Escuela Politécnica Superior de Jaén
Grado en Ingeniería Informática

Universidad de Jaén
Escuela Politécnica Superior
de Jaén

TRABAJO FIN DE GRADO

REMAKE DEL VIDEOJUEGO CLÁSICO SPACE INVADER

Alumno

José Carlos Miras Cabrera

Tutor

Carlos J. Ogayar Anguita
(Departamento de Informática)

Febrero, 2022

(Página intencionalmente en blanco)

Universidad de Jaén

Departamento de Informática

Don Carlos J. Ogayar Anguita, tutor del Trabajo Fin de Grado titulado: '**Remake del videojuego clásico Space Invader**', que presenta Don José Carlos Miras Cabrera , otorga el visto bueno para su entrega y defensa en la Escuela Politécnica Superior de Jaén.

Jaén, Febrero de 2022

El alumno:

MIRAS CABRERA JOSE CARLOS - 26529772P

Firmado digitalmente por MIRAS CABRERA JOSE CARLOS - 26529772P
Fecha: 2022.02.15 18:07:47 +01'00'

José Carlos Miras Cabrera

El tutor:

OGAYAR ANGUITA CARLOS JAVIER - 26039170L

Firmado digitalmente por OGAYAR ANGUITA CARLOS JAVIER - 26039170L
Nombre de reconocimiento (DN): c=ES, serialNumber=IDCES-26039170L, givenName=CARLOS JAVIER, sn=OGAYAR ANGUITA, cn=OGAYAR ANGUITA CARLOS JAVIER - 26039170L
Fecha: 2022.02.16 12:35:21 +01'00'

Carlos J. Ogayar Anguita

(Página intencionalmente en blanco)

Agradecimientos

Mis agradecimientos a mis familiares amigos que me han ayudado y levantado el ánimo para continuar en los momentos más complicados del proyecto y cuando las cosas no avanzaban al ritmo que deberían.

Agradecer también a mi tutor, que ha estado ahí siempre que lo he necesitado y revisando el trabajo además de recomendarme las cosas que se podrían añadir o mejorar.

FICHA DEL TRABAJO FIN DE TÍTULO	
Titulación	Grado en Ingeniería Informática
Modalidad	Proyecto de Ingeniería
Especialidad (solo TFG)	Tecnologías de la Información
Mención (solo TFG)	Sistemas Gráficos
Idioma	Español
Tipo	General
TFT en equipo	No
Autor/a	José Carlos Miras Cabrera
Fecha de asignación	25/03/2021
Descripción corta	En este trabajo se tratará de hacer una reinterpretación de un videojuego clásico ya existente con las tecnologías y herramientas actuales. En este caso dicho juego es el conocido Space Invaders. El objetivo es que quede un juego que pueda disfrutarse actualmente y que se sienta relativamente actual, pero sin perder la esencia de lo que fue en su día dicho juego.

NORMAS APLICADAS EN ESTE DOCUMENTO	
LOCALES	
TFT-UJA:2017	Normativa de Trabajos Fin de Grado, Fin de Máster y otros Trabajos Fin de Título de la Universidad de Jaén (Normativa marco UJA aprobada en Consejo de Gobierno)
TFT-EPSJ:2017	Normativa sobre Trabajos Fin de Grado y Fin de Máster en la Escuela Politécnica Superior de Jaén (Normativa EPSJ aprobada en Junta de Escuela)
TFT-EPSJ	Criterios de evaluación y normas de estilo para TFG y TFM de la Escuela Politécnica Superior de Jaén
NACIONALES E INTERNACIONALES	
ISO 2145:1978	Documentación - Numeración de divisiones y subdivisiones en documentos escritos
UNE 50132:1994	Traducción de la ISO 2145
APA 6ª edición	Estilo de referencias y citas de APA (American Psychological Association)

NORMAS UTILIZADAS COMO BASE O REFERENCIA	
NACIONALES	
UNE 157001:2014	Criterios generales para la elaboración formal de los documentos que constituyen un proyecto técnico
UNE 157801:2007	Criterios generales para la elaboración de proyectos de sistemas de información
*Estas normas se han utilizado **como base o referencia** para la inclusión de algunos contenidos y definiciones sobre elaboración de proyectos, entendiendo como **proyecto** la documentación consensuada entre una empresa y un cliente, que da lugar al perfeccionamiento de un contrato para la elaboración de una obra o la prestación de un servicio. Por consiguiente, no debe esperarse la aplicación de estas normas en cuanto a la completitud de los contenidos ni a la organización de los mismos.*	

Contenido

Índice de ilustraciones

Índice de Tablas

1 ESPECIFICACIÓN DEL TRABAJO

En este capítulo se presenta la especificación del trabajo, con una estructura y contenidos **inspirados** en los criterios y recomendaciones que establece la norma UNE 157801:2007 - "*Criterios Generales para la elaboración de proyectos de Sistemas de Información*". A lo largo del documento se utilizarán términos y acrónimos cuya descripción aparecen en el apartado **6 (Definiciones y abreviaturas).**

1.1 Introducción

En la actualidad, los videojuegos se están haciendo un hueco importante en la industria del entretenimiento. El pasado año, y potenciado por el confinamiento, los videojuegos generaron 127 000 millones de dólares. La mayor parte de esta suma proviene de videojuegos denominados "free-to-play", juegos gratuitos que generan ingresos a partir de micro pagos que venden cosméticos o diferentes mejoras temporales para avanzar más rápido en el juego.

Esta joven industria comenzó a surgir sobre las décadas de los 60-70 con juegos como Pong, Space Invader y Asteroids. Desde entonces, los videojuegos han evolucionado mucho tanto en el ámbito gráfico como en complejidad de mecánicas, narrativa etc. Es por esto que, dado que los videojuegos más antiguos se van quedando desfasados, unas de las tendencias actuales es la de realizar "remakes" o "remasters" de los mismos.

Las empresas aprovechan los juegos que ya tienen desarrollados y el factor nostalgia de los jugadores para darle un "lavado de cara" a dicho juego y añadirle ciertas mejoras para así volver a sacar ese juego al mercado. Esta práctica es cada vez más común porque, como es lógico, parte del trabajo ya está hecho: diseño de personajes, escenarios, guion de la historia, mecánicas jugables y claro esto se traduce en poder sacar juegos más rápidamente a menor coste y con mayores beneficios. Generalmente hay una ligera confusión entre lo que es un remaster y lo que es un remake.

Denominamos remaster cuando se adapta un videojuego modificando la forma en que se ve, modelados, texturas, iluminación o bien, que modifica sonidos y música. Es decir, el juego será exactamente el mismo en todo lo demás: mecánicas, historia, etc. Últimamente las remasterizaciones son muy comunes incluso es juegos más recientes porque se pretende sacar al mercado en una consola de nueva generación.

Un remake implica un cambio más radical: nuevas mecánicas, modificaciones de la ya existentes además por supuesto, de las mejoras gráficas y sonoras pertinentes. La idea es que el juego conserve la esencia del original y mejorarlo para hacerlo más atractivo.

En este proyecto nos centraremos en realizar un remake al conocido y afamado Space Invader. Este videojuego de arcade hizo su primera aparición en 1978 y fue diseñado por Tomohiro Nishikado. Se trata de un “mata marcianos” clásico en el que el jugador dirige un cañón y trata de defenderse de una invasión alienígena eliminándolos lo más rápido posible antes de que estos le alcancen. Este ciclo se repite indefinidamente y el objetivo principal es conseguir la puntuación más alta posible.

1.2 Objetivos del trabajo

El objetivo de este trabajo no es otro que el de generar un videojuego que recoja las ideas principales del ya existente Space Invader y le añada los cambios necesarios para poder competir con juegos más actuales. Debido a que el desarrollo de un videojuego, por poco ambicioso que sea requiere de mucho tiempo y un amplio número de personas especialistas en distintos sectores (música, animación, diseño, programación…), este proyecto será un prototipo, una versión beta del producto que permitirá probar la jugabilidad y las distintas mecánicas para determinar lo más importante en este tipo de producto: que sea divertido y funcione como se esperaba.

1.3 Antecedentes y estado del arte

Space Invader salió como un juego arcade en Japón fabricado y vendido por Taito co. Es uno de los primeros juegos del género mata marcianos y uno de los más importantes de la historia. Para su diseño Nishikado se inspiró en Breakout, la guerra de los mundos y Star Wars.

En él los jugadores estaban al mando de un cañón que se movía únicamente de izquierda a derecha y su tarea era la de defenderse de oleadas de marcianos que llegaban desde arriba. Una de las peculiaridades es que solo podía haber una bala del jugador simultáneamente en pantalla, es decir, no se podía volver a disparar hasta que esta, impactaba o se salía de nuestra vista.

Había 3 tipos de marcianos que otorgaban puntuaciones diferentes al matarlos y eventualmente aparecía un ovni en la parte superior moviéndose en horizontal y este era el enemigo que más puntación daba. El comportamiento de los marcianos era simple, aparecían seis filas de marcianos (2 de cada tipo) y se movían lentamente en horizontal todos en la misma dirección como un bloque, cuando alguno de ellos alcanzaba un extremo todos bajaban una fila y la velocidad de movimiento aumentaba cada vez que descendían. Además de esto había una serie de 4 barreras que se rompían poco a poco si eran alcanzados por proyectiles tanto enemigos como aliados. El jugador disponía de 3 vidas o lo que es lo mismo si era alcanzado más de tres veces el juego terminaba, ¡Game Over!

Ilustración 1. Pantalla de juego de Space Invaders

Aunque su premisa es aparentemente sencilla para los entandares de la época actual, sentó un precedente es su día y podemos considerarlo uno de los precursores de los videojuegos modernos. Su éxito fue abrumador, tanto que se cuenta que hubo una escasez de monedas de 100 yenes en Japón debido a que fueron utilizadas en el arcade para jugar a este juego y el gobierno se vio obligado a aumentar el número de las mismas para hacer frente a esta emergencia.

Al poco tiempo de su lanzamiento, comenzaron a aparecer clones tales como Space Invader Deluxe ya que Space Invaders no estaba sujeto a derechos de copyright. El juego ya ha sufrido numerosas adaptaciones a lo largo de los años en el que cabe destacar el más reciente para Nintendo DS y PSP en 2008 que posteriormente también salió para PC. Para nuestro proyecto tomaremos algunas de las ideas que implementó este remake ya que son muy interesantes como veremos más adelante.

1.4 Descripción de la situación de partida

1.4.1 Descripción del entorno actual

En cuanto al entorno actual de la franquicia de Space Invader, hace más de una década se publicó Space Invaders Extreme para conmemorar su 30 aniversario. Era un remake de estilo clásico que incluía un montón de ideas interesantes y era bastante divertido y adictivo.

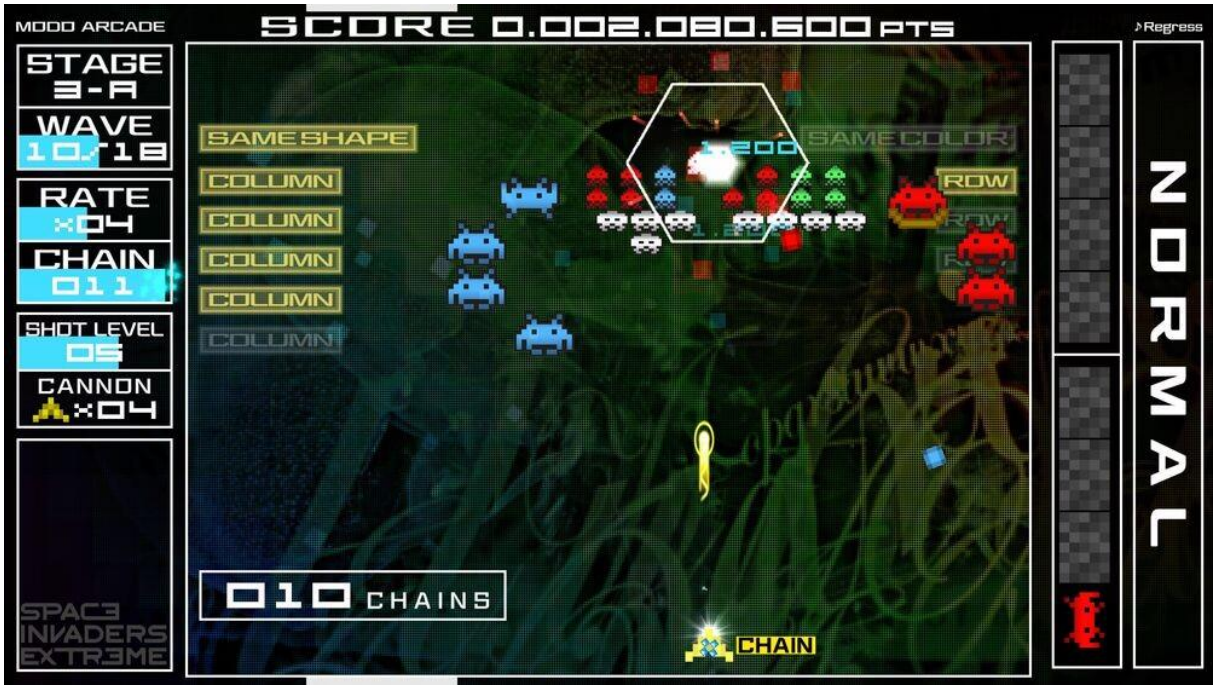


Ilustración 2. Space Invader Extreme

Para su 40 aniversario anunciaron Space Invader Gigamax, pero esta fue algo especial y exclusivo de Japón ya que se jugaba en una pantalla gigante colocada en dicho país durante un periodo de tiempo limitado y era jugable con hasta diez jugadores simultáneos.

Ilustración 3. Space Invader Gigamax

Por último, en 2020 se estrenó un recopilatorio de los anteriores más un juego de móvil que era una mezcla de Space Invader y Arkanoid, otro juego muy conocido. El videojuego exclusivo y multijugador se redujo a 4 jugadores para adaptarse al tamaño de las pantallas de los dispositivos actuales.

1.4.2 Resumen de las deficiencias y carencias identificadas

A pesar de todos estos juegos aún hay cosas que se podrían añadir o mejorar para hacer una experiencia aún más satisfactoria y variada y mantener a los jugadores activos durante cientos de horas. Para ello partiremos de la base de Space Invader Extreme que es el remake que más cambios incluye y más ideas nos da para seguir expandiéndose.

1.5 Requisitos iniciales

Antes de comenzar con un proyecto se debe tener claro ciertas cuestiones relacionadas con el mismo:

- ¿A quién va dirigido?
- ¿Qué motor de videojuegos se utilizará?
- ¿En qué plataformas estará disponible?

1.5.1 Audiencia

Habitualmente, en la industria se usa un sistema conocido como PEGI (“Pan European Game Information”) para dotar a sus productos sobre la edad apropiada para su consumo. Este sistema consta de dos tipos de etiquetas uno en referente a la edad recomendada y el otro relativo a contenido susceptible.

El diseño de los logotipos sobre la edad contiene un código de color basado en las luces de seguridad vial o semáforos. Siendo el verde para edades más tempranas, amarillo para intermedio y rojo para adultos.

Ya que nuestro juego es un remake, en principio se tomará la clasificación por edades del original a no ser que se incluya algún contenido más adulto de tipo lenguaje soez, desnudos, violencia explícita etc. Space Invader tiene una clasificación de PEGI 3. Esta es la etiqueta que se utiliza en los juegos adecuados para todas las edades. Estos no deben contener palabras malsonantes, contenido que pueda asustar a los pequeños ni personajes que puedan ser identificables en la vida real. El siguiente icono deberá acompañar al juego ya sea en la portada y contraportada del juego en físico o en su distribución de forma digital.

Ilustración 4. Logo PEGI 3

1.5.2 Motor de videojuegos

A la hora de crear un videojuego, es muy aconsejable, por no decir imprescindible la utilización de un motor de videojuegos. Un motor de videojuegos es un término que hace referencia a una serie de librerías y rutinas de programación que permiten el diseño, creación y representación de un videojuego. También capacita a un programador a desarrollar su producto sin tener conocimientos del hardware de la plataforma a la que dirigida su obra.

Los aspectos que más se tienen en cuenta a la hora de elegir un motor son sus capacidades gráficas, es decir su potencial para mostrar imágenes en 2D y 3D, así como cálculo de iluminación, polígonos, texturas…

Otro punto a tener en cuenta es la facilidad de uso y aprendizaje del mismo y su capacidad para exportar el juego a las distintas plataformas. Existen numerosos motores de videojuegos, pero los dos más conocidos y que ofrecen una versión gratuita son Unreal Engine y Unity.

1. Unreal Engine

Ilustración 5. Logo Unreal Engine

Unreal Engine es un motor desarrollado por la compañía Epic Games que es, además, la productora de uno de los juegos en línea más populares del momento, Fornite. La versión más reciente de este motor es Unreal Engine 4 y está pendiente de publicarse en este 2021 su versión 5.

Este motor se puede usar de manera completamente gratuita siendo requerido, en caso de publicar el juego y obtener ingresos, pagar a Epic Games un 5% en caso de superar la cifra de 1.00.000$ totales. Esto hace un tiempo no era así, había que pagar el 5% al superar los 3.000$ trimestrales por lo que actualmente es una mejor opción para un desarrollo pequeño de lo que era anteriormente.

Unreal Engine es el motor gráfico más potente del mercado y grandes títulos triple A han sido construidos con él. La gran potencia de este motor ha hecho dar el salto a otros campos tales como la arquitectura, el cine, simuladores, etc. El lenguaje

utilizado es C++, por lo que requiere conocimiento de este lenguaje para la programación de los scripts.

Algunos de los aspectos más negativos es que debido a su complejidad su curva de aprendizaje es bastante pronunciada y su comunidad no es tan grande como podría ser la de Unity. También tiene serios problemas de rendimiento a la hora de ejecutar juegos para móviles. Actualmente Unreal Engine 4 permite crear juegos para las siguientes plataformas:

- IOS
- Android
- Linux
- Windows
- MacOS
- SteamOS
- SteamVR/HTC Vive
- Oculus Rift
- OSVR
- Samsung Gear VR
- Nintendo Switch
- HTML5
- Xbox One
- PS4
- Playstation VR
- Google Daydream

2. Unity

Ilustración 6. Logo de Unity

Unity es una herramienta de desarrollo de videojuegos creada por Unity Technologies. Unity proporciona motores de render, Físicas 2D y 3D, audio, animación, etc. Además, su interfaz es sencilla e intuitiva y su documentación es excelente. Esto sumado a la gran comunidad que posee hace de ella una de los motores con una curva de aprendizaje más fácil del mercado.

Es compatible con el lenguaje de programación C# por lo que se recomienda conocimientos de él para utilizarlo, aunque Unity ha desarrollado una herramienta llamada Bolt que permite a los usuarios desarrollar scripts de forma visual sin líneas de código. Por si fuera poco, dispone de numerosos servicios que se ofrecen dependiendo del plan contratado como almacenamiento en la nube, capacidad de trabajar en proyectos compartidos, monetización de los juegos y un largo etc.

Unity pone a nuestra disposición la Assets store, esto es una tienda de assets, que son los elementos de lo que se compone un videojuego. Un asset puede ser desde sonidos, modelos 2D/3D, sprites hasta cosas más complejas como animaciones, scripts, etc. Estos assets son realizados tanto por la propia Unity Technologies como por la comunidad y se publican allí ya sea de forma gratuita o por un precio establecido. Hay numerosos assets gratuitos que son muy útiles para dar los primeros pasos con esta plataforma.

En cuanto al precio, Unity dispone de tres tipos de licencias según el uso que se le vaya a dar:

- Personal: Para usuarios o pequeñas empresas que no hayan facturado más de 100.000$ en los últimos 12 meses. Esta licencia es la más básica y es completamente gratuita pero no dispone de algunos servicios más avanzados.
- Plus: En este caso elegiremos este plan si los ingresos obtenidos el último año no superan los 200.000$ tiene un precio anual de 399$ o mensual de 40$
- Pro: Esta opción tiene un precio de 1800$ anuales o si decidimos pagarlo de manera mensual nos costaría 150$/mes. La licencia pro será la adecuada si la limitación de los ingresos no nos permite escoger ninguna de las anteriores, pero no somos una empresa suficiente grande para aprovechar la licencia Enterprise.
- Entreprise: Esta licencia consta de diversos servicios que la Pro no tiene y se caracteriza por incluir 10 activaciones, es decir que podría usarlo en 10 ordenadores simultáneamente. Solo tiene un plan anual y este asciende a 2.000$.

En cuanto a plataformas Unity soporta más de 25 entre las que se encuentran:

- Windows
- UWP
- MacOS
- Linux
- FB Gameroom
- WebGL
- Android
- iOS
- Amazon Fire OS
- PS4

- PSVita
- Xbox One
- Wii U
- 3DS
- Nintendo Switch
- Stadia
- Oculus Rift
- Steam VR
- PSVR

1.5.3 Tecnología utilizada

Para este proyecto utilizaremos Unity en su versión 2020.1.4f1. Dado que el tiempo es limitado es la mejor opción por su facilidad de aprendizaje, su excelente documentación y la posibilidad de obtener los componentes que nos harán falta en su assets store. Otra razón para elegir Unity en lugar de Unreal Engine es que el videojuego que tenemos en mente no requiere una calidad gráfica demasiado alta y además tenemos intención de llevarlo a dispositivos móviles y como hemos comentado Unreal Engine no optimiza demasiado bien sus juegos para esta plataforma. Como lenguaje de programación usaremos c#.

El pc en que será desarrollado es un Asus TUF Dash F15 con un procesador Intel Core I7 11370H, RAM DDR4, 3200Mhz de 16GB, un disco duro SSD de1TB y una tarjeta gráfica RTX 3060.

1.5.4 Plataforma

Este proyecto será multiplataforma, es decir podrá jugarse en pc o en dispositivos móviles. Esto no hará tomar una serie de decisiones a lo largo del proyecto como optimizaciones de rendimiento y una interfaz adaptada para móviles dado que en este caso los controles serán parte de la interfaz por la carencia de estos de botones.

1.6 Hipótesis y restricciones

El TFG se define como una asignatura de 12 créditos, lo que supone que la duración total del proyecto será de 300 horas, incluyendo todas las etapas del ciclo de vida, con la excepción del mantenimiento. Por lo tanto, la principal restricción aplicable es la limitación de la duración del trabajo. Por consiguiente, es posible que algunas de las características definidas para el proyecto no se puedan llevar a cabo y quedaran propuestas como mejoras para más adelante.

1.7 Estimación del tamaño y esfuerzo

Ya que el presente proyecto es un TFG, no existen restricciones de tipo económico, sino de tipo temporal (un número aproximado de horas).De esta forma, los cálculos de tamaño del proyecto están supeditados por el tiempo disponible. En cuanto al esfuerzo, se dispone de tan solo un efectivo (la persona autora del trabajo).

1.8 Definición y características

En este apartado desarrollaremos una a una, todas las ideas y mecánicas que nos gustaría ver en el producto final. Dadas las limitaciones tanto de tiempo como de recursos humanos, puede que algunas de estas características no estén incluidas al final de este proyecto por lo que serán propuestas como posibles mejoras.

En primer lugar, la mecánica principal de juego será defenderse de una invasión de marcianos que irá apareciendo en una serie de oleadas e intentará destruirnos por todos los medios de lo que dispongan. El jugador controlará una nave (cuyo diseño podrá modificar entre una serie de ellos si han sido desbloqueados) que podrá moverse de izquierda a derecha hasta los bordes de la pantalla. Deberá disparar a los enemigos que generalmente, salvo alguna excepción, se acercaran lentamente hacia el jugador.

1.8.1 Condiciones de éxito y de fracaso

Dado que el juego cuenta con dos modos de juego bien diferenciados los criterios de éxito o fracaso pueden variar de uno a otro. Por lo tanto aquí solo describiremos de forma genérica aquellos objetivos y condiciones de derrota que serán compartidos por ambos modos.

1.8.1.1 Condición de victoria

Nuestro objetivo será sobrevivir el mayor tiempo posible y defender la galaxia exterminado a los malvados aliens. Cada tipo de alien nos aportará una puntuación base específica (esta podrá multiplicarse mediante modificadores y combos) por lo tanto trataremos también de obtener la mayor puntuación posible.

1.8.1.2 Condición de derrota

La partida termina cuando el jugador pierda todas sus vidas. Comenzará con 3 vidas y podrán perderse de distintas formas:

- Siendo alcanzado por un proyectil enemigo.
- Siendo alcanzado directamente por un enemigo.
- Si algún enemigo toca la línea inferior de la pantalla.

Al perderse una vida, si aún le queda alguna, el jugador reaparecerá con un breve tiempo de invulnerabilidad. Si la vida se perdió por contacto directo o por llegada del enemigo a la parte inferior todos los enemigos se reubicarán unos niveles más arriba ya que de otra forma sería imposible que el jugador saliera ileso de esa oleada. Una excepción a esta regla consistiría en que el contacto se hubiera producido con un enemigo que se mueve libremente independiente del resto (esto se detallará más adelante en la sección de enemigos).

1.8.2 Modos de juegos

El juego dispondrá de dos modos de juego: Arcade y aventura.

1.8.2.1 El modo aventura

En el modo historia el jugador se enfrentará a fases predefinidas es decir cada fase es igual cada vez que la juegues, pero solo podrá jugarse una fase si se han completados las anteriores. En cada fase se definirá ciertos parámetros como el número de oleadas y que restricciones o mecánicas implementará por ejemplo puede que haya fases con un tiempo determinado para superarla o que se requiera una determinada puntuación.

A medida que se va avanzando en la aventura se podrán ir desbloqueando ciertas mejoras que harán el videojuego mucho más variado y divertido: nuevos enemigos o modificaciones de los anteriores, mejoras temporales como velocidad de movimiento o mejoras del arma, escudo, etc. Una vez que se ha desbloqueado algo esto podrá encontrarse en futuras fases o en el modo arcade.

Eventualmente cada cierto número de fases nos encontráremos que la última oleada es un enemigo inusual conocido como jefe o boss. Los jefes serán más complicados de eliminar y habrá que buscar su punto débil y será aconsejable aprenderse su patrón de ataque para que no nos alcance.

1.8.2.2 Modo arcade

Este modo de juego es el clásico utilizado por el juego original en 1978. El jugador se enfrentará a oleadas de enemigos que se sucederán de forma indefinida y serán generadas aleatoriamente. Para generar oleadas se recorre cada posición de la horda y se calcula si aparecerá un enemigo en ella, y si este es favorable se elige cuál de los disponibles aparecerá y su color. Por último, se decidirá si este alien llevará alguno de los modificadores disponibles. Todas estas probabilidades podrán variar en función de la oleada en la que se encuentre para ir incrementado la dificultad.

El objetivo principal será el de sobrevivir el mayor tiempo posible y obtener la mayor puntuación posible. En este modo de juego podrán encontrarse de manera aleatoria todas las mejoras y enemigos que se hayan desbloqueado hasta el momento

en el modo aventura. Esto hará que a mayor cantidad de contenido desbloqueado, más variado podrá ser este modo y más divertido. Esto proporcionará un incentivo al jugador para jugar el modo aventura y completar todos los logros que ofrece el juego.

1.8.3 Sistema de puntuación

El sistema de puntuación funcionará de la siguiente forma:

- Cada enemigo otorgará una puntuación base predefinida que se multiplicará por el indicador de bonus actual.
- Matar enemigos consecutivos aumentará el indicador de cadena o combo.
- El indicador de cadena solo sumará cuando un enemigo muera.
- El indicador de cadena se resetea si pasan 5 segundos sin que un proyectil impacte sobre un enemigo. Si un proyectil impacta sobre un enemigo o escudo, pero este no muere simplemente se restablece el tiempo que falta hasta que acabe la cadena.
- Cada cierto nivel de cadena se aumenta el nivel de bonus.
- No hay un máximo de nivel de cadena, pero a partir de 100, el bonus no aumentará más.
- Al perder la cadena se restablece también el indicador de bonus.

A continuación, se describe a qué nivel de cadena se obtiene el siguiente incremento del multiplicador de puntuación.

BONUS	CADENA
x02	02
x03	05
x04	08
x05	12
x06	16
x07	20
x08	25
x09	30
x10	36
x11	42
x12	50
x13	60
x14	72
x15	86
x16	100

Tabla 1. Relación bonus-cadena

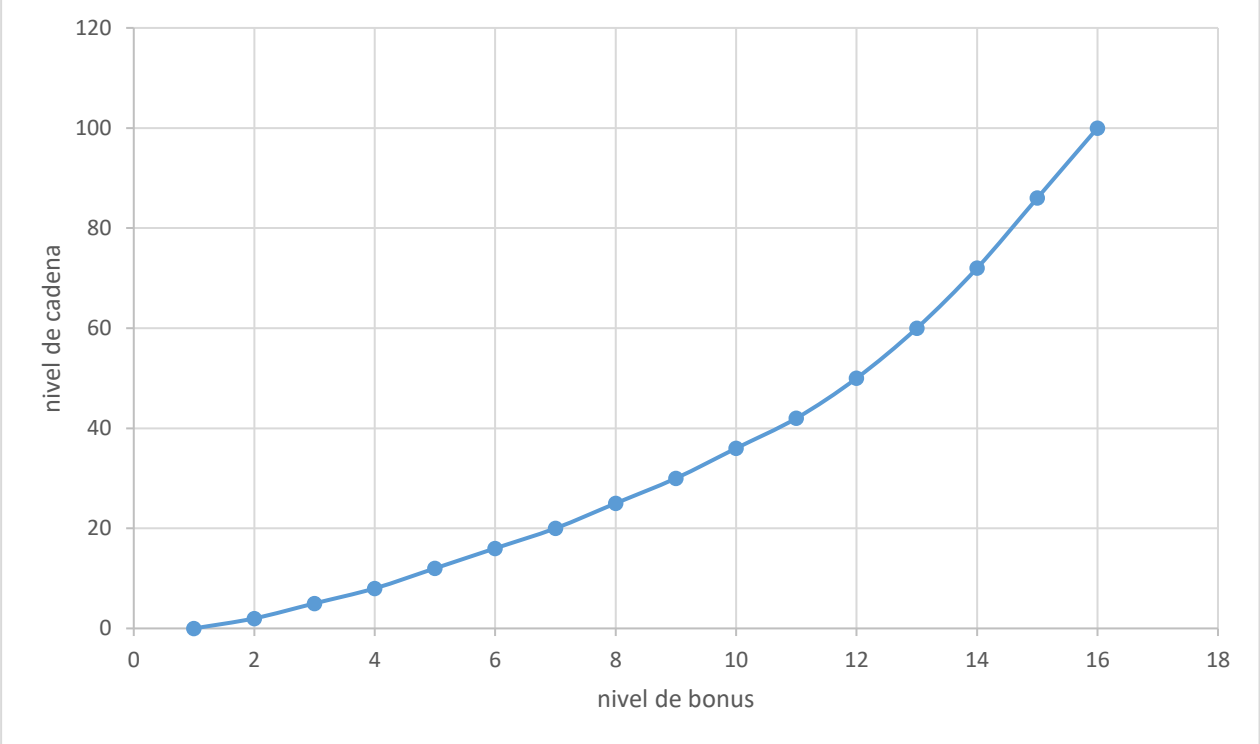


Ilustración 7. progresión del nivel de bonus

1.8.4 Enemigos

A lo largo del videojuego nos encontraremos con diversos enemigos que intentarán acabar con nosotros. Hay que diferenciar enemigos normales de los enemigos únicos o jefes finales de ciertas fases del modo aventura.

Los enemigos normales son los que pueden encontrarse en cualquier fase del modo aventura u oleada de modo arcade, mientras que los jefes solo se encontrarán en la última oleada de ciertas fases del modo aventura (cada tres fases habrá un jefe). Cada jefe tendrá una mecánica única para eliminarlo, por ejemplo, destruir con nuestros disparos sus puntos débiles. El comportamiento de cada jefe se detallará en su sección correspondiente ya que no comparten ningún parecido unos con otros.

En cuanto a diseño se optará por un estilo que se asemeje a los del videojuego original ya que estos eran muy característicos del mismo y cambiarlos podría alejar este proyecto de su objetivo principal, que recuerde al Space Invader de 1978.

1.8.4.1 Comportamiento de enemigos normales

A grandes rasgos podríamos decir que existen dos tipos de enemigos normales según su comportamiento:

Movimiento en manada: estos enemigos se distribuirán en filas y columnas con un máximo de 5 y 18 respectivamente. Estos enemigos se moverán todos al unísono como si fueran un bloque. Comenzarán moviéndose hacia la derecha hasta que alguno de los integrantes toque el borde de la pantalla. En ese momento todos los miembros de la manada descenderán una fila y cambiarán su sentido de la marcha. Cada vez que desciendan, la velocidad de movimiento horizontal se incrementará un poco. Cada cierto tiempo el alien disparará en caso de que se le permita. Ya que habrá un control para que no se llene la pantalla de proyectiles y que sea imposible de esquivar. El límite de proyectiles enemigos en pantalla será establecido en 3 pudiendo aumentarse este límite conforme se avance en la partida para ir incrementado la dificultad.

Movimiento libre: algunos enemigos podrán moverse de forma independiente al resto con algún patrón específico. Existirán varios patrones de movimiento libre. En el modo aventura como las fases están más diseñadas a medida podrían crearse situaciones en las que un enemigo de la manada cambia su comportamiento a libre si se cumple alguna condición.

Existirán 4 formas básicas de enemigos más los tres jefes que son únicos. Las formas básicas son las del juego original: Calamar, cangrejo, pulpo y ovni. Cada uno tendrá unas variantes que modificaran su aspecto. Las tres primeras formas tendrán dos patrones, normal (cuatro colores) y enfurecido (tres colores). Durante el proyecto diferenciaremos entre enemigos básicos, variantes y jefes. Los que poseen un patrón de color normal serán considerados básicos y los de patrón enfurecido serán variantes ya que cada uno de ellos tienen alguna característica distintiva.

Ovnis: solo tendremos 4 y serán considerados variantes ya que su movimiento es siempre de tipo libre (nunca aparece en una horda, sino que aparece espontáneamente en la parte superior, recorre la pantalla en horizontal y desaparece) y por tanto requiere un desarrollo específico.

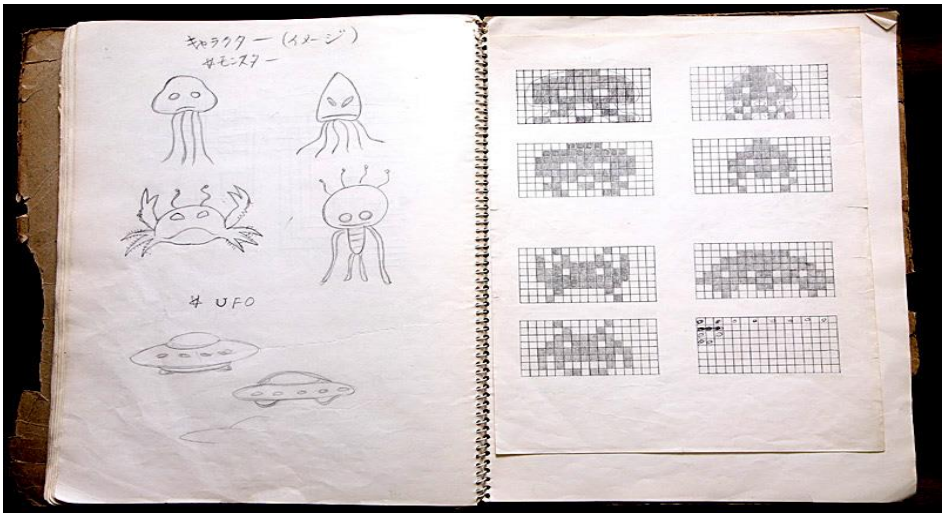


Ilustración 8. Boceto original de los enemigos

Calamar		
Rojo	Versión normal de color rojo.	Puntuación:150
Azul	Versión normal de color azul.	Puntuación:150
Verde	Versión normal de color verde.	Puntuación:150
Blanco	Versión normal de color blanco.	Puntuación:150
Rojo enfurecido	Versión modificada de color rojo, se caracteriza por cambiar su disparo por una bomba. Si el jugador entra en el rango de explosión morirá.	Puntuación:200
Azul enfurecido	Versión modificada de color azul, se caracteriza por cambiar su disparo por un rayo láser que se mantendrá durante un breve tiempo.	Puntuación:200
Verde enfurecido	Versión modificada de color verde, se caracteriza por cambiar su disparo por tres proyectiles simultáneos (este solo contará como uno en el recuento total de proyectiles en escena).	Puntuación:200

Tabla 2. Variantes de calamar

Cangrejo		
Rojo	Versión normal de color rojo.	Puntuación:100
Azul	Versión normal de color azul.	Puntuación:100
Verde	Versión normal de color verde.	Puntuación:100
Blanco	Versión normal de color blanco.	Puntuación:100
Rojo enfurecido	Versión modificada de color rojo, se caracteriza por tener más vida que los demás. Morirá si recibe tres disparos.	Puntuación:150
Azul enfurecido	Versión modificada de color azul, se caracteriza por tener un escudo que refleja el primer proyectil que lo alcance.	Puntuación:150
Verde enfurecido	Versión modificada de color verde, se caracteriza por que al morir aparecerán dos cangrejos normales de color verde.	Puntuación:150

Tabla 3. Variantes de cangrejo

Pulpo		
Rojo	Versión normal de color rojo.	Puntuación:70
Azul	Versión normal de color azul.	Puntuación:70
Verde	Versión normal de color verde.	Puntuación:70
Blanco	Versión normal de color blanco.	Puntuación:70
Rojo enfurecido	Versión modificada de color rojo, se caracteriza por lanzar un proyectil que al ser destruido se divide en dos.	Puntuación:120
Azul enfurecido	Versión modificada de color azul, se caracteriza por volverse invulnerable cada cierto tiempo se representa con un ligero cambio físico que lo hace semitransparente.	Puntuación:120
Verde enfurecido	Versión modificada de color verde, se caracteriza por disparar un proyectil que si llega a la altura del jugador libera un rayo horizontal que destruirá nuestra nave. Por lo tanto, la estrategia será destruir este proyectil antes de que llegue abajo.	Puntuación:120

Tabla 4. Variantes de pulpo

OVNI		
Rojo	Versión normal de color rojo.	Puntuación: 300
Azul	Versión normal de color azul.	Puntuación:300
Verde	Versión normal de color verde.	Puntuación:300
Blanco	Versión normal de color blanco.	Puntuación:300

Tabla 5. Variantes de OVNI

1.8.5 Modificadores de enemigos

Además de los distintos tipos de enemigos, se añade un nivel de dificultad más al desbloquear estos modificadores. Cualquier enemigo puede aparecer con uno de estos modificadores lo que le añade un extra de puntuación y le otorga una habilidad especial o cambiar su comportamiento que son los siguientes:

Modificadores		
Escudo reflector	Devuelve hacia abajo los proyectiles que impacten en él. Se rompe a los 2 usos.	Puntuación: 100
Escudo fragmentable	Escudo que protege de los ataques de jugador. Se va consumiendo visualmente con cada uso. Solo defiende de 3 impactos.	Puntuación: 150
Kamikaze	El enemigo se lanzará hacia abajo a velocidad constante (ignorando el movimiento del resto de la manada si estuviera en una) al recibir daño.	Puntuación: 200

Tabla 6. Tipos de modificadores

1.8.6 Mejoras de arma temporal

Estas mejoras podrán aparecer eventualmente tras ser desbloqueadas. Serán objetos que los enemigos podrán dejar caer al ser eliminados y se obtendrán si el jugador los recoge cada arma tendrá un tiempo de uso máximo. Si antes de que el tiempo finalice se recoge una nueva mejora del mismo tipo el tiempo de uso volverá a ser el máximo. El método que definirá su aparición es el siguiente:

- Cada mejora estará ligada a un color de enemigo específico.
- Los enemigos normales tendrán un 5% de posibilidades de soltar una mejora de su color y los enfurecidos un 10%.
- Esta probabilidad aumenta un 2% por cada enemigo consecutivo eliminado del mismo color es decir por cada incremento en la cadena de color.

Mejoras		
Lanzagranadas	Ligado al color rojo esta mejora permite lanzar granadas que eliminan a varios enemigos a la vez a su alrededor desde la posición en la que explota.	Desbloqueo: Completando jefe 1 Tiempo: 15 segundos
Disparo triple	Mejora ligada al color verde permite disparar tres proyectiles que avanzan simultáneamente cubriendo una mayor área.	Desbloqueo: completando jefe 2 Tiempo 15 segundos
Láser	Mejora ligada al color azul. Permite disparar un rayo láser que atraviesa a los enemigos. Los escudos reflectores no devuelven el rayo, pero si lo frenan hasta que son destruidos protegiendo también a todos los que se encuentren por encima de él. El disparo será continuo mientras se mantenga pulsado el botón.	Desbloqueo: completando jefe 3 Tiempo: 10 segundos

Tabla 7. Tipos de armas

2 PLANIFICACIÓN DEL TRABAJO

2.1 Metodología de desarrollo de software

El primer paso a realizar en la elaboración de un proyecto software es la elección de una metodología de desarrollo. Para este proyecto se ha optado por una metodología de desarrollo ágil basada en interacciones. Al final de cada iteración se obtendrá un entregable o incremento del producto final que será susceptible de ser testeado y enseñado al cliente en caso de que lo solicite. Esta metodología es realmente útil ya que podremos ir viendo cómo avanza el proyecto al final de cada sprint y poder ir detectando y sorteando los diversos problemas que vayan surgiendo durante el desarrollo en etapas muy tempranas de este. Para llevarla a cabo, lo primero es dividir el proyecto es pequeñas partes que puedan ser manejadas con relativa independencia. A esto lo llamaremos listado de requisitos.

2.2 Listado de requisitos

Requisitos	Duración (horas)
1. **El usuario debe ser capaz de controlar a la nave en movimiento horizontal y disparar con ella.**	4
2. **Los enemigos se moverán horizontalmente hasta que alguno toque el borde de la pantalla, momento en el cual todos descenderán una fila y cambiaran el sentido del movimiento. Además, serán capaces de lanzar proyectiles eventualmente.**	10

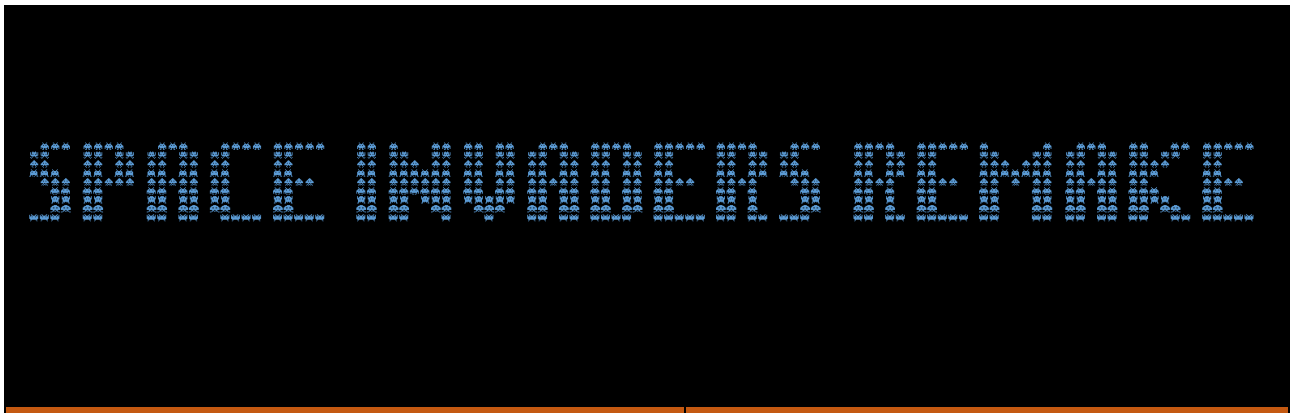
3. **Existirán enemigos que se muevan de forma diferente al resto como los OVNI o algún enemigo especial dadas unas condiciones.**	8
4. **Una serie de capas de fondo con motivo espacial se desplazarán verticalmente hacia abajo a distintas velocidades para dar sensación de que la nave avanza.**	2
5. **En la interfaz se mostrará las vidas, puntuación y los niveles de cadena y bonus. Además, si se ejecuta en un dispositivo móvil aparecerán el joystick y el botón de disparo.**	3
6. **Se implementará el sistema de puntuación tal y como se detalla en la definición del proyecto.**	8
7. **Se implementará el sistema de vidas.**	3
8. **Se implementará el sistema de creación de oleadas de la forma descrita. Este será utilizado en el modo arcade.**	8
9. **Se creará un sistema de logros que desbloquearan elementos del juego (modificadores, armas, enemigos).**	20

10. Se implementarán todas las variantes de enemigos, modificadores y armas.	30
11. Se implementará un modo aventura que constará en un principio de tres zonas y en cada una habrá 3 fases. En la última fase de cada zona habrá un Boss. Cada fase tendrá 8 oleadas que serán diseñadas a mano. Para avanzar a la siguiente fase habrá que cumplir algunos requisitos en la anterior, no solo completar las oleadas.	90
12. Se implementará una música de fondo y los sonidos al disparar, destruir un enemigo y morir.	6

Tabla 8. Listado de requisitos

2.3 Tareas e iteraciones

Una vez que tenemos la lista de requisitos y funcionalidad comenzaremos a planificar las iteraciones. Para ello desglosaremos cada requisito en pequeñas tareas sencillas e iremos tomando aquellas que más prioridad tenga para el estado del proyecto, además de mirar el tiempo que estimamos que nos llevará para que cubra el rango de dos o tres semanas que nos marca cada iteración. Este tiempo ha sido estimado en base a la complejidad que puede tener cada tarea. Evidentemente es posible que una tarea que a priori parecía más compleja se resuelva antes de lo esperado y viceversa. Es por esto que al final de cada sprint mediremos el tiempo que se ha tardado y se comparará con el estimado para ver qué tan buena fue la estimación.



TAREAS	TIEMPO ESTIMADO (HORAS)
1. Adecuación del espacio de trabajo	
1.1 Creación del sistema de directorios	0.5
1.2 Configuración pixel Perfect (cámara y PPU)	1.5
1.3 Efecto Parallax	
1.3.1 Diseño	1
1.3.2 Implementación	2
2. Control del jugador	
2.1 Avatar	
2.1.1 Diseño gráfico	1
2.1.2 Implementación	3
2.2 Movimiento	
2.2.1 Diseño	1
2.2.2 Implementación	2
2.2.3 Pruebas y ajustes	0.5
2.3 Disparo	
2.3.1 Diseño	1
2.3.2 Implementación	2
2.3.3 Pruebas y ajustes	0.5
3. Enemigos	
3.1 Diseño gráfico	
3.1.1 Diseño enemigos básicos (12)	6
3.1.2 Diseño de variantes (13)	6.5
3.1.3 Diseño de jefes (3)	3

3.2 comportamiento enemigos básicos	
3.2.1 Sistema de movimiento horda	
3.2.1.1 Diseño	2
3.2.1.2 Implementación	5
3.2.1.3 Pruebas y ajustes	0.5
3.2.2 Sistema de ataque	
3.2.2.1 Diseño	1.5
3.2.2.2 Implementación	3
3.2.2.3 Pruebas y ajustes	0.5
3.2.3 Sistema de muerte	
3.2.3.1 Diseño	1
3.2.3.2 Implementación	2
3.2.3.3 Pruebas y ajustes	0.5
3.3 Comportamiento de variantes x 13	
3.3.1 Diseño	6
3.3.2 Implementación	16
3.3.3 Pruebas y ajustes	4
3.4 Comportamiento de jefes x3	
3.4.1 Diseño	6
3.4.2 Implementación	12
3.4.3 Pruebas y ajustes	1.5
4. Sistemas del juego	
4.1 Sistema de vidas	
4.1.1 Diseño	0.5
4.1.2 Implementación	1.5
4.1.3 Pruebas	0.5
4.2 Sistema de puntuación	
4.2.1 Diseño	2
4.2.2 Implementación	2
4.2.3 Pruebas	1
4.3 Sistema de oleadas	
4.3.1 Diseño	2
4.3.2 Implementación	2

4.3.3 Pruebas	1
4.4 Sistema de logros	
4.4.1 Diseño	**3**
4.4.2 Implementación	**15**
4.4.3 Pruebas	**2**
5.Interfaz	
5.1 Interfaz de juego	
5.1.1 Diseño	1
5.1.2 Implementación	2
5.2 Menú principal	
5.2.1 Diseño	1
5.2.2 Implementación	2
5.3 menú de pausa	
5.3.1 Diseño	1
5.3.2 Implementación	2
5.4 menú modo aventura	
5.4.1 Diseño	2
5.4.2 Implementación	4
5.5 Pruebas	2
6.Efectos	
6.1 Diseño (gráfico y software)	2
6.2 Implementación	1
6.3 Pruebas	0.5
7. Sonido	
7.1 Composición	3
7.2 Implementación	2
7.3 Pruebas	1
8.Modificadores x3	
8.1 Diseño (gráfico y software)	**6**
8.2 Implementación	**6**
8.3 Pruebas	**1**
9. Armas x3	
9.1 Diseño	**6**

9.2 Implementación	**6**
9.3 Pruebas	**1**
10. Modo aventura	
10.1 Diseño de fases (x9)	
10.1.1 Diseño de oleadas	
10.1.1.1 Disposición de enemigos	**45**
10.1.1.2 Mecánica de oleadas	**1**
10.1.2 Diseño de objetivos de fase	**5**
10.2 Implementación de fases	
10.2.1 Implementación de oleadas	**36**
10.2.2 Implementación de objetivos	**3**
	Total: 286

Tabla 9. Tareas

A continuación, veremos que tareas se asignaran a cada iteración.

Iteraciones	Tareas	Duración estimada
1	1, 2, 3.1.1, 3.2, 4.1, 4.2*, 5.1*, 6*, 7*	44
2	4.2, 4.3, 5.2, 5.3, 3.3, 6, 7	45.5

Tabla 10. Iteraciones

**Estas tareas no podrán ser terminadas al completo en esta iteración ya que faltarían elementos que se desarrollaran más adelante, pero se hará una pequeña aproximación para ir testeando.*

2.4 Presupuesto

Para estimar un presupuesto debemos tener en cuenta los recursos tanto humanos como técnicos que vamos a necesitar.

2.4.1 Recursos humanos

Para un proyecto real nos haría falta como mínimo el siguiente personal:

- 1 diseñador de videojuegos
- 1 compositor o ingeniero de sonido
- 1 diseñador grafico
- 1 programador de videojuegos con experiencia en Unity
- 1 gestor de proyectos

Como para este trabajo todas las tareas las realiza una misma persona haremos el presupuesto en base a esto. Según la web de “Jobted” un programador de videojuegos en España que tenga de 0 a 3 años de experiencia puede esperar un sueldo mensual de unos 1160€. Como trabajaremos sobre 286 horas a 5 días laborables de 8 horas necesitaremos abarcar 2 meses por tanto el sueldo total asciende a 2320€.

2.4.2 Recursos

El utilizado en este caso tiene un coste de 1300€. Si asumimos una amortización del PC de 5 años obtenemos un coste de 21.66€ al mes. El alquiler de un piso en Jaén ronda los 450€ al mes. Esto sumado al gasto de luz, agua e internet nos da un total de 1151.33€ en gastos de infraestructura durante dos meses.

servicios en 2 meses		
Concepto	**Precio (mes)**	**Coste (2 meses)**
PC	21.66 €	43.33 €
Alquiler	450 €	900 €
Luz	35 €	70 €
Agua	15 €	30 €
Internet	54 €	108 €
	Total:	**1151.33 €**

Tabla 11. Desglose de costes de servicios

2.4.3 Licencias Software

En este apartado no se tendrá en cuenta otro software que en un proyecto real se requerirían como los programas de edición de sonido, solo los que han sido utilizados realmente.

Gastos en software		
Concepto	**Motivo**	**Coste (2 meses)**
Unity 3d	Desarrollo del videojuego	----- €
Microsoft office	Documentación	21€
Piskel	Diseño de los PixelArt	------ €
Proyecto Libre	Gestión y planificación del proyecto	------ €
	Total:	**21 €**

Tabla 12. Costes en software

Teniendo todo esto en cuenta el presupuesto queda así:

Presupuesto	
Recursos humanos	2320 €
Servicios	1551.33€
Licencias software	21 €
Total:	**3892.33 €**

Tabla 13. Presupuesto

3 DESARROLLO

Una vez que ya tenemos clara la planificación del proyecto, procederemos a ir desarrollando el videojuego iteración por iteración. Por cada una de ella iremos especificando cómo se ha implementado y desarrollado cada una de las tareas además de añadir los diagramas que se han realizado para dicha iteración.

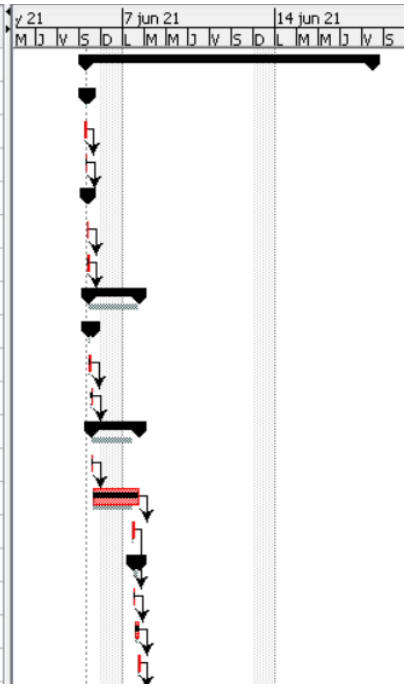
3.1 Primera iteración

Esta iteración está compuesta por las tareas especificadas en la *Tabla 10. Iteraciones.* Como vemos contiene tareas de 7 grupos distintos por lo que, aunque muchas tareas de un grupo guarden relación con las de otros, (por ejemplo, el sonido de disparo está relacionado con el desarrollo del disparo en sí) vamos a tratarlas por separado.

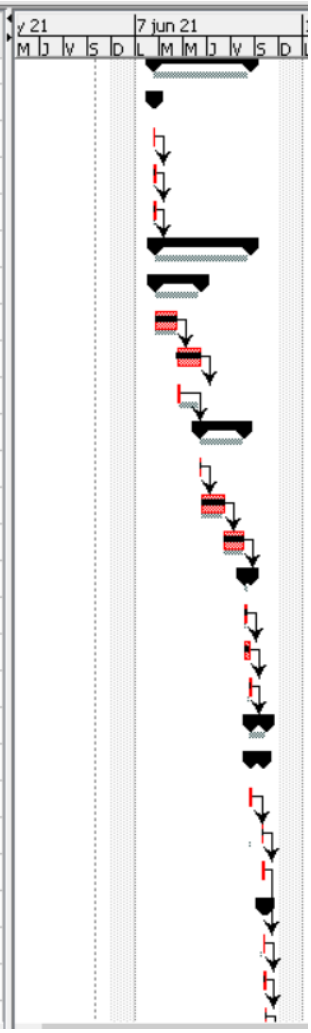
Esta documentación ha sido escrita tras finalizar la iteración así que no incluiremos las fases de pruebas de cada sección, sino que hablaremos de todas estas pruebas más adelante. Simplemente mostraremos como queda cada script y cada elemento después de comprobar que funcionan como se esperaba. Tampoco haremos referencia a la sección de efectos ya que cada efecto se incluirá como parte de del desarrollo de la función que los requiere.

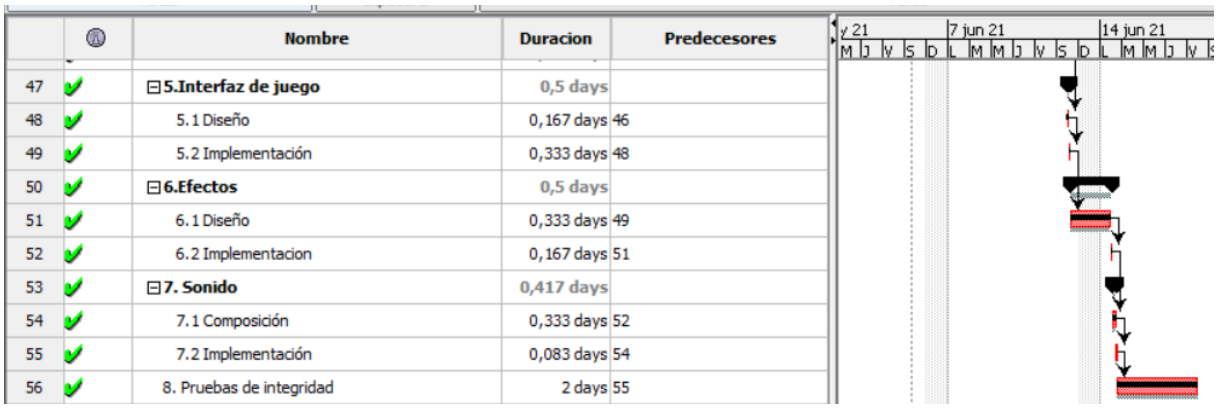
A continuación, mostraremos como queda el diagrama de Gantt, el cual ha sido creado a través de la aplicación Proyecto Libre.

	Nombre	Duracion	Predecesores
1	**Primera Iteración**	8,917 days	
2	**1. Adecuación del espacio de trabajo**	0,625 days	
3	1.1 Creación del sistema de directorios	0,042 days	
4	1.2 Configuración pixelPerfect	0,25 days	3
5	**1.3 Efecto Parallax**	0,333 days	**4**
6	1.3.1 Diseño	0,167 days	
7	1.3.2 Implementación	0,167 days	6
8	**2. Control del jugador**	1,5 days	**7**
9	**2.1 Avatar**	0,667 days	
10	2.1.1 Diseño	0,167 days	
11	2.1.2 Implementación	0,5 days	10
12	**2.2 Movimiento**	1 day	**11**
13	2.2.1 Diseño	0,333 days	
14	2.2.2 Implementación	0,833 days	13
15	2.2.3 Pruebas y ajustes	0,083 days	14
16	**2.3 Disparo**	0,583 days	
17	2.3.1 Diseño	0,167 days	15
18	2.3.2 Implementación	0,333 days	17
19	2.3.3 Pruebas y ajustes	0,083 days	18



	Nombre	Duracion	Predecesores
20	**3. Enemigos**	2,625 days	**19**
21	**3.1 Diseño**	0,167 days	
22	3.1.1 Calamar blanco	0,083 days	
23	3.1.2 Cangrejo blanco	0,083 days	22
24	3.1.3 Pulpo blanco	0,083 days	23
25	**3.2 implementación de comportamiento**	2,5 days	**24**
26	**3.2.1 Sistema de movimiento**	0,833 days	
27	3.2.1.1 Diseño	0,333 days	
28	3.2.1.2 implementacion	0,667 days	27
29	3.2.1.3 Pruebas y ajustes	0,083 days	28
30	**3.2.2 Sistema de ataque**	1,333 days	**29**
31	3.2.2.1 Diseño	0,25 days	
32	3.2.2.2 Implementación	0,5 days	31
33	3.2.2.3 Pruebas y ajustes	0,833 days	32
34	**3.2.3 Sistema de daño**	0,583 days	**33**
35	3.2.3.1 Diseño	0,167 days	
36	3.2.3.2 Implementación	0,333 days	35
37	3.2.3.3 Pruebas y ajustes	0,083 days	36
38	**4. Sistemas del juego**	0,75 days	**37**
39	**4.1 Sistema de vidas**	0,417 days	
40	4.1.1 Diseño	0,083 days	
41	4.1.2Implementación	0,25 days	40
42	4.1.2 Pruebas	0,083 days	41
43	**4.2 Sistema de puntuación Básico**	0,333 days	
44	4.2.1 Diseño	0,083 days	42
45	4.2.2 Implementación	0,167 days	44
46	4.2.3 Pruebas	0,083 days	45





		Nombre	Duracion	Predecesores
47	✓	⊟5.Interfaz de juego	0,5 days	
48	✓	5.1 Diseño	0,167 days	46
49	✓	5.2 Implementación	0,333 days	48
50	✓	⊟6.Efectos	0,5 days	
51	✓	6.1 Diseño	0,333 days	49
52	✓	6.2 Implementacion	0,167 days	51
53	✓	⊟7. Sonido	0,417 days	
54	✓	7.1 Composición	0,333 days	52
55	✓	7.2 Implementación	0,083 days	54
56	✓	8. Pruebas de integridad	2 days	55

Ilustración 9. Diagrama de Gantt iteración 1

3.1.1 Configuración inicial

3.1.1.1 Adecuación del espacio de trabajo

En este apartado se trata de preparar el espacio de trabajo en Unity para un correcto desarrollo y funcionamiento del proyecto. Para empezar, creamos un árbol de directorios básico en los que irán repartidos cada uno de los componentes de nuestro juego.

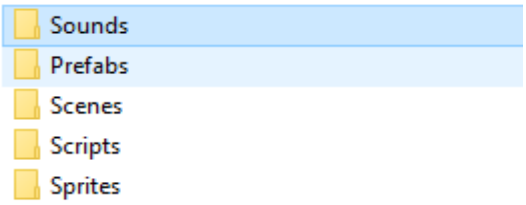


Ilustración 10. Árbol de directorios

En Unity cada proyecto está dividido en escenas. Esto supone un ahorro en memoria ya que solo es necesario cargar el contenido de una escena. Nuestro proyecto constara de las siguientes escenas:

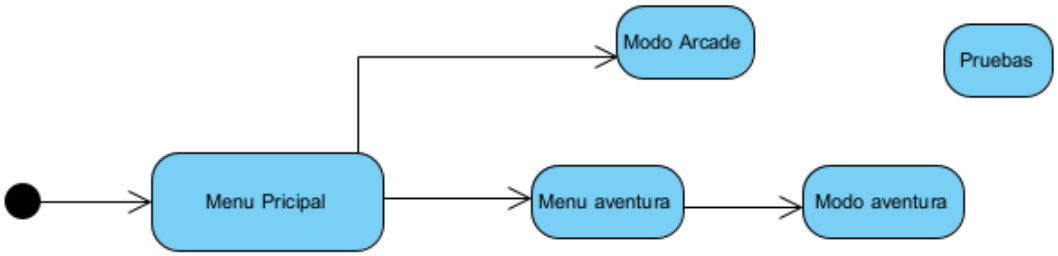


Ilustración 11. Escenas del proyecto

En realidad, cada nivel modo aventura deberá ser una escena diferente ya que tendrán una configuración de oleadas y objetivos distintos y predefinidos, pero para simplificar, se han resumido en el diagrama como modo aventura. En la primera iteración solo crearemos la escena Pruebas debido a que en ella solo desarrollaremos y probaremos las mecánicas más básicas de nuestro juego que son comunes a ambos modos de juego. Dejaremos también la del Menú principal para más adelante.

3.1.1.2 Pixel Perfect

Una vez creada nuestra escena “Pruebas” procederemos a configurar la cámara principal para que sea Pixel Perfect. Dado que nuestro juego estará basado en su mayoría por sprites pixelados, necesitamos que estos se vean bien en cualquier posición en la que se encuentren. Con la configuración por defecto de Unity, este requisito no se cumple y podríamos observar cómo según la posición y la resolución en la que se encuentre un sprite, este se deforma.

Ilustración 12. Sprite deformado

Ilustración 13. Sprite Pixel Perfect

Para conseguir este resultado haremos una serie de cálculos y modificaremos una serie de parámetros en la configuración de Unity. Lo primero que necesitaremos es fijarnos una resolución objetivo. Como este juego está pensado para jugarse en dispositivos móviles la resolución que tomaremos será de 18.5:9 (2960x1140 pixeles). Aunque esta sea la resolución objetivo, no quiere decir que no funcione en otras, ya que se escalará automáticamente.

Ahora dividiremos nuestra pantalla en una cuadrícula (móvil en posición horizontal) cuyo alto queremos que sea 16, por tanto, su ancho será:

$$16*9 / 18.5 = 32.888$$

Cada uno de estos cuadraditos son una unidad de distancia de Unity. Para conseguir esto solo tenemos que ajustar el tamaño de la cámara a la mitad del alto que queremos con una proyección ortográfica. En nuestro caso:

$$16 / 2 = 8$$

Otros ajustes necesarios son desactivar el “Anisotropic Textures” y el “Anti Aliasing”. A continuación, hay que tener en cuenta todas las configuraciones que han de tener los sprites que se vayan a utilizar. El parámetro “Filter mode” debe ajustarse como “Point (no filter)” y la Compresión se configurará como “None”.

Además, El “Pixels Per Unit” (PPU) debe cambiarse en cada sprite por el valor calculado con la siguiente formula.

$$PPU * PPUScale = ((Vertical\ Resolution) /\ (Orth.\ Size)) * 0.5$$

La escala siempre será de 1 y el tamaño ortogonal lo habíamos fijado en 8 por lo que:

$$PPU = (1440 / 8) * 0.5 = 90$$

Esto nos indica que el PPU de cada Sprite será de 90. Por último, en “Edit>Grid and snap settigs” cambiamos el “increment snap move” a 0.5 en todos los ejes. Esto hará que podamos mover a nuestros aliens fácilmente de 0.5 en 0.5 manteniendo pulsado el CTRL.

3.1.1.3 Efecto Parallax

Este efecto consiste en hacer que una o varias imágenes de fondo se muevan de forma indefinida en una dirección para dar sensación de movimiento.

3.1.1.3.1 Diseño

Para lograr este efecto dispondremos de una serie de 6 capas de sprites superpuestas, Están son fondo negro, estrellas lejanas, estrellas a media distancia,

(esta capa incluirá también un planeta ya que este no puede ser una capa como las demás porque su tamaño es más pequeño), nébula, estrellas grandes y meteoros. Cada una de estas se moverá verticalmente hacia abajo a una velocidad distinta para dar sensación de profundidad y movimiento. Las imágenes han sido tomadas de forma gratuita de la asset Store de Unity (“MK - Space Background Parallax Maker”). Habrá dos repeticiones de esta composición una debajo de la otra. Cada tipo de capa será el padre de la misma capa que se encuentra por encima. Es decir, la capa de arriba se mueve únicamente porque se mueven sus padres.

El funcionamiento es sencillo: cuando alguna capa padre (y que por tanto ya está completamente fuera de cámara) llegue a una altura determinada volverá a colocarse (arrastrando a la capa hija) hacia arriba de tal forma que el padre ocupará exactamente la misma posición en la que estaba su hijo en el instante anterior.

3.1.1.3.2 Implementación

Para empezar, colocaremos cada imagen formando la composición descrita en el diseño obteniendo el siguiente resultado.

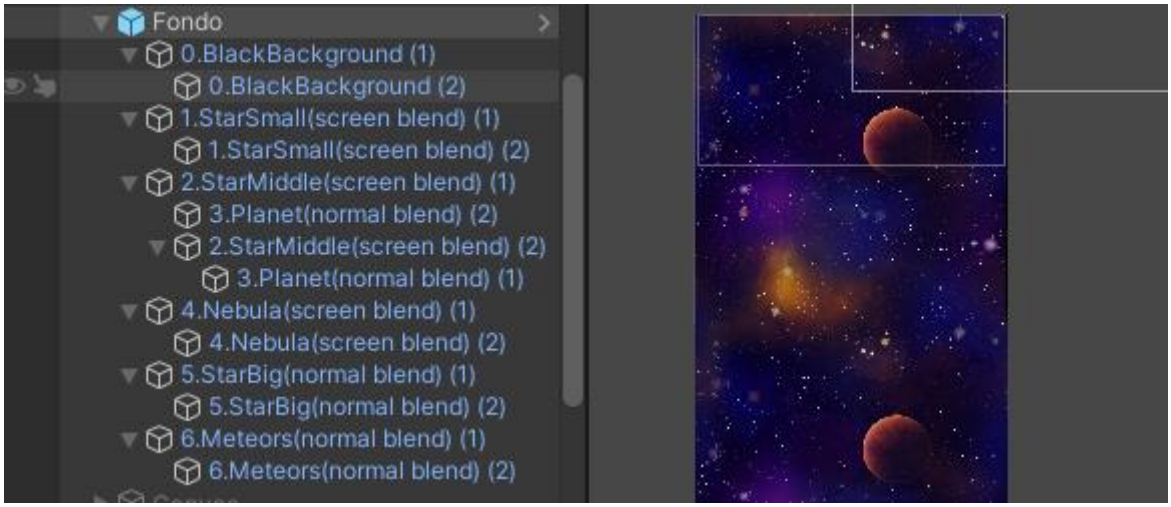


Ilustración 14. Composición efecto Parallax

En la asset store, estos componentes incluían script ya definidos pero eran más complejos de lo que necesitamos, así que creamos un script más sencillo para este fin (“Parallax Movement”) que será asignado únicamente a cada capa padre.

En el script, se define una velocidad de forma pública para poder asignarle una distinta a cada capa desde el editor y un offset que será igual para todos. Esta variable representa la distancia entre en centro de una repetición de capas, con respecto a la otra, o lo que es lo mismo, cuanto mide de alto cada capa.

Para determinarlo se deshizo momentáneamente la relación padre e hijo de una de las capas y se restó la posición de uno frente a la del otro (era necesario separarlos porque si no se estaría restando la posición relativa del objeto con respecto al otro y no sus posiciones globales). También tomaremos la posición inicial del objeto.

En cada “frame” se calculará la distancia hacia arriba que moveremos la capa con respecto a la posición inicial. Este valor al que llamaremos “NowPos”, estará entre 0 y el offset, ya que se obtiene mediante el módulo de un número variable y el offset (es decir el resto de la división). Este número variará con el tiempo ya que se calcula como tiempo * - velocidad. El resultado de todo esto es el siguiente:

```
NowPos = Mathf.Repeat(Time.time * -ScrollSpeed, ScrollOfset);
        transform.position = StartPos + Vector3.up * NowPos;
```

El movimiento hacia abajo se consigue de esta forma sumando cada vez un valor menor a la posición inicial (que se calcula como “ Vertor3.up * NowPos”) y cuando la variable “NowPos” se acerque a cero será porque “tiempo * velocidad” están cerca del offset y por tanto las capas habrán llegado al tope inferior. Cuando “tiempo * velocidad” superen por poco el valor del offset, la variable “NowPos” volverá a ser cercana al offset y por tanto toda la capa se desplazará hasta arriba.

3.1.2 Control Del Jugador

El desarrollo del control del jugador se divide en tres grupos de tareas

3.1.2.1 Avatar

3.1.2.1.1 Diseño

Para el diseño del avatar del jugador se ha optado por un asset gratuito de la tienda de unity (Star Sparrow Modular Spaceship).

3.1.2.1.2 Implementación

Creamos un objeto vacío al que llamaremos nave y como hijo de este, (Sinicki, 2017) iremos colocando cada uno de los componentes que forman la nave hasta obtener el resultado que vemos en la imagen del diseño. Colocaremos nuestra nave en el centro y pegada a la parte inferior de la zona que se capta en cámara. El resultado en la jerarquía queda así:

Ilustración 16. Componentes de la nave

3.1.2.2 Movimiento

3.1.2.2.1 Diseño

Para el movimiento tenemos que tener en cuenta en que plataforma estamos. En móviles utilizaremos un joystick mientras que en pc se usaran las flechas o ‘a’ para izquierda y ‘d’ para derecha. El movimiento estará restringido por la izquierda y por la derecha para no salirse de la pantalla.

3.1.2.2.2 Implementación

Se creará un nuevo script al que llamaremos “PlayerController”. Este se le asignara a un nuevo objeto vacío al que denominaremos Player y que colocaremos en la misma posición que la nave. Posteriormente haremos a la nave “hija” del Player. A nuestro Player lo dotaremos de un componente Rigidbody2D. En el script definiremos una variable pública para la velocidad y otra en la que guardaremos la referencia al Rigidbody2D como “player”. Usaremos el método de Unity “FixedUpdate” que es el que se encarga de los cálculos de las físicas, y haciendo uso de la librería “CrossPlatformInputManager” definiremos la velocidad del RigidBody2D como:

```
player.velocity = new Vector2 (CrossPlatformInputManager.GetAxis("Horizontal"),
0.0f) * velocidad * Time.deltaTime;
```

El “getAxis” nos devuelve un valor entre -1 y 1 dependiendo de la entrada que esté recibiendo. Dicho valor determinará la dirección. Esto, multiplicado por la velocidad y por el “deltaTime” (tiempo trascurrido desde el último fotograma) nos da como resultado el movimiento del jugador. Tras hacer unas pruebas, la velocidad se establece en 450 (Murray, 2014).

Ahora crearemos un nuevo script con una clase estática (que no heredará de “MonoBehaviour”) al que denominaremos “MisFunciones”. Esta clase no se asigna a ningún objeto porque es estática y no puede ser instanciada. En ella definiremos distintas funciones de utilidad. En este caso comenzaremos con la función que determina cual es el borde de la pantalla. Esta función nos transforma coordenadas de pantalla en coordenadas de mundo.

```
public static float CordenadaXpantalla()  {
return Camera.main.ScreenToWorldPoint(new Vector3(Screen.width,     0)).x;
}
```

En nuestro “PlayerControler” definiremos dos variables “bordePantalla” y “LimiteX”. “bordePantalla” tendrá el valor que devuelve nuestra función “CordenadaXPantalla” y “limiteX” será igual a “bordePantalla” – 1. Le restamos uno por que nuestra nave tiene 2 unidades de ancho y si nos quedamos con el valor sin la resta, la mitad de la nave desaparecerá al acercarse a los bordes y lo que nosotros queremos que la nave se “choque” con el límite de la pantalla y no pueda avanzar más en esa dirección. Con este fin, en el método “Update” haremos que se compruebe la posición de la nave y si su coordenada X es menor al límite inferior(“-limiteX”) o mayor al límite superior (“limiteX”), la posición permanecerá constante en ese límite.

Para el control de dispositivos móviles utilizaremos los componentes de “CrossPlatformInput” contenidos en el paquete gratuito “StandardAssets” de Unity. Crearemos un “Canvas” al que le colocaremos el script “mobileControlRig” que hace entre otras cosas que solo sea visible cuando se ejecuta en un dispositivo móvil. Formaremos un joystick con los dos sprites que nos vienen para este fin (parte fija y parte móvil) y lo colocaremos en la esquina inferior izquierda como hijo del “Canvas”. En el sprite de la parte móvil colocaremos el script “Joystick” que convertirá el movimiento de esta, en un valor entre -1 y 1 que pasará al “CrossPlatformInputManager” para que pueda ser usado por el “PlayerController”. El “CrossPlatformInput” engloba bajo el mismo evento dos o más acciones diferentes, en este caso la entrada de movimiento en horizontal del joystick, flechas o las teclas “a” y “d”. (Sinicki, 2017) En el editor de Unity configuramos este script para que solo permita moverse en horizontal y su rango de movimiento sea 25.

Ilustración** 17**. Control Para Móviles

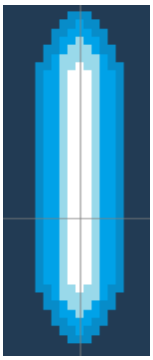


Ilustración 18. Proyectil del Jugador

3.1.2.3 Disparo

3.1.2.3.1 Diseño

El Disparo del jugador se efectuará con un clic del ratón en PC o pulsando un botón en la esquina inferior derecha en dispositivos móviles. Ambas entradas tendrán como nombre “Fire1” en el “CrossPlatformInputManager”. Por defecto solo puede haber un disparo del jugador en pantalla y no se podrá disparar hasta que este sea destruido (impactando en un enemigo o proyectil o bien al salir de la pantalla por la parte superior). Aunque el proyectil sea destruido muy pronto, de todas formas, habrá un tiempo de recarga mínimo que tendrá que pasar entre disparo y disparo. Para el diseño del proyectil se ha recurrido a una “SpriteSheet” descargada gratuitamente del portar “OpenGameArt.org” (Bullet Collection 1 (M484))

3.1.2.3.2 Implementación

Para implementar el disparo deberemos:

- Por un lado, tener un script que detecte cuando se pulsa el botón adecuado para instanciar un disparo y se encarga de comprobar si se cumplen las condiciones para ello.
- Por otro lado, necesitamos programar el comportamiento de nuestro proyectil. Es decir, que se mueva hacia arriba en línea recta. (Posteriormente cuando implementemos a los enemigos volveremos aquí para hacer que se detecte el impacto).
- Además, deberemos hacer un sistema mediante el cual los proyectiles sean eliminados cuando salga de pantalla y así liberar memoria.

Para empezar, crearemos nuestro “prefab” del proyectil del jugador. Arrastramos el sprite de nuestro proyectil a la escena. Le añadiremos un “Rigidbody2D” cuyo parámetro “body Tipe” configuraremos como “Kinematic” en el editor. Esto hará que no se vea afectado por la gravedad u otras fuerzas. Agregaremos también un “Capsule Collider 2D” que ajustaremos para que abarque a nuestro “Sprite” y que marcaremos como “IsTrigger”. Esto quiere decir que este objeto podrá ser atravesado por otros cuerpos y se podrá detectar cuando otro objeto entra en contacto con este.

Posteriormente, crearemos otro Script llamado “BulletBehaviour” que también añadiremos a este objeto. En dicho script, moveremos indefinidamente el proyectil hacia arriba de forma parecida a como lo hacíamos con la nave salvo que ahora el valor distinto de 0 que se multiplica a la velocidad y el tiempo está en el eje “Y” siendo 1 constantemente.

En otro script que denominaremos “DisparoJugador” y que asociaremos a nuestro “Player”, tendremos una referencia al “prefab” del proyectil para poder instanciarlo cuando se pulse el botón asociado al evento “Fire1”. Crearemos un objeto vacío que colocaremos en la parte superior de la nave y lo haremos hijo de “Player”. Con este objeto determinaremos en qué posición debe aparecer nuestro proyectil. Antes de disparar se comprueba si se puede disparar y si quedan balas en el cargador. En el momento en que se instancie un proyectil se restara de la variable “cargador”. (que se inicializa como máximo en 1). De esta forma podemos controlar cuantos proyectiles tiene el jugador en pantalla por si más adelante se añade un modificador que lo modifique. La variable booleana “PuedeDisparar” se pone en falso cuando se dispara y volverá a ser verdadera cuando pase el tiempo de recarga establecido. En este caso se fijó en 0.6 segundos (Murray, 2014).

Crearemos también un método llamado “Recargar” que devolverá una bala al “cargador” si este no está al máximo ya. Este método será llamado a través de una referencia al “Player”, mediante paso de mensajes desde el script de la bala en el momento en que colisione con otro objeto (Alien, proyectil enemigo o “borradorBalas”). Para detectar colisiones usamos el método “OnTriggerEnter2D (Collider2D other)”. Inmediatamente después se autodestruirá.

```
    jugador.SendMessage("Recargar");
     Destroy(this.gameObject);
```

Para que los proyectiles se destruyan cuando salen de la pantalla colocaremos allí un objeto vacío con un “BoxCollider2D” que abarque como mínimo todo el ancho de la pantalla, pero este situado fuera de la misma por arriba.

Para accionar el disparo en dispositivos móviles utilizaremos un botón en la esquina inferior derecha. Este botón usa el script “buttonHandler” que hace de puente entre dos entradas, en este caso enlaza el pulsar el botón con la entrada llamada “Fire1” que es la que dispara. En caso de PC dicha entrada está ya enlazada por defecto con el clic izquierdo del ratón por eso no hace falta hacer nada más.
Como en el caso del joystick, este botón se colocará como hijo del “canvas especial”, que solo se muestra si se ejecuta en un dispositivo móvil (Sinicki, 2017).

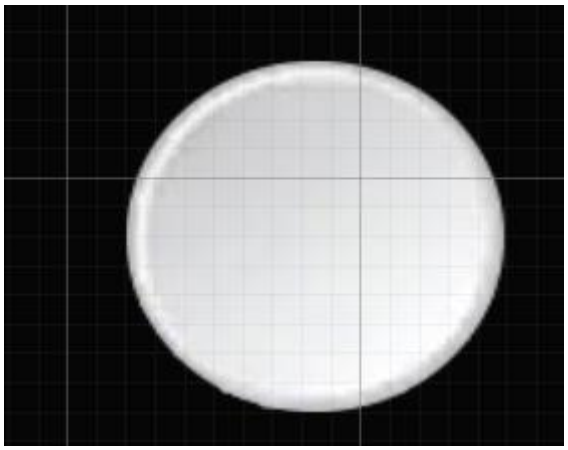


Ilustración 19. Botón de disparo

3.1.3 Enemigos

En esta iteración nos centraremos en el diseño e implementación de los enemigos básicos, es decir aquellos que no tienen ninguna característica única.

3.1.3.1 Diseño gráfico

El diseño gráfico ha sido realizado con la herramienta Piskel, que es una herramienta gratuita para diseñar y dibujar ilustraciones pixelArt. Se ha tomado como referencia los diseños originales y se han recreado con la misma forma, pero coloreados por una paleta de colores compuesta por 4 colores distintos. Además, se han creado variantes de distintos colores ya que el en original solo eran blancos. Cada enemigo está compuesto por dos imágenes que se irán alternado para conformar la animación.

Calamar		
	Versión normal de color rojo	Puntuación:150
	Versión normal de color azul	Puntuación:150
	Versión normal de color verde	Puntuación:150
	Versión normal de color blanco	Puntuación:150

Tabla 14. Diseños básicos del Calamar

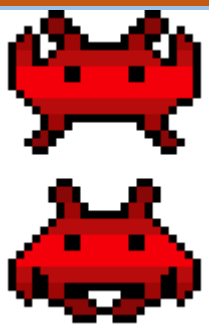
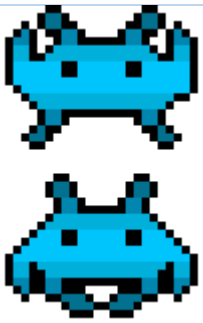
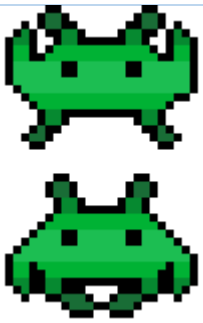
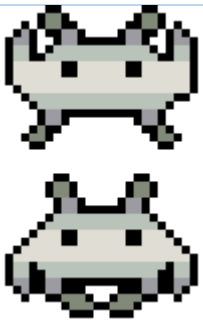
Cangrejo		
	Versión normal de color rojo	Puntuación:100
	Versión normal de color azul	Puntuación:100
	Versión normal de color verde	Puntuación:100
	Versión normal de color blanco	Puntuación:100

Tabla 15. Diseños básicos del cangrejo

Pulpo		
	Versión normal de color rojo	Puntuación:70
	Versión normal de color azul	Puntuación:70
	Versión normal de color verde	Puntuación:70
	Versión normal de color blanco	Puntuación:70

Tabla 16. Diseños básicos del pulpo

3.1.3.2 Implementación

Cabe destacar que se creará un único prefab y una vez que tengamos todas las funcionalidades añadidas, entonces lo copiaremos cambiando los sprites y variables púbicas que varíen de uno a otro.

Para crear el prefab del enemigo, primero seleccionamos un sprite, “Calamar blanco” por ejemplo, y lo configuramos como se dijo en la sección de “Pixel Perfect”. Como es una imagen que contiene más de un sprite, seleccionamos “Multiple” en el campo “Sprite mode” y luego pulsamos el botón “Sprite Editor”. Posteriormente en la ventana desplegada, pulsamos slice, configuramos como queremos que se corten los sprites y pulsamos el botón “Slice”. Una vez hecho esto, aplicamos y ya tenemos los sprites separados. (Este procedimiento se efectuará con cualquier “SpriteSheet” o imagen que contenga varios Sprite y por tanto se omitirá a partir de ahora).

Abrimos los sprites, cogemos el primero de ello y lo arrastramos hasta la escena. Posteriormente, lo llevamos a nuestra carpeta prefabs y se creara automáticamente un prefab simple que solo contiene, de momento, un “Sprite Renderer”.

3.1.3.3 Sistema de movimiento en horda

3.1.3.3.1 Diseño

Para que todos nuestros aliens se muevan al unísono, los colocaremos como hijos de un objeto vacío al que llamaremos “Horda”. Este se situará en el centro de la pantalla. Nuestra horda definirá un ritmo al que se moverá (este ritmo se irá incrementando con el tiempo) y lógicamente todos sus hijos aliens se moverán con él. Cada vez que los alien se muevan cambiaran de posición (Cambiaran su Sprite) generando una animación de movimiento individual. Además, cada alien comprobará si ha llegado al límite de la pantalla, en cuyo caso se cambiará la dirección del movimiento de la Horda. Cuando la Horda recibe este cambio de dirección además se moverá un paso hacia abajo si ningún alien ha alcanzado la línea de movimiento del jugador. En este caso se restará una vida al jugador y se volverá a la posición inicial de la horda. (Solo la posición, los aliens que ya habían sido destruidos no vuelven).

3.1.3.3.2 Implementación

Para llevar a cabo esta funcionalidad necesitamos 2 scripts, uno asociado a cada enemigo individual y otro al objeto horda. El script “MovimientoHorda” definirá dos eventos uno que se lanzará con cada “golpe” del ritmo, es decir cada vez que haya un movimiento en horizontal y otro cada vez que haya un descenso o movimiento en vertical. Además, se declarará cuanta distancia supone dar un paso en horizontal y cuanto en vertical. También se guardará la posición inicial donde comienza y se llevará un control de la dirección de movimiento actual y la que llevaba anteriormente. Con estas dos últimas podemos concretar cuándo se ha realizado un cambio de dirección. También tendremos una variable booleana para determinar cuándo se ha producido una “Invasión”, es decir cuando los alien han alcanzado el área por la que se mueve el jugador. Al comienzo del juego la dirección inicial se determina aleatoriamente con un 50% de probabilidades. (Para las probabilidades se ha creado una función en el script “mis funciones” que te devuelve verdadero o falso según una probabilidad que le pases como parámetro de 0 al 100). El funcionamiento durante el juego será el siguiente:

En cada frame se irá acumulando el tiempo que ha pasado desde el fotograma anterior por medio del “deltaTime” y si este alcanza al tiempo que marca el ritmo actualmente se lanzará el evento del ritmo, se comprueba si la dirección ha cambiado. Si la respuesta es sí, se desciende un paso en vertical lanzando el evento descender y se cambia la dirección anterior por la nueva para que no vuelva a descender. Si la dirección no había cambiado se mueve un paso horizontal en la dirección actual. Si se ha descendido, se reduce en 0.2 el ritmo de juego (se moverán con más frecuencia) siempre y cuando esta disminución de ritmo no sea inferior al límite inferior que marcaremos en 0.2. (en realidad se ha puesto 0.18 porque a veces las restas con decimales no son exactas del todo). Este límite es necesario ya que con un ritmo inferior a este nuestros marcianos irían demasiado rápido y sería casi imposible destruirlos.

Otra acción que realizaremos antes de descender será comprobar si se va a producir una invasión (lo veremos con más detalle en el script de “Enemigo”) en cuyo caso, restaremos una vida al jugador enviándole un mensaje a través de una referencia al él, y reposicionaremos la horda en su posición original.

El script “enemigo” contiene como variables un vector de Sprites que contendrá en cada tipo de alien los dos sprites que conforman su animación, una referencia a su “spriteRenderer” y una referencia a la “Horda”. Utilizaremos los métodos “OnEnable” y “OnDisable” para subscribirnos y desubscribirnos a los eventos que hemos definido en la “Horda”.

```
MovimientoHorda.OnRiytmBeat += AnimarSprite;
 MovimientoHorda.descenso += ComprobarAltura;
MovimientoHorda.OnRiytmBeat += CambiarDireccion;
```

Para desubscribirse se sustituye el ‘+’ por un ‘-’. Estas tres líneas lo que hacen es que cada vez que se lanza un evento se ejecutará un método asociado a él. En este caso cada vez que hay un movimiento de la horda (golpe del ritmo), se ejecutan los métodos “AnimarSprite” y “CambiarDirección” y cuando La horda desciende se lanza “ComprobarAltura”.

“AnimarSprite” cambia el Sprite actual del alien por el siguiente en el vector de sprites. (en caso de no haber más volverá al primero). “CambiarDirección” Se encarga de cambiar la dirección del movimiento de la horda a través de su referencia, si la distancia al borde de la pantalla (función definida en “MisFunciones”) es menor que la distancia de paso horizontal de la horda, o lo que es lo mismo, si en caso de dar un paso más, el alien se sale de la pantalla. Claro está, también hay que tener en cuenta hacía que borde nos dirigimos ya que no nos interesa que se mida la distancia hacia el borde opuesto al el que nos dirigimos. Además, debemos tener en cuenta que se mide desde el centro del Sprite por tanto si no le restamos la mitad del ancho de este, medio Sprite se perderá por el borde antes de cambiar de dirección. Para ello usamos el siguiente método:

```
public float DistanciaAlBorde()
                {
                        float distancia;
                        float bordeX = movimientoHorda.Direccion.x;
        distancia = Mathf.Abs((bordeX * MisFunciones.CordenadaXpantalla()) -
        transform.position.x - bordeX * mySpriteRenderer.bounds.size.x / 2);
                        return distancia;
}
```

"Comprobar altura" pondrá el valor de "Invasión" de la Horda en verdadero si la posición en Y es menor que -5.5. Se toma este valor ya que en esta posición es la última altura en la que la nave puede destruir a un enemigo (justo pegado al morro de la nave).

3.1.3.4 Sistema de ataque

3.1.3.4.1 Diseño

Cada enemigo disparará pasado un tiempo para el próximo disparo, generado aleatoriamente, que será entre 1 y 4 segundos para los enemigos normales. Este tiempo cambia cada vez que dispara o en caso de no poder disparar. (por que ya se ha superado el máximo de proyectiles enemigos). Para empezar, el máximo de proyectiles enemigos será de 4. Este contador se llevará a cabo a través de una variable estática propia del proyectil la cual se suma cuando este se instancia y se restará cuando desaparezca. Una vez superado el tiempo de próximo disparo y no se haya alcanzado el máximo de proyectiles, se tendrá que comprobar si finalmente disparará, por medio de una probabilidad que comenzará como 30%. Los sprites del disparo han sido dibujados con la herramienta Piskel.

Ilustración 20. Sprites de disparo

3.1.3.4.2 Implementación

Para comenzar con esta tarea, como a partir de aquí vamos a tener detección de colisiones, vamos a configurar las capas en las que estarán cada elemento de nuestro juego y su interacción entre ellas. Además, dotaremos a cada elemento de una etiqueta para diferenciar con que se ha colisionado en cada momento. Para crear etiquetas seleccionamos cualquier elemento de la jerarquía por ejemplo, al jugador (al que le seleccionamos la etiqueta Player que viene por defecto) y en el inspector, en el desplegable llamado “Tags” pulsamos “Add tags”. Una vez abierta la pestaña “Tags and Layers”, en la sección tags añadimos 3 etiquetas por ahora:

- Bullet: se le asignará al proyectil del jugador.
- EnemyBullet: se colocará esta etiqueta a todo tipo de proyectil enemigo.
- -Enemigo: cada tipo de enemigo llevará esta etiqueta.

Ya que estamos en esta ventana, podemos aprovechar para crear nuestras capas o “Layers”. Abrimos la sección correspondiente y a partir de la numero 8 (ya que las demás están reservadas por Unity) creamos las siguientes:

- Jugador
- Enemigo
- ProyectilJugador
- ProyectilEnemigo

Asignamos a cada prefab su capa correspondiente. Creamos las mismas capas en la sección anterior “sorting Layers” y los colocamos en orden inverso a la lista anterior de tal forma que el jugador quede abajo, por encima Enemigo, etc. Esto se asignará en el Sprite Renderer de cada elemento y servirá para diferenciar que elementos tienen más prioridad de verse que otro. Por ejemplo, el jugador tiene prioridad máxima y se verá por delante de cualquier otro elemento al encontrarse ocupando el mismo espacio de la pantalla y si un proyectil enemigo atraviesa a un enemigo, pasará por detrás de este ya que tiene prioridad más baja.

Para finalizar con esto de las capas, vamos a configurar sus interacciones ya que, por ejemplo, un proyectil enemigo atraviesa a los enemigos sin impactar en ellos (no hay fuego amigo entre Aliens). Esto lo podemos hacer si nos dirigimos Edit>Proyect Settings>Physics 2D>Layers Collision Matrix. El resultado quedará así:

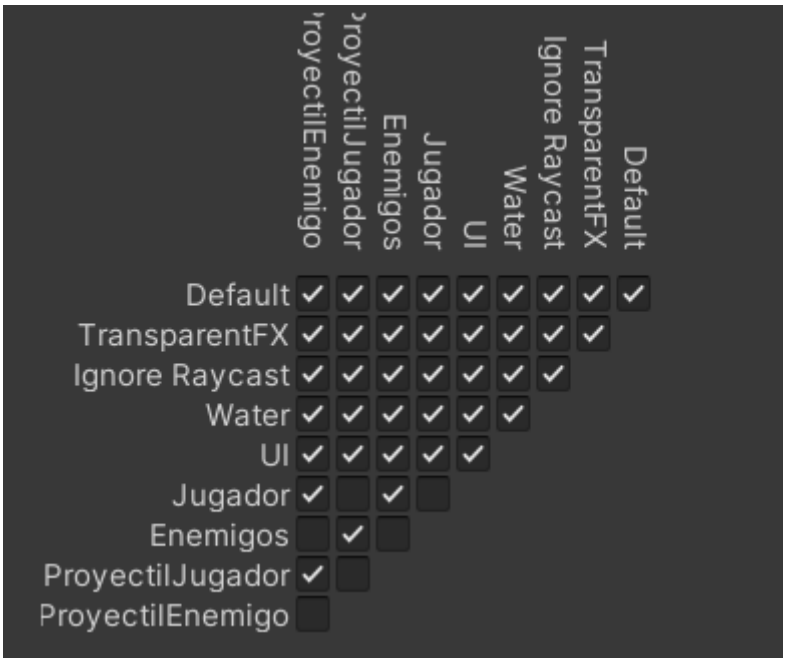


Ilustración 21. Matriz de colisiones ente capas

A continuación, crearemos el prefab para el proyectil enemigo. Para ello arrastramos el conjunto de sprites que hemos diseñado a la escena. Esto genera también la animación de forma automática. Después Lo arrastramos a la capeta prefabs para crearlo. Una vez creado el prefab podemos borrarlo de la escena. Seguidamente, abrimos el prefab para editarlo y le añadimos un “Box Collider 2D” y un “Rigidbody 2D” configurado como “kinematic”. Restaría crear y añadir los scripts que definen su comportamiento.

Como más adelante pretendemos crear más tipos de proyectiles, vamos a separar las funciones que tienen que tener todos los proyectiles en el script “EnemyBullet” que básicamente contiene una variable estática, es decir compartida con todas las instancias, y suman uno cuando “despiertan” y se restan al ser destruidos. De esta forma comprobaremos si se puede disparar en caso de no tener instanciados ya el máximo de proyectiles. Por otro lado, crearemos el comportamiento propio de un proyectil normal. Este será, descender progresivamente a una velocidad

constante y cuando colisiona con algo se autodestruye (recordemos que no cuenta como colisión ni los enemigos ni los demás proyectiles enemigos, pero si el proyectil del jugador el cual también se autodestruye cuando colisiona por lo que no hay que hacer nada al respecto), pero antes, si ese algo es el jugador (lo reconoceremos por la etiqueta “Player”), le envía un mensaje para que se destruya también. Es decir, perderá una vida. Programaremos eso más adelante, por el momento comentaríamos la orden de paso de mensaje si quisiéramos probar nuestro juego.

```
private void OnTriggerEnter2D(Collider2D other)
{
        if (other.gameObject.tag == "Player")
       {
            //quitar una vida al jugador
            other.gameObject.SendMessage("Destruir");
        }
        Destroy(this.gameObject);

}
```

Hecho esto copiaremos el “borrador de balas” que colocamos en la parte superior para los proyectiles del jugador y lo colocaremos del mismo modo en la zona inferior para que estos proyectiles también se eliminen al salir de la pantalla. (Realmente este, no necesitaría el script de borrar balas porque estos proyectiles se autodestruyen al chocar, pero tampoco pasa nada si se lo dejamos).

Una vez tenemos nuestros proyectiles funcionales procederemos a programar a los enemigos para que nos disparen siguiendo las indicaciones propuestas en el diseño. Para ello creamos dos scripts. Uno que llamaremos “ControladorAliens” que se encargara de ir actualizando una variable booleana “disparoPermitido” en función de si se ha superado el número máximo de disparos en pantalla o no. Le asignaremos este script a un nuevo objeto vacío al que llamaremos “GameManager” que gestionará diversas cuestiones relacionadas con el funcionamiento del juego en general. En el otro script, “Enemigo” tendremos una referencia a este controlador para consultar el valor de esa variable y comprobar así, si podría disparar cuando se pase el tiempo generado aleatoriamente entre 1 y 4 segundos. Si “supera” esta prueba tendrá que superar una más, la que determina la probabilidad de disparar. Esta se iniciará en 30%.

Si finalmente dispara, se instanciará un proyectil del “prefab” que hemos creado que se lo asignaremos a través de una variable pública desde el inspector. Independientemente de si dispara o no, todos los tiempos se restablecen y se vuelve a generan un tiempo de disparo (Murray, 2014).

3.1.3.5 Sistema de daño a enemigos

3.1.3.5.1 Diseño

Para finalizar con los aliens, debemos hacer que reciban daño y sean destruidos por los proyectiles del jugador. Cuando un proyectil del jugador impacte contra un alien (objetos con la etiqueta “enemigo”), este les mandara un mensaje “Destruir”. Lo que quiere decir que se ejecutará en el enemigo alcanzado el método con dicho nombre, obtenido de cualquiera de sus scripts. (si tiene más de un script, no puede haber más de uno que tenga un método “Destruir”). El cual, se restará uno de vida, y si quedó a cero, se destruirá sumando al marcador los puntos correspondientes. Las vidas por defecto son una pero si lo recordamos, había un enemigo enfurecido que no moría hasta recibir tres golpes, entonces usaremos este sistema para que solo el “prefab” de dicho alien tenga tres vidas. Ya veremos en su momento que hacemos con este alien que no muere de un golpe para que haya un “feedback” y el jugador pueda saber si realmente lo ha alcanzado.

Por otro lado, cuando el enemigo muere, debe dejar una pequeña animación de explosión del color correspondiente al mismo. Estas explosiones también han sido dibujadas con la herramienta piskel. Se creará una animación jugando con la escala del Sprite.

Ilustración 22. Sprites de explosión

3.1.3.5.2 Implementación

En primer lugar, comenzaremos haciendo que el proyectil del jugador detecte si ha impactado contra un enemigo y lo destruya con un mensaje. Además de esto, haremos también que destruya los proyectiles enemigos. Para los proyectiles usaremos el método “Destroy” directamente desde el del jugador ya que no requiere hacer nada más al ser destruido. Nuestro método quedara así:

```
private void OnTriggerEnter2D(Collider2D other)
{
        if (other.gameObject.CompareTag("Enemigo"))
        {
        //destruimos al primer enemigo que impacte la bala, si
        detectaramos la colision desde el enemigo un proyectil podría
        eliminar a varios
         other.gameObject.SendMessage("Destruir");
        }
         else if (other.gameObject.CompareTag("enemyBullet")) {
        Destroy(other.gameObject);
        }

        jugador.SendMessage("Recargar");
        Destroy(this.gameObject);
}
```

Seguidamente, crearemos los prefabs de explosiones para asignárselos a cada uno de nuestros aliens. Arrastramos cualquiera de los sprites a la escena y luego a la capeta prefab. A nuestro prefab lo llamaremos Explosión “Color”, siendo “Color” el color correspondiente. Le añadiremos un sencillo script que pasado un tiempo estipulado (1.5 segundos.) se autodestruya. Con el prefab en escena seleccionado, abrimos la pestaña “Animation” y añadimos la propiedad Transform>Scale. Ahora solo tenemos que pulsar el botón de grabar, colocar la línea de tiempo en aproximadamente 20 segundos y reducir la escala de nuestro objeto hasta llegar al mínimo de tamaño que alcanzará durante la animación. Paramos de grabar y guardamos la animación en una capeta de animaciones. Nos aseguraremos que este activado el “loop Time” de la animación para que se reproduzca en bucle. Solo nos queda añadirle al prefab un “animator” y asignarle nuestra animación. Por último, replicamos tres veces más el prefab cambiado el color de su nombre y su Sprite asociado y ya tendríamos las cuatro explosiones listas. A continuación, crearemos el método Destruir en el enemigo. Simplemente se resta una vida y comprueba si se le han acabado, en cuyo caso instancia una explosión y se destruirá. Por ahora no hará

falta nada más ya que no tenemos desarrollados aún algunos de los sistemas, como por ejemplo el de puntuación.

3.1.4 Sistemas del juego

Los sistemas del juego son aquellos que nos determinan las condiciones de victoria y de derrota. En nuestro caso, consideramos victoria hacer muchos puntos, pero finalmente siempre se alcanzará la perdida de las 3 vidas (en el modo de juego arcade) lo cual es una derrota. Estos sistemas funcionan de forma independiente, pero suele ir ligado a la interfaz ya que el jugador debe conocer esa información en todo momento. En la interfaz profundizaremos en la siguiente sección mientras que en esta, nos centraremos solo en su funcionamiento interno.

3.1.4.1 Sistema de vidas

3.1.4.1.1 Diseño

El sistema de vidas es algo bastante sencillo a priori. El jugador cuenta de entrada con tres vidas y cada vez que es alcanzado por los aliens se reducen en uno y la nave del jugador estalla, apareciendo al poco tiempo. Si se pierden todas las vidas, se acaba la partida. Ahora bien, hay que tener en cuenta algunas casuísticas para que el juego sea justo. Por ejemplo, cuando el jugador reaparece después de morir, tendrá unos segundos de invencibilidad los cuales solo podrán utilizarse para reposicionarse y esquivar posibles proyectiles que estuvieran cerca en el momento de reaparecer, es decir, no podrá morir de nuevo durante ese periodo, pero a cambio se le impide disparar. Esto equilibra un poco ya que le otorgaremos una ventaja temporal a cambio de sacrificar su único método de seguir sumando puntos. Por otro lado, también tenemos que tener en cuenta que mientras se ejecuta la explosión, tampoco podremos movernos ni disparar (no tendría sentido). Los sprites que conforman la animación de explosión ha sido descargados gratuitamente de la página web “pngWing”.

Ilustración 23. Animación de explosión

3.1.4.1.2 Implementación

Nuestro prefab de explosión solo tendrá un script en cual espera un tiempo determinado y luego se destruye. Además, tendrá una función pública para poder modificar ese tiempo desde otro sitio. Por ejemplo, desde el jugador. Esto nos hace más cómodo a la hora de hacer las pruebas para establecer ese tiempo y no tener que ir al prefab de la explosión cada vez (Como vemos es exactamente igual que para las explosiones enemigas por tanto lo reutilizaremos, solo hay que añadirle la nueva función). Por supuesto tiene un "Animator" con una animación, pero todo esto se hace automáticamente al arrastrar el "SpriteSheet" completo a la escena.

Para este sistema de momento solo necesitamos un nuevo script que asignaremos al jugador. Tendrá una referencia al script "player controller" para poder comunicarse con él a la hora de restringir el movimiento y otra, con el "Disparo jugador" para hacer lo propio. Tendremos también una variable booleana que nos determina si es vulnerable (si puede verse alcanzado por un ataque), otra para hacerlo inmortal independientemente de su vulnerabilidad (solo para pruebas en el juego real siempre será falso y no afectará). Cuando un proyectil alcanza al jugador se llama a su método "Destruir", este comprueba si es vulnerable, en cuyo caso desactiva el disparo e instancia una explosión. Ajusta la duración de la explosión con el valor de una variable que tendremos pública "duracionExplosion". Pasamos a estado invulnerable y hacemos invisible el objeto "Nave" el que contiene solo las piezas que conforman la nave (Murray, 2014). Esto lo hacemos para ahorrarnos una destrucción e

instanciación de dicho objeto ya que se sabe que o bien se ha acabado la partida o bien volverá a aparecer. Se resta una vida y se espera a que acabe la explosión lanzando una corutina usando la siguiente expresión.

```
        StartCoroutine("ExperarExplosion");
```

La corutina es la siguiente:

```
        IEnumerator ExperarExplosion()
{
         playerController.SendMessage("SetPuedeMoverse", false);
         yield return new WaitForSeconds(duracionExplosion);
         if (vidas <= 0)
        {
           //game over

                Time.timeScale = 0f;
         }
         else
         {
                StartCoroutine("Parpadeo");
                playerController.SendMessage("SetPuedeMoverse", true);
        }

}
```

Simplemente desactiva el movimiento espera un tiempo y después si no tiene vidas se para el tiempo, se acabó el juego. Sin embargo, si aún le quedan vidas se inicia una corutina de “Parpadeo” que hará parpadear la nave para informar al jugador de su estado de invulnerabilidad y reactivamos el movimiento.

```
        IEnumerator Parpadeo()
        {
                while (!vulnerable)
                {
                Parpadear();
                 yield return new WaitForSeconds(tiempoParpadeos);
                }
        }

         void Parpadear()
        {
                visible = !visible;
                nave.SetActive(visible);
        }
```

Para hacer el parpadeo mientras sea invulnerable alternaremos entre activar y desactivar la nave con un tiempo entre ambas acciones. Por último, en el “Update” comprobaremos cada frame si se ha vuelto invulnerable. En ese caso comenzaremos una cuenta de tiempo tras la cual se volverá vulnerable de nuevo y volveremos todo a su valor original.

```
void Update()
{
        if (!vulnerable && !inmortal)
        {
            tiempoDesdeMuerte += Time.deltaTime;
            if(tiempoDesdeMuerte >= tiempoInvulnerablidiad)
            {
                disparo.enabled = !disparo.enabled;
                vulnerable = true;
                StopCoroutine("Parpadeo");
                tiempoDesdeMuerte = 0f;
                nave.SetActive(true);
            }
        }
    }
```

3.1.4.2 Sistema de puntuación

3.1.4.2.1 Diseño

Como se describió en la descripción de los requisitos, nuestro sistema de puntuación no es únicamente sumar los puntos de cada enemigo destruido, pero vamos a hacer precisamente eso de momento ya que el sistema completo está programado para el siguiente sprint.

3.1.4.3 Implementación

Comenzaremos creando un script “ContadorPuntos” que tendrá una variable puntos y una referencia a un objeto “Text”. Para crear este objeto “Text” Crearemos un “Canvas” nuevo (ya que el que hay solo aparece cuando nos encontramos en móvil) y lo añadimos dentro de él en la esquina superior izquierda. Este texto tendrá escrito “Puntuación: 0” y dentro del script iremos actualizando este valor sumándole los puntos que los enemigos le pasen por mensaje al morir.

3.1.5 Interfaz

3.1.5.1 Diseño

Para la interfaz queremos dos franjas negras con un poquito de trasparencia a cada lado de la pantalla para mostrar allí la información. De momento, solo colocaremos las vidas en la parte superior derecha por medio de un letrero indicativo y tres corazones que irán desapareciendo cada vez que muera.

Como vimos en la sección anterior, el elemento que indica la puntuación ya está colocado solo aclarar que este elemente va fuera de la franja negra, ya que podrá alcanzar cifras muy grandes si el jugador es habilidoso y necesitamos casi todo el ancho de la pantalla. Ni que decir tiene que será necesario tener en cuenta el ancho de estas franjas para determinar los nuevos “bordes de la pantalla”. El corazón ha sido dibujado en Piskel.

Ilustración 24. corazón para vidas

Ilustración 25. Diseño de la interfaz

3.1.5.2 Implementación

Para empezar, creamos el nuevo “Canvas” y lo ajustamos para que tenga como resolución objetivo 2960 x 1440 y el “Match” se colocamos en uno para que el escalado sea 100% fiel al original. Gracias a esto, el espacio que ocupa tendrá siempre las mismas proporciones en cualquier resolución.

Creamos dentro dos imágenes que usaremos para crear las franjas, las configuramos con un color negro con en canal alpha un poco bajado para que se distingan las estrellas del fondo. Las ajustaremos para que ocupen todo el alto de la pantalla y midan lo mismo de ancho. Posteriormente colocaremos cada una a un lado. Desplazamos el Texto de la puntuación de forma que no lo ocultemos. Colocamos un nuevo texto “Vidas”. Como fuente usaremos “Jupiter” incluida en el paquete descargado gratuitamente de la tienda llamado “Unity simples: UI” que contiene algunos elementos que nos será útiles para la creación de los menús.

Justo debajo colocaremos los tres sprites de corazones. El texto y los corazones los englobaremos en otro objeto vacío al que llamaremos “Vidas” y haremos hijo del panel de la derecha. Es importante que los el texto sea el último de los hijos y que los corazones vayan en orden empezando por la de la izquierda. Este nuevo objeto contendrá el script “ContadorVidas” que almacenara en un vector los sprites que representa las vidas y que son hijas de este objeto. Esta es la razón por la que el orden era importante ya que recorre los hijos de arriba hacia abajo y no podría encontrarse con el texto ya que solo almacena sprites. El funcionamiento es tan simple como que, cuando el jugador es destruido, justo después de restar las vidas, le envié un mensaje a este pasándole como parámetro la variable vidas (ya restada). Luego en nuestro script recibiremos esta llamada con un método que oculta el Sprite cuya posición en el vector corresponde con el valor recibido. De esta forma al perder la primera vida ocultara la posición 2, con la segunda la 1 y con la ultima la 0.

```
void RestarVidas(int pos) {
	vidas[pos].gameObject.SetActive(false);
}
```

3.1.6 Sonido

3.1.6.1 Diseño

Lo apropiado en un proyecto de estas características sería grabar y componer cado sonido y música que se vayan a utilizar para evitar todo problema de copyright, pero como este juego no se va a publicar en un principio, usaremos sonidos y música descargada de internet. Pondremos sonidos, de momento, al disparo del jugador, a la muerte del jugador y de los aliens, a cada movimiento de los aliens marcando el ritmo, y además pondremos una música de fondo como ambiente. La canción elegida está basada en Space Invader Extreme. Es la música del primer nivel de este juego y ha sido obtenida de la página Video Game Music. Los demás son de Sonidos MP3 gratis y obtenidos de YouTube.

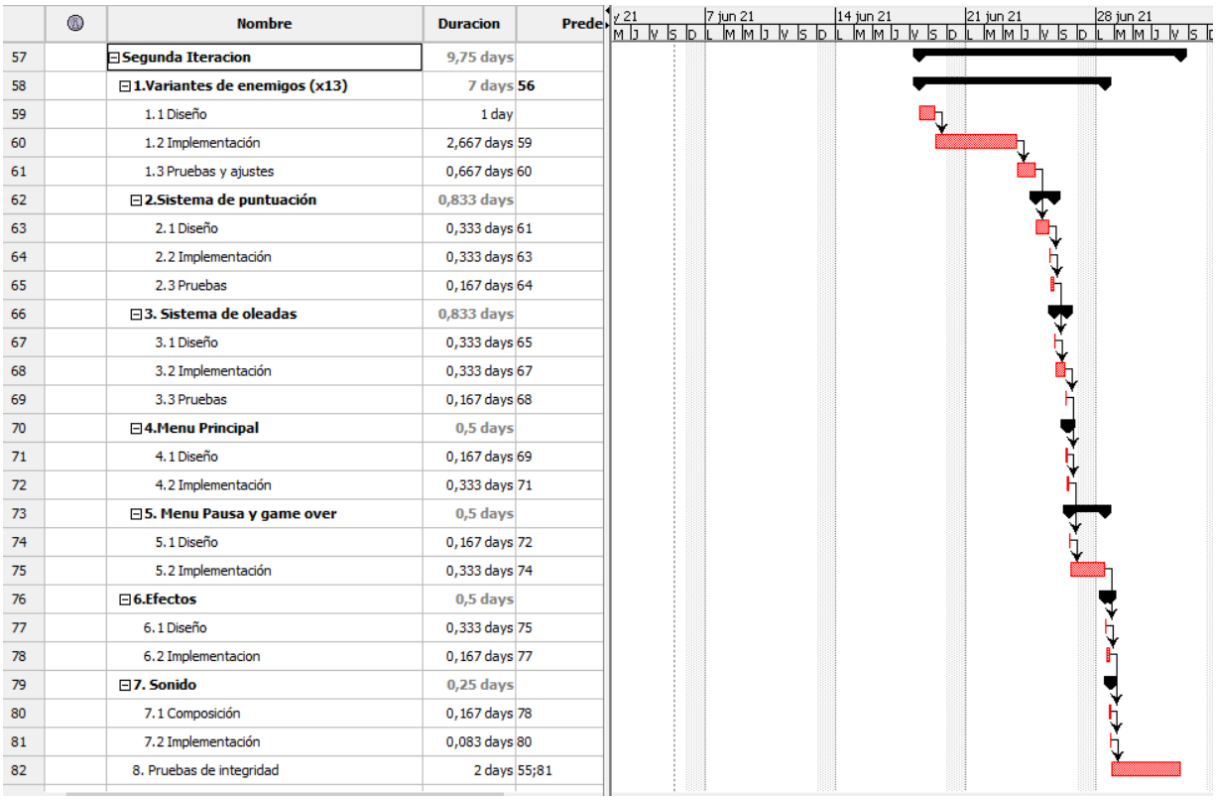
3.1.6.2 Implementación

En la cámara principal añadimos el Componente “Audio Source” y le ponemos como clip la música que hemos obtenido para ambiente. Para que se reproduzca nada más empezar el nivel y se repita en bucle cada vez que termina la música. Necesitamos configurar el “Audio Source” activándole las opciones “Play On Awake” y “Loop”. Este será el único que tenga la opción “Loop” ya que será el único audio que se repita constantemente. Eso sí, tendremos que bajarle el volumen más que al resto para que todo se escuche correctamente. el “audio Source” al “Player”, la explosión del jugador y la de los enemigos. Para las explosiones, le asignamos a cada una su clip y con el “Play On Awake” activado una vez se instancien se reproducirán.

Para el jugador tenemos que Escribir un poco de código ya que se tiene que reproducir el sonido de disparo solo cada vez que dispare. Tampoco le asignaremos el clip directamente sino por código ya que están planificados otros tipos de disparos cuyo sonido será distinto. Le añadimos al script “Disparo Jugador” una propiedad que obtendrá el “Audio Source” y otra que guardará el clip del disparo. En el momento de disparar asignamos el clip al “Audio Source” y lo reproducimos (Murray, 2014).

3.2 Iteración 2

Una vez ya tenemos las mecánicas principales de nuestro juego desarrolladas, es hora de entrar en profundidad en otras más complejas. Al final de la segunda iteración tendremos completo nuestro modo de juego Arcade, el cual incluye un sistema de generación de oleadas aleatorias, un sistema de puntuación que premia al jugador por conseguir eliminar a muchos enemigos de forma consecutiva aplicando un multiplicador a las puntuaciones base que estos te dan, podremos encontrarnos con versiones modificadas de cada alien, los cuales tendrán habilidades especiales y para finalizar contaremos con un sistema de incremento de dificultad gradualmente a medida que el jugador va sobreviviendo cada vez a más oleadas. También desarrollaremos el menú de pausa, de "Game Over" y la escena del menú principal.



	Nombre	Duracion	Prede
57	**Segunda Iteracion**	9,75 days	
58	**1.Variantes de enemigos (x13)**	7 days	56
59	1.1 Diseño	1 day	
60	1.2 Implementación	2,667 days	59
61	1.3 Pruebas y ajustes	0,667 days	60
62	**2.Sistema de puntuación**	0,833 days	
63	2.1 Diseño	0,333 days	61
64	2.2 Implementación	0,333 days	63
65	2.3 Pruebas	0,167 days	64
66	**3. Sistema de oleadas**	0,833 days	
67	3.1 Diseño	0,333 days	65
68	3.2 Implementación	0,333 days	67
69	3.3 Pruebas	0,167 days	68
70	**4.Menu Principal**	0,5 days	
71	4.1 Diseño	0,167 days	69
72	4.2 Implementación	0,333 days	71
73	**5. Menu Pausa y game over**	0,5 days	
74	5.1 Diseño	0,167 days	72
75	5.2 Implementación	0,333 days	74
76	**6.Efectos**	0,5 days	
77	6.1 Diseño	0,333 days	75
78	6.2 Implementacion	0,167 days	77
79	**7. Sonido**	0,25 days	
80	7.1 Composición	0,167 days	78
81	7.2 Implementación	0,083 days	80
82	8. Pruebas de integridad	2 days	55;81

Ilustración 26. Diagrama de Gantt Iteración 2

3.2.1 Variantes de enemigos

3.2.1.1 Diseño

Los diseños han sido creados con Piskel basados en los anteriores, pero co un patrón distinto que los hace más amenazantes. También incluiremos los Ovnis que ya estaban también en el juego original.

Calamar enfurecido		
	Versión modificada de color rojo, se caracteriza por cambiar su disparo por una bomba. Si el jugador entra en el rango de explosión morirá.	Puntuación:200
	Versión modificada de color azul, se caracteriza por cambiar su disparo por un rayo láser que se mantendrá durante un breve tiempo.	Puntuación:200
	Versión modificada de color verde, se caracteriza por cambiar su disparo por tres proyectiles simultáneos (este solo contará como uno en el recuento total de proyectiles en escena).	Puntuación:200

Tabla 17. Diseños variantes de Calamar

Cangrejo enfurecido		
	Versión modificada de color rojo, se caracteriza por tener más vida que los demás. Morirá si recibe tres disparos.	Puntuación:150
	Versión modificada de color azul, se caracteriza por tener un escudo que refleja el primer proyectil que lo alcance.	Puntuación:150
	Versión modificada de color verde, se caracteriza por que al morir aparecerán dos cangrejos normales de color verde.	Puntuación:150

Tabla 18. Diseños variantes de Cangrejo

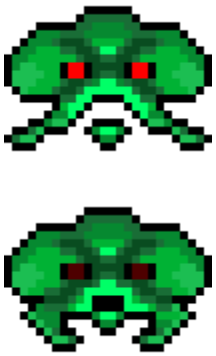
Pulpo enfurecido		
	Versión modificada de color rojo, se caracteriza por lanzar un proyectil que al ser destruido se divide en dos.	Puntuación:120
	Versión modificada de color azul, se caracteriza por volverse invulnerable cada cierto tiempo se representa con un ligero cambio físico que lo hace semitransparente.	Puntuación:120
	Versión modificada de color verde, se caracteriza por disparar un proyectil destructor que si llega a la altura del jugador libera un rayo horizontal que destruirá nuestra nave. Por lo tanto, la estrategia será destruir este proyectil antes de que llegue abajo.	Puntuación:120

Tabla 19. Diseños variantes de Pulpo

Para los Ovnis solo se ha diseñado el blanco y se le cambiara el color desde Unity.

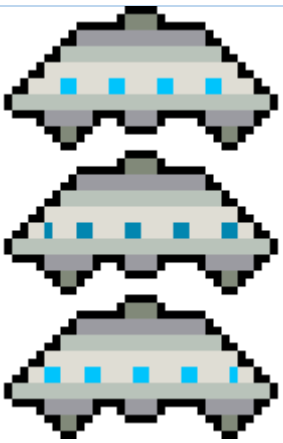
OVNI		
rojo	Versión normal de color rojo.	Puntuación: 300
azul	Versión normal de color azul.	Puntuación:300
verde	Versión normal de color verde.	Puntuación:300
	Versión normal de color blanco.	Puntuación:300

Tabla 20. Diseño de Ovnis

Por otro lado, los proyectiles únicos de cada enemigo han sido obtenidos del Spritesheet del que fue extraído el del jugador. Mientras que los efectos, como escudo, carga de láser han sido diseñado con piskel y la explosión de la bomba se obtuvo de freepng.es.

Ilustración 27. Sprites de animación de cargado

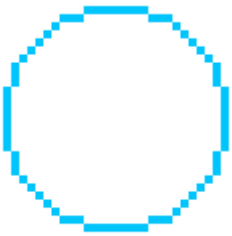


Ilustración 28. Sprite de escudo

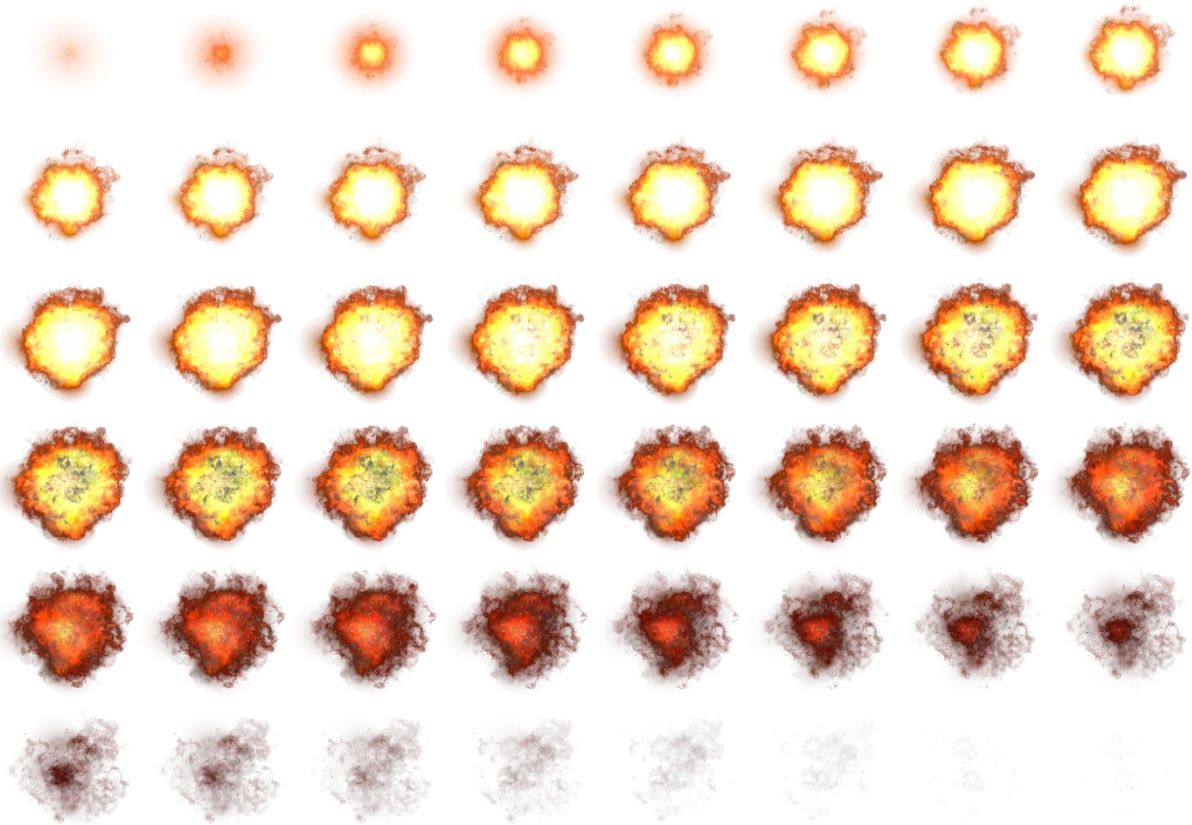


Ilustración 29. Animación de explosión de bomba

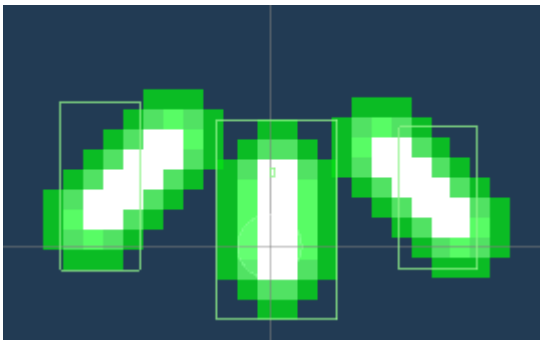


Ilustración 30. Sprite de Proyectil triple

Proyectil bomba

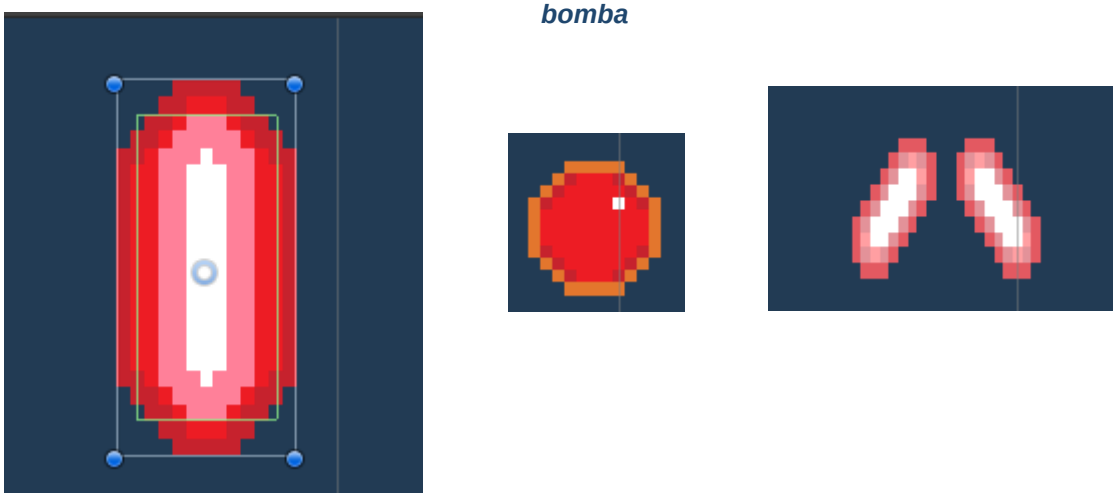


Ilustración 31. Sprites de proyectil divisible

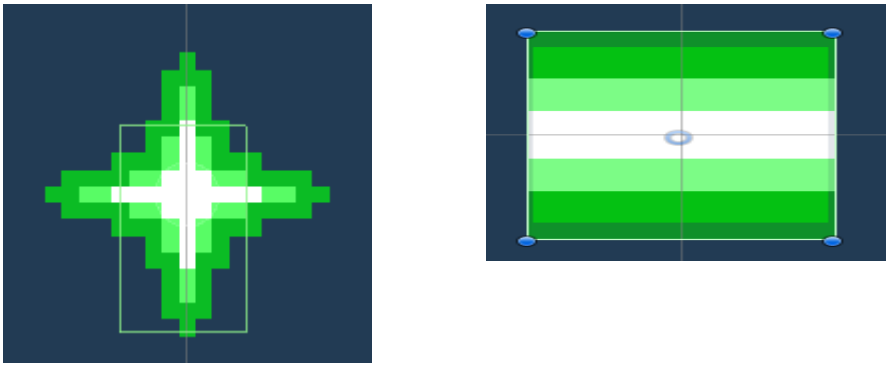


Ilustración 32. Sprites de destructor y rayo

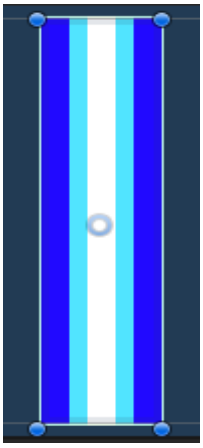


Ilustración 33. Sprite de rayo láser

3.2.1.2 Implementación

Para empezar, crearemos los prefabs de todos los enemigos copiando lo ya existentes y modificándoles los sprites y la puntuación que otorga cada uno de ellos. A continuación, Comenzaremos a desarrollar las características únicas de cada uno de los enemigos.

3.2.1.2.1 Calamar rojo enfurecido

A este prefab solo habría que añadirle como proyectil la bomba, la cual veremos a continuación como programarla.

Creamos el prefab con su correspondiente sprite y “box collider”. Además, le añadiremos el script “EnemyBullet“.

Añadimos ahora un nuevo script al que llamaremos “Bomba”. En el detectaremos el momento en que alcanza la altura del jugador para instanciar la explosión. Si toca al jugador antes, explotará en ese momento. Tras explotar, esperará al tiempo que dure la misma y luego se destruirá a sí misma junto con la explosión.

La explosión es otro prefab que contiene la animación de los sprites acompañada de una animación que ajusta el “collider” según el estado de la explosión. El “collider” se ajustará al tamaño del Sprite en cada momento de la animación. Por último, añadiremos un script que dañe al jugador si entra en el “collider” en cualquier momento durante la exploxión.

3.2.1.2.2 Calamar azul enfurecido

Como hemos definido, este alien cargará y disparará un rayo láser que se mantendrá durante un corto periodo de tiempo. El procedimiento será instanciar un objeto que se colocará debajo del enemigo y cargará el láser con una animación y un sonido que advierta al jugador. Cuando se pase el tiempo estimado de carga, este objeto instanciará tantas copias de láser con sean necesarias para llegar hasta abajo. Colocaremos el tiempo entre disparos de este enemigo entre 4 y 7 segundos.

Creamos el prefab de “Cargar Láser” con su correspondiente animación y el script “EnemyBullet”. En otro script definiremos el comportamiento específico de este objeto. Tendremos un tiempo de carga y una duración del rayo. Ambas se han definido en 2 segundos. Cada frame irá contando hasta que se cumpla el tiempo de carga y después, hasta que se cumpla el tiempo del rayo. Una vez cargado, se instanciará en vertical desde la posición del objeto “cargar láser”, tantos rayos como sean necesarios para que este llegue a la parte inferior. Cada rayo ocupará una unidad de espacio.

Realmente no es este script como tal el que instancia todos los láseres, sino el que inicia la iteración instanciando el primero. Después, en el láser, su script comprueba a que altura esta y si aún no está a la altura del jugador, instanciará uno más debajo de él. Cuando se acabe el tiempo de rayo, tanto estos como el objeto cargar láser se destruirán. En lo referente a sonido, simplemente comenzará desde el principio con el sonido de cargar láser y en el momento que se lance el rayo se cambiará mediante código al sonido del láser. Po último, el láser detectará el contacto con el jugador para destruirlo.

3.2.1.2.3 Calamar verde enfurecido

Este enemigo instanciará al disparar un “Disparo triple” con intervalos de entre 2 y 6 segundos. El prefab de este disparo está formado por tres proyectiles. “Disparo triple” tiene el script “EnemyBullet” ya que los tres proyectiles cuentan como un único disparo. Además, tendrá el propio script “Disparo Triple” que simplemente desliga al proyectil de su padre en la jerarquía (el enemigo que lo disparó) y le proporciona el movimiento vertical. El proyectil del centro solo tiene que detectar al jugador para matarlo mientras que los laterales tendrán “Proyectil Diagonal” en el cual cada uno definirá mediante un desplegable en el inspector, una de las dos direcciones (izquierda o derecha) y según la misma se añadirá un movimiento en dicha dirección. Esto sumado al movimiento vertical del padre de los proyectiles en la jerarquía hará que el movimiento real sea en diagonal. Evidentemente los diagonales también mataran al jugador.

3.2.1.2.4 Cangrejo rojo enfurecido

Este enemigo sencillamente tiene 3 vidas en lugar de una, pero tendremos que hacer una pequeña modificación en el script “Enemigo” para mostrar un poco de “feedback” al recibir el golpe y no morir que indique al jugador que ese enemigo ha recibido el golpe, pero tiene más vida. Esto lo conseguimos haciendo que parpadee como el jugador al resucitar, pero en este caso con una sola vez con un intervalo de 0.2 segundos es suficiente.

3.2.1.2.5 Cangrejo azul enfurecido

Para este enemigo crearemos un prefab de escudo que asignaremos como hijo al de este Alien. Para evitar que se elimine antes de que el escudo haya desaparecido del todo por un fallo de colisiones, desactivaremos el “collider” mientras el escudo está activo. El escudo simplemente devolverá el proyectil del jugador transformado para que pueda hacer daño al jugador (instancia un proyectil con etiqueta de “EnemyBullet”). Y después desaparece activando el collider del enemigo.

3.2.1.2.6 Cangrejo verde enfurecido

El cangrejo verde enfurecido soltará dos enemigos normales más pequeños al ser destruido. (Ambos ocupan el mismo espacio que uno de tamaño normal). Para ello al morir instanciará un objeto vacío que tiene el script asociado que hará aparecer estos dos enemigos de tamaño reducido Cada enemigo hijo es exactamente igual que los enemigos normales, pero con un tamaño reducido, por tanto se suman individualmente al recuento de enemigos restantes ya que tienen el script enemigo.

3.2.1.2.7 Pulpo rojo enfurecido

Este enemigo dispara un proyectil de color rojo que si es alcanzado por un disparo del jugador se dividirá en dos, los cuales saldrán cada uno en una diagonal. Para conseguirlo, el prefab de este proyectil tendrá como hijos a los dos fragmentos pero estarán desactivados. Estos se activarán al recibir el golpe por parte del proyectil del jugador. Cada uno de los fragmentos tendrá asociado un script que le proporciona movimiento en horizontal ya que el vertical se lo proporciona el padre que no se destruye simplemente se oculta y se desactiva su “collider” para que no mate al jugador por error.

3.2.1.2.8 Pulpo azul enfurecido

Este enemigo se hará invulnerable cada cierto tiempo. Para representarlo le bajamos el canal alfa para hacerlo semi transparente y desactivamos su “collider”.

3.2.1.2.9 Pulpo verde enfurecido

Este enemigo disparará un proyectil que se desplazará lentamente hacia abajo y si alcanza la altura del jugador extenderá un rayo a izquierda y derecha cubriendo todo el ancho de la pantalla lo que será letal para el jugador. La única forma de librarse, será destruyendo el proyectil antes de que sea demasiado tarde. Ese enemigo cargará el disparo de igual forma que el del rayo y también tendrá un sonido que alarmará al jugador. El proyectil al que llamaremos destructor, tendrá el script “bomba”, pero en lugar de instanciar una explosión instanciará el prefab rayo destructor. Este, se encargará de instanciar tantos rayos como sean necesarios para cubrir el ancho de la pantalla.

3.2.1.2.10 OVNIS

Para que aparezcan los ovnis eventualmente tenemos que crear unos "generadores" que lo instancien. Colocaremos uno a la izquierda y otro a la derecha a altura de 6 y fuera de la vista de la cámara. Englobamos ahora estos generadores en un objeto vacío el cual será quien tenga la función de instanciar y elegir por qué lado aparecerá. Cada 10 segundos ejecutará la función de probabilidad para ver si instancia un ovni o no. Si supera "la prueba" se escogerá un número aleatorio que determinará qué color de ovni sale (este es su posición en el array) y otro para el generador (izquierda o derecha). Echo esto se instanciará un ovni del tipo elegido.
Los ovnis determinaran la dirección en la que se mueven según su posición de inicio.

3.2.2 Sistema de puntuación.

3.2.2.1 Diseño

El sistema de puntuación a grandes rasgos, parte de la base de que cuantos más enemigos seguidos destruyas más puntos obtienes de ellos. Esto se obtiene con un multiplicador que aumenta según el contador de cadena que se lleve, siguiendo la Tabla 1. Relación bonus-cadena. Tanto la puntuación como la cadena y el bonus se mostrarán constantemente en pantalla durante la partida.

3.2.2.2 Implementación

Para llevar a cabo este sistema comenzaremos creando un script que asignaremos al Game Manager. Este se encargará de llevar el recuento de la cadena de asesinatos que se lleva y el multiplicador de bonus correspondiente.

Para empezar, crearemos un arreglo de 16 posiciones en el que introduciremos el número de cadena que hace cambiar el multiplicador al marcado por el índice más uno, por ejemplo, en la posición 15 colocaremos un 100, lo que nos indica que necesitas encadenar 100 enemigos para que el multiplicador aumente a x16 (15+1) .

Con otra variable que llamaremos fase, almacenaremos el índice del arreglo que contiene el número de cadena que tenemos que comprobar si se ha alcanzado para aumentar el multiplicador. De esta forma cada vez que se destruya un alien este llamará a una función que aumentará la cadena y posteriormente comprobara con la

fase actual si el recuento de cadena ha alcanzado la fase requerida para cambiar el multiplicador. Este script también se encarga de comprobar si ha pasado el tiempo suficiente desde el último aumento de cadena, lo que implica el reseteo de todas las variables (fase, cadena, bonus).

Por otro lado, tendremos el contador de puntos, que asignaremos al elemento del "Canvas" de tipo texto encargado de mostrarlo. Cada enemigo le dirá al morir cuantos puntos debe aumentar con el multiplicador ya aplicado y este lo sumará a la variable puntos y lo actualizará en el texto. De igual forma, los textos cadena y bonus consultaran al script controlador estas variables y las mostraran. (Murray, 2014)

3.2.3 Sistema de oleadas

3.2.3.1 Diseño

El sistema de oleadas es el que se encarga de generar las diferentes oleadas de enemigos en el momento en que se acaban los de la oleada anterior. Para ello debe llevar el recuento de las oleadas y comunicárselo al Regulador de dificultad. Este último se encarga de reajustar la dificultad modificando las diferentes variables que se utilizan en la generación.

El regulador de dificultad actuará de la siguiente manera:

- **Cada 10 oleadas** si la altura inicial es mayor a 0 se disminuye en uno, es decir los enemigos comenzaran en una fila más abajo. Si esta es menor que 0 se restablece a 4.5.
- **Cada 3 oleadas** se aumenta la probabilidad de disparo en 10 % siempre que esta, sea menor a 100. En caso contrario solo si los proyectiles permitidos son menos de 8, se restablece a 30 la probabilidad de disparo y se aumenta en uno los proyectiles permitidos.
- **Cada 2 oleadas** si el ritmo de Juego con el que comienza la oleada es mayor a 0.2, se reduce en 0.2. Si no, al ritmo inicial (al comienzo de la partida, el cual no se ha modificado aun, ya que se almacenará aparte) se reduce en 0.2 y el ritmo de inicio actual se reajusta a este nuevo ritmo por defecto. Esto hará que la siguiente aleada sea más lenta que la anterior ya que se ha reseteado, pero será 0.2 más rápida que la primera de todas.

Esto se repetirá cada vez con más frecuencia (debido a que se tarda 2 oleadas menos en llegar valor mínimo de 0.2). Si llega el momento en que el valor por defecto ya está en 0.2 no se continuará disminuyendo y el ritmo inicial será 0.2 para todas las oleadas hasta el final de la partida.

- **Cada oleada** se reajustará la probabilidad de instanciarse de forma parecida a la del ritmo de juego. Por un lado, tenemos la probabilidad inicial y por otro la que se está efectuando actualmente. Cuando la actual llega a 100 sumándose de 10 en 10 cada oleada, se aumentará en 10 la probabilidad inicial y la actual se igualará a esta nueva probabilidad inicial.

3.2.3.2 Implementación

Creamos el Script de "regulador de dificultad" (que asignamos al "GameManager") con las variables especificadas en el diseño y con una función que pasándole como referencia un entero, el número de oleadas, calcule que variables tienen que modificarse.

Para saber si han pasado X oleadas, se divide el número de oleada por X y nos quedamos con el resto. Si este es 0 se cumplirá la condición. A continuación, creamos el script "Generador de oleadas" asignado a "HordaAlien". Este dispondrá de un vector de "GameObject" que contendrá todos los enemigos disponibles para instanciar. En este punto haremos un pequeño "truco" para aumentar la probabilidad de que aparezcan enemigos normales en contra de los enemigos enfurecidos. Para ello meteremos en el vector 2 enemigos normales de cada tipo y color en lugar de solo uno. Tendremos también dos variables que nos indican las filas y columnas de enemigos que tendremos que son 5 y 13 respectivamente. También llevaremos la cuenta de los enemigos restantes que quedan para saber cuándo cambiar de oleada. Además, es aquí donde llevaremos la cuenta de oleadas con una variable estática que podrá consultarse desde el texto de la interfaz para actualizarse.

Antes de generar la primera oleada creamos una matriz de valores booleanos igual a la disposición de los enemigos. Cada valor determinará si en esa posición se instanciará un enemigo o no. Recorremos esta matriz y la rellenamos con el valor que nos devuelve nuestra función de probabilidad según la probabilidad de aparición que

consultamos del regulador, el cual tendremos referenciado. Establecemos la altura de la primera fila con el valor de “AlturaInicial” dada por el regulador.

Hecho esto, procedemos a recorrer el vector de nuevo y para cada fila el valor de X se pondrá en -6 que es la posición más a la izquierda en la que aparecerán enemigos. Para cada posición de la matriz si se debe instanciar un enemigo se genera un valor aleatorio entre 0 y la longitud del vector de enemigos -1 para determinar cuál se instanciará. Posteriormente el enemigo generado pasará a ser hijo de “HordaAlien” para que pueda funcionar el sistema de movimiento. Esta función de generación se llamará cada ver que la cuenta de alien restantes llegue a 0 o al comienzo de la partida.

Tendremos que modificar a los aliens para que se sumen al ser instanciados o se resten al morir. Para ello creamos dos funciones públicas que se podrán llamar desde el script de los enemigos. No podemos sumar los enemigos directamente desde el generador de oleadas porque recordemos que hay enemigos que instancian a otros, por lo que fastidiarían el recuento y se generarían enemigos antes de que se acaben con todos.

3.2.4 Menús e interfaz

3.2.4.1 Diseño

En esta sección definiremos el diseño final de la interfaz durante el juego y de los menús de inicio, Game over y pausa. Para la interfaz solo hay que añadir os contadores de oleadas, cadena y bonus en la franja negra de la izquierda.

Ilustración 34. Interfaz definitiva

Para los menús usaremos componentes descargados de la “asset store”. El de inicio tendrá un fondo con temática espacial y un panel en el que se encuentre el título y los botones de modo arcade, modo aventura, salir, etc. El menú de pausa el cual se activará al pulsar escape en pc, mientras que usaremos un botón en la esquina superior derecha en dispositivos móviles. Contará con un botón para reanudar y otro para salir al menú principal, mientras que el de “Game Over” será igual, pero sustituyendo reanudar por reiniciar.

Ilustración 35. Icono de menú de pausa

3.2.4.2 Implementación

Colocamos los tres textos para cadena, oleada y bonus en el panel izquierdo dentro del canvas. Cada uno de ellos tendrá asociado un script muy parecido al que hicimos para la puntuación que irá cogiendo el valor a representar y actualizará su texto con el mismo.

Para hacer el menú de pausa Creamos en el canvas el objeto vacío “Pausa” que incluirá los dos botones ya mencionados, un texto, un panel que los engloba con una imagen extraída del pack “UI Samples” el cual es un panel azul con un borde en blanco. También añadiremos un objeto al que llamaremos oscuridad que constará de un Sprite negro que abarque toda la pantalla y cuyo canal alfa este más o menos a la mitad de su valor máximo. Esto le dará un toque semitransparente y oscurecerá la vista del juego mientras está en pausa. Copiaremos este menú para con un par de cambios tener el de “Game Over”. Estos objetos estarán desactivados por defecto siendo activados cuando el jugador para el juego o muere respectivamente.

Para detectar cuando se ha pausado el juego (ESC o botón de pausa en móviles (Sinicki, 2017)) usaremos la siguiente función en el PlayerController dentro del Update.

```
if ( CrossPlatformInputManager.GetButtonDown("Cancel"))
        {
            if (!pausado)
            {

                Time.timeScale = 0f;
                pausado = true;
                pausa.SetActive(true);
                FindObjectOfType<DisparoJugador>().puedeDisparar = false;


                puedeMoverse = false;
            }
            else
            {
                Reanudar();

            }

           }
```

Para dispositivos móviles copiaremos el botón de disparar, le cambiaremos el Sprite por el de menú, y en el script “button handle” le pondremos de nombre “Cancel” para que sea reconocido de igual forma que la tecla “Scape” en PC. La función Reanudar devolverá todos los valores modificados tras la pausa a la normalidad.
Para los botones Reanudar, Reiniciar y salir crearemos un script con diferentes funciones y cada botón hará uso de una de ellas. Reanudar le envía un mensaje al “PlayerController” el cual ejecutará su propia función Reanudar ya mencionada. Reiniciar, vuelve a cargar la escena de nuevo mientras que salir simplemente nos carga la escena que contiene el menú principal.

Para dicha escena, colocaremos un fondo animado de la temática espacial y un panel que también se girará ligeramente según la posición del ratón. Tanto el fondo como el panel y su funcionalidad viene incluido en el pack “UI Samples”. En el panel colocaremos los diferentes botones que nos cargaran las distintas escenas. En nuestro caso solo tenemos la escena de juego del modo arcade. También incluiremos un botón que finalice la ejecución del juego.

Ilustración 36. Menú de pausa

Ilustración 37. Menú Game Over

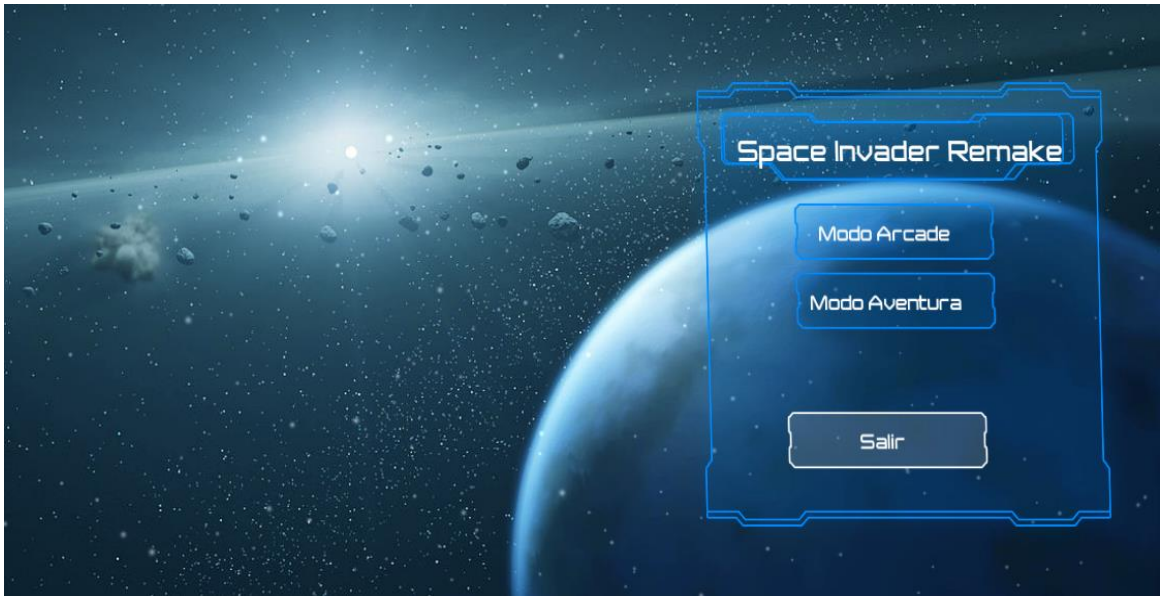


Ilustración 38. Escena menú principal

3.3 Pruebas finales

Las pruebas, son una parte muy importante durante el desarrollo de cualquier producto software y como tal, se han ido haciendo durante la ejecución del mismo. Unity permite ejecutar el juego en cualquier momento con lo cual podemos probar cada nueva mecánica o elemento de nuestro juego. En estas pruebas se han ido encontrando y corrigiendo diversos bugs o ajustando las diversas variables como probabilidades, velocidades, etc.

En los apartados anteriores se ha explicado y detallado solo la versión final de cada componente y scripts después de todas las pruebas y cambios ya que si pusiera todas las versiones y cambios este documento se haría demasiado largo, pero en este apartado enumeraremos algunos de los bugs más importantes que se han detectado y solucionado gracias a las pruebas realizadas tanto en el editor como en las pruebas finales tras exportar el juego para pc y como .apk para Android.

- Para ajustar el canvas hemos puesto en su “canvas scaler” en 2960 y 1440 como resolución de referencia. Además, el Match se colocó en uno para que el escalado sea 100% fiel al original. gracias a esto el espacio que ocupa

tendrá siempre las mismas proporciones en cualquier resolución y por tanto podemos decir después de comprobarlo a mano moviendo nuestro “player” que cada franja ocupa 4 unidades en coordenadas de mundo que serán restadas a la hora de calcular el ancho de la pantalla para definir los límites del movimiento del jugador y los aliens.

- Se han cambiado de script algunos parámetros que varían durante la partida para ajustar la dificultad y se han concentrado en un script, “regulador dificultad”.
    - “Proyectiles permitidos” se traslada desde “controlador alien”
    - La variable “ritmoDejuego” se traslada desde “movimientoHorda”
- Se ha ajustado el script enemigo en el método destruir para poder ponerles vidas para enemigos que aguanten más de un golpe.

- Se ha separado el comportamiento de un proyectil normal del script “enemyBullet”, este último ahora solo cuenta proyectiles sea cual sea tu tipo.
- Se ha parametrizado en el enemigo el tiempo máximo y mínimo entre disparos que antes era fijo. ahora cada enemigo podrá tener unos tiempos. El motivo fue que como el rayo destructor permanece más tiempo en pantalla que un proyectil normal a veces se solapaban dos disparos de rayo del mismo alien.
- La explosión ahora se instancia como hijo de horda alien para que el cangrejo verde enfurecido, al instanciar a sus hijos se instancien allí. Posteriormente en el script explosión quitamos este parentesco ya que sino cuando se mueve la horda se movería también la explosión.
- Se ha corregido un bug mediante el cual no se respetaba el máximo cargador de proyectiles del jugador y a veces había más balas de las permitidas por esta variable.
- Se ha cambiado la forma de llevar la cuenta de enemigos restantes. Antes se sumaban en el generador cuando los instanciaba y ahora cada enemigo al instanciarse, indica que sume uno a dicha variable. Este cambio se hace

pensando en el modo aventura en el cual, el generador no generará a los enemigos ya que ya estarán preestablecidos, además de que algunos enemigos instancian a otros durante el juego y también tiene que sumarse como enemigos restantes sino se crearía una nueva oleada antes de eliminar a todos los de la anterior.

- Se ha modificado el "player controller" para permitir activar y desactivar el menú de pausa.
- Se ha creado los métodos "get" y un "set" de la propiedad privada "PuedeDisparar" para que, al poner en pausa se desactive el disparo desde el "Player controller".
- Se han ajustado el tamaño de los controles de dispositivos móviles porque tras la prueba se ha comprobado que el tamaño inicial dificultaba la jugabilidad.

- Se ha corregido un bug mediante el cual, cuando se ejecutaba el juego en dispositivos móviles o en pc fuera del editor de Unity, un error en la instanciación o recuento de enemigos restantes provocaba que al reiniciar una partida o volver a jugar después de salir al menú principal, hacía que de repente los alien se movieran a una velocidad demasiado rápida que se sale de los límites establecidos para el movimiento de los aliens y hacía imposible sobrevivir.

3.4 Tiempo estimado vs Tiempo real empleado

	Tiempo estimado	**Tiempo real**
Primera iteración	44 horas	40 horas
Segunda Iteración	45,5 horas	43 horas

Tabla 21. Comparativa de tiempos

Como podemos apreciar la estimación de tiempo no está mal realizada y se acerca bastante al tiempo real invertido, sin embargo, podemos apreciar que en general se ha tardado un poco menos de lo estimado.

4 CONCLUSIONES Y TRABAJOS FUTUROS

Desde muy pequeño me había interesado por el mundo de los videojuegos y poco a poco se fue expandiendo mi interés, de solo jugarlos a cómo se desarrollan. Gracias a este proyecto he podido poner en práctica todo lo que he aprendido cursando ingeniería informática y combinarlo con los conocimientos sobre programación de videojuegos adquiridos por mi cuenta.

El videojuego es uno los productos software más complejos por que incluye conceptos de muchos ámbitos tanto de la informática como fuera de esta. Este proyecto también me ha ayudado a la hora de organizarme, planificar y desarrollar una idea previamente a su desarrollo, así como a su documentación posterior. Dadas las limitaciones de tiempo, muchas de las características ideadas para el juego, se han quedado fuera de este desarrollo por tanto podría ser futuras mejoras al estado actual del videojuego:

1. Creación del modo Aventura con una cantidad considerables de niveles con objetivo y limitaciones variadas en cada uno.
2. Diseño de jefes para determinados niveles del modo aventura.
3. Diseño e implementación de distintas armas temporales que modifiquen la forma de disparar del jugador que dejarían caer ocasionalmente los enemigos al derrotarlos.
4. Implementación de modificadores para los enemigos que cambien por completo el comportamiento de los mismos y la forma de derrotarlos.
5. Diseño de nuevos tipos de enemigos.
6. Implementación del sistema de desbloqueo de enemigos y mejoras. Estas se desbloquearían en el modo aventura y a partir de entonces poder encontrarlos en el modo arcade.
7. Sistema de ranking de puntuaciones online para competir con otros jugadores por alcanzar la mayor puntuación.
8. Sistema de logros.

9. Ampliar la variedad de escenarios y pistas musicales para que no se hagan repetitivas tras un largo tiempo jugando.
10. Monetizar el juego de alguna forma ya sea con anuncios o compras internas en el juego como aspectos para el jugador, nuevos escenarios, nuevos niveles, etc.

En conclusión, embarcarse en un desarrollo de este tipo puede resultar duro y amenazador, pero con la constancia y los recursos necesarios es posible obtener un resultado más que decente y aprender muchas cosas durante el proceso. Es un camino muy bonito sobre todo para los que amamos esta industria y animo a todo el mundo interesado a recorrerlo.

5 BIBLIOGRAFÍA Y FUENTES

La bibliografía consultada para aplicar algunas de las metodologías utilizadas durante el proyecto son:

- Murray, J. W. (2014). *C# Game Programming Cookbook for Unity 3DComputer.* Boca Raton London New York: CRR Press.
- Sinicki, A. (2017). *Learn Unity for Android.* Guildford, Surrey, United Kingdom: Apress.
- Material teórico de la asignatura Desarrollo de Videojuegos, J. Roberto Jiménez
- Material teórico de la asignatura Fundamentos de Ingeniería del Software, L. Alfonso Ureña López, J. Ignacio Gómez Espínola

También ha supuesto numerosa información otras fuentes como son:

- Documentación oficial de Unity (unity3d.com)
- Comunidad y foros de Unity (Community Unity)
- Wikipedia (Wikipedia)

- Canales de YouTube dedicados a desarrollo de videojuegos (GSL Programación , Don Pachi , Drosgame , Padre Gamer)

6 DEFINICIONES Y ABREVIATURAS

Anti Aliasing: filtro de postprocesado que se realiza después de generar la imagen y que se encarga de suavizar los dientes de sierra que se puedan generar.

Asset: Elemento o recurso que forma parte de un videojuego.

Canvas: Elemento de un videojuego que permite generar gráficos en 2D y superponerlos en la pantalla de nuestro juego.

Feedback: Acción de ofrecer información a alguien sobre un resultado.

Hardware: Partes físicas o tangibles de un sistema informático.

Joystick: Es un periférico de entrada que consiste en una palanca que gira sobre una base e informa su ángulo o dirección al dispositivo que está controlando.

Jugabilidad: Capacidad que tiene un juego, y especialmente un videojuego, para entretener a los diferentes jugadores ofreciendo opciones interesantes y atractivas.

Librería: Es un archivo o conjunto de archivos que se utilizan para facilitar la programación.

Mecánica: Cualquier acción realizada por el jugador que modifique el estado del juego, es decir, la posición y características concretas de todos los objetos y entornos de un juego en un momento preciso en el tiempo.

PEGI: Pan European Game Information (en español: información europea sobre videojuegos) o PEGI es un sistema de clasificación europeo del contenido de los videojuegos y otro tipo de software de entretenimiento.

Pixel: La menor unidad homogénea en color que forma parte de una imagen digital.

pixelArt: Es una forma de arte digital, creada a través de una computadora mediante el uso de programas de edición de gráficos rasterizados, donde las imágenes son editadas al nivel del píxel.

Prefab: objeto reutilizable, y creado con una serie de características dentro de la vista proyecto, que será instanciado en el videojuego cada vez que se estime oportuno y tantas veces como sea necesario.

Render: Imagen digital que se crea a partir de un modelo o escenario 3D realizado en algún programa de computadora especializado, cuyo objetivo es dar una apariencia realista desde cualquier perspectiva del modelo.

Remake: Nueva versión de un videojuego estrenado hace un tiempo y que se crea completamente desde cero con un nuevo motor gráfico para adaptarse a las plataformas actuales.

Remaster: Tomar un videojuego antiguo y modificar sus gráficos para mejorar su aspecto visual, por eso es una "remasterización". Los remasters pueden incluir nuevas texturas y algún efecto nuevo, pero nunca se les cambian cosas en la jugabilidad o mecánica de juego.

Rigidbody: Componente que se añade a los objetos para indicar que son cuerpos rígidos y como tales, deben verse afectados por las físicas.

Script: Documento que contiene instrucciones, escritas en códigos de programación.

Software: Soporte lógico de un sistema informático, que comprende el conjunto de los componentes lógicos necesarios que hace posible la realización de tareas específicas.

Sprite: mapa de bits dibujado en la pantalla de ordenador por hardware gráfico especializado sin cálculos adicionales de la CPU.

SpriteSheet: Conjunto de Sprites en una sola imagen lo que mejora el rendimiento y reduce la memoria utilizada.