UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

ESCOLA TÈCNICA SUPERIOR D'ENGINYERIA INFORMÀTICA
UNIVERISTAT POLITÈCNICA DE VALÈNCIA

Extensión de Unity para la creación de HUDs modulares

Trabajo Fin de Grado
Grado en Ingeniería Informática

Autor: Armando Miguel Díaz Bruñó
Tutor: Javier Lluch Crespo
2014 - 2015

Resumen

En este proyecto se crea una extensión o *plugin* para el motor de videojuegos *Unity3D* que facilita y acelera la creación de interfaces en la partida (comúnmente llamadas *HUDs* o *heads-up displays*), además de una serie de elementos listos para ser añadidos a ellas. Se implementa también una librería de clases que provee de funcionalidad al conjunto, acompañada de una *API* que permite su utilización de forma directa y sencilla. Adicionalmente, se desarrolla un juego completo con el que mostrar el uso de la extensión.

Palabras clave: extensión, *plugin*, *Unity*, interfaces, *HUD*, videojuego

Resum

Al present projecte es crea una extensió o *plugin* per al motor de videojocs *Unity3D* que facilita i accelera la creació d'interfícies dins la partida (comunament anomenades *HUDs* o *heads-up displays*), a més d'una sèrie d'elements preparats per ser afegits a aquestes. S'implementa també una llibreria de classes que proveeix de funcionalitat al conjunt, acompanyada d'una *API* que permet la seua utilització de manera directa i senzilla. Addicionalment, es desenvolupa un joc complet amb el qual mostrar l'ús de l'extensió.

Paraules clau: extensió, *plugin*, *Unity*, interfícies, *HUD*, videojoc

Abstract

In this project, an extension or *plugin* for the videogame engine *Unity3D* that eases and speeds the creation of interfaces inside the game (commonly called *HUDs* or *heads-up displays*) is created, besides a number of elements ready to be added to them. Moreover, a library of classes that provide funcionality to the whole set is implemented, along with an *API* that allows its utilization in a direct and simple way. Additionally, a complete game with which show the use of the extension is developed.

Keywords: extension, *plugin*, *Unity*, interfaces, *HUD*, videogame

Agradecimientos

Al equipo de WildFrame Media, porque son incapaces de no ayudar.

A mi tutor, Javier Lluch, porque él sí me ha dado cuartelillo.

A Aitor, por ofrecerse a crear el excelente sonido del juego.

A Meritxell, por acompañarme en tantas tardes y noches de trabajo.

ÍNDICE DE CONTENIDOS

Índice de imágenes

1. Introducción

Rara es la vez que jugamos a un videojuego y no observamos ningún tipo de sistema de interfaces en la partida. Quizá en algunas aventuras gráficas no sea necesario presentar información alguna al jugador y, por tanto, tampoco lo sean las interfaces, pero en la gran mayoría de títulos se requiere mantener al jugador informado acerca de diversos aspectos. Suele tratarse de retroalimentación acerca de su actuación en la partida o datos referentes al estado de la misma. Tal es la función que suele cumplir una *heads-up display* en un videojuego, también conocidas simplemente como *HUD*: presentar información relevante mientras se juega.

Este proyecto tratará de aportar a la comunidad de desarrolladores de *Unity* una herramienta que facilite la creación de *HUDs* personalizadas y modulares de forma sencilla, con funcionalidad completa ya implementada y lista para ser utilizada.

A lo largo de este documento, daremos un repaso al camino que han seguido los sistemas de interfaces en la partida a lo largo de la historia y cómo han evolucionado con las incesantes innovaciones en el mundo del entretenimiento digital, para después realizar una comparativa entre las opciones en materia de motores gráficos de las que un desarrollador dispone para crear su videojuego, prestando especial atención a las posibilidades que ofrece cada una acerca del uso de las interfaces.

Después de una breve presentación de las herramientas que van a utilizarse durante el desarrollo del proyecto y sus características principales, se hará un detallado recorrido por los aspectos más importantes del desarrollo con el motor gráfico seleccionado para trabajar en este proyecto, *Unity3D*, tanto en cuestión de programación y ejecución como en la forma de trabajar con los niveles y objetos de un videojuego en él. Además, se repasará el uso de su entorno de desarrollo, el *UnityEditor*, y sus ventanas, para terminar con un vistazo a la trayectoria que han llevado los distintos sistemas de interfaces que han estado presentes en la historia de *Unity* y, finalmente, seleccionar uno de ellos para

basar en él la herramienta desarrollada, objeto de este proyecto.

Seguidamente, se presentará el proceso de diseño de la extensión o *plugin* del *Editor* de *Unity* que será ofrecida al público, detallando cómo serán las nuevas ventanas de las que dispondrán los desarrolladores y cuáles serán sus funciones, así como la estructura modular y funcionamiento del sistema de *HUD* que se administrará con ellas y los elementos que lo compondrán.

Al proceso de diseño le seguirá la implementación, y en este documento se dará una visión general de, por un lado, el *scripting* para el *Editor* con el fin de crear las nuevas ventanas y, por otro, del desarrollo de las distintas entidades de las que se componen las *HUDs*, tanto a nivel de trabajo en *Unity3D* como de programación orientada a objetos con el lenguaje *C#*.

El documento finalizará relatando todo el proceso de creación de un videojuego original desde su diseño conceptual hasta la implementación de sus diferentes partes, con el fin de demostrar el uso de la extensión añadiéndole una *HUD* completa y utilizando la librería de llamadas que traerá consigo para dotarla de funcionamiento.

2. Objetivos

En el presente proyecto se trabaja en tres ejes distintos con el motor de videojuegos *Unity3D*, todos ellos relacionados con el uso del sistema interfaces de usuario en él. Más específicamente, centra su atención en las llamadas interfaces en la partida o *HUDs* (*heads-up displays*).

La misión del proyecto es proporcionar a la comunidad de desarrolladores una herramienta que simplifique, facilite y acelere la creación de *HUDs*, comprimida en un archivo *unitypackage* que cualquiera pueda importar a su proyecto y hacer uso de él en el editor de *Unity*.

Veamos a continuación una breve presentación de cada uno de los ámbitos de trabajo que abarcará el proyecto.

2.1. Extensión del *Unity Editor*

En un primer estadio del trabajo que comporta este proyecto, se diseñarán unas nuevas ventanas para el editor del entorno de desarrollo en *Unity*, conocido como *Unity Editor*, que permita crear sencillos y versátiles sistemas de *HUD*. Esta extensión del editor será <<la herramienta>> propiamente dicha. Se implementarán mediante *scripting* en el lenguaje de programación *C#*.

Se pretenderá que, gracias a estas ventanas, un desarrollador pueda distribuir de forma automatizada en la pantalla de su juego diversos elementos de interfaz destinados a mostrar información relevante durante la partida con unos pocos clics de ratón y acciones de arrastrar y soltar. De este modo, el tiempo que normalmente tomaría crear y ajustar cada uno de los objetos que componen cada elemento y su posterior posicionamiento se verá reducido en gran medida.

2.2. Controlador y elementos de interfaz

Paralelamente al desarrollo explicado en el apartado anterior, se diseñará una serie de elementos de interfaz personalizables y preparados para ser añadidos a una *HUD* con solo pulsar un botón, así como un controlador general para ella. Este bloque de trabajo está íntimamente relacionado con el anterior, por lo que el

avance en ambos también lo estará.

El desarrollo de este apartado implicará, por un lado, la implementación de una serie de clases en *scripts* de *C#* que contendrán toda la lógica de comportamiento de los objetos de interfaz con los que se trabajará en todo el proyecto. Cabe dar especial importancia a la funcionalidad del objeto controlador de la *HUD*, que concentrará todas las llamadas que el programador necesite para utilizarla. Será este el encargado de redirigir las llamadas a los elementos individuales para que realicen la función que deban.

Por otro lado, en este apartado se creará una serie de *prefabs* (de <<prefabricado>>, es el nombre dado por *Unity* a objetos de juego ya creados y organizados guardados en disco), que serán los que constituyan los elementos de interfaz que nuestra herramienta cargará, modificará acordemente a las elecciones del desarrollador en la herramienta, y añadirá a la *HUD*.

2.3. Desarrollo de un juego

El tercer y último gran bloque del que se compone el proyecto es la creación de un videojuego original en *Unity* que sirva para demostrar la facilidad y rapidez de uso de la herramienta de creación de *HUDs* personalizables.

Para este ello, se propone el desarrollo desde cero de un colorido juego de disparos poligonal. Tomando como influencia el clásico *Space Invaders* de la japonesa *Taito*, el jugador debe moverse y disparar para derrotar hordas de enemigos que se aproximan a él, buscando aportar una sensación de juego más fresca ubicando al jugador en un espacio tridimensional muy limitado y rodeado de amenazas.

Al juego se le añadirá una *HUD* completa mediante el uso de la extensión desarrollada y se llamará a las funciones pertinentes de la librería para darle el uso deseado.

3. Estado del arte

Los sistemas de interfaces en la partida dieron sus primeros pasos a la vez que los videojuegos dieran los suyos. Desde el albor de esta forma de entretenimiento hasta nuestros días, ha existido la necesidad de mantener al jugador informado acerca de aspectos variados relativos a su desempeño en el juego, como puedan ser la puntuación obtenida hasta el momento, el tiempo restante para superar un nivel, la cantidad de vitalidad de un personaje o el porcentaje actual de completitud de determinados objetivos. De no proveerse al jugador con estos datos, este se encontraría notablemente falto de retroalimentación y más que probablemente desamparado al desconocer si las acciones que están tomando lugar están afectándole positiva o negativamente, a la vez que no sabría si realmente está haciendo algún progreso en su partida.

En respuesta a esta necesidad tomaron forma los llamados *Heads-up display*, conmúnmente abreviados como *HUD*. Con el paso de los años se han ido modernizando y adaptando a cada vez más tipos de información que representar, pero en esencia su propósito se ha mantenido invariante: mantener siempre en pantalla un conjunto de datos.

Nos podemos remontar a la época del nacimiento de la compañía desarrolladora de ordeadores y videojuegos *Atari, Inc.* para contemplar populares ejemplos primerizos de este tipo de sistemas de interfaces. Su clásico simulador de tenis *Pong* [Imagen 2] mostraba la puntuación de cada jugador sobre el terreno de juego y una función similar cumplían las *HUDs* del juego *Computer Space* [Imagen 1], creado anteriormente por los fundadores de la empresa.

Imagen 1: Máquina con Computer Space

Imagen 2: Máquina con el juego Pong

Con el paso del tiempo, el mundo del videojuego ha evolucionado enormemente, con mejoras y avances cada vez más revolucionaros en el área de

los gráficos por computador, que aportan más realismo y variedad a los jugadores con cada año que pasa. Tales cambios aparecen debido al desarrollo de los llamados motores gráficos, herramientas consistentes en conjuntos de programas de renderizado de gráficos en pantalla, modelos de cómputo físico y abanicos de librerías y funcionalidades diversas que permiten la creación de videojuegos completos en sus entornos de desarrollo; además de la innovación en materia de *hardware,* cada vez más eficiente, potente y accesible. ¿Cómo han, pues, evolucionado hasta el presente los sistemas de interfaces?

En nuestros días, podemos disfrutar de una gran cantidad de títulos que poseen elaboradas y flamantes *HUDs*, con elementos tan variados como barras de salud, medidores de energía, contadores de balas en el cargador de un arma, marcadores de puntos, radares o mapas de la zona donde transcurre la acción y un largo etcétera dependiendo de las necesidades del juego concreto. Tales elementos pueden tener infinidad de formas, acabados y colores según la inventiva del artista tras cada trabajo, llegando a incluso a en algunos casos ser tridimensionales e incluir efectos tales como una oscilación debida al caminar del jugador, o sacudidas cuando se recibe un golpe.

Como ejemplo ilustrativo, mencionaremos el sistema de interfaces del título de 2002 *Metroid Prime* [Imagen 3], de la desarrolladora *Retro Studios* y distribuido

Imagen 3: La HUD tridimensional de Metroid Prime

por *Nintendo Company Limited*. En ella podemos apreciar una *HUD* tridimensional con al menos ocho elementos diferentes de interfaz, siendo algunos de ellos bidimensionales pero ubicados en el mismo espacio, simulando estar colocadas en el visor del casco. No olvidando nunca la importancia que tiene la retroalimentación, cabe destacar además que utilizarla produce una grata experiencia al jugador ya que, además de estar sus elementos animados acordemente a la acción, toda la interfaz se tinta de color, oscila y se mueve dependiendo de qué sufra el jugador, séase un impacto, frío extremo o calor intenso.

Pero, ¿cuáles son las posibilidades que tiene una persona o equipo que desea desarrollar su juego? Y, en relación con el presente proyecto, ¿de qué facilidades dispone a la hora de implementar un sistema de interfaces como los que hemos comentado? El desarrollador —o equipo de desarrollo— deberá decidir en qué motor de videojuegos basar su proyecto, que podrá ser uno diseñado e implementado por él mismo, o bien alguno de los motores comerciales existentes en el mercado.

En la mayoría de casos, el coste del desarrollo y del control de calidad de la primera opción en cuestión de tiempo y dinero invertido resulta inasumible para equipos pequeños o que cuenten con un presupuesto limitado, pero a menudo es la opción elegida por estudios consagrados. Ejemplos de ello son *Frostbite* del estudio *EA Digital Illusions CE*, *Unreal Engine* de *Epic Games, Inc.*, el motor propietario usado por *Frozenbyte Inc.* para el desarrollo de sus juegos, el aclamado *Source* de la compañía *Valve* o la enorme cantidad de motores distintos que *Nintendo Company Limited* ha creado para un título o serie de títulos en concreto.

Por otro lado, existe un abanico de opciones muy asequibles, incluso gratuitas, en lo que a motores gráficos comerciales se refiere. Nos centraremos en estos últimos debido a que son la elección más común para todo aquel que desee dar vida a un juego que haya ideado dadas su facilidad y rapidez de uso, ya que traen consigo numerosas herramientas y utilidades para la creación del juego en su entorno de desarrollo y, por norma general, cuentan con una gran

cantidad de extensiones o *plugins* desarrollados por terceros que uno puede adquirir y utilizar para mejorar su experiencia de uso de las herramientas, o bien añadir funcionalidad ya diseñada a su juego.

Haremos a continuación un breve análisis comparativo de los motores gráficos comerciales disponibles al público más populares, prestando atención a las facilidades que proporcionan para el desarrollo de sistemas de interfaces en la partida, para después comentar más detalladamente la situación de las interfaces en *Unity3D*, en el motor utilizado en este proyecto.

3.1. Motores gráficos populares

Cuando nos lanzamos al desarrollo de un juego, son diversas las posibilidades que se nos presentan en el mercado para seleccionar un motor gráfico para nuestro propósito. Destacaremos entre ellos los que responden al nombre de *Unreal Engine*, *Unity3D* y *CryEngine*, por ser las opciones más elegidas hoy en día para la creación de un videojuego tridimensional.

3.2.1. *Unreal Engine*

Creado por la empresa *Epic Games, Inc.*, conocida en aquel entonces como *Epic MegaGames*, para su exitoso título de 1998 *Unreal*. A día de hoy se encuentra en su cuarta versión, y desde prácticamente el principio fue muy bien recibido por la comunidad de jugadores y desarrolladores. No solo exhibía un rico abanico de mejoras en materia de estructura, renderizado y rendimiento, sino que además permitía la alteración de los juegos hechos con este motor gracias a la inclusión de un lenguaje de *scripting*, hecho que atrajo la atención de los llamados *modders*.

Actualmente, este motor es más accesible que nunca, pues desde marzo de 2015 *Unreal Engine 4* (abreviado como *UE4*) y su código fuente están abiertos a todos los usuarios [1]. Presenta aspectos de lo más interesantes como algoritmos de iluminación global altamente eficientes y siempre en proceso de mejora, módulos para el manejo de *shaders* con multitud de opciones o la programación mediante *blueprints*, que permite la definir visualmente comportamientos y rutinas mediante sencillos diagramas de flujo para aquellos

que no necesiten el absoluto control del proceso. En caso contrario se dispone de programación en el lenguaje *C++* para un manejo preciso de la aplicación.

Estas características resultan en una calidad gráfica y fluidez de ejecución sobresaliente, además de en una notable sencillez de uso, propiciando así que *UE4*, además de ser el preferido por muchos para el desarrollo de videojuegos [2], sea la opción elegida para proyectos de renderizado en tiempo real de calidad fotorrealista, como puedan ser muestras interactivas de carácter arquitectónico.

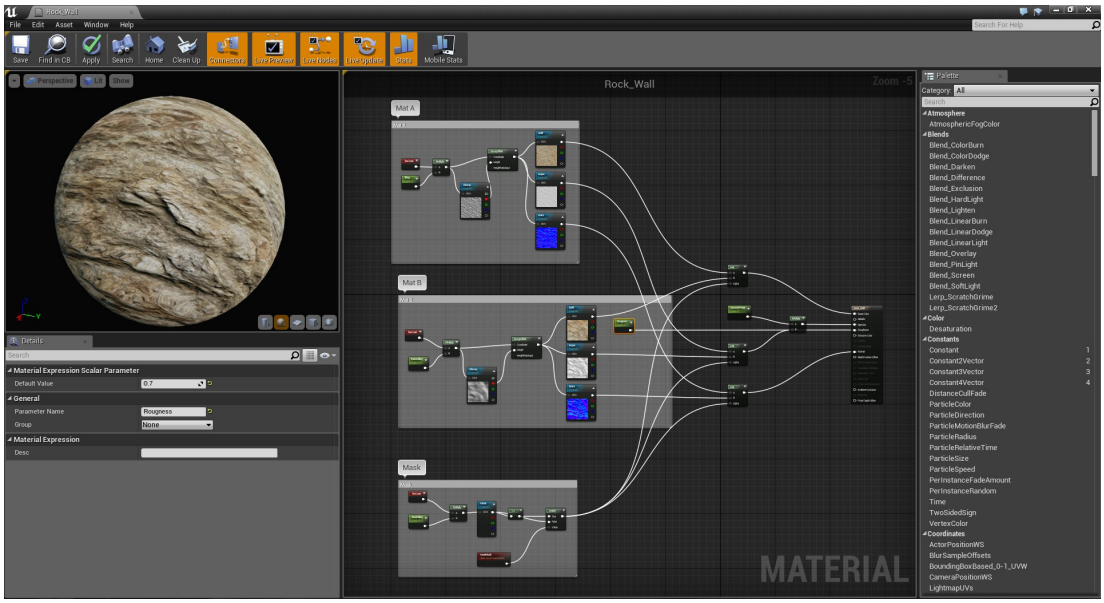

Imagen 4: Edición de blueprints en el editor de Unreal Engine 4

El campo de las interfaces no ha sido uno de los puntos fuertes de *Unreal Engine*, sin embargo. Hasta su tercera versión, para que las interfaces tuvieran un aspecto mínimamente moderno, animado y actual; los desarrolladores normalmente tenían que echar mano de la tecnología *Scaleform Gfx* [3], actualmente en posesión de la compañía de software *Autodesk, Inc.*, para integrar en sus juegos interfaces basadas en la tecnología *Adobe Flash*, de *Adobe Systems*, tanto de propósito general como *HUDs*. Estas debían ser previamente construidas usando algún *software* de edición *Flash* externo, por lo que podemos afirmar que no estaban integradas dentro del entorno de desarrollo de *Unreal Engine*.

El cambio en este ámbito llegó con la llegada de *UE4* en el año 2014, a partir del cual los desarrolladores pudieron hacer uso del llamado *Unreal Motion Graphics UI Designer*, una herramienta para la construcción de interfaces, ahora

sí integrada en el motor gráfico. Se trata de una *suite* completa de elementos básicos de interfaz personalizables, por lo que si deseamos crear nuestra *HUD* con ella tendremos que hacerlo desde cero, como es normal.

3.1.2. *UNITY3D*

Veamos a continuación qué nos ofrece la alternativa de la desarrolladora *Unity Technologies,* lanzada en el año 2005. Desde aquel entonces, la compañía ha tratado de mantener la portabilidad entre plataformas de destino como piedra angular de su motor.

Tanto es así, que el desarrollador de un juego en *Unity* no tiene más que pulsar un botón para cambiar entre las plataformas para las cuales compilar su proyecto. El motor reimportará automáticamente los recursos (imágenes, audio, modelos tridimensionales, etc.) del juego con el formato adecuado para la plataforma objetivo y procederá con la compilación, que se hará sobre la *API* de gráficos utilizada en la plataforma seleccionada, como puedan ser *OpenGL*, *DirectX* o *Metal.* Es este, por tanto, un factor muy a tener en cuenta por aquellos que deseen hacer llegar su producto al máximo número de plataformas posibles.

También lo es el hecho que desde que apareciese su segunda versión en el año 2007, el motor ha tenido una licencia gratuita y disponible para todos además de la versión profesional de pago, aumentando notablemente su accesibilidad y, en consecuencia, su presencia en el mercado [4].

Precisamente por estas razones, *Unity3D* es actualmente el motor elegido por una enorme cantidad de desarrolladores de juegos para móviles, tanto de dos como de tres dimensiones, ya que normalmente buscan desarrollar su juego en poco tiempo y lanzarlo en las tiendas de aplicaciones para móviles como *Google Play Store, Apple App Store* o *Amazon Appstore*, sin perder tiempo en el cambio de plataforma ni en desarrollar elementos específicos de cada una.

Cabe destacar que, contrariamente a los ofrecidos por motores de la compentencia, el paradigma de desarrollo en *Unity* suele ser preferido por programadores ya experimentados en programación orientada a objetos puesto que utiliza *scripts* en lenguajes como *C#* o *Boo* para definir el comportamiento de

los objetos del juego, y luego esas piezas de código son alterables o utilizables desde el editor del entorno de desarrollo, conocido como *Unity Editor* [Imagen 5], además de poderse reutilizar para cualquier cantidad de objetos.

Por otro lado, siempre ha sido comúnmente aceptado que este motor nunca ha gozado del realismo y calidad a nivel gráfico del que sí gozan sus competidores. No fue hasta la llegada de la quinta y actual versión, *Unity 5*, que se introdujesen las mejoras que acortarían la distancia entre ellos **[5]**. Tecnologías punteras como la iluminación global en tiempo real fueron añadidas además de mejoras en la iluminación dinámica de entornos y de largo alcance, en los sistemas de audio y en el manejo de *shaders*, entre otras.

Imagen 5: Editor de Unity 5 Pro

En contraposición, *Unity3D* ofrece mejores posibilidades para el desarrollo de juegos bidimensionales que *Unreal Engine* o *CryEngine*, compitiendo en el sector con motores como *Cocos2D* o *GameMaker: Studio*.

En materia de interfaces, como detallaremos más adelante, los usuarios de *Unity* han tenido que valerse de técnicas arcaicas para el diseño de interfaces como el pintado directo de texturas en pantalla, práctica nada flexible, o recurrir a *plugins* de interfaz de usuario de terceros si querían tener sistemas elaborados.

El motor estrenó en 2014 un sistema propio, en el cual centramos este proyecto. Profundizaremos en el tema después de este análisis comparativo porque resulta importante conocer el camino que han seguido las interfaces en

Unity, dado que es el motor con el que nos disponemos a trabajar. Esta elección ha sido tomada debido a la experiencia obtenida por trabajar con *Unity3D* en una empresa de desarrollo de videojuegos durante un año.

3.1.3. *CryEngine*

Presentamos como tercera alternativa el motor gráfico desarrollado por la alemana *Crytek*, usado por primera para la creación del videojuego *Far Cry* en el año 2004. Actualmente está disponible por tarifas desde 9,99 dólares al mes.

Ha demostrado poseer desde sus inicios calidad puntera en materia de potencia y rendimiento gráficos, produciéndose con él videojuegos en los que destaca su atractivo visual [6]. Ofrece tecnologías que permiten trabajar con extensísimos terrenos y con grandes cantidades de elementos en ellos, además de disponer de excelentes técnicas de iluminación y sombreado y un motor físico que es capaz de dotar a elementos del entorno de un realismo más que notable. El lenguaje de programación utilizado para desarrollar en *CryEngine* es *LUA*.

Su editor *Sandbox* [Imagen 6] incluye herramientas de lo más variadas tanto para la creación de elementos como para el refinamiento visual del juego. Creadores de ríos, caminos, vehículos o editores de animaciones faciales son ejemplos de ello. Este hecho lo sitúa en una posición ventajosa frente a otros motores como *Unity*, que sí incluye algunos elementos de terreno por defecto

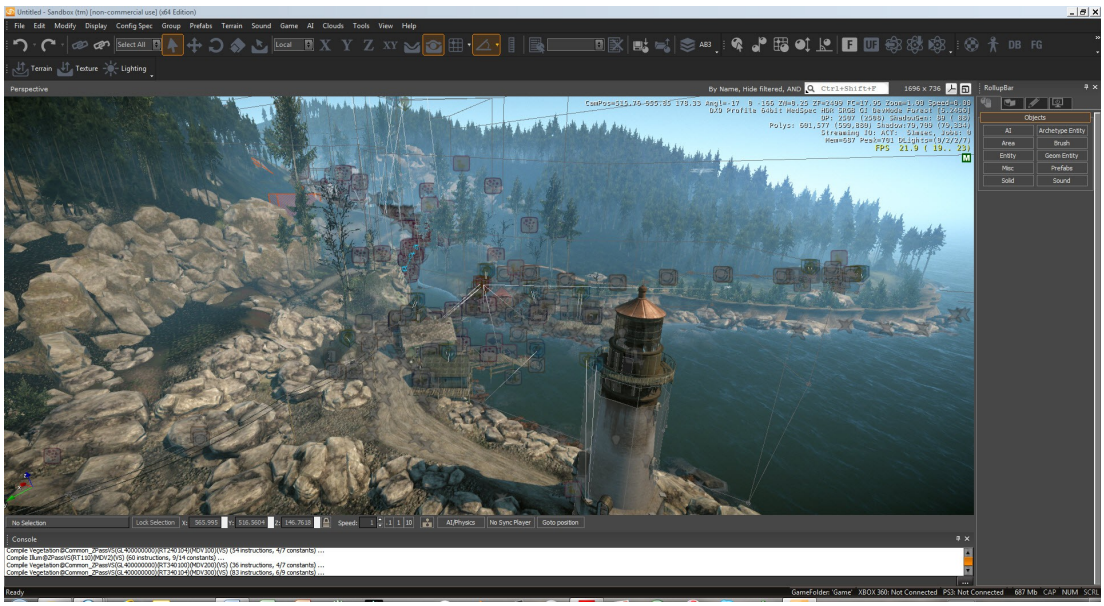

Imagen 6: Editor Sandbox de CryEngine 3

pero carece de potentes herramientas de este tipo integradas en el motor, por lo que normalmente se recurre al uso de *software* externo.

En materia de interfaces, no obstante, se encuentra en la misma situación en la cual se encontraba *Unreal Engine*. Se recurre al uso del anteriormente mencionado *Scaleform* para integrar interfaces dentro del juego [7], por lo que deben ser creadas externamente en programas de edición *Flash*. El motor no destaca, por tanto, en este aspecto al no incorporar un sistema de interfaces propio ni una herramienta para trabajar con él.

En resumidas cuentas, podemos afirmar que *CryEngine* es una opción muy recomendable para aquellos cuyo objetivo sea obtener unos resultados visuales detallados e impresionantes en su juego, pese a que el hecho de no ser gratuito suponga un considerable impedimento para según qué bolsillos; pero no es una elección adecuada para trabajar con interfaces en la partida, dado que no dispone de un sistema propio. Se recomendaría, dado el caso, del estudio del funcionamiento de *Scaleform*.

4. Herramientas utilizadas

Presentaremos a continuación una breve relación del *software* utilizado para trabajar en este proyecto, sin entrar en más detalle que el uso principal que le ha sido dado a cada programa. Algunos, como las herramientas de edición de imagen, han tenido un uso puntual o esporádico.

- *Unity3D*: el motor de videojuegos creado por *Unity Technologies,* elegido para el presente proyecto. Detallaremos más adelante los aspectos más relevantes de este programa.
    - *Unity Editor*: el entorno de trabajo que ofrece *Unity*, donde tiene lugar la mayor parte del desarrollo [8]. En él se han construido tanto los elementos de *HUD* objeto de este proyecto como el juego presentado.
    - *Animator*: la herramienta incorporada en el *Editor* para realizar modificaciones en los objetos de juego en tiempo real, dotándolos de animación. Se ha utilizado tanto en los efectos de los elementos de *HUD* como en los efectos visuales de los enemigos del juego.
    - *Mecanim*: un controlador de animaciones basado en una máquina de estados, que permite alternar entre clips de animación con fluidez y dinamismo. Con él se han definido los estados y transiciones de los elementos de *HUD* que pueden ocultarse y mostrarse dinámicamente.
- *MonoDevelop*: un entorno de desarrollo integrado creado por *Xamarin* y la comunidad *Mono* [9] para proyectos que usen los marcos de trabajo *Mono* y *.NET*, apuntando especialmente al desarrollo multiplataforma. Una versión adaptada de él viene incluida en la descarga de *Unity*, que es la utilizada en este proyecto para la programación en el lenguaje *C#*.
- *Sublime Text 3*: un editor de texto plano [10] versátil y lleno de funcionalidades. Se ha utilizado complementariamente a *MonoDevelop* para cuestiones puntuales.
- *Git*: la herramienta de control de versiones diseñada por Linus Torvalds,

que permite la administración distribuida de ficheros de forma eficiente y segura mediante el uso de ramas independientes [11]. Con este sistema se ha administrado el conjunto de ficheros del proyecto desde su creación.

  - *Bitbucket*: el servicio de alojamiento de repositorios *Git* o *Mercurial* en la nube de la empresa *Atlassian Software* [12]. Permite la administración remota de ficheros gratuitamente para equipos de hasta cinco personas, sin necesidad de configuración de servidores previa. El proyecto completo está alojado en él.
  - *SourceTree*: creado también por *Atlassian Software*, se trata de un cliente de escritorio para la administración de repositorios locales con comunicación con servidores remotos [13]. Se ha utilizado como interfaz gráfica para la intercción con la herramienta *Git*.

- *GIMP*: la herramienta gratuita de edición de imagen propio de *GNU* [14]. Ha sido utilizada para crear las imágenes de ayuda existentes en la extensión, así como aquellas mostradas en los diferentes elementos de *HUD* del juego.
- *ShadowPlay*: la utilidad de *Nvidia* para la grabación de pantalla por *hardware* [15], que se ha utilizado para grabar algunos vídeos de demostración del proyecto.

5. Características de *Unity3D*

Con el fin de familiarizarnos con el motor que vamos a utilizar, así como con el entorno de desarrollo asociado a él, revisaremos una serie de conceptos básicos acerca de su funcionamento, las tecnologías que incorpora y la forma en la que se trabaja con él. Después, y más en relación con los objetivos del proyecto, daremos un repaso a la trayectoria que han llevado los sistemas de interfaces en *Unity*, presentando las alternativas existentes y eligiendo una de ellas para el desarrollo del proyecto.

5.1. El motor

A partir del lanzamiento oficial de *Unity 5* en 2015, todos los usuarios pueden gozar del abanico completo de características que ofrece. Hasta entonces, las diferencias de las versiones *Free* y *Pro* del motor incluían la exclusividad de una notable cantidad de tecnologías para la versión *Pro*, tales como efectos de cámara o mejoras de renderizado. Ahora, las diferencias entre estas versiones son existentes pero no afectan en absoluto al material incluido en el motor.

Como tal, el motor gráfico *Unity3D* ofrece soluciones al usuario en materia de gráficos bidimensionales y tridimensionales, detección de colisiones, iluminación, simulación de físicas, audio, inteligencia artificial, animación, etcétera. Es decisión del desarrollador ceñirse a utilizar el conjunto de útiles que ofrece *Unity* o reemplazar el uso de ciertas partes por otras diseñadas por él mismo o por terceros.

Veamos en los siguientes apartados los preceptos más importantes que un desarrollador de *Unity* debe conocer.

5.1.1. Las escenas

Las *scenes* de *Unity* cumplen con una función básica y esencial: contienen todo el material que se vaya a utilizar durante la ejecución del juego [16]. En ellas, distribuimos en el espacio elementos tangibles como terrenos, edificios, decorados, vegetación, animales o personajes; así como fuentes de luz o sonido.

También existen en el espacio de las escenas las interfaces que creamos, pero son visualizadas por una cámara aparte, y no interfieren en el juego.

Uno debe pensar en las escenas como niveles de su videojuego. La razón para ello es que al cargarse una escena, se trae a la memoria del dispositivo todo aquello va a ser utilizado en dicha escena. Resultaría ilógico cargar los elementos de un nivel en el que el jugador no está, puesto que no van a ser utilizados y supondrían una sobrecarga innecesaria. De la misma forma, se podría crear una escena destinada únicamente a ser el menú principal del juego, desde el cual se accederá posteriormente a otras escenas.

La carga de escenas es un proceso bloqueante hasta que se completa, pero se puede hacer de forma asíncrona mientras se sigue jugando para suavizar el mal efecto que produce un parón en la ejecución. Debe planificarse correctamente cómo organizar las escenas de un juego y qué objetos deberían ir en cada una.

5.1.2. Los objetos de juego y los componentes

Todo aquello que hay en una escena de *Unity* es, por fuerza, un *GameObject*: se trata de la entidad básica de funcionamiento en el motor [17]. Pueden crearse desde código durante la ejecución llamando al constructor de la clase *GameObject*, que inmediatamente lo ubicará en la escena, y destruirlo con *Destroy*. Esto mismo puede hacerse en el *Editor*.

No obstante, un *GameObject* de por sí apenas es poco más que un contenedor vacío con nombre. Para que esta unidad tenga sentido y pueda desempeñar una función, requiere de componentes. Los componentes son pequeñas piezas de *software* que van adheridas a un *GameObject*, y que lo dotan de significado y funcionalidad [18].

Cualquier objeto de la escena puede poseer cualquier cantidad de *components*. *Unity* pone a nuestra disposición un amplio rango de componentes ya creados, más o menos personalizables, con los que dar vida a los objetos. Pueden añadirse a un *GameObject* durante la ejecución con el método *AddComponent<T>* y quitárselo con *Destroy*. Veamos una breve relación de los

más importantes:

- *Transform*: el componente básico que todo objeto de juego posee. Define su posición, rotación y escalado en el espacio [19].
    - También sirve para establecer relaciones jerárquicas de parentesco entre objetos. Un *GameObject* es hijo de otro en la medida en que su *Transform* es hijo de otro *Transform*.
- *Rigidbody*: este componente somete al *GameObject* al motor de físicas de *Unity* [20].
    - Desde el mismo momento en que se le añade a un objeto, la fuerza de la gravedad actuará sobre él y su posición y movimiento podrán ser alterados por fuerzas externas.
    - Incluye parámetros para ajustar cómo se ve el objeto afectado por tales fuerzas físicas.
- *Collider*: define el volumen o superficie en el cual el objeto puede recibir impactos de rayos o colisionar contra otros objetos [21].
    - Puede ser *trigger* en lugar de rígido, de tal forma que los objetos que se dirijan hacia él lo atravesarán sin ser golpeados, pero este disparará eventos para notificar que se ha entrado en contacto con otro cuerpo.
    - Existen *Colliders* específicos con formas determinadas cuyo uso es muy eficiente, tales como *BoxCollider*, *SphereCollider* o *CapsuleCollider*.
    - El componente *MeshCollider* convierte la malla de un modelo 3D en una superficie colisionable.
    - Para la simulación de físicas en vehículos, se utiliza *WheelCollider*.
- *MeshFilter*: permite ubicar en la posición del objeto una malla tridimensional. Puede ser una primitiva sencilla como un cubo, o puede ser un elemento complejo, como un personaje diseñado en un *software* de modelado como *Maya* o *3ds Max* de *Autodesk, Inc.*
    - El componente *MeshRenderer* permite el renderizado la malla definida

en el *MeshFilter* del mismo objeto, incluyendo ajustes sobre cómo debería colorearse o iluminarse dicha malla.

- *SpriteRenderer*: premite renderizar un *sprite* (imagen bidimensional) en el espacio.
- *Light*: convierte al *GameObject* en un emisor de luz [22]. Su uso es clave, ya que son las luces las que determinan cómo se sombrean los cuerpos y, en definitiva, cómo luce una escena.
    - Disponemos de los tipos de luz *Directional*, *Point*, *Spot* y *Area*. Incluyen una serie de parámetros para ajustar su efecto, tales como el color emitido o su intensidad.

Destacaremos finalmente el concepto de *prefab*, de gran importancia y versatilidad. Un *prefab* no es más que un *GameObject* guardado en disco. Con un procedimiento tan simple como arrastrar un objeto cualquiera de la jerarquía a una carpeta del proyecto, se crea una copia de él que podemos utilizar para replicarlo tantas veces como queramos, tanto arrastrándolo de vuelta a la escena como con el método *Instantiate*.

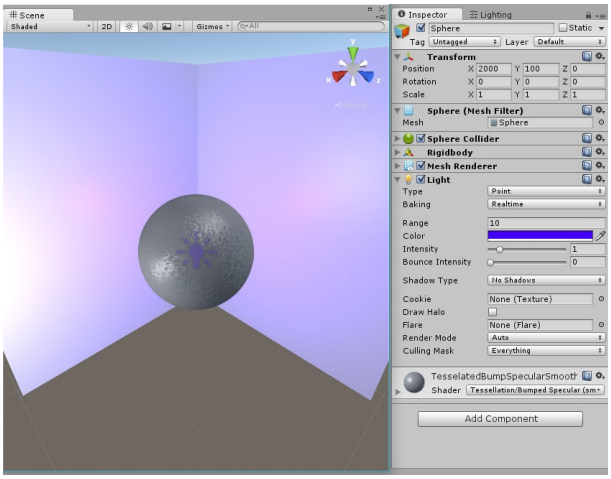

Imagen 7: GameObject con gran cantidad de componentes

5.1.2. La programación

Para dotar a los objetos de funcionalidad añadida, deberemos recurrir a la programación en *scripts*. Para *Unity* podemos desarrollar indistintamente en los lenguajes *Boo*, *C#* o *UnityScript*. Los tres son lenguajes orientados a objetos y, en consecuencia, podemos trabajar con clases sin ningún problema.

El aspecto realmente interesante de la programación en *Unity* es el hecho que cualquier clase que herede de *MonoBehaviour*, existente en el espacio de nombres de *UnityEngine*, que implementemos en un script se convierte automáticamente en un componente. A partir de que *Unity* lo compile, podremos

arrastrar ese script a un objeto para que se añada a él. Además, desde un componente adherido a un objeto podremos acceder a otros componentes del mismo objeto con el método *GetComponent<T>*.

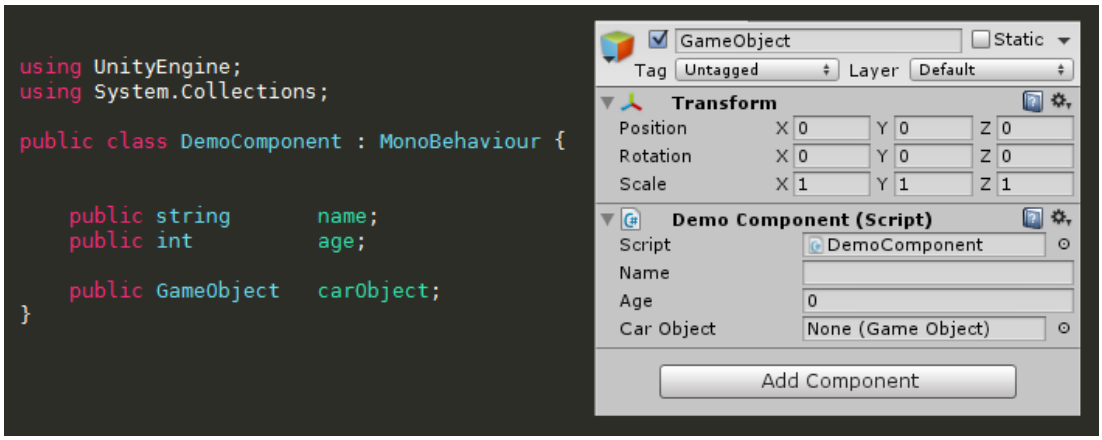

Imagen 8: Componente resultante de la implementación de un script

Desde el editor podremos visualizar y manipular todos los atributos de esa instancia que se hayan definido como públicos, incluso durante la ejecución [Imagen 8]. Este paradigma de desarrollo destaca por sus cualidades de modularidad y reusabilidad, además de constituir una forma bastante intuitiva de trabajar a posteriori con el código implementado.

5.1.2. La ejecución

Entender cuál es el flujo de ejecución de *Unity* es un factor clave para que el programador organice su código convenientemente, y así alcance los resultados que desee. Hemos mencionado la clase *MonoBehaviour* en el apartado anterior, y es en esta misma en la que debemos centrarnos. Veamos a continuación cómo interacciona el motor ella.

Unity manda en cada ciclo de ejecución una serie de mensajes predefinidos a todos los componentes (que heredan de *MonoBehaviour*) activos en la escena [Imagen 9]. Un componente está activo si tanto él como su *GameObject* lo están. Un *GameObject* desactivado puede ser manipulado externamente, pero no será visible en la escena ni realizará ninguna acción por iniciativa de ninguno de sus componentes.

Lo que deberá hacer el programador, si desea que alguno de sus componentes realice acciones en determinados instantes del ciclo será

simplemente definir en su *script* métodos con el nombre correspondiente al mensaje enviado por *Unity*, de tal forma que serán llamados llegado el momento. No ocurre absolutamente nada si no se definen, sin embargo.

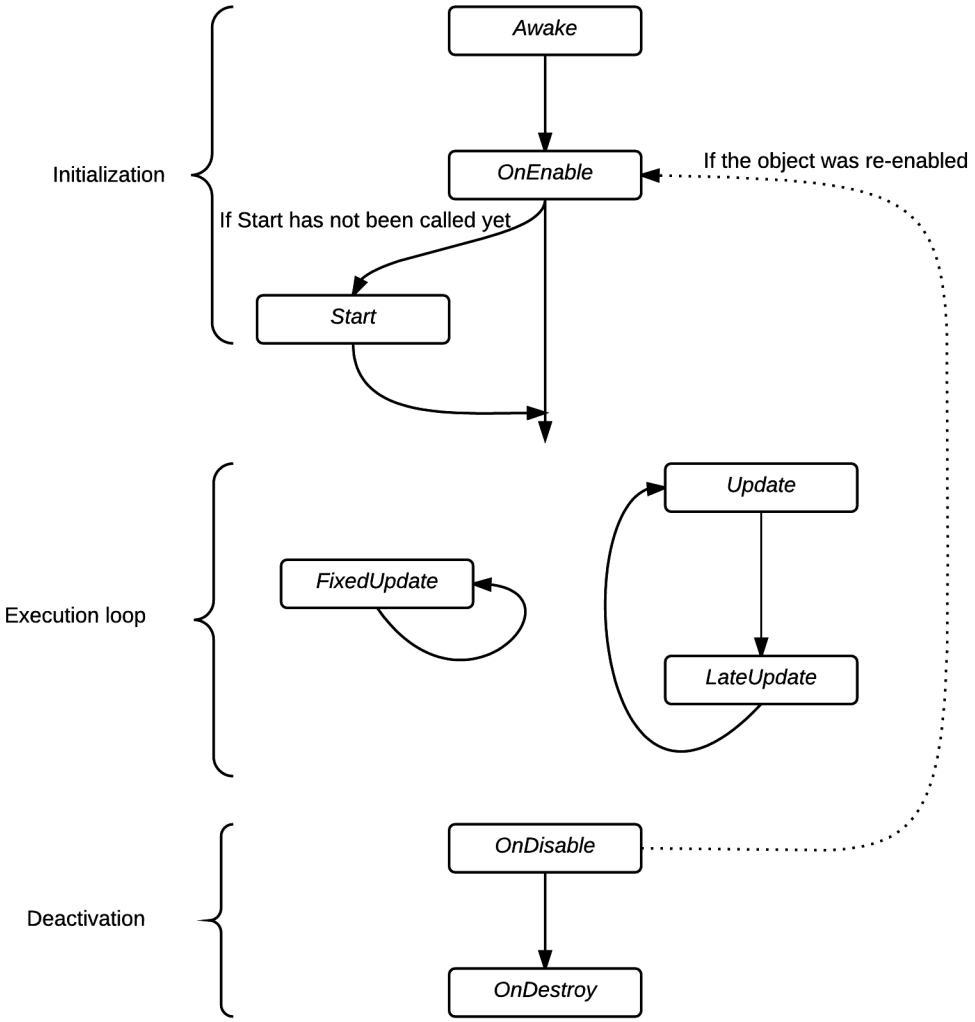

Imagen 9: Ciclo de vida básico de un MonoBehaviour

Veamos a continuación una relación de los más conocidos. Comencemos por aquellos que se ejecutan en momentos específicos del ciclo de vida de un componente:

- *Awake*: llamado únicamente cuando el objeto aparece por primera vez en la escena.
- *Start:* llamado después de *Awake*, sólo una vez y justo antes del primer fotograma en que el objeto está presente en el juego.

- *OnDestroy*: se ejecuta cuando el objeto va a ser destruido por acción del programa, por un cambio de escena o por el cierre de la aplicación.
- *OnEnable*: se llama cada vez que el componente (o el objeto) son activados en la escena. Si está activado cuando se crea el objeto, las acciones definidas en *OnEnable* suceden antes que las definidas en *Start*.
- *OnDisable*: llamado cada vez que el componente es desactivado o justo antes de *OnDestroy* si fuera a ser destruido.

Son muy importantes, también, aquellos mensajes que *Unity* envía mientras el componente está activo, ya que ofrecen la posibilidad al programador de controlar el comportamiento del objeto y sus componentes en tiempo real. Los más utilizados son:

- *Update*: este método es llamado una vez por fotograma o *frame*. Muy útil para el control de la aplicación, pero debe utilizarse con cuidado, porque introducir operaciones costosas aquí penalizarán fuertemente el rendimiento general del juego.
- *LateUpdate*: tiene el mismo comportamiento que *Update*, pero es llamado estrictamente después de los *Updates* de todos los objetos. Se recomienda su uso para realizar acciones que requieran que cálculos o modificaciones hechas en *Update* hayan terminado completamente.
- *FixedUpdate*: llamado después de cada paso del bucle de físicas del motor, y es independiente de los fotogramas por segundo a los que funcione la aplicación. Útil para acciones que deban ser independientes en el tiempo del período de los fotogramas, como pueda ser el movimiento de un vehículo.

Mencionaremos, finalmente, la existencia de una variedad de mensajes que reciben los componentes por acción de un *collider* adherido a su mismo *GameObject*. Estos son:

- *OnCollisionEnter*, *OnCollisionStay* y *OnCollisionExit*: son llamados cuando hay contacto con otro cuerpo rígido, en las diferentes etapas del contacto.

- *OnTriggerEnter*, *OnTriggerStay* y *OnTriggerExit*: similares a los anteriores, pero son llamados cuando otro *collider* atraviesa uno que está marcado como *trigger*, tal que no colisiona con nada pero manda estos mensajes cuando hay interacción.

Existen otros mensajes de los cuales se puede hacer uso relacionados con, por ejemplo, los pasos de renderizado, pero con los detallados anteriormente es más que suficiente para tener un control eficaz del juego que se desarrolle.

5.2. El editor

En el llamado *Unity Editor* es donde tiene lugar la mayor parte de la creación de un juego en *Unity3D*. En él es donde se construyen las diferentes escenas que formarán parte de él y se administran aspectos varios de la ejecución. Incorpora además un compilador y un entorno de ejecución para los lenguajes de programación compatibles. Cada una de las diferentes ventanas existentes en el editor desempeña una función distinta y todas ellas pueden desplazarse por la pantalla o cambiarse de ubicación.

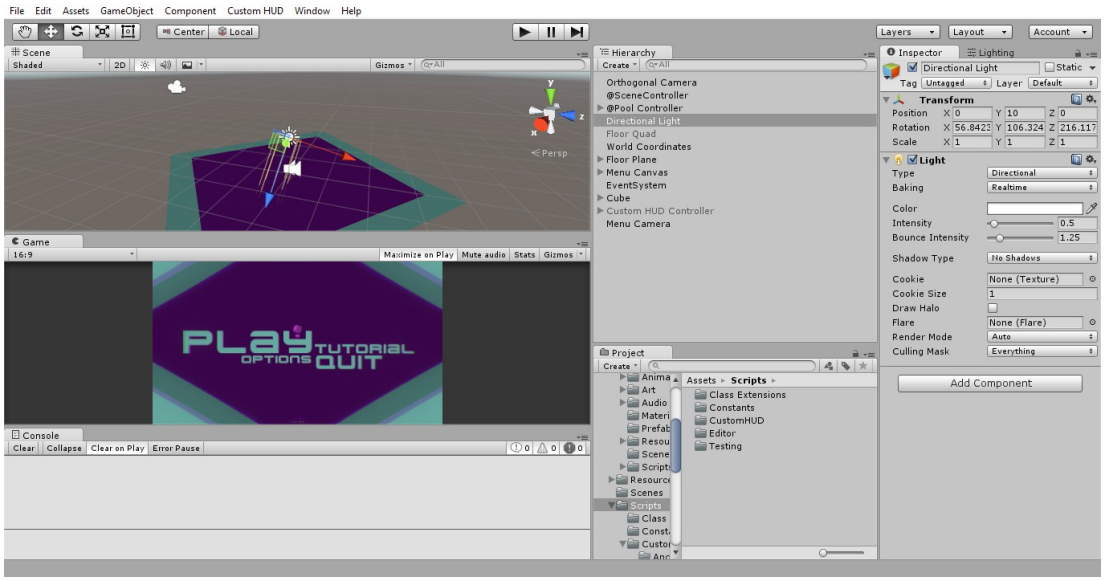

Imagen 10: Vista de las ventanas básicas del editor

A continuación veremos un conjunto de ventanas y herramientas clave para el uso del entorno de desarrollo.

5.2.1. Ventanas básicas

Cuando instalamos *Unity* y abrimos por primera vez el editor, se nos presenta

una serie de ventanas listas para ser utilizadas. Tales ventanas son las mínimas necesarias para desarrollar con el programa, y son:

- *Scene*: es una representación tridimensional interactiva de la escena de juego actual.
    - En ella podemos manipular a nuestro antojo todos los objetos de juego.
    - Aparecen elementos meramente indicativos llamados *gizmos* que señalan la posición de distintos elementos, como cámaras o luces.
- *Game*: representa el resultado del renderizado de las cámaras ubicadas en la escena.
    - Muestra cómo luce el juego en cada momento.
- *Hierarchy*: contiene una vista de todos los *GameObjects* en la escena, agrupados por relaciones de parentesco.
    - Permite copiar, pegar y reorganizar objetos.
    - Permite crear *prefabs* arrastrando objetos a una carpeta del proyecto.
- *Inspector*: muestra todos los componentes del *GameObject* seleccionado y permite manipularlos, así como el nombre del objeto o si está activo.
- *Project*: contiene toda la estructura de carpetas que conforman el proyecto y los llamados *assets*, que es todo aquel material disponible en el proyecto para su uso: *prefabs*, imágenes, scripts, modelos 3D, sonidos...
    - Premite pasar la referencia de un *asset* a una variable de un componente simplemente arrastrándola hasta el hueco pertinente del inspector.
    - Permite instanciar copias de un *prefab* simplemente con arrastrarlos hasta la vista de escena o la jerarquía.
- *Console*: muestra los mensajes del compilador y los generados por los scripts.

5.2.2. Utilidades avanzadas

Si bien es cierto que *Unity* no ofrece tanta cantidad de herramientas para el refinamiento del juego como sus competidores, sí ofrece unas cuantas dignas de mención cuyo uso puede aprenderse en poco tiempo, y pasan a ser imprescindibles en estadios más avanzados del desarrollo. Cada una posee su ventana de editor asociada. Las más destacables son las siguientes:

- *Animation*: permite crear clips de animación de forma intuitiva, consistentes en la alteración de parámetros públicos de los componentes de uno o varios *GameObjects* a lo largo del tiempo.
- *Animator*: mediante el uso de una máquina de estados, permite transicionar entre las animaciones de un objeto.
    - Si el objeto tiene estructura humanoide, permite controlar cómo sus movimientos se alternan o mezclan.
- *Lighting*: ofrece una serie de ajustes acerca de la iluminación global de la escena [23]. Incluye, además, configuraciones específicas para objetos, así como otros ajustes como la niebla.
- Audio Mixer: una ventana con la que agrupar fuentes de sonido, mezclarlas y aplicarles efectos [24].
- *Navigation*: permite la creación de una malla de navegación, que define por qué áreas pueden moverse los agentes controlados por inteligencia artificial, y ajustar sus parámetros.
- *Profiler*: monitoriza en tiempo real el consumo de recursos que está produciendo el juego en desarrollo, con métricas de muchísimas índoles.

Cabe recordar que los desarrolladores disponen de un creciente abanico de utilidades desarrolladas por terceros, que pueden encontrarse en la web o en la tienda *Unity Asset Store*. Muchas de ellas aportarán nuevas ventanas al editor.

La extensión del editor que se desarrolla en este proyecto definirá nuevas ventanas que estarán disponibles para su uso desde el instante en que se añada el *plugin* al proyecto.

5.3. Sistemas de interfaces

Si hace algunos años uno tratara de crear un juego en *Unity3D* sin ayuda de herramientas externas, se toparía con dificultades que entorpecerían y ralentizarían el desarrollo de sistemas de interfaces en versiones antiguas del motor. Si bien es cierto que sí ha existido soporte pare ello, su uso no era todo lo práctico que cabría desear. Demos a continuación un repaso a las más notables opciones disponibles para la construcción de interfaces y *HUDs* en este motor.

5.3.1. Interfaces originales

Primeramente, centremos nuestra atención en *UnityGUI*, ahora conocido como *Legacy GUI*. Se trata de una funcionalidad de *Unity3D* que, mediante una variada jerarquía de clases, permite superponer a la imagen mostrada en pantalla elementos de interfaz a golpe de código. Con esto nos referimos a que toda la construcción de los sistemas de interfaz con las clases de *Legacy GUI* a la par que su funcionalidad se realiza mediante *scripting*, de tal modo que el programador debe utilizar en uno o múltiples ficheros instrucciones que, únicamente cuando el juego esté en ejecución, pinten en cada fotograma o *frame* los elementos de interfaz definidos en ellas y controlen la respuesta a la interacción del jugador con dichos elementos.

Veamos un ejemplo de implementación del método *OnGUI* [Imagen 11], que

```
using UnityEngine;
using System.Collections;

public class GUITesting : MonoBehaviour {

    string sentence = "The button has not been clicked.";

    void OnGUI () {

        if (GUI.Button (new Rect (10, 10, 250, 40), "Click me!")) {

            sentence = "The button has been clicked.";
        }

        GUI.Label (new Rect (10, 30, 250, 20), sentence);
    }
}
```

Imagen 11: Código que renderiza un botón y una etiqueta en pantalla

es llamado por el motor en cada fotograma y es el lugar donde debe definirse toda interfaz que deba visualizarse en pantalla. Utilizaremos el lenguaje *C#* por ser el elegido para este proyecto. En el fragmento de código presentado, se define un botón estándar, que en los *frames* en que es pulsado devuelve el valor de verdad *true*, y en el resto de ellos devuelve *false*; y a continuación una etiqueta de texto común y corriente. Observemos, además, cómo el programador debe definir también las coordenadas de pantalla (siendo el origen la esquina superior izquierda de la pantalla) y las dimensiones del rectángulo contenedor del elemento.

La implementación resulta directa y sencilla, aunque quizá no intuitiva la primera vez que se realiza. Podemos contemplar el resultado, que se muestra a continuación [Imagen 12], ejecutando el juego.

Imagen 12: Interfaz resultante, antes y después de pulsar el botón

¿Era este el resultado esperado? Probablemente no sea así, dado que en una primera toma de contacto con la librería, cabría esperar que la etiqueta de texto no se superpusiese al botón, dado que se han definido una detrás del otro y por separado. Fijándose, el desarrollador aprenderá que debe definir las coordenadas de los elementos disjuntamente e ir así construyendo su interfaz, o investigar cómo maquetar el contenido usando estructuras más completas; pero en cualquier caso tendrá que volver al código, corregirlo, y volver a probar.

Este hecho pone de manifiesto un gran impedimento en la productividad a la hora de trabajar con esta sistema de interfaces: el desarrollador no dispone de retroalimentación instantánea del resultado de lo que está implementando ni puede realizar cambios en caliente que permitan experimentar activamente con los elementos de interfaz que ha ubicado. Cada vez que se desee modificar cualquier cosa, por minúscula que sea, se debe volver al código, buscar la línea adecuada, modificarla, compilar el código y probarlo; lo que induce a incontables retrasos a lo largo del proceso de desarrollo. Esta forma de trabajar dista mucho

de la usual y conveniente en *Unity*, que es mediante la administración de objetos de juego o *GameObjects* y sus componentes directamente en el editor de escenas del enorno.

Así pues, destacamos como notable defecto del sistema *Legacy GUI* el que no disponga de un editor de interfaces propiamente dicho. El problema que este hecho produce se agrava considerando otros factores como que la librería, pese a ser muy completa, resulta muy poco flexible de utilizar, ya que al necesitar definir todos sus atributos a código, el programador tiene que manejar explícitamente asuntos relacionados con el escalado de la interfaz dado que las coordenadas se definen en número de píxeles y, por tanto, son insensibles a las variaciones de relaciones de aspecto y de densidad de píxel que sufrirá el juego dependiendo de la pantalla del dispositivo en el que se ejecute Otra deficiencia apreciable es que al ser meras instrucciones codificadas y no objetos del juego como tal, no son directamente susceptibles de manipularse a posteriori con utilidades que incluye *Unity* como el gestor de animaciones *Animator*, que permitiría alterar parámetros de las interfaces en tiempo real y aportaría mucho dinamismo a las mismas.

5.3.2. Textos y texturas en pantalla

Acompañando al sistema de interfaces basado en *scripting* que acabamos de ver, *Unity3D* dispone de un par de componentes determinados, llamados *GUIText* y *GUITexture*, que permiten, en esencia, el pintado de texto e imagen bidimensional sobre la pantalla, respectivamente [25] [26]. No obstante, dada su naturaleza, para ser utilizados deben estar agregados a un objeto de la escena de juego o *GameObject*. Este hecho implica una mejora directa frente a *Legacy GUI*, ya que permite al desarrollador posicionar y manipular los elementos manualmente en el editor de escena incluso mientras el juego está en ejecución, como podría hacer con cualquier otro objeto y sus componentes. De este modo, el proceso de construcción de interfaces se acelera notablemente, ya que uno puede observar el resultado simultánemante a su trabajo.

Los componentes *GUIText* y *GUITexture* son mucho más adecuados para la implementación de una *HUD*, ya que el uso de la transparencia en las texturas y

la posibilidad de colocación manual de los elementos, que se renderizan directamente sobre la pantalla [Imagen 13], facilitan la obtención de resultados deseables en relativamente poco tiempo.

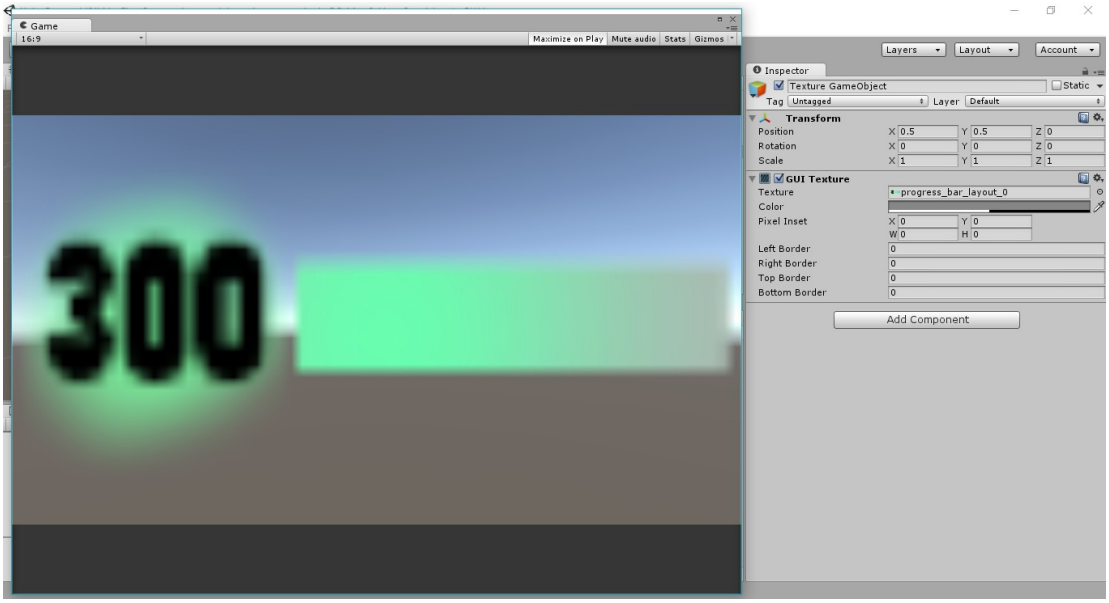

Imagen 13: GUITexture renderizada en pantalla y opciones del componente

Adicionalmente, estos componentes podrán ser alterados y animados en tiempo real directamente desde el editor o mediante otros *scripts* que el desarrollador añada a los *GameObjects* para dotarlos de más funcionalidad, como puedan ser que una textura rote sobre sí misma a lo largo del tiempo o que un texto cambie de color dependiendo de la situación. Se produce así un gran salto en materia de posibilidades de uso, como hemos podido apreciar.

Como cabría esperar, sin embargo, los mencionados componentes distan mucho de ser pefectos y no están faltos de inconvenientes. El primero está relacionado con los propios componentes y los valores que el desarrollador puede manipular en el *Editor*, como puedan ser los de *GUITexture*, apreciables en la imagen.

Un desarrollador sin experiencia en programación gráfica ni conocimientos sobre el funcionamiento de las texturas no comprenderá por qué cuando cambia la pantalla en la cual se renderiza el juego, sus texturas aparecen deformadas siendo que él o ella había ajustado adecuadamente las proporciones y el escalado de las mismas y se visualizaban correctamente mientras desarrollaba. Las consecuencias de la forma de ser y renderizar las texturas, ciertamente

lógicas pero intrincadas, restan flexibilidad y adaptabilidad a las interfaces desarrolladas de este modo, ya que el programador debe encargarse explícitamente de controlar el escalado y posicionamiento de todos los elementos según la pantalla en uso.

Otro muy notable, que podríamos considerar incómodo antes que inconventiente, es que en objetos con *GUIText* y *GUITexture* el uso del componente predefinido *Transform*, que todo objeto posee y determina su posición, rotación y escalado en el espacio, no es coherente con su uso general. Los mencionados componentes requieren que las coordenadas de pantalla donde deberán ubicarse los elementos estén especificadas en el *Transform*, de tal modo que si los valores en anchura y altura de dicho componente se salen del intervalo [0.0, 1.0], el texto o textura será pintado con centro en el exterior de la pantalla, quedando parcial o totalmente fuera de la visión.

No obstante, los valores del componente *Transform,* siguen definiendo inevitablemente las coordenadas del objeto en el mundo tridimensional, porque en él existen los elementos de interfaz pero después se renderizan totalmente aparte. Este hecho resulta chocante las primeras veces que se utiliza y engorroso todas las demás, y causa la impresión de un pobre trabajo de diseño por parte de *Unity*.

Destacaremos, finalmente, un impedimento para nada ilógico que presenta el uso de *GUIText* y *GUITexture*: permiten pintar elementos, pero no al jugador interaccionar con ellos directamente. El desarrollador podrá implementar dicha interacción de diversas maneras, como pueda ser controlar la posición del puntero en pantalla, pero será un trabajo aparte que deberá realizar, ya que la funcionalidad de estos componentes se limita a renderizar elementos en pantalla. Por este motivo, no son directamente viables para la implementación de menús, por ejemplo, pero sí lo son para el desarrollo de *HUDs*.

5.3.3. *NGUI: Next-Gen UI kit*

Tal es el nombre que recibe la extensión de *Unity3D* publicada en 2011 por *Tasharen Entertainment* [27]. Se trata de un extensísimo *plugin* concebido para

llenar cubrir la notable falta de soporte para la implementación de interfaces en *Unity*, y durante los últimos años ha sido prácticamente un estándar *de facto* en la mayoría de proyectos. Ofrece licencias desde 95 dólares hasta los 2000, según las necesidades del desarrollador o estudio.

Esta herramienta es completamente *WYSIWYG* (*What You See Is What You Get*), de tal manera que el resultado es idéntico al trabajo realizado. Tanto es así, que *NGUI* incorpora las mejoras del propio *Unity Editor* necesarias para que el desarrollador manipule los objetos de interfaz (ubicados en un mundo tridimensional) como si se estuviese trabajando en una herramienta de maquetado bidimensional. Uno puede estirar, rotar y posicionar cualquier elemento tan solo usando el ratón; y huelga decir que tal facilidad supuso un avance gigánteo con respecto a las herramientas existentes hasta el momento.

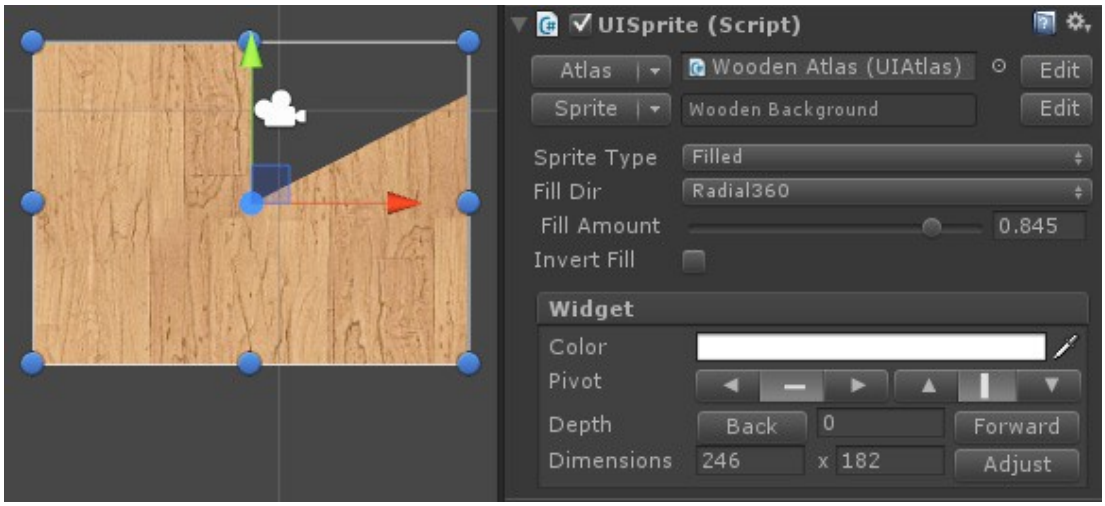

Imagen 14: Manipulación directa de un sprite con NGUI

En líneas generales, destacaremos que *NGUI* posee un riquísimo abanico de elementos de interfaz ya diseñados y con funcionalidad implementada internamente, por lo que su uso acelera el proceso de desarrollo notablemente. Incorpora elementos clásicos como botones, etiquetas de texto, paneles o *sliders* (selectores de valor deslizantes) además de otros más complejos, como componentes de organización flexible de elementos en cuadrícula o paneles deslizantes.

También trae consigo utilidades varias como animaciones ya implementadas, utilizables con cualquier elemento de interfaz, un sistema de anclas entre

elementos que logra que estos escalen o se posicionen según las relaciones de unos con otros, o un muy destacable constructor de atlas [Imagen 15] de texturas (grandes imágenes que contienen numerosas imágenes o *sprites* organizados de tal forma que el espacio ocupado sea el mínimo), que permite que la tarjeta gráfica del dispositivo pinte todos los elementos gráficos de la interfaz con una sola llamada, cuyo uso tiene un efecto vastamente positivo en el rendimiento.

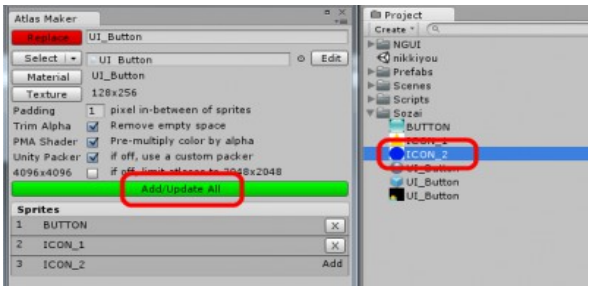

Imagen 15: Adición de un sprite a un atlas con NGUI

La experiencia de uso de la herramienta es por lo general gratificante, dado que gracias al sistema de anclas, al que uno deberá acostumbrarse, y la existencia de elementos ya definidos, resulta bastante sencillo distribuir elementos de un sistema de interfaces y que, además, estos se reposicionen o escalen según el área renderizada por la cámara, normalmente la pantalla al completo, sin problema alguno.

Algunos aspectos en concreto, como la correcta disposición jerárquica de los paneles de *NGUI* y su orden de renderizado o el funcionamiento adecuado de listas deslizantes con elementos posicionados dinámicamente [Imagen 16], sí son más costosos de conseguir, de aprender e interiorizar; y su proceso de consecución puede resultar algunas veces frustrante, dada su alta complejidad.

Imagen 16: Botones en cuadrícula en un panel deslizante de NGUI

Sin entrar en más detalle, sobre *NGUI* concluiremos que es una *suite* muy completa para la creación de interfaces interactivas que, pese a algunas dificultades que su uso pueda presentar, permite la creación de todo tipo de sistemas de interfaz con una eficiencia notable y con una gran amplitud de posibilidades y herramientas adicionales varias. Será, pues, una opción altamente recomendable y muy a tener en cuenta para el desarrollador que

quiera tener una *HUD* completa, versátil y resultona en su juego.

5.3.4. La nueva interfaz de usuario

Con la publicación de la versión 4.6 del motor en noviembre de 2014, *Unity3D* incluyó en sus librerías su nuevo sistema de interfaces, llamado *The New UI*, o simplemente *UI* [28]. Esta novedad fue bien recibida por muchos, ya que por fin tuvieron a su disposición una herramieta gratuita para la construcción de interfaces competente en los tiempos actuales. Las características del nuevo sistema son, en una amplia variedad de casos, similares a las que ya ofrecía *NGUI*, con algunas diferencias.

La actualización incorporó tres grandes novedades. La primera y más relevante de ellas fue la inclusión de una componente derivado de *Transform*, llamado *RectTransform* [29], que cumple las mismas funciones pero en lugar de administrar las coordenadas y atributos de un punto en el espacio, sirve para administrar los parámetros de un rectángulo. De igual forma que con *Transform* normalmente se hacen ajustes en un sistema de coordenadas local, es decir, definido por las coordenadas del *Transform* del padre del objeto, con *RectTransform* administraremos la relación del rectángulo con la del rectángulo padre, que también poseerá un *RectTransform*.

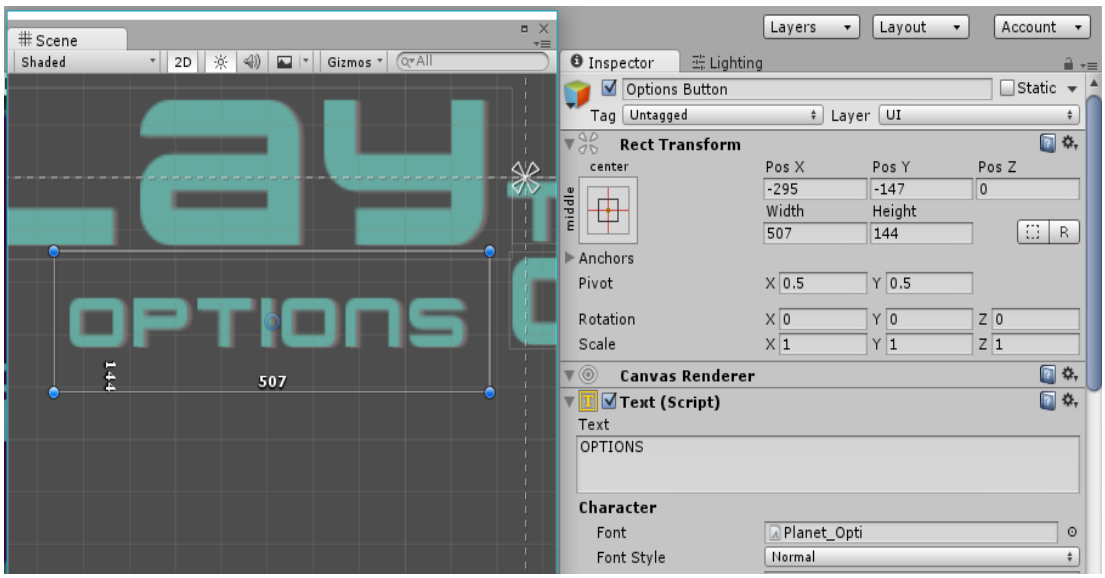

Imagen 17: Ajuste del rectángulo de un texto manipulando su RectTransform

Para los objetos destinados a ser elementos de interfaz, que comentaremos a continuación, el *RectTransform* sustituye al convencional *Transform* en el editor.

No obstante, la coexistencia y relación directa de estos dos componentes les aporta una versatilidad inusitada: todo objeto puede manipularse en el espacio como si tuviera cualquiera de dichos dos componentes indistintamente [Imagen 18]. Esta característica permite escalar, rotar y posicionar objetos de la forma más intuitiva y conveniente en cada caso: desde su punto central o desde sus extremos.

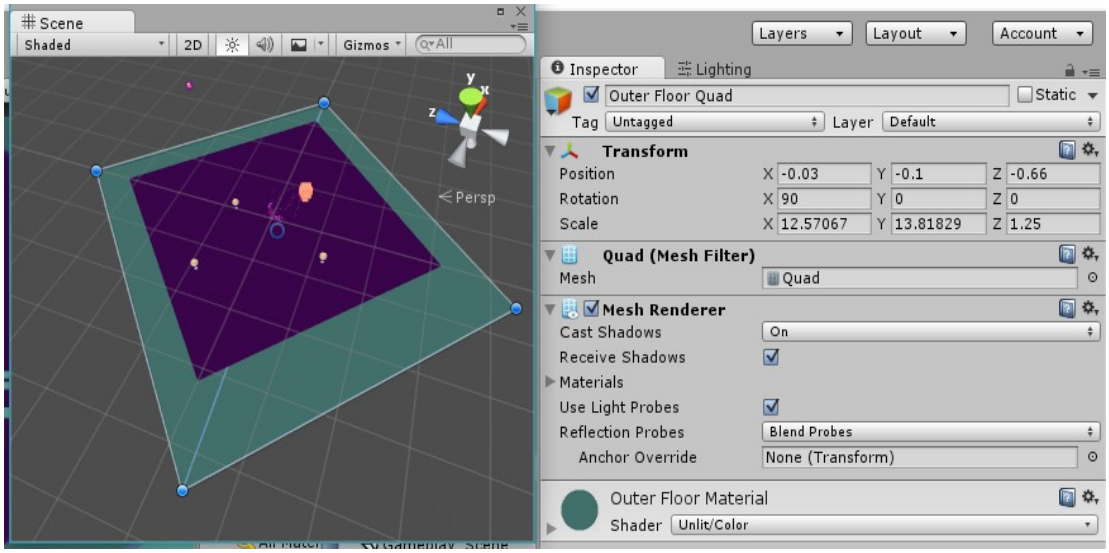

Imagen 18: Manipulación un objeto del espacio como si fuera de intefaz

Cabe destacar, para entender el funcionamiento del *RectTransform*, su sistema de anclas. A diferencia de con *NGUI*, donde cada lado de un elemento o *widget* se podía anclar a cualquier otro, en la nueva interfaz de *Unity* representan únicamente la relación de escalado de un rectángulo con respecto a su padre. Las esquinas del hijo se mantendrán siempre a una distancia invariante del punto anclado al del padre, por lo que resulta bastante sencillo predecir cómo escalarán los elementos dependiendo de la relación de aspecto de las pantallas, y diseñar en consecuencia. En la Imagen 9, los cuatro vértices del rectángulo están anclados a un solo punto del padre, indicado por las cuatro flechas blancas. Dichos vértices mantendrán la distancia al punto anclado independientemente de si el padre es alterado.

La segunda gran novedad fueron en sí los nuevos elementos de interfaz. Una serie de componentes dentro de la nueva librería *UnityEngine.UI* que definen elementos usuales como cajas de texto, imágenes, botones o barras de desplazamiento; y utilidades más completas, como *layouts* para organizar

elementos automáticamente, máscaras para ocultar elementos bajo otros o efectos visuales de transición para cuando se interacciona con determinados elementos. Todos ellos deben colocarse bajo un lienzo o *Canvas*, que delimita el área renderizada y define cómo escalará la interfaz a lo ancho y alto de la pantalla. Además, los nuevos componentes disponen de un amplio plantel de funciones y propiedades para su administración desde código, además de las opciones ajustables en el editor.

Finalmente, la tercera novedad es la que dota a las interfaces construidas con la nueva *UI* de interacción. Se trata de un conjunto de componentes que se crean automáticamente en nuestro juego al crear una nueva interfaz. Son los siguientes:

- *EventSystem*: permite la interacción con los elementos de interfaz, teniendo en cuenta la posción de la pantalla donde se produce una acción.
- *StandaloneInputModule* y *TouchInputModule*: permiten el uso de teclado y ratón, en el primer caso, y de toques en pantalla táctil, en el segundo. El *EventSystem* los requiere para saber qué interacciones han tenido lugar o dónde está el cursor.
- *GraphicRaycaster*, que unido a un *Canvas*, permite que se interaccione con los elementos bajo él.

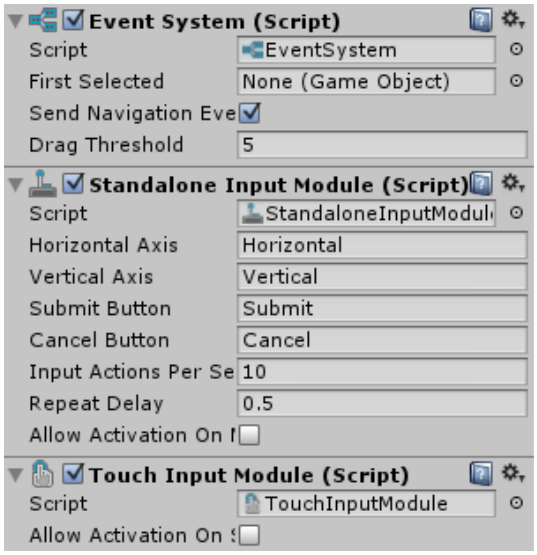

Imagen 19: El sistema de eventos y módulos de interacción por defecto

El funcionamiento, en líneas generales, es sencillo. El *EventSystem* lanza rayos desde los puntos de la pantalla donde los *InputModules* le comuniquen que hay acciones, para detectar con qué elemento del *Canvas* está tratando de interactuar el jugador. El elemento en cuestón recibirá una variedad de eventos a los que puede responder, como *OnPointerClick* (disparado al clicar sobre él), y ejecutar las funciones necesarias, si se le asignan.

Nada está libre de defectos, por supuesto. Este sistema de interfaces cuenta con unos cuantos, que pueden pasar desapercibidos si el desarrollador los conoce y construye sus interfaces en consecuencia. El más grave es, posiblemente, que el empaquetado de *sprites* en *atlas* no está optimizado y pintar las interfaces en pantalla consume muchos. Para sufragar este problema, se deberán construir *atlas* en alguna herramienta que lo haga adecuadamente, para que la tarjeta gráfica pueda pintar todos los elementos de una pasada.

Otros dos aspectos que pueden resultar molestos o un impedimento si no se tienen cuenta, son la limitación de que un elemento sólo se puede anclar a su padre, y el hecho de que *Unity*, por norma general, renderiza los elementos de interfaz por orden de aparición en la jerarquía. El primero puede no suponer un problema si se diseña cuidadosa y jerárquicamente la interfaz, y el segundo es fácilmente solventable si se fuerza una ordenación determinada entre diferentes *Canvas*. Cabe mencionar que *NGUI* disponía de sus propios métodos para lidiar con estos asuntos, más intuitivos y perceptibles a simple vista.

5.3.5. Elección de un sistema para el proyecto

En el presente proyecto, utilizar *NGUI* podría haber sido una opción totalmente viable dada su versatilidad y excelente rendimiento, razones por las cuales es preferido por muchos, pero se hará uso de la nueva *UI* de *Unity* para la construcción de *HUDs* modulares. Se ha optado por ella por las siguientes razones:

- Viene integrada y lista para usarse en *Unity3D*.
    - Utilizar la extensión desarrollada en este proyecto no requerirá instalar nada más aparte del propio *Unity Editor*.
- Recibirá futuras mejoras por parte de *Unity Technologies*.
- Es gratuita para cualquier desarrollador, por tanto plenamente accesible.
    - Se garantiza así que el número de personas que pueden utilizar la extensión es máximo.
- Su uso es sencillo e intuitivo a nivel básico, igual que su integración.

- Meses de uso de todas los sistemas presentados muestran que este es el menos frustrante.
- La curva de aprendizaje es menos pronunciada que la de *NGUI* y puede competir con él a nivel de características.

6. Desarrollo del creador de *HUDs* modulares

Para documentar el proceso de creación de la extensión, bautizada como *CustomHUD*, expondremos primeramente cómo se ideó y diseñó su apariencia y funcionalidad para luego comentar los puntos más relevantes de su implementación, así como de los objetos de juego y componentes relacionados.

6.1 Diseño

La sencillez y rapidez de uso son siempre dos factores de mérito a tener en mente cuando se trata de ofrecer un producto al público. Nada más lejos de la realidad, el *plugin* diseñado apunta a cumplir con su propósito sin suponer esfuerzo alguno al desarrollado a la hora de utilizarlo.

Se definirán tres tipos de entidades clave alrededor de los cuales girará toda la implementación posterior. Estas entidades serán:

- *HUDController*: encargado de manejar toda las acciones relacionadas con la *HUD* que puedan tener lugar.
    - Tendrá un componente *Canvas* que contendrá los detalles acerca de cómo deben escalar los elementos de la *HUD*.
- *HUDAnchor(s)*: servirán como contenedores bajo los que agrupar los elementos de interfaz en la *HUD*.
    - Cada *HUDAnchor* determina a qué parte de la pantalla estarán anclados los elementos.
    - Cada una dispondrá de un componente *Canvas* independiente que permitirá la ordenación del renderizado.
- *HUDElement(s)*: serán las interfaces visibles propiamente dichas, que se ubicarán bajo cualquier ancla.
    - Se dispondrá de una variedad de elementos de utilidad diversa.
    - Se suscribirán al controlador desempeñando un rol específico para que este pueda utilizarlos cuando sea necesario.

Es importante tener estos tipos de entidades definidos desde un principio, pues se harán menciones a ellos a lo largo de los apartados siguientes. En una escena habrá un único controlador de *HUD*, y un ancla por cada lugar de la pantalla al que anclar elementos: arriba, abajo, derecha, izquierda y centro.

6.1.1. Las ventanas

Para la creación de *HUDs* modulares, se propone el diseño de dos nuevas ventanas distintas para el *UnityEditor*, cada una desempeñando una función diferente. Estas son:

- Ventana de creación de *HUD*: presentará una interfaz muy sencilla para que el desarrollador realice, si quiere, ajustes relacionados con el escalado de su interfaz según el tamaño de la pantalla y dote al objeto que representa la *HUD* de un nombre y de una fuente por defecto para sus elementos.

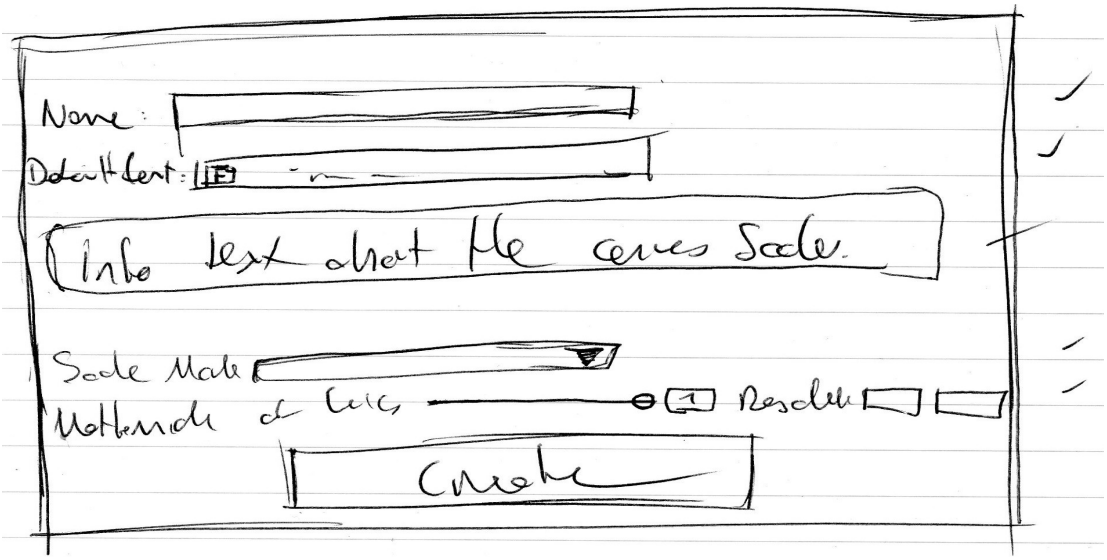

Imagen 20: Boceto original de la ventana de creación de HUDs

- Ventana de edición de *HUD*: con una interfaz más completa, permitirá añadir elementos prediseñados a la interfaz, pero con un alto nivel de personalización.
    - Diferentes elementos de interfaz disponibles con las opciones de configuración pertinentes a cada uno.
    - Manejo intuitivo mediante acciones de arrastrar y soltar o entrada

directa de valores.

La función de la primera de las ventana es sencilla. Consistirá en recopilar los datos básicos acerca de cómo desea el desarrollador que sea su *HUD* y, después, crearla. Instanciará un nuevo *GameObject* con el nombre elegido y se le añadirá el componente *HUDController* (que detallaremos en el apartado relativo a la implementación) y un *Canvas* con los parámetros elegidos. Después se crearán, como hijos del controlador, las cinco *HUDAnchor* pertinentes.

La segunda ventana es más compleja, pues consta de una gran cantidad de opciones. A través de esta ventana, el desarrollador podrá elegir qué tipo de elemento de interfaz desea añadir mediante unos botones ubicados en la parte superior y personalizar diferentes aspectos acerca de ellos a continuación, bien introduciendo datos manualmente o añadiendo material desde las carpetas del proyecto, como puedan ser imágenes, buscándolo o arrastrándolo. Algunos de dichos aspectos serán comunes a todo *HUDElement*, como los relacionados con el *HUDAnchor* al que irán anclados y su ubicación respecto a esta. Finalmente, podrá crearse una o más copias del elemento que se estaba personalizando pulsando el botón pertinente al final de la pantalla.

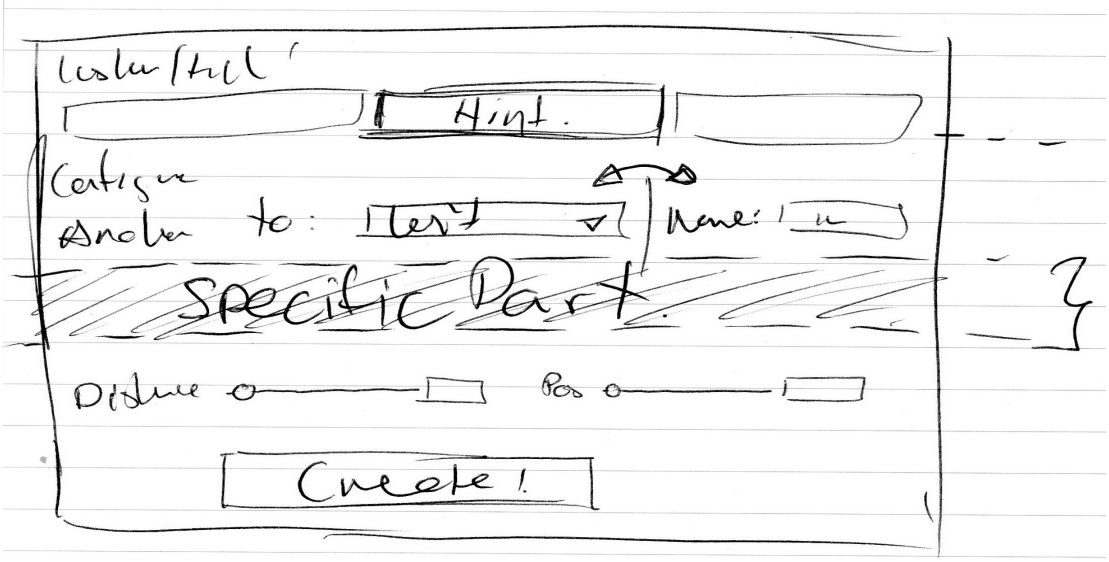

Imagen 21: Boceto original de la ventana de edición de HUD

Como se puede apreciar en el boceto [Imagen 21], la intención es que la ventana disponga de un espacio flexible habilitado para mostrar un apartado de configuración específica al tipo de elemento de interfaz que se está creando,

mientras que el resto, común a todos, se mantiene invariante. Cabe destacar que este boceto fue realizado al principio del diseño y algunos controles más fueron añadidos en la versión final.

Enumeremos, a continuación, las opciones de configuración disponibles comunes a todos los elementos de *HUD*:

- Configuración básica:
  - Nombre para el *GameObject* que se creará
  - Área de la pantalla a la cual anclar el elemento
- Configuración del *RectTransform* del objeto:
  - Distancia al borde de la pantalla
  - Posición/desplazamiento a lo largo de la pantalla
  - Inclinación del objeto
- Ajustes de visualización:
  - Si el objeto en cuestión implementa los métodos necesarios para ello, se presentan las siguientes opciones de visualización:
    - Si el elemento de interfaz debe comenzar visible o no
    - Si su entrada o salida de la pantalla deben ser animadas
      - Efecto de fundido
      - Efecto de deslizamiento
      - Tiempo de duración de dichos efectos

En cuanto a los ajustes específicos para cada tipo de elemento, veremos cuáles son las opciones de personalización que presenta cada uno en el siguiente apartado cuando se expongan los elementos prediseñados proporcionados junto a la extensión.

6.1.2. Los elementos prediseñados

Ofrecer una herramienta novedosa para crear *HUDs* al público resultaría poco

más que inútil si no se entregara con elementos de interfaz ya preparados para su uso, por lo que se propone el desarrollo de cuatro tipos diferentes de ellos, cada uno con distinta complejidad y funcionalidad. Veamos cuáles son, qué funciones suelen desempeñar en los juegos y el conjunto de ajustes de personalización para el desarrollador que presentan.

En primer lugar, se ofrecerá un elemento para mostrar subtítulos en la *HUD*. Su uso común consistirá en la transcripción de diálogos o narraciones de voces en *off* que tengan lugar en el juego, pero también puede ser fácilmente utilizado para traducir a otros idiomas o facilitar la lectura de texto que haya escrito en algún objeto de la escena mientras se mira a él. Acerca de sus características, destacamos:

- Puede suscribirse al controlador como subtítulo primario o secundario, por si se quiere alternar en el uso de dos subtítulos con aspecto diferente.
- Se le puede dotar de un texto inicial.
- Puede dársele una fuente distinta a la por defecto del controlador.
- Puede seleccionarse el color del texto.
- Se le puede añadir un sombreado y/o un contorno de colores.
- Dispone de animaciones de fundido para mostrarlo u ocultarlo.
- Se debe anclar exclusivamente al borde superior o inferior de la pantalla, pero se puede distanciar de ellos tanto como se desee.
- Su presentación será siempre centrada en la pantalla.
    - El campo de posición es sustituido por el de margen, que indica cuánto margen tendrá el rectángulo de texto con respecto a los bordes laterales de la pantalla.
    - El campo de inclinación es sustituido por el de altura, que indica cómo de alta será la caja de texto y, en consecuencia, qué tamaño tendrá la letra del texto.

Como segunda elemento se diseñará un panel de alerta o advertencia, cuyo

uso más corriente es advertir al jugador de algún hecho de altísima importancia. Normalmente, se trata de un mensaje grande y vistoso ubicado en el centro o la parte superior de la pantalla para que no pueda ser obviado con facilidad. Presentará, como opciones de configuración:

- Puede dársele un texto y un icono de advertencia iniciales.
- Es posible ponérsele una imagen de fondo al rectángulo de la advertencia.
- Se le puede dar color al texto y elegir su fuente.
- Puede dársele sombreado y contorno de colores al texto.
- Se puede modificar el tamaño del icono o dejar el propio de la imagen.
- Puede ajustársele el tamaño del rectángulo.
- Se debe elegir si el icono aparece a un lado o a otro del texto.
- Dispone de animaciones de fundido al mostrarlo u ocultarlo.

Similar al anterior, aunque más versátil y menos vistoso, se proveerá un panel de indicio. Es un recurso muy utilizado en el mundo de los videojuegos cuya función más corriente es indicar al jugador que pulse una tecla o realice una acción determinada según la circunstancia en que se encuentre. Ofrecerá características tales como:

- Puede suscribirse como indicación de acción, de recordatorio, u otros (*Action*, *Reminder* y *Other Hint*).
- Se le puede dar un texto y una imagen iniciales, así como ajustar su tamaño.
- Puede cambiársele la fuente por otra distinta a la por defecto.
- Puede cambiársele el color del texto y darle sombreado o borde de color.
- Permite posicionar el icono arriba, abajo, a la derecha o a la izquierda del texto.
- Dispone de animaciones de fundido y de entrada al ocultarlo o mostrarlo.

Finalmente, como cuarto elemento prediseñado se aportará la clásica barra

de progreso, utilizada para mostrar cantidades de cualquier índole. En los videojuegos suelen utilizarse para indicar la cantidad de salud, energía o munición restante o el progreso hacia un objetivo, entre otros muchos. Incluirá las siguientes opciones de personalización:

- Puede ser suscrita al controlador como barra de salud, energía, experiencia, nivel, u otros.
- Se le pueden asignar los valores mínimos y máximos que puede representar.
- Puede disponer de un indicador numérico para su valor.
- Puede decidirse si la barra utilizará valores enteros o con coma flotante.
    - En el segundo caso, puede definirse el formato de estos números al ser mostrados en el indicador numérico.
- Se puede cambiar la fuente y el color del indicador numérico.
- Puede añadírsele al indicador un sombreado o borde de color.
- Puede elegirse una imagen para el rectángulo de fondo del indicador numérico.
    - Se elija una o ninguna, se puede y debe elegir un tamaño para el espacio que abarca el rectángulo que contiene el indicador numérico.
- Puede elegirse una longitud de la barra de progreso y su altura/grosor.
- Puede elegirse una imagen para el fondo y otra para el rellenado de la barra de progreso.

El desarrollador podrá, una vez ubicados elementos en la *HUD*, realizar todos los ajustes posteriores que desee modificando directamente los *GameObjects* que la herramienta habrá creado, o jugando con los parámetros de sus componentes. Podrá también duplicarlos, cambiarlos de un ancla de la pantalla a otra o eliminarlos manipulando la jerarquía de objetos en la ventana de *Hierarchy*, así como activar, desactivar y añadirles componentes en el *Inspector*.

Además, podrán crearse nuevos elementos de *HUD* que, si siguen una serie

de pautas adecuadas, podrán añadirse como nuevas opciones a la ventana de editor de *HUD* y podrá dotarse al controlador de la funcionalidad necesaria para trabajar con ellos.

6.1.3. El funcionamiento

Dedicaremos este apartado al diseño de la lógica que permite el correcto funcionamiento de los objetos que formarán el sistema de *HUD*. No se detallará aquí el diseño de la funcionalidad de las ventanas, pues ya se ha comentado cómo es la interacción del usuario con ellas y que su labor es únicamente la de recopilar información introducida por el desarrollador, y crear los objetos acordemente a esos datos.

Veamos, por partes, cómo todas las piezas de nuestro sistema encajarán entre sí. Se presenta en diagrama [Imagen 22] la relación existente entre las entidades de nuestro sistema, ahora que las conocemos todas. Servirá como base de la implementación de las clases pertinentes cuando se proceda con la

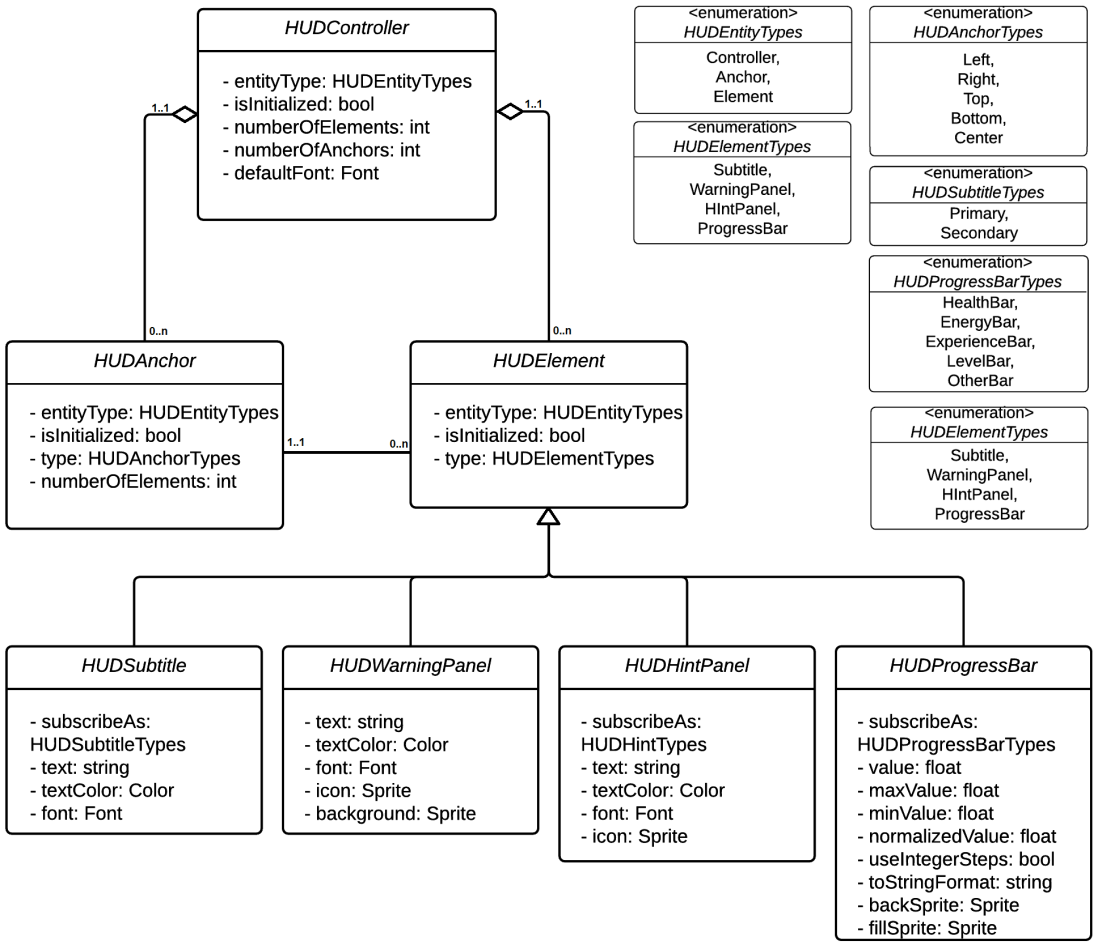

Imagen 22: Diagrama de clases de las entidades del sistema de HUD

programación de la lógica de todo el sistema de *HUD* que permita la su automatización y uso sencillo y rápido. Pasemos ahora al diseño de cómo interactuarán las entidades durante la ejecución, comenzando por la inicialización.

Cuando se disponga de un *HUDController* y tantos *HUDElements* como se desee anclados a él y se inicie una escena de juego, la inicialización de la *HUD* tendrá lugar sin necesidad de que el programador tome parte en ella.

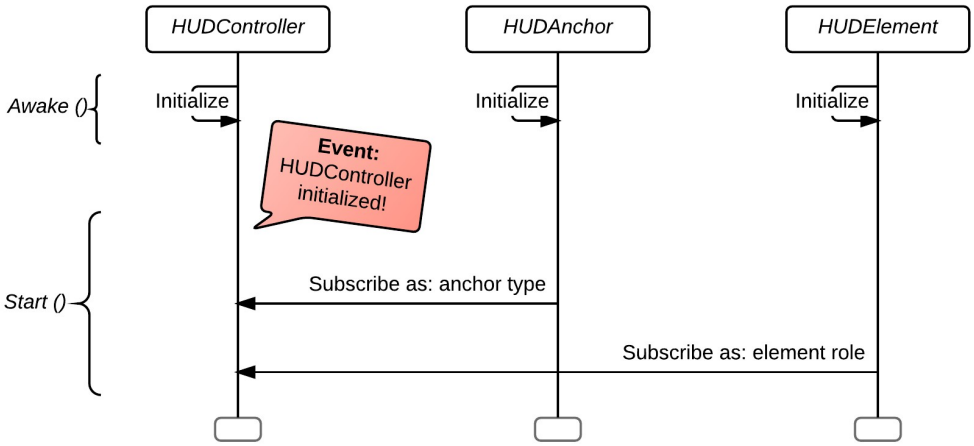

Imagen 23: Inicialización de la HUD al iniciar una escena

Cada entidad por sí misma inicializará lo que necesite en *Awake* y, cuando un controlador de *HUD* anuncie que se ha inicializado en la escena, el resto de entidades se suscribirán a él para prestar servicio según el rol con el que se hayan suscrito. Es precisamente este sistema de <<roles>> el que dota a la *HUD* de su facilidad de uso para el programador y de su modularidad a nivel de funcionalidad. Veamos qué significa.

En el apartado anterior hemos enumerado como qué puede suscribirse al controlador cada uno de los elementos prediseñados, salvo para el panel de alerta, que sólo puede cumplir la función de alertar al jugador. El abanico de posibilidades de suscripción de los que dispone cada elemento son los roles que puede desempeñar. Sirven para, por un lado, que el desarrollador defina desde el mismo momento en que crea cada elemento cuál será su cometido; y por otro, para que el controlador de *HUD* sepa que <<dispone de un elemento cuyo es rol es *X*>>.

Gracias a esto, el programador no necesitará ninguna referencia al elemento concreto de la *HUD* que quiere utilizar en un instante determinado en su código, sino que simplemente deberá utilizar el rol que desempeña en su llamada, y el controlador se encargará del resto.

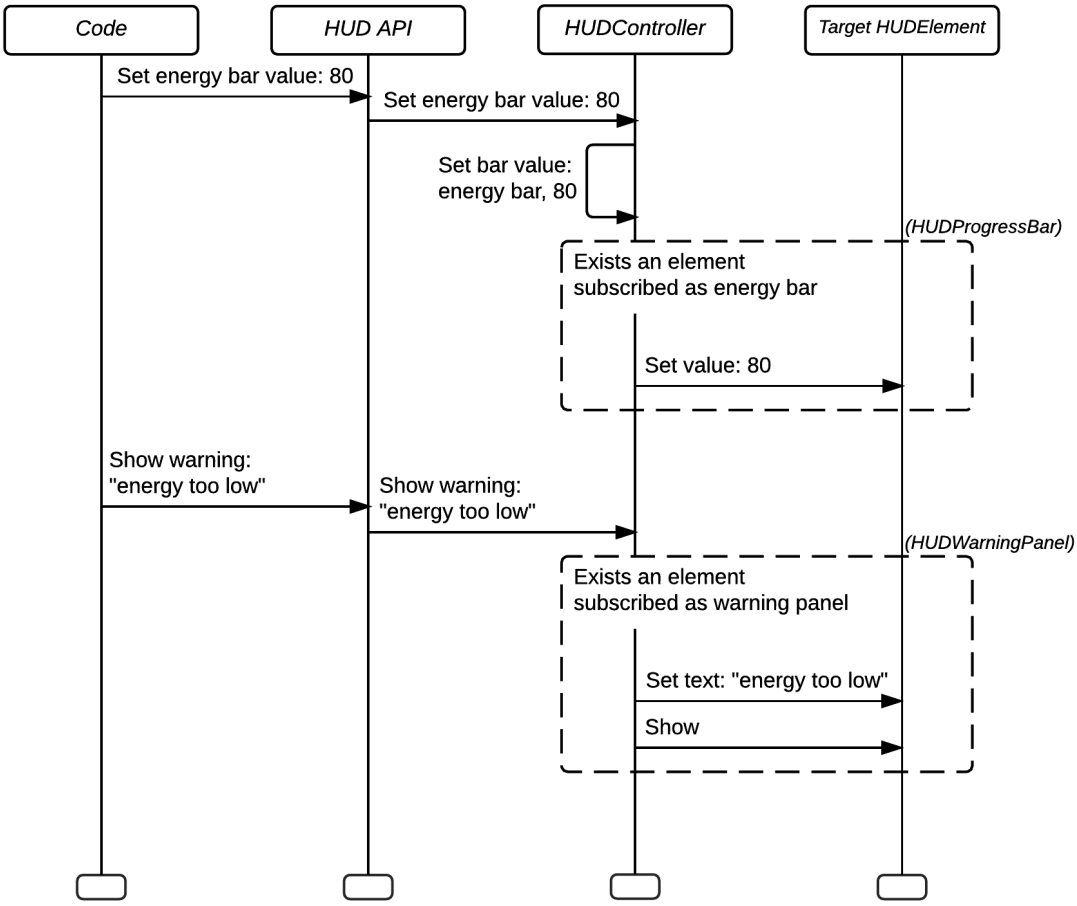

Imagen 24: Ciclo de llamadas a la HUD por parte del código del desarrollador

En este aspecto, el beneficio es triple: se obtiene mayor abstracción en la programación, se consigue un buen nivel de encapsulamiento ya que sólo se requiere de comunicación con el controlador (mediante una *API* de llamadas estáticas o con una referencia a él) y que sólo el controlador tenga referencias a los elementos de interfaz; y se gana modularidad pues modificar desde el *Editor* qué elementos de *HUD* desempeñan qué roles no afectará en absoluto a la correctitud del código que los utilice.

6.2. Implementación

Para transformar nuestros diseños en resultados tangibles, utilizaremos

principalmente el *UnityEditor*, las herramientas *MonoDevelop* y *Sublime Text 3* para la escritura de código y software de edición de imagen para diseñar material gráfico que sirva como ayuda en la extensión. Se utilizará el lenguaje *C#* durante todo el desarrollo.

Daremos un vistazo general a las directrices seguidas durante la implementación y algunos detalles más concisos de ella para destacar los aspectos del desarrollo en *Unity* que más útiles hayan resultado. En primer lugar veremos los aspectos clave de la programación realizada para la extensión del *Editor* creando ventanas nuevas. Después, la creación de los elementos de interfaz predefinidos para su posterior uso y, finalmente, haremos un breve repaso de los puntos más importantes de la implementación realizada de las clases y lógica.

6.2.1. *Scripting* para el *Editor*

Llega ahora el momento de ponerse manos a la obra con la extensión de *Unity* propiamente dicha. El motor pone a nuestra disposición todo un repertorio de clases, funciones y demás utilidades para desarrollar nuestras ventanas con total libertad, reciclando en gran medida la forma de trabajar de la antigua interfaz. Pueden realizarse virguerías visuales haciendo uso de todo ello, pero en nuestro caso buscamos que el uso de las ventanas sea directo y sencillo, por lo que principalmente constarán de métodos de entrada enfocados a la personalización de las interfaces.

En todos los *scripts* del proyecto se ha trabajado bajo el espacio de nombres *CustomHUD*, tal que todas las clases definidas sean accesibles desde cualquier otro *script* si se importa dicho espacio de nombres, o mediante acceso explícito. Así pues, comencemos con los dos primeros ficheros que fueron creados bajo este espacio:

- *HUDCreatorWindow.cs* es el archivo en el cual se implementó la primera de las dos ventanas diseñadas, al completo.
- *HUDEditorWindow.cs* contiene la segunda y más compleja de ellas.

Adicionalmente, por el bien de la organización del código, se crearon

prácticamente a su vez dos ficheros más para contener definiciones globales a todo el espacio de nombres:

- *ConstantsEditorHUD.cs* define una clase llamada *HUDConst*, que permite un rápido acceso a toda una serie de constantes relativas a tamaños y posiciones para los *RectTransform* de los objetos de interfaz, y que serán utilizados múltiples veces.
- *EnumsHUD.cs* contiene todos los tipos enumerados que se van a utilizar en todo el desarrollo, relativos a la lógica de las entidades de las *HUDs*, pero necesarios también en las ventanas.

Para trabajar con el editor de *Unity* desde código [30], necesitaremos importar el paquete *UnityEditor* en nuestro *script*. Una vez hecho, podremos definir una nueva ventana de editor simplemente con crear una nueva clase que herede de *EditorWindow*. Para visualizar una ventana que hayamos creado, basta con llamar a la función *EditorWindow.GetWindow* con la clase que hemos definido como argumento. Lo que haremos en nuestro caso será definir una entrada en el menú principal de *Unity*, que al clicar en ella muestre la ventana pertinente.

Para construir las ventanas, debe hacerse uso de la función *OnGUI*, que pintará todo lo definido dentro de ella en la ventana. En ellas, se hará uso de numerosos campos de entrada de distinta índole, que son a través de los cuales el desarrollador personalizará su *HUD*. Estos campos funcionan tal que cada vez que se repinta la ventana, muestran el valor que se les pase como parámetro y devuelven uno nuevo si ha variado, o el mismo en caso contrario.

Veamos un sencillo ejemplo, donde se crea un campo de entrada para una ventana que permite al usuario seleccionar un color [Imagen 25].

Imagen 25: Código que define un selector de color y su resultado

En el ejemplo, el método predefinido del editor de *Unity ColorField* produce un campo de selección de color que devuelve el color seleccionado. Este valor se guarda en la variable *selectedColor* (declarada previamente como atributo privado) y, como puede verse, esta misma variable es la que utiliza el método para mostrar un valor, por lo que cada vez que se repinte la ventana, el selector de color mostrará la información actualizada, que acaba de recopilar. Además, vemos que permite el uso de ajustes como, en el caso presentado, la anchura máxima del campo.

El patrón de funcionamiento de todos los campos de entrada de información es similar si no idéntico, aunque varíen en la forma que presentan. La forma más usual de trabajar con ellos es guardar su valor en variables no temporales, como hemos visto en el ejemplo, hasta que se haga uso de ellas. Enumeremos unos cuantos de los que más utilizaremos:

- *TextField*, para la introducción de una cadena de texto.
- *IntField* y *FloatField*, que permiten la entrada de valores numéricos.
- *IntSlider* y *Slider*, de igual utilidad que los anteriores pero muestra un selector deslizante acotado entre dos valores determinados.
- *Toggle* y *ToggleLeft*, que permiten poner un valor *bool* a verdadero o a falso mediante una casilla de verificación o *checkbox*.
- *ColorField*, para seleccionar un color y visto en el ejemplo anterior.
- *ObjectField*, que permite asignar a una referencia un objeto arrastrándolo hasta un campo.
- *Vector2Field*, para introducir dos valores coordenados X e Y.
- *EnumPopup*, que permite la selección de un valor entre todas las posibilidades de un tipo enumerado.

Además, se ha echado mano de una buena cantidad de otros métodos que ayudan a la construcción de ventanas, como:

- *Label*, para mostrar un texto o una imagen.

- *Box*, que desempeña la misma función que una label pero ubicando el contenido dentro de una caja.
- *Space*, que introduce una cantidad determinada de píxeles de espaciado entre un elemento y el siguiente.
- *FlexibleSpace*, que introduce el máximo espacio posible entre dos elementos.
- *BeginHorizontal* y *EndHorizontal*, para iniciar y terminar bloques de elementos adyacentes ordenados horizontalmente.
- *BeginVertical* y *EndVertical*, de igual funcionalidad pero en vertical.
- *Button*, que muestra un botón con un texto o imagen y devuelve *true* cuando ha sido pulsado.

Combinando todos ellos, construiremos nuestras ventanas. Como última parte de este punto, veamos algunas particularidades de las ventanas y el resultado final de cada una.

Para la primera de las dos ventanas, cuando el desarollador elija la opción del menú de crear una nueva *HUD*, se comprobará si existe alguna en la escena. En caso negativo, se mostrará la ventana para que pueda proceder con su trabajo. En caso afirmativo, sin embargo, se le mostrará una pequeña ventana a modo de *popup* que le advertirá de la situación y le dará la opción de editar el existente o reemplazarlo por uno nuevo, o no hacer nada.

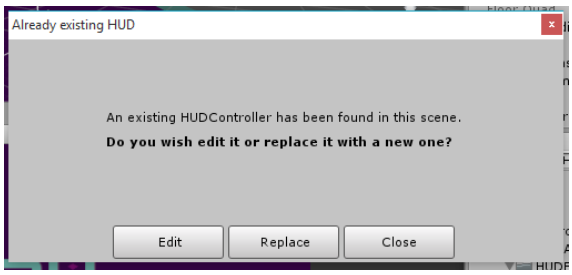

Imagen 26: Popup mostrado si ya existe una HUD

Hay un único aspecto a destacar de ella: cómo desempeña su función. Sencillamente, crea un nuevo *GameObject*, se le añaden los componentes *HUDController, Canvas* y *CanvasScaler* y se les ajustan los parámetros según las elecciones del usuario. Luego, se crean cinco *GameObjects* como hijos de este, y a cada uno se les añade un componente *HUDAnchor* y un *Canvas*. El resultado de su implementación se muestra en la imagen siguiente [Imagen 27].

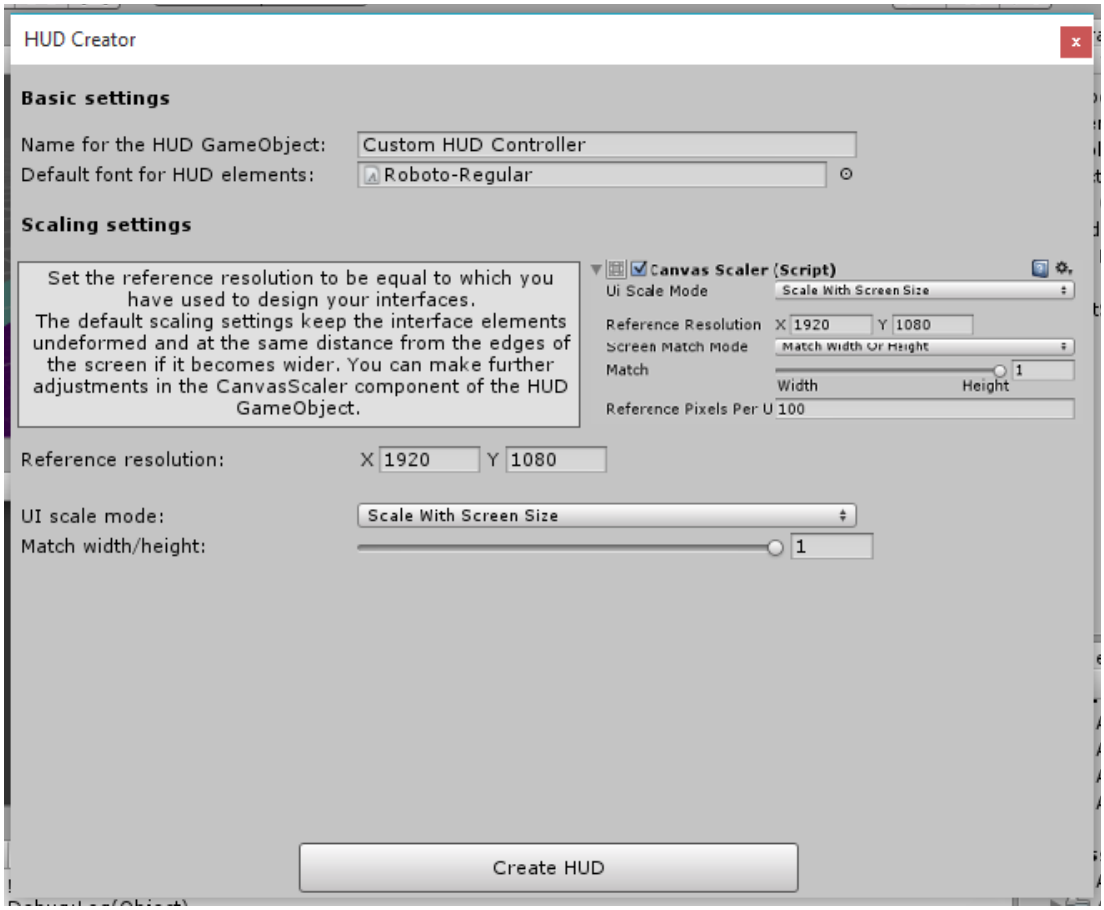

Imagen 27: Ventana de creación de HUD terminada

La segunda ventana, también muestra un popup si no existe una *HUD* que editar, dando opción a crear una. Acerca de su implementación, que es extensa cuanto menos, cabe destacar un par de aspectos. El primero de ellos es que se ha decidido utilizar una serie de variables de tipos estructurados o *structs* para mantener en memoria, mientras la ventana esté en uso, todos los detalles que el desarrollador ha ido configurando acerca de un elemento de interfaz, y así mantenerlos organizados y accesibles.

```
struct RectTransformParams {
    public int      distance;
    public int      position;
    public float    tilt;
}

struct SubtitleParams {
    public HUDSubtitleTypes type;

    public int margins;
    public int height;

    public Font     font;
    public string   text;
    public Color    textColor;
    public bool     shadow;
    public Color    shadowColor;
    public bool     outline;
    public Color    outlineColor;
}
```

Imagen 28: Ejemplos de structs usados en la ventana de edición de HUD

El segundo aspecto es cómo actúa la ventana cuando se pulsa el botón de crear el elemento. El procedimiento es bastante directo: carga en memoria desde la carpeta de recursos del proyecto el *GameObject* prediseñado correspondiente al elemento que se desea añadir a la *HUD*, lo instancia como hijo de la *HUDAnchor* pertinente sirviéndose de las llamadas de utilidad del archivo

HUDElementCreatorUtility.cs y realiza en su estructura y componentes todos los cambios necesarios para aplicar las preferencias seleccionadas por el desarrollador, resultando en una nueva pieza para la interfaz lista para ser utilizada.

El resultado de la implementación de la ventana es el siguiente:

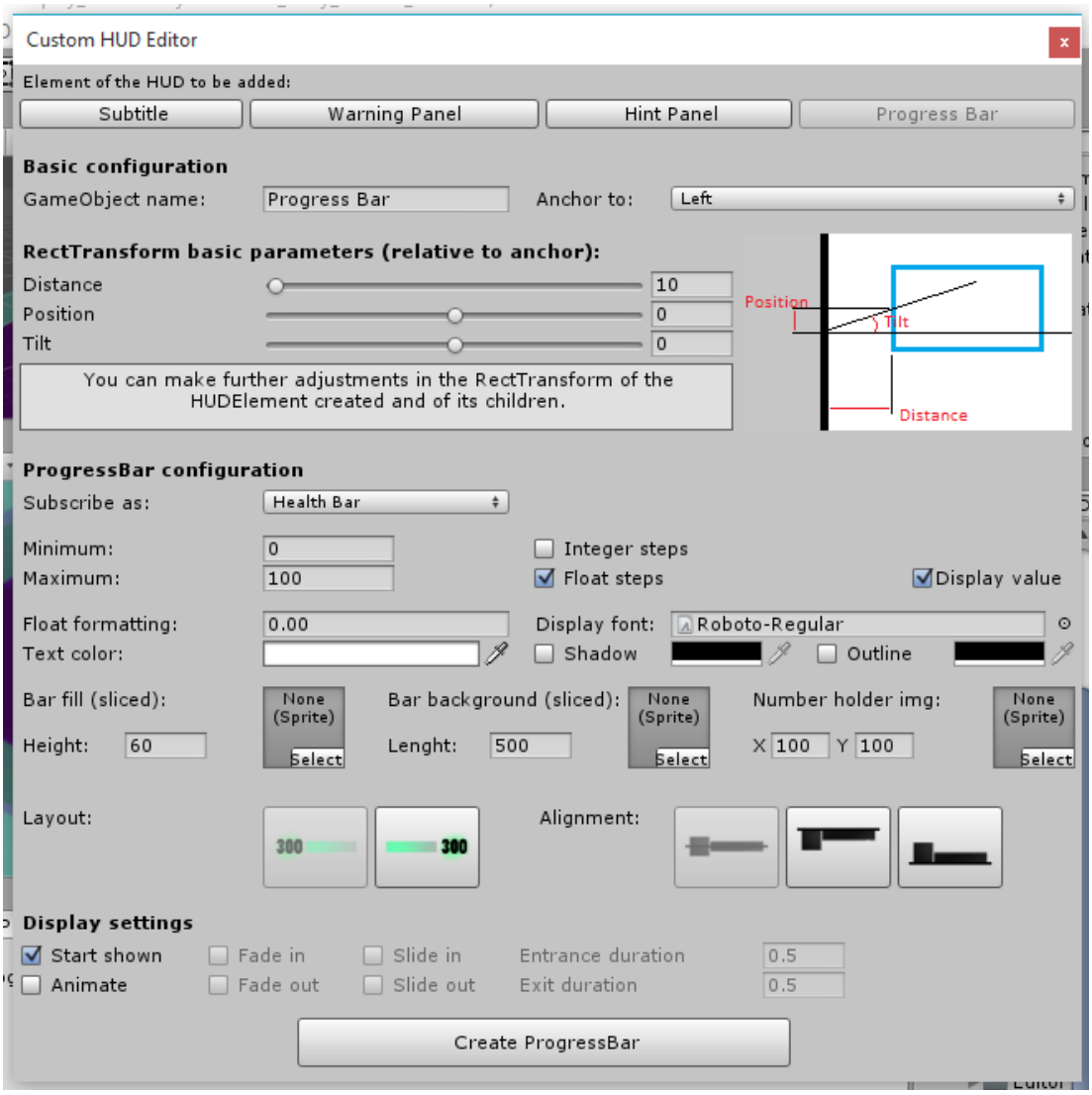

Imagen 29: Ventana de edición de HUD terminada

6.2.2. *Prefabs* de los elementos de interfaz

En este apartado nos limitaremos a presentar cómo fueron construidos los cuatro elementos que se proporcionan a modo de *prefabs* junto a la extensión para ser añadidos a la *HUD*. Como nota adicional, comentaremos brevemente que se ha utilizado un *script* para abstraer la carga de dichos *prefabs* desde el

disco duro. Se trata del fichero *HUDResourcesLoaderUtility.cs*, que implementa una serie de llamadas de acceso a los recursos del proyecto (*Unity* provee un rápido acceso desde código a todo elemento ubicado bajo una carpeta que se llame *Resources*, mediante los métodos *Resources.Load* y *Resources.LoadAll*).

En la viniente enumeración, se hará referencia a componentes predefinidos de la nueva interfaz de *Unity* y a unos pocos desarrollados en este proyecto, los cuales fueron presentados en el diagrama de clases de la [Imagen 22] y serán detallados en el apartado siguiente. Además, se presupone la posesión de un componente *RectTransform* en todos los objetos de juego enumerados, pues es obligatoria su existencia en todo elemento de la nueva interfaz.

Los elementos de interfaz y sus estructuras de objetos, en el mismo orden en que se presentaron en el apartado 6.1.2, son:

- Subtítulos: un *GameObject* con el componente *HUDSubtitle* y, como hijo:
    - Un objeto con un componente *Text*, que permite el renderizado de un texto en una caja.
        - A este *Text* se le añadirán los componentes *Shadow* y *Outline* si el desarrollador así lo ha querido.

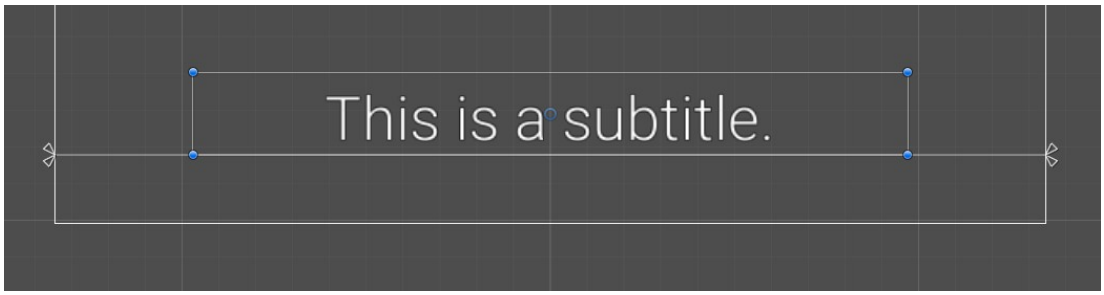

Imagen 30: Ejemplo de HUDSubtitle

- Panel de alerta: objeto con el componente *HUDWarningPanel*, y:
    - Un objeto con el componente *Image* para renderizar la imagen de fondo del panel de alerta y el componente *HorizontalLayoutGroup*, que permitirá mantener a los hijos de este siempre organizados y espaciados. Tiene como hijos dos objetos:
        - Uno con el componente *Image* para el icono de la advertencia y

- Otro con el componente *Text* para el mensaje.

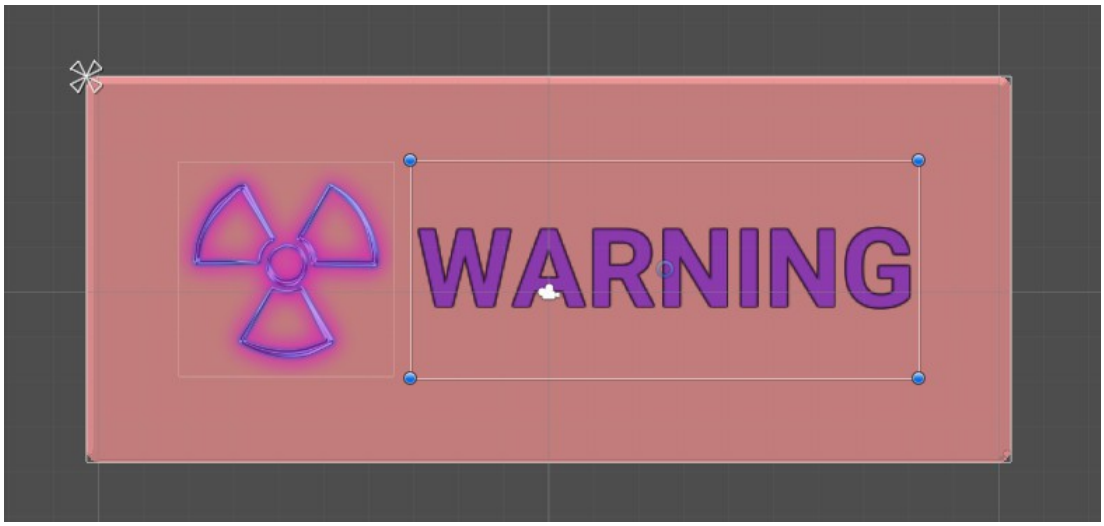

Imagen 31: Ejemplo de HUDWarningPanel

- Panel de indicio: similar al anterior, pero con el componente *HUDHintPanel* y con un compontente *Animator* que maneja las animaciones de entrada y salida y, como hijos tiene:
  - Un *HorizontalOrVerticalLayout*, que ordena los dos objetos bajo él:
    - Uno con una *Image* para el icono del indicio.
    - Y otro *GameObject* con un componente *Text* para el mensaje de ayuda o indicación.

Imagen 32: Ejemplo de HUDHintPanel

- Barra de progreso: utiliza el componente *HUDProgressBar* y un *Animator* para animar sus entradas y salidas. Su estructura de *GameObjects* es más compleja:

- Un *GameObject* con un *HorizontalLayoutGroup*, que organiza:
  - Un objeto que contendrá el indicador numérico del valor de la barra, con un *Canvas* que permite superponerlo a otros elementos, incluso a la propia barra, si el desarrollador lo desea.
    - Bajo él, un *GameObject* con un *Image* para mostrar un fondo sobre el cual irá ubicado el indicador numérico, que puede utilizarse o dejarse desactivado.
      - Y como hijo de este, uno con el componente *Text* que muestre el valor de la barra de progreso.
  - Un segundo objeto con el componente *Slider* (que permite elegir un valor numérico entre dos límites) con la interacción deshabilitada, tal que su única función sea la de mostrar un valor determinado. En el mismo objeto hay un componente *Image* para representar el fondo de la barra.
    - Como hijo tiene un objeto con un componente *Image*, que representará la porción de barra llena. Se expande sobre la imagen de fondo a lo largo de ella, creando así la sensación de rellenado.

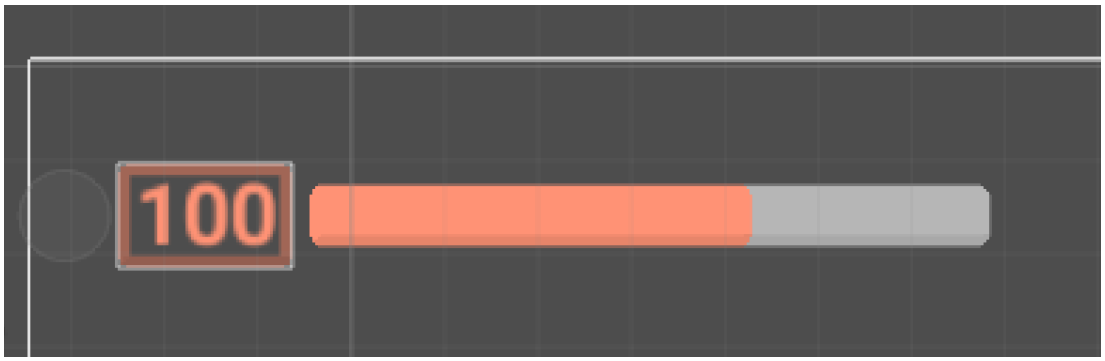

Imagen 33: Ejemplo de HUDProgressBar

Tales son, pues, las estructuras que presentan los cuatro *GameObjects* que se adjuntan a la extensión, en la carpeta *Resources/HUDPrefabs*.

6.2.3. Clases y utilización

Hasta este punto, hemos nombrado algunos de los *scripts* o componentes

que han sido desarrollados dentro de la librería de clases del proyecto. Ahora es el momento de revisar los aspectos y patrones utilizados más relevantes de las clases implementadas bajo el espacio de nombres *CustomHUD*.

En primer lugar, el uso de las *interfaces* de *C#* ha estado presente durante todo el desarrollo. Las *interfaces* definen grupos de métodos y propiedades que otras clases deberán implementar, estableciendo así un contrato tal que <<si la clase A implementa la *interface* B, A se compromete a disponer implementados todos los métodos y propiedades definidos en B>>. Gracias a ello, se han podido crear unas férreas normas alrededor de las entidades de la *HUD*, con el fin de que si cualquier otro desarrollador extiende la funcionalidad de nuestra extensión, lo haga acorde a unas normas y cumpla unos mínimos.

En el fichero *CustomHUDInterfaces.cs* se encuentra la defínción de todas las utilizadas. En ellas, se especifican las características mínimas que, por ejemplo, todo buen controlador de *HUD* debería poseer, o las propiedades que todo panel de alerta o toda barra de progreso deberían presentar. Merece mención especial la *interface IHUDShowable* y las que heredan de ella, ya que permiten al controlador de *HUD* saber si un elemento dispone de métodos propios para ser mostrado y ocultado, o simplemente debe activar y desactivar su *GameObject* para tal propósito.

Otro de los patrones popuplares que se ha utilizado es el de llamado *singleton débil*. Un *singleton* es aquella instancia de una clase, tal que ella es la única existente de su clase. Se implementa comúnmente creando una variable pública estática del mismo tipo que la clase en sí, tal que sea asignada a la instancia *singleton*. De este modo la instancia *singleton* es accesible desde fuera de la clase como cualquier otra variable de clase.

En el caso de *Unity*, lo mismo puede hacerse en un *MonoBehaviour*. Queremos que sólo exista un objeto controlador de *HUD* en la escena, por lo que tendremos una variable estática de tipo *HUDController* en su misma clase, y en el *Awake* del componente, él mismo se asignará a la variable. De este modo, el desarrollador puede acceder a la funcionalidad del *HUDController* desde cualquier punto de su código accediendo a dicha variable de clase.

El tercer patrón de desarrollo utilizado a destacar es en sí la forma de trabajar con las referencias en los componentes de *Unity*: las variables de instancia declaradas como públicas serán visibles y manipulables en el editor. Este hecho ha sido de gran utilidad a la hora de construir los *prefabs* de los elementos de *HUD*, pues el hecho de poder arrastrar un objeto de la jerarquía hasta el hueco pertinente y que quede asignado a una variable (si la variable es del tipo de un componente, al arrastrar el objeto queda asignada a ese componente, no a todo el objeto) ahorra búsquedas durante la inicialización. Como ejemplo, presentamos la asignación de los *Image* y *Text* de un *HUDHintPanel*, en objetos distintos a él [Imagen 34].

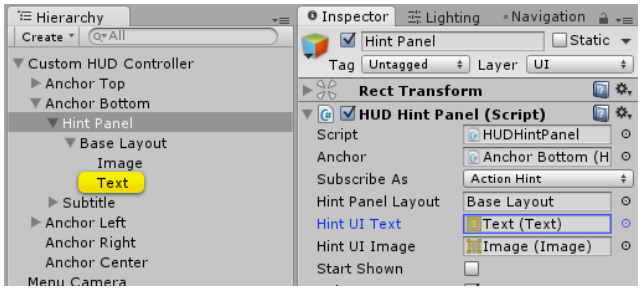

Imagen 34: Componentes asignados a un HUDHintPanel

Otras utilidades de las que se ha hecho uso puntualmente son las *coroutines*, que son métodos que permiten la suspensión de su ejecución durante un tiempo, o los *Invoke* para retrasar en el tiempo la llamada a un método determinado. Fueron de ayuda para implementar funcionalidades del tipo "mostrar un elemento de *HUD* y ocultarlo pasados unos segundos".

Por lo que respecta al resto de la implementación de los componentes, se reduce a programación orientada a objetos común y corriente, de forma jerárquica y estructurada. Se creó la clase abstracta *HUDElement*, de la cual heredan *HUDSubtitle*, *HUDWarningPanel*, *HUDHintPanel* y *HUDProgressBar*, según se diseñó. También se dispone de las clases *HUDAnchor* y *HUDController*, todas ellas resultando en componentes de *Unity*.

La última de las clases mencionadas, *HUDController*, merece especial atención, pues es la que ata todos los cabos y dota a la *HUD* de funcionalidad. De ella, destacaremos el uso de un diccionario clave-valor para gestionar el sistema de la suscripción de los elementos según roles y para administrar las llamadas recibidas por parte del código del desarrollador.

El diccionario utilizará como claves cadenas de texto o *strings*, y los valores asociados a ellas serán objetos de tipo *HUDElement*. Esta decisión fue tomada

debido a que dado que a un contolador sólo puede suscribírsele un único elemento de interfaz desempeñando un rol determinado y dichos roles están representados por valores de tipos enumerados (o por el tipo en sí, en el caso del panel de alerta), la representación como cadena de texto provista por el sistema para tales tipos resulta válida como única representación posible de un rol.

Así pues, cuando un controlador anuncia que ha sido inicializado en la escena y los elementos de *HUD* se le suscriben [ref. Imagen 23], los introduce en su diccionario de elementos suscritos, bajo la clave que representa cuál es su rol. Así, en todo momento sabe si dispone de una barra suscrita como barra de energía o de un panel de alerta.

Todas los métodos de los que dispone el *HUDController* relacionados con mostrar, ocultar o manipular elementos de forma externa, utilizan este sistema de roles. El programador podrá llamar a métodos como *ShowElement*, *HideElement* o *ShowAndHideElement* utilizando la clave que representa al rol del elemento, sin necesidad de tener una referencia al objeto en sí. Todavía hay más, sin embargo.

Se han implementado varias decenas de métodos públicos que hagan este trabajo tan intuitivo como sea posible tal que, por ejemplo, si el programador desea que se muestre un panel de indicio con un nuevo mensaje, dispondrá de una llamada *ShowHint* a la que pasarle como argumento el tipo (rol) de *Hint* al que se refiere y el mensaje que quiera mostrar. Para más comodidad si cabe, se han desdoblado estas llamadas tantas veces como roles hay, para así disponer de métodos con nombres aún más útiles: *ShowActionHint, HideReminderHint*... Y lo mismo ocurre para el resto de tipos de elementos: existen métodos tal que *ShowPrimarySubtitle*, *SetEnergyBarValue*, *SetHealthVarBalueOverTime*, *ShowAndHideWarning*, y un largo etcétera.

Ante una invocación de entre cualquiera de estas, el *HUDController* sólo tiene que buscar en su diccionario si existe un elemento suscrito con el rol por el que el programador está preguntando y, si existe, ejecutar las órdenes sobre él que se le indican [ref. Imagen 24]. Concluye así el resumen general del funcionamiento del sistema de *HUD*.

Como comentarios adicionales, merece destacar la existencia de una clase estática llamada simplemente *HUD* dentro del fichero *HUDCallsAPI.cs*, cuyo único propósito es contener llamadas estáticas para redirigirlas al *HUDController* de la escena, tal que utilizar por ejemplo *HUD.ShowWarning* en cualquier punto del código tenga el resultado esperado. También es digno de mención el archivo *HUDEvents.cs*, donde se encuentran todos los eventos que el controlador lanza. El desarrollador puede suscribirse a cualquiera de ellos si lo requiere, o puede extender el abanico de los existentes para dotarlo de nuevas funcionalidades específicas.

7. Desarrollo del videojuego

Como parte final del trabajo de este proyecto, se desarrollará un juego acorde con lo propuesto en los objetivos del proyecto [Apartado 2.3]. Será creado desde cero y enteramente en *Unity*. Aunque el objetivo de su desarrollo sea la demostración del uso de una *HUD* modular en él, creada con la extensión que se ha implementado, se pretenderá crear un videojuego completo hábil de ser presentado al público.

Primeramente, en una etapa de diseño del juego, se expondrán los aspectos más relevantes acerca de su jugabilidad y la interfaz de usuario que se planea construir. Después se darán detalles de su implementación, centrando la atención especialmente en los patrones seguidos y prácticas más comunes del desarrollo en *Unity* utilizadas.

En referencia a la estética del juego, se dedicará a ella un breve punto al final de este apartado, ya que es un aspecto más bien relacionado con el resultado final y presentación del videojuego que no con su desarrollo.

7.1. Diseño

El proceso de diseño de un videojuego comprende numerosos aspectos, desde su guionización, la toma de decisiones acerca de sus mecánicas, su estilo artístico, el desarrollo de los personajes, etcétera. Dado que nos hallamos ante un juego realmente sencillo de género *arcade*, la mayor parte de la atención recaerá en sus mecánicas: cómo van a desenvolverse los niveles del juego y los elementos que hay en ellos, cúales son los objetivos a cumplir y cuáles son las acciones que el jugador puede realizar. Además, se hará un hueco para detallar el diseño de las interfaces en la partida que se van a utilizar.

7.1.1. Mecánicas

Como hemos dicho, se tratará de un sencillo juego de disparos tridimensional. El jugador podrá moverse a lo largo y ancho de una superfície limitada, seguido por una cámara aérea. Además, podrá girar sobre sí mismo

para orientarse y apuntar y disparar proyectiles hacia donde esté orientado. El control del juego, pues, se compone de tres entradas:

- Palanca analógica o botones de dirección: desplazarse por el área.
    - Este control normalmente se asocia al *stick* izquierdo de un mando de juego o a las cuatro teclas con flechas de un teclado.
- Palanca analógica (eje horizontal) o botones: rotar en sentido horario o antihorario para orientarse.
    - Bastará con dos botones para girar en un sentido u otro, o con inclinar un *stick* de un mando, comúnmente el derecho, hacia los lados.
- Gatillo o botón: disparar un proyectil en la dirección apuntada.

En cuanto a la forma en la que transcurrirá el juego, se planea que los niveles tengan forma de oleadas, donde un número determinado de enemigos ataque a la vez, y cuando sean derrotados, se lance otra oleada. Se prevé la creación de diez oleadas para una primera versión del juego. La primera vez que se ejecute la aplicación se mostrará un tutorial interactivo, donde una voz en *off* dará indicaciones, para luego pasar directamente a los niveles. El resto de veces, se mostrará un menú principal desde el cual lanzar las oleadas o volver a probar el tutorial.

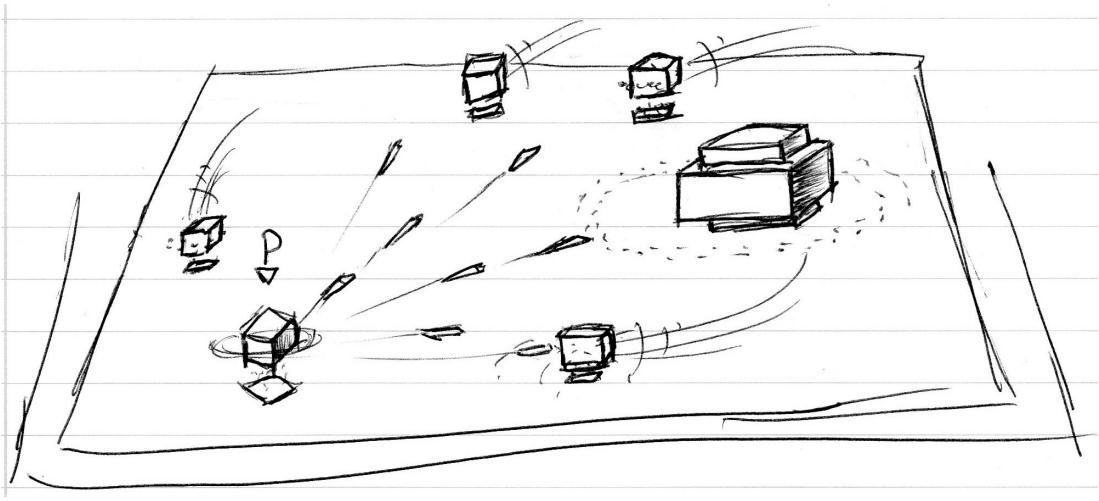

Imagen 35: Boceto concepto de una oleada de enemigos

En materia de enemigos, la versión que se presentará del juego contará con dos tipos diferentes de ellos. Uno más básico y débil (llamado "*Basic Enemy*"),

que se limitará a perseguir al jugador hasta chocar con él y causarle daño, y uno más grueso, lento y resistente ("*Enforcer Enemy*"), que producirá una onda expansiva cuando el jugador esté cerca para dañarlo y empujarlo. Por otro lado, sólo se dispondrá de un tipo de proyectil, que volará en línea recta y podrá ser disparado repetidamente.

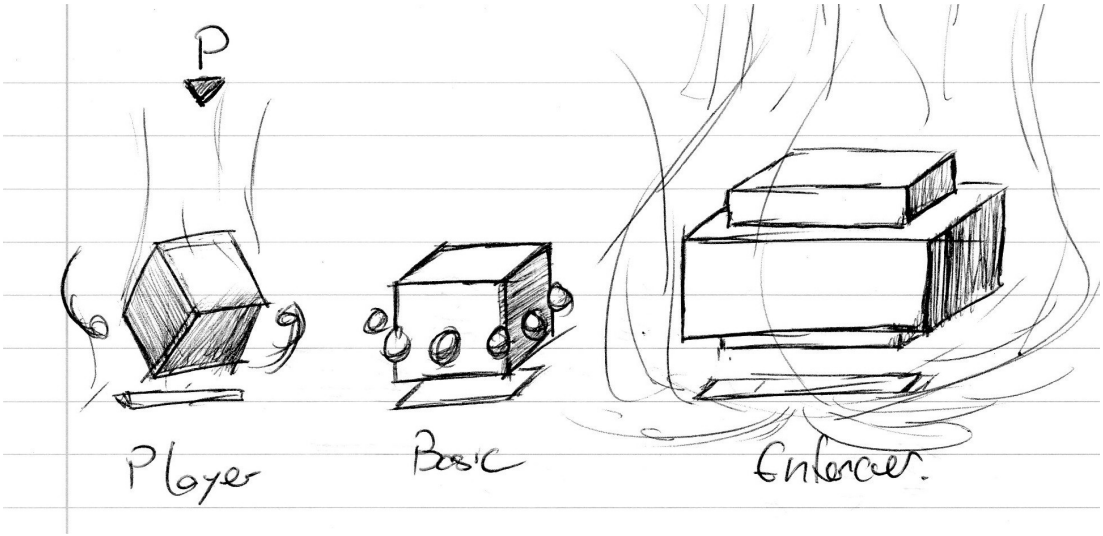

Imagen 36: Concepto original del jugador y los dos enemigos

El juego poseerá un sencillo sistema numérico para representar la salud restante de los enemigos y del jugador, así como para cuantificar el daño recibido. Una unidad de daño sustrae una unidad de salud del agente que lo sufre. Un primer diseño en este aspecto será el siguiente:

- El jugador tendrá una cantidad de salud inicial igual a 100.
    - El daño infligido por el jugador al contacto con su cuerpo es 5.
- Los proyectiles básicos del jugador hacen un daño igual a 2.
- Los *Basic Enemies* comienzan con 5 de salud.
    - Producen 10 de daño al contacto con el jugador.
- Los *Enforcer Enemy* tendrán una salud inicial igual a 24.
    - Infligen 15 de daño al contacto.
    - Su onda expansiva produce 30 de daño al jugador si lo alcanza.

El menú principal del juego permitirá iniciar una partida desde la primera

oleada, volver a jugar el tutorial, salir de la aplicación o personalizar algunas opciones. Dichas opciones no estarán disponibles para la primera versión de demostración del videojuego, pero siempre hay lugar para añadirlas. Se realizará con un sencillo *Canvas* de la nueva interfaz de *Unity* y cuatro cajas de texto que permitan hacer clic en ellas gracias a un componente *Button*.

7.1.2. Interfaces en la partida

El videojuego desarrollado contará con un sistema de *HUD* en todo momento, aunque no siempre será visible. Será construida con la extensión desarrollada y utilizará una instancia de cada uno de los cuatro tipos de elementos de interfaz predefinidos de esta:

- Una barra de progreso para mostrar la cantidad de salud restante del jugador.
    - Se actualizará con cada golpe que reciba.
- Un panel de alerta que anunciará la proximidad de nuevas oleadas de enemigos.
- Unos subtítulos que serán utilizados para transcribir las frases de la voz en *off* que habla durante el tutorial.
- Un panel de indicio que muestre al jugador qué controles utilizar para que aprenda a jugar en el tutorial.

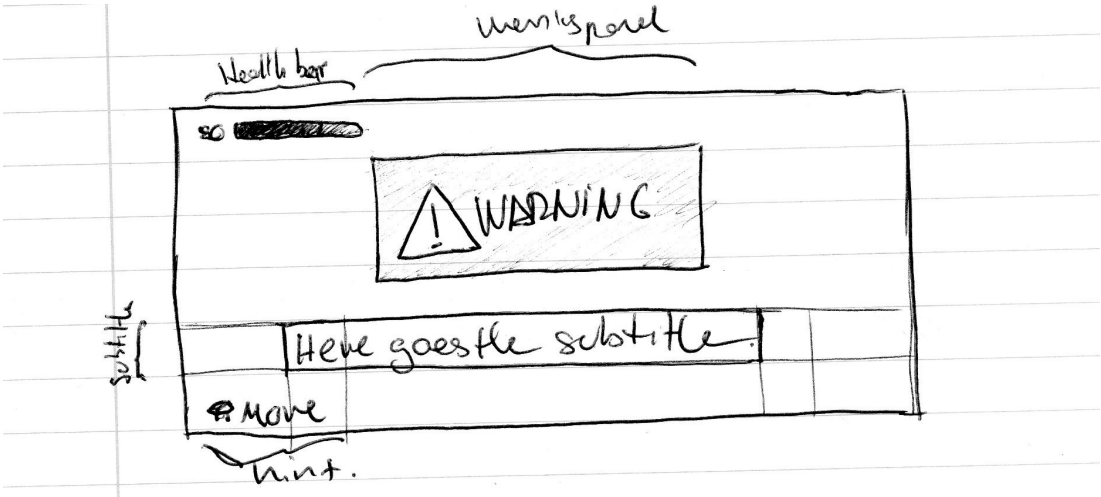

Imagen 37: Boceto de la HUD diseñada para el juego

Una vez distribuidos y, si hiciera falta, editados estos elementos, simplemente

deberemos colocar en el código del juego las llamadas oportunas a la *API* del espacio de nombres *CustomHUD* para que cumplan con su función.

Terminaremos este apartado presentando algunos *sprites* segmentados (se estiran sin deformar sus esquinas) que se diseñaron para la *HUD* del juego.

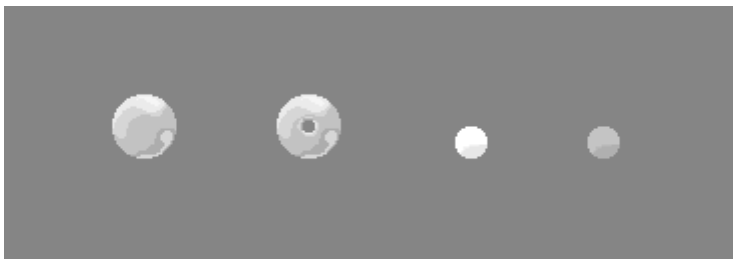

Imagen 38: Sprites diseñados para la HUD del juego

7.2. Implementación

En este último apartado daremos un vistazo a los aspectos más relevantes del proceso de desarrollo del juego. Primero acerca del jugador y los enemigos de forma aislada, repasando cómo se han creado a nivel de apariencia y de funcionamiento, y después, veremos cómo se integran con el resto de elementos de la partida y cómo se desarrolla esta. Finalmente, comentaremos la implementación de las llamadas a la *HUD* que se han utilizado para su uso dentro del juego.

7.2.1. El jugador

Respecto al *GameObject* que representa al jugador hay que destacar que posee un componente *Box Collider* para la detección de las colisiones contra los enemigos, que determina el volumen real que ocupa en el espacio (independientemente de su aspecto visual), y un componente *Rigidbody* para la aplicación de fuerzas sobre él.

Dispone de dos componentes para la administración de su lógica y control:

- *PlayerController.cs*: controla aspectos como la vida restante del jugador, y efectos sobre él como la inmovilización temporal, el desenfoque de cámara o la pérdida de color a medida que decrementa su salud.
- *PlayerMovement.cs*: es un script dedicado expresamente al manejo de las entradas del jugador, que se traducen en translaciones de su cuerpo,

rotaciones de su punto de disparo alrededor del cuerpo o en la emisión de proyectiles.

- Las translaciones y rotaciones se realizan con los métodos *Translate* y *Rotate* de los *Transform* pertinentes.
- La clase *Input* de *Unity* permite el muestreo de los controles con métodos como *GetAxis* o *GetButton*. Este muestreo se realiza dentro del cuerpo del método *Update*, una vez por fotograma.
- Maneja el movimiento de la cámara que sigue al jugador, interpolando con suavidad su posición hacia la del jugador, en *LateUpdate*.

Puede observarse una relación de los atributos utilizados para incidir en estos aspectos en el diagrama de clases de la [Imagen 40].

Para disparar, el jugador mantiene una referencia a la *pool* de proyectiles que esté utilizando en cada instante (veremos el concepto de *pool* y su utilización más adelante, en el apartado 7.2.3) y ordenará el lanzamiento de uno de ellos en su punto de disparo cuando las condiciones lo permitan y el botón de disparo esté pulsado. Por ahora, sólo se dispone de un único proyectil, el básico, que vuela en línea recta mientras no choque con un enemigo.

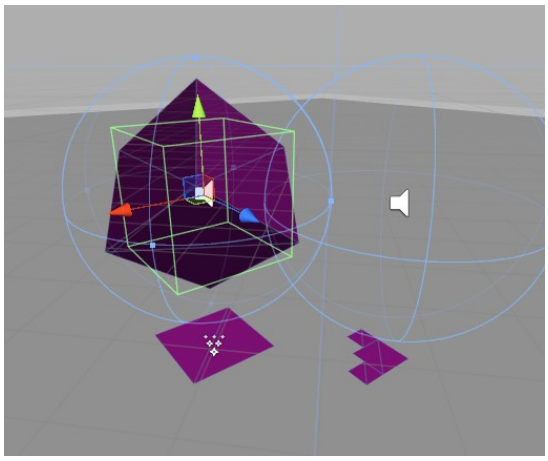

Imagen 39: Objeto jugador en la vista de escena

En cuanto a la apariencia del jugador, se ha optado por un cubo en constante rotación acompañado de dos cintas que giran a su vez y dejan un rastro por donde el jugador pasa. A modo de base y como ayuda para el jugador, dispone de un plano cuadrado sobre el suelo y, en una de sus esquinas una sencilla flecha que indica la dirección de disparo. Dispone de dos *AudioSources* para emitir sonidos. Uno de ellos en su cuerpo, para emitir un sonido al ser derrotado, y otro en el punto de disparo para producir un sonido cuando se lanza un proyectil.

7.2.2. Los enemigos

De forma similar al jugador, los enemigos disponen de dos scripts que manejan su lógica, comportamiento y movimiento, pero siendo este último automático. Cada tipo de enemigo dispondrá de sus dos componentes, heredados de dos abstractos, que son:

- *Enemy.cs*: contiene los atributos y métodos básicos para controlar aspectos como su preparación para atacar al jugador, su vida restante o los efectos aspectuales al recibir daño o morir, si los hubiera.
- *EnemyMovement.cs*: contiene tan solo los atributos utilizados para un movimiento constante hacia un objetivo.
    - El objetivo del movimiento de un enemigo viene dado por un *Transform* cualquiera existente en la escena, que por defecto será el del jugador.

Para el *Basic Enemy*, se ha optado por un cubo rotatorio paralelo al suelo de un tamaño similar al del jugador, rodeado por un cilindro. Bajo él tiene un cuadrado que indica su posición sobre el suelo, y dispone de tres sistemas de partículas que cumplen funciones diferentes.

Los sistemas de partículas, mediante el componenete *ParticleSystem* de *Unity*, permiten la emisión de texturas coloreadas en puntos la escena según diferentes patrones. Este enemigo utiliza uno constante bajo su cuerpo para ilustrar su flotación, otro en forma de anillo para indicar que ha sido golpeado y un tercero más explosivo cuando es derrotado.

En cuanto a su comportamiento, se lanzarán a perseguir al jugador en cuanto aparezca en la escena a una velocidad base similar a la del movimiento del jugador. Cuando son golpeados, se detienen unos instantes antes de reanudar la persecución. Al inicio, su velocidad y su aceleración serán modificadas por un factor aleatorio entre el -5% y 5% para aportar algo de variedad.

En los *scripts BasicEnemy.cs* y *BasicEnemyMovement.cs*, que heredan de los explicados anteriormente, implementan estas características específicas. Cabe destacar cómo funciona la orientación del enemigo hacia el jugador. Después de

que tanto los enemigos como el jugador se hayan transladado en *Update*, en *LateUpdate* cada enemigo actualiza un *Transform* auxiliar del que dispone mediante el método *LookAt*, que lo orienta hacia el jugador. Será en la dirección resultante de este vector en la cual se desplace en el siguiente *Update*.

Para el *Enforcer Enemy,* se ha creado un cubo tal que cuatro veces mayor a los otros agentes, rodeado por tres anillos circulares. Presenta un anillo a su alrededor formado por partículas que marca el área donde detecta la presencia del jugador. Dispone, además de otros tres sistemas de partículas que cumplen las mismas funciones que los del *Basic Enemy*.

Su movimiento es mucho más lento que el del resto, pero es imposible detener su avance. Cuando el jugador entra en su área de acción, el aspecto de las partículas del anillo cambia a uno más agresivo, para después emitir una onda expansiva que daña y empuja al jugador, acompañado de una oleada esférica de partículas. El empuje se ha implementado con el método *AddExplosionForce* sobre el *Rigidbody* del jugador, con centro en la posición del enemigo.

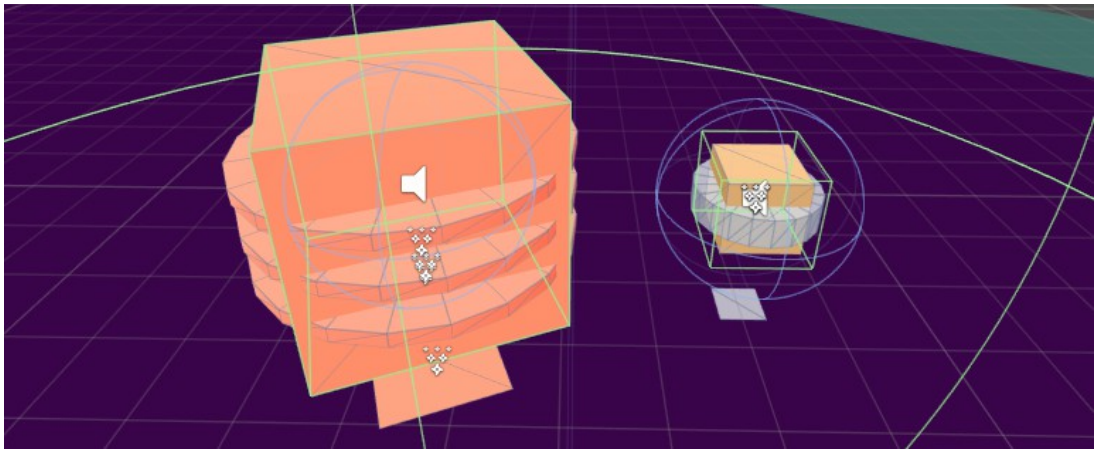

Imagen 40: Estructura de un Enforcer Enemy y un Basic Enemy

Los enemigos también disponen de efectos de sonido, normalmente reproducidos al recibir un golpe o ser derrotados mediante el método *PlayOneShot*.

7.2.3. La partida

La situación de la escena y lo que ocurre en ella es manejado en todo momento por el objeto *SceneController*, cuyos componentes (*scripts* de *C#*)

desempeñan funciones clave para el desarrollo de la partida. Estos son:

- *SceneController.cs*: controla todas las partes de la ejecución y administra la lógica del juego.
    - Carga al inicio, de dos archivos *XML*, la información referente a cómo serán las oleadas y cuáles son los pasos del tutorial.
    - Muestra el tutorial o el menú principal dependiendo de si el usuario ya ha jugado al juego anteriormente.
    - Tiene una referencia al jugador y le permite o denega el control dependiendo de la situación.
    - Dispone de llamadas para ubicar en la escena enemigos.
        - Permite pasar un método como *callback* cuando los enemigos sean derrotados (un *callback* es una llamada a procedimiento dada para que se llame en un momento futuro).
    - Controla la lógica de sustracción de salud a los agentes del juego cuando haya colisiones entre ellos, contra proyectiles o por efecto de una onda expanisva.
        - Se encarga de llamar a las funciones pertinentes de los agentes para que realicen las acciones debidas cuando toman daño o mueren.
    - Administra si la cámara del jugador debe seguirle o debe ubicarse en un punto fijo, así como de colocarla si fuera necesario.
    - Maneja el cierre y reinicio de la aplicación.
- *LevelController.cs*: contiene todo el ciclo de niveles u oleadas de la partida.
    - Contiene una lista de *LevelItems* (clase definida en *LevelItem.cs*), construida a partir del *XML* donde se definen las oleadas.
    - Se encarga de pedir cuántos enemigos de cada tipo deben lanzarse contra el jugador, con un tiempo aleatorio de aparición para cada uno.

        - Utiliza una estructura de datos para llevar el recuento de cuántos de estos enemigos quedan vivos.
        - Decrementa estas cantidades según recibe los *callbacks* al ser derrotados.
        - Cuando ya no queda ninguno, se pasa a la siguiente oleada.
- *TutorialController.cs*: controla el paso del jugador por todos las etapas del tutorial.
    - Utiliza una lista de *TutorialSteps*, resultado de la lectura del *XML* al inicio de la aplicación.
    - Muestrea las acciones del jugador con la clase *Input* para cerciorarse de que cumple los objetivos (moverse, girar y disparar) antes de pasar a la siguiente etapa.
    - Utiliza un subtítulo para comunicarse con el jugador y un panel de indicio para mostrar cómo ejecutar la acción que se le pide.

Un punto interesante de la implementación del juego, que anticipábamos en el apartado 7.2.1, es el uso del patrón de desarrollo de las *pools* de objetos.

Podría entenderse como una reserva de, en el caso de *Unity*, *GameObjects* de los cuales se van a requerir muchas instancias iguales a lo largo de la partida. La instanciación y destrucción de objetos de forma constante durante la ejecución tiene una sobrecarga notable, pues requiere de un alojamiento dinámico en memoria y de una limpieza de objetos sin referencia constante, generando normalmente fragmentación interna de la memoria y, por lo general, una caída del rendimiento.

Para nuestro juego, se ha optado por una *pool* de objetos de implementación más que sencilla para administrar todos los objetos que deban ser reciclables. Se dispondrá de un número determinado de copias de ellos ya preparados en la escena y desactivados, como hijos de un *GameObject* que administre la *pool*. Este administrador se encargará de activar y preparar los objetos según le sean encargados. Cuando un objeto ha cumplido su cometido, se <<recicla>>

volviendo a estar disponible en la *pool*. En nuestro caso, el reciclaje pasará por sencillamente desactivar el objeto en la escena. El impacto en el rendimiento de activar y desactivar unos mismos *GameObjects* constantemente es despreciable en comparación al de instanciarlos y destruirlos.

El funcionamiento es realmente simple: los objetos que deban ser lanzados y reciclados dispondrán de unos métodos para ellos, que deberán ser llamados cuando sea pertinente para asegurar el bueno uso de los recursos. Se han definido dos *pools* abstractas y objetos para ellas igualmente abstractos, que luego se especializarán en tantas clases que hereden de ellas como haga falta.

La primera de ellas es para los proyectiles. Se definen las clases abstractas:

- *ProjectilePool*: rellena una lista de proyectiles a partir de todos los componentes *Projectile* que tengan sus *GameObjects* hijos, y la administra.
    - Tiene el método abstracto *InitProjectile*, que se encargará de preparar un proyectil en la ubicación de un *Transform* dado.
    - Con el método *RecycleProjectile* ordena la desactivación de un proyectil dado.
- *Projectile*: contiene una referencia a su *pool* y los cuatro métodos básicos para su administración, que son:
    - *Ready*, que lo ubica y orienta en la posición y sentido de un *Transform*.
    - *Launch*, que lo activa en la escena y le activa el movimiento.
    - *Stop*, que lo detiene y desactiva (normalmente ordenado por la *pool*).
    - *RecycleMe*, que notifica a la *pool* que ha cumplido su función y debe ser reciclado.

Dado el caso de que una *pool* de proyectiles tuviese que lanzar uno que estuviera en ese instante activo en la escena (que no haya sido reciclado), simplemente llamaría a *Ready* y *Launch* como si nada pasase, ubicando el proyectil donde deba ser disparado y continuando sin problema alguno. En

relación con este aspecto, se debe jugar con cuántas copias tener instanciadas, para asegurarse que son las necesarias pero no excesivas.

La segunda de las *pools* es para enemigos, y su funcionamiento es prácticamente idéntico al de la anterior. Se compone de:

- *EnemyPool*, con los métodos *InitEnemy* y *StopEnemy*.
    - Contiene además el método *PlayerHasDied*, para notificar a las unidades enemigas que administra que dejen de perseguirlo porque ha muerto.
- *Enemy*, que tiene los métodos *Ready*, *Spawn*, *Stop* y *RecycleMe*.

Cabe comentar, además, que en las *pools* se han definido algunos atributos propios de los objetos que contienen y no de ellas mismas, pero al ser comunes a todos los elementos en ella y estos tienen una referencia a su *pool*, se ha preferido centralizarlos en un solo sitio con el fin de evitar redundancias innecesarias.

Accesoriamente, para mayor comodidad y mejor organización, se ha creado un componente *PoolController*, que es añadido al *GameObject* padre de todas las *pools* del juego y mantiene referencias a todas ellas, que son accesibles fácilmente gracias al uso del patrón singleton en *PoolController*. De este modo, por ejemplo, el controlador de escena tiene un acceso fácil a los enemigos para ordenar que se ubiquen en el mapa.

Las especializaciones de las clases abstractas que se han implementado para el juego puede observarse en detalle en el diagrama de clases presentado [Imagen 41], que resume la estructuración del conjunto de los *scripts* desarrollados para el videojuego.

Se dispone de la clase estática *PrefController*, que sencillamente aporta una forma rápida de acceder a las preferencias persistentes del jugador proporcionadas por la clase de *Unity PlayerPrefs*, y del componente *MenuController*, que se encargará de la administración de los objetos de interfaz que componen el menú principal del juego.

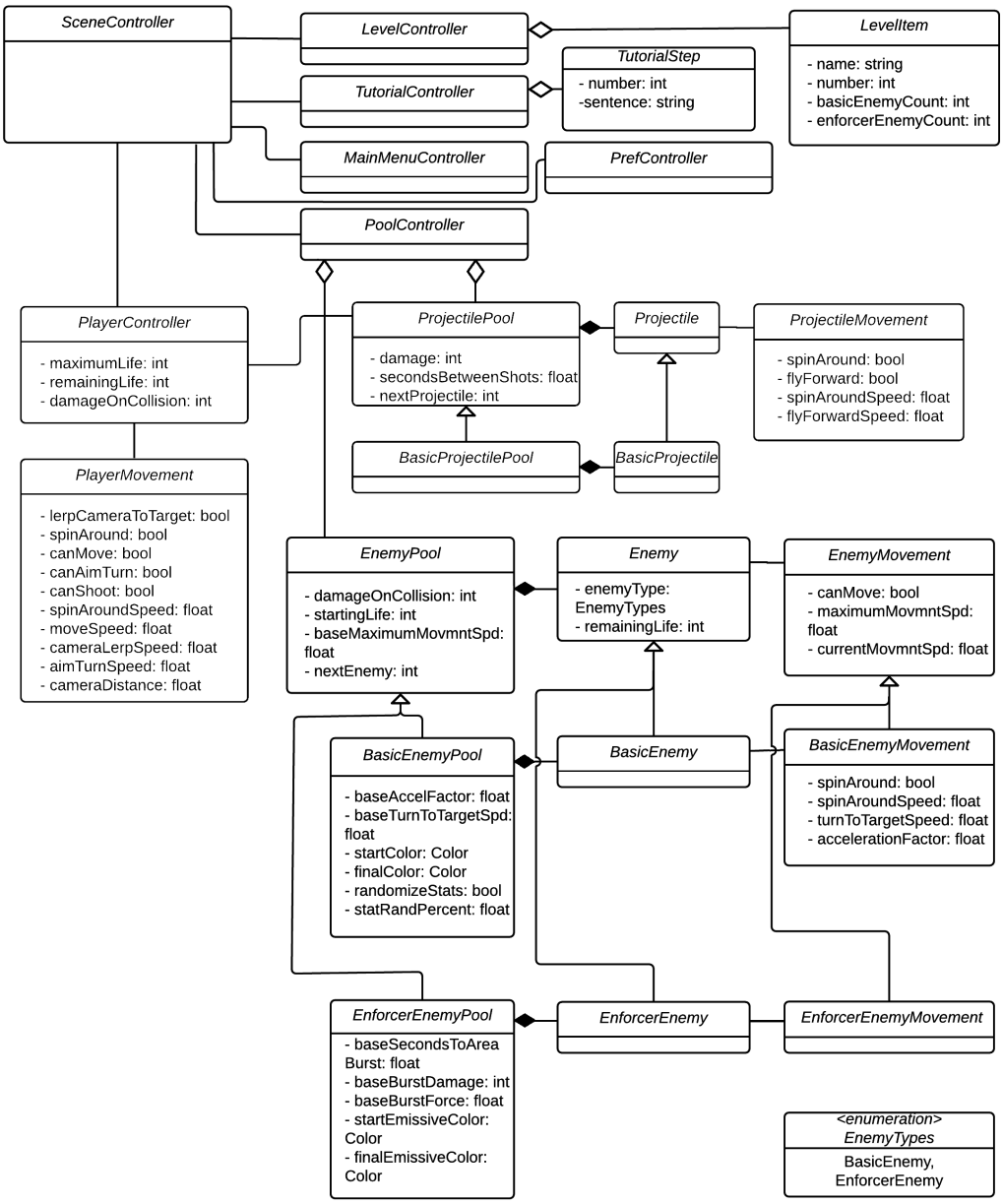

Imagen 41: Diagarma de clases seguido en el desarrollo del videojuego

Como comentario final de este apartado y complemento al diagrama [Imagen 41], cabe destacar que en las estas clases existen atributos públicos no reflejados en el diagrama, principalmente porque son simplemente utilizados para pasar referencias a *GameObjects* o componentes de la escena, y o bien no tienen impacto alguno en el diseño lógico del juego o son inferibles desde las relaciones marcadas.

7.2.4. Las interfaces

Para terminar el juego, se añadirá un nuevo *HUDController* a la escena, y a él

se anclarán los elementos de interfaz comentados en el apartado 7.1.2. No hay necesidad de entrar en demasiados detalles acerca de cómo se dispusieron, pues la ventana de edición de *HUD* lo hace todo por nosotros. Sólo ha sido necesario seleccionar los *sprites*y fuentes adecuados para cada uno. La selección de tintes de color para algunos elementos se hará a posteriori, cuando se decida una estética general para el juego.

Presentamos primeramente una visión general del resultado con los elementos ya dispuestos [Imagen 42]. No presentaremos imágenes separadas de cada elemento, pues en las imágenes [Imagen 30], [Imagen 31], [Imagen 32] y [Imagen 33] ya se muestran, a modo de ejemplo, las versiones primitivas de ellos.

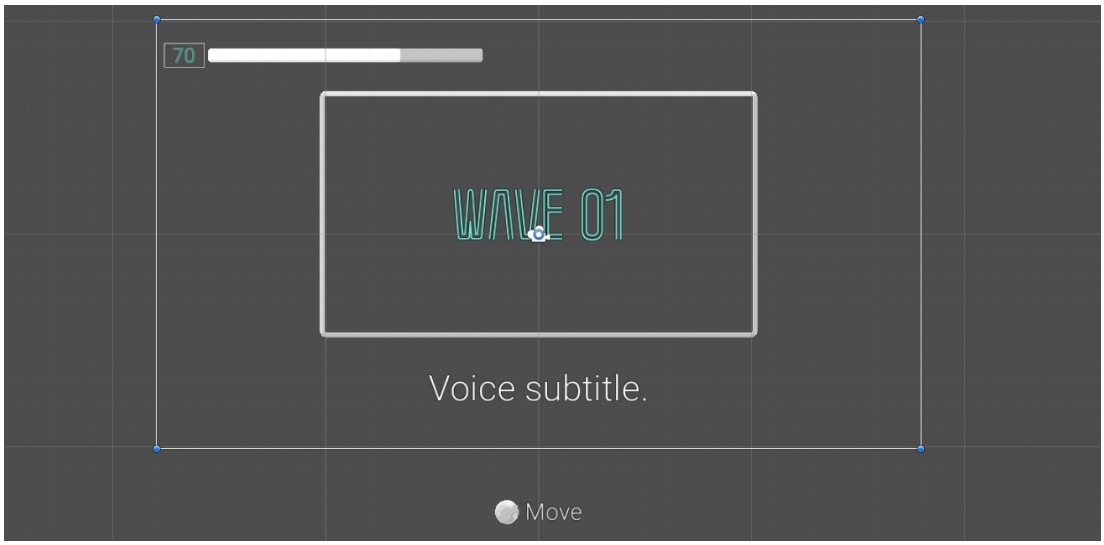

Imagen 42: HUD del juego en la vista de escena

Para dotarla de funcionalidad, sencillamente ubicaremos las siguientes líneas a, que llaman a métodos de la clase estática *HUD* en los lugares pertinentes:

- Para la barra de progreso (suscrita como barra de salud):
  - *HUD.ShowHealthBar* en el método *StartLevels* de *SceneController.cs*, que hace la barra visible al iniciar las oleadas.
  - *HUD.SetHealthBarValueOverTime* tanto en *EnemyCollisionPlayer* como en *EnforcerBurstPlayer* de *SceneController.cs*, para decrementar la barra de vida progresivamente hasta un valor deterimado cuando el jugador ha sido dañado.
  - *HUD.HideHealthBar* en el método *PlayerHasDied* del mismo *script*,

cuando la partida ha terminado.

  - La barra dispone de animaciones de fundido cuando se muestra y se oculta.

- Para el panel de alerta, sencillamente usaremos:

  - *HUD.HideAndShowWarning* con la frase necesaria. La frase será el nombre de la oleada en el método *CoroutineSetUpLevel* del fichero *LevelControler.cs* y un mensaje de victoria en el método *CheckLevelBeaten* del mismo *script*, para cuando el jugador haya superado todas las oleadas.

  - Utiliza animaciones de fundido al ser mostrada u ocultada.

- El panel de indicio (como indicio de acción) será utilizado en los pasos del tutorial, en el script *TutorialController*:

  - *HUD.ShowActionHint* con el texto de ayuda y el *sprite* del botón a pulsar para indicar al jugador que realice una acción.

  - *HUD.HideActionHint* cuando la ha realizado satisfactoriamente.

  - Su entrada y salida están animadas con deslizamientos desde el borde de la pantalla.

- El subtítulo (como subtítulo primario) se utiliza también en cada paso del tutorial, para comunicarse con el jugador a la vez que la voz en *off*:

  - *HUD.ShowAndHidePrimarySubtitle* con la frase a presentar y el tiempo de estancia en pantalla, normalmente de un segundo.

  - También tiene su entrada y salida animadas con fundidos.

El funcionamiento de estas llamadas es el esperado, siempre y cuando el controlador de la *HUD* esté activo en la escena.

7.3. Estética

En este último punto, veremos algunos detalles acerca del apartado visual del videojuego, como puedan ser el conjunto de colores elegido, el diseño final de los

enemigos y sus animaciones o los efectos de cámara.

Se ha optado por un juego de colores que oscila entre el morado y el verde para todo el entorno e interfaces. Para el jugador y sus proyectiles, colores más fuertes pero en sintonía con ellos, hacia el fucsia. Por contra, para los enemigos se ha elegido una gama de naranjas que contrasten con el entorno. Para los elementos de *HUD* se ha utilizado la familia de fuentes *Roboto*, y para el panel de alerta, la fuente *Marquee Moon* [Imagen 42]. Para el menú principal, en cambio, se ha utilizado la fuente *Planet Opti* [Imagen 43].

Imagen 43: Menú principal del juego

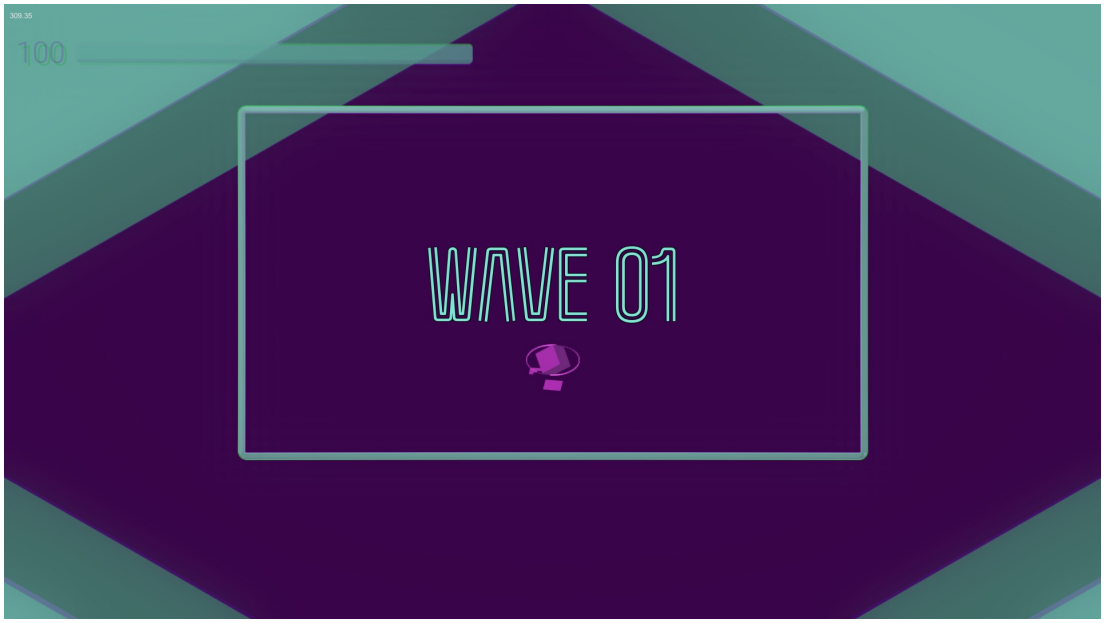

Imagen 44: Apariencia de la HUD dentro del juego

El jugador va perdiendo la opacidad de su color según decrece su vida y emite un sonido al ser vencido. Las dos cintas, producto del componente *TrailRenderer* de *Unity*, siguen su movimiento en todo instante. Los proyectiles, por su lado, no son más que un cubo deformado a lo largo que rota sobre sí mismo y emite una partícula en su cola que cae con la gravedad [Imagen 45].

Imagen 45: Jugador en movimiento disparando a un enemigo

Una serie de animaciones para los enemigos ha sido desarrollada para cuando aparecen, son golpeados o son derrotados [Imagen 46], que incluyen desde el uso de sistemas de partículas hasta la alteración de sus dimensiones o posición simulando un crecimiento o un empequeñecimiento; y suelen ir acompañadas de sonidos. Además, su color va cambiando según son dañados [Imagen 47].

Imagen 46: Enemigo básico en movimiento, siendo golpeado y derrotado

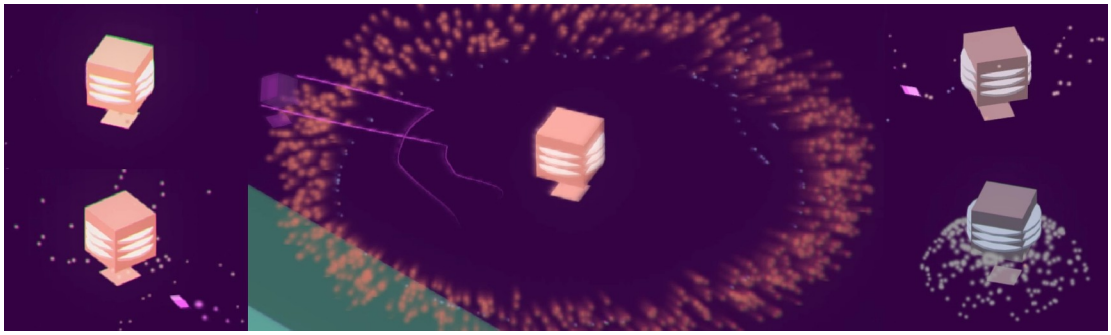

Imagen 47: Enemigo Enforcer atacando y siendo derrotado

Finalmente, para mejorar la ambientación del juego, se ha recurrido al uso de algunos efectos de cámara que *Unity* proporciona como componentes dentro del

paquete de utilidades por defecto *UnityStandarAssets.ImageEffects*:

- *VignetteAndChromaticAberration*: simula efectos propios de la fotografía en la imagen del juego.
    - El viñetado reduce el brillo de la imagen hacia los extremos de la imagen, resultando en unas esquinas más oscuras y emborronadas.
    - La aberración cromática simula un efecto que aparece comúnmente en lentes de baja calidad, donde los canales rojo, verde y azul de la luz se descompasan entre ellos y dan lugar a bordes distorsionados o a la aparición de contornos duplicados de las figuras en colores verdes o morados. Suele ser más intenso hacia los bordes de la imagen.
- *AntiAliasing*: permite la suavización de las imágenes renderizadas, reduciendo bordes duros y dentados mediante técnicas variadas.
- *MotionBlur*: produce desenfoque en la imagen cuando hay movimiento en ella. En el juego se utiliza cuando el jugador es atontado por la onda expansiva de un enemigo, emborronando la visión.

8. Conclusión

De entre los motores gráficos comerciales disponibles al público, *Unity3D* fue el seleccionado para este proyecto debido a su total accesibilidad, su facilidad y versatilidad de uso, la comodidad de su paradigma de desarrollo para personas experimentadas en programación orientada a objetos y por poseer soluciones para la creación de sistemas de interfaces más maduras y menos dependientes de tecnologías externas, como venía siendo el caso con *Unreal* y *CryEngine*, que necesitaban del *middleware Scaleform*; además de por la experiencia de uso de él de más de un año.

La elección fue acertada, pues tres de sus características facilitaron enormemente el desarrollo del *plugin* para la creación de *HUDs* modulares:

- Extender el *UnityEditor* y crear ventanas nuevas para él resultó realmente sencillo, y sus posibilidades son amplísimas.
- El nuevo sistema de interfaces de *Unity*, sobre el cual se basan las *HUDs* diseñadas en este proyecto, es bastante completo y flexible y los nuevos componentes traen consigo una gran cantidad de funcionalidad implementada que facilita el trabajo en muchas ocasiones.
- El hecho de que pueda trabajarse tanto extendiendo el editor o programando para un juego en un mismo lenguaje de programación, en nuestro caso *C#*, ayuda a unificar todas las partes del desarrollo.

Los otros motores gráficos comentados son más recomendados para tareas donde prime el resultado visual y se prefiera mayor sencillez en el desarrollo, pero para el trabajo con interfaces y un mayor control del flujo del programa, *Unity* resulta ser la mejor opción.

En cuanto al desempeño de la herramienta, podemos asegurar que el resultado es más que satisfactorio. Su utilización en el desarrollo de un juego como parte final del proyecto se tradujo en un tiempo dedicado a la creación de una *HUD* para el juego realmente pequeño: apenas diez minutos para la creación

y personalización de los cuatro elementos que la componen con algún ajuste manual y tintado posterior de algunos elementos, y tan solo la adición de nueve llamadas a la *API* de la *HUD* en puntos precisos del código del juego una vez ya era plenamente funciona para utilizarla, cuyo tiempo invertido es despreciable.

Como comentario adicional, cabría destacar que el juego desarrollado, en su primera y actual versión, ha tenido una buena acogida entre aquellos que lo han probado, incluso jugado más de una vez.

8.1. Trabajo futuro

Mucho puede hacerse todavía para mejorar lo desarrollado. Como tareas previstas a corto y medio plazo podemos destacar:

- La creación de más elementos prediseñados para las *HUDs*, y la implementación de su funcionalidad.
- El desarrollo de más extensiones del *Editor* que faciliten la manipulación de los elementos de *HUD* después de haber sido creados.
- La publicación del *plugin* en la tienda de *Unity*.
    - No existe actualmente ninguna herramienta en ella dedicada expresamente a la creación y administración de *HUDs*.
- El aumento del contenido del juego desarrollado.
    - Más variedad de enemigos y proyectiles.
    - Distintos modos de juego: superviviencia, contrarreloj...
    - Mayor variedad musical y de esquemas de color.
    - Más modos de control, pensados para distintas plataformas (en especial, el control táctil para dispositivos móviles).
- La publicación del juego en tiendas digitales de entretenimiento: *Steam*, *GOG*, *Google Play*...
    - Gracias a *Unity*, el juego podrá ser idéntico en plataformas de diversa índole y así poder aparecer en sus respectivas tiendas.

9. Referencia bibliográfica

[1] IGN. Unreal Engine 4 is free for everyone (2015). [Internet]

Disponible en: ign.com/articles/.../unreal-engine-4-is-free-for-everyone

[2] Gamasutra. The Engine Survey: General Results (2009). [Internet]

Disponible en: gamasutra.com/blogs/MarkDeLoura/20090302/83321/

[3] Epic Games. Scaleform Gfx (2012). [Internet]

Disponible en: udn.epicgames.com/Three/Scaleform.html

[4] Unity3D. Relaciones Públicas. [Internet]

Disponible en: unity3d.com/public-relations

[5] Unity3D. Versión 5. [Internet]

Disponible en: unity3d.com/es/5

[6] CryEngine. Features (2015). [Internet]

Disponible en: cryengine.com/features

[7] CryEngine Documentation. User Interface (2014). [Internet]

Disponible en: cryengine.com/display/SDKDOC4/User+Interface

[8] Unity Documentation Manual. UnityEditor (2014). [Internet]

Disponible en: unity3d.com/es/current/Manual/Editor.html

[9] MonoDevelop (2015). [Internet]

Disponible en: monodevelop.com

[10] Sublime Text 3 (2015). [Internet]

Disponible en: sublimetext.com/3

[11] Git. About (2015). [Internet]

Disponible en: git-scm.com/about

[12] Atlassian. Bitbucket (2015). [Internet]

Disponible en: bitbucket.org

[13] Atlassian. SourceTree (2015). [Internet]

Disponible en: sourcetreeapp.com

[14] GNU. GIMP (2015). [Internet]
Disponible en: gimp.org.es

[15] Nvidia. Geforce Shadowplay (2015). [Internet]
Disponible en: nvidia.es/object/geforce-experience-shadow-play-es.html

[16] Unity Documentation Manual. Construcción de escenas (2015). [Internet]
Disponible en: unity3d.com/Manual/BuildingScenes.html

[17] Unity Documentation. GameObject (2015). [Internet]
Disponible en: unity3d.com/ScriptReference/GameObject.html

[18] Unity Documentation. Component (2015). [Internet]
Disponible en: unity3d.com/ScriptReference/Component.html

[19] Unity Documentation. Transform (2015). [Internet]
Disponible en: unity3d.com/ScriptReference/Transform.html

[20] Unity Documentation. Rigidbody (2015). [Internet]
Disponible en: unity3d.com/ScriptReference/Rigidbody.html

[21] Unity Documentation. Collider (2015). [Internet]
Disponible en: unity3d.com/ScriptReference/Collider.html

[22] Unity Documentation Manual. Light (2015). [Internet]
Disponible en: unity3d.com/Manual/class-Light.html

[23] Unity Documentation Manual. Global Illumination (2015). [Internet]
Disponible en:unity3d.com/es/current/Manual/GlobalIllumination.html

[24] Unity Documentation Manual. Audio Mixer (2015). [Internet]
Disponible en: unity3d.com/es/current/Manual/AudioMixer.html

[25] Unity Documentation Manual. GUI Texture (2015). [Internet]
Disponible en: unity3d.com/es/current/Manual/class-GuiTexture.html

[26] Unity Documentation Manual. GUI Text (2015). [Internet]
Disponible en: unity3d.com/es/current/Manual/class-GuiText.html

[27] Tasharen. Next-Gen UI kit (2011). [Internet]
Disponible en: tasharen.com/?page_id=140

[28] Unity Documentation Manual. Interfaz de Usuario (2014). [Internet]
Disponible en: unity3d.com/es/current/Manual/UISystem.html

[29] Unity Documentation. RectTransform (2015). [Internet]
Disponible en: unity3d.com/ScriptReference/RectTransform.html

[30] Unity Documentation Manual. Extending the Editor (2015). [Internet]
Disponible en: unity3d.com/Manual/ExtendingTheEditor.html

10. Anexos

Se proporciona el siguiente hipervínculo al repositorio *git* alojado en *Bitbucket* que contiene todo el material utilizado y desarrollado a lo largo de todo el proyecto: bitbucket.org/a_diaz/modular_unity_custom_hud.