



Escuela de Ingenierías Industrial, Informática y Aeroespacial

GRADO EN INGENIERÍA INFORMÁTICA

Trabajo de Fin de Grado

**MIC OUT OF CONTROL: DESARROLLO DE UN VIDEOJUEGO EN
UNITY 3D UTILIZANDO EL EDIFICIO MIC COMO ENTORNO
JUGABLE**

**MIC OUT OF CONTROL: DEVELOPMENT OF A VIDEOGAME IN
UNITY3D USING THE MIC BUILDING AS A PLAYABLE
ENVIRONMENT.**

Autor: Iván Castro Martínez

Tutor: Fernando Jorge Fraile Fernández

(Febrero, 2021)

**UNIVERSIDAD DE LEÓN
Escuela de Ingenierías Industrial, Informática y
Aeroespacial**

**GRADO EN INGENIERÍA INFORMÁTICA
Trabajo de Fin de Grado**

ALUMNO: Iván Castro Martínez

TUTOR: Fernando Jorge Fraile Fernández

TÍTULO: *MIC out of control*: Desarrollo de un videojuego en *Unity3D* utilizando el edificio MIC como entorno jugable.

TITLE: *MIC out of control*: Development of a videogame in *Unity3D* using the MIC building as a playable environment.

CONVOCATORIA: Febrero, 2021

RESUMEN:

Este proyecto consiste en el desarrollo de un videojuego estilo *Survival Horror* en el edificio MIC. El argumento básico del juego es la búsqueda de la cura contra el Covid19. Cuenta con sistema de misiones en modo historia y modo desafío.

Para su implementación se utiliza el motor gráfico *Unity3D* y el control de versiones *Perforce* para salvar los cambios y conservar el historial de modificaciones durante la implementación. Desarrollado en la Escuela de Ingenierías de la Universidad de León. El videojuego se crea desde cero y es implementado en el lenguaje de programación C#.

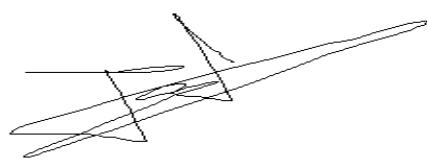
ABSTRACT:

This Project consists of the development of a *Survival Horror* style videogame in the MIC building. The basic argument of the game is the search for the cure against Covid19. It has a mission system in story mode and challenge mode.

For its implementation, the *Unity3D* graphic engine and *Perforce* version control are used to save the changes and conserve the history of changes in development. Developed at the School of Engineering of Leon. The videogame is created from scratch and is implemented in the C# programming language.

Palabras clave: unity3d, C#, survival horror games, control de versiones

Firma del alumno:



VºBº Tutor/es:

Fernando Jorge Fraile Fernández

Índice de contenidos

1.	Introducción	1
1.1	MOTIVACIÓN.....	1
1.2	OBJETIVOS	2
1.3	ESTRUCTURA DE LA OBRA	2
1.4	ESQUEMA DEL TRABAJO	3
1.5	METODOLOGÍA.....	4
2.	Estado del arte	5
2.1	SITUACIÓN DEL GÉNERO SURVIVAL.....	5
2.1.1	Survival Horror	6
2.1.2	Wilderness Survival.....	7
2.1.3	Survival Híbrido.....	7
2.2	ESTUDIO DE LA COMPETENCIA.....	8
2.3	MOTORES GRÁFICOS	12
2.3.1	Unreal Engine	12
2.3.2	Unity.....	12
2.3.3	RPGMaker	12
2.3.4	GameMaker Studio	13
2.3.5	Godot	13
2.3.6	CryEngine	13
2.4	SELECCIÓN DE LAS HERRAMIENTAS	14
2.4.1	Motor gráfico	14
2.4.2	Control de versiones	18
2.4.3	Plugins y librerías.....	19
3.	Diseño.....	21
3.1	DISEÑO CONCEPTUAL	21
3.2	DISEÑO ESTRUCTURAL	21
3.2.1	Diseño estructural de cada <i>Modo de juego</i>	22
3.3	DISEÑO DE LA INTELIGENCIA ARTIFICIAL (IA)	23
3.3.1	Que es una inteligencia artificial.....	23

3.3.2 Cómo usamos la inteligencia artificial	24
3.4 DISEÑO DE PERSONAJES ELEGIBLES	25
3.5 DISEÑO DEL ARMA	26
4. Planificación	27
4.1 PARTE 1 – Configuración del servidor de <i>Perforce</i> , aprendizaje y búsqueda de recursos	27
4.2 PARTE 2 – Programación del videojuego	28
4.3 PARTE 3 – Modelado del nivel.....	29
4.4 PARTE 4 – Testeo y redacción de la memoria.....	29
5. Implementación.....	31
5.1 CONFIGURACIÓN DEL CONTROL DE VERSIONES PERFORCE	31
5.1.1 Funcionamiento de P4V Client.....	32
5.1.2 Interfaz de <i>Perforce</i> en <i>Unity3D</i>	33
5.1.3 Porqué usar <i>Perforce</i>	34
5.2 EL PERSONAJE PRINCIPAL	35
5.2.1 Movimiento del personaje	35
5.2.2 Parámetros del personaje	36
5.3 INTELIGENCIA ARTIFICIAL DE LOS ENEMIGOS	37
5.3.1 Inteligencia artificial de los enemigos	38
5.3.2 Daño y transformación de los enemigos	39
5.4 EL SISTEMA DE MISIONES Y DIÁLOGOS	39
5.4.1 Detección de misiones y desarrollo de diálogos	40
5.4.2 Elaboración y distribución de las misiones.....	42
5.5 EL FUNCIONAMIENTO DEL ARMA	46
5.6 FUNCIONALIDADES EXTRA	47
5.6.1 El cronómetro.....	47
5.6.2 Guardado y carga de personajes	49
5.6.3 Colliders para recoger objetos	50
5.6.4 Vacunas e <i>Incrementos de dosis</i> del <i>Modo Desafío</i>	51
6. Diseño conceptual del juego	53
6.1 DISEÑO DE LOS CONTROLES	53
6.2 DISEÑO DE LOS PERSONAJES	54
6.2.1 Creación del personaje principal con técnicas de scanner 3D.....	54
6.2.2 Personajes disponibles para jugar	56

6.2.3 <i>Mutantes</i> (profesores y estudiantes contagiados).....	57
6.2.4 <i>Quinto Paciente</i> (enemigo principal del videojuego)	57
6.2.5 <i>Científico de Guardia, Científico Principal</i> y el <i>Director</i>	58
6.2.6 Pacientes y Cuarto Paciente	59
6.3 DISEÑO DEL ARMA Y FUNCIONALIDADES EXTRA.....	59
6.4 DISEÑO DEL MENÚ	61
6.4.1 Menú Principal	61
6.4.2 Menús dentro del juego.....	63
6.4.3 Menús del Modo Historia y Modo Desafío	66
6.5 DISEÑO DEL MODO DESAFÍO.....	66
6.6 DISEÑO DEL MODO HISTORIA.....	68
6.6.1 Diseño de misiones del modo historia.....	68
6.7 DISEÑO DE LA INTERFAZ.....	71
7. Conclusiones	72
8. Trabajos futuros	74
9. Agradecimientos	75
10. Referencias.....	76
Anexo 1: Documento de diseño.....	78
A1.1 Información general.....	79
A1.1.1 CONCEPTO GENERAL.....	79
A1.1.2 OBJETIVO	79
A1.1.3 GÉNERO	79
A1.1.4 HISTORIA O SINOPSIS	79
A1.1.4.1 Modo Desafío	79
A1.1.4.2. Modo Historia.....	80
A1.1.5 ESTILO VISUAL	87
A1.1.6 MOTOR Y EDITOR.....	87
A1.1.7 NÚCLEO DEL GAMEPLAY	87
A1.1.8 PÚBLICO OBJETIVO.....	87
A1.1.9 CARACTERÍSTICAS DEL JUEGO	88
A1.1.9.1 Ambientación	88
A1.1.10 ALCANCE DEL PROYECTO	88
A1.1.10.1 Ubicaciones del juego.....	88

A1.1.10.2 Descripción de los enemigos	88
A1.1.10.3 Descripción de las armas o defensas	88
A1.1.10.4 Descripción del nivel	88
Anexo 2: Recursos Utilizados.....	89

Índice de figuras

Figura 2.1. Cronología aparición de videojuegos más importantes del género de Supervivencia. [4]	5
Figura 2.2 Juego <i>Resident Evil</i> . (Fuente: https://www.residentevil.com/)	6
Figura 2.3. Juego <i>Unreal World</i> . (Fuente: https://store.steampowered.com/).....	7
Figura 2.4. Juego <i>Minecraft</i> .(Fuente: https://www.minecraft.net)	8
Figura 2.5. Juego <i>Fortnite</i> . (Fuente: https://www.epicgames.com/fortnite/)	8
Figura 2.6. Juego <i>Outlast</i> . (Fuente: https://www.eneba.com/latam/steam-outlast-steam-key-global).....	9
Figura 2.7. Juego <i>Resident Evil VII</i> (http://umlconnector.com/2017/02/resident-evil-7-puts-survival-back-in-horror-genre/)	10
Figura 2.8. Juego <i>Silent Hill 4</i> (Fuente: https://www.ecranlarge.com/jeux-video/1119575-silent-hill-4-the-room).....	11
Figura 2.9. Interfaz de <i>Unity</i>	16
Figura 3.1. Diagrama del <i>GameManager</i> que muestra que datos guarda entre escenas. ..	23
Figura 3.2. Tipos de <i>inteligencia artificial</i> . (Fuente: businessgoon.com/inteligencia-artificial-que-es/)	24
Figura 3.3. Máquina de estados de los enemigos.	24
Figura 3.4. <i>Blend Tree Movement</i> del movimiento básico de los personajes seleccionables.....	25
Figura 3.5. <i>Animator Controller</i> de las acciones de los personajes seleccionables.....	26
Figura 5.1. Ventana de <i>login</i> de conexión remota de <i>P4V</i>	32
Figura 5.2. Configuración de la ruta del <i>Workspace</i> y directorio local del proyecto en <i>P4V</i>	32
Figura 5.3. Interfaz de enlace de <i>Perforce</i> en <i>Unity3D</i>	33
Figura 5.4. <i>Nomenclatura</i> y <i>Version Control</i> de la interfaz de <i>Perforce</i> en <i>Unity3D</i>	33
Figura 5.5. Uso de un <i>singleton</i> en la clase <i>MainController.cs</i>	35
Figura 5.6. Traza de código de la clase <i>MainController.cs</i>	36
Figura 5.7. Traza de código de la clase <i>Health_and_Damage().cs</i>	37
Figura 5.8. Traza de código de la clase <i>GameOver().cs</i>	37
Figura 5.9. Traza de código de la clase <i>IAEnemies().cs</i>	38
Figura 5.10. Funciones <i>TakeDose()</i> y <i>personSaved()</i> de la clase <i>health_and_damageIA.cs</i>	39

Figura 5.11. Traza de código de la clase <i>DialogueTrigger.cs</i>	40
Figura 5.12. Función <i>StartDialogue()</i> de la clase <i>DialogueManager().cs</i>	41
Figura 5.13. Función <i>DisplayNextSentence()</i> de la clase <i>DialogueManager().cs</i>	41
Figura 5.14. Foto de algunos de los diálogos del Científico de Guardia.....	42
Figura 5.15. Función <i>EndDialogue()</i> de la clase <i>DialogueManager().cs</i>	42
Figura 5.16. Traza de la clase <i>QuestGiver().cs</i>	43
Figura 5.17. Método <i>AcceptQuest()</i> de la clase <i>QuestGiver.cs</i>	44
Figura 5.18. Método <i>loadQuestBox()</i> de la clase <i>GameManager.cs</i>	45
Figura 5.19. Método <i>loadElements()</i> de la clase <i>GameManager.cs</i>	45
Figura 5.20. Métodos de la clase <i>changingCameras.cs</i>	46
Figura 5.21. Método <i>Update()</i> de la clase <i>syringe.cs</i>	47
Figura 5.22. Métodos <i>TimerEnum()</i> y <i>FormatText()</i> de la clase <i>CounDownTimer.cs</i>	48
Figura 5.23. Clase <i>ChooseCharacter()</i>	49
Figura 5.24. Clase <i>LoadCharacter()</i>	50
Figura 5.25. Clase <i>CollElectricity()</i>	51
Figura 5.26. Clase <i>VacuneDose()</i>	52
Figura 6.1. Teclas de control del teclado. (Fuente: https://www.solotodo.cl/products/52270-xtrike-me-kb-301-kb-301)	53
Figura 6.2. Control del mouse para seleccionar. (Fuente: https://afnboysandgirlsclub.wordpress.com/mouse-skills/)	53
Figura 6.3. Resultado de escanear a Iván (autor del proyecto) utilizando escáneres en el MIC	54
Figura 6.4. Resultado de escanear a Iván utilizando <i>Scandy Pro 3D Scanner</i>	54
Figura 6.5. Antes y después de la malla de la cabeza en <i>Blender</i>	55
Figura 6.6. Reducción de detalle de la malla de la cabeza en <i>Blender</i>	55
Figura 6.7. Resultado final del personaje principal del videojuego.	56
Figura 6.8. Profesores y estudiantes seleccionables en el menú del juego.....	56
Figura 6.9. <i>Mutante</i> , enemigo del videojuego	57
Figura 6.10. <i>Quinto Paciente</i> , enemigo principal del videojuego.	58
Figura 6.11. <i>Científico de Guardia</i> , <i>Científico Principal</i> y <i>Director</i> del videojuego.	58
Figura 6.12. <i>Pacientes</i> y <i>Cuarto Paciente</i> del videojuego.	59
Figura 6.13. Imagen de la jeringuilla, arma principal del juego.	59
Figura 6.14. Imagen de una vacuna en el <i>Modo Desafío</i>	60
Figura 6.15. Imagen del bonus <i>incremento de dosis</i> en el <i>Modo Desafío</i>	60

Figura 6.16. Menú inicial de <i>MIC out of control</i> .	61
Figura 6.17. Selección del <i>Modo Historia</i> .	61
Figura 6.18. Interfaz <i>Multijugador</i> del <i>Modo Desafío</i> .	62
Figura 6.19. Interfaz <i>Selección de personaje</i> en <i>Multijugador</i> del <i>Modo Desafío</i> .	62
Figura 6.20. Interfaz <i>pantalla de carga</i> .	63
Figura 6.21. Interfaz <i>menú de pausa</i> .	63
Figura 6.22. Interfaz <i>Controles</i> .	64
Figura 6.23. Interfaz <i>Objetivo conseguido</i> .	64
Figura 6.24. Interfaz <i>Reintentar</i> .	65
Figura 6.25. Interfaz <i>Se acabó el tiempo</i> .	65
Figura 6.26. Interfaz <i>Modo Historia</i> .	66
Figura 6.27. Interfaz <i>Modo Desafío</i> .	66
Figura 6.28. <i>Nivel Fácil, Nivel Medio y Nivel Difícil</i> del <i>Modo Desafío</i> .	67
Figura 6.29. Fotograma del <i>Modo Historia</i> .	68
Figura 6.30. Primera misión del <i>Modo Historia</i> .	68
Figura 6.31. Segunda misión del <i>Modo Historia</i> .	69
Figura 6.32. Tercera misión del <i>Modo Historia</i> – <i>Vista General</i> .	69
Figura 6.33. Tercera misión del <i>Modo Historia</i> – <i>Primer Plano</i> .	69
Figura 6.34. Tercera misión del <i>Modo Historia</i> – Activación de mutantes.	70
Figura 6.35. Cuarta misión del <i>Modo Historia</i> – Activación de corriente.	70
Figura 6.36. Cuarta misión del <i>Modo Historia</i> – Obtención de vacuna.	70
Figura 6.37. Quinta misión del <i>Modo Historia</i> .	71
Figura 6.38. Barra de mutación y munición de dosis.	71

Índice de tablas

Tabla 2.1. Comparativa de las características de los motores gráficos más populares.	14
Tabla 2.2. Ventajas e inconvenientes de los mejores motores gráficos. (Fuente: https://www.reddit.com/r/gamedev/comments/dk1w8o/what_are_the_pros_cons_of_unity_vs_unreal).....	14
Tabla 4.1. Cronograma de tareas realizadas y sus respectivas fechas.....	30
Tabla 5.1. Ventajas y desventajas de <i>Perforce</i> sobre <i>Git</i> en <i>Unity3D</i> . Fuente: https://www.quora.com/What-are-some-advantages-of-Perforce-over-git	34
Tabla A1.1. Tabla que explica las diferencias entre los 3 niveles del <i>Modo Desafío</i>	80

1. Introducción

En este documento se describe el videojuego que se ha realizado, llamado MIC Coronavirus. Se trata de un videojuego estilo Survival Horror en 3D, con vista de tercera persona, realizado en el motor de Unity 3D (versión 2019). [1]

Se ha desarrollado con **fines académicos** dado el uso de recursos sin licencia¹ para comercializar, aunque no se descarta que a largo plazo se pueda comercializar obteniendo la licencia para ello y competir contra otros juegos del mismo género. Las herramientas escogidas para su realización se decidieron debido a la gran documentación que tienen y al conocimiento previo autodidacta obtenido de ellas.

Para la confección del trabajo fin de grado y considerando la diversidad de aspectos teóricos y prácticos abordados ha sido necesario hacer uso de gran parte de las competencias adquiridas durante la carrera.

1.1 MOTIVACIÓN

A lo largo de mi trayecto por la carrera, he tenido gran afán por usar los conocimientos adquiridos de la programación para crear algo visualmente atractivo y divertido, los videojuegos. La experiencia adquirida en las asignaturas ha sido la principal motivación para seguir creciendo en el diseño y desarrollo de videojuegos.

Desde la última década, la industria de los videojuegos está experimentando un gran crecimiento. Tanto ha sido su crecimiento que España ya es el quinto mercado de videojuegos más grande de Europa. Aunque el principal canal son los juegos en consolas como fuente de ingresos, se espera que se experimente un mayor crecimiento en los juegos casuales que son aquellos basados en apps para smartphones o tablets o accesibles desde un navegador. [2]

Este crecimiento es debido a la introducción de herramientas como *Unity* o *Unreal Engine* que facilitan la creación de videojuegos de alta calidad e incluso aplicaciones con el uso de muy pocos recursos. Esto implica que realizar juegos pequeños de manera independiente está al alcance de todos.

¹ Recursos tales como modelos 3D y efectos de partículas que se han utilizado para facilitar el desarrollo del videojuego con fines académicos.

Además, el interés por la inteligencia artificial en los videojuegos, ha sido una motivación extra.

El desarrollo de un videojuego necesita de conocimientos en muchos de los campos que un ingeniero informático debe conocer, como son la programación, ingeniería de software, inteligencia artificial, teoría de autómatas o redes de computadores. Así como otros más generales, como son la física o las matemáticas, el diseño de niveles e iluminación. Por todo esto se considera un proyecto completo y una forma adecuada de demostrar los conocimientos que se han aprendido en la carrera.

1.2 OBJETIVOS

El objetivo de este proyecto es el desarrollo de un videojuego estilo *Survival Horror* con modo historia en 3D, con la ayuda del motor gráfico de *Unity3D* ya que permite realizar un mismo proyecto en distintas plataformas. Entre estas plataformas destacan, *Windows* y *Android*, son las más conocidas e importantes en el mercado actual y en las que está implementado este proyecto.

No se pretende realizar el videojuego completo, sino una demo para mostrar las posibilidades de las herramientas elegidas. La demo consta de la jugabilidad clásica de un juego de tercera persona, con cadena de misiones, sistema shooter, además de añadir ciertos comportamientos inteligentes a los enemigos.

Se desea ampliar los conocimientos previos de las herramientas elegidas y con ello ganar experiencia en el proceso de creación de un videojuego en general, de cara a un futuro profesional en este campo.

Una vez finalizado, se espera poder complementar el proyecto con un desarrollo del multijugador del videojuego y la creación de avatares haciendo uso de impresoras 3D.

1.3 ESTRUCTURA DE LA OBRA

En este apartado se describe la estructura del documento. El documento comienza con una breve introducción explicando los objetivos que se quieren conseguir y las motivaciones que han llevado a la realización del mismo.

A continuación, se presenta el apartado del estado del arte donde se expone brevemente la situación actual del género del juego, junto a un estudio de la competencia donde se analiza los juegos más exitosos del género hasta la fecha. En último lugar, se describen las

herramientas que hay en el mercado para el desarrollo de videojuegos y concluye el apartado con la selección de las herramientas utilizadas en este proyecto.

En el siguiente apartado se aborda el diseño del videojuego de una manera resumida y visual.

La planificación es el siguiente apartado, donde se describe la distribución del tiempo para el desarrollo del proyecto.

Después la implementación constituye el siguiente apartado donde se explica todas las partes del desarrollo del proyecto. El documento sigue con el apartado conclusiones donde se analiza el resultado final del proyecto.

Finalmente se presentan los apartados agradecimientos, trabajos futuros, bibliografía, el apéndice y un anexo que contiene el documento de diseño.

1.4 ESQUEMA DEL TRABAJO

Para la realización de este proyecto se ha seguido el siguiente esquema de trabajo:

- ❖ Creación del documento de diseño del videojuego (*Game Design Document - GDD*).
 - ❖ Creación del diagrama de Gantt para planificar el proyecto.
 - ❖ Búsqueda de recursos audiovisuales.
 - Búsqueda de modelos 3D para enemigos.
 - Búsqueda de recursos para la decoración del modelado.
 - ❖ Virtualización 3D: Importación al videojuego de una copia exacta de la persona a un avatar que desarrollaremos con el uso de impresoras 3D.
 - ❖ Programación del videojuego.
 - Programación de los enemigos.
 - Programación de los movimientos del personaje.
 - Programación de la interacción del jugador con el nivel.
 - Programación de todas las animaciones del modo historia.
 - Programación de menús e interfaces.
 - Pantalla de inicio.
 - Pantalla de opciones.
 - Interfaz del juego.
 - ❖ Creación de ajustes finales del nivel.
 - Iluminación y decoración

- Inclusión de audio.
- ❖ Corrección de errores, ajuste de dificultad del juego y ajustes finales.
 - Testeo del juego y búsqueda de posibles bugs en la jugabilidad.
- ❖ Redacción de la memoria.

1.5 METODOLOGÍA

De las diferentes metodologías estudiadas para el desarrollo de software en la carrera se ha optado por una metodología basada en un modelo de desarrollo iterativo e incremental. Este modelo combina el modelo en cascada donde las fases de desarrollo (análisis, diseño, implementación y pruebas) se realizan de forma secuencial, con la reiteración de este proceso en diversos estadios de la implementación. Empleando el desarrollo incremental se obtiene, al final de cada iteración, una versión funcional del producto de esta manera el sistema se desarrolla poco a poco y se obtiene siempre un *feedback* continuo con el usuario. Se ha elegido esta metodología porque es la que mejor se adapta y la que más se suele emplear en proyectos destinados al desarrollo de videojuegos. Este método se ha aplicado en cada una de las subtareas asignadas a las tareas principales descritas en el apartado planificación.

2. Estado del arte

En este apartado se hará un estudio del mercado actual de videojuegos tanto para PC como plataformas de videojuegos, profundizando en los *Survival Horror* para valorar las posibles opciones que se ofrecen en este género y sus puntos fuertes. También se identificarán las características comunes para plantear mejoras.

Para finalizar, se describirán las herramientas que se han utilizado para la realización de este proyecto.

2.1 SITUACIÓN DEL GÉNERO SURVIVAL

El género *Survival* se encuentra en pleno auge a día de hoy. El primer juego de supervivencia fue lanzado en 1878, se llamaba *Where's My Pouch*, aunque no se haría ninguno comercial y popular hasta 1982 con *The Oregon Trail*. [3]

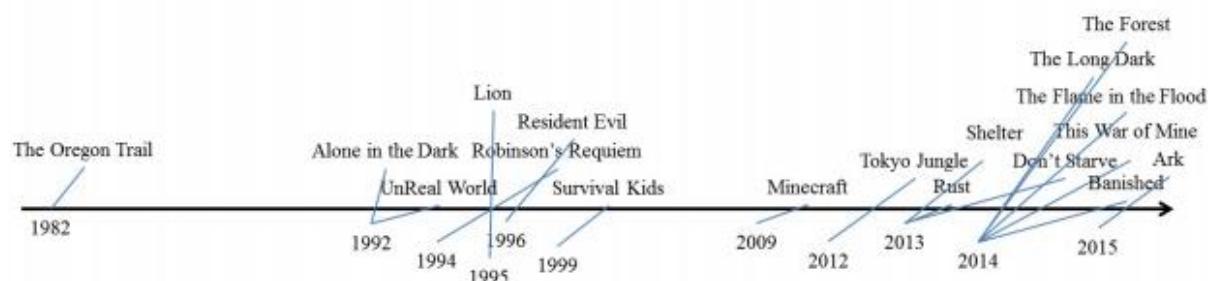


Figura 2.1. Cronología aparición de videojuegos más importantes del género de Supervivencia. [4]

Sin embargo, hubo una pequeña pausa en esta industria hasta 1992, fecha en la cual llegó un enorme aumento de emisiones de videojuegos del género hasta la actualidad. [4]

Los juegos de supervivencia se caracterizan porque requieren que el jugador normalmente esté sólo o separado de los demás, y debe hacer uso de sus habilidades de planteamiento, pensamiento e innovación para, como el propio nombre del género indica, tratar de sobrevivir y completar un objetivo. [5]

En general los videojuegos de supervivencia son juegos de acción y aventura, requiriendo la adquisición y gestión de recursos escasos, confrontaciones con amenazas humanas o animales y elaboración de objetos.

El jugador, generalmente se coloca solo en el mundo del juego y con pocos recursos. Los NPC (*non-player character*) suelen ser jugadores hostiles, por lo que se hace más hincapié

en la evitación, en lugar de la confrontación.

Raramente no suele haber una condición ganadora en este género. Normalmente, el desafío para el jugador es durar el mayor tiempo posible con vida.

El género presenta una oportunidad interesante para examinar las capacidades de emerger, evolucionar y comprometer y restringir al jugador simultáneamente, pues refleja los temores culturales sobre la escasez y recesión inevitable. Esta dinámica va más allá de las reglas e interfaz del juego, está presente en todos los juegos del género.

Si bien bases de datos de juegos de *Giantbomb* calculan 103 subgéneros *survival*, vamos a analizarlos desde una visión más general. Destacan los *Survival Horror* y *Wilderness Survival*, los cuales analizaremos detenidamente.

2.1.1 Survival Horror

Son juegos que tratan de hacer sentir al jugador como un humano en un entorno invadido por demonios, zombies, fantasmas, etc.

En 1992, se lanzó el primer juego de terror “*Alone in the Dark*” en el que el jugador tiene muy limitadas las formas de enfrentarse a monstruos y lo más inteligente es evitarlos por completo.

Pero sin duda alguna, un gran ejemplo de este subgénero es “*Resident Evil*”. [6] Su entorno suele ser una gran mansión repleta de zombies, siendo necesaria la adquisición de recursos como hierbas curativas o munición para sobrevivir.



Figura 2.2 Juego *Resident Evil*. (Fuente: <https://www.residentevil.com/>)

2.1.2 Wilderness Survival

La mejor forma de entender este subgénero es pensar en la primera persona que llegase a la Tierra, sin recurso alguno. Por lo cual, el jugador se encuentra en lugares de mundo abierto, inhóspitos o con escasez de recursos.

Normalmente debe buscar comida y crear herramientas para obtener materias primas útiles para sobrevivir.

Un ejemplo podría ser *Unreal World* [7], un RPG 2D (*Role Playing Game*) de fantasía en el que hay que sobrevivir buscando alimento y herramientas. Aunque debido a sus últimas actualizaciones podría catalogarse como *Survival Híbrido*.



Figura 2.3. Juego *Unreal World*. (Fuente: <https://store.steampowered.com/>)

2.1.3 Survival Híbrido

Este subgénero es el que mayor aceptación ha tenido, pues une todas las herramientas de los *survival* mezcladas con mecánicas de otros géneros.

Aquí tenemos numerosos ejemplos, pero sin duda los más aclamados han sido *Minecraft* [7] y *Fortnite* [8].

Minecraft es un mundo abierto en la que todo está constituido de cubos. En el que debes recolectar comida y minerales, construir edificios y equiparte para sobrevivir tanto del hambre como de los monstruos.



Figura 2.4. Juego *Minecraft*. (Fuente: <https://www.minecraft.net>)

Fortnite es un juego multijugador que mezcla supervivencia y acción. En su principal modo multijugador, el jugador deberá abrir cofres para equiparse y recolectar materias primas para construir y defenderse de los otros 100 jugadores. El último con vida gana.



Figura 2.5. Juego *Fortnite*. (Fuente: <https://www.epicgames.com/fortnite/>)

2.2 ESTUDIO DE LA COMPETENCIA

Por tratarse de un juego del subgénero *Survival Horror* encontramos variada competencia a día de hoy, la mayoría orientada a zombies. Sin embargo, esta competencia no afecta directamente las posibilidades de éxito del juego de cara al mercado, pues la mayoría de

juegos del género se caracterizan por tener un modo historia limitado y monótono, dando a los jugadores la posibilidad de probar varios en un corto periodo de tiempo.

Outlast [9] es un juego en primera persona desarrollado y publicado por la conocida compañía *Red Barrels Games* lanzado para PC y consolas de videojuegos.

El protagonista del juego, a diferencia de muchos protagonistas de juegos de *Survival Horror*, es incapaz de combatir; en su lugar, el personaje es capaz de desplazarse, escalar u ocultarse en lugares de su entorno para así poder eludir a los enemigos. Para desplazarse por el entorno, el personaje es capaz de escalar lugares altos, evitar obstáculos, arrastrarse, y deslizarse en espacios estrechos usando el parkour. La resistencia del personaje es limitada y se puede recuperar progresivamente de las heridas si permanece un determinado tiempo quieto, sin ser atacado [10].

La mayor parte de los enemigos en el juego son variantes o pacientes del manicomio que presentan mutaciones y comportamientos agresivos respecto al resto de internos del asilo; otros, en cambio, se encargan de brindar ayuda o consejos a *Miles Upshur*.



Figura 2.6. Juego *Outlast*. (Fuente: <https://www.eneba.com/latam/steam-outlast-steam-key-global>)

Resident Evil [11], con más de 61 millones de copias vendidas, encabeza el género *Survival Horror*. Han lanzado varias series, todas ellas para consolas de videojuegos (*PlayStation*, *Xbox One*).

Lanzado el 24 de Enero de 2017, *Resident Evil 7* es la última serie lanzada. El jugador controla a Ethan Winters, quien a diferencia de los protagonistas de los otros *Resident Evil*,

es solo un civil con pocas habilidades especiales de lucha, y sin ningún entrenamiento de combate policial o militar. Aun así, es capaz de usar un considerable repertorio de armas (tanto armas de tipo cuerpo a cuerpo como de disparos), y tiene la opción de oponerse, y luchar contra los distintos enemigos. Este repertorio de armas tiene en el inicio del juego unas opciones de armamento bastante limitadas, ampliéndose estas tanto en la cantidad como en su potencia, conforme el jugador progresá en la historia y desbloquea nuevos elementos de lucha. Además, el jugador al igual que en las últimas entregas, puede girar rápidamente 180 grados para localizar y evitar a los enemigos. Por primera vez en la saga principal se tiene la posibilidad de bloquear los ataques de los enemigos para reducir el daño. Varios tramos del juego consisten en ser acechados por los miembros de la familia *Baker*, quienes no pueden ser aniquilados en los primeros combates, solo pueden ser temporalmente incapacitados. Sin embargo, estos encuentros son evitables mediante el sigilo o huyendo. [12]



Figura 2.7. Juego *Resident Evil VII* (<http://umlconnector.com/2017/02/resident-evil-7-puts-survival-back-in-horror-genre/>)

Silent Hill [13], el top 2 en el subgénero con más de 2 millones de copias vendidas ha creado varias series, la última lanzada en 2004 para todas las plataformas es *Silent Hill 4: The Room*.

Silent Hill 4, a diferencia de los títulos anteriores, los cuales tenían como escenario el pueblo de Silent Hill, se desarrolla en el pueblo ficticio de *South Ashfield*, y se enfoca en el personaje de *Henry Townshend*, el cual trata de escapar de un encierro sobrenatural en su apartamento. Al hacerlo termina explorando una serie de mundos extraños y se encuentra a sí mismo en medio de un conflicto con un asesino en serie relacionado con la mitología de *Silent Hill*. [14]

En los niveles principales el juego usa la perspectiva en tercera persona tradicional de la serie *Silent Hill*. A diferencia de otros títulos, el jugador sólo tiene un inventario de objetos limitado que puede organizarse dejando objetos no necesarios en un baúl en el apartamento de Henry. *Silent Hill 4: The Room* hace énfasis en el combate; emplea rompecabezas simples y basados en buscar objetos en lugar de acertijos complicados.

Durante la segunda mitad del juego, *Eileen Galvin*, la vecina de *Henry*, lo acompaña; ella no puede morir mientras esté con *Henry*, aunque cuanto más daño recibe más sucumbe a una posesión por el antagonista de la historia. El jugador también puede equipar a *Eileen* con un arma para que se una a *Henry* en el combate. El daño que *Eileen* recibe determina sus posibilidades de sobrevivir a la batalla final, afectando directamente al final del juego.

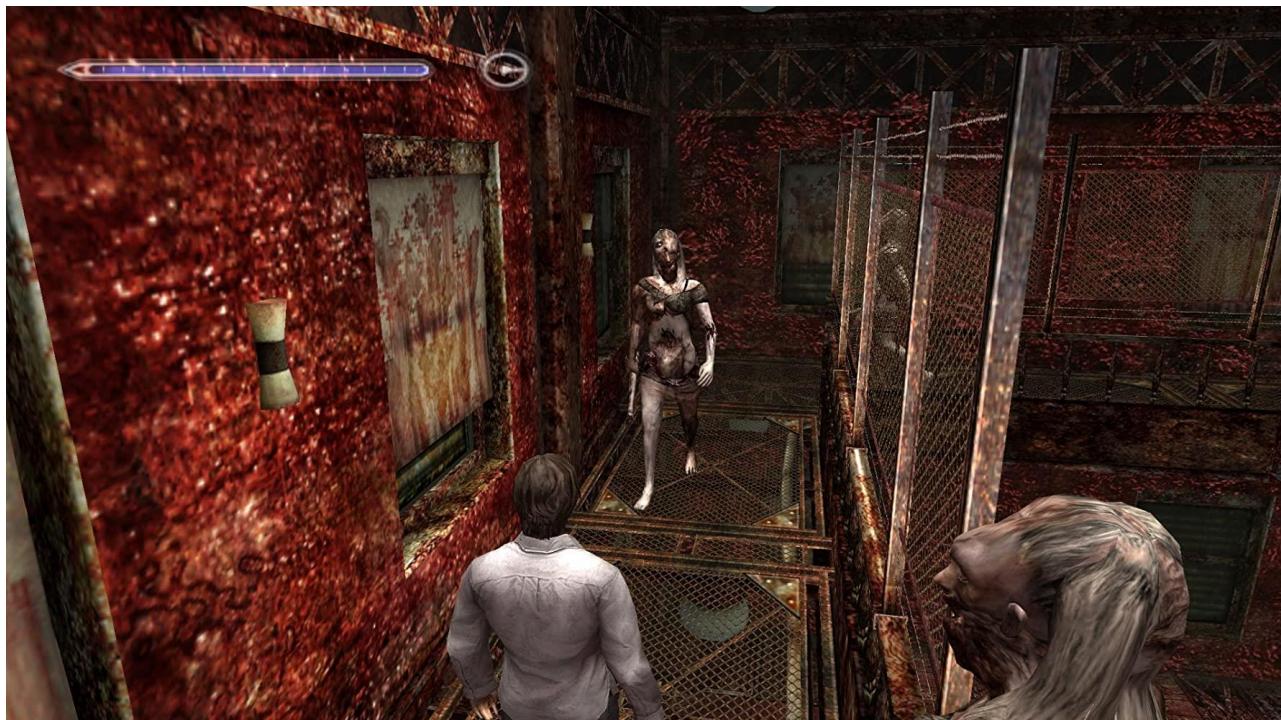


Figura 2.8. Juego *Silent Hill 4* (Fuente: <https://www.ecranlarge.com/jeux-video/1119575-silent-hill-4-the-room>)

2.3 MOTORES GRÁFICOS

En la actualidad existen una gran cantidad de motores gráficos para el desarrollo de videojuegos. Cada uno de ellos presentan unas características que se deben valorar a la hora de realizar un videojuego. A continuación, se describen algunos de los motores más importantes, así como sus ventajas y desventajas.

2.3.1 Unreal Engine

Unreal Engine [15] es el motor de *Epic Games*. Estuvo orientado solo para desarrolladores. Desde 2015 es una herramienta totalmente gratuita con el único requisito de que hay que pagar un 5% al comercializar un producto realizado con este motor. Se usa C++ como lenguaje de programación y también cuenta con un editor visual de *scripting*, también conocido como sistema de *blueprints*. Como ventajas cabe destacar la calidad gráfica y el poder crear juegos complejos, además de ser gratuito. Sin embargo, su curva de aprendizaje es complicada y cuenta con una comunidad inferior con respecto a otros motores, siendo su principal desventaja.

2.3.2 Unity

Unity3D [16] es, probablemente, el motor de juegos más usado en la actualidad. Permite desarrollar juegos en 2D y en 3D, cuenta con una documentación muy amplia y una inmensa comunidad. Cuenta con una versión gratuita y una de pago, aunque sus diferencias son mínimas. Soporta el lenguaje C# con sintaxis similar a Java y es un editor multiplataforma permitiendo exportar a muchas plataformas distintas incluyendo hasta consolas. Puede usarse con juegos comerciales siempre y cuando no se superen unos ingresos de 100.000 dólares al año. Recibe constantes actualizaciones cada cierto periodo de tiempo y su curva de aprendizaje es sencilla.

2.3.3 RPGMaker

RPGMaker [17] es un software para hacer juegos hecho como base para los usuarios que no saben programación. Aunque si se poseen conocimientos de programación puede ser útil para desarrollar *plugins* usando *JavaScript*. Este software sirve para desarrollar juegos mayoritariamente del género RPG orientado a 2D, ya que posee un editor para la creación

de escenarios y personajes mediante sprites. Fue desarrollado por *KADOKAWA Corporation*.

2.3.4 GameMaker Studio

GameMaker [18] tuvo su origen en los años 90, cuando *Mark Overmars* (su creador) empezó a crear una herramienta de animación para ayudar a sus estudiantes, con el paso del tiempo se ha vuelto una herramienta muy popular para la creación de videojuegos, principalmente en 2D. Este motor utiliza su propio lenguaje GML, el cual es muy flexible, su sintaxis es comparable con la de C++. Es capaz de exportar a una gran cantidad de plataformas distintas, móviles o escritorio, sin embargo, el editor sólo funciona en *Windows*. La versión gratuita es limitada ya que solo permite la exportación a *Windows*.

2.3.5 Godot

Godot [19] es un motor de código abierto, totalmente gratuito, para desarrollar juegos en 2D y 3D. Creado por *OKAM Studio* y funciona en *Windows*, *OS X* y *Linux* entre otros. Hace uso de un lenguaje de scripting propio basado en *Python*. Permite exportar a plataformas móviles y de escritorio. La documentación es limitada y está únicamente en inglés, es una desventaja en comparación con otros motores que sí tienen su documentación traducida a otros idiomas.

2.3.6 CryEngine

CryEngine [20] un motor de videojuegos lanzado en 2002 y creado por un estudio alemán llamado *Crytek*. Fue lanzado originalmente para hacer una demo técnica para la empresa de tarjetas gráficas *Nvidia*. Tras su gran éxito los mismos creadores del motor acabaron desarrollando el videojuego *FarCry*. En 2016 pasarían a pertenecer a la empresa *Ubisoft* los derechos de dicho motor. El motor está programado con *Lua*, *C++* y *C#*. Las primeras versiones del motor tenían una licencia de pago. Actualmente con la versión 5 el motor es totalmente gratuito y se limita a las donaciones que los usuarios quieran realizar de forma voluntaria. Se utiliza una programación basada en nodos muy parecida a los *blueprints* del motor *Unreal*. Cuenta con una comunidad pequeña con una amplia documentación que se puede encontrar en su página oficial [17], además de una gran cantidad de videotutoriales en *Youtube*. Toda esta documentación se encuentra en inglés.

A continuación, se muestra una tabla comparativa de los motores mencionados y sus características más importantes:

Motor gráfico & Atributos	Unity	Unreal Engine	RPG Maker	GameMaker Studio	Godot	CryEngine
Orientación	2D y 3D	2D y 3D	2D	2D	2D y 3D	3D
Sistema Operativo	Windows, Mac	Windows	Windows	Windows	Windows, Linux, OS X	Windows, Linux, OS X
Lenguaje de Programación	Lenguaje C#	Lenguaje C++, Editor visual Blueprints	Javascript	Lenguaje propio GML	Lenguaje de scripting	Programación basada en nodos
Licencia	Gratis	Paga	Paga	Versión gratuita limitada	Gratis	Gratis en la versión 5
Documentación	Muy amplia	Escasa	Escasa	Amplia	limitada	Amplia
Curva de aprendizaje	Sencilla	Complicada	Sencilla	Media	Media	Media

Tabla 2.1. Comparativa de las características de los motores gráficos más populares.

2.4 SELECCIÓN DE LAS HERRAMIENTAS

2.4.1 Motor gráfico

Para la elección del motor se han optado por dos opciones (*Unity* o *Unreal Engine*). Para decidir una de las dos opciones se ha realizado una tabla comparativa donde se muestran las diferentes ventajas e inconvenientes de ambos motores.

Motor gráfico	Unity	Unreal Engine
Ventajas	<ul style="list-style-type: none"> Curva de aprendizaje sencilla. Soporta el lenguaje C# Gran facilidad para realizar juegos en 3D o 2D Cuenta con una comunidad muy amplia y documentación muy detallada. 	<ul style="list-style-type: none"> Buen rendimiento y gráficos realistas. Soporta el lenguaje C++ y también el uso de <i>Blueprints</i>. Control total del código fuente del motor.
Inconvenientes	<ul style="list-style-type: none"> Calidad gráfica menor que <i>Unreal Engine</i> No es de código abierto por lo que los errores no los puede solucionar el propio programador. 	<ul style="list-style-type: none"> Comunidad pequeña con pocos recursos. La dificultad del lenguaje C++

Tabla 2.2. Ventajas e inconvenientes de los mejores motores gráficos. (Fuente: https://www.reddit.com/r/gamedev/comments/dk1w8o/what_are_the_pros_cons_of_unity_vs_unreal/)

Teniendo en cuenta el estudio realizado de ambos motores, *Unity* es el mejor posicionado para la elaboración de este proyecto. Pese a que *Unreal* tiene un mejor rendimiento y su nivel gráfico es mejor que *Unity*, la poca experiencia que se tiene sobre este motor dificultará considerablemente este proyecto. Además, la API que se va a usar en este proyecto está implementada en C#, lenguaje que no es compatible con el motor *Unreal* ya que soporta C++.

Sin embargo, *Unity* soporta el lenguaje C#, tiene una amplia comunidad y una cantidad de recursos disponibles en la red que facilitan el desarrollo de cualquier juego y se cuenta con una experiencia previa con la herramienta. Es por ello que finalmente se ha decidido utilizar *Unity*.

INTRODUCCIÓN A UNITY

Unity se fundamenta principalmente en los *GameObjects*. Estos objetos son los elementos que contienen una escena en *Unity*. Una escena es el espacio donde se representan todos los elementos que componen nuestro juego, así como los personajes, la cámara, el terreno... Todos ellos se encuentran dentro de la misma.

Todo *GameObject* que se encuentra dentro de la escena está compuesto por un componente llamado *Transform*. Este componente es único y lo poseen todos los *GameObjects*. El componente *Transform* se encarga de dar la posición, rotación y escala del objeto representado en la escena.

También existen otros componentes como el *Renderer* que se encarga de hacer visible a los objetos en la escena, además de dar un color y una textura, el *Rigidbody* y el *Collider*, que gestionan las colisiones con otros elementos y las características de la física gestionada por el propio motor.

Otro componente importante es la *Camera*. Toda escena en *Unity* tiene un objeto llamado *MainCamera* que se encarga de renderizar la escena.

INTERFAZ DE UNITY

La interfaz de *Unity* es muy intuitiva y consta de una serie de ventanas principales que el usuario puede colocar a su gusto. Las ventanas principales son las siguientes:

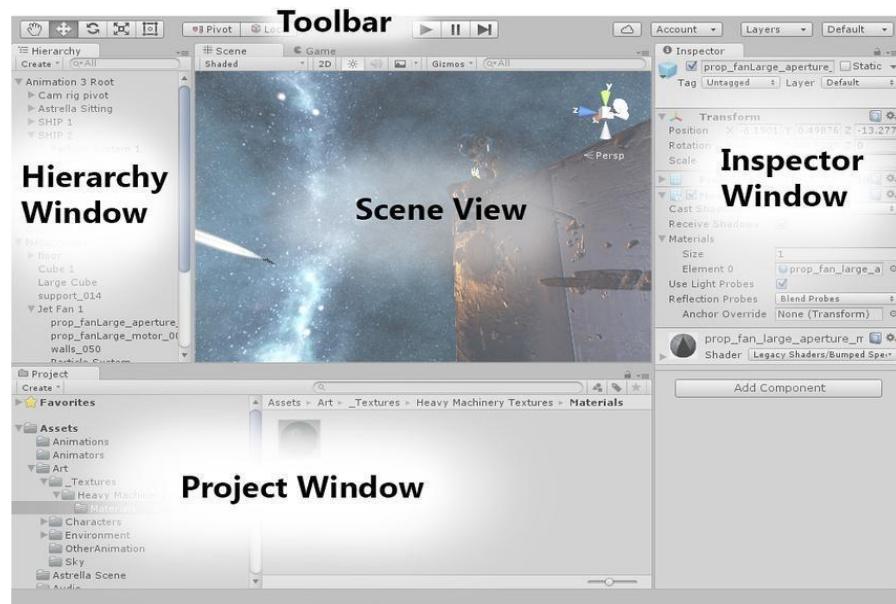


Figura 2.9. Interfaz de *Unity*.

La **ventana del proyecto** consta de un explorador de archivos que nos permite añadir, eliminar y organizar elementos que se denominan *assets*. Esta ventana consta de dos pestañas. En la izquierda se encuentra toda la jerarquía de carpetas de nuestro proyecto cuyo directorio raíz es *Asset*. Este directorio se sincroniza en tiempo real con la carpeta del sistema: si se realiza cualquier cambio, *Unity* actualizará el proyecto. En la pestaña de la derecha se encuentran todos los archivos que aparecen dentro la carpeta seleccionada de la jerarquía de carpetas. También cuenta con un buscador para facilitar la búsqueda de archivos o carpetas en caso de que el usuario no recuerde dónde se encuentran.

La **ventana de *Hierarchy* (Jerarquía)** contiene una lista de cada *GameObject* (también llamados “*Objects*”) en la escena. Estos objetos pueden ser instancias directas de archivos de *assets*, y otras son instancias de *prefabs*, que son objetos personalizados que una vez creados en la escena al desplazarse a la ventana de proyecto se convierten en *prefabs*. Del mismo modo cuando el usuario importa modelos 3D estos son *prefabs* que cuando son desplazados a la escena se convierten en *GameObjects* y estos aparecen en la ventana jerarquía en orden de creación. Esta ventana también permite crear objetos y formar una jerarquía con solo arrastrar el objeto hijo al objeto padre.

La **ventana de escena** es la vista interactiva del mundo que se está creando. La vista escena se utiliza para seleccionar y posicionar paisajes, personajes, luces y todos los demás objetos que componen el juego.

La **ventana de juego** muestra el resultado representado en la escena mediante una o más cámaras que se encargan de renderizar los objetos que hay en ella. Desde el editor se pueden realizar diferentes acciones como poder pausar el juego, ajustar el juego a distintas resoluciones o ejecutar el juego *frame a frame* para mayor detalle.

El **inspector** se encarga de mostrar de forma detallada toda la información asociada al *GameObject* seleccionado en la escena o en la ventana jerarquía. Incluye información de todos los componentes asociados y sus propiedades. También permite modificar la funcionalidad del *GameObject* y añadir nuevos componentes.

La **ventana de animación** es donde se pueden realizar clips de animación desde cero. Consta de una línea del tiempo donde se puede crear cualquier animación.

La **ventana del animator** nos permite crear máquinas de estados y asignar animaciones a cada uno de ellos.

SCRIPTS

Los scripts son un componente más de *Unity*, se crean para darle un comportamiento a los objetos que se crean en la escena de nuestro juego. Para crear un script solo hay que hacer *click* con el botón derecho del ratón dentro de la ventana proyecto y se desplegará un menú con varias opciones.

La primera opción es *create* y desde ahí se despliega otro menú donde aparece la opción *C# script* [18]. Una vez creado aparecerá un fichero con el símbolo de C# pidiendo que se introduzca un nombre al script. Es importante saber que el nombre del script es el nombre de la clase que se acaba de crear y es por ello que tiene que coincidir ya que de no ser así se producirá un error de compilación.

Otra forma de crear un script es seleccionando el objeto al que se le quiere asignar, y en la ventana *inspector* aparece un botón llamado *add component*, el cual, si se hace *click* sobre él, se despliega una serie de opciones entre las cuales se encuentra la opción *script*.

Todos los scripts creados heredan de la clase que contiene todas las clases y funciones necesarias para poder programar nuestro juego. El ciclo de vida de un script se compone de varias funciones principales que son las siguientes:

- **Awake.** Este método se ejecuta antes de cualquier función *Start* y también justo después de que un *prefab* es instanciado. Hay que tener en cuenta que si un *GameObject* está inactivo durante el comienzo, *Awake* no es llamado hasta que se vuelva activo.

- **Start.** Es llamado antes de la primera actualización de *frame* sólo si la instancia del script está activada. Este método se ejecuta después del *Awake*. Por defecto *Start* siempre aparece cuando se crea un script. Se suele utilizar para la inicialización de variables.
- **Update.** Se llama una vez por *frame*. Es la función principal para las actualizaciones de *frames*. Esta función aparece siempre por defecto cuando se crea un script. El tiempo que tarda en ejecutarse puede variar dependiendo de las imágenes por segundo en ese preciso instante.
- **LateUpdate.** Es llamada una vez por *frame* después de que *Update* haya finalizado. Cualquier cálculo que sea realizado en *Update* será completado cuando *LateUpdate* comience. Un uso común de *LateUpdate* sería una cámara en tercera persona que sigue.
- **FixedUpdate.** Es invocado en cada iteración del bucle de físicas. Esta función se utiliza para todo lo relacionado con las físicas de nuestro juego. La diferencia con la función *Update* es que esta función sí que tiene un tiempo fijo entre llamada. Se debe tener en cuenta que la sobrecarga de esta función puede repercutir negativamente en el rendimiento de nuestro juego.

Lo explicado anteriormente no es todo lo que puede ofrecer *Unity*. *Unity* tiene una gran cantidad de funciones y de características que se pueden encontrar en la documentación oficial [19] [20] y en su página web.

2.4.2 Control de versiones

El control de versiones es una herramienta imprescindible para proyectos de este calibre, ya que permite evitar pérdidas de información y llevar un control de las modificaciones que se hacen durante el desarrollo de un proyecto. A continuación, se presentan diferentes herramientas para el control de versiones:

Unity Collaborate, [21] permite equipos pequeños de desarrollo, así como guardar, compartir o sincronizar un proyecto de *Unity* en un entorno alojado en la nube. Esta herramienta es bastante interesante ya que permite tener un control de versiones. También permite recuperar archivos individuales o el proyecto en un estado anterior. Esta herramienta está disponible a partir de la versión de *Unity* 2017.4 y cuenta con dos licencias. Una gratuita que permite almacenar en la nube hasta un 1Gb y otra de pago que cuesta 9 dólares por mes y permite almacenar hasta 25Gb.

Git, [22] es un sistema de control específico de versión de fuente abierta creada por *Linus Torvalds* en el 2005. Específicamente, *Git* es un sistema de control de versión distribuida, lo que quiere decir que la base del código entero y su historial se encuentran disponibles en la computadora de todo desarrollador, lo cual permite un fácil acceso a las bifurcaciones y fusiones.

GitHub, [23] es una compañía sin fines de lucro que ofrece un servicio de hosting de repositorios almacenados en la nube. Esencialmente hace que sea más fácil para los desarrolladores usar *Git* como la versión de control y colaboración. [24]

Perforce, [25] es un control de versiones que funciona en modo cliente/servidor. El servidor gestiona una base de datos central que contiene uno o más repositorios (*depots*) con versiones de los ficheros. Los clientes importan los ficheros del repositorio a su taller de trabajo (*workspace*) para trabajar en ellos, y posteriormente los devuelven modificados agrupados en listas de cambio. La conexión se realiza mediante TCP (*Transmission Control Protocol*) usando protocolos propietarios de RPC (*Remote Procedure Call*) y streaming. [26]

SVN, (*Apache Subversion*) es una herramienta de control de versiones de código abierto basada en un repositorio cuyo funcionamiento se asemeja enormemente al de un sistema de ficheros. Es un software libre bajo una licencia de tipo *Apache/BSD* [27].

Comparando cada una de las herramientas, se ha elegido utilizar *Perforce* por ser una herramienta gratuita que, además, no tiene límite de almacenamiento en la nube. Al contrario que *Unity Collaborate* que sí tiene un límite de almacenamiento y es de pago. Sin embargo, la alternativa de *SVN*, al no tener yo apenas experiencia con la herramienta, queda descartada.

2.4.3 Plugins y librerías

Unity cuenta con una gran cantidad de herramientas que mejoran algunas de las características del motor, permitiendo facilitar, también, las tareas de implementación. A continuación, se muestran diferentes herramientas que se han utilizado en este proyecto:

Log Viewer, [28] esta herramienta permite verificar fácilmente los registros de la consola del editor dentro del juego. Esto permite encontrar rápidamente los errores que se puedan producir en tiempo de ejecución. Para poder mostrar esta consola lo que se tiene que hacer es un gesto circular con el mouse en *Windows* (hacer clic y arrastrar), o el dedo (tocar y arrastrar) en la pantalla del dispositivo móvil, para mostrar todos esos registros. Esta herramienta es gratuita. Aunque hay otras alternativas, como son *Reporter* o *LogCat Viewer*,

son herramientas de pago más completas pero su funcionalidad es la misma, por lo que la herramienta gratuita es suficiente para el proyecto.

3. Diseño

3.1 DISEÑO CONCEPTUAL

“MIC Coronavirus” es una demo del género *Survival Horror* en 3D donde el jugador es el protagonista en todo momento.

En el *Modo Historia*, se encuentra en su centro de enseñanza, en el cual se llevaban a cabo investigaciones para encontrar la vacuna del Coronavirus. Pero algo salió mal y una mutación muy peligrosa se expandió contagiando a todas las personas del edificio. El jugador debe evitar ser contagiado mientras busca pistas de lo sucedido, encuentra a los científicos a cargo de la investigación y juntos buscan la forma de curar a todos los mutados del edificio.

Mejoras en el juego a medida que avanzas. Al inicio del juego, el jugador sólo podrá esquivar a los *mutantes*. Una vez avance en el *modo historia* y obtenga la vacuna de la anomalía, podrá dispararles cargas de cura limitadas y así sanarlos.

En el *Modo Desafío* tendremos que sanar una cantidad de mutantes determinada antes de que se acabe el tiempo. Si lo conseguimos y no somos contagiados, pasaremos de nivel de dificultad.

Los *mutantes* se encuentran en un estado de reposo, pero cuando el jugador entra en su radio de detección o le intentamos curar, el enemigo va hacia él y sólo estando cerca, contagiará al jugador.

3.2 DISEÑO ESTRUCTURAL

El diseño estructural abarca las diferentes clases empleadas para la implementación del proyecto. Cada clase está acomodada en un único script y los scripts se interconectan entre sí. Se pueden dividir en: las clases para el jugador principal, las clases para los enemigos, las clases para la interfaz gráfica, las clases dedicadas a iteración con objetos del mundo, la clase de la jeringuilla, las clases para intercambios de escenas y las clases dedicadas a la iluminación y sonido. Aunque más adelante veremos con más detalle los scripts (apartado 5. Implementación) describiremos brevemente los más importantes.

Las clases dedicadas al jugador principal se encargan de gestionar tanto el movimiento como la vida y daño del jugador. Lo veremos con más detalle en el apartado 3.3.

Las clases dedicadas a la IA de los enemigos se encargan de gestionar su movimiento, por donde pueden pasar y su rango de visión al jugador (script *IAEnemies.cs*). También tienen una barra de mutación que puede descender si les disparan vacunas. Una vez son sanados, vuelven a ser estudiantes o profesores (script *Health_and_damageIA.cs*). El funcionamiento de su *animator controller* será detallado en el punto 3.3.

Las clases dedicadas a la iteración con objetos del mundo se encargan de diversas funciones: gestionar lo que sucede cuando estamos cerca de un objeto o cuando lo agarramos, cuando lo activamos o desactivamos, etc. (ej: scripts *countDownTimer.cs*, *CollElectricity.cs*, *IncrementDamage.cs*, etc).

La clase de la jeringuilla se encarga de la utilización del arma, la cantidad de munición, el daño y los impactos de las dosis (script *Syringe.cs*).

Las clases para intercambios de escena sirven para poder pasar de un terreno a otro funcionando como un teletransporte (scripts *AreaEntrance.cs*, *AreaExit.cs*).

Las clases dedicadas a la iluminación y sonido se encargan de encender/apagar/alterar las luces del edificio, así como de los efectos de sonido del jugador o los enemigos en cada momento (*FlickerLight.cs*, *HorrorSounds.cs*, etc).

3.2.1 Diseño estructural de cada *Modo de juego*

En el **Modo Desafío** tendremos tres niveles diferentes pero estructuralmente idénticos: un cronómetro en modo cuenta atrás, munición de dosis infinita y varios *mutantes* repartidos por todo el mapa con todos sus scripts (que, dependiendo de la dificultad del nivel, serán más débiles o más fuertes). Lo más interesante del Modo Desafío es la funcionalidad de sus *Vacunas* e *Incrementos de dosis*. Las *vacunas* son usadas para quitarnos mutación, y los *incrementos de dosis* para aumentar el daño de dosis a los enemigos durante unos segundos. Serán explicados en detalle en el apartado 5.6.3.

En el **Modo Historia**, debemos hacer numerosos cambios de escena. Esto conlleva salvaguardar datos y de esto se encarga la clase-script *GameManager.cs*. Para ello tiene acceso a varios scripts. Guarda información fundamental como la misión en la que nos encontramos, el progreso que llevamos en la misión, la cantidad de vida que nos queda o el tiempo restante para completarla. También guarda referencias a *gameObjects* que necesitaremos tener ubicados para poder acceder a sus *scripts* (la jeringuilla, el cronómetro, los científicos, etc). Veámoslo de manera más intuitiva:

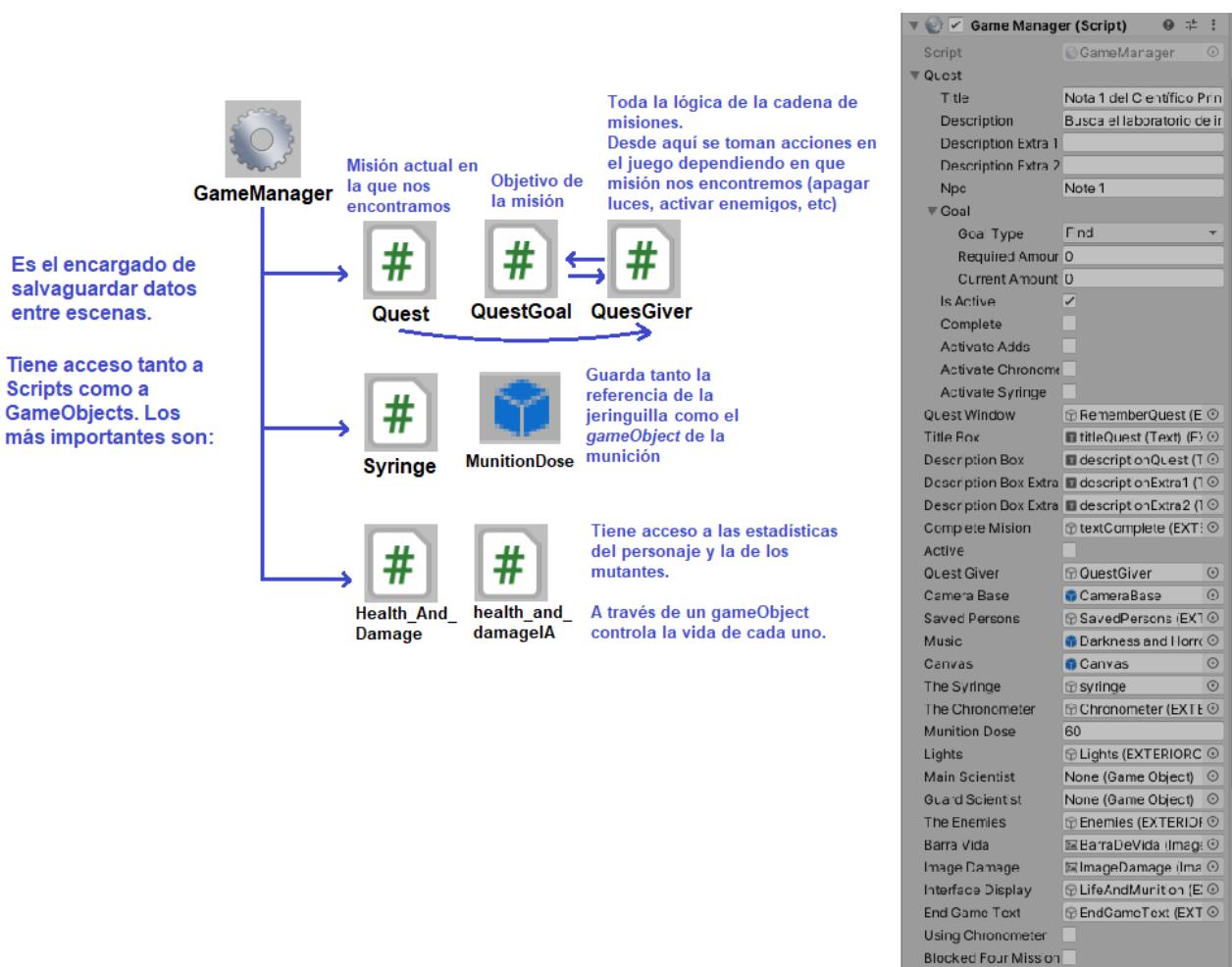


Figura 3.1. Diagrama del *GameManager* que muestra que datos guarda entre escenas.

3.3 DISEÑO DE LA INTELIGENCIA ARTIFICIAL (IA)

Los *mutantes* son los encargados de dar vida a los escenarios del juego por donde pasamos a través sus comportamientos. Para ello deben realizar movimientos y ataques de forma autónoma. Esto lo haremos a través de una pequeña *inteligencia artificial*.

3.3.1 Que es una inteligencia artificial

La *inteligencia artificial* (IA) es una combinación de algoritmos que permite a una máquina ejecutar procesos de forma similar a un ser humano [29]. Según los algoritmos y procesos empleados, tenemos varios tipos de *inteligencias artificiales*: máquinas reactivas, memoria limitada, teoría de la mente, autoconciencia, etc.



Figura 3.2. Tipos de *inteligencia artificial*. (Fuente: businessgoon.com/inteligencia-artificial-que-es/)

3.3.2 Cómo usamos la inteligencia artificial

En este caso, la IA de los *mutantes* es de tipo *máquina reactiva*. Es el tipo más básico de inteligencia artificial. Se basa en decisiones sobre el momento presente, pero no evoluciona. Hay dos tipos de enemigos que la usan: los *mutantes* y el *quinto paciente*, pero se comportan prácticamente igual.

Según nuestras acciones, los *mutantes* podrán ignorarnos, perseguirnos o atacarnos. Para ello hacemos uso de su *animator controller* con tres estados posibles: en reposo, correr o atacar. Aquí podemos ver de manera más precisa su comportamiento con una máquina de estados:

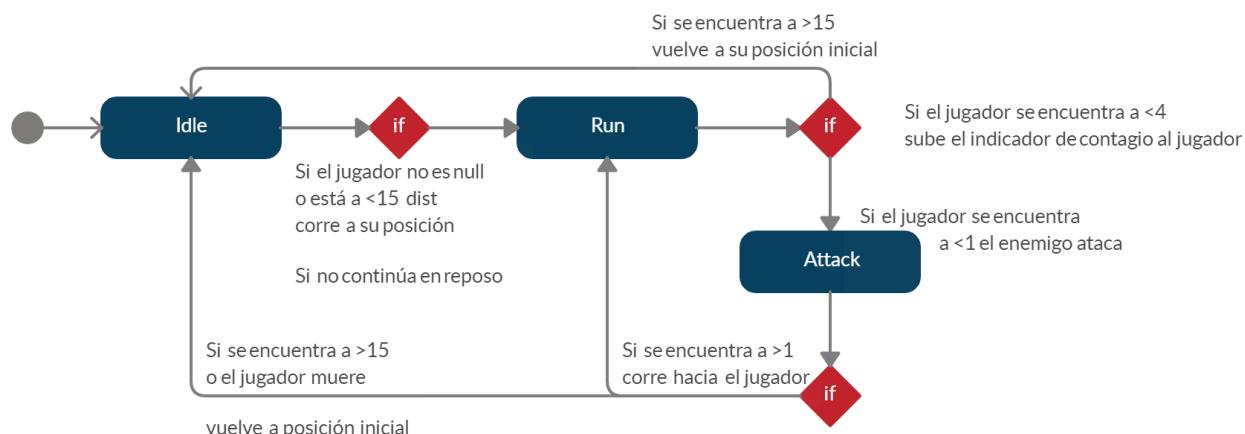


Figura 3.3. Máquina de estados de los enemigos.

3.4 DISEÑO DE PERSONAJES ELEGIBLES

El diseño visual de los personajes se ha basado en la temática del videojuego. Hemos creado y buscado personajes que encajen con el rol de estudiantes o profesores. Para ello, hemos hecho uso de dos herramientas de creación de personajes de Adobe: *Adobe Fuse* y *Mixamo* [30] (en el apartado 6.2.2 entraremos más en detalle).

La programación de los personajes elegibles está basada en dos scripts principales: *Maincontroller.cs*, se encargará del movimiento del jugador a través del movimiento de la cámara del jugador.

Health_and_Damage.cs se encarga de gestionar varias estadísticas del personaje, pero principalmente su *barra de mutación*, y cómo reacciona el jugador.

Los veremos en detalle en la sección de Implementación (apartado 5.2).

El diseño estructural de los personajes es parecido al de los *mutantes* en cuanto al movimiento, pero aquí dispondremos de más acciones y las controlaremos nosotros. Las animaciones de movimiento son gestionadas por el *animator controller*. En él, tenemos un *Blend Tree* llamado *Movement* que se encargará de generar la animación *Idle* (en reposo), *Walking* (caminando) y *Running* (corriendo) de manera que la transición sea suave y ordenada. Esto se hará acorde a la velocidad de movimiento del personaje. *Movement*, a su vez estará relacionado con las acciones saltar, apuntar, disparar y recibir daño. Para saber cuando podemos apuntar o disparar, crearemos la variable *canShoot* en el *animator controller* y la vincularemos por código con el script del arma *Syringe.cs*. Veamos el *animator controller*.

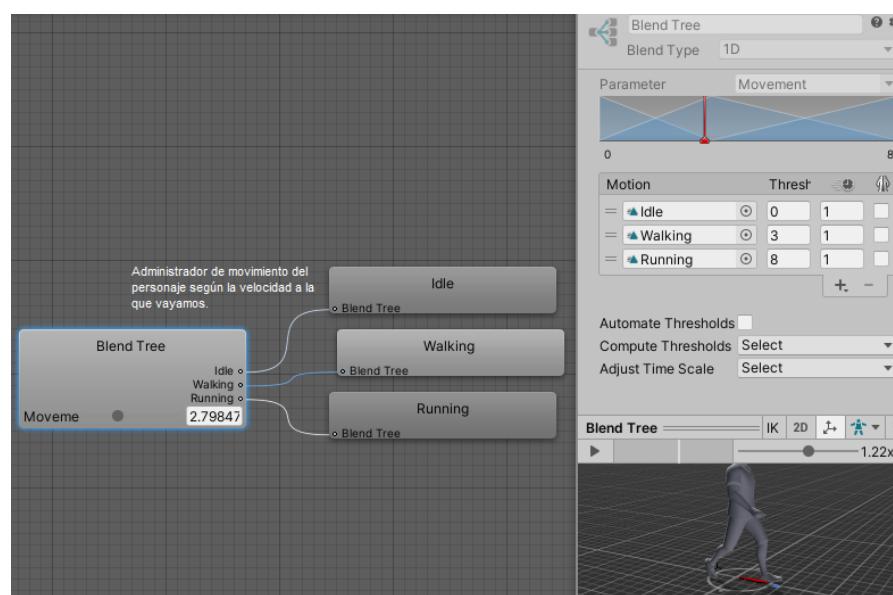


Figura 3.4. *Blend Tree Movement* del movimiento básico de los personajes seleccionables.

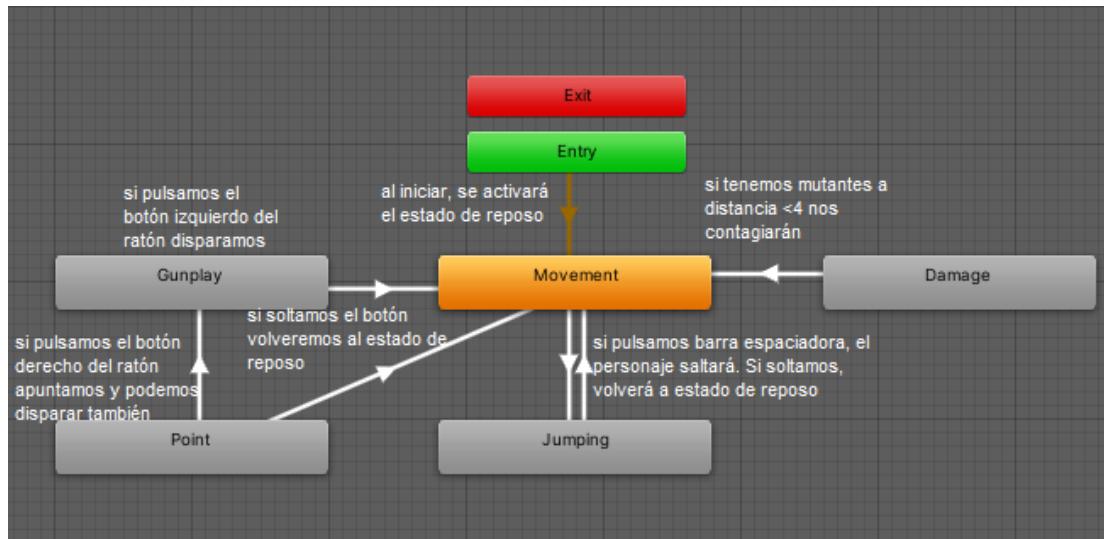


Figura 3.5. *Animator Controller* de las acciones de los personajes seleccionables.

3.5 DISEÑO DEL ARMA

Para diseñar el arma se quiso dar un enfoque totalmente distinto a un juego *shooter* estándar, pues en vez de matar, nosotros vamos a sanar al tratarse de una jeringuilla. Veremos todo esto en detalle en el apartado 6.3.

Cada personaje seleccionable llevará una jeringuilla en la mano con su respectivo script, de manera que podamos controlar cuándo podemos apuntar o disparar, o ambas.

Pensando en este novedoso modo de juego, también invertimos la barra de mutación del personaje. Normalmente, cuando te hacen daño, la barra merma de derecha a izquierda hasta llegar a 0. Aquí irás acumulando mutación de izquierda a derecha hasta llegar a 100. Sin embargo, cuando disparamos a los mutantes la barra irá de derecha a izquierda, ya que les quitaremos mutación.

4. Planificación

Para el correcto desarrollo del proyecto se ha llevado a cabo una planificación muy necesaria, ya que sin ella sería muy complicado llevar un seguimiento del progreso y marcar los límites del desarrollo.

El proyecto se ha comenzado con la intención de hacer un videojuego estándar con temática *survival horror* en las instalaciones del MIC, pero una vez se fue profundizando en el aprendizaje y la evolución del juego, se decidió ampliar el proyecto y abarcar más contenido (más personajes, *Modo Desafío*, misiones más completas, etc.). Esto permitió hacer un proyecto de mayor calidad y, a nivel académico, poder formarme más en el campo de la programación orientada a motores gráficos. El videojuego se comenzó a principios de Julio de 2020 y fue finalizado en Febrero de 2021.

Para cumplir con los plazos establecidos, la planificación se ha dividido en varias partes fundamentales que veremos en detalle.

4.1 PARTE 1 – Configuración del servidor de *Perforce*, aprendizaje y búsqueda de recursos

Antes de comenzar el proyecto, previamente, se realizó el documento de diseño o GDD en el que se incluyen los puntos más importantes para la conceptualización y el desarrollo del juego como son la descripción de los personajes, la jugabilidad, la ambientación, la historia y todo lo relacionado con el diseño de personajes y la interfaz.

Se estudió y elaboró cómo serían los escenarios y los diálogos en el juego. El GDD se encuentra en el anexo A de esta memoria.

A continuación, se hizo un estudio para integrar el videojuego en el servidor, permitiéndonos así, entre otras muchas posibilidades, poder administrar las diferentes versiones del juego con un control de versiones y en un futuro, implementar un sistema multijugador al mismo.

Después se realizó una búsqueda intensiva por internet de recursos o assets que conformarán los elementos del terreno, decoraciones, efectos en los disparos, etc.

Se utilizó *Adobe Fuse* para el desarrollo de los modelos 3D del personaje principal, los pacientes, médicos, personas salvadas y enemigos. En esta parte se ha previsto una duración de tres semanas para la realización de todas las tareas descritas.

4.2 PARTE 2 – Programación del videojuego

En esta parte se va a realizar toda la programación del videojuego. Se crearán todos los personajes y objetos necesarios, todas sus animaciones, diálogos y se les dará vida con programación. Finalmente se implementará el *modo multijugador*.

Esta parte ha sido la más costosa y duradera. La curva de aprendizaje fue muy buena, sin embargo, debido a errores y *bugs* durante su desarrollo, fue necesario más tiempo. Para esta parte se había previsto una duración de tres meses. Finalmente fueron cuatro meses.

Para la realización de esta parte se han llevado diferentes tareas que se describen a continuación:

- **Creación de un nivel provisional:** para ello se utilizaron elementos sencillos como un plano para representar el suelo y cubos o esferas para ver cómo reacciona el personaje con los enemigos y objetos.
- **Creación e importación de los objetos.** Esta fase llevó más tiempo del esperado. Creación de los personajes y sus texturas con la herramienta *Adobe Fuse CC*, a los que les dimos animación con *Mixamo*. Una vez tuvimos todas las animaciones de cada personaje, las importamos a *Unity3D*, creando su *character controller*, asignándole esqueleto y textura.
- Se creó todo un laboratorio, se hizo una búsqueda intensiva de materiales y objetos necesarios (armarios, escritorio, camillas, material de laboratorio, jeringuilla, etc.) y otros fueron creados junto a sus texturas. También se creó una escena final en la que aparece un dirigente de la Universidad y las personas salvadas.
- **Programación del personaje principal.** Como era el más necesario para testear, fue el primero en programarse. A través de scripts, se programó todo el movimiento, parámetros de vida (convertido), ejecución de animaciones, etc.
- **Programación de los enemigos.** Programamos su inteligencia artificial, la ejecución de sus animaciones, cuánto daño hacen y por donde pueden caminar.
- **Programación de los científicos.** Programación del científico de guardia y científico principal, así como los diálogos correspondientes.
- **Programación de la interfaz.** Creación de la interfaz del usuario y del juego. Barra de mutación, ventana de misión disponible, cantidad de munición actual, etc.
- **Programación del sistema de misiones.** Creación del sistema de misiones y su programación. Asignación de las misiones al personaje. Administración de los

diálogos y los mensajes de las notas. Objetivos de las misiones y cuándo se completan.

- **Programación de las notas y objetos interactuables.** Creación de la programación de las notas. Cómo interactúan con el usuario al estar cerca, poder agarrarlos o activarlos, etc.

4.3 PARTE 3 – Modelado del nivel

Al tener toda la programación realizada y funcionando correctamente, lo siguiente paso fue ponerse a detallar el nivel (escenario, entorno, iluminación, decoración). También se hicieron pruebas con escáneres 3D.

La duración de esta parte ha sido de tres semanas y se han realizado las siguientes tareas:

- **Creación del menú del juego.**
- Integración de toda la programación en el nivel.
- **Decoración y detalles del nivel.** Se han hecho uso de diferentes recursos para la decoración como sangre por las paredes y pasillos, cartas del diario, etc.
- **Iluminación y la incorporación de niebla.**
- **Inclusión de música y efectos de sonidos.**
- **Testeo de posibles errores con las colisiones de los objetos,** como por ejemplo las colisiones de los ataques del jugador contra los infectados.

Sobre estas fechas también se hizo un estudio de aplicaciones de escáner 3D. También se hicieron pruebas en el MIC y con ayuda de un escáner 3D tratamos de escanear al autor del proyecto y tratar de meterlo en el videojuego y darle vida. Hemos conseguido buenos resultados. Estos estudios seguirán llevándose a cabo.

4.4 PARTE 4 – Testeo y redacción de la memoria

Con toda la funcionalidad del proyecto terminada, en esta parte nos centramos en el testeo y corrección de errores y ajustes finales para el correcto funcionamiento del juego.

Cabe destacar que la memoria también fue avanzándose a la par que el proyecto para una mejor redacción del progreso.

La duración de esta parte ha sido de un tres meses repartidos en: 1 mes para el testeo y 2 meses para continuar la redacción de la memoria.

Para esta parte se han realizado las siguientes tareas:

- **Redacción de la memoria.**

- **Más decoración en los niveles.**
- **Animación del título del juego.**
- **Testeos de posibles errores en el juego.**
- **Ajustes del nivel del juego,** así como de la vida de los enemigos y de los daños.

A continuación, se muestra una tabla con las diferentes tareas realizadas y sus correspondientes fechas:

ACTIVIDAD	DESCRIPCIÓN	Fecha Inicio	Fecha Fin
Parte 1. Configuración del servidor, aprendizaje y búsqueda de recursos		03/07/2020	23/07/2020
Descripción del videojuego	Como se va a desarrollar el videojuego, historia, personajes, etc	03/07/2020	13/07/2020
Estudio del servidor	Enlazar el control de versiones Perforce	14/07/2020	17/07/2020
Aprendizaje de Assets y objetos 3D	Herramientas Adobe Fuse CC y Mixamo	18/07/2020	23/07/2020
Parte 2. Programación del videojuego		25/07/2020	12/11/2020
Creación de un nivel provisional	Escena de pruebas. Testeo de funcionalidad de los personajes 3D	25/07/2020	05/08/2020
Creación e importación de los objetos	Desarrollo de objetos con Adobe Fuse CC y Mixamo	06/08/2020	16/08/2020
Funcionamiento animaciones, texturas	Testeo de las animaciones, orden y solución de texturas	16/08/2020	21/08/2020
Programación del personaje principal	Programación de su movilidad, variables de vida, cámaras, etc	21/08/2020	11/09/2020
Programación enemigos	Programación de su inteligencia artificial y daño	11/09/2020	25/09/2020
Programación científicos	Asignación de dialogos, posición, funciones, etc	25/09/2020	30/09/2020
Programación de la interfaz	Componentes UI, cartel de misiones, notas, dialogos, municion, barras	30/09/2020	07/10/2020
Programación del sistema de misiones	misiones, diálogos, cuando completadas, items necesarios, etc	07/10/2020	27/10/2020
Programación de objetos interactuables	Jeringuilla, notas, contador de la luz, etc	27/10/2020	30/10/2020
Salvado de datos entre escenas y testeos	VARIABLES como la cantidad de mutación, munición, gameObjects adjuntos	30/10/2020	12/11/2020
Parte 3. Modelado del nivel		12/11/2020	19/12/2020
Creación del menú del juego	Interfaz de arranque, posibles modos, etc	12/11/2020	05/12/2020
Decoración y detalles del nivel	Recursos para la decoración: sangre, luces, laboratorio,etc	15/12/2020	17/12/2020
Illuminación y sonido	Testeo de luminosidad, concordar los efectos de sonido a los personajes	17/12/2020	19/12/2020
Parte 4. Testeo y redacción de la memoria		19/12/2020	15/02/2021
Redacción de la memoria	Continuación de la memoria (sobre todo la parte de desarrollo del proyecto)	19/12/2021	28/01/2021
Más decoración en los niveles	Mejorar efectos de luces, mayor nitidez de texturas, ambientación	03/02/2021	06/02/2021
Animación del título del juego	Decorar el menú de inicio, animaciones restantes, etc	06/12/2021	09/02/2021
Testeo de errores en el juego	Buscar bugs, probar modos de juego	09/12/2021	15/02/2021

Tabla 4.1. Cronograma de tareas realizadas y sus respectivas fechas.

5. Implementación

En este apartado se va a describir la implementación de las partes más importantes del proyecto (incluyendo sus scripts) que son:

- La configuración del servidor
- El funcionamiento del personaje
- Inteligencia artificial de los enemigos
- El sistema de misiones y diálogos (y brevemente el *GameManager.cs*)
- El funcionamiento del arma
- Funcionalidades extra

5.1 CONFIGURACIÓN DEL CONTROL DE VERSIONES PERFORCE

Para la configuración del servidor donde se ubicará todo el proyecto se ha hecho uso de un servidor *Ubuntu 18.04.2 Its 64 bits*. Este servidor se encuentra alojado en un *VPS (Servidor Virtual Privado)* y la instalación de *Perforce* se ha realizado a través del protocolo *ssh* (método seguro de acceso remoto de un ordenador a otro).

Con el objeto de ir actualizando continuamente el proyecto se hizo uso del control de versiones de *Perforce* [35]. A continuación se indican las instrucciones y comandos necesarios para configurarlo con el server:

- Descargamos la herramienta *Helix Core (P4D)* de *Perforce* de su sitio web:
<https://www.perforce.com/downloads/helix-core-p4d>
- Hacemos clic en la pestaña *Clients*.
- Buscamos la sección llamada *P4: CommandLine Client* y damos *continuar*.
- En el menú desplegable, seleccionamos nuestro sistema operativo y damos *continuar*.
- Abrimos una ventana de terminal.
- Hacemos que los archivos descargados sean ejecutables escribiendo:
- `chmod +x _downloads_path_ /p4 *`
- Movemos los archivos a una ruta de ejecución común:
- `sudo mv _downloads_path_ /p4* /usr/local/etc/perforce`

5.1.1 Funcionamiento de P4V Client

Una vez todo está configurado, procedemos a abrir *P4V*. Seleccionamos conexión remota y creamos un usuario para poder acceder. Luego pondremos la ip del servidor, nuestro usuario y el lugar de trabajo que hayamos seleccionado para salvaguardar el proyecto en el PC.

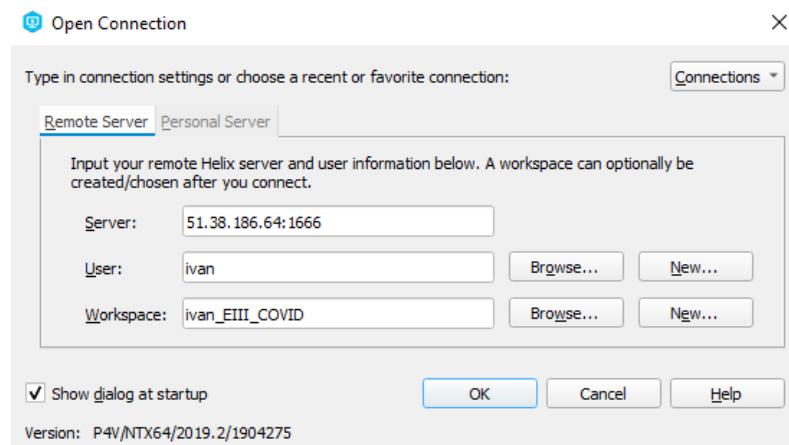


Figura 5.1. Ventana de *login* de conexión remota de *P4V*.

Una vez dentro, configuraremos el nombre del *Workspace* (debemos ponerle el mismo nombre en todos los PC, en este caso *prueba*) y la ruta raíz del mismo. Además, para vincular cliente y servidor debemos poner correctamente la ruta de *depot* (ruta donde se aloja el proyecto en el servidor). Una vez configurados ya estarán vinculadas ambas partes: en la pestaña *depot* tendremos el lado del servidor, y en *Workspace* el cliente.

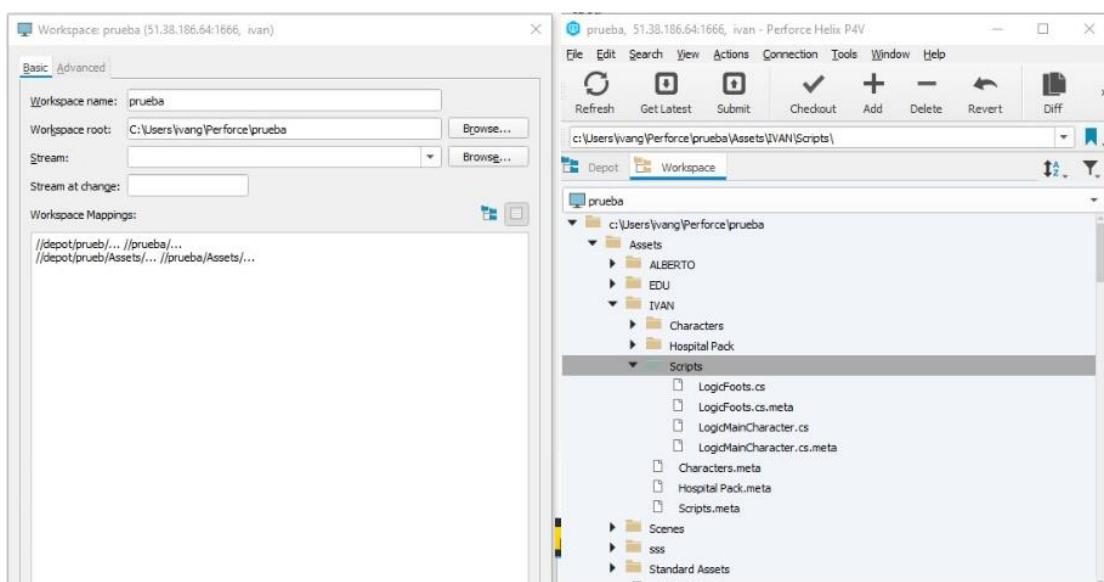


Figura 5.2. Configuración de la ruta del *Workspace* y directorio local del proyecto en *P4V*.

5.1.2 Interfaz de *Perforce* en *Unity3D*

Una vez configurado el cliente, entraremos en nuestro proyecto en *Unity* y nos conectaremos a *Perforce* desde dentro. Esto es posible gracias a su interfaz incorporada en el mismo motor gráfico. Nos pedirá nuestro usuario, contraseña, la dirección del servidor y el *Workspace* donde vayamos a alojar el proyecto.



Figura 5.3. Interfaz de enlace de *Perforce* en *Unity3D*.

Una vez conectados, habremos vinculado el proyecto con *Perforce*. Cuando queramos subir al *Workspace* nuestro trabajo actual, debemos ir a la pestaña *Version Control* y seleccionar que archivos queremos guardar. Para saber el estado en que se encuentran los archivos, *Version Control* dispone de una nomenclatura que nos dirá si el archivo sólo se encuentra localmente o si ya ha sido subido al *Workspace* y *checkeado* remotamente. También nos avisará si hay algún tipo de conflicto con el archivo, entre otras opciones.

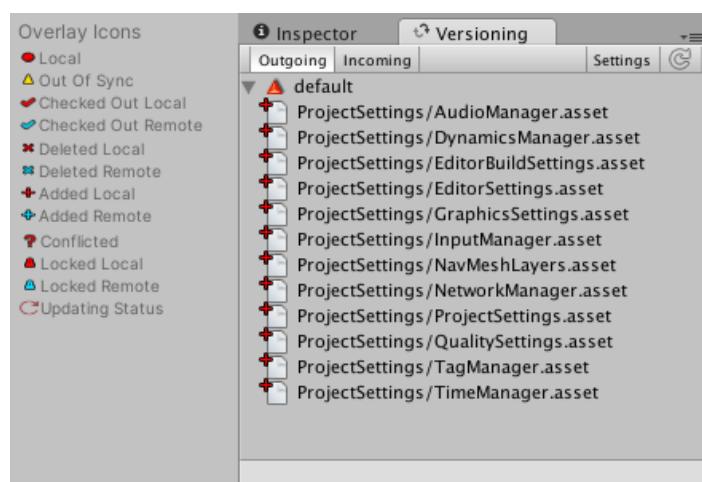


Figura 5.4. Nomenclatura y *Version Control* de la interfaz de *Perforce* en *Unity3D*.

5.1.3 Porqué usar *Perforce*

Perforce es un control de versiones intuitivo, y tiene numerosas ventajas a la hora de actualizar los datos. *Perforce* tiene un fuerte cliente GUI (PV4) que puede ejecutar casi todos los clientes de *Git*.

La razón más importante por la que usar *Perforce* es que, como acabamos de ver, se encuentra integrado en la interfaz gráfica de *Unity*. Esto nos permite realizar todas las acciones del control de versiones dentro de la propia aplicación (es decir, el editor), mostrando gráficos interactivos que indican el estado de cada fichero.

Dado que en el mundo de los controles de versiones no hay claros ganadores (algunos funcionan mejor con unas herramientas y otros con otras) nos basaremos en la experiencia obtenida con *Perforce* en *Unity3D* y consolidaremos datos con experiencias de otros usuarios. [31]

La posible mejor alternativa podría haber sido *Git*. Veamos una comparativa de los dos con esta tabla:

Comparación en Unity3D	Perforce	Git
Rapidez para repositorios grandes	Más rápido	Más lento
Siempre centralizado (todo ocurre en el servidor)	Si	No
Tiene un proxy de escritura directa	Si	No
Seguimiento de la integración por cada archivo	Si	No
Mejor soporte de Windows	Si	No
Comprobaciones muy flexibles	Si	No

Tabla 5.1. Ventajas y desventajas de *Perforce* sobre *Git* en *Unity3D*. Fuente:
<https://www.quora.com/What-are-some-advantages-of-Perforce-over-git>

Como podemos comprobar, tiene numerosas ventajas que otras herramientas no tienen.

En cuanto a inconvenientes tiene pocos, pero son importantes.

Como punto a favor, una vez usado, cabe destacar la rapidez con la que automáticamente se enlaza al servidor una vez se ejecuta *Unity*, lo que da muy buenas sensaciones y, por consiguiente, se puede decir que es una buena herramienta para desarrollar un juego.

Sin embargo, como crítica personal, reseñar un inconveniente grave. Una vez que se intenta actualizar los datos con el servidor y son subidos, *Perforce* hace un seguimiento archivo a archivo, no por *commit* realizado, y esto genera problemas cancelando toda una subida de ficheros porque un solo archivo esté incorrecto o no se encuentre. Esto se traduce en pérdida de tiempo volviendo a subir los archivos y actualizando.

5.2 EL PERSONAJE PRINCIPAL

En este apartado se va a explicar la implementación del movimiento del personaje, su vida y sus dos cámaras. Es importante destacar que se han creado varios personajes jugables y todos ellos cuentan con la misma programación. Para la implementación se ha hecho uso de dos scripts para el personaje y dos scripts para las cámaras.

5.2.1 Movimiento del personaje

La clase *MainController* es la encargada de toda la movilidad del personaje y animaciones. Es la clase más importante de todas dado que será la que nos permitirá movernos por el mundo.

Para que no se pierda nunca, haremos uso de un *singleton* que se encargará de que el objeto siempre sea el mismo, esté actualizado y no pierda ninguna referencia.

El bucle funciona así:

- si no hay un personaje en la escena, ponemos el personaje que arrastramos desde otras escenas.
- si ya hay un personaje en la escena, lo eliminamos y ponemos el que traemos desde otra escena para no perder los datos y referencias a objetos que trae consigo.

```
public static MainController instance;  
  
private void Awake()  
{  
    if (instance == null)  
        instance = this;  
    else if (instance != null)  
        Destroy(gameObject);  
    DontDestroyOnLoad(gameObject);  
}
```

Figura 5.5. Uso de un *singleton* en la clase *MainController.cs*

La función *Start* es la encargada de cargar el personaje principal, su controlador, su velocidad y sus animaciones. La función *Update* se utiliza para actualizar los datos cada *frame*, lo que nos permitirá una jugabilidad fluida.

A través de vectores de movimiento y de los parámetros configurados en *Unity*, obtenemos el movimiento del personaje. Una vez configurado, en el bucle *if* y *else* que vemos, podemos caminar o correr según que tecla pulsamos o dejemos de pulsar.

```
0 referencias
private void Start()
{
    controller = GetComponent<CharacterController>();
    anim = GetComponent<Animator>();
    auxSpeed = speed;
    main = GameObject.FindWithTag("MainCamera").GetComponent<Camera>();
}

0 referencias
void Update()
{
    if (main == null)
    {
        main = GameObject.FindWithTag("MainCamera").GetComponent<Camera>();
    }

    isGrounded = controller.isGrounded;
    horizontal = Input.GetAxis("Horizontal");
    vertical = Input.GetAxis("Vertical");
    input = new Vector3(horizontal, 0, vertical);
    input = Vector3.ClampMagnitude(input, 1);

    //Running - Walking
    if (Input.GetKey(KeyCode.LeftShift)){
        speed = 8;
        anim.SetFloat("Movement", input.magnitude * speed);
    }
    else
    {
        speed = auxSpeed;
        anim.SetFloat("Movement", input.magnitude * speed);
    }

    camDirection();
    movePlayer = input.x * camRight + input.z * camForward;

    movePlayer = movePlayer * speed;
    controller.transform.LookAt(controller.transform.position + movePlayer);
    SetGravity();
    PlayerSkills();

    controller.Move(movePlayer * Time.deltaTime);

    //if(Rigidbody.CompareTag("Scientist"));
}
```

Figura 5.6. Traza de código de la clase *MainController.cs*.

También está implementado el salto a través de un *collider* que detecta si el objeto está tocando el suelo o no. En la caída del salto también modificamos la velocidad de caída para más realismo.

5.2.2 Parámetros del personaje

En la clase *Health_and_Damage.cs* se implementa todo lo respectivo al daño hacia el jugador. Desde la Jerarquía de *Unity*, en el *gameObject Canvas* (lugar donde almacenamos

todos los *gameObject* de la interfaz del juego) se almacena la barra de cantidad de mutación del jugador y la utilizamos en esta clase.

En la función *RestarVida()* se implementa la acción de que un enemigo le quite vida, un tiempo de inmunidad y lo que sucede si la barra de vida llega al máximo. En este caso llamamos a la función *GameOver()*, se acaba el juego y se transforma en mutante. Aquí podemos ver cómo funciona:

```
0 referencias
private void Start()
{
    anim = GetComponent<Animator>();

    if(barraDeVida == null)
        barraDeVida = GameManager.manager.interfaceDisplay.transform.GetChild(3).gameObject.GetComponent<Image>();

    barraDeVida.fillAmount = 0;
}

//Function that controls the amount of life that enemies take from the character
2 referencias
public void RestarVida(float damage)
{
    if (barraDeVida == null)
        barraDeVida = GameManager.manager.interfaceDisplay.transform.GetChild(3).gameObject.GetComponent<Image>();

    converted = Mathf.Clamp(converted, 0, 100);

    //imagedamage = GetComponent<Image>();
    //var tempColor = imagedamage.color;

    if (!invencible && converted < 100)
    {
        converted += damage;
        anim.Play("Damage");
        //damageImage.color = damageColor;
        //tempColor.a += 0.2f;
        //imagedamage.color = tempColor;

        barraDeVida.fillAmount = converted / 100;
        StartCoroutine(Invulnerabilidad());
        StartCoroutine(FrenarVelocidad());
        if (barraDeVida.fillAmount >= 1) GameOver();
    }
    //damageImage.color = Color.Lerp(damageImage.color, Color.clear, colorSmoothing * Time.deltaTime);
}
```

Figura 5.7. Traza de código de la clase *Health_and_Damage().cs*

```
//End of game
1 referencia
void GameOver()
{
    Debug.Log("Has sido convertido. GAME OVER.");
    endGameText.SetActive(true);
    //GameObject mutant = GameObject.Find("/Adds/ADDSmutantpersons");
    Instantiate(mutant, gameObject.transform.position, Quaternion.identity);
    Destroy(gameObject);
    Time.timeScale = 0;
}
```

Figura 5.8. Traza de código de la clase *GameOver().cs*

5.3 INTELIGENCIA ARTIFICIAL DE LOS ENEMIGOS

Los enemigos, también llamados *mutantes*, son otra parte esencial del videojuego. Son los encargados de ponerle dificultad al éxito del jugador. Su programación se compone de dos

scripts, *IAEnemies.cs* y *health_and_damageIA.cs*.

5.3.1 Inteligencia artificial de los enemigos

IAEnemies.cs se encarga de toda la inteligencia artificial del personaje. Todo se detecta a través de vectores. Se calcula la distancia al jugador. Si entra dentro de su rango de visión el enemigo corre hacia el jugador y ataca. Hemos representado el diagrama de estados en el punto 3.3 Diseño estructural de la IA (Figura 3.1). Ahora veámoslo en código:

```
private void Update()
{
    if (player == null)
    {
        GetComponent<health_and_damageIA>().dosed = false;
        agent.SetDestination(initialPosition);
    }
    //For default our objective will be always our initial position
    target = initialPosition;
    //But if the distance to the player is less than the vision radius, the objective will be the him
    float dist = Vector3.Distance(player.transform.position, transform.position);
    if (dist < visionRadius)
    {
        agent.speed = speedInitial;
        target = player.transform.position;
        agent.SetDestination(target);
        transform.LookAt(target);
        anim.SetBool("detected", true);
    } else
    {
        agent.SetDestination(initialPosition);
    }

    //si le damos con una dosis a algun enemigo anim.detected true y nos sigue hasta dist 8
    if (GetComponent<health_and_damageIA>().dosed == true)
    {
        agent.speed = speedInitial;
        target = player.transform.position;
        agent.SetDestination(target);
        transform.LookAt(target);
        anim.SetBool("detected", true);
        if (dist > 15)
        {
            GetComponent<health_and_damageIA>().dosed = false;
            agent.SetDestination(initialPosition);
        }
    }

    if(dist < 1 )
    {
        anim.Play("Zombie Attack");
        player.GetComponent<Health_and_Damage>().RestarVida(damage);
    }
}
```

Figura 5.9. Traza de código de la clase *IAEnemies()*.*cs*

5.3.2 Daño y transformación de los enemigos

health_and_damageIA.cs se encarga de las funciones relativas al daño que le causan las dosis y qué le ocurre cuando su barra de mutación llega a 0.

La función *TakeDose()* es llamada cuando le disparamos dosis al enemigo, restándole mutación. Una vez que su barra llega a 0 se llama a *personSaved()*, que vuelve a la persona del edificio a la normalidad. Veámoslo en código:

```
2 referencias
public void TakeDose (float amount)
{
    slider.gameObject.SetActive(true);
    if (slider.value >= 0f)
    {
        dosed = true;
        slider.value -= amount;
        Debug.Log(slider.value);
    }

    if (slider.value <= 0) personSaved();
}
1 referencia
public void personSaved()
{
    FindObjectOfType<savedPersonsArray>().choosePerson(gameObject);
    Destroy(gameObject);
}
```

Figura 5.10. Funciones *TakeDose()* y *personSaved()* de la clase *health_and_damageIA.cs*

5.4 EL SISTEMA DE MISIONES Y DIÁLOGOS

Esta fase fue toda una sorpresa, tenía muchas ganas de hacerla, sin embargo lo esperaba más sencillo de lo que fue, a continuación lo vemos detalladamente.

El sistema de misiones era totalmente necesario en un modo historia. Lo que tenía claro es que debía ser una cadena de misiones, sin posibilidad de retorno o de que haya más misiones activas simultáneamente.

Para esta fase se usaron varios scripts:

Por una parte, para la detección de las misiones *DialogueTrigger.cs* y para el desarrollo de los diálogos *DialogueManager.cs*.

Por otra parte, para la elaboración y distribución de las misiones se usaron *QuestGiver.cs* y *GameManager.cs*, y para el tipo de misión, completar o establecer objetivos *Quest.cs* y *QuestGoal.cs*.

5.4.1 Detección de misiones y desarrollo de diálogos

En esta fase desarrollamos toda la funcionalidad para obtener misiones. La iteración del jugador con los distintos componentes o personajes del juego, también llamados *NPC* (*non player character*), los cuales poseen o completan una misión, es llevada a cabo por el script *DialogueTrigger.cs*.

Para ello, en la Jerarquía de *Unity*, a los objetos o *NPC* que porten misiones se les asigna este Script. Así estarán constantemente calculando la distancia al jugador mediante vectores.

Funcionamiento: Si el jugador se encuentra muy cerca se activará un botón de hablar con el *NPC*, sino desaparecerá. Si se acepta, se iniciará el diálogo mediante la función *TriggerDialogue()* en la que cambiamos a la cámara del *NPC* y llamaremos a *DialogueManager.cs*. Veámoslo en código:

```
0 referencias
private void Update()
{
    float dist = Vector3.Distance(player.transform.position, transform.position);

    if (dist > 1)
    {
        cam.moveCam = true;
        buttonTalk.gameObject.SetActive(false);

        //Cursor.lockState = CursorLockMode.Locked;
        //Cursor.visible = false;
    }

    if (dist < 1)
    {
        cam.moveCam = false; //revisar, igual no necesario
        buttonTalk.gameObject.SetActive(true);

        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
    }
    //FindObjectOfType<DialogueManager>().dialogueOrNote.gameObject.activeInHierarchy

    if (dist < 1.5 && NPCCam.GetComponentInParent<DialogueManager>().dialogueOrNote.gameObject.activeInHierarchy)
        buttonTalk.gameObject.SetActive(false);
}

0 referencias
public void TriggerDialogue()
{
    buttonTalk.gameObject.SetActive(false);

    //switch off the rest of interface
    //GameObject.Find("Vida y munición").SetActive(false);

    //switch off the main camera, switch on the NPC camera
    NPCCam.gameObject.SetActive(true);
    FindObjectOfType<CameraFollow>().enabled = false;

    //switch off the player and his control
    FindObjectOfType<MainController>().enabled = false;
    player.SetActive(false);

    NPCCam.GetComponentInParent<DialogueManager>().StartDialogue(dialogue, gameObject, NPCCam);
    ActivarWindow();
}
```

Figura 5.11. Traza de código de la clase *DialogueTrigger.cs*

DialogueManager.cs se encarga de gestionar los diálogos. Cabe destacar que gestiona tanto los diálogos de los *NPC* como los mensajes de las Notas. Se distinguirán por el *gameObject* que por cercanía al jugador se active.

Podemos dividir el script por funciones:

StartDialogue().cs se encarga de asignar títulos, nombre y descripciones a los *NPC* o *gameObject* dueños de esa misión. Después inicia los diálogos y los pone en cola de manera que vayan en orden. El diálogo actual se lo pasamos a *DisplayNextSentence()*.

```
1 referencia
public void StartDialogue(Dialogue dialogue, GameObject NPC, Camera NPCcam)
{
    //GameObject.FindGameObjectWithTag("mainInterface").SetActive(false);

    thisNPC = NPC;
    thisNPCcam = NPCcam;

    dialogueOrNote.SetActive(true);
    continueDialogue.SetActive(true);
    acceptQuest.SetActive(false);
    nameText.text = dialogue.name;
    extraInformation.text = dialogue.extraInformation;

    sentences.Clear();

    foreach (string sentence in dialogue.sentences)
    {
        sentences.Enqueue(sentence);
    }
    DisplayNextSentence();
}
```

Figura 5.12. Función *StartDialogue()* de la clase *DialogueManager().cs*

DisplayNextSentence() se encarga de enseñar por pantalla cada diálogo y desencolarlo para dar entrada al siguiente diálogo. En el último diálogo se llama a la función *EndDialogue()*.

```
1 referencia
public void DisplayNextSentence()
{
    if (sentences.Count == 1)
    {
        acceptQuest.SetActive(true);
        continueDialogue.SetActive(false);
    }

    //FindObjectOfType<DialogueTrigger>().anim.Play("Talk");
    string sentence = sentences.Dequeue();
    description.text = sentence;

    //if we activate the quest, we finish the dialogue
    acceptQuest.GetComponent<Button>().onClick.AddListener(() => EndDialogue());
}
```

Figura 5.13. Función *DisplayNextSentence()* de la clase *DialogueManager().cs*

Resulta importante aclarar que estos diálogos son escritos directamente en el *NPC* desde la Jerarquía. Esto se debe a un *[Serializable]* en la clase *Dialogue.cs*. Aquí podemos ver una traza del diálogo de un *NPC*:

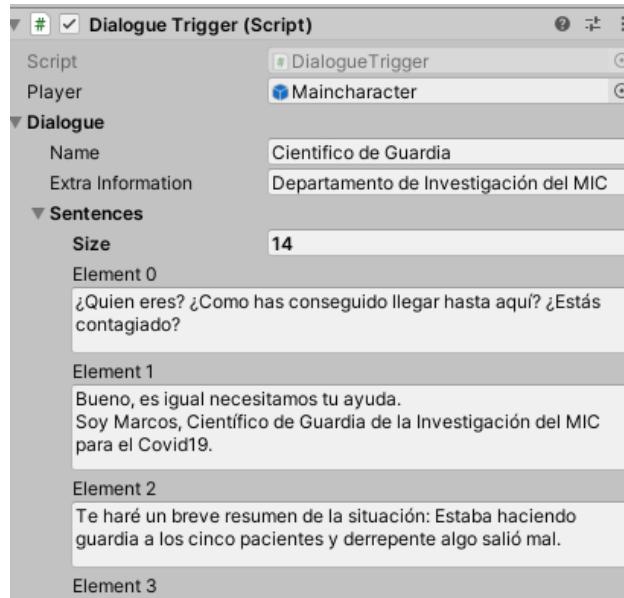


Figura 5.14. Foto de algunos de los diálogos del Científico de Guardia.

EndDialogue() se encarga de finalizar el diálogo y reactivar la cámara del personaje. Finalmente acepta la misión del jugador llamando a *QuestGiver()*. Veámoslo en código:

```
1 referencia
public void EndDialogue()
{
    FindObjectOfType<DialogueTrigger>().player.SetActive(true);
    dialogueOrNote.gameObject.SetActive(false);
    FindObjectOfType<MainController>().enabled = true;
    FindObjectOfType<CameraFollow>().enabled = true;

    thisNPCcam.enabled = false;
    FindObjectOfType<QuestGiver>().AcceptQuest(thisNPC);

    //GameObject.FindGameObjectWithTag("mainInterface").SetActive(true);
}
```

Figura 5.15. Función *EndDialogue()* de la clase *DialogueManager().cs*

5.4.2 Elaboración y distribución de las misiones

Una vez hemos hablado con el *NPC* correcto, toca asignar la misión correspondiente de la cadena de misiones. De esto (entre otras cosas) se encarga *QuestGiver.cs*

QuestGiver.cs es el contenedor y repartidor de las misiones. Expliquemos su funcionamiento:

Dado que esta clase debe mantener el orden de la cadena de misiones en todo momento, para no descuadrar valores entre escenas se le ha asignado otro *Singleton* (como ya vimos en *MainController().cs*).

Comienza en la función *Start()* inicializando todas las misiones y dándoles formato con la clase *Quest()*.

Una vez inicializadas, con *chainQuests()* añadimos los detalles de la misión, y asignamos el tipo de misión con la clase *QuestGoal()*, como veremos a continuación.

Cabe aclarar que los detalles de la misión son un recordatorio para guiar al jugador en lo que debe hacer en cada misión. Esta información aparecerá en una ventana amarilla que pone “*Misión activa*” en tiempo de ejecución. Veámoslo en código:

```
private void Awake()
{
    if (instance == null)
    {
        instance = this;
        DontDestroyOnLoad(gameObject);
    }
    else if (instance != null)
        Destroy(gameObject);
}
0 referencias
public void Start()
{
    //GameManager.manager.questWindow.SetActive(false);

    quests = new Quest[6];
    for (int i = 0; i < quests.Length; i++)
    {
        quests[i] = new Quest();
        quests[i].goal = new QuestGoal();
    }
    chainQuests();
}

1 referencia
public void chainQuests()
{
    quests[0].npc = "Note 1";
    quests[0].title = "Nota 1 del Científico Principal";
    quests[0].description = "Busca el laboratorio de investigación.";
    quests[0].goal.goalType = QuestGoal.GoalType.Find;

    quests[0].activateAdd = false;
    quests[0].activateSyringe = false;

    quests[1].npc = "Note 2";
    quests[1].title = "Nota 2 del Científico Principal";
    quests[1].description = "Habla con el cuarto paciente.";
    quests[1].goal.goalType = QuestGoal.GoalType.Find;
```

Figura 5.16. Traza de la clase *QuestGiver().cs*

Finalmente, mediante la función *AcceptQuest()* mantenemos ordenada toda la cadena de misiones y no dejamos lugar a dos misiones activas a la vez. Funciona así:

- Creamos el caso particular inicial:
 - Si la primera misión (la actual) no está completada y el *NPC* que quiere activar misión corresponde con el *NPC* dueño de la misión, esta es la misión buscada.
 - Si la primera misión está completada y el *NPC* no corresponde, recorremos el bucle de misiones y vamos a la siguiente misión en búsqueda de una incompleta y que coincida el *NPC*. La misión es encontrar la siguiente misión sin completar para continuar la cadena de misiones.
- El caso general es casi igual:
 - Si la anterior misión está acabada, es nuestra misión. Sino, vamos a la siguiente. Una vez encontramos la misión, llamamos a *GameManager.cs* (clase encargada de mantener a salvo los datos entre escenas), que es donde se alojan las misiones, y rellenamos los detalles de la misma. Veámoslo en código:

```
1 referencia
public void AcceptQuest(GameObject NPC)
{
    if (!quests[0].complete && quests[0].npc == NPC.name)
    {
        GameManager.manager.questWindow.SetActive(true);
        GameManager.manager.titleBox.text = ""+quests[0].title;
        GameManager.manager.descriptionBox.text = ""+ quests[0].description;

        quests[0].isActive = true;
        GameManager.manager.quest = quests[0];
        GameManager.manager.active = false;
    }
    else
    {
        for (int i = 1; i < quests.Length; i++)
        {
            //si la misión anterior ya se completó y la misión actual le corresponde a este NPC
            if ((quests[i - 1].complete) && (quests[i].npc == NPC.name))
            {
                GameManager.manager.active = false;
                GameManager.manager.questWindow.SetActive(true);
                GameManager.manager.titleBox.text = ""+ quests[i].title;
                GameManager.manager.descriptionBox.text = ""+ quests[i].description;

                quests[i].isActive = true;
                GameManager.manager.quest = quests[i];

                return;
            }
        }
    }
}
```

Figura 5.17. Método *AcceptQuest()* de la clase *QuestGiver.cs*

La clase *GameManager()* como bien hemos dicho, es “*la madre de todos los datos*”. Para ello, los datos son almacenados en un objeto de esta clase, de tipo *static*. De esta manera, no se puede modificar su valor. *GameManager()* también lleva asignado un *Singleton* para traspasar escenas sin destruirse.

En ella se llevan a cargo varias funciones. Cabe destacar dos:

loadQuestBox(). Se encarga de cargar los datos de las misiones que acabamos de ver en *AcceptQuest()* de *QuestGiver()*.

```
1 referencia
public void ...loadQuestBox()
{
    if (questWindow == null)
    {
        questWindow = GameObject.Find("/Canvas/RememberQuest");
        titleBox = GameObject.FindGameObjectWithTag("titleQuest").GetComponent<Text>();
        descriptionBox = GameObject.FindGameObjectWithTag("descriptionQuest").GetComponent<Text>();
        titleBox.text = quest.title;
        descriptionBox.text = quest.description;
    }
}
```

Figura 5.18. Método *loadQuestBox()* de la clase *GameManager.cs*

La función ***loadElements()*** se encarga de activar o desactivar objetos en el nivel según el progreso que llevemos en la cadena de misiones. Veámoslo en código:

```
1 referencia
public void ...loadElements()
{
    if (interfaceDisplay == null)
    {
        completeMision = GameObject.Find("/Canvas/ButtonsTalk/textComplete");
        interfaceDisplay = GameObject.Find("/Canvas/LifeAndMunition");
    }

    if (theEnemies == null)
        theEnemies = GameObject.FindGameObjectWithTag("Enemies");

    if (quest == null)
    {
        theEnemies.SetActive(false);
        theSyringe.transform.GetChild(0).gameObject.SetActive(false);
    }

    //we arrived to the mission that activate adds?
    if (quest.activateAdds)
    {
        interfaceDisplay.transform.GetChild(2).gameObject.GetComponent<Image>().enabled = true;
        interfaceDisplay.transform.GetChild(3).gameObject.GetComponent<Image>().enabled = true;
        interfaceDisplay.transform.GetChild(4).gameObject.GetComponent<Image>().enabled = true;
        theEnemies.SetActive(true);
    }
    else
    {
        interfaceDisplay.transform.GetChild(2).gameObject.GetComponent<Image>().enabled = false;
        interfaceDisplay.transform.GetChild(3).gameObject.GetComponent<Image>().enabled = false;
    }
}
```

Figura 5.19. Método *loadElements()* de la clase *GameManager.cs*

5.5 EL FUNCIONAMIENTO DEL ARMA

El arma es una jeringuilla bastante grande que por munición dispara *dosis de vacunas*. Estas dosis, tanto al disparar como al impactar en un *mutante* no estallan como lo haría una bala. Se esparce una especie de neblina de color azul verdoso a modo de vacuna. La jeringuilla es el “arma” que nos permitirá sanar a los mutados y que se recuperen, volviendo a ser estudiantes o profesores del MIC.

En el *Modo Historia*, esta arma no estará disponible hasta la *cuarta misión* de la cadena de misiones, en el momento en que consigues vacunas para curar a los mutados.

Para el *Modo Desafío* se ha modificado levemente el script de manera que la tendremos disponible desde el inicio y con munición ilimitada.

Su funcionamiento es sencillo: disponemos de una cantidad de dosis de vacunas. Si las acabamos y aún no hemos curado la cantidad de *mutantes* requeridos, perdemos y deberemos reintentar la misión.

La jeringuilla tiene su propia cámara, de estilo *primera persona*, que se activará en lugar de la *MainCamera* cuando se dispara. Este control de cámaras lo gestiona la clase *changingCameras()*. Veámoslo en código:

```
2 referencias
public void ThirdPersonPoint() //Apuntando sin mirilla
{
    mira.SetActive(true);

    fpsCam.GetComponent<AudioListener>().enabled = false;
}

4 referencias
public void Point() //Apuntando con mirilla
{
    mira.SetActive(true);
    fpsCam.enabled = true;
    tpsCam.enabled = false;

    fpsCam.GetComponent<AudioListener>().enabled = true;
}

2 referencias
public void Dispoint() //Desapuntando
{
    mira.SetActive(false);
    fpsCam.enabled = false;
    tpsCam.enabled = true;

    fpsCam.GetComponent<AudioListener>().enabled = false;
}
```

Figura 5.20. Métodos de la clase *changingCameras.cs*

Toda la lógica de la jeringuilla la lleva la clase *syringe.cs*. Una vez conseguimos el arma, entraremos en la función *Update()*, que estará continuamente actualizando si podemos disparar o no. Para apuntar usaremos el botón derecho del mouse y para disparar el botón izquierdo. Podemos hacer ambas acciones simultáneamente. Para desapuntar basta con soltar los botones.

En código, para disparar, analizaremos si ha pasado el tiempo necesario entre dosis y estamos pulsando el ratón. Veámoslo en código:

```
referencias
void Update()
{
    if(fpSCam == null)
        fpSCam = GameObject.Find("/CameraBase/GunCamera").GetComponent<Camera>();

    if(ammoDisplay==null)
        ammoDisplay = GameManager.manager.interfaceDisplay.transform.GetChild(0).gameObject.GetComponent<Text>();

    ammoDisplay.text = GameManager.manager.munitionDose.ToString();

    //apuntar y disparar
    if (canShoot)
    {
        if (Input.GetButton("Fire1") && Input.GetButton("Fire2") && Time.time >= nextTimeToFire && GameManager.manager.munitionDose > 0)
        {
            player.anim.Play("Gunplay");
            cameras.Point();
            Shooting();
        }
        //disparar
        else if (Input.GetButton("Fire1") && Time.time >= nextTimeToFire && GameManager.manager.munitionDose > 0)
        {
            player.anim.Play("Gunplay");
            cameras.ThirdPersonPoint();
            Shooting();
        }
        //apuntar
        else if (Input.GetButton("Fire2"))
        {
            player.anim.Play("Point");
            cameras.Point();
        }
        else cameras.Dispoint();
    }
    //FIN de bucle de disparo
}
```

Figura 5.21. Método *Update()* de la clase *syringe.cs*

5.6 FUNCIONALIDADES EXTRA

Como ya mencionamos, hay más funcionalidades que forman parte del proyecto. En esta sección explicaremos algunos scripts que merece la pena mencionar: el cronómetro, el guardado y carga del personaje seleccionado en el menú y el uso de *colliders* para la detección del personaje y obtención de objetos.

5.6.1 El cronómetro

El cronómetro es una herramienta muy interesante en un juego con esta temática. Aporta al jugador sensaciones como agobio, prisa y mayor atención en el objetivo que debe cumplir.

En el *Modo Historia*, el cronómetro es usado como tiempo para completar la cuarta misión.

En el modo desafío estará presente en los tres niveles de dificultad.

Para poder iniciarla en cualquier momento y desde cualquier escena, llamamos a su función *BeginChronometer()*, la cual iniciará la subrutina *TimerEnum()*. Esta subrutina se encargará de llamar a *FormatText()* y no saldrá de la subrutina hasta que se acabe el tiempo. Esta función es la encargada de transformar los segundos en minutos. que iniciará la cuenta atrás y no dejará que termine hasta no acabarse el tiempo.

Veámoslo en código:

```
        Debug.Log("cronometro a correr");
        StartCoroutine(TimerEnum());
    }

    1 referencia
    private IEnumerator TimerEnum()
    {
        timer = startTime;

        do
        {
            timer -= Time.deltaTime;
            FormatText();
            yield return null;
        }
        while (timer > 0);

        //Se acabó el tiempo GAME OVER.
        timeOver.SetActive(true);
        Time.timeScale = 0;

        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
    }

    1 referencia
    private void FormatText()
    {
        int minutes = (int)(timer / 60) % 60;
        int seconds = (int)(timer % 60);

        timerText.text = "";
        if (minutes >= 0)
        {
            if (minutes < 10)
                timerText.text += "0" + minutes + ":" ;
            else timerText.text += minutes + ":" ;
        }
        if (seconds >= 0)
        {
            if (seconds < 10)
                timerText.text += "0" + seconds;
            else timerText.text += seconds;
        }
    }
}
```

Figura 5.22. Métodos *TimerEnum()* y *FormatText()* de la clase *CounDownTimer.cs*

5.6.2 Guardado y carga de personajes

Para darle más variedad y diversión al juego, se crearon varios personajes jugables. Estos pueden ser elegidos desde el menú del juego.

Para guardar el personaje seleccionado y cargarlo en la escena correspondiente se usaron los scripts *chooseCharacter()* y *loadCharacter()*.

- ***ChooseCharacter()*** se encarga de la selección y guardado del personaje elegido.

Tenemos un array de personajes el cual será utilizado a través del índice de cada personaje. Para ello, contamos con las funciones *NextCharacter()* y *PreviousCharacter()*, que corresponden a la flecha izquierda y derecha en la interfaz de selección de personaje.

Por último, al dar a aceptar iniciaremos la función *StartGame()* encargada de guardar en memoria el índice del personaje seleccionado.

```

        characters[i] = gameobject.transform.GetChild(1).gameObject;
        characters[i].SetActive(false);
    }
    selectedCharacter = 0;

    labelNameMultiStory.text = characters[selectedCharacter].name;
    labelNameStory.text = characters[selectedCharacter].name;
    labelNameMultiChallenge.text = characters[selectedCharacter].name;
    labelNameChallenge.text = characters[selectedCharacter].name;
}

0 referencias
public void NextCharacter()
{
    characters[selectedCharacter].SetActive(false);
    selectedCharacter = (selectedCharacter + 1) % characters.Length;
    characters[selectedCharacter].SetActive(true);

    labelNameMultiStory.text = characters[selectedCharacter].name;
    labelNameStory.text = characters[selectedCharacter].name;
    labelNameMultiChallenge.text = characters[selectedCharacter].name;
    labelNameChallenge.text = characters[selectedCharacter].name;
}

0 referencias
public void PreviousCharacter()
{
    characters[selectedCharacter].SetActive(false);
    selectedCharacter--;
    if(selectedCharacter < 0)
    {
        selectedCharacter += characters.Length;
    }
    characters[selectedCharacter].SetActive(true);

    labelNameMultiStory.text = characters[selectedCharacter].name;
    labelNameStory.text = characters[selectedCharacter].name;
    labelNameMultiChallenge.text = characters[selectedCharacter].name;
    labelNameChallenge.text = characters[selectedCharacter].name;
}

0 referencias
public void StartGame()
{
    PlayerPrefs.SetInt("selectedCharacter", selectedCharacter);
}

```

Figura 5.23. Clase *ChooseCharacter()*

LoadCharacter() es la clase encargada de cargar en la nueva escena, el personaje seleccionado. Teniendo el mismo array de personajes que la clase *ChooseCharacter()*, cargamos el índice de personaje guardado y activamos sólo ese personaje.

Cabe recalcar que para no perder el personaje seleccionado entre escenas de ambos modos, se usó un *singleton*.

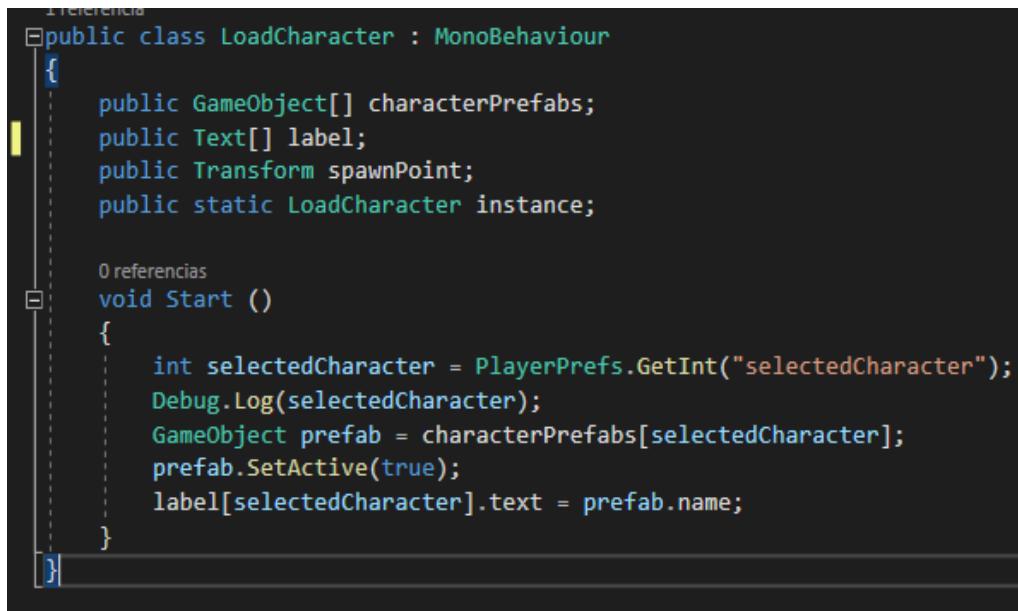


Figura 5.24. Clase *LoadCharacter()*

5.6.3 Colliders para recoger objetos

El uso de colliders para detectar la ubicación del personaje es una herramienta muy útil para obtener objetos, activar un teletransporte que cambie de escena, o para que la cámara del personaje no atraviese las paredes.

En este caso, veremos cómo se usó un collider para la activación de la luz en la misión *Sin luz y sin defensas*.

Para activar la luz del edificio, primero debemos llegar a la zona de suministro eléctrico. Una vez allí, pulsando un botón se encenderán todas las luces y completaremos esa parte de la misión. De esto se encargará el script *CollElectricity()*.

A través de la función *OnTriggerStay()* detectaremos cuándo entra el personaje en el collider del suministro eléctrico y con la función *OnTriggerExit()* cuándo sale. Si entra, pondremos visible un botón para activar la corriente y gracias a la función *AddListener()* sabremos cuando le damos click. Al darle click, aumentaremos en 1 el número de ítems recogidos de la misión. En el cuadro de misión activa, aparecerá así: “1/1 Activa la corriente del MIC”.

Desde la misma interfaz, dentro del botón haremos llamada al método *turnOn()*, lo que hará que se activen los *gameObjects* de las luces.

Veámoslo en código:

```
public class CollElectricity : MonoBehaviour
{
    public GameObject buttonElectricity;
    public GameObject lights;
    // Referencias
    private void Start()
    {
        lights = GameObject.FindGameObjectWithTag("Lights");
    }
    // Referencias
    public void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Player")
        {
            buttonElectricity.SetActive(true);
        }

        buttonElectricity.GetComponent<Button>().onClick.AddListener(() => GameManager.manager.quest.goal.ItemCollected());
        Debug.Log(GameManager.manager.quest.goal.currentAmount);
    }

    // Referencias
    public void OnTriggerExit(Collider other)
    {
        buttonElectricity.SetActive(false);
    }

    // Referencias
    public void turnOn()
    {
        GameManager.manager.quest.goal.ItemCollected();
        Debug.Log("Corriente reestablecida.");
        buttonElectricity.SetActive(false);
    }
}
```

Figura 5.25. Clase *CollElectricity()*

5.6.4 Vacunas e Incrementos de dosis del Modo Desafío

Para hacer más divertido e interesante el *Modo Desafío* se implementó la posibilidad de curarse y hacer más daño con estos dos objetos. Su programación inicial es sencilla, cuando comienza la partida les damos una posición aleatoria por todo el mapa. Ahora veamos como funcionan al recogerlos:

- Si nos encontramos en el caso de las vacunas, una vez entramos en el *collider* del objeto, llamaremos al método *SumarVida()* alojado en la clase *Health_and_Damage()* del personaje, así quitándonos un 30% de la barra de mutación. Una vez hecho, destruiremos el objeto.
- Si nos encontramos en el caso de los incrementos de dosis será casi igual, pues accederemos al método *incrementDose()* que se aloja en la clase *syringeChallenge()* de la jeringuilla. Dicho método activará una imagen de la bonificación del daño para que el jugador sepa cuanto tiempo estará activo. Para contar el tiempo se disparará una subrutina que durará 15 segundos y luego desactiva la imagen.

Veámoslo en código:

```
public class VacuneDose : MonoBehaviour
{
    public float dose = 40f;
    float x, y, z;
    Vector3 pos;

    0 referencias
    void Start()
    {
        x = Random.Range(-100, 140);
        y = 1f;
        z = Random.Range(-100, 140);
        pos = new Vector3(x, y, z);
        transform.position = pos;
    }

    0 referencias
    public void OnTriggerEnter(Collider other)
    {
        if (other.tag == "Player")
        {
            other.GetComponent<Health_and_Damage>().SumarVida(dose);
            Destroy(gameObject);
        }
    }
}
```

Figura 5.26. Clase *VacuneDose()*

6. Diseño conceptual del juego

6.1 DISEÑO DE LOS CONTROLES

En todo momento sólo podremos controlar un personaje de todos los seleccionables en el menú. Podremos moverlo en todas direcciones utilizando las teclas WASD. Cabe destacar que con la “*barra espaciadora*” podremos saltar y con la tecla *shift* podremos correr.

Si el jugador pulsa la tecla *Esc*, irá al *menú de pausa*.



Figura 6.1. Teclas de control del teclado. (Fuente: <https://www.solotodo.cl/products/52270-xtrike-me-kb-301-kb-301>)

Podrá controlar libremente la cámara de visión del jugador con el uso del mouse.

Con el botón derecho del mouse, el jugador podrá apuntar con la jeringuilla.

Con el botón izquierdo, podrá disparar dosis de vacunas en ambos Modos e interactuar con la interfaz. Ambos pueden ser utilizados simultáneamente.



Figura 6.2. Control del *mouse* para seleccionar. (Fuente: <https://afnboysandgirlsclub.wordpress.com/mouse-skills/>)

6.2 DISEÑO DE LOS PERSONAJES

6.2.1 Creación del personaje principal con técnicas de scanner 3D.

Descripción: Es el estudiante protagonista de la historia. Este personaje tiene una característica única: presenta el aspecto facial del autor del proyecto.

Para su desarrollo, se han llevado a cabo los siguientes pasos:

- **Escaneado facial 3D del autor.** Inicialmente, para crear la malla personal, se hicieron pruebas de escaneado facial 3D en las instalaciones del MIC. Se consiguieron *nubes de puntos* [37] con resultados bastante buenos (los compañeros hicieron un gran trabajo escaneándose y uniendo varias versiones de escaneado en una versión más completa), pero la falta de luz nos seguía dejando una malla irregular. El tamaño de las mallas también fue un gran inconveniente, lo que me llevó a buscar otra solución.



Figura 6.3. Resultado de escanear a Iván (autor del proyecto) utilizando escáneres en el MIC.

Continuamos investigando y con el uso de la aplicación *Scandy Pro 3D Scanner* [38] logramos mayor cantidad de detalle. Estos fueron los resultados conseguidos:



Figura 6.4. Resultado de escanear a Iván utilizando *Scandy Pro 3D Scanner*.

- Corrección de errores en la malla con Blender. Una vez conseguida la malla, utilizando la herramienta Blender se hicieron algunas modificaciones para poder adaptar la cabeza a un personaje predeterminado:
- Corrección y cierre de huecos en la malla por falta de luz.

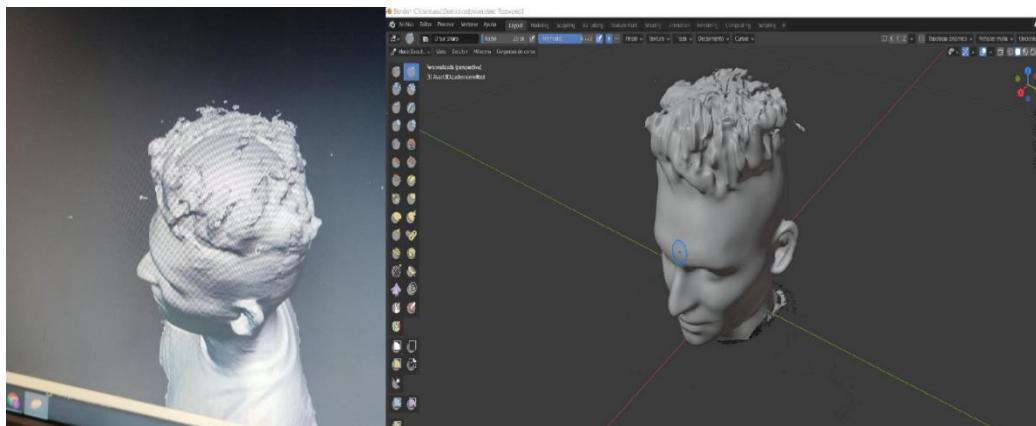


Figura 6.5. Antes y después de la malla de la cabeza en *Blender*.

- Reducir detalle para mejorar contraste con el personaje al que se unirá la cabeza.
- Corte cuidadoso de la malla por el cuello para quedarnos sólo con la cabeza.



Figura 6.6. Reducción de detalle de la malla de la cabeza en *Blender*.

- Corte cuidadoso de la malla del personaje seleccionado para quedarnos únicamente con el cuerpo.
- Unión de ambas mallas en una sola
- Modificaciones en la malla para adaptar mejor la unión.
- Unir el esqueleto de la cabeza y el del personaje.
- **Importación del modelo final en *Unity*.** Una vez tenemos el personaje modificado, lo importamos en *Unity*, lo animamos, lo metemos en los scripts con los demás personajes seleccionables y lo testeamos.



Figura 6.7. Resultado final del personaje principal del videojuego.

6.2.2 Personajes disponibles para jugar

Descripción: Son los personajes que podremos seleccionar para usar en el menú del juego. En un principio, en los personajes, se pensó incluir sólo alumnos y profesores (el personaje principal es un alumno). Sin embargo, para darle más variedad y diversión al juego se añadieron todos los personajes que aparecen en el juego (*mutantes*, *científicos*, incluso el *director*). Cabe destacar que, si no se selecciona ningún personaje, el personaje predeterminado es el personaje modificado.

Una vez en el juego, si la barra de contagio del personaje llega al máximo, se transforma en un *mutante* y vuelve a empezar desde su ubicación hace 10 segundos (*checkpoint*).

Velocidad: Tiene una velocidad de 5 puntos.

Puntos de vida: Tiene 100 puntos de vida.



Figura 6.8. Profesores y estudiantes seleccionables en el menú del juego.

6.2.3 Mutantes (profesores y estudiantes contagados)

Descripción: Son profesores y estudiantes mutados por el contagio del virus. Son los enemigos más comunes. Si te ven, te perseguirán hasta que salgas de su radio de visión. Si te atacan, te contagiarán. Si están cerca, te contagiarán como un virus (pero menor cantidad). Una vez curado aparece un estudiante o un profesor.

Velocidad: Tienen una velocidad de 3.5 puntos.

Puntos de vida: Tiene 100 puntos de vida. Requiere 3 dosis para curarlo.

Daño: Causa 10 puntos de mutación por zarpazo y 5 por cercanía al jugador.

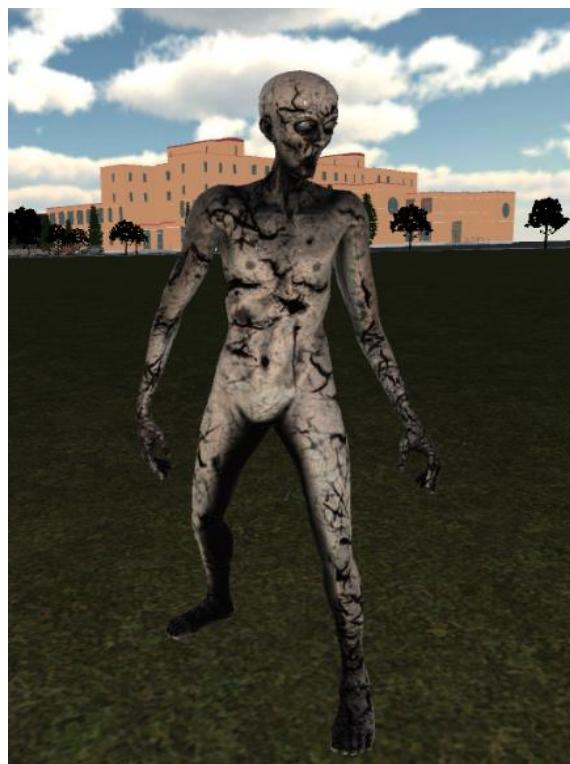


Figura 6.9. *Mutante*, enemigo del videojuego.

6.2.4 Quinto Paciente (enemigo principal del videojuego)

Descripción: Es el mayor enemigo del *Modo Historia*. Es más fuerte y resistente que los demás, ya que lleva más tiempo sometido a la mutación del virus. Se caracteriza por su color más rojizo y siempre se encuentra rodeado de otros *mutantes*. Es el enemigo final del *Modo Historia*.

Velocidad: Tiene una velocidad de 4 puntos.

Puntos de vida: Tiene 500 puntos de vida. Requiere 10 dosis para curarlo.

Daño: Causa 20 puntos de mutación por golpe y 5 por cercanía al jugador.



Figura 6.10. *Quinto Paciente*, enemigo principal del videojuego.

6.2.5 Científico de Guardia, Científico Principal y el Director

Descripción: Son los principales *NPC* (*non player character*) del *Modo Historia*.

- El *Científico Principal* está a cargo de la investigación Covid19 en el edificio MIC. Tiene 4 diálogos y dos notas que inician dos misiones.
- El *Científico de Guardia* vigila los pacientes por las noches. Tiene 34 diálogos e inicia dos misiones.
- El *Director* hace alusión a un alto cargo de la Universidad de León. Aparece en la etapa final del *Modo Historia*. Tiene 5 diálogos e inicia 1 misión.

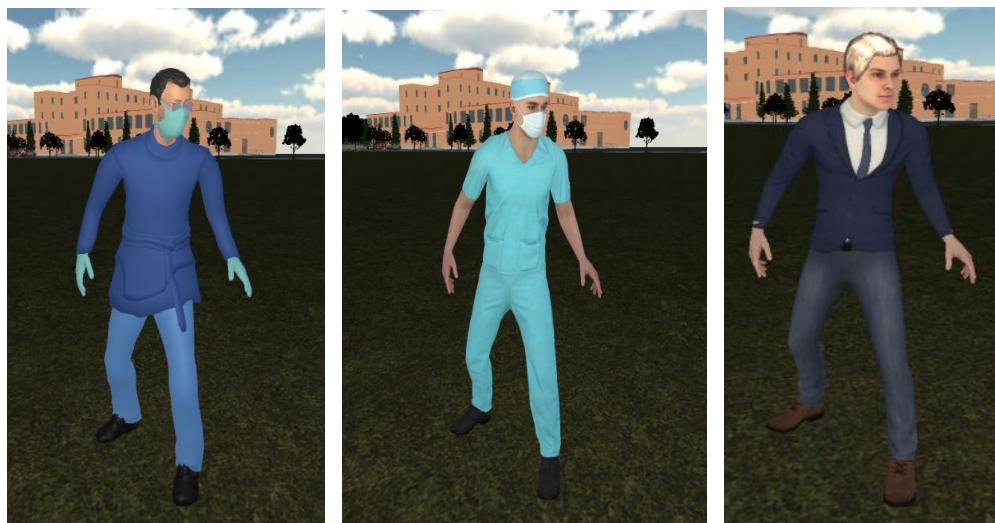


Figura 6.11. *Científico de Guardia, Científico Principal y Director* del videojuego.

6.2.6 Pacientes y Cuarto Paciente

Descripción: Son los pacientes sometidos a las 5 posibles vacunas del Covid19. Tres de ellos han muerto. El *Cuarto Paciente* ha obtenido la vacuna correcta, pero acaba muriendo también dado su grave estado de salud.



Figura 6.12. *Pacientes y Cuarto Paciente* del videojuego.

6.3 DISEÑO DEL ARMA Y FUNCIONALIDADES EXTRA

Probablemente, lo más interesante y novedoso de este juego radica en el arma y su forma de disparar: una jeringuilla de tamaño grande que por munición dispara *dosis de vacunas*. Estas dosis, tanto al disparar como al impactar en un *mutante* no estallan como lo haría una bala, se esparce una neblina color azul verdoso a modo de vacuna. La forma de disparo, rango de disparo y cadencia de disparo son estándar de un arma rápida, estilo *fusil de asalto*. Este arma la llevarán todos los personajes seleccionables en la mano derecha.

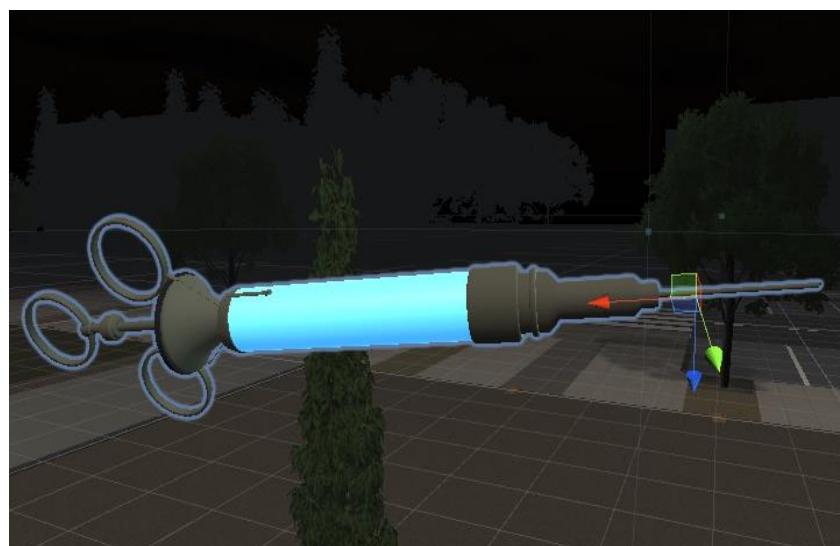


Figura 6.13. Imagen de la jeringuilla, arma principal del juego.

En cuanto a las funcionalidades se han creado objetos que nos otorgarán sanación y más daño de dosis (estas funcionalidades sólo estarán presentes en el *Modo Desafío*).

- **Vacunas:** Las vacunas son unos objetos con forma de cápsula que albergan un color verdoso en su interior. Sirven para que podamos curarnos una vez nos han contagiado. Se encuentran esparcidas por todo el mapa de manera aleatoria. Son escasas y limitadas.

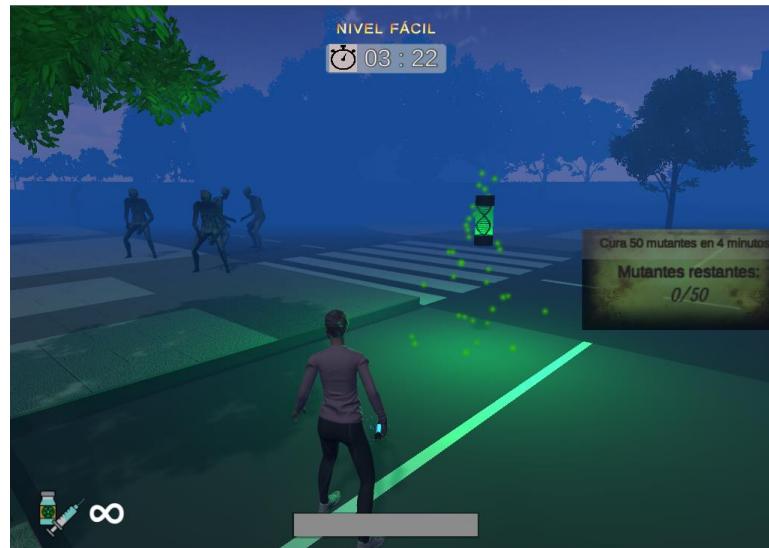


Figura 6.14. Imagen de una vacuna en el *Modo Desafío*.

- **Incrementos de dosis:** Con forma de dos vacunas en cruz de color azul claro, este objeto nos otorgará un bonus de daño durante 15 segundos. Mientras dure el efecto, podremos sanar a cualquier mutante con una sola dosis.

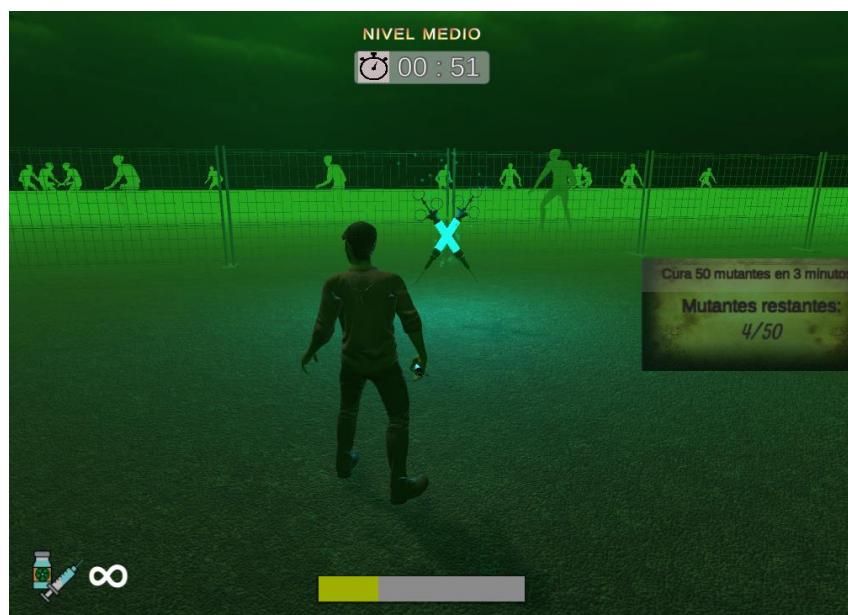


Figura 6.15. Imagen del bonus *incremento de dosis* en el *Modo Desafío*.

6.4 DISEÑO DEL MENÚ

6.4.1 Menú Principal

Descripción: Al iniciar el videojuego, podrás elegir entre *Modo Historia*, *Modo Desafío* o *Salir*. Si escoges un modo, se cargará ese modo.

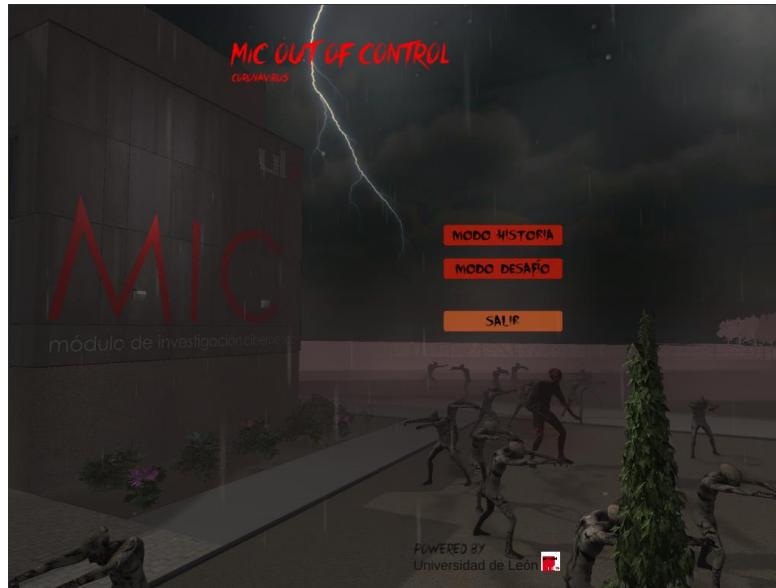


Figura 6.16. Menú inicial de *MIC out of control*.

Tanto si hemos seleccionado el *Modo Historia* o el *Modo Desafío*, debemos escoger si jugar en *Modo Un Jugador* (con lo cual jugaremos solos) o *Modo Multijugador* (podremos jugar con otros jugadores *online*).



Figura 6.17. Selección del *Modo Historia*.

Ahora bien:

- si hemos seleccionado Un Jugador en cualquiera de los modos, pasaremos directamente a la ventana de *Selección de personaje*.
- si hemos seleccionado Multijugador en cualquiera de los modos, primero pasaremos a la ventana de *búsqueda de partida* o *creación de partida*. Una vez hayamos creado partida o nos hayamos unido a una creada, podremos pasar a la ventana de *Selección de personaje*.

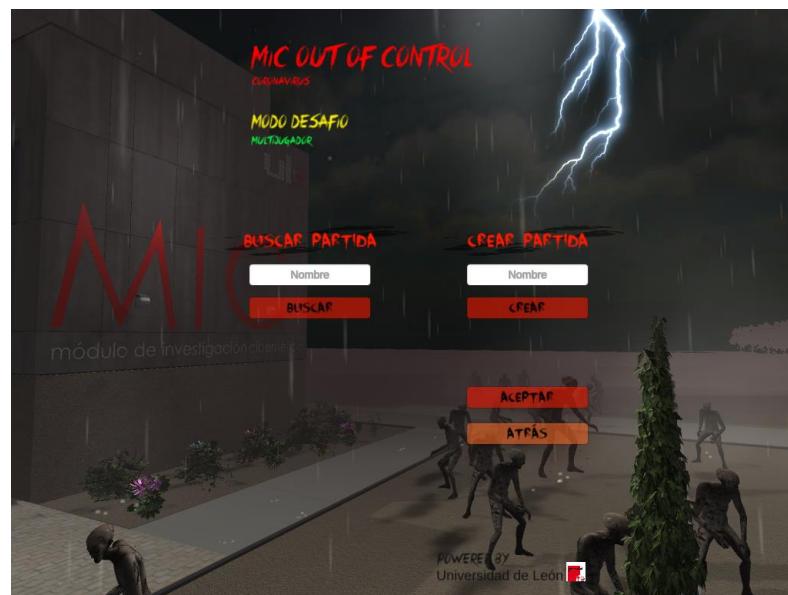


Figura 6.18. Interfaz *Multijugador* del *Modo Desafío*.

En la interfaz de *Selección de Personaje* podremos poner un nombre y elegir con qué personaje de los 22 que hay queremos jugar.



Figura 6.19. Interfaz *Selección de personaje* en *Multijugador* del *Modo Desafío*.

Una vez hayamos elegido el personaje que queremos usar, le daremos al botón aceptar y aparecerá la pantalla de carga. Una vez finalice ésta, se accederá al respectivo modo seleccionado.



Figura 6.20. Interfaz *pantalla de carga*.

6.4.2 Menús dentro del juego

A continuación, veremos los menús intermedios que podrán aparecer en el juego según nuestras acciones.

- *Menú de pausa*. Este menú pausará el juego en cualquier momento. Sólo debemos pulsar la tecla Esc. Este menú nos dará la opción de retomar el juego donde lo dejamos, ver los controles o salir del modo y volver al menú principal.

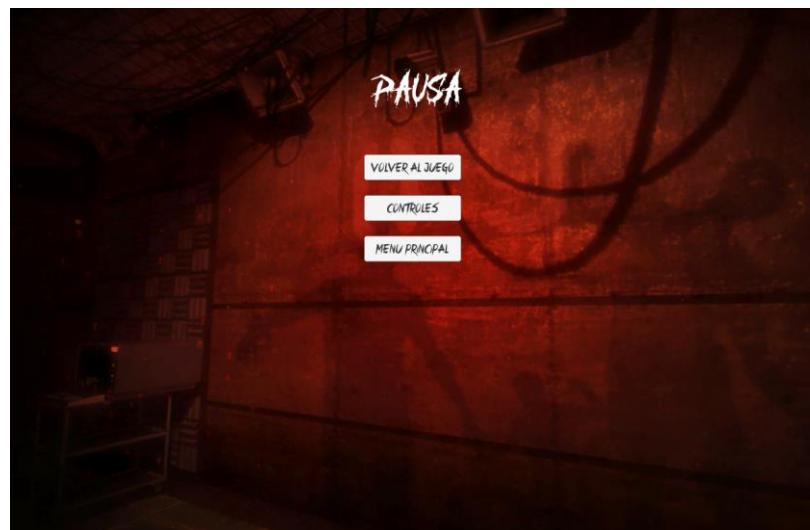


Figura 6.21. Interfaz *menú de pausa*.

- **Controles.** En esta ventana se nos informará de los controles del personaje en el juego. También nos informarán de acciones adicionales del *Modo Desafío* como poder coger vacunas o *incrementos de dosis*.

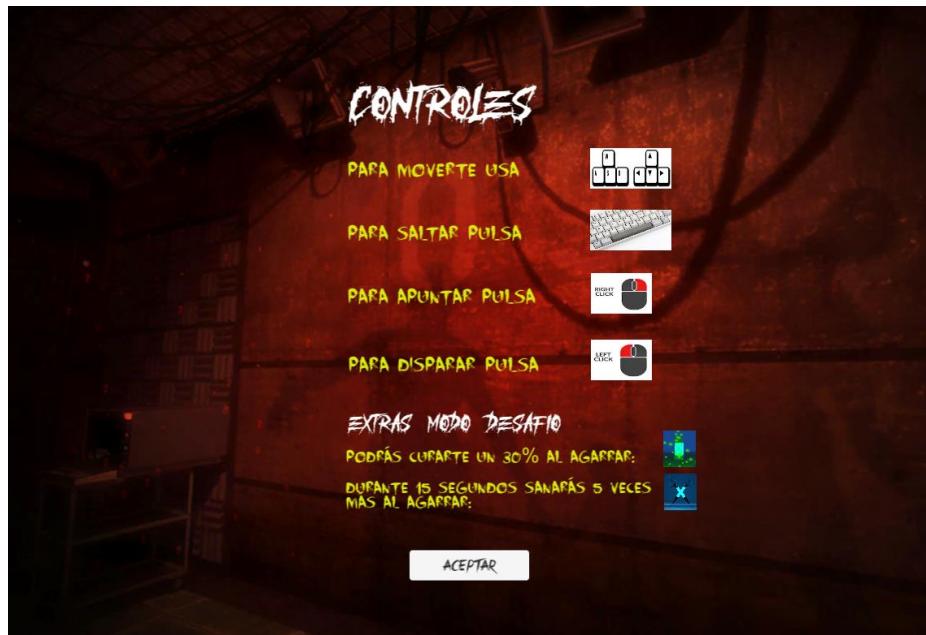


Figura 6.22. Interfaz *Controles*.

- **Objetivo conseguido.** Si somos capaces de sanar la cantidad de *mutantes* pedida, nos aparecerá esta ventana para informarnos de que hemos superado el nivel. Nos dará opción a *ir al siguiente nivel* o *salir*. Esta ventana sólo aparecerá en el *Modo Desafío*.



Figura 6.23. Interfaz *Objetivo conseguido*.

- *Reintentar.* Si los *mutantes* nos contagian hasta niveles críticos y la *barra de mutación* llega al máximo, nos convertiremos en *mutantes* y habremos perdido. En ese momento nos aparecerá la ventana *Reintentar*. Nos dará la opción de reintentar el nivel o salir al menú del juego.
 - Si le damos a *Reintentar* en el *Modo Historia*, nos pondrá la barra de mutación a cero y nos llevará a nuestra ubicación 10 segundos atrás (*checkpoint*).
 - Si damos a Reintentar en el *Modo Desafío*, se vuelve a cargar el nivel de cero.
- *Se acabó el tiempo.* Si la cuenta atrás llega a cero y aún no hemos alcanzado el objetivo pedido, también habremos perdido. Tendremos las mismas opciones que en la interfaz *Reintentar* y las mismas funcionalidades.



Figura 6.24. Interfaz *Reintentar*.



Figura 6.25. Interfaz *Se acabó el tiempo*.

6.4.3 Menús del Modo Historia y Modo Desafío

En la introducción de cada modo tendremos una interfaz introductoria que nos ponen en situación del modo al que vamos a jugar. Al dar al botón *Aceptar*, nos enseñará la interfaz *Controles* y seguidamente nos permitirá comenzar a jugar.



Figura 6.26. Interfaz *Modo Historia*.



Figura 6.27. Interfaz *Modo Desafío*.

6.5 DISEÑO DEL MODO DESAFÍO

Descripción: El *Modo Desafío* es un modo de juego en el que debemos sanar 50 *mutantes* antes de que el tiempo se acabe. Si lo conseguimos, pasaremos de nivel de dificultad. Hay tres niveles de dificultad: el *nivel fácil* (tiene una tonalidad azul), el *nivel medio* (tiene una tonalidad verde) y *nivel difícil* (tiene una tonalidad roja). Las mecánicas y diferencias de cada nivel han sido detalladas en el Anexo (sección 1.4.1 *Modo desafío*).



Figura 6.28. Nivel Fácil, Nivel Medio y Nivel Difícil del Modo Desafío.

6.6 DISEÑO DEL MODO HISTORIA

Descripción: El *Modo Historia* es un modo de juego en el que deberás ir avanzando en la historia para saber qué ha sucedido y cómo puedes solucionar lo ocurrido. Tú eres el protagonista en todo momento. Este modo de juego, su cadena de misiones y diálogos correspondientes han sido mucho más detallados en el Anexo (1.4.2 *Modo Historia*).

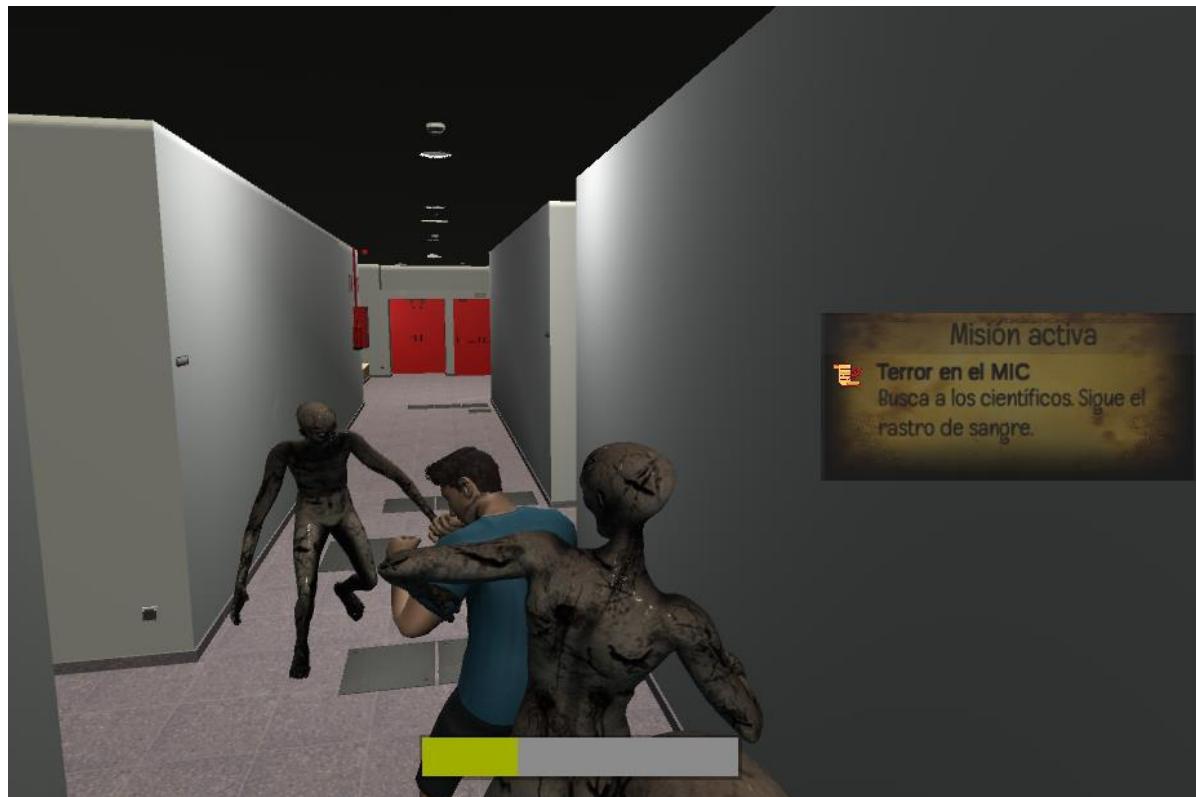


Figura 6.29. Fotograma del *Modo Historia*.

6.6.1 Diseño de misiones del modo historia

- Misión 1. *Día 1 de la Investigación.*

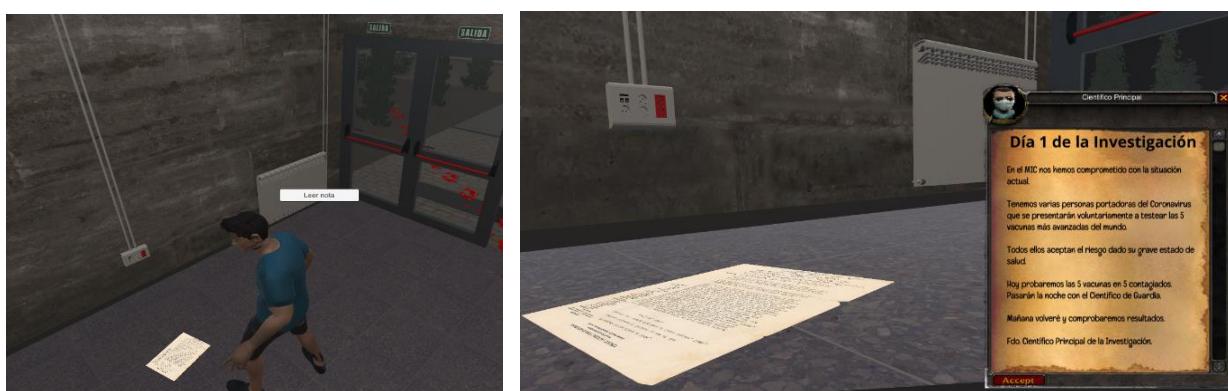


Figura 6.30. Primera misión del *Modo Historia*.

- Misión 2. Día 2 de la Investigación.

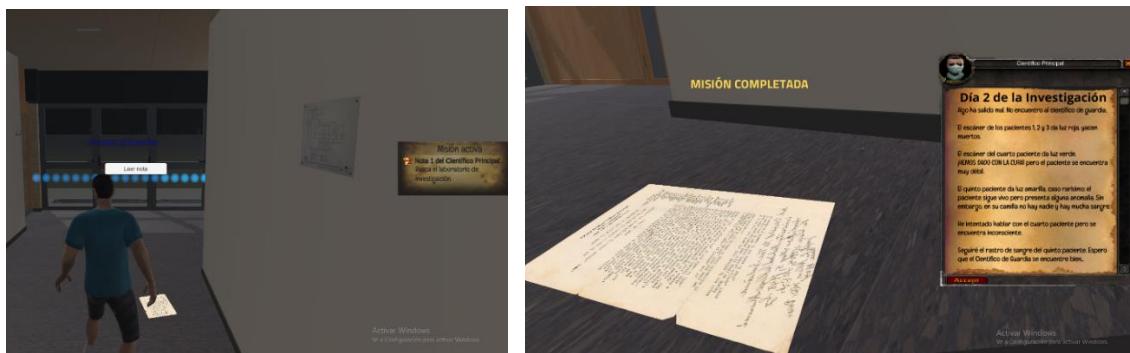


Figura 6.31. Segunda misión del *Modo Historia*.

- Misión 3. Busca a los científicos. Sigue el rastro de sangre.



Figura 6.32. Tercera misión del *Modo Historia* – Vista General.

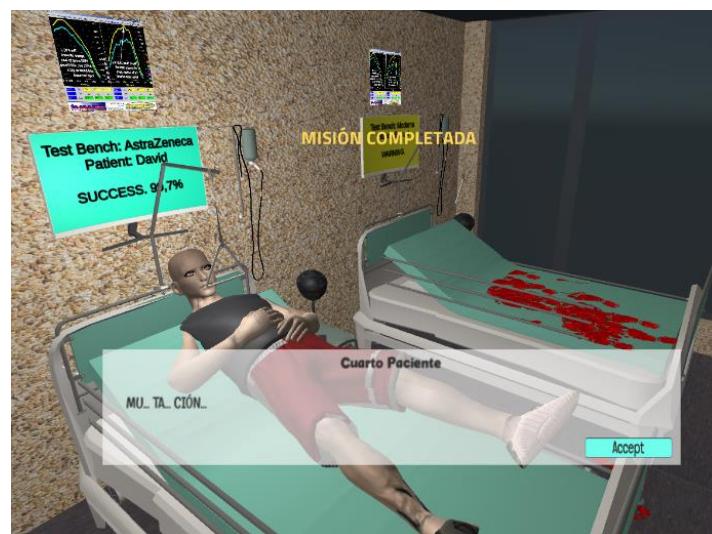


Figura 6.33. Tercera misión del *Modo Historia* – Primer Plano



Figura 6.34. Tercera misión del *Modo Historia* – Activación de mutantes

- Misión 4. Activa la corriente del MIC, consigue la vacuna y tráela de vuelta.



Figura 6.35. Cuarta misión del *Modo Historia* – Activación de corriente



Figura 6.36. Cuarta misión del *Modo Historia* – Obtención de vacuna

- Misión 5. *Cura con dosis a los mutados para que vuelvan a la normalidad.*



Figura 6.37. Quinta misión del *Modo Historia*.

6.7 DISEÑO DE LA INTERFAZ

Una vez dentro del nivel se mostrarán diferentes elementos que estarán presentes de manera continua o en distintas partes del *Modo Historia* (en el *Modo Desafío*, la barra de mutación y la munición siempre estarán presentes).

Empezando por la parte inferior izquierda tendremos la cantidad de munición de dosis restante. En el caso del *Modo Historia* será X cantidad y en el caso del *Modo Desafío* será infinita. Abajo en el centro de la pantalla encontraremos la barra de mutación.



Figura 6.38. Barra de mutación y munición de dosis.

Veamos cómo y cuándo se activan en el *Modo Historia*:

- Una vez activamos la Misión 3: *Busca a los científicos. Sigue el rastro de sangre* activamos la barra de mutación del personaje. Si nos atacan, la barra de mutación aumentará, reflejándose con un tono amarillento. Además, se activarán los mutantes y serán visibles para el jugador.
- Una vez activamos la Misión 4: *Activa la corriente del MIC, consigue la vacuna y tráela de vuelta* se desactivarán todas las luces del edificio. Una vez el jugador active la corriente, volverán a activarse.
- Una vez activamos la Misión 5: *Cura con dosis a los mutados para que vuelvan a la normalidad* se activará la jeringuilla y la munición de dosis.

7. Conclusiones

Para finalizar esta memoria, se exponen las conclusiones extraídas de la realización de este proyecto.

En el apartado técnico, este proyecto me ha servido para mejorar mis conocimientos casi nulos en este campo y aprender muchas características sobre el motor *Unity*. Desarrollar este proyecto me ha ayudado a entender mejor la programación en todos sus aspectos (programación orientada a objetos, recursividad, delegación y herramientas vistas a lo largo de la carrera como la utilización de patrones de diseño *Singleton*). Nunca había programado un videojuego y me llevo una gran experiencia conectando programación y motor gráfico.

Por otro lado, me llevo gran aprendizaje del campo del modelado, animación y texturas. También aportó notablemente la experiencia en investigación en escaneado 3D llevada a cabo en el MIC.

En el aspecto personal, he disfrutado muchísimo haciendo este proyecto. Jamás me habría imaginado ser capaz de plasmar una historia con tantos detalles, que a priori sólo se encontraba en mi cabeza (y mucho menos ser capaz de programarlo).

Este proyecto me ha enseñado a afrontar las dificultades que tiene diseñar un videojuego desde cero. Ha sido una excelente experiencia, ahora tengo clarísimo que decantaré mi futuro profesional hacia el desarrollo de videojuegos.

Como dato, el proceso de redacción de memoria ha sido algo tedioso a la hora de redactar la parte teórica, pero ameno explicando la implementación del proyecto.

Con respecto a la planificación, se estimó una duración de 4 meses, finalmente han sido unos 8 meses. Ese tiempo adicional ha surgido de dos principales motivos:

Una curva de aprendizaje inicial casi nula, debido a mi falta de conocimientos en desarrollo de videojuegos (tanto en el motor gráfico, como en la programación en C# ligada a *Unity*), el cual, a base de la superación de numerosos obstáculos, se ha incrementado notablemente. También contar que no fui consciente del tamaño del proyecto ni de la duración que algunas fases podrían llegar a tener.

Aprovechar el TFG para continuar aprendiendo. Una vez que he ido comprendiendo mejor el funcionamiento de todo, me han surgido muchas ganas de aprender a hacer nuevas funcionalidades para el videojuego (un *Modo Desafío* con distintos niveles de dificultad, el incremento de daño en el tiempo en el modo desafío, la utilización de un cronómetro, la

creación de un menú con varios personajes seleccionables, etc) y vi este un buen momento para hacerlo.

Los mayores problemas surgieron en:

El salvaguardar variables entre escenas. Sin duda, este fue el mayor problema. Desde un inicio el proyecto se comenzó en una escena provisional, por lo cual, una vez pasado al modelado del MIC, aparecieron miles de errores. Casi todos, debidos a la pérdida de variables entre escenas. Gracias a la clase *GameManager.cs* (donde centralicé todas las variables que no podían perderse) con tiempo y testeo se acabaron los errores.

El diseño de la cámara principal del personaje principal (al inicio giraba muy rápido y la cámara golpeaba con los demás objetos), hice varios scripts hasta dar con la cámara buena.

El movimiento del personaje principal. Hubo dos scripts iniciales fallidos dada la torpeza en el movimiento. Se acabó consiguiendo la fluidez deseada.

La implementación del sistema de misiones. Esta fase fue amena pero cargada de aprendizaje. El problema aquí fue no medir bien el tiempo que llevaba esta tarea.

Por último, los resultados de este proyecto han sido muy productivos como formación personal y, académicamente hablando, satisfactorios al cumplirse la mayoría de los objetivos propuestos.

8. Trabajos futuros

El juego se diseñó como una demo, que presenta mecánicas de un videojuego estilo *Survival Horror* mezcladas con uno tipo *MMORPG* (*massively multiplayer online role-playing game*) por el sistema de misiones implementado. El fin era conocer cómo funcionan ambos estilos tocando varios puntos en común.

En un futuro cercano, se espera desarrollar un sistema de puntos. Se iniciaría únicamente con el personaje principal, y los personajes secundarios tendrán que ser desbloqueados a través del sistema de puntos). Para conseguir dichos puntos tendríamos misiones (ej: pasarnos el nivel medio del *Modo Desafío* en menos de 2 minutos).

Con esta base de conocimiento, tengo en mente desarrollar varios proyectos:

Un MMORPG 3D entero de ambientación moderna, Contará con sistema de niveles, sistema de misiones, sistema de comercio, clases de personajes, enemigos, profesiones, *gathering*, etc

Un juego de móvil 2D estilo “Brawl Stars” (juego shooter 2D de tres contra tres jugadores con varios personajes a elegir, cada uno con diferentes estadísticas, en el que debes vencer en diversos mapas a tus contrincantes). Pero me gustaría darle un enfoque con temática de “Los Juegos del Hambre”, 12 jugadores, 12 armas en el centro, el último en pie gana la partida.

Un videojuego en realidad virtual estilo “Sword Art Online”. Desafortunadamente tardaré en comenzar este proyecto, pues aún debo formarme mucho en el campo de la realidad aumentada y virtual.

9. Agradecimientos

A mi tutor *Fernando Jorge Fraile Fernández* por darme la oportunidad de desarrollar este videojuego, creer en mí y estar disponible siempre que lo he necesitado. Gracias por compartirme el increíble modelado del edificio MIC y sugerirme la temática del Covid19 para el *Modo Historia* del videojuego.

También quiero agradecer a la profesora *Susana Martínez Pellitero* por darme la oportunidad de probar el escaneado 3D con los equipos que tienen en el edificio MIC, y a la profesora *Rebeca Martínez García* y al personal del laboratorio por su paciencia y dedicación a la hora de escanearme.

Quisiera agradecer a mi buen compañero *Enrique Gonjar Verdejo* por haber estado ahí de manera totalmente voluntaria siempre que me he atascado programando. Me ha compartido sus conocimientos y sin duda alguna haremos grandes videojuegos juntos.

Por último me gustaría dar las gracias a mi hermana *Brisa Castro Martínez* por ayudarme a imaginar cómo debía ser el *Modo Historia*. Además, me ayudó en la creación de mi personaje jugable con *Blender*.

10. Referencias

- [1] N. A. Borromeo, *Hands-On Unity 2020 Game Development*, Birmingham: Packt publishing, 2020.
- [2] A. E. d. Videojuegos, «El videojuego en el mundo,» [En línea]. Available: <http://www.aevi.org.es/la-industria-del-videojuego/en-el-mundo/>. [Último acceso: 16 09 2020].
- [3] K. Taite, «The history of Survival Games,» [En línea]. Available: <https://survivethis.news/en/history-of-survival-games/>. [Último acceso: 18 09 2020].
- [4] S. Reid y S. Downing , «Survival Themed Video Games and Cultural Constructs of Power,» *The Journal of the Canadian Game Studies Association*, vol. 11, nº 18, pp. 41-57, 2018.
- [5] K. MacDonald, «6 of the best survival games,» 16 09 2020. [En línea]. Available: <https://www.ign.com/articles/6-of-the-best-survival-games>.
- [6] Capcom, «Resident Evil Portal,» 17 09 2020. [En línea]. Available: <https://game.capcom.com/residentevil/en/>.
- [7] Mojang Synergies A, «Sitio oficial | Minecraft,» 18 09 2020. [En línea]. Available: <https://www.minecraft.net/es-es/>.
- [8] Epic Games, «Fortnite | Juego Multiplataforma Gratuito,» 18 09 2020. [En línea]. Available: <https://www.epicgames.com/fortnite/es-MX/home?lang=es-MX>.
- [9] Red Barrels, «Home - Red Barrels,» 18 09 2020. [En línea]. Available: <https://redbarrelsgames.com/>.
- [10] Red Barrels, «Outlast | Red Barrels,» 18 09 2020. [En línea]. Available: <https://redbarrelsgames.com/games/outlast/>.
- [11] CAPCOM, «CAPCOM : RESIDENT EVIL,» [En línea]. Available: <https://www.residentevil7.com/>. [Último acceso: 19 09 2020].
- [12] CAPCOM, «CAPCOM : RESIDENT EVIL,» [En línea]. Available: https://www.residentevil7.com/es/#_about. [Último acceso: 18 09 2020].
- [13] Konami Digital Entertainment Co., «Silent Hill Memories | news, information, media and donwloads,» [En línea]. Available: https://www.silenthillmemories.net/main/main_en.html. [Último acceso: 19 09 2020].
- [14] Silent Hill Wiki, «Silent Hill 4: The Room | Silent Hill Wiki en Español | Fandom,» [En línea]. Available: https://silenthill.fandom.com/es/wiki/Silent_Hill_4:_The_Room. [Último acceso: 20 09 2020].
- [15] Epic Games, «Features - Unreal Engine,» [En línea]. Available: <https://www.unrealengine.com/en-US/features>. [Último acceso: 12 10 2020].
- [16] Unity Technologies, «Plataforma en tiempo real de Unity,» [En línea]. Available: <https://unity.com/es>. [Último acceso: 03 10 2020].

- [17] Atlassian Confluence, «CRYENGINE V MANUAL,» [En línea]. Available: <https://docs.cryengine.com/>. [Último acceso: 04 10 2020].
- [18] H. Ferrone, Learning C# by Developing Games with Unity 2020, Birmingham: Packt publishing, 2020.
- [19] Unity Technologies, «Unity User Manual (2019.4 LTS),» [En línea]. Available: <https://docs.unity3d.com/es/current/Manual/UnityManual.html>. [Último acceso: 05 11 2020].
- [20] Unity Technologies, «Unity Scripting API,» [En línea]. Available: <https://docs.unity3d.com/es/current/ScriptReference/index.html>. [Último acceso: 05 10 2020].
- [21] Unity Technologies, «Unity Collaborate - Unity Manual,» [En línea]. Available: <https://docs.unity3d.com/es/2017.4/Manual/UnityCollaborate.html>. [Último acceso: 04 11 2020].
- [22] Unity Technologies, «Git,» [En línea]. Available: <https://git-scm.com/>. [Último acceso: 05 11 2020].
- [23] GitHub. Inc, «GitHub: Where the world builds software,» [En línea]. Available: <https://github.com/>. [Último acceso: 04 11 2020].
- [24] Kinsta Inc, «¿Qué es GitHub? Una Guía para Principiantes sobre GitHub,» [En línea]. Available: <https://kinsta.com/es/base-de-conocimiento/que-es-github/>. [Último acceso: 05 11 2020].
- [25] Perforce Software, «Perforce Software | Development Tools For Innovation at Scale,» [En línea]. Available: <https://www.perforce.com/>. [Último acceso: 06 11 2020].
- [26] Perforce Software, «Perforce Helix Core Version Control | Perforce,» [En línea]. Available: <https://www.perforce.com/products/helix-core>. [Último acceso: 05 11 2020].
- [27] The Apache Software Foundation, «About the Apache HTTP Server Project,» [En línea]. Available: https://httpd.apache.org/ABOUT_APACHE.html. [Último acceso: 09 11 2020].
- [28] Unity Technologies, «Log Viewer | Integration | Unity Asset Store,» [En línea]. Available: <https://assetstore.unity.com/packages/tools/integration/log-viewer-12047> . [Último acceso: 09 11 2020].
- [29] IAT, «Inteligencia artificial: Qué es, tipos, ventajas - IAT,» [En línea]. Available: <https://iat.es/tecnologias/inteligencia-artificial/>. [Último acceso: 02 12 2020].
- [30] Adobe Systems Incorporated, «Mixamo,» [En línea]. Available: <https://www.mixamo.com/#/?page=1&type=Character>. [Último acceso: 10 12 2020].
- [31] Perforce Software, «Installing Perforce Server 15.1 on Linux,» [En línea]. Available: https://www.perforce.com/manuals/v15.1/dvcs/_installing_perforce_server_15_1_on_linux.html. [Último acceso: 11 09 2020].

Anexo 1: Documento de diseño

MIC out of control. Desarrollo de un videojuego en *Unity3D*
utilizando el edificio MIC como entorno jugable

por Iván Castro Martínez

A1.1 Información general

A1.1.1 CONCEPTO GENERAL

El juego tiene lugar en unas instalaciones universitarias donde se llevan a cabo pruebas para encontrar la vacuna contra el Coronavirus. Algo sale mal. El virus muta con una de las posibles vacunas, volviéndose letal. El jugador deberá seguir unas pistas e ir completando las diferentes misiones para finalmente descubrir lo ocurrido, y así salvar la humanidad.

A1.1.2 OBJETIVO

El jugador tendrá como objetivo evitar contagiarse por los infectados, y posteriormente curarlos. Todo esto evitando que la nueva anomalía salga al exterior y se propague a todo el mundo.

A1.1.3 GÉNERO

Es un videojuego *Survival* que pertenece al subgénero de videojuegos *Survival Horror*.

A1.1.4 HISTORIA O SINOPSIS

Al abrir el juego aparecerá el menú inicial que preguntará en qué modo deseas jugar: *Modo Historia* o *Modo Desafío*. Una vez elijas uno, te preguntará si quieres jugar modo *Un jugador* o *Multijugador*. Realizada la elección, se pedirá escribir un nombre y permitirá elegir un personaje de entre todos los disponibles. Una vez seleccionado, aceptaremos y se mostrará la pantalla de carga del modo elegido.

A1.1.4.1 Modo Desafío

En este modo de juego aparecerás en el exterior del edificio MIC con una extraña y densa niebla. Dispondrás de jeringuilla como arma y munición infinita de dosis de vacunas.

Cuando inicies, aparecerá un cronómetro, y deberás curar a la cantidad de *mutantes* pedida, antes de que la cuenta atrás llegue a su fin.

Funciones adicionales:

- **Vacunas del personaje:** Al coger una vacuna, sana al jugador restándole un 30% de la cantidad de mutación. Las vacunas son limitadas y aparecen aleatoriamente por todo el mapa.
- **Incremento de daño de dosis:** Al coger un incremento de daño, durante 15 segundos tendremos un *bonus de daño* de dosis. Una vez pasado el tiempo, volveremos al daño de dosis normal. Esto nos permitirá sanar mutantes con un solo disparo (normalmente son necesarios 5). Los incrementos de dosis son limitados y aparecen aleatoriamente por todo el mapa.

Hay tres tipos de niveles: *nivel fácil*, *nivel medio* y *nivel difícil*. Empezaremos en el nivel fácil. Si lo completamos pasaremos al nivel medio, y si completamos este, pasaremos al nivel difícil. Este último, será el último nivel de este modo de juego.

A continuación, se enseñará una tabla a modo de referencia de cada nivel. Importante destacar que por “*Mutantes normales*” entendemos los “*Mutantes*”. Y por “*Mutantes Reforzados*” entendemos réplicas del *boss* del Modo Historia, “*Quinto paciente*” (el cual tiene el doble de vida y hace el doble de daño que los mutantes normales). Todos los niveles han sido testeados y son posibles de realizar (el modo difícil es muy difícil, pero siempre pueden reintentarse todos los niveles).

Estas son sus diferencias:

Nivel	Fácil	Medio	Difícil
Mutantes solicitados	50	50	50
Mutantes normales	100%	50%	0%
Mutantes reforzados	0%	50%	100%
Tiempo máximo	4 minutos	3 minutos	2 minutos
Vacunas	30 vacunas	20 vacunas	10 vacunas
Incrementos de dosis	20 incrementos	15 incrementos	10 incrementos

Tabla A1.1. Tabla que explica las diferencias entre los 3 niveles del *Modo Desafío*.

A1.1.4.2. Modo Historia

Este es el modo principal del juego, el jugador aparece en la entrada del MIC dispuesto a ir a su clase de prácticas de Robótica. Es última hora, ya es de noche y hay mucha niebla. Al entrar, se encuentra el edificio en estado de alarma y no hay nadie por ninguna parte.

Mira hacia el suelo y encuentra una nota de uno de los científicos que trabajaban allí en colaboración con el Ministerio de Sanidad para encontrar una vacuna para el Covid19. La **nota 1** pone:

Nota 1 del Científico Principal.

Día 1 de la Investigación Covid19.

En el MIC nos hemos comprometido con la situación actual.

Tenemos varias personas portadoras del Coronavirus que se presentarán voluntariamente a testear las 5 vacunas más avanzadas del mundo.

Todos ellos aceptan el riesgo dado su grave estado de salud.

Hoy probaremos las 5 vacunas en 5 contagiados.

Pasarán la noche con el Científico de Guardia.

Mañana volveré y comprobaremos resultados.

Fdo. Científico Principal de la Investigación”.

Y entonces se desbloquea una misión “**Busca el laboratorio de investigación**”. El jugador se pone en búsqueda del laboratorio, tal vez alguien pueda explicar qué está sucediendo. Una vez lo encuentra, en la puerta se encuentra la **nota 2** y se completa la misión 1. La nota dice:

Nota 2 del Científico Principal.

Día 2 de la Investigación Covid19.

Algo ha salido mal. No encuentro al científico de guardia.

El escáner de los pacientes 1, 2 y 3 da luz roja, yacen muertos.

El escáner del cuarto paciente da luz verde.

¡HEMOS DADO CON LA CURA! pero el paciente se encuentra muy débil.

El quinto paciente da luz amarilla, caso rarísimo: el paciente sigue vivo, pero presenta alguna anomalía. Sin embargo, en su camilla no hay nadie y hay mucha sangre.

He intentado hablar con el cuarto paciente, pero se encuentra inconsciente.

Seguiré el rastro de sangre del quinto paciente. Espero que el Científico de Guardia se encuentre bien..."

Y se desbloquea otra misión “**Habla con el cuarto paciente**”.

Una vez entra al laboratorio y comprueba todo lo que pone en la nota, se acerca al cuarto paciente. Se encuentra muy debilitado pero curado. Aparece un diálogo con el:

Diálogo con el Cuarto Paciente

Paciente nº4: “Mu...ta...ción” y fallece el paciente

Finaliza el diálogo y se desbloquea otra misión: “**Terror en el MIC**”.

Debe buscar a los científicos y seguir el rastro de sangre.

Sale del laboratorio y sigue unas huellas rojas que salen desde la camilla del *cuarto paciente* hacia fuera del laboratorio.

Por el camino se encuentra *mutantes* (estudiantes y profesores mutados por el contagioso virus, prácticamente sin ropa, la piel quemada y los ojos blancos). Si le ven, corren hacia él y lo contagian o atacan.

Corre mientras sigue las huellas hasta la segunda planta y llega a una puerta de cristal cerrada donde se encuentran refugiados los dos científicos. Rápidamente te abren y cierran. Los mutantes se quedan intentando entrar. Aparece un diálogo con el *científico de guardia*:

Diálogo con el Científico de Guardia

- ¿Quién eres? ¿Cómo has conseguido llegar hasta aquí? ¿Estás contagiado?

- Bueno, es igual necesitamos tu ayuda.

Soy Marcos, Científico de Guardia de la Investigación del MIC para el Covid19.

- *Te haré un breve resumen de la situación: Estaba haciendo guardia a los cinco pacientes y de repente algo salió mal.*
- *El quinto paciente empezó a tener convulsiones, igual que las que tiene el Científico Principal ahora mismo. Su piel empezó a cambiar mientras gritaba de sufrimiento.*
- *La situación se descontroló, los escáneres daban errores rarísimos y salí corriendo. Me siguió hasta aquí corriendo y me refugié aquí.*
- *Hace poco apareció corriendo el Científico Principal. Supongo que seguía el rastro del que me perseguía, pero le han mordido. Lo conseguí meter aquí a rastras.*
- *El quinto paciente ha ido contagiando a todo el edificio. A estas alturas todos deben haber mutado. Y queda poco tiempo para que él también se transforme.*
- *La posible vacuna se alió con el Covid19 y los resultados han sido: una enfermedad casi tan letal como el Ébola sumado al nivel de contagio del Covid19.*
- *Tengo una idea, pero es muy descabellada. Tampoco tenemos alternativa. Necesito que vayas y me traigas... ¿¿QUE HA PASADO??*
- *¡Han cortado la corriente de todo el edificio! Si consiguen salir del edificio, ¡morirá toda la humanidad! Cambio de planes, ahora debes hacer dos cosas.*
- *Primero debes ir al fondo de la Planta Baja 1, buscar el panel de suministro eléctrico y reactivarlo para bloquear las puertas.*
- *Luego debes ir al Laboratorio donde están los pacientes y traerme la vacuna del Covid19 del cuarto paciente. Está en la estantería y es de color verde.*

- Todo esto claro está, sin llegar a niveles de contagio críticos. Si no, te transformarás en uno más y estaremos perdidos.
- Recuerda que sólo el hecho de estar cerca aumenta el nivel de contagio. ¡Date prisa y buena suerte! La humanidad está en tus manos.

Se acaba el diálogo y aparece nueva misión, “**Sin luz y sin defensas**”.

Misión activa

- 0/1 Activa la corriente del MIC.
- 0/1 Coge la vacuna del laboratorio.

Aparece un cronómetro en cuenta atrás de 3 minutos (es el tiempo máximo que queda para que los mutantes escapen del edificio y contagien a todo el mundo).

El jugador baja las escaleras hacia la primera planta, y busca el panel de la corriente esquivando mutantes. Activa la corriente pulsando un botón. La misión activa marca “1/1 *Activa la corriente del MIC*”.

Luego se dirige hacia el laboratorio, recoge la vacuna de la estantería pulsando un botón. La misión marca “1/1 *Coge la vacuna del laboratorio*” y vuelve a la segunda planta donde se encontraban los científicos.

Aparece otro diálogo con el *científico de guardia*. El *científico principal* sigue en el suelo en estado crítico.

Diálogo con el Científico de Guardia

- OH DIOS, ¡SIGUES VIVO! ¿Tienes la vacuna? Excelente, dámela. ¡El Científico Principal está a punto de mutar!
 - *Le das la vacuna*
- + Científico Principal: ¿Qué ha pasado? ¿Dónde estoy?

- *Me alegra de que se encuentre bien. Es una larga historia.*
- *Ahora que hemos visto que funciona, tenemos una oportunidad.*
- *Con esta vacuna me dará para cargar tres jeringuillas: una para el científico principal, otra para ti y otra para mí.*
- *No tenemos más. No las malgastéis o estaremos perdidos.*
- *Nos dividiremos para sanarlos. Tú debes sanar la mayor amenaza que es el Quinto Paciente y los que te encuentres por el camino. Nosotros nos encargaremos del resto.*
- *Recuerda que lleva más tiempo mutado y tiene el doble de resistencia a la dosis.*
- *La última vez que lo vi se encontraba en la Planta Baja 2.*
- *Tener mucho cuidado porque los mutados lo tienen de jefe y siempre está acompañado.*
- *Espero volvamos a vernos todos. ¡Suerte!*

Aparece una nueva misión “**Acaba con la mutación**”.

Aparece la jeringuilla en la mano, una cantidad limitada de dosis y unos requisitos para completar la misión y se actualiza la cifra por cada uno salvado:

Misión activa

- 0/10 Mutantes sanados.
- 0/1 Quinto paciente sanado.

El jugador cura, uno a uno, a los *mutantes* con los que se va encontrando procurando que su barra de contagio no llegue al máximo. Los estudiantes y profesores vuelven a la normalidad reaccionando desconcertados, al igual que el *Científico Principal*.

Una vez la misión marca “10/10 Mutantes sanados”, sólo queda el *Quinto paciente*.

Se encuentra rodeado de otros muchos mutantes y le quedan pocas dosis. Finalmente, el Quinto paciente es sanado “1/1 Quinto paciente sanado”. Vuelves con los científicos y se completa la misión.

Aparece un diálogo con el *científico principal*:

Diálogo con el Cuarto Paciente

- ¡Lo conseguimos! ¡Hemos salvado a muchísimas personas y tenemos la vacuna!
- Vayamos a informar de todo esto. Habrá que despertar a todo el mundo.
- Dios, ha sido como vivir una película de terror...

Se carga una nueva escena. En las puertas del MIC hay una ceremonia. Aparecen todas las personas sanadas, los científicos y un directivo de la Universidad de León.

Aparece un diálogo del directivo dando una charla:

Diálogo con el Directivo

- Hoy no es un día normal, ni mucho menos.
- Hoy han muerto 3 pacientes experimentando posibles vacunas, descansen en paz.
- Hoy, podríamos haber amanecido siendo víctimas de la mayor pandemia de todos los tiempos.
- Sin embargo, gracias al fabuloso trabajo de este estudiante y nuestros científicos, no sólo hemos eliminado la peligrosa mutación...
- Hemos dado con la vacuna del Covid19 y ya está siendo producida en masa en laboratorios de todo el país.
- En nombre de la Universidad de León os agradecemos enormemente vuestro trabajo. Sin vosotros no habría sido posible.

- *Como agradecimiento, nos gustaría entregarte una Beca.*
- *Gracias por todo.*

Horas después aparece en los medios la vacuna y los agradecimientos a los científicos y el jugador. Fin del Juego.

A1.1.5 ESTILO VISUAL

El juego tendrá un estilo 3D realista con diseños proporcionales y gráficos medios.

A1.1.6 MOTOR Y EDITOR

Se usará *Unity* y *Visual Studio* como IDE. Se hará uso de impresoras 3D y *Adobe Fuse CC* para los personajes. Se utilizará Ubuntu para el servidor y el control de versiones *Perforce*.

A1.1.7 NÚCLEO DEL GAMEPLAY

El jugador podrá realizar las siguientes acciones:

- Podrá interactuar con el menú principal (escoger personaje y modo de juego).
- Podrá caminar, correr, saltar, coger o activar objetos y completar misiones.
- Mover la cámara de visión en todas direcciones.
- Activar y completar misiones.
- Disparar.

A1.1.8 PÚBLICO OBJETIVO

Dirigido a cualquier usuario, pero sobre todo a jugadores que tengan experiencia en juegos *Survival* o jugadores que les guste el suspense y los retos.

A1.1.9 CARACTERÍSTICAS DEL JUEGO

A1.1.9.1 Ambientación

La historia se desarrolla en el mundo real, concretamente en el edificio MIC de la Universidad de León. Aquí transcurre toda la historia. La época es actual, Febrero de 2021, en tiempos de pandemia global.

A1.1.10 ALCANCE DEL PROYECTO

A1.1.10.1 Ubicaciones del juego

La acción se desarrolla en clases, laboratorios y pasillos del edificio MIC.

A1.1.10.2 Descripción de los enemigos

Encontraremos dos tipos de enemigos infectados: los profesores/científicos y los alumnos. Tienen como objetivo contagiarte y finalizar la partida.

A1.1.10.3 Descripción de las armas o defensas

El arma para vacunar a los infectados son jeringuillas médicas y la munición dosis de vacunas.

A1.1.10.4 Descripción del nivel

El nivel cuenta con pasillos, escaleras, clases y laboratorios. Una vez desbloqueada determinada fase del modo historia, los enemigos podrán estar por todo el edificio.

Anexo 2: Recursos Utilizados

Se incluyen, en este apartado, diferentes assets de Dominio Público, disponibles en Internet, que han sido modelados por terceras personas y que el autor de este TFG ha modificado a conveniencia del desarrollo del juego. La utilización de estos recursos se hace con fines académicos sin intención de comercialización.

- Desfibrilador del laboratorio:

https://s3.amazonaws.com/tsinternal_standard/Internal/2016/06/30_13_50_51/glass_door_free_model_gray.obj?AWSAccessKeyId=AKIAIJ6EELKKOTA5UAUA&Expires=1607040777&Signature=XhHWnAQPSZAK9x0hJkDbcxjLAEq%3D

- Lámpara del laboratorio:

<https://sketchfab-prod-media.s3.amazonaws.com/>

- Escritorio del laboratorio:

https://static.free3d.com/models/123d/printable_catalog/10120_LCD_Computer_Monitor_v01_L3.123cec7599fe-42d9-472b-b37a-9b96a100f29d.zip

- Mesa del laboratorio:

<https://sketchfab-prod-media.s3.amazonaws.com/archives/c229ec7fe42440f1847ea69b13082328/source/06664e73760d428aa4751a4a06a86ef7/hospital-table-mc-model.zip?>

- Mesita del laboratorio:

<https://sketchfab-prod-media.s3.amazonaws.com/archives/e3ef6fd54af440efa31984214dd6f11d/source/0eb8fe763ef14d68a98903377dd3b9fd/adjustable-bedside-table.zip?>

- Instrumentos médicos:

<https://sketchfab-prod-media.s3.amazonaws.com/archives/2971a44d334b45ff926792e92a6bcee7/source/b72ebb696bd241abafb4a8b987e5be2a/surgical-instrument.zip?>

- Pack Hospital Horror:

https://s3.amazonaws.com/tsinternal_standard/Internal/2011/09/26_09_07_55/Hospital_3D_Model_Pack.zip?AWSAccessKeyId=AKIAIJ6EELKKOTA5UAUA&Expires=1606486541&Signature=CdStubjp%2FSANgADS5ov%2F%2FbGp0Mk%3D

- Camas del laboratorio:

https://download1.3dcadbrowser.com/models/2447516812121951625113137821116316826114191185121105/73915_obj.zip