los cambios en el escenario. Esta tabla muestra los distintos terminales, su localización y su efecto:

Terminal	Localización	Función
TW_1	Almacén 1	Abre la puerta del almacén
TW_2	Almacén 2	Activa el ascensor
TE_1	Patio exterior	Inhabilitado, da acceso a una sala de la siguiente fase
TE_2	Patio exterior	Deshabilita los sensores del centinela más próximo
TJ	Prisión	Acceso a la celda del Dr. Dorman, funciona con una llave

Ilustración 37: Tabla de terminales

5.16. Implementación de los menús y el HUD

Siguiendo el diseño de la interfaz del GDD, se han creado dos escenas adicionales para el menú principal y el menú de opciones. Estas escenas están compuestas por nodos tipo *Button*. Como cabe esperar, este tipo de nodos puede recibir todo tipo de señales típicas de cualquier entorno gráfico. Para cumplir con el objetivo del proyecto se han conectado las señales de tipo *ButtonPressed* para detectar cuando el usuario selecciona cada opción. Como podemos ver en la siguiente captura, algunos de los botones como son los del menú de carga y selección del nivel permanecen inhabilitados:

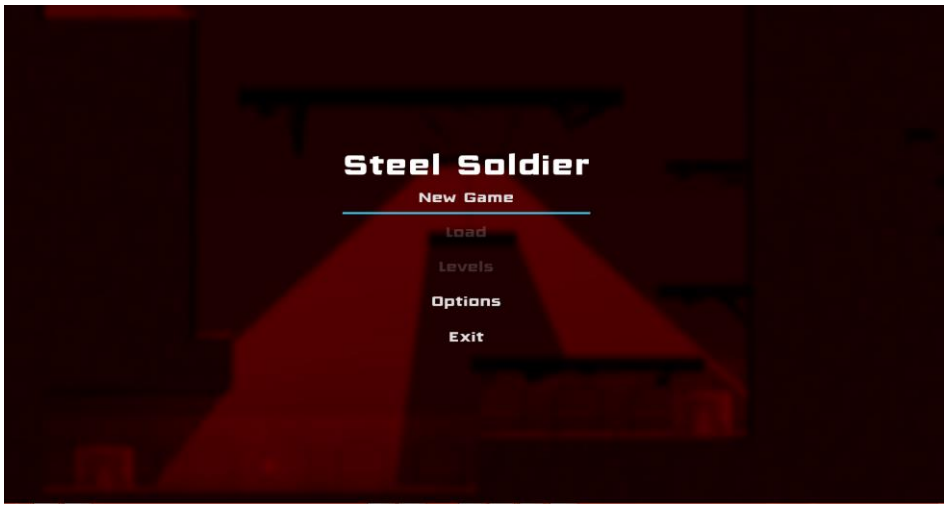


Ilustración 38: Aspecto final del menú inicial

El menú de opciones es un poco más complejo, ya que es necesario acceder a parámetros de la configuración del juego desde la interfaz. Para mantener las opciones de forma consistente a lo largo de toda la ejecución del juego, se ha creado un nodo adicional llamado *GameManager*. Este se encarga de mantener las preferencias como el idioma, así como de almacenar la escena en la que está el jugador, mandar eventos de pausa cuando este presiona la tecla correspondiente, etc. También tiene conectadas todas las señales de la interfaz y se encarga de hacer aparecer y desaparecer los menús. Este es el aspecto de los menús de opciones y pausa respectivamente:

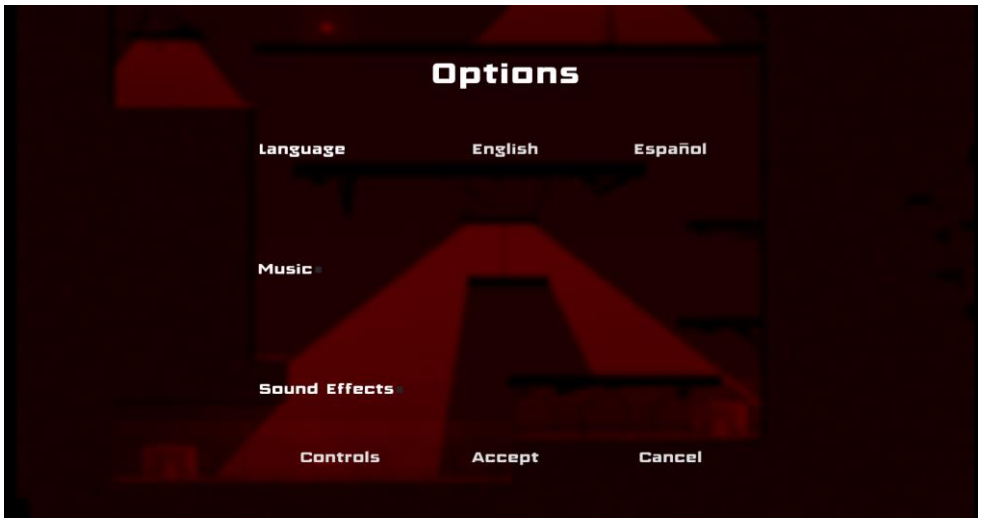


Ilustración 39: Aspecto final del menú de opciones

Ilustración 40:Aspecto del menú de pausa durante la partida

5.17. Iluminación

Dado que los entornos son estáticos, se ha optado por crear en un *Sprite* dentro de cada habitación el contorno de las zonas iluminadas. Estas están superpuestas en una capa por encima del jugador y los enemigos, de forma que se simula la iluminación sin apenas consumir recursos.

No obstante, la iluminación no es un efecto meramente decorativo, sino que afecta al juego. Por ejemplo, los soldados solo pueden detectar al personaje principal cuando este atraviesa una zona iluminada. Por ello, los mapas de luz cuentan con un Area2D asociada. De forma similar a los otros elementos interactivos, esta área detecta cuando el jugador está en las zonas iluminadas, y cambia el valor de la variable *darkness* del jugador. Los soldados consultan el valor de esta variable antes de pasar al estado de alerta o disparar.

5.18. Música y efectos de sonido

Asociado al *GameManager* existe un nodo de tipo *StreamPlayer*. Este nodo puede almacenar las distintas canciones del juego, reproducirlas, y cambiar de canción cuando sea necesario. Este nodo, sin embargo, requiere mayor capacidad de procesamiento para reproducir los sonidos. No hay problema en reproducir una canción, pero para los efectos de sonido es mejor utilizar el nodo *SamplePlayer2D*.

Para emplear estos nodos, es necesario importar las muestras de los efectos de sonido dentro de librerías de muestras. Tras esto es posible asociar diferentes librerías a diferentes nodos, para que cada personaje del juego tenga sus propios sonidos asociados de forma independiente.

5.19. Detalles finales

Tras implementar todos los sistemas de juego, se añadieron al proyecto los gráficos definitivos de los escenarios. Los gráficos de cada habitación están divididos en tres capas: fondo (*Background)*, luces (*Lights*) y primer plano (*Foreground*). Las texturas originales fueron adaptadas al tamaño del escenario construido a partir del *TileMap* en la primera iteración del juego y superpuestas sobre el mismo. Sin embargo, como no todas se adaptaban bien al mapa original, cada habitación se separó en una escena distinta y se creó un nuevo mapa de colisiones a partir de nodos de tipo *CollisionPolygon2D*.

Posteriormente se refinó el comportamiento de las puertas, para que se coordinase la carga de escenas utilizando las funciones definidas en el script del *GameManager*. También se introdujeron algunos cambios en el diseño del nivel y la posición de los enemigos atendiendo a las pruebas realizadas, antes de dar por finalizado el proyecto.

6. Resultados

Tras la realización del proyecto, es necesario pasar a evaluar los resultados obtenidos. En este apartado se hace una valoración del proyecto desarrollado, así como del seguimiento de la planificación inicial a lo largo del mismo.

6.1. Análisis de los resultados

El trabajo desarrollado se ha cerrado con la creación de aproximadamente 1500 líneas de código y más de 20 escenas distintas sin contar las pruebas y tutoriales realizados para obtener experiencia con el motor, que dan forma al prototipo final.

Retomando los objetivos iniciales del proyecto, llega la hora de repasar cuántos de ellos se han podido cumplir. El objetivo principal de desarrollar un prototipo completo empleando el motor Godot ha sido satisfecho. Si bien el comienzo del proyecto fue especialmente duro, dada la nula experiencia previa con el uso de Godot, a lo largo del proceso se ha ganado habilidad y soltura. Una vez familiarizado con la herramienta, esta ha supuesto una gran ayuda para dar forma al videojuego, a pesar de los inconvenientes que han surgido a lo largo del proyecto.

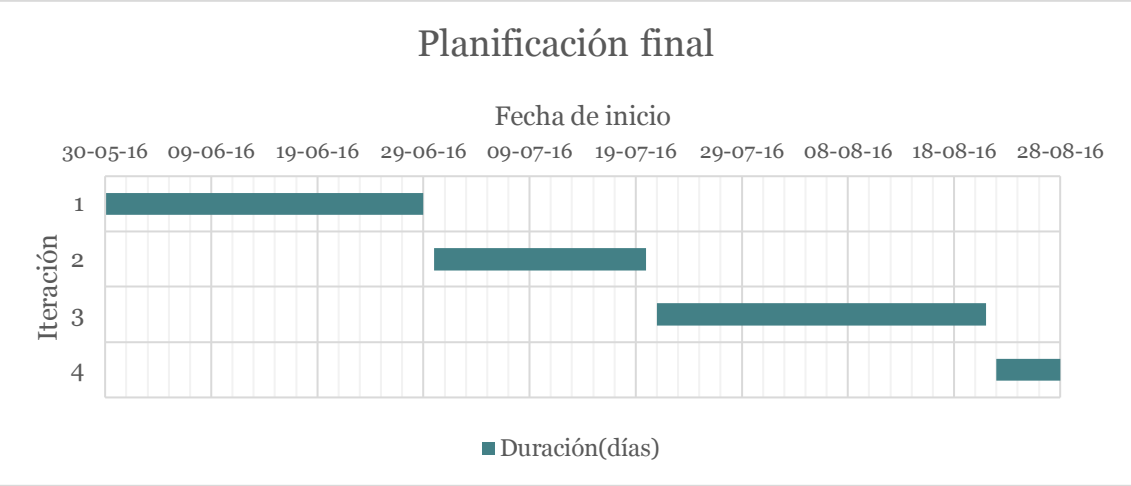
El otro gran objetivo, adaptar el módulo UGK_AI dentro del propio motor, no ha podido completarse con éxito. A pesar de haber dedicado más tiempo del previsto a esta tarea, no ha sido posible resolver los problemas de incompatibilidad con el motor. Hacerlo requería tener un conocimiento más avanzado de Godot a nivel interno, así como del UGK, y la magnitud de trabajo que esto suponía se escapaba del alcance de este proyecto. No obstante, a pesar de este gran problema, ha sido posible llevar el proyecto a buen término y se ha encontrado una buena solución empleando los recursos proporcionados por Godot.

Por otra parte, también se ha cumplido con el objetivo de aportar otro ejemplo que cualquier compañero de la Universidad o bien otros usuarios interesados en el aprendizaje del motor puedan emplear como base para sus propios proyectos. Tras haber trabajado con Godot y comprobar de primera mano que es una herramienta potente y muy prometedora, resulta una gran satisfacción poder aportar un ejemplo público a la comunidad de desarrollo y darle algo más de visibilidad.

Hay que hablar también de la experiencia del trabajo en equipo. Este ha sido muy positivo y el proyecto se ha visto muy beneficiado por el mismo, ya que el apartado artístico del juego supera con creces los resultados que se habrían obtenido de

haberlo realizado en solitario. Además, la experiencia obtenida ha permitido al equipo crecer y prepararse para afrontar proyectos a mayor escala en el futuro.

Por último, se presenta a continuación el diagrama de Gantt correspondiente al desarrollo real del proyecto:



Iteración	Fecha de inicio real	Duración real	Fecha fin real
1	30/05/2016	30	29/06/2016
2	30/06/2016	20	20/07/2016
3	21/07/2016	31	21/08/2016
4	22/08/2016	6	28/08/2016

Ilustración 41: Diagrama de Gantt final

Como puede observarse, ha habido un importante retraso en el desarrollo. La finalización estaba prevista inicialmente a finales de julio, pero el proyecto se ha alargado abarcando también el mes de agosto. Este retraso ha sido producido por dos factores principales. El primero de ellos es la falta de experiencia inicial. Si bien hacer un videojuego partiendo desde cero siempre es complicado, lo es todavía más si no se conocen a fondo las herramientas de desarrollo. El proceso de aprendizaje se prolongó nueve días más de lo previsto. La necesidad de aprender a manejar nodos algo más complejos como el *Area2D* también alargó la segunda iteración seis días más de lo previsto. Una vez superado estos primeros obstáculos, el segundo gran retraso se produjo principalmente por el intento fallido de adaptar el UGK. Como puede observarse en la planificación, el intento de cumplir este objetivo pospuso el resto del proyecto a mediados de agosto. Fue en estas fechas cuando se decidió replantear el proyecto para adaptarlo a los medios ofrecidos por el propio motor, consiguiendo así que estuviese listo para la fecha de entrega.

7. Conclusiones

En vista de todos los puntos expuestos con anterioridad, cabe realizar una serie de reflexiones finales para cerrar el trabajo. Este apartado recoge una reflexión sobre la relación entre los estudios cursados y el trabajo desarrollado, así como una serie de trabajos futuros a desarrollar a partir de los resultados.

7.1. Relación con estudios cursados

El desarrollo de un videojuego engloba en un solo proyecto prácticamente todos los campos vistos a lo largo de la realización del Grado. En este punto se va a exponer de forma más concreta una serie de asignaturas y conceptos estudiados, y su relación con el proyecto.

- **Teoría de Autómatas y Lenguajes:** Primera aproximación a los autómatas finitos o máquinas de estado, la base conceptual tras el modelo de IA del proyecto.
- **Introducción a la Programación de Videojuegos**: Nociones básicas de diseño de juego, técnicas de programación aplicadas a videojuegos, introducción al UGK y C++, etc.
- **Sistemas Multimedia Interactivos Multicanal**: Introducción práctica al desarrollo de videojuegos empleando el motor *Unity*, metodología básica de desarrollo con motores de videojuego y entorno gráfico.
- **Gestión de Proyectos**: Conceptos de planificación y evaluación del proyecto.
- **Ingeniería del Software**: Metodologías de diseño y desarrollo de *software*, creación de documentación empleando UML.

7.2. Trabajos futuros

Tras haber desarrollado el prototipo, cabe la posibilidad de plantear una serie de futuros trabajos relacionados con el mismo. Su realización puede ampliar y mejorar la experiencia con el juego, y puede servir a otros compañeros como base para futuros trabajos.

- **Implementación del módulo UGK_AI en el motor de videojuegos Godot**: Es posible dedicar un trabajo a estudiar más a fondo la integración de ambas tecnologías y realizar un *plugin* gráfico de edición de máquinas de estados. Resultaría también interesante hacer una comparativa del funcionamiento de

los enemigos integrando un gestor de IA y viendo cómo se comporta frente a esta versión.

- **Desarrollo de una versión de Steel Soldier para dispositivos Android**: Fue una de las ideas iniciales del proyecto. Podría servir para comprobar las capacidades multiplataforma del motor.
- **Implementación del módulo de internacionalización del UGK en el motor de videojuegos Godot**: Siguiendo la línea de ampliaciones del motor, podría desarrollarse una adaptación del módulo de internacionalización para facilitar el proceso de traducción del videojuego.
- **Desarrollo de la versión definitiva de Steel Soldier**: Implementando el resto de niveles y enemigos, a partir de esta demostración se podría crear un juego completo empleando como guía el GDD.

8. Agradecimientos

Para finalizar, quisiera dedicar unas breves palabras a las personas que se han implicado en el proyecto. Su colaboración ha sido una de las claves por las que este trabajo ha llegado a buen puerto.

En primer lugar, he de agradecer al Dr. Ramón Mollá, tutor del proyecto, su interés y la dedicación con la que me ha guiado a lo largo del desarrollo de este trabajo.

También quiero agradecer la colaboración del resto de miembros del equipo de desarrollo: Mauro Beltrán, Carlos Mercé y Germán Reina, cuyo talento ha hecho posible que este pequeño proyecto brille con luz propia.

No puedo olvidar mencionar a Xavier Mahiques Sifres, cuyo excelente trabajo ha servido como base para realizar la Inteligencia Artificial del juego.

Por último, pero no por ello menos importante, doy gracias a los familiares y amigos que me han apoyado durante la realización del proyecto por animarme a seguir adelante.

Anexo 1: Bibliografía

En esta sección se recogen algunos documentos de apoyo empleados para la realización del trabajo. Además de la referencia bibliográfica se incluye un breve comentario sobre lo que se puede encontrar en cada uno de ellos.

Trabajo de Fin de Grado de Xavier Mahiques Sifres, donde se profundiza en el uso y diseño de la librería UGK_AI, así como en su implementación en C++:

MAHIQUES SIFRES, Xavier. (2016). Desarrollo del módulo de Inteligencia Artificial basada en máquinas de estado para el motor UPV Game Kernel de UGK-IA. [En línea] Disponible en: http://hdl.handle.net/10251/68616 [Último acceso 26/08/2016]

Manual de documentación de Godot, donde se encuentra una extensa recopilación de tutoriales que cubren desde aspectos básicos a detalles del motor a nivel de funcionamiento interno:

LINIETSKY, Juan; MANZUR, Ariel and the Godot community (2016). *Godot engine documentation.* [En línea] Disponible en: https://media.readthedocs.org/pdf/godot-wiki/latest/godot-wiki.pdf [Último acceso 28/08/2016]

Libro sobre la programación de IA aplicada a videojuegos, cubre un extenso número de tecnologías y aplicaciones, proporcionando ejemplos prácticos de implementación:

SCHWAB, Brian. (2009). *AI game engine programming*. Boston, Mass.: Course Technology (2ª ed.).

Anexo 2: Glosario de términos

GDD: Del inglés *Game Design Document*, es un documento que desarrolla todos los aspectos del diseño del juego: datos generales, historia, interfaz, mecánicas de juego, etc.

Motor de juego: Herramienta de desarrollo de videojuegos que proporciona una serie de características para facilitar la creación de juegos.

Mecánicas de juego: Conjunto de acciones que forman el núcleo de la jugabilidad del videojuego, a través de las que el jugador interacciona con el mundo del juego y cumple los objetivos.

UGK: UPV Game Kernel, es un motor de juego desarrollado en la Escuela Técnica Superior de Ingeniería Informática.

API: Del inglés *Application Programming Interface,* conjunto de subrutinas, funciones y métodos que ofrece una librería para ser empleados por otros programas como capa de abstracción.

UGK_AI: API perteneciente al UPV Game Kernel, destinada a la creación y gestión de Inteligencia Artificial en videojuegos.

Juego de plataformas: Videojuego donde las mecánicas jugables principales se basan en el salto y la habilidad para desplazarse por los escenarios, como *Super Mario Bros.*

Shooter: Juego de disparos

Colisiones unidireccionales: Colisión entre dos objetos que se detecta únicamente al acercarse el uno al otro desde un determinado ángulo.

Steam: Plataforma de distribución digital de videojuegos

Godot: Motor de juego empleado para el desarrollo del proyecto.

Sprite: Textura gráfica asociada a un elemento de juego

Tilemap: Conjunto de *sprites* repetidos llamados *tiles*, estructurados en una rejilla, que sirven para dar forma al mundo del juego.

Tileset: Archivo donde se guardan todos los elementos que dan forma al *tilemap*.

Vector: Estructura de datos que almacena diversas variables dentro de un objeto.

Script: Archivo de código que sirve para modelar comportamientos dentro del videojuego.

Plugin: Extensión del editor que añade funcionalidades extra.

Máquina de estados finitos: Modelo computacional que se basa en la creación de una serie de estados, transiciones y eventos. Su uso en IA de videojuegos está muy extendido.

Anexo 3: El motor Godot

Este anexo tiene como objetivo ofrecer una visión general del motor, así como servir de ayuda para que cualquier persona interesada en usarlo pueda dar sus primeros pasos en la creación de videojuegos utilizando Godot. Para ello, se explicará en profundidad cuál es el concepto de los nodos y escenas, cómo funcionan algunos de ellos, las propiedades editables desde el entorno gráfico, etc. Se incluye para apoyar estas explicaciones una serie de capturas de la interfaz del motor.

También se hará una explicación de cómo estructurar un proyecto en Godot y se incluirán algunos tutoriales adicionales para reforzar los conceptos presentados. En definitiva, es una guía que trata de cubrir todos los aspectos básicos de la creación de un videojuego para ayudar a comprender como está hecho Steel Soldier y también cómo crear nuevos juegos.

Esta guía ha sido escrita tomando como referencia la versión 2.0.2 del motor, por lo que todas las capturas de pantalla, consejos y explicaciones se basan en esta versión. El aspecto del motor y su funcionamiento pueden cambiar en posteriores versiones, añadiendo funcionalidades nuevas. Por ello se recomienda también la lectura de la documentación oficial de Godot[20].

Nodos y escenas

Los nodos son los componentes básicos con los que dar forma al mundo del juego. Existe una gran variedad de tipos de nodos, y cada uno de ellos cumple un objetivo diferente. Por ejemplo, los nodos de tipo *Sprite* sirven para almacenar y mostrar una textura, y los nodos de tipo *Camera2D* sirven para añadir una cámara a una escena en 2D. Cada tipo de nodo posee una clase dentro del lenguaje de *scripting* de Godot, el GDScript, con una serie de funciones y variables que sirven de ayuda para programar el videojuego.

Las escenas dentro de Godot son agrupaciones de uno o más nodos. Los nodos pueden relacionarse entre sí para complementar su funcionalidad, permitiendo la creación de objetos y comportamientos más complejos que los tipos básicos. Por ejemplo, pueden juntarse un *Sprite* que muestra el aspecto de un objeto del juego con un polígono de colisión, definido con un nodo *CollisionPolygon2D*, para crear así un

[20] Documentación oficial de Godot, disponible en este enlace: http://docs.godotengine.org/en/latest/index.html

elemento sólido dentro del mundo del juego, capaz de detectar colisiones. Estas agrupaciones de nodos crean una estructura en forma de árbol, donde cada nodo excepto el nodo raíz, cuelga a su vez de un nodo "padre". Además, cada escena tiene un único nodo raíz.

Lo interesante de esta organización es que permite un diseño e implementación modulares. En otros motores de juego, cada escena es una fase o nivel de juego, con todos sus elementos incluidos. En Godot, una escena puede ser solamente el personaje principal, otra escena puede ser el escenario, y una escena global puede agrupar ambos elementos. Esto hace más sencillo dividir tareas y trabajar simultáneamente en distintos aspectos del juego sin que estos influyan en el trabajo de otros miembros del equipo.

Creando un nuevo proyecto

Al abrir Godot por primera vez, este mostrará el gestor de proyectos con la lista de proyectos creados. Como es de esperar, esta lista estará inicialmente vacía, aunque es posible descargar una serie de ejemplos[21] junto con el motor. Este es el aspecto de dicho diálogo, con algunos de los ejemplos descargados:

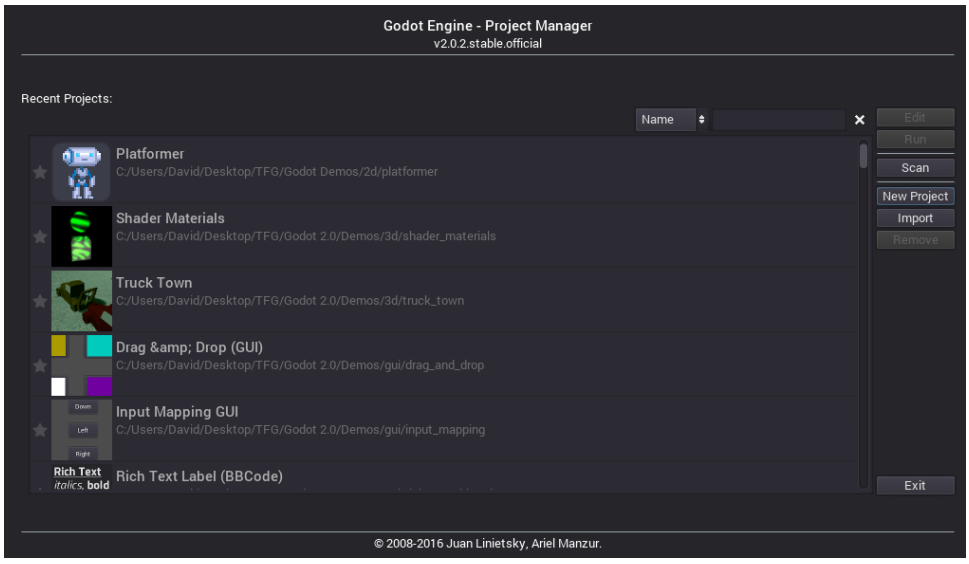


Figura III.1: Gestor de proyectos

Los botones *Edit* y *Run* permiten, como su nombre indica, editar un proyecto o ejecutarlo. Permanecen inactivos hasta que se selecciona un proyecto de la lista. El botón *Scan* abre el explorador de archivos para seleccionar una carpeta y añadir los proyectos que contenga a la lista. *New Project* permite crear un nuevo proyecto desde

[21] Página de descarga del motor, incluye proyectos de ejemplo: https://godotengine.org/download

cero. *Import* sirve para importar un único proyecto proporcionando la ruta donde está almacenado. Por último, *Remove* elimina un proyecto seleccionado de la lista del gestor.

Al crear un nuevo proyecto aparece un diálogo adicional en el que hay que introducir la ruta del proyecto y el nombre del mismo. Este nombre podrá cambiarse más adelante desde las propiedades del proyecto. Una vez introducidos estos datos, el proyecto es añadido a la lista y podemos utilizar el botón *Edit* para abrir la interfaz de edición, que aparecerá así:

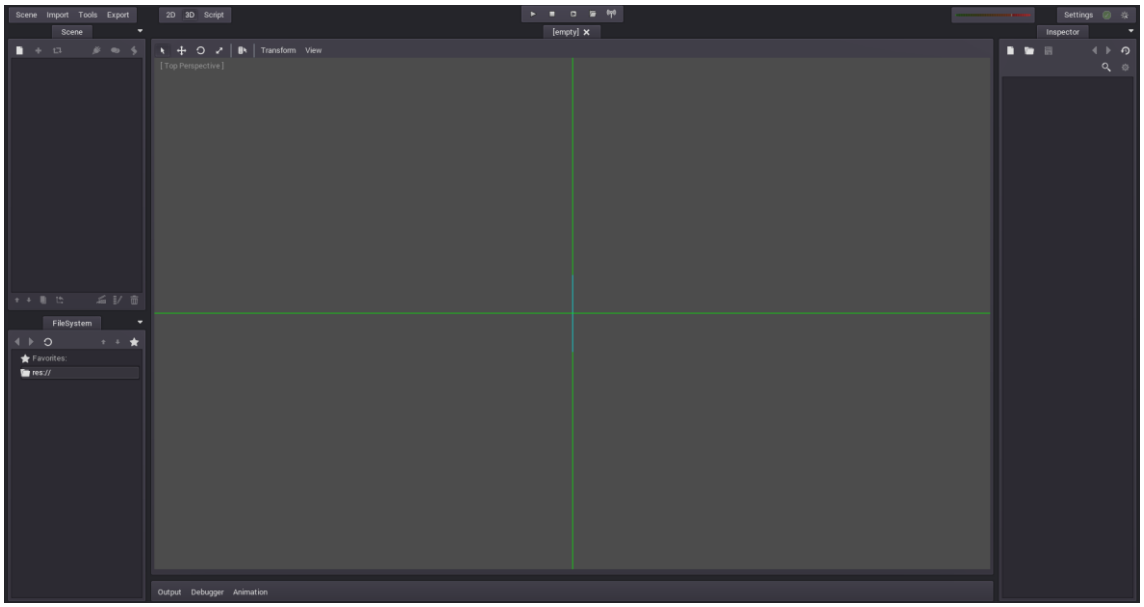


Figura III.2: Ventana de edición de un nuevo proyecto

Antes de continuar explicando cómo funcionan los nodos y las escenas, conviene realizar un repaso a las opciones que presenta la interfaz de edición.

Interfaz del editor

Como puede verse en la Figura III.2, el editor está organizado en distintos paneles con barras de herramientas. En este apartado se va a explicar cuál es la función de cada uno de estos paneles, así como de los botones que se pueden ver en pantalla.

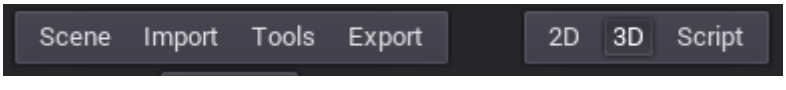


Figura III.3: Barra de herramientas principal y de cambio de vista

En la parte superior izquierda se puede ver la barra de herramientas principal, compuesta por los botones *Scene*, Import, *Tools* y *Export*.

El botón *Scene* da acceso a las opciones de crear, guardar y cargar una escena, rehacer y deshacer los últimos cambios, etc. Además, también permite acceder a la configuración del proyecto, cerrar el editor o volver a la ventana del gestor de proyectos.

El botón *Import* permite importar nuevos recursos como texturas en 2D, modelos 3D, fuentes de texto, muestras de audio, nodos de otras escenas, etc.

El botón *Tools* por el momento da acceso únicamente al explorador de recursos *huérfanos*, es decir, recursos que no están siendo utilizados por ningún nodo en el proyecto.

Por último, el botón *Export* permite construir versiones ejecutables del proyecto para distintas plataformas. Este proceso se abordará con más detalle en el apartado *Exportando un proyecto*.

A la derecha de esta barra de herramientas se pueden ver otros tres botones, 2D, 3D y Script. Estos botones controlan la apariencia de la ventana central de edición. Permiten cambiar entre las vistas bidimensional, tridimensional y el editor de scripts incluido dentro del propio motor.

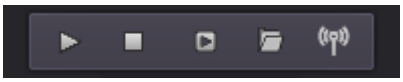


Figura III.4: Barra de reproducción

Los botones de la barra de reproducción permiten ejecutar el proyecto, parar la ejecución, ejecutar la escena que se está editando, ejecutar una escena con opciones personalizadas y cambiar las opciones del modo *Debug*. Estas opciones incluyen la posibilidad de editar la escena en tiempo real mientras se prueba, crear versiones de prueba en dispositivos remotos y hacer visibles los polígonos de colisión y caminos definidos en la escena.

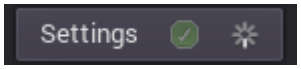


Figura III.5: Opciones e indicadores adicionales

El último botón de la zona superior abre el menú de opciones del editor, donde se puede personalizar el aspecto y la disposición de los elementos de la interfaz. El icono central es un indicador que muestra cambios externos en alguno de los archivos de recursos del proyecto, y el icono de la derecha muestra una animación cada vez que la ventana de edición se vuelve a pintar.

Figura III.6: Panel de escena

El panel de escena muestra el contenido de la escena que se está editando actualmente. La zona central es donde se mostrará la estructura de nodos una vez que estos sean creados. Los botones de la parte superior son, de izquierda a derecha, los siguientes:

- Añadir/Crear un nuevo nodo
- Instanciar otra escena como nodo de la escena actual
- Cambiar el tipo del nodo seleccionado en la lista
- Editar y conectar las señales de los nodos
- Gestionar los grupos de nodos
- Añadir o editar el script asociado a un nodo

Si bien la funcionalidad de estos botones es bastante clara, cabe hacer algunas aclaraciones sobre los botones de edición de señales y gestión de grupos.

En Godot, algunos de los nodos tienen asociadas unas señales predeterminadas[22]. Estas señales se disparan al ocurrir un determinado evento. Por ejemplo, un nodo *AnimationPlayer*, que sirve para reproducir animaciones, contiene una señal *finished* que marca el final de la animación. Estas señales resultan muy útiles para comunicar eventos a otros nodos, sincronizar la ejecución del código, etc. También se pueden crear y emitir señales personalizadas, y asociar funciones a las mismas. Se mostrará un ejemplo práctico más adelante.

Los nodos pueden también pertenecer a un grupo de nodos definido por el usuario. No hay que confundir estos grupos de nodos con la estructura de las escenas, pues son diferentes. Por ejemplo, un enemigo en el juego puede estar formado por diferentes nodos asociados entre sí en la escena formando un árbol. El nodo raíz de este árbol, a su vez, puede pertenecer al grupo “Enemigos”. Los grupos sirven para organizar el proyecto y acceder a nodos concretos desde los *scripts* cuando por ejemplo no sabemos su ruta exacta o bien esta sufre un cambio.

Los botones de la parte inferior tienen como objetivo reorganizar la lista de nodos, y sus funciones son las siguientes:

- Subir el nodo en la lista
- Bajar el nodo en la lista
- Duplicar un nodo
- Cambiar el nodo “padre” de otro nodo
- Crear una nueva escena a partir de un nodo
- Editar múltiples nodos seleccionados a la vez
- Borrar un nodo de la escena

Como nota adicional, en la creación de un juego 2D el orden en el que aparecen listados los nodos es también el orden de dibujado de los mismos. Por ello, los nodos que aparecen más abajo en esta lista son dibujados por encima de los nodos superiores. Esto puede cambiarse alterando la posición Z dentro de las propiedades del nodo, pero es un detalle a tener en cuenta ya que si la lista se organiza de forma correcta no es necesario hacerlo.

También cabe destacar que el diálogo que se abre al seleccionar el botón de nuevo nodo muestra inicialmente la lista de todos los nodos disponibles e incluye un buscador para encontrar nodos más fácilmente.

[22] Los tipos de nodo y las señales que tienen asociadas pueden consultarse en la referencia de clases del lenguaje GDScript, disponible en este enlace: http://docs.godotengine.org/en/latest/classes/_classes.html

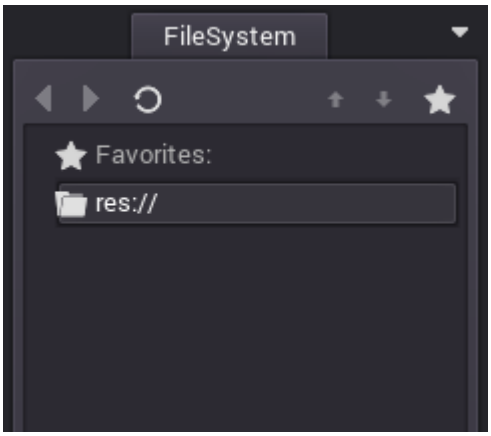


Figura III.7: Panel de sistema de archivos

Este panel muestra el sistema de archivos del proyecto, incluyendo todos los recursos almacenados dentro de las carpetas que forman parte del mismo. Al seleccionar una carpeta o archivo aparecen botones con funcionalidades típicas tales como abrir el archivo, cambiar su nombre, borrarlo, etc.

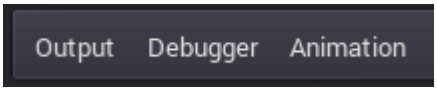


Figura III.8: Salida, Debugger y Animaciones

A la derecha de este panel se encuentran los botones que muestran la salida por consola de texto, el menú de debug y el editor de animaciones. Estos paneles pueden mostrarse u ocultarse cuando el usuario lo necesite. Más adelante se hablará en detalle del panel de edición de animaciones.

Figura III.9: Herramientas de selección y edición de escena 2D

Encima del panel de previsualización de la escena encontramos las herramientas de escena. Estas herramientas ayudan a la navegación por la escena y el posicionamiento de nodos u objetos en la misma. Las funciones de estos botones son:

- Modo de selección de objetos
- Cambio de posición del objeto
- Cambio de la escala del objeto
- Listado de todos los objetos seleccionados
- Desplazamiento de la vista, se puede realizar también pulsando la rueda del ratón.
- Fijar un objeto en su posición

- Liberar el objeto de su posición
- Vincular todos los hijos de un nodo al mismo, para que se muevan de forma conjunta.
- Desvincular los nodos hijos para que sean independientes
- Editar las propiedades de edición: Ver una rejilla como guía, aplicar y configurar guías de posicionamiento, etc.
- Cambiar las propiedades de la vista como el zoom, que también puede modificarse con la rueda del ratón, o centrar la vista en el objeto seleccionado.

Esta barra cambia ligeramente en la vista 3D, ya que por ejemplo no tienen sentido las opciones de fijar objetos en una posición de la rejilla guía. En la vista de script este menú también cambia, mostrando opciones propias de un editor de texto estándar.

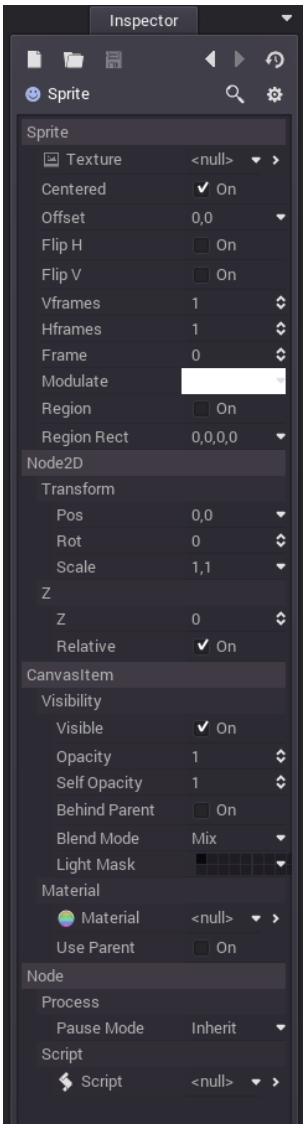


Figura III.10: Inspector de objetos – ejemplo de Sprite

El último panel, situado a la derecha de la pantalla, es el inspector de objetos. En este inspector podemos ver las propiedades editables del nodo seleccionado. Por ejemplo, algunas propiedades en el caso de los *sprites* son la textura asociada al mismo, el número de *frames* en horizontal y vertical si la textura es una hoja de animaciones, etc.

Como el nodo *Sprite* hereda atributos de los tipos *Node2D* y *CanvasItem*, también se pueden ver algunas propiedades que pertenecen a estos nodos, como por ejemplo la posición, rotación y escala del *sprite* definidos en la clase *Node2D*. En próximos apartados se detallarán más algunos de estos atributos, pero para una descripción completa de todos ellos se recomienda leer la referencia de clase del lenguaje *GDScript* dentro de la documentación de Godot.

Creando una nueva escena

Al crear una nueva escena vacía, lo primero que debe hacerse es añadir o crear el nodo raíz. Este nodo será el padre del resto de nodos de la escena, y será posible consultar todos los atributos de estos cuando se esté escribiendo el *script* de comportamiento del nodo.

Los nodos hijos pueden complementar la funcionalidad del nodo padre, por lo que en la mayoría de los casos no importa cuál sea el nodo raíz de la escena. No obstante, es una buena práctica elegir siempre el mismo tipo de nodos como raíz para crear elementos similares, y cambiar los nombres apropiadamente para saber qué función cumple cada uno dentro de la escena.

Dado que el nodo raíz tiene fácil acceso al resto de nodos mediante script utilizando la función *get_node()*, es recomendable asociar el script de comportamiento siempre al nodo raíz.

Tipos de nodos y aplicaciones

Cubrir todos los tipos de nodos que tiene Godot sería tarea suficiente para un libro entero. En este apartado se va a hacer una explicación detallada de algunos de los tipos más útiles y sencillos, que permiten crear juegos básicos.

Node2D

Es el tipo básico del que descienden el resto de nodos en un entorno 2D. Almacena los parámetros de posición, rotación y escala de los objetos. También almacena la variable de profundidad que indica la capa en la que el nodo tiene que ser renderizado.

Normalmente se utiliza como nodo raíz en las escenas globales de juego para integrar otras escenas más pequeñas, puesto que su funcionalidad es limitada. Es un nodo que también se puede emplear para crear entidades invisibles que gestionan el juego a nivel interno.

Sprite

Nodo que almacena una textura básica. Esta textura puede ser el aspecto de un objeto estático, una hoja de animaciones o incluso un *tileset* con el que construir una escena *tilemap*. Algunos valores adicionales son *Flip H* y *Flip V*, que permiten voltear la textura horizontal y verticalmente, muy útiles en algunas animaciones.

También es posible emplear los valores *Vframes* y *Hframes* para almacenar el número de columnas y filas de una hoja de animaciones. Una vez dados estos valores, Godot trocea automáticamente la imagen siguiendo el número de divisiones proporcionado por el usuario, y se puede elegir el cuadro de animación actual estableciendo el valor de la variable *frame*.

Otra posibilidad es seleccionar un área concreta de la textura, pasando las coordenadas X, Y, el ancho y el alto, y activando la variable *Region*. Este es un procedimiento habitual cuando se quiere evitar fraccionar un *tileset* en decenas de archivos distintos.

Tiene algunas propiedades adicionales heredadas de los *CanvasItem*, como opacidad, visibilidad, etc.

AnimatedSprite

Nodo similar al *Sprite*, que en lugar de una sola textura guarda todos los cuadros de animación de forma separada. Resulta útil cuando las animaciones de objetos y personajes están en archivos distintos en lugar de una sola hoja de animaciones.

StaticBody2D

Nodo que cuenta con un cuerpo estático. Especialmente útil para crear elementos sólidos que ningún personaje del juego pueda mover. También es posible dotarlos de velocidad linear y angular para darles un movimiento independiente a las físicas del juego, así como definir un factor de fricción y elasticidad para hacer que los cuerpos que toquen este elemento reboten.

KinematicBody2D

Nodo que cuenta con un cuerpo cuyo movimiento está controlado de forma independiente a las físicas. Es el nodo ideal para hacer personajes principales que dependen de la entrada por teclado del usuario, o bien personajes y elementos cuyo movimiento se quiere controlar mediante un *Script* personalizado, y en el que no se quiere que intervenga la gravedad del motor de físicas.

RigidBody2D

Nodo que cuenta con un cuerpo cuyo movimiento se ve afectado por el motor de físicas. Es un nodo más complejo que los dos anteriores, al que se puede asociar una masa, peso, escala de gravedad, etc. Su uso requiere de una mayor capacidad de procesamiento. Resulta apropiado para crear elementos con físicas más avanzadas.

Propiedades comunes de los distintos tipos de cuerpo

Una propiedad interesante que merece la pena mencionar es lo que en el editor aparece como *One Way Collision*, las colisiones unidireccionales. Si dotamos al cuerpo de un vector de dirección, podemos controlar que sea sólido únicamente por una de sus partes. Esto permite por ejemplo crear plataformas que sólo son sólidas al caer en ellas desde arriba, o elementos que permitan el paso de los objetos solo desde una posición determinada. Lo primero se conseguiría definiendo la variable *Direction* como (0,1), haciendo que el objeto solo sea sólido por la parte superior. Para el segundo caso, si por ejemplo se quiere hacer una puerta que solo se pueda atravesar por la derecha, se lograría definiendo *Direction* como (-1,0). Es muy importante asignar un valor mayor que 0 a la variable *Max Depth*, que define el número de píxeles que otros objetos pueden atravesar en la dirección elegida. De lo contrario, aunque se defina la dirección, la colisión no tiene efecto y el objeto sería completamente sólido.

También es importante tener en cuenta que los tres nodos anteriores por sí solos no dotan a los objetos de una forma de colisión, y sin ella, no hay forma de que interactúen con otros cuerpos. Para hacer esto es necesario añadir un nodo hijo de tipo *CollisionShape2D* o *CollisionPolygon2D* que los complemente.

CollisionShape2D y CollisionPolygon2D

Estos nodos complementan la funcionalidad de los nodos derivados de *PhysicsBody2D*, dotándolos de una forma de colisión funcional. *CollisionShape2D* es un tipo de nodo disponible únicamente en el editor gráfico. A este nodo se le pueden

asociar distintas formas predefinidas como rectángulos, círculos, cápsulas, etc. Es posible modificar parámetros como la posición y la escala de las formas para adaptarlas a las necesidades del cuerpo en cuestión.

Otra posibilidad para crear formas más complejas, o tener un mayor control de edición de las mismas, es utilizar los nodos CollisionPolygon2D. Cuando se selecciona uno de estos nodos, aparece en el editor un icono en forma de lápiz. Haciendo *click* sobre la ventana de escena con la herramienta seleccionada se pueden crear todos los vértices del polígono. El polígono resultante se cierra cuando se vuelve a seleccionar el primer punto, y Godot se encarga de dividir las figuras complejas en polígonos más simples de forma automática. También es posible mover estos puntos en caso de error, o volver a redefinir el polígono por completo si fuese necesario volviendo a elegir la herramienta de edición.

TileMap

Nodo que crea una rejilla sobrepuesta sobre la escena, en la que es posible situar elementos gráficos repetidos, los tiles, para dar forma al mundo del juego. La variable *Mode* permite por defecto crear rejillas con forma cuadrada o con celdas isométricas. También es posible definir el número de *tiles* en la variable *Size*, y el tamaño de las celdas con la variable *Quadrant Size*. Y-Sort permite simular la profundidad en juegos con perspectiva cenital o isométrica. Utilizar *tilemaps* evita los problemas de cargas de textura demasiado grandes, como pueden ser la ralentización del juego o fallos gráficos por exceder el tamaño máximo de textura que Godot puede cargar sin problemas.

Timer

Permite crear un cronómetro. Desde el propio editor se puede establecer la duración de la cuenta atrás en segundos, en la variable *Wait Time*. La variable *One Shot*, cuando se activa, hace que la cuenta atrás solo se realice una vez. Si está desactivada, el nodo realiza la cuenta atrás una y otra vez sin pausa. *Autostart*, como su propio nombre indica, hace que la cuenta atrás empiece de forma automática. Dentro de *GDScript* existen las funciones Timer.start() y Timer.stop() para activar o desactivar la cuenta atrás a voluntad. Este tipo de nodos es muy útil para crear pausas dentro de la ejecución de los scripts.

Crear y utilizar un *Tilemap*

Para poder emplear un nodo de tipo *TileMap*, es necesario disponer primero de un *TileSet* o conjunto de *tiles*. Este recurso es similar a una hoja de animaciones, puesto que contiene todos los gráficos de *tiles* de forma ordenada. En Godot, es posible crear una escena para dar forma a un *TileSet* sin recurrir a herramientas externas, y además se puede utilizar como base de los *tilemaps*. Para ello hay que crear una nueva escena, con todos los nodos *Sprite* correspondientes al *tileset* que queremos crear, dispuestos en forma de rejilla, como se muestra a continuación:

Figura III.11: Escena de edición de un tileset

Hay que hacer algunas anotaciones interesantes sobre este tipo de escenas. Para empezar, el nombre que se asigne a cada *Sprite* será el nombre con el que aparezca en el editor de *tilemaps* de Godot. Es recomendable, sobretodo trabajando con *tilesets* grandes, que cada *Sprite* esté nombrado correctamente. También es recomendable utilizar la opción *Edit/Use* Snap, para que los *tiles* queden perfectamente alineados. Se puede cambiar la dimensión de la rejilla en el menú *Edit/Configure Snap…*, cambiando el valor de la variable *Snap Step*.

Se puede además asignar un *StaticBody2D* acompañado de una *CollisionShape2d* o un *CollisionPolygon2D*, para darles un cuerpo físico y crear así un mapa de colisión del escenario de forma automática. Si se mantiene activada la opción *Use Snap*, los polígonos se ajustarán de forma perfecta a los *tiles.* Por último, hay que tener en cuenta que los *tilesets* solo almacenan colisiones de forma muy básica. Si por ejemplo se quiere crear un objeto que contenga *One-Way Collisions*, será necesario hacerlo de forma independiente fuera del tileset.

Una vez creado el *tileset*, hay que exportarlo para crear un recurso que Godot pueda utilizar como base para crear un nodo *TileMap*. Para ello hay que seleccionar la opción *Scene/Convert To…/Tileset*, y asignar un nombre al nuevo tileset. Podemos exportar la misma escena varias veces, añadiendo nuevos *sprites* para ampliar el *tileset*. Para ello, en el menú de conversión, se muestra la opción *Merge with Existing*. Hay que tener en cuenta que a veces esta opción no gestiona de manera adecuada cambios en los polígonos de colisión del *tileset*, por lo que a veces resulta conveniente crear un nuevo recurso para introducir los cambios con éxito.

Ejemplo: Plataforma estática

En el primer ejemplo se mostrará cómo crear una plataforma estática, haciendo uso de colisiones unidireccionales para que esta sea sólida únicamente por su parte superior. Para empezar, hay que crear una nueva escena cuya raíz será un nodo de tipo *StaticBody2D.* Como ya se ha mencionado, estos nodos necesitan un nodo *CollisionShape2D* o bien uno tipo *CollisionPolygon2D* para que las colisiones tengan efecto. Por su mayor facilidad de edición y comodidad, en este ejemplo se utilizará un nodo *CollisionPolygon2D*. Además de esto, se añadirá un *Sprite* para mostrar el aspecto de la plataforma. Una vez añadidos todos los nodos a la escena, esta es la estructura resultante:

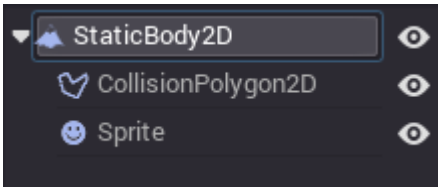


Figura III.12: Estructura de nodos de la plataforma

No importa la posición del *CollisionPolygon2D* y el *Sprite* dentro del árbol de escena, siempre y cuando sean hijos del nodo *StaticBody2D*. Ahora toca personalizar los nodos para que cumplan su función. Empezando por el *StaticBody2D*, hay que añadir un valor a los parámetros *Direction* y *Max Depth* del apartado *One Way Collision*. Como se busca que la plataforma sea sólida por arriba, seleccionamos la coordenada y del vector *Direction* y ponemos su valor a 1.0. En *Max Depth* puede ponerse un valor arbitrario, por ejemplo, 20 deja que otros nodos se superpongan ligeramente sobre la plataforma.

Ahora hay que darle una forma a la plataforma. Es mejor empezar asignando un *Sprite,* ya que de lo contrario habría que editar el polígono a ciegas. Con el *Sprite* seleccionado se puede pasar a cargar la textura, eligiendo la opción *Load* que aparece

al pinchar sobre la pestaña desplegable junto a la previsualización de la textura. Una vez cargada la textura, se puede mover el *Sprite* donde sea necesario, aunque es recomendable mover toda la escena de forma conjunta una vez que se integre en otros escenarios.

Ahora que la plataforma ya tiene textura, podemos crear su polígono de colisión. Para ello se selecciona el nodo *CollisionPolygon2D*. Una vez hecho esto, se puede ver como aparecen dos nuevos iconos en la barra de edición de escena, en forma de lápiz y punto rojo. Pulsando el botón del lápiz se puede entrar al modo de edición de polígonos. A continuación, se pulsa sobre los cuatro extremos de la parte superior de la plataforma, volviendo a pulsar en el punto inicial para cerrar el polígono. Si todo ha ido bien, la escena debería tener un aspecto parecido a este:

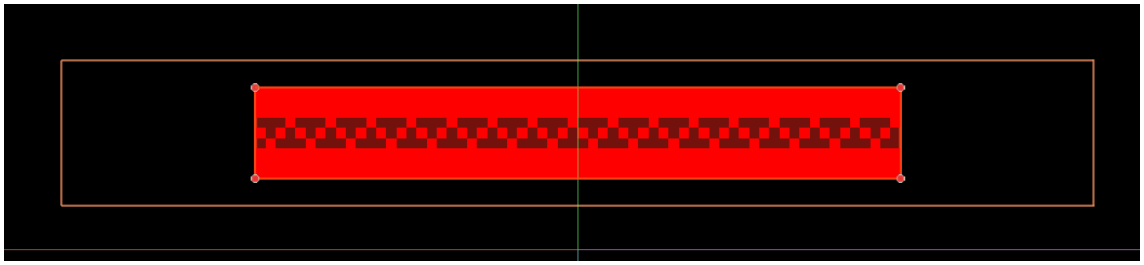


Figura III.13: Aspecto de la plataforma

Igual que con la creación de *tilemaps*, puede resultar muy útil habilitar las opciones de *Snap* y configurar el tamaño de la rejilla. Incluso si la plataforma no se adapta por completo a la rejilla, es muy recomendable habilitar la opción *Use Pixel Snap*, ya que esta solo permite pulsar sobre las coordenadas exactas del límite entre los píxels. Así se puede realizar un ajuste fino de la forma del polígono, y es más fácil y rápido crear rectas perfectas.

Gestión del *Input*

En cualquier videojuego es de vital importancia gestionar la entrada de instrucciones del jugador. Como la mayoría de motores gráficos, *Godot* cuenta con una serie de facilidades para realizar esta tarea. La primera y más útil de ellas es la creación de *Input Actions* (Acciones de Entrada). Estas acciones son etiquetas de texto a las que se pueden asignar una o más teclas del teclado, botones del ratón o incluso de un joystick compatible. Definirlas permite mapear los controles del juego siguiendo el GDD y asociar la entrada a etiquetas fáciles de memorizar creadas por el propio usuario. Para hacerlo, hay que acceder al menú *Scene/Project Settings*, y seleccionar la pestaña *Input Map*

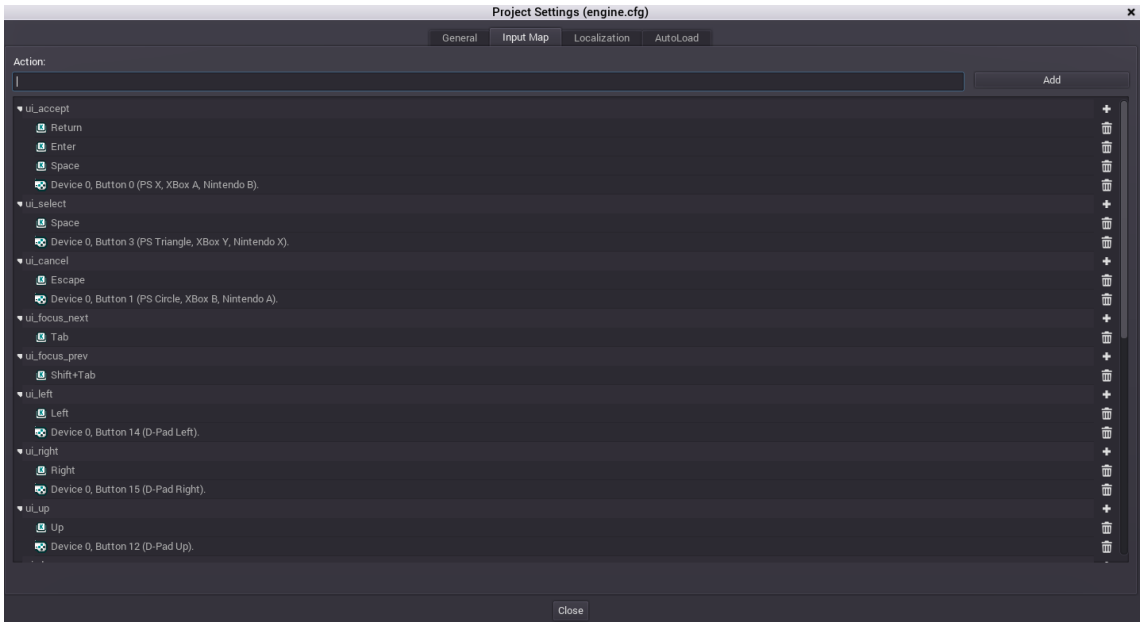


Figura III.14: Aspecto del menú Input Map

Es posible añadir tantas acciones como sea necesario, e incluso asignar las mismas teclas o botones a distintas acciones para que cumplan funciones distintas según el contexto. Por ejemplo, la flecha derecha puede controlar el movimiento del personaje durante el juego y seleccionar opciones en las pantallas de menú.

Dentro de *GDScript*, se puede detectar la pulsación de una de las teclas definidas como acción con la función *Input.is_action_pressed("action")*, sustituyendo "action" por el nombre de la acción que se quiera comprobar. Esto facilita la creación de código ya que permite no tener que recordar todos los nombres asociados a cada tecla o botón de entrada por el motor.

Ejemplo: Teletransportador

El segundo ejemplo es algo más complejo que el anterior. Esta vez, se creará un teletransportador capaz de enviar un objeto cercano a otro punto de la habitación cuando se pulse la tecla de acción, X. Lo primero es definir una nueva acción en el menú *Input Map*, que se llamará *teleport*. Para hacerlo hay que seguir las indicaciones del punto anterior.

Una vez hecho esto, se creará una nueva escena con un nodo *Area2D* en la raíz. De este nodo colgarán dos hijos, un *Sprite* y un *CollisionPolygon2D*. Será necesario añadir un pequeño *Script* al *Area2D* para gestionar la entrada y hacer el teletransporte, pero eso vendrá más adelante. Por ahora, hay que añadir una textura al *Sprite* y darle forma al polígono de colisión. Los polígonos de colisión que descienden de un nodo

Area2D no son sólidos, sino que delimitan el contorno del área. Para hacer el teletransportador, se utilizará el gráfico de las puertas de *Steel Soldier*. Este es el aspecto inicial de la escena tras hacer todos los pasos que se han explicado hasta ahora:

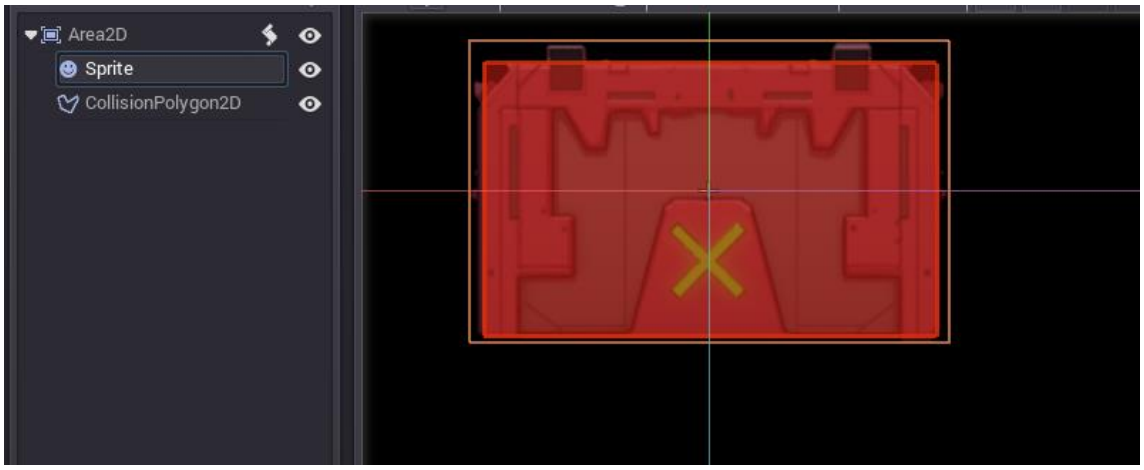


Figura III.15: Aspecto inicial del teletransportador

Ahora es cuando las cosas se complican un poco. Hay que crear un *script* que detecte los objetos que entran en el teletransportador. Solo se habilitará el teletransporte cuando haya un objeto cerca. Antes de empezar a programar el *script*, conviene dar un repaso a distintas funciones que Godot ejecuta por defecto en cualquier *script*:

- **_ready()**: La función que se ejecuta cuando el nodo que contiene el *script* y todos sus nodos hijos han sido cargados en la escena. Es el lugar ideal para definir variables de inicialización que dependan del parámetro de uno de los hijos.
- **_process() y _fixed_process(delta):** Son las funciones que se ejecutan con cada pasada del bucle principal del juego. *Fixed_process* se ejecuta con una frecuencia constante siguiendo el *deltaTime* del juego, mientras que *process* aprovecha el tiempo de descanso o *idle* entre frames.
- **_input(event)**: Una función dedicada exclusivamente a la lectura de la entrada. Para controles sencillos es posible emplear comprobaciones en *fixed_process*, pero *input* da también la posibilidad de hacer comprobaciones adicionales, como por ejemplo si se acaba de pulsar la tecla o si el evento que se recoge es porque se pulsó en un *frame* anterior, empleando *InputEvent.is_echo(event)*.

Para crear el *script*, se selecciona el nodo *Area2D* y se pulsa sobre el botón de creación de *script*. En el menú que aparece, se selecciona la ruta de destino y se le

pone un nombre al nuevo *script*, rellenando el campo *Path*. Después, se pulsa el botón *Create*. La pantalla de edición pasa al modo *Script*, este es su aspecto:

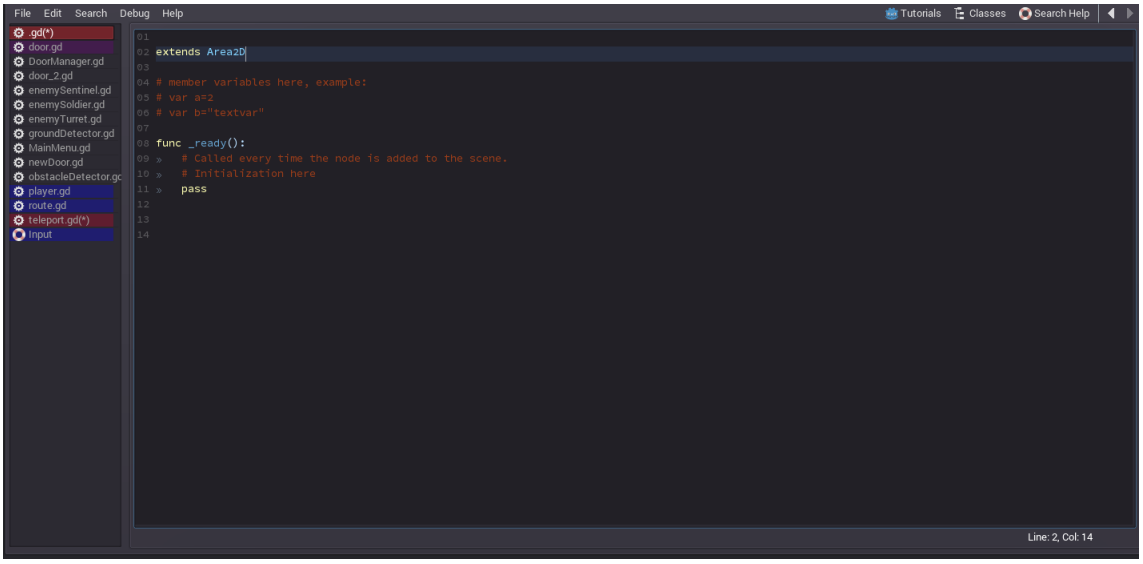


Figura III.16: Aspecto del editor de scripts

Por defecto se crea la función _ready(), que puede borrarse si no se utiliza. Hay que destacar que en la parte superior izquierda aparecen nuevos iconos que dan acceso a tutoriales y la referencia de clase de *GDScript* incluida dentro del propio motor. Esta es una gran ayuda, especialmente al principio, para evitar tener que buscar la información en línea una y otra vez. Además, al hacer que el *script* extienda la clase *Area2D* el editor mostrará sugerencias para autocompletar el código siempre que se usen funciones de la clase *Area2D*.

Ahora, antes de comenzar a programar el *Script*, se va a crear el evento de detección de personajes. Para ello se selecciona el nodo *Area2D*, y se pulsa sobre el botón *Edit Connections*. Aquí aparecerá el menú de edición de señales, que muestra todas las señales que los nodos *Area2D* pueden emitir.

Para este ejemplo, se necesita conectar las señales *body_enter(Object body)* y *body_exit(Object body)*. Se selecciona cualquiera de ellas y se pulsa el botón *Connect*. Luego se selecciona el nodo *Area2D* en la lista que aparece y se pulsa de nuevo el botón *Connect*. Como la opción *make function* está marcada por defecto, el motor cambia de nuevo a la vista de *script* y muestra la función vacía que acaba de crear. Dentro de esta función se introducirá el código a ejecutar cuando un cuerpo externo entre o salga del área, según la señal conectada. Hay que repetir esta serie de pasos para crear la segunda conexión.

Figura III.17: Aspecto del selector de conexiones

Una vez creadas las conexiones ya se puede empezar a escribir el código. Para este ejemplo, se deshabilitará la detección de entrada en la función _ready(). La idea es habilitar la detección de entrada cuando un cuerpo entre en el área, y volver a deshabilitarla cuando este salga. Así, se fuerza a que el cuerpo esté cerca del teletransportador y se ahorran recursos de cálculo. Este es el aspecto de una posible implementación:

```
01
02 extends Area2D
03
04 #Variable que almacena el cuerpo cercano al area
05 var closeBody
06
07 #Deshabilitar la detección de entrada cuando el nodo y sus hijos han sido cargados
08 func _ready():
09
10 »   pass
11
12 #Proceso de los eventos de entrada
13 func _input(event):
14 »   #Si el jugador pulsa la acción "teleport", coge el cuerpo cercano y lo manda a 100 pixels de su altura actual
15 »   if event.is_action("teleport") and not event.is_echo():
16 »   »   var newPosition = closeBody.get_pos() + Vector2(0,-100)
17 »   »   closedBody.set_pos(newPosition)
18
19 #Cuando un cuerpo externo entra en el área del teletransportador
20 func _on_Area2D_body_enter( body ):
21 »   #Si no hay otro cuerpo que haya entrado antes
22 »   if closedBody != null:
23 »   »   #Actualizar el cuerpo más cercano, buscando el cuerpo entre los hijos directos del nodo raíz
24 »   »   closedBody = get_tree().get_root().get_node(body.get_name())
25
26 #Cuando el cuerpo sale del área del teletransportador
27 func _on_Area2D_body_exit( body ):
28 »   #Si es el objetivo que se había fijado...
29 »   if body.get_name() == closeBody.get_name():
30 »   »   #Se elimina la referencia al cuerpo cercano
31 »   »   closeBody = null
32
```

Figura III.18: Implementación de un teletransportador básico

En este ejemplo se manda el objeto a una altura de 100 píxels por encima de su posición antes del teletransporte. Sin embargo, sería posible modificarlo de otras

maneras, por ejemplo: cambiando la forma en la que lo afecta la gravedad, su escala, transportándolo a la posición de otro nodo, etc.

Crear y reproducir animaciones

Las animaciones ayudan a dar vida a los personajes y escenarios, y son uno de los elementos más básicos de un videojuego. Godot cuenta con un editor de animaciones que funciona empleando *fotogramas clave*. Los fotogramas clave de una animación marcan los cambios en las propiedades del objeto animado. Normalmente se hacen animaciones de *sprites*, cambiando el cuadro seleccionado dentro de la hoja de animaciones. Sin embargo, Godot también permite animar otros parámetros como la escala de los objetos, su posición, la opacidad, etc. Para hacerlo es necesario emplear los nodos *AnimationPlayer.* Estos nodos permiten crear, editar y reproducir animaciones.

Figura III.19: Aspecto del editor de animaciones

El editor de animaciones aparece en la parte inferior de la pantalla. En el editor hay varios botones para crear nuevas animaciones, reproducirlas hacia delante y atrás, etc. También en la parte superior del editor se pueden ver los botones *AutoStart*, para establecer la animación por defecto, *BlendTimes*, para elegir el tiempo de mezcla con otras animaciones o la animación que se reproduce a continuación de forma automática, y un botón para copiar o pegar animaciones.

La zona central muestra las propiedades animadas, en este caso el *frame* del nodo *sprite*. Cada punto azul es un fotograma clave. Pulsando sobre el botón de la línea discontinua que aparece a la derecha de la propiedad se puede establecer un salto continuo, con el que Godot rellena de forma automática los distintos *frames* de la animación entre dos fotogramas clave, o bien dejarlo como está para crear la animación de forma manual fotograma a fotograma.

Por último, la barra inferior tiene las opciones para modificar las pistas de la animación. Se puede modificar el nivel de *zoom* de la pista, la longitud en segundos, el tiempo entre cada paso de la animación, y también se puede cambiar el orden de las pistas, borrarlas o añadir pistas nuevas. Es posible añadir una pista de cualquier

propiedad que tenga dibujada una llave en la parte derecha del inspector cuando el editor de animaciones está abierto, tal y como se muestra en la siguiente captura:

Figura III.20: Detalle del icono de fotograma clave

Es habitual que los nodos *AnimationPlayer* cuelguen de un nodo *Sprite* o *AnimatedSprite*. Dependiendo del nodo donde se haya guardado la animación, el proceso para obtener los fotogramas clave es distinto. Hay un tutorial completo explicando las diferencias en detalle en la sección final de tutoriales.

Música y Efectos de Sonido

Para añadir música al nivel se pueden utilizar nodos de tipo *SamplePlayer2D* o *StreamPlayer*. Los primeros consumen menos recursos y es habitual emplearlos para reproducir efectos de sonido cortos, que pueden comprimirse sin que la posible pérdida de calidad sea muy perceptible. Sin embargo, para utilizar muestras o *Samples* primero hay que importarlas al proyecto por medio del menú *Import/AudioSample*. Las muestras se almacenan en bibliotecas de sonido que pueden asociarse a distintos nodos. El formato compatible con este tipo de nodos es el wav. Respecto a los nodos *StreamPlayer*, estos pueden emplearse directamente con canciones en distintos formatos como el ogg.

Cómo crear módulos adicionales

Godot integra una gran cantidad de funciones, pero es una tecnología joven y a veces puede surgir la necesidad de ampliar sus capacidades. Para ello existe la posibilidad de crear módulos adicionales. Para ello se debe descargar el código fuente de

Godot[23]. Una vez descargado, hay que entrar en la carpeta *modules*. Dentro de esta carpeta, se creará una nueva carpeta con el nombre del nuevo módulo. En esta nueva carpeta se introducirán todos los archivos de cabeceras y código, escritos en C++, que contengan el funcionamiento del nuevo módulo. Si dentro de los archivos de código se incluye una referencia a la clase *Node*, se crearán nuevos nodos a los que se podrá acceder desde el propio editor gráfico de Godot.

Una vez desarrollado el código del módulo, hay que añadir una serie de métodos adicionales para enlazar los objetos y sus funciones a la base de datos de Godot. También es necesario añadir un archivo de configuración escrito en *Python*, así como un archivo adicional que proporcionará instrucciones al compilador. El código fuente de *Godot* incluye algunos ejemplos. Además, en la documentación se puede consultar un ejemplo[24] sencillo y muy ilustrativo de todo el proceso.

Crear el módulo es tan solo la primera parte del proceso. También es necesario recompilar el motor con todo el código de los nuevos módulos que se quieran añadir. Los requisitos y herramientas necesarias para recompilar el motor dependen de la plataforma en la que vayamos a hacerlo, así como de la plataforma de destino. Se recomienda leer la documentación acerca del proceso de compilación de Godot[25] para obtener información detallada sobre cualquier plataforma.

Tutoriales y páginas de consulta

A continuación, se incluyen una serie de tutoriales y páginas útiles para principiantes, que amplían y complementan el contenido de esta guía.

Documentación oficial de Godot: Incluye una sección de tutoriales y la especificación del lenguaje GDScript entre otros recursos: http://docs.godotengine.org/en/latest/#

Serie de tutoriales dedicada a Godot del sitio web *GameFromScratch*, que contiene abundante material de vídeo y texto: http://www.gamefromscratch.com/page/Godot-Game-Engine-tutorial-series.aspx

Pequeño tutorial sobre la creación y el uso de animaciones, del usuario *adolson* de YouTube: https://www.youtube.com/watch?v=X3flgK75Oz0

[23] Código fuente disponible en: https://github.com/godotengine/godot
[24] Ejemplo disponible en: http://docs.godotengine.org/en/latest/reference/custom_modules_in_c++.html
[25] Documentación disponible en el siguiente enlace: http://docs.godotengine.org/en/latest/reference/_compiling.html

Página de preguntas y respuestas de Godot, donde se puede pedir ayuda a otros usuarios de la comunidad para resolver dudas: https://godotengine.org/qa/

Anexo 4: GDD

Steel Soldier

Game Design Document

David Morales Cortés

Información general

Título: Steel Soldier

Autores: David Morales Cortés, Mauro Beltrán, Carlos Mercé Vila, Germán Reina Carmona

Año: 2016

Género: Shooter/Plataformas 2D

Plataformas de destino: PC, dispositivos Android

Público objetivo

Edad: Entre 20 y 35 años

Sexo: Masculino

Tiempo de juego: 8 horas/semana

Aficiones: cine y literatura de ciencia-ficción, acción, videojuegos clásicos

Juegos similares: Contra, Metal Gear, Mark of the Ninja.

Descripción

El juego toma la perspectiva clásica de los juegos de acción de avance lateral y plataformas como Contra o MegaMan añadiendo además componentes de infiltración que complementan las fases de acción y disparos. A lo largo del juego el jugador se encontrará con niveles de infiltración en los que no debe ser detectado por el enemigo, por lo que será necesario buscar coberturas y escondites. Además, tendrá la posibilidad de *hackear* terminales repartidos por el nivel que le permitirán manipular los sistemas de seguridad enemiga, así como tomar control de algunos enemigos de naturaleza mecánica. En las fases de acción se ponen a prueba los reflejos del jugador, que deberá enfrentarse a numerosos enemigos sin perder toda su salud, cumpliendo objetivos diversos en función de la fase.

Historia

El juego pone al jugador en el papel de un soldado anónimo que tiene como objetivo infiltrarse en un complejo militar para rescatar al Dr. Joseph Dorman, que está siendo forzado a desarrollar un exoesqueleto que permitirá a la organización fabricar un ejército de soldados robots. Durante el transcurso del juego, el doctor revelará al

protagonista que es en realidad el primer prototipo creado para este fin, lo cual da sentido al título del juego. En la fase final, el protagonista debe luchar contra un soldado de acero de última generación para poner fin al plan de la organización militar.

Presentación

El juego contará con una ambientación futurística poniendo énfasis en diseños oscuros y mecanizados, tanto para los elementos del escenario como para objetos, enemigos, etc. Los entornos bidimensionales contarán con zonas de contraste entre luces y sombras para que el jugador sea capaz de pasar inadvertido frente a algunos enemigos.

Habrá pequeños diálogos entre el personaje del jugador y el Dr. Dorman al pasar de una escena a otra que servirán para profundizar en el mundo del juego y contar la historia. Estos diálogos se presentarán con animaciones sencillas en un entorno estático, y en ellos podremos ver al protagonista y el resto de personajes dibujados con más grado de detalle.

Respecto al apartado musical, se ha optado por escoger sonidos electrónicos que concuerdan con la atmósfera industrial y futurista del complejo militar. La música resulta intensa y trepidante en las fases de disparos. En las fases de infiltración, adquiere un carácter más sutil y ambiental, que ayuda al jugador a sentirse más inmerso en el juego.

Interfaz de usuario

En el menú inicial, el jugador puede optar por empezar una nueva partida, cargar una partida anterior o seleccionar uno de los niveles del juego, que van desbloqueándose conforme son completados. También dispone de un menú de opciones que le permite seleccionar el idioma del juego entre inglés y español, activar o desactivar la música y los efectos de sonido y ajustar los controles del juego a su gusto. Por último, el menú inicial cuenta con un botón para salir del juego.

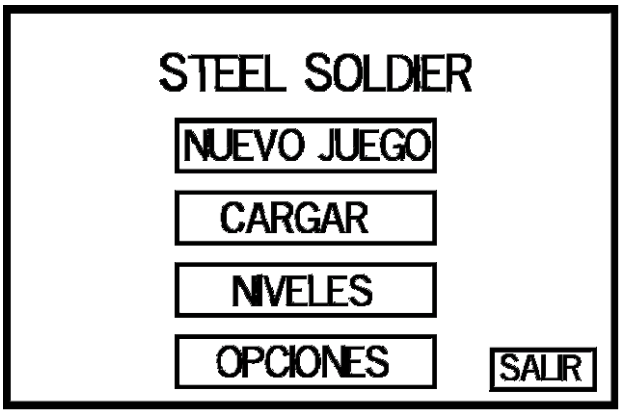


Diagrama del menú inicial

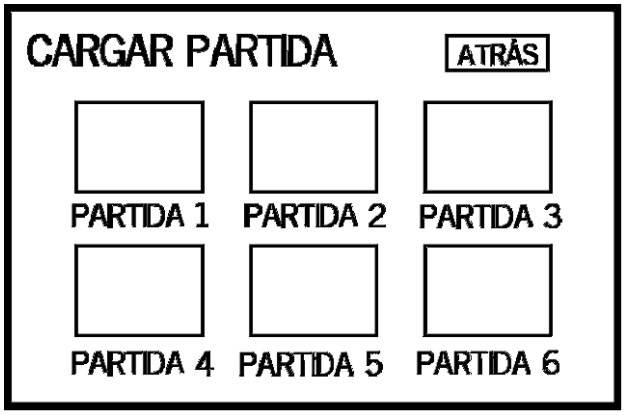


Diagrama del menú de carga

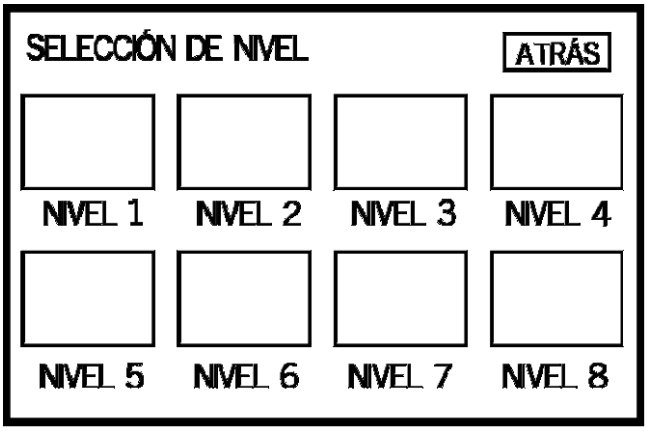


Diagrama del menú de selección de niveles

Diagrama del menú de opciones

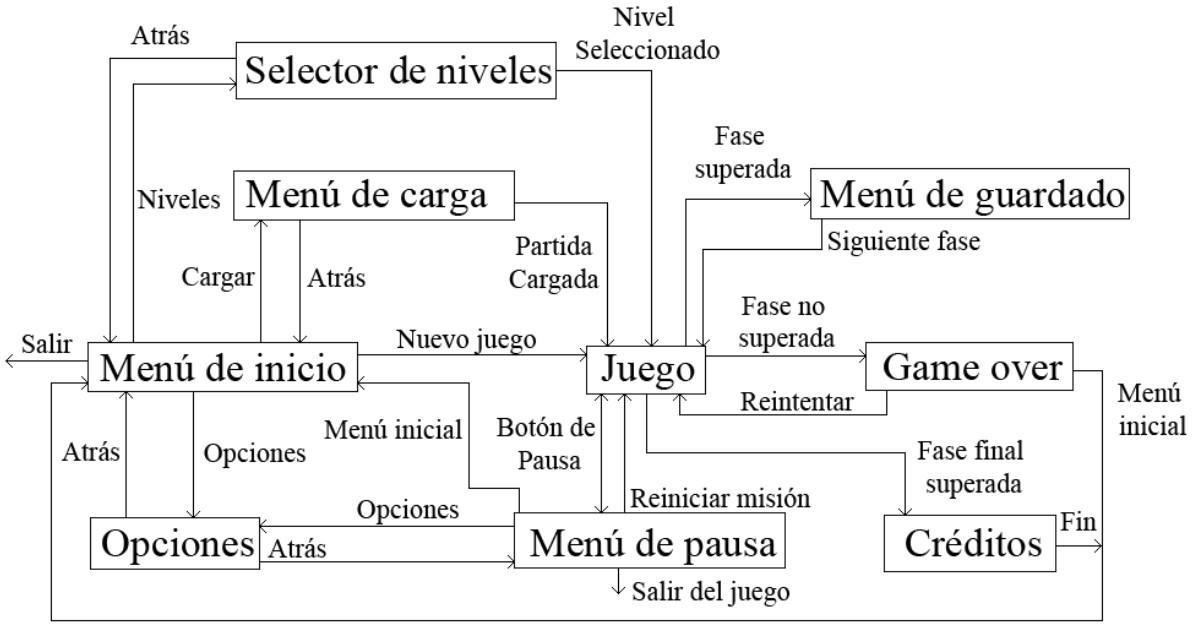
Durante el propio juego, la interfaz muestra al jugador cuanta salud le queda mediante una barra en la esquina superior izquierda de la pantalla que se va vaciando conforme recibe daños. La esquina superior derecha puede mostrar diversos mensajes informativos cuando el jugador se acerca a un objeto con el que puede interactuar, ya sea una cobertura para esconderse, un terminal que *hackear* o una puerta que abrir.

También se da la opción de pulsar una tecla que abre el menú de pausa, en el que podemos volver a comenzar la fase, acceder al menú de opciones descrito en el párrafo anterior, salir al menú principal o salir del juego. Al final de cada fase, se ofrece al jugador la oportunidad de guardar su progreso, dando a escoger si así lo eligiese, entre sobrescribir una partida guardada o crear un nuevo archivo de guardado. La interfaz es similar al menú de carga. No se contempla una opción para borrar partidas ya que estas pueden sobrescribirse, lo cual tiene el mismo efecto.

Diagrama del menú de pausa

La interfaz por tanto resulta rápida y fácil de usar ya que no hay un gran número de menús y estos presentan opciones claras y sencillas, las cuales resultan comprensibles para todo tipo de usuarios estén o no acostumbrados a jugar a

videojuegos. La navegación por las interfaces se realizará mediante el ratón en plataformas de escritorio y tocando los botones en dispositivos Android. El diagrama de flujo del juego, incluyendo los menús descritos en este apartado, sería el siguiente:



Habilidades del personaje protagonista

Todas las interfaces mostradas en la sección anterior serán navegables con el cursor del ratón. Durante las fases de juego, los controles son los siguientes:

Acción	Descripción	Tecla por defecto
Movimiento	Permite desplazarse por el nivel y elegir la trayectoria de los saltos.	Flechas izquierda y derecha
Salto	Permite al jugador sortear obstáculos y alcanzar plataformas elevadas	Flecha arriba
Agacharse	Permite al jugador sortear ataques enemigos. Es posible moverse y disparar estando agachado.	Flecha abajo
Disparo	Utilizar el arma que el jugador tiene equipada.	Z
Hackear	Acción disponible cuando el jugador está cerca de un terminal informático. Efectos variados.	X
Esconderse	Acción disponible cuando el jugador está cerca de un punto en el que puede ocultarse. Permite evadir a los enemigos.	X
Abrir puertas	Acción disponible cuando el jugador está cerca de una puerta. Algunas de estas puertas requerirán pases especiales.	X
Abrir/Cerrar menú de Pausa	Da acceso al menú de pausa durante la partida.	Escape

Hay que comentar algunos aspectos de la jugabilidad que complementan la información de esta tabla:

1. La munición del arma es infinita, no es necesario recargar.

2. Al recibir daño, el jugador parpadeará durante unos segundos en los que será invencible.

3. Los objetos que el jugador pueda encontrar repartidos por el escenario se cogen de forma automática al tocarlos.

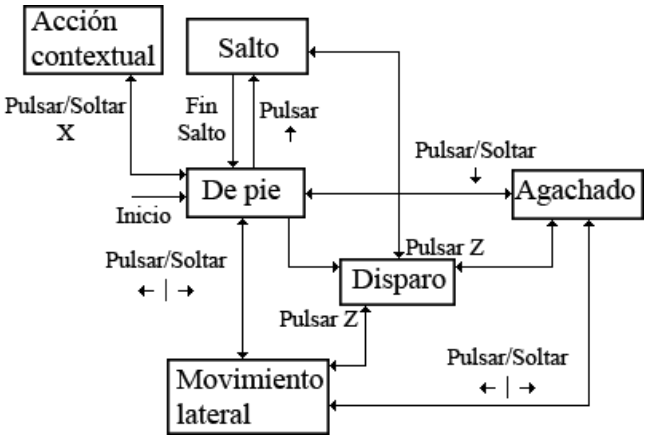
4. La tecla X varía entre las tres funciones dependiendo de los objetos cercanos al jugador. A partir de ahora esta tecla será identificada como la tecla de acción.

5. Como se mencionó en el apartado de interfaz, el jugador recibirá información de qué puede hacer con la tecla de acción por medio de un mensaje en la esquina superior derecha.

6. Hay que recordar que los controles son personalizables en el menú de opciones.

7. Los controles mostrados corresponden a la versión de PC. En dispositivos Android se utilizarán botones semitransparentes a los que se asignará las mismas funciones.

A continuación, se muestra la máquina de estados correspondiente a los controles del jugador:



Progresión y curva de dificultad

El juego comienza con un primer nivel a modo de tutorial, con pocos enemigos y muchas coberturas, para que el jugador se familiarice con los controles. El objetivo será encontrar una de las tres armas disponibles al final de la fase. Cada arma tiene un comportamiento distinto, siendo el juego más sencillo o complicado dependiendo del arma seleccionada. Esto anima al jugador a volver a completar el juego con distintas armas, aumentando el valor de rejugabilidad.

Tenemos a continuación las distintas fases del juego, con una breve descripción de los objetivos a cumplir en cada una de ellas:

- Fase 1 (infiltración): Encontrar un arma con la que defenderse.
- Fase 2 (infiltración): Llegar hasta la celda del Dr. Dorman, el jugador debe encontrar la llave escondida en la fase para poder liberar al doctor.
- Fase 3 (acción): Escoltar al Dr. Dorman hasta la salida del complejo.
- Fase 4 (infiltración): Encontrar una tarjeta de acceso para entrar al laboratorio de investigación.
- Fase 5 (infiltración): *Hackear* todos los terminales conectados a la base de datos para borrar los registros de la investigación. Si los enemigos nos detectan activarán la cámara de gas.
- Fase 6 (acción): Llegar a la meta antes de que acabe el tiempo.
- Fase 7 (acción): Eliminar a todos los enemigos para acceder al terminal central y hacer estallar el complejo.
- Fase 8 (acción): Eliminar al jefe final.

La dificultad va en aumento añadiendo objetivos adicionales, así como incorporando nuevos tipos de obstáculos y enemigos.

Enemigos

El juego cuenta con dos tipos de enemigos distintos: Humanos y máquinas. Los humanos son vulnerables a un ataque directo del jugador, y su resistencia a los disparos varía en función de su clase. Las máquinas, por otra parte, harán sonar las alarmas del complejo si tratamos de destruirlas, aunque es posible *hackear* algunas de ellas mediante sus terminales de control.

Humanos

Por una parte, para que los enemigos humanos detecten al jugador, este debe encontrarse en una zona iluminada, ya que de lo contrario no son capaces de verlo. Además, los enemigos humanos pueden sufrir daños si el jugador les dispara. Cuando detectan al protagonista, hacen saltar las alarmas de todo el complejo, provocando la aparición de más enemigos humanos y el estado de alerta en todos los enemigos mecánicos. Tras esto, tratan de perseguir al jugador y dispararle para que pierda la partida. En las fases de acción, su estado natural es el de alerta.

La siguiente tabla muestra una relación de todos los tipos de enemigos humanos, su comportamiento y la fase en la que aparecen por primera vez:

Nombre	Descripción	Resistencia	Primera aparición
Soldado raso	Enemigo básico, se enfrenta directamente al jugador sin cubrirse. En estado normal patrulla una zona fija. En estado de alerta, cuando ve al jugador, lo persigue. Sus disparos quitan un 5% de la barra de vida	Resiste un disparo con arma de fuego. Puede aturdirse con pistola eléctrica	Fase 1
Soldado blindado	Soldado con armadura pesada. Capaz de emplear coberturas y agacharse durante el combate. Es incapaz de saltar, lo cual puede ser aprovechado por el jugador para huir. Sus disparos quitan un 15% de la barra de vida	Resiste cuatro disparos con arma de fuego. Puede aturdirse con pistola eléctrica	Fase 3
Soldado de élite	Combina la movilidad de los soldados rasos y la habilidad de los soldados blindados. Equipados con máscaras de gas, activarán la cámara de gas si descubren al jugador en una misión de infiltración. Sus disparos quitan un 10% de la barra de vida	Resiste siete disparos con arma de fuego. No puede aturdirse con pistola eléctrica.	Fase 5
Experto en seguridad	Recorre la zona de patrulla inspeccionando las coberturas. En estado de alerta, huye del jugador tras dar la voz de alarma. El jugador puede aturdirlos permanentemente, pero si otro enemigo ve el cuerpo aturdido dará la voz de alarma	Resiste dos disparos con arma de fuego. Puede aturdirse permanentemente con la pistola eléctrica	Fase 2
Investigador	No es un enemigo hostil, pero dará la voz de alarma al detectar al jugador y tratará de huir. El jugador puede aturdirlos permanentemente, pero si otro enemigo ve el cuerpo aturdido dará la voz de alarma	Resiste un disparo con arma de fuego. Puede aturdirse permanentemente con la pistola eléctrica	Fase 5
Ingeniero	Capaz de reparar los terminales de seguridad y los enemigos mecánicos. En estado de alerta se dirige al terminal más cercano para atacar al jugador controlando un enemigo mecánico	Resiste dos disparos con arma de fuego. Puede aturdirse con la pistola eléctrica	Fase 4

Mecánicos

Los enemigos mecánicos cuentan con los estados “Activo”, “Inactivo” y “Alerta”. Todos los enemigos, a excepción del “centinela láser”, son invulnerables a las armas convencionales. Completar el juego recompensa al jugador con la pistola de datos, capaz de *hackear* a estos enemigos. Una vez *hackeados,* estos enemigos no serán agresivos hasta que un ingeniero los repare. A continuación, se incluye una tabla que describe el comportamiento de estos enemigos:

Nombre	Descripción	Resistencia	Primera aparición
Centinela	Robot de seguridad equipado con una cámara. Al detectar al jugador propaga el estado de alerta al resto de enemigos. No es agresivo	Invencible. *Hackeable* por terminal o pistola de datos	Fase 1
Torreta	Comienza en estado inactivo. En estado de alerta se pone en funcionamiento y dispara al jugador. Cuando se *hackea* puede emplearse contra enemigos, aunque esto hará saltar las alarmas. Sus disparos quitan un 15% de la barra de vida	Invencible. Hackeable por terminal o pistola de datos	Fase 1
Robot de limpieza	Recorre los conductos de ventilación. Puede descubrir al jugador cuando este está escondido si entra en contacto directo.	Una descarga de pistola eléctrica lo desactiva.	Fase 4
Centinela láser	Variante del centinela que persigue al jugador y lo daña con un rayo láser. No propaga el estado de alerta, pero su láser quita un 25% de la barra de vida	Resiste cuatro disparos con armas de fuego.	Fase 5
Soldado de acero	Enemigo final. Un soldado de élite ágil con las mismas habilidades que el jugador. Antes de atacarlo, el jugador debe desactivar los terminales que protegen su sistema para hacerlo vulnerable durante unos instantes. Cada disparo quita un 15% de la barra de vida	Resiste veinte disparos de la pistola de datos.	Fase 8

Objetos

Repartidos por las distintas fases, el jugador podrá encontrar una serie de objetos que le proporcionarán ayuda para cumplir sus objetivos. Los enemigos no son capaces de interactuar con ninguno de los siguientes objetos:

Categoría	Nombre	Descripción	Aparece en
Arma	Pistola Eléctrica	Arma de corto alcance, silenciosa. Aturde momentáneamente a todo tipo de enemigos. Fuerza a luchar cuerpo a cuerpo en las fases de acción	Fase 1
Arma	Pistola	Arma de alcance medio, que genera ruido en la zona donde se dispara. Útil tanto en las fases de acción como en las de infiltración	Fase 1
Arma	Ametralladora	Alma de gran alcance. Genera un gran ruido que activa automáticamente el estado de alerta en las fases de infiltración	Fase 1
Arma	Pistola de datos	Arma cargada con un virus, capaz de dañar al soldado de acero e inutilizar por completo a los enemigos mecánicos	Obtenida al inicio de la fase 8
Consumible	Pack de salud pequeño	Restaura un 20% de la barra de salud del jugador	Todas las fases
Consumible	Pack de salud mediano	Restaura un 40% de la barra de salud del jugador	Todas las fases
Consumible	Pack de salud grande	Restaura un 60% de la barra de salud del jugador	Todas las fases
Objeto Clave	Llave de la celda	Da acceso a la celda del Dr. Dorman. Se utiliza automáticamente al llegar a la celda.	Fase 2
Objeto Clave	Tarjeta de acceso	Da acceso a los laboratorios de investigación. Se utiliza automáticamente al llegar a la entrada.	Fase 4

Elementos interactivos

En los escenarios existen diversos elementos interactivos de comportamiento muy similar. Estos cuentan con un área de detección de personajes para poder detectar cuando el jugador entra en contacto con ellos. Son los encargados de activar la tecla de acción e indicar en la interfaz que hay una interacción disponible. Dichos elementos son:

- **Puertas**, que interconectan las diferentes estancias que componen un nivel. Pueden estar abiertas o cerradas dependiendo del estado de alerta de los enemigos o terminales cercanos.

- **Escondites**, que permiten al jugador refugiarse para que los enemigos no puedan detectarlo.
- **Terminales**, que cumplen varias funciones como desbloquear puertas, apagar las luces de una sala, manipular enemigos mecánicos, etc.

Terminales

Los terminales son el elemento interactivo más complejo. Controlan algunos enemigos mecánicos y algunos sistemas de seguridad como barreras láser y puertas de seguridad. Cuando el jugador mantiene pulsada la tecla de acción cerca de un terminal, una barra de progreso aparece indicando el porcentaje del *hackeo*, que va llenándose mientras el jugador mantenga pulsada la tecla de acción. Al llegar al 100%, el terminal es *hackeado* y el jugador inutiliza temporalmente los sistemas asociados al terminal. Solo los ingenieros enemigos son capaces de arreglar los terminales.

Tipos de plataformas

Para ofrecer una experiencia más interesante desde el aspecto de juego de plataformas, es necesario crear distintos tipos de terreno que se comporten de forma ligeramente distinta. Podemos distinguir tres tipos distintos:

- **Suelo y paredes.** Otorgan el diseño básico del nivel y delimitan su espacio. Son elementos sólidos y estáticos que el jugador no puede atravesar.
- **Semi-sólidas.** Tienen este nombre porque solo detectan las colisiones cuando estas se producen con un objeto que está situado por encima de la plataforma, no por debajo. Se utilizan para otorgar a los niveles de más libertad de movimiento, verticalidad, y variedad visual, ya que su aspecto es distinto al de las paredes y el suelo. Son un recurso recurrente en todo tipo de juegos de plataformas.
- **Móviles.** Como su nombre indica, estas plataformas no son estáticas, sino que cuentan con un desplazamiento a través de una ruta predefinida. Sirven para añadir mayor interés y dinamismo a los niveles del juego, añadiendo importancia no solo a la trayectoria del salto sino también a cuándo se produce el mismo. Se requiere de un script adicional para controlar la trayectoria de estas plataformas.