UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Memoria Scares For Sale: Diseño y desarrollo de un videojuego 3D sobre Unity: Arquitectura de GUI

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Agustín Esteve Guinot

Tutor: Javier Lluch Crespo

2013/2014

Agradecimientos

He de agradecer el apoyo recibido por mi familia y amigos, ya que sin ellos no podría haber llegado hasta donde estoy hoy en día.

También agradezco a la empresa BraveZebra por su confianza desde el principio. Gracias a ellos he aprendido todo lo que se hoy en día sobre programación de videojuegos.

Por último agradezco a la Universidad y a todo el profesorado el interés y los conocimientos que comparten con los alumnos a diario. En especial a Javier Lluch Crespo, por el tiempo invertido en que este proyecto salga adelante.

Resumen

En este proyecto tratamos la implementación de las interfaces y las trampas estilo *Tower Defense* para el juego *Scares for Sale*. También desarrollaremos una herramienta de gestión de menús sobre el *plugin NGUI* que permita configurar las transiciones en tiempo de editor.

Palabras clave: Unity3D, interfaces, menús, NGUI, videojuego.

Abstract

In this project we will talk about the platform Unity3D and some plugins we used to deliver videogames. Specifically we will explain how to program a menu controller and a trap system for a tower defense.

Keywords : Unity3D, menu, controller,videogame, NGUI.

Tabla de contenidos

Índice de imágenes

Índice de tablas

1. Introducción

El proyecto que se describe en esta memoria trata sobre la realización de un videojuego a partir de los diferentes elementos que lo componen, diseño, programación, integración de modelos en escena y diseño e implementación de interfaces utilizando para ello un motor profesional de videojuegos: Unity.

El videojuego desarrollado por tres estudiantes del grado de ingeniería informática especializados en computación será del género “Tower Defense”. En el juego el jugador encarnará a un fantasma cuyo objetivo es proteger una mansión de los compradores que la visitan, si un cierto número de estos compradores llega a hacer un recorrido completo hasta el tejado de la mansión, tendrá lugar la subasta de la casa y por lo tanto el fin del juego.

Para ahuyentar a las oleadas de compradores de la mansión, controlados por la IA del juego, el usuario deberá utilizar los sustos o habilidades que poseen los objetos encantados de la mansión, como pueden ser, las cortinas, sillas, bustos, armaduras, lámparas, etc. Cada uno de estos elementos encantados poseerá una habilidad única con tres niveles de efectividad, dichos niveles deberán ser adquiridos por el jugador con el dinero que dejan los visitantes al huir despavoridos tras superar un nivel.

La jugabilidad del proyecto se verá acompañada de las interfaces necesarias para llegar a jugar un nivel así como para comprar las mejoras de cada uno de los objetos disponibles. De la misma manera, a diferencia de las aplicaciones convencionales, un sistema de sonidos y música se utiliza en el juego para dar mayor ambiente a la temática escogida.

De esta manera el proyecto supone un corte vertical a la realización completa del juego, es decir, se desarrollará una versión reducida del juego que incluirá las características y funcionalidades representativas del producto completo, una mansión (varios niveles), objetos comunes a varias mansiones y específicos, interfaces, enemigos (IA), controles, sonidos y música.

En este trabajo no abordaremos todos los aspectos del videojuego *Scares for Sale*. Concretamente trataremos el diseño y creación de las interfaces y de las trampas propias de un *Tower Defense*, además de crear un sistema de gestión de menús reutilizable para otros proyectos. De la inteligencia artificial de los enemigos se encargará David Fernández Molina en el proyecto *Diseño y desarrollo de un videojuego 3D sobre Unity: Inteligencia Artificial*, y a su vez Sergio Alapont Granero tratará los gestores de audio y de entrada en el proyecto *Diseño y desarrollo de un videojuego 3D sobre Unity: Audio e introducción al Leap Motion*. Estos tres proyectos se han realizado bajo el tutelaje de Javier Lluch Crespo.

Empezaremos explicando el estado del arte con una comparativa de los motores del mercado actual que más éxito han tenido, junto con una comparativa de algunos de los lenguajes que ponen estos motores a disposición de sus usuarios.

A continuación en el apartado antecedentes procederemos a elegir el motor y el lenguaje que utilizaremos, explicando brevemente sus características principales. También realizaremos una comparación entre la interfaz nativa del motor utilizado y la propuesta por un *plugin* que utilizaremos en este proyecto.

Seguidamente, en el punto análisis y diseño, realizaremos el análisis de requisitos y el diseño del gestor de menús, de las trampas y de las principales interfaces del videojuego.

Continuaremos con el apartado implementación en el cual explicaremos elemento a elemento como hemos realizado los *scripts* del gestor de menús y de las trampas, y en el apartado siguiente observaremos los resultados obtenidos.

Por último tendremos las conclusiones y trabajos futuros explicando que hemos aprendido a lo largo de este desarrollo y explicaremos que trabajos podrían ampliarse en un futuro.

2. Objetivos

A la hora de empezar a programar un videojuego es de mucha utilidad disponer de herramientas personalizadas que ayuden al desarrollo de la aplicación. No es difícil darse cuenta de que hay funcionalidades que se repiten o que se pueden reutilizar de un juego para otro. Los *plugins* para *Unity* tienen esta finalidad y pueden resultar muy útiles, pero es muy difícil encontrar alguno que cubra a la perfección todos los requisitos de la aplicación. Al descubrir *NGUI* solucionamos algunos requisitos para mostrar objetos de interfaz por pantalla, pero este complemento no dispone de ningún tipo de gestión de interfaces ni de movimiento entre ellas.

Desde que comenzamos en el desarrollo de videojuegos nos hemos enfrentado al problema de las interfaces de maneras muy diferentes, pero todas tenían algo en común: era necesario escribir código nuevo para moverse entre los paneles. Por eso los objetivos de este trabajo son:

- Construir un gestor para los menús que pueda ser utilizado para cualquier videojuego.
- Que este gestor no requiera escribir código para realizar las transiciones entre las interfaces
- Diseñar las interfaces específicas para este videojuego y utilizar el gestor para construirlas.
- Asegurar que este gestor nos permita seguir utilizando las herramientas del *plugin NGUI*.

Por otro lado en este proyecto abordaremos la creación de las trampas que el usuario va a necesitar para ganar el juego. Por la tanto tenemos que añadir los siguientes objetivos:

- Diseñar unos *scripts* que permitan configurar de manera sencilla y escalable las trampas disponibles en el videojuego

- Utilizar este sistema para crear las trampas diseñadas

3. Estado del arte

El mercado de los videojuegos avanza a pasos agigantados en múltiples direcciones. Esto se debe a que han surgido nuevas plataformas y utilidades que han acercado el mundo de los videojuegos a nuevos desarrolladores y a nuevos tipos de jugadores. Los géneros más afectados son los independientes y los juegos *casual*. Los primeros, al surgir nuevas herramientas o simplificarse las ya existentes, han recibido a muchos desarrolladores que sin arriesgar demasiado capital pueden permitirse publicar una app, o una aplicación en internet. Al no haber gran inversión de capital se pueden permitir el lujo de innovar más lo cual aumenta las probabilidades de éxito, pero también de fracaso. Los segundos han recibido un gran aumento de jugadores sobre todo en plataformas móviles, y por tanto muchas empresas han decidido trabajar en más juegos de este estilo, que tienen mecánicas muy simples y con poca diferencia entre ellos, pero que tienen buenos resultados en el mercado.

Nuestro juego se sitúa un poco entre ambos mercados. Por un lado innovamos en las mecánicas básicas de un *Tower Defense* y por otro lado intentamos atraer a los jugadores *casual* haciendo que las partidas no sean muy largas y que estas mecánicas no sean muy difíciles de aprender.

3.1. Comparativa con otros motores

Existen diversos motores de videojuegos y cada uno destaca en ciertos aspectos. A lo largo de este apartado vamos a repasar los motores más conocidos en el mercado y a compararlo con el motor escogido, *Unity*.

3.1.1. Unreal Engine

Unreal Engine es, sin duda, el motor de videojuegos 3D más extendido en cuanto a videojuegos profesionales se refiere. Su desarrollo está a cargo de la compañía *Epic Games*, creadores de juegos como *Gears of War*, quienes comenzaron el desarrollo del motor en 1998 aplicando en él las más novedosas técnicas de los videojuegos hasta llegar a su versión actual: *Unreal Engine 4*.

Unreal Engine solía ser el motor elegido por la mayoría de desarrolladores debido a la fácil accesibilidad del motor, pues constaba de un kit de desarrollo totalmente gratuito (con ciertas limitaciones) con numerosos modelados, sistemas de partículas, scripts e incluso proyectos ya terminados con los que aprender a utilizar el motor.

Hasta la versión cuatro, los usuarios de UDK podían utilizar el kit de manera gratuita hasta comenzar a comercializar el juego, en este caso los desarrolladores debían pagar 100 dólares y un 25% del total de ganancias después de conseguir unos beneficios totales de 50000 dólares. Si los desarrolladores poseían cierto renombre podían

contactar con *Epic Games* para adquirir total acceso al motor UE3 negociando el precio.

Todo ello cambió con la llegada de UE4, el nuevo modelo de negocio de *Epic Games*, pasa a cobrar 20 dólares al mes más un 5% de las ventas, para tener acceso Al código de la última versión de *Unreal Engine* directamente desde el repositorio en GITHUB, teniendo que compilar los propios usuarios el proyecto desde Visual Studio.

A continuación se muestra una comparación resumen sobre los pros y contras que *Unreal Engine* presenta sobre *Unity*:

	Unreal	Unity
Pros	Al pagar una mensualidad ya se posee una versión (no actualizada) del proyecto. Acceso al código fuente del motor (C++). Resultados visuales más aparentes que con otros motores debido sobre todo a los efectos de iluminación. Creación de materiales, *shaders* incluso scripts desde editores gráficos (con grafos).	Versión gratuita, ampliable mediante plugins, sufiente para la mayoría de proyectos. Posibilidad de programar con lenguajes más conocidos y con gestión dinámica de memoria (C# y Javascript). Shaders programados mediante ShaderLab, un lenguaje muy similar a Cg y GLSL.
Contras	Imposibilidad de probar el motor sin pagar. Comunidad pequeña con pocos tutoriales y facilidades para aprender en comparación con otros motores. Imposibilidad de publicar un juego si no se ha pagado la mensualidad requerida.	Versión Pro demasiado cara para pequeños desarrolladores. Falta de comodidades para los no programadores.

Tabla 1: Comparación Unreal-Unity

3.1.2. CryEngine

CryEngine se ha mostrado desde 2002 como uno de los motores más potentes de la historia, superando en varios aspectos a UnrelEngine.

Diseñado por Crytek, CryEngine no se ha extendido entre los desarrolladores de manera tan amplia como si lo han hecho Unreal o Unity, pero los pocos juegos en los que ha sido utilizado han hecho que el motor se gane su fama, sobre todo por su gran potencia gráfica.

Algunos de los juegos desarrollados con *CryEngine* son: *Crysis* (desarrollado por la propia *Crytek*), *Los Sims 3*, *Aion Online*, *Monster Hunter Online* o la franquicia *Far Cry*.

De la misma forma que *Unreal*, *CryEngine* cambió su modelo de negocio en 2014 para pasar a cobrar 9.90 al mes por obtener la última versión del motor, recuperando parte del terreno que tenía perdido ante *Unity* y *Unreal* por su exclusividad.

A continuación se muestra una comparación resumen sobre los pros y contras que *Unreal Engine* presenta sobre *Unity*:

	CryEngine	*Unity*
Pros	Al pagar una mensualidad ya se posee una versión del proyecto. Gran acabado visual de los videojuegos sin necesidad de añadidos. Creación de scripts de juego (eventos, reglas de juego, etc) con editor de flujo gráficos. Uso del lenguaje LUA para describir reglas de juego, Inteligencia artificial, comunicación en red, si no se quiere utilizar los editor gráficos, aunque la programación general se realiza en C++.	Versión gratuita, ampliable mediante *plugins*, suficiente para la mayoría de proyectos. Soporte para más plataformas, no solo consolas (como Xbox, Wii o PS4) si no también Mac, flash, *webgl*, iOS, Android y *Blackberry*. Posibilidad de programar con lenguajes más conocidos y con gestión dinámica de memoria (C# y *Javascript*). *Shaders* programados mediante *ShaderLab*, un lenguaje muy similar a Cg y GLSL.

Contras	Imposibilidad de probar el motor sin pagar. Comunidad muy reducida, foro con pocos usuarios y con escasos tutoriales. No se puede publicar un juego si no se ha pagado la mensualidad requerida.	Versión Pro demasiado cara para pequeños desarrolladores. Falta de comodidades para los no programadores.

Tabla 2: Comparación CryEngine-Unity

3.2. Comparativa de lenguajes

Cuando se comienza a trabajar empleando *Unity* hay una serie de premisas que deben ser cumplidas, una de ellas es la de escoger el lenguaje o lenguajes de programación de los que se hará uso a lo largo del proceso de scripting en las aplicaciones pertinentes.

En el caso de *Unity* el usuario puede elegir trabajar con C#, con *UnityScript* o con Boo, siendo este último el menos utilizado por la comunidad.

Escoger uno de estos tres lenguajes para comenzar a escribir código para *Unity* es un debate que comenzó en los inicios del propio entorno, dado que cada lenguaje proporciona sus propias características.

- C#: Gran cantidad de desarrolladores escogen C# para sus aplicaciones, esto se debe al propio origen del lenguaje.

  Este lenguaje de programación orientada a objetos fue desarrollado por Microsoft e introducido dentro de la plataforma .NET.

  Fue creado con influencias de lenguajes basados en C entre los que se encuentran C++ y Java, esto ha desencadenado que muchos desarrolladores provenientes de estos lenguajes escojan C# como el lenguaje a emplear en los scripts usados en *Unity*.

- *UnityScript*: Este lenguaje es a menudo confundido con JavaScript por muchos usuarios entre los que se encuentran miembros de la propia compañía.

  Mientras que JavaScript es un nombre el cual hace referencia a la especificación *ECMAScript*, *UnityScript* (.js) es un lenguaje creado por los desarrolladores de *Unity* para su uso dentro del entorno. Ese es el motivo por el que hay mucha confusión en la red cuando se trata de buscar información fiable acerca de *UnityScript*.

- *Boo*: Este lenguaje basado en Python fue creado con el objetivo de proporcionar al usuario la agilidad propia de Python y las potentes características de la plataforma .NET.

  Se caracteriza por su sintaxis y sistema de identación semejante al de Python el cual permite declaración automática de variables, currificación e inferencia de tipos.

  A diferencia de otros lenguajes POO como C# o Java, *Boo* no necesita la creación de una clase para escribir código ejecutable, es decir el uso de clases es algo opcional.

Sin embargo, son C# y *UnityScript* las opciones comúnmente escogidas por la comunidad a la hora de trabajar con los scripts de *Unity* dado que aunque *Boo* es un lenguaje de fácil comprensión y aprendizaje, su influencia en la red no tiene una gran extensión lo que provoca dificultades por parte de los programadores a la hora de buscar documentación y tutoriales útiles para las necesidades que aparecen durante el proceso de desarrollo.

Por otro lado, toda la documentación y librerías propias de *Unity* aportan el código de sus ejemplos escritos en *UnityScript*, siendo el número de ejemplos escritos en C# más reducido. En la **Tabla 3**: Ventajas e inconvenientes de C#,

C#	
Ventajas	Inconvenientes
- Proviene de .NET lo que es un estándar ya establecido. - Documentación extensa y completa. - Ligeramente más eficiente que *UnityScript* y Boo en tiempo de ejecución en *Unity*. - Encuentra errores en tiempo de compilación.	- Menos fácil de aprender para alguien sin experiencia con lenguajes POO. - Menos ejemplos en la documentación de *Unity*.

Tabla 3: Ventajas e inconvenientes de C#

UnityScript	
Ventajas	Inconvenientes
- Tiene una rápida curva de aprendizaje - Documentación extensa y completa de la documentación de *Unity*.	- Es un lenguaje creado por los integrantes de Unity para *Unity* y no tiene un uso externo al entorno. - Encuentra errores en tiempo de ejecución.

Tabla 4: Ventajas e inconvenientes de UnityScript

Boo	
Ventajas	Inconvenientes
- Tiene una rápida curva de aprendizaje - Fácil de hacer código rápido y limpio. - Encuentra errores en tiempo de compilación.	- Está claramente menos extendido que C# y *UnityScript* entre la comunidad. -Dificil de encontrar tutoriales y ejemplos.

Tabla 5: Ventajas e inconvenientes de Boo

Es común encontrar proyectos en los que hay fragmentos codificados en un lenguaje y otros fragmentos codificados en otro. Esto es posible en *Unity* pero hay que asegurarse de que se sigue un estricto orden de compilación de los scripts (el orden de compilación de los scripts es algo parametrizable dentro del editor).

4. Antecedentes

La utilización de un motor de videojuegos se está convirtiendo en requisito indispensable para poder programar videojuegos de manera profesional. Este hecho no debe de extrañarnos ya que este software no solo facilita las partes de más bajo nivel de la programación de videojuegos, sino que muchas de estas plataformas simplifican de manera muy efectiva problemas como la portabilidad a otras consolas, manejo de grafos de escena o librerías con funcionalidad ya implementada.

4.1. Motor de videojuegos utilizado: Unity

Se ha elegido *Unity* **[1]** como motor para el desarrollo del proyecto al ser uno de los motores de videojuegos profesionales cuya fama y uso más se ha extendido en los últimos tres años ante competidores como *Unreal Engine* o *CryEngine*.

Una de las razones por las que *Unity* ha destacado con respecto a otros motores de videojuegos es que ha sabido reunir tanto a empresas profesionales de videojuegos 3D, desarrolladores de juegos para plataformas móviles, así como a desarrolladores independientes o particulares debido a varias características:

La posibilidad de escribir código en tres lenguajes: C#, *JavaScript* o *BooScript*. Lo que atrajo a programadores de XNA (C#), programadores de Cocos2D (*Javascript*) y otros usuarios acostumbrados a lenguajes como java.

Creación de juegos en 2D o 3D. Aunque *Unity* era originalmente un motor orientado a los videojuegos en 3D los usuarios que provenían de *Frameworks* como *Cocos2D* siempre lo utilizaron para realizar juegos en 2D por lo que *Unity* tratando de complacer a todos sus usuarios fue realizando cada vez más cambios para habilitar la creación de videojuegos 2D de manera más cómoda con componentes específicos para físicas, *colliders*, atlas de *sprites*, etc en 2D.

Extensiones del editor y creación de *plugins*. El editor de *Unity* puede ser extendido mediante scripts en los tres lenguajes descritos superiormente heredando de la clase "*EditorWindow*". Esto ha permitido a los usuarios de *Unity* crear extensiones del editor para agilizar o incluso automatizar ciertas tareas repetitivas o que requerían la coordinación con otros componentes del equipo de desarrollo como artistas o diseñadores de niveles.

De la misma forma, estas extensiones han permitido la creación de una amplia tienda (Store) de *plugins* que de manera gratuita o de pago permiten ampliar, o incluso sustituir, funcionalidades en *Unity*. Un ejemplo de ello es el *plugin NGUI*, un *framework* que suple las necesidades de los programadores para crear interfaces de usuario y *HUDs* debido a que lo ofrecido por Unity en este apartado resulta ser muy

escaso (aunque un nuevo sistema de interfaces ha sido anunciado para la versión 4.6 por parte de *Unity*).

La comunidad de *Unity*. El motor cuenta con una amplia comunidad que a través de los foros ayuda a los desarrolladores con cualquier problema o duda sobre cómo implementar o diseñar diferentes componentes de un videojuego. Además de ello *Unity* cuenta con una extensa documentación, así como, de numerosos tutoriales oficiales en *YouTube* que enseñan cómo utilizar algunas de las herramientas que el editor dispone.

Por otra parte *Unity* presenta algunos inconvenientes debidos principalmente a la existencia de dos versiones del motor, la versión Pro y la versión Free **[3]**.

Como el propio nombre indica, la versión Pro contiene numerosas herramientas y utilidades necesarias en proyectos más ambiciosos, de la misma forma esta versión requiere un pago para poder ser utilizada y compilar el juego para PC, MAC o Linux, si además se quiere exportar el juego a otras plataformas como consolas o móviles utilizando herramientas de la versión pro es necesario adquirir al versión anteriormente mencionada pero también la compra de manera individual de la extensión pertinente para la plataforma objetivo.

Algunas de las herramientas de la versión Pro que no pueden ser utilizadas en la versión Free son:

- *Navmesh*: Compilación de una malla de navegación sobre los diferentes elementos de la escena marcados como navegables. Utilizado para el movimiento de la inteligencia artificial por entornos con otras entidades y obstáculos dinámicos.

- Efectos de post-procesado a pantalla completa: Efectos como *motion blur*, corrección de colores, etc.

- Creación de *lightmaps* con luces globales o de área: Se precalculan las sombras causadas por luces globales o de área estáticas aplicando el resultado sobre las texturas que componen la escena, ahorrando el cálculo de dichas luces en tiempo de ejecución. El uso de *lightmaps* con resto de luces disponibles (*spotlights, point lights*, etc) si está disponible en la versión Free.

- Texturas 3D: Uso de texturas 3D utilizando efectos de *shaders* avanzados.

- Sombras suaves causadas por luces focales o puntuales en tiempo real: Pero si están disponibles sombras con menor resolución causadas por luces direccionales.

- LOD (*Level Of Detail*): Cuando una escena se hace demasiado grande se aplican técnicas de reducción del nivel de detalle. Estas permiten disponer de varias versiones de una misma malla con diferentes niveles de complejidad (número de vértices, *shaders*, etc) utilizándola de manera adecuada al distancia al jugador o jugadores.

Y algunas características de uso menos frecuente como filtros de audio, *occlusion culling y profiler* (monitorización) del tiempo de *renderización*, físicas y código en gráficas.

En la mayoría de proyectos la versión Free es más que suficiente, sobre todo, si se trata de proyectos 2D.

Además de estas dos versiones cabe mencionar que *Unity* pone a disposición de los usuarios una prueba de 30 días de la versión pro con la que realizar pruebas de rendimiento y comprobar si de verdad interesa la compra de dicha versión.

En este proyecto se usa la versión pro de Unity haciendo uso de la versión de prueba de 30 días, junto con el *plugin* NGUI así como las librerías necesarias para adaptar Leap Motion al videojuego. La razón por la que se utiliza la versión pro de *Unity* es la creación de mallas de navegación para el movimiento de los enemigos (inteligencia artificial) por ellos de manera más profesional y similar a lo que haría una empresa de videojuegos que utilizase el motor.

4.2. Lenguage de programación utilizado: C#

En el caso de "*Scares For Sale*" se ha escogido C# como lenguaje de programación. El motivo de esta elección radica en la familiaridad con los lenguajes basados en C aprendidos durante la etapa universitaria donde se adquirieron conocimientos de Java, C y C++ entre otros lenguajes. Otro punto a favor de esta elección se debe a que *Untiy* está construido sobre la infraestructura .NET siendo el uso de este framework algo cotidiano durante la elaboración de los scripts. Además hay que destacar que a la hora de buscar información acerca de las librerías .NET solo se encuentran ejemplos de código en C++, C#, F# y VB.

4.3. Fundamentos del motor Unity

Unity como motor ofrece muchas y diversas utilidades que ayudan tanto a los programadores como a los diseñadores de niveles o a los artistas. Estas herramientas van desde clases ya programadas o editores de terreno hasta *shaders*. A continuación vamos a dar un repaso a las herramientas básicas que utilizaremos para este proyecto.

4.3.1. Interfaz

La interfaz de *Unity* consta de diversas ventanas integradas. Cada ventana se puede desplazar por la interfaz de *Unity* para crear una configuración personalizada a nuestro gusto. Las ventanas principales que proporciona Unity son: la ventana del proyecto, la ventana de la jerarquía, el inspector, la ventana de escena, la de juego.

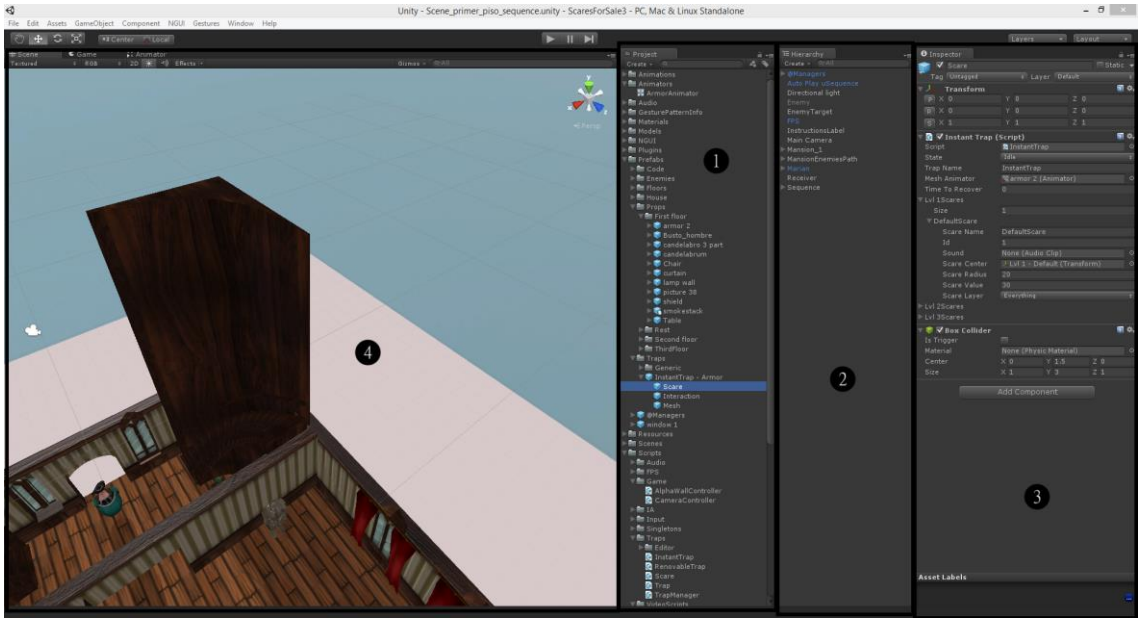

Imagen 1: Ventanas de Unity

La ventana del proyecto es como un explorador de archivos, como podemos observar en la **Imagen 1-1**, en el cual podemos organizar, agregar y modificar todos los elementos o *assets* que puede llegar a tener nuestro juego. Todos los archivos que se muestran se corresponden con los de la carpeta llamada *assets* dentro de la carpeta del proyecto. Se sincronizará a tiempo real con la carpeta del sistema, esto significa que si añadimos algo a la carpeta y volvemos a *Unity*, se actualizará. También podemos arrastrar cualquier objeto dentro de esta ventana y se añadirá al proyecto.

No hay que tener miedo a la hora de añadir elementos a la carpeta *assets*, ya que no todo lo que se encuentra en esta carpeta será exportado a la hora de compilar. Solo se añadirán al ejecutable los *assets* que se usen en alguna escena. En esta ventana también dispondremos de un buscador que nos permite buscar y filtrar todos los componentes.

La ventana de la jerarquía**,** que podemos observar en la **Imagen 1-2,** es la que nos sirve como herramienta de organización de los elementos de una escena. Es decir, aquí podremos organizar, añadir, eliminar o buscar cualquier elemento ya instanciado. Para instanciar un nuevo elemento solo tenemos que arrastrar un a*sset* desde la ventana del proyecto a esta ventana o a la escena. También se pueden crear algunos objetos genéricos desde el menú, como por ejemplo formas geométricas predefinidas (cubo, esfera, cilindro) o algunos *prefabs* ya configurados, como un sistema de control en primera persona.

Esta ventana también aporta algunas funciones útiles, como por ejemplo centrar la ventana de la escena sobre el objeto seleccionado (haciendo doble *click* izquierdo con el ratón) o arrastrar uno de los objetos creados a la ventana del proyecto, lo que creará un p*refab* con la configuración actual del objeto.

El inspector es la ventana mostrada en la **Imagen 1-3** y permite observar qué componentes tiene añadido el objeto seleccionado y modificar sus propiedades. Se puede modificar la apariencia del inspector para los *scripts*. Esto puede resultar muy útil para poder configurar desde el editor y por tanto reutilizarlos para otras tareas

similares. Aun así, el inspector predeterminado ya dispone de los elementos más comunes como los elementos primitivos, vectores 2D y 3D, curvas de animación, colores, y más.

La ventana de escena corresponde con la **Imagen 1-4** y contiene una representación del mundo 3D que conforman todos los *assets* instanciados del videojuego. Esta ventana es muy útil sobre todo para poder hacerte una idea de la composición de todos los elementos, y permite editarlos, mover, rotar o escalar. Se puede extender de esta ventana y darle funcionalidad extra, como añadir objetos, cambiar propiedades o dibujar elementos aclaratorios como por ejemplo la dirección en la que se va a mover una plataforma, o crear una linea que dibuje la trayectoria de un enemigo, estos elementos se llaman *guizmos*. Ademas esta ventana permite que nos movamos por los elementos la escena con unos controles muy limitados, por supuesto, pero podemos observar un acabado parecido al que se verá al final.

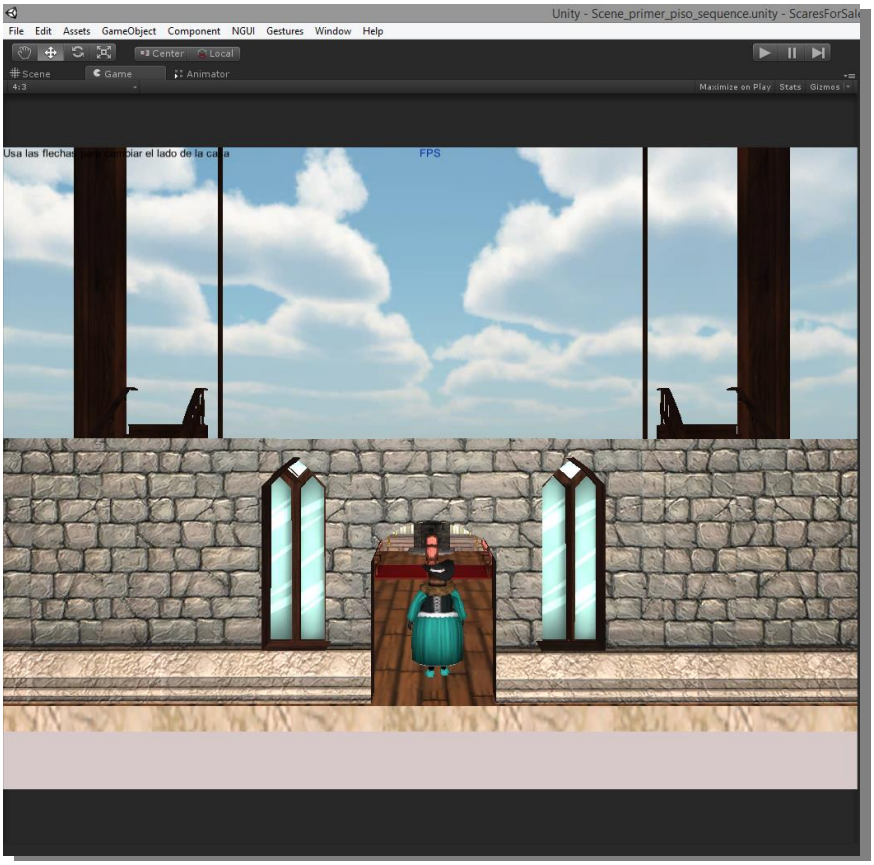

Imagen 2: Ventana de juego

La **Imagen 2** muestra la ventana de juego. *Unity* puede ejecutar el juego dentro del editor y será en esta ventana donde simularemos el juego. En esta ventana se pueden poner distintas resoluciones de pantalla para probar que se vea bien en cualquier resolución. Justo debajo de la barra de menús tenemos el botón de *play* que comenzará la simulación del videojuego, y el botón de pausa que congelará la ejecución en caso de que queramos para a inspeccionar algún elemento. Además de estos dos botones tenemos el botón paso a paso que ejecutará *frame* a *frame* para poder inspeccionar con mayor exactitud.

Unity tiene más ventanas, obviamente, pero hemos repasado las más utilizadas e indispensables para la programación de videojuegos. Otras ventanas son la ventana de la consola, que muestra los fallos y te redirige a la línea de código correspondiente, la ventana de animación, que permite crear animaciones mediante interpolación de valores o el *animator*, un editor de grafos de animaciones.

4.3.2. Escena

Una escena en *Unity* se trata de una composición de *assets*, *scripts* y eventos que conformarán los niveles de juego. Lo más normal es organizar un proyecto de *Unity* por una escena principal con el menú de juego y después una escena por cada nivel. Esto no solo es conveniente para la organización de un videojuego sino que además tiene ciertas repercusiones.

Cuando en *Unity* se carga una escena, se almacena en memoria todo lo que esa escena vaya a utilizar, de esa forma podemos ir liberando espacio en memoria organizando bien las escenas y los cambios entre ellas. El cambio entre escenas no es un procedimiento suave, aunque se puede ir cargando la escena siguiente mientras el juego sigue ejecutándose. En el momento en el que demos la orden para cambiar de escena se eliminarán todos los objetos de la escena antigua y se crearán los objetos de la nueva escena. Si no hemos cargado los objetos con anterioridad, tendremos que acceder a memoria para cargarlos y esto puede ser un procedimiento costoso.

Por si acaso fuera necesario pasar cierta información entre escenas (datos del usuario, alguna clase, puntuaciones, etc...) y no quisiéramos depender de una base de datos como intermediario, podemos usar la instrucción *DontDestroyOnLoad* que le indica a *Unity* que ese objeto no debe eliminarse al cambiar de escena. Evidentemente tendremos que tener en cuenta que objetos perdurarán entre escenas para evitar duplicados.

Otra manera de intercambiar datos entre escenas es usar la clase *PlayerPrefs*. Esta clase es abstracción de una tabla hash limitada en la que podremos guardar números enteros, de coma flotante, y cadenas de texto. Para ello le asignaremos a cada valor una clave como en cualquier tabla hash, que será una cadena de texto. Estos valores almacenados son persistentes por lo que además también podemos usarla para diferentes sesiones de juego. También existen *plugins* que permiten guardar objetos enteros en *playerprefs* o directamente en un archivo binario.

4.3.3. El GameObject y sus componentes

Un *GameObject* o instancia de objeto de juego es la unidad básica que conforma *Unity*. Cada uno de estos objetos va a almacenar una serie de componentes que dotarán de comportamiento a este objeto. Para saber si este elemento está activado tenemos una variable *enabled* que modula si está activo o no, cuando este desactivado todos sus componentes dejarán de ejecutar ciertos métodos. Cada componente por separado tendrá también un *booleano* para indicar si está activo o no, por si queremos desactivar solo parte de la funcionalidad de este objeto. Los métodos que no se ejecutarán cuando el *GameObject* o el componente este desactivado son *start, update, fixedupdate* y *ongui*.

Los *GameObject* siguen una organización jerárquica en forma de padre/hijo, propia de los grafos de escena, que servirá entre otras cosas para organizar los elementos de las

escenas. Como el comportamiento de los *GameObjects* depende de sus componentes se vuelven objetos muy versátiles. Además se pueden añadir/obtener componentes en tiempo de ejecución de un *GameObject* para añadir funcionalidades o modificar los valores de las variables para alterar su comportamiento.

Un componente es básicamente un *script* que tiene que heredar de la clase *monobehaviour*. *Unity* ya tiene diversos componentes implementados, pero lo más común es que la inmensa mayoría de los *scripts* que utilicemos sean nuestros. Algunos de los más importantes son:

- **Transform:** Esta componente contendrá la posición, rotación y escala de objeto. Todo *GameObject* tendrá siempre un componente *transform*. Esta componente también tendrá una variable llamada *parent* que referencia a su padre directo en la jerarquía. Si esta componente es nula este objeto no tiene padre y la posición mostrada será la posición global (respecto al punto 0,0,0 de la escena) pero si el padre es otro *GameObject* la posición mostrada será la posición local (con respecto a la posición del padre). También nos ofrecerá métodos para mover y transladar objetos, rotarlos con respecto a un punto o con respecto a su propio pivote, entre otros.

- **Collider:** Se trata de un componente que va a crear un área de inclusión (caja, esfera, capsula, …). Esto permitirá que el objeto pueda colisionar con otros objetos y que se disparen ciertos eventos en los *scripts* de ese *GameObject*. Este componente tiene un *booleano* para indicar si actúa como un *trigger*, lo que significa que no bloqueará el paso a otros *colliders* pero si que disparará los eventos pertinentes.

- **Rigidbody:** Este componente va a permitir que el *GameObject* interaccione con las físicas. Añadir este componente permitirá que la gravedad le afecte y permitirá también añadir fuerzas con una magnitud, un sentido y un tipo de fuerza. Para algunos objetos puede resultar más fácil (o necesario) realizar movimientos mediante su componente *transform* pero aun así quizá queramos que use el motor físico en algún momento. Para ello deberíamos activar el *booleano* i*skinematic* que limitará el cálculo del motor físico y podremos mover el objeto cambiando su posición en el *transform* de manera segura, si no lo hiciéramos de esta manera al cambiar su posición el motor físico lo tomaría como un teletransporte y si se produjera alguna colisión, la fuerza de repulsión sería infinita y por tanto las reacciones serían imprevisibles.

- **AudioSource:** Gracias a este componente el *GameObject* podrá emitir sonidos ya sean en 3D (con atenuación) o en 2D. Pero para que este componente funcione correctamente, es necesario que haya un a*udiolistener* colocado en la escena y a una distancia determinada. Aqui en el a*udiosource* sera donde modifiquemos el volumen y el timbre, la caida del sonido con la distancia (solo para sonidos 3D) y la prioridad.

- **AudioListener:** Este componente recibirá los sonidos que otros componentes a*udiosources* emitan. Se pueden tener más de uno por escena pero no es recomendable, ya que pueden producirse efectos inesperados.

4.3.4. Monobehaviour

Es la clase principal que cualquier *script* que desee formar parte de un *GameObject* debe de heredar. Esta clase nos aportará métodos y eventos útiles para la inicialización, ejecución y destrucción. Algunos de los más importantes son:

- **Start:** Se llama al inicio de la creación del *GameObject* que lo contiene. Este método se usa principalmente para dotar de valores por defecto al s*cript*. A no ser que alteremos el orden de ejecución de los *scripts,* no podemos controlar en que orden se ejecutarán.

- **Awake:** Este método se ejecutará justo antes del s*tart*. Esta iniciación en dos fases es muy útil ya que puede que algunos *scripts* necesiten para su propia inicialización que otros *scripts* ya tengan sus valores calculados. Y aunque podamos alterar el orden de ejecución de los *scripts* siempre será una solución más práctica usar el a*wake*.

- **Update:** Se trata de un método que se va a ejecutar una vez por cada *frame*, por tanto tenemos que tener mucho cuidado al introducir bucles y operaciones costosas dentro de este método. Sin embargo, es muy útil para realizar movimientos de personajes y/o control del *input*.

- **LateUpdate:** Este método se ejecuta justo después del u*pdate*. Esto puede ser muy útil para que algún objeto reaccione ante otro y siempre reaccione después. Por ejemplo si tenemos un *script* que mueve un personaje y queremos que la cámara le siga, el *GameObject* que contiene la cámara deberá tener un *script* que implemente el *lateupdate* y aquí modificaríamos la posición de la cámara.

- **FixedUpdate:** Este método se llama justo después de cada iteración del bucle de físicas. Debemos tener en cuenta que no podemos sobrecargar mucho este método ya que esto podría repercutir en reacciones inesperadas dentro del juego.

Heredar de *monobehaviour* nos aporta también muchos otros métodos que se accionarán en respuesta a diferentes interacciones con el *GameObject* asociado. Algunos de estos métodos se llaman desde otros componentes del mismo *GameObject* y dependen de ellos para su ejecución. Los más importantes son:

- **OnEnable/OnDisable**: Este método se ejecuta siempre que se ejecute el método *SetActive* con el valor true/false sobre el *GameObject* asociado. Este método puede servir para preparar una interfaz o para guardar objetos siempre que se desactive el objeto.

- **OnDestroy:** Siempre que un *GameObject* se destruya (llamando a la función Destroy o DestroyImmediate) se ejecutará este método en todos sus componentes, lo cual permitirá guardar los progresos o avisar de que se va a destruir a otro *GameObject*.

- **OnMouseDown/Enter/Exit:** Estos métodos ayudan a la hora de capturar cuando el ratón pasa por un *Coliider* asociado a ese *GameObject*. Esto permite capturar respectivamente clicks de ratón y cuando entra o sale el ratón del *Collider* asociado.

- **OnCollisionEnter/Stay/Exit:** Este conjunto de métodos se dispararán si el *GameObject* dispone de un *collider* que no esté en modo *trigger* y se produce una colisión con otro *collider* con las mismas características. Reciben un parámetro del tipo *collision* que nos ofrece toda la información necesaria acerca de la colisión entre dos colliders.

- **OnTriggerEnter/Stay/Exit:** Funcionan igual que los anteriores pero solo cuando un *collider* que no sea *trigger* colisiona/entra/sale de otro *collider* que sí que sea *trigger*.

4.3.5. Prefabs

P*refab* es la contracción de prefabricado. Se trata de un *GameObject* que ya está configurado y que puede replicarse con la configuración de inicio. Junto con el *prefab* también se guarda la jerarquía que depende de este *GameObject*, de ese modo cuando lo instanciemos también crearemos a todos sus hijos. Gracias a todo esto podemos crear objetos complejos de manera rápida y sencilla, sin tener que estar añadiendo componentes en tiempo de ejecución. Las ventajas de los *prefabs* no se quedan aquí, si tenemos más de una instancia creada a partir de un *prefab* y cambiamos algún componente o variable del *prefab* cambiará en todas las instancias.

4.4. NGUI vs GUI nativa de Unity

A pesar de que últimamente la interfaz gráfica de *Unity* ha mejorado mucho (añadiendo *sprites* y *sprite sheets*), *NGUI* sigue ofreciendo la mejor solución para mostrar texturas, textos y botones tanto en 2D como en 3D dentro de *Unity*.

4.4.1. GUI nativa

La GUI nativa de *Unity* tiene varias deficiencias, entre ellas destacamos que es necesario crear un *script* que tenga una función *OnGUI* y en este *script* definir todos los elementos de la GUI que deba mostrar. Para indicar donde se coloca cada elemento tenemos que definir un objeto de la clase *Rect*. La clase *Rect* se define como una posición *x, y*, ancho y alto, lo cual colocará el elemento *x* pixeles desde la izquierda de la pantalla, *y* pixeles desde arriba de la pantalla y con el ancho y alto que hemos especificado. La profundidad con la que se dibuja cada elemento ésta definida en una propiedad estática de la clase GUI, cuanto mayor sea esa propiedad más atrás se dibujará.

La implementación de la GUI como un método tiene diversas consecuencias negativas. La primera de ellas es que hasta que no se ejecute el juego no tenemos una previsualización de lo que obtendremos. La manera más sencilla para poder ajustar los valores viendo cómo quedarán es exponer los parámetros necesarios como públicos para que aparezcan en el inspector de *Unity*, ejecutar el juego, cambiar los valores viendo donde quedará, copiar esos valores y al dejar de ejecutar el juego volver a introducir los valores otra vez en el *script*. Como se puede observar el proceso es un poco costoso aunque los resultados son buenos, tanto en eficiencia como en calidad.

Por último cabe destacar la creación de *skins* para aplicar a los componentes de la GUI. Estas *skins* permiten configurar como funcionarán las etiquetas, botones, texturas y

demás piezas que componen la interfaz. También permite añadir las nuestras propias. Para aplicar una *skin* a un elemento o series de elementos tenemos que cambiar la propiedad estática *skin* de la clase GUI y hasta que no la volvamos a cambiar todos los elementos se dibujarán con respecto a esta configuración.

4.4.2. NGUI

Por otro lado dentro de NGUI cada elemento es un *GameObject* y por tanto se puede ver en la jerarquía todos los elementos de la GUI. El elemento más importante es un *script* que modifica la cámara normal de *Unity*. Cada uno de los elementos que utiliza NGUI para mostrar la GUI heredan de una clase llamada UIWidget. Esta clase define el área que ocupa cada elemento en ancho y alto, también permite la elección del pivote a partir del cual se aplicarán esos valores. Esta clase también tiene un *depth* que modifica la profundidad de los elementos.

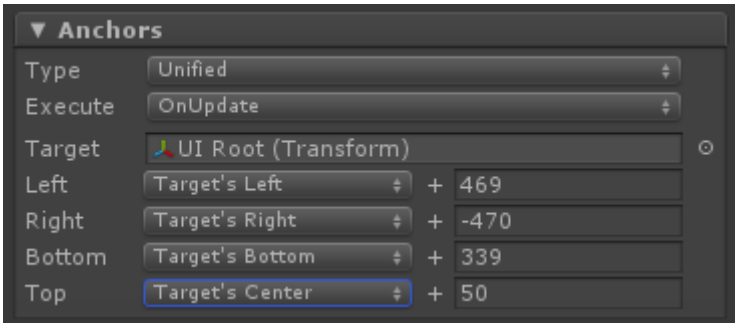

Imagen 3: Anchors simples

Dentro de NGUI la previsualización es instantánea ya que en la ventana de escena se ve en tiempo real como quedará la interfaz, además de poder modificarla desde la misma ventana de escena. Es posible que el resultado varíe un poco de la previsualización ya que dependiendo de la resolución de la pantalla puede variar la posición de los objetos. NGUI soluciona este problema añadiendo una propiedad llamada *anchor* a los UIWidget. Esta propiedad contiene los desplazamientos relativos de los cuatro lados del cuadrado que define lo que ocupará el elemento con respecto al padre. En la **Imagen 3** tenemos un ejemplo de *anchor* simple, y en la **Imagen 4** tenemos un ejemplo de *acnhor* avanzado

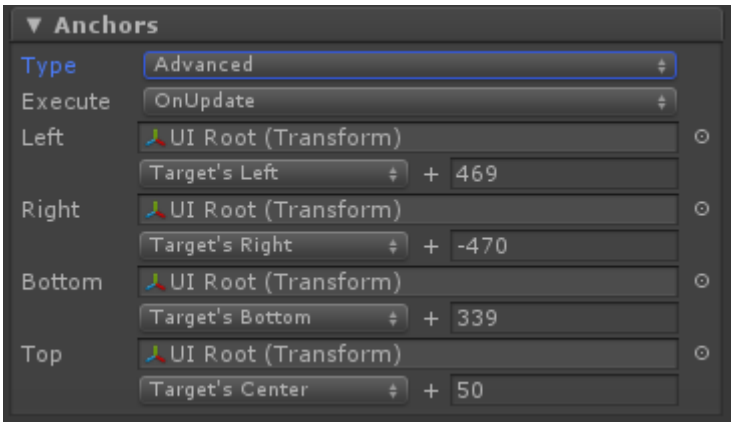

Imagen 4: Anchors anvanzados

Cada vez que se requiere de una textura NGUI dispone de un creador de *atlas*. Un *atlas* es una aglomeración de imágenes dentro de una misma textura. Cabe destacar que esta textura además guarda la información de donde están estas imágenes y cuánto ocupan en pixeles. Una vez definido el *atlas* al crear un boton o textura indicaremos en que *atlas* se encuentra y la imagen que necesita. En el caso de etiquetas solo necesitaremos indicar la fuente que se va a utilizar. Para más información acerca de *NGUI* se puede consultar su página web **[6]**, consultar su API **[7]**, probarlo de manera gratuita **[8]** o mirar los tutoriales **[9]**.

5. Análisis y diseño

Buscamos un gestor de menús que permita las operaciones básicas de movimiento entre paneles sin que tengamos que escribir código personalizado para cada proyecto. Para esto el gestor tiene que funcionar en tiempo de editor y exponer las variables que necesitemos configurar. También tendremos que asegurarnos de que todos los cambios realizados en el editor se conserven a la hora de ejecutar el juego.

5.1. Análisis y diseño del gestor de menús

Para cumplir con todos los requisitos hemos diseñado una serie de *scripts* que contendrán pedazos de funcionalidad. En primer lugar tendremos el *script CustomPanel* que utilizaremos para identificar los objetos del juego que actuarán como paneles, además contendrá un par de métodos que gestionarán la activación y ocultación del panel. Esta clase hereda de *UIPanel*, una clase que esta incluida dentro del *plugin NGUI*.

Si lo que deseamos es una ventana emergente deberemos agregar el *script CustomPopUp* que hereda la funcionalidad de *CustomPanel* añadiendo un par de instrucciones para asegurarnos de que la ventana este por encima al activarla.

Por otro lado tendremos el controlador de los menús, el *script MenuManager* que seguirá un patrón parecido al *singleton* adaptado a *Unity* y que tendrá las referencias a todos los paneles de la escena y también expondrá de manera pública una serie de métodos para la gestión de los paneles. Estos métodos son:

- *MoveToPanel* – Este método actúa como lanzadera para el método *MakeMovement*.
- *MakeMovement* – Es el método que ejecutará el movimiento según el tipo del origen y del destino. Es posible que un movimiento genere o modifique más movimientos.
- *Back* – Esta función tiene como objetivo deshacer el último movimiento realizado. Utilizará la pila de movimientos y servirá como método principal para cerrar las ventanas emergentes.
- *GetPanel* – Sirve para poder realizar una búsqueda de entre la lista de paneles. Devolverá null en caso de no existir

Para poder almacenar todos los datos necesarios en la pila de movimientos hemos diseñado una clase llamada *PanelMovement* que contiene una referencia al origen y al destino del movimiento y un tipo que nos ayudará a diferenciar los tipos de movimientos que podemos hacer.

Por último tenemos el *script CustomButton* que hereda de la clase *UIButton* de *NGUI* y que le añade diferentes comportamientos a este botón. Por cuestiones de mal funcionamiento de algunos comportamientos hemos deshabilitado el *OnHover* de la clase base y otros eventos. Las modalidades expuestas en *CustomButton* son:

- *Transition* – Si el botón es de tipo *transition* mostrará en el editor la lista de los paneles a los que puede transitar, elegiremos uno y este valor se guardara para esta instancia en tiempo de editor. Los paneles mostrados son indistintamente ventanas emergentes o paneles normales, por lo que antes de llamar a *MoveToPanel* comprobaremos de que tipo se trata.

- *Back* – No muestra ninguna propiedad nueva en el editor. Deshará el último movimiento que se encuentre en la pila llamando a la función *Back* de *MenuManager*.

- *Toggle* – Esta función requiere de un *GameObject* y de un valor determinado. Basicamente esta funcionalidad se encarga de activar, ocultar o alternar el estado de activación u ocultación de un *GameObject*.

- *OnOff* – Un botón de *OnOff* utilizará los estados de los botones de *NGUI* para definir los dos estados, presionado o no, del mismo. Expondrá en el editor un *booleano* para indicar el estado actual.

- *Group* – Agrupará mediante un entero diferentes botones *OnOff* para asegurar que solo uno de ellos este presionado. Internamente los guardaremos en una lista de listas de botones donde cada una de las listas internas solo tendrá activado un botón. En el editor se nos mostrará un *booleano* para indicar si esa instancia de botón estará presionada al inicio o no.

5.2. Análisis y diseño de las trampas

Para gestionar las trampas que estarán a disposición del jugador tendremos que tener en cuenta diferentes aspectos. En primer lugar vamos a definir que constituye una trampa y que funcionalidades van a tener. Una trampa será un elemento del mundo del juego con la que se podrá interactuar para asustar a los posibles compradores de la mansión.

El jugador interactuará con las trampas haciendo *click* sobre la misma. Dependiendo del tipo de trampa, reaccionará de manera diferente. Por el momento hemos diseñado dos tipos de trampas. Las instantáneas que al interactuar con ellas activarán el susto del nivel correspondiente. Por otro lado las trampas renovables, se activarán solas cuando un objetivo la entre en su radio de acción, y solo requerirán del usuario para rearmarse. La relación entre los *scripts* puede verse en la **Imagen 5: Esquema de las trampas**.

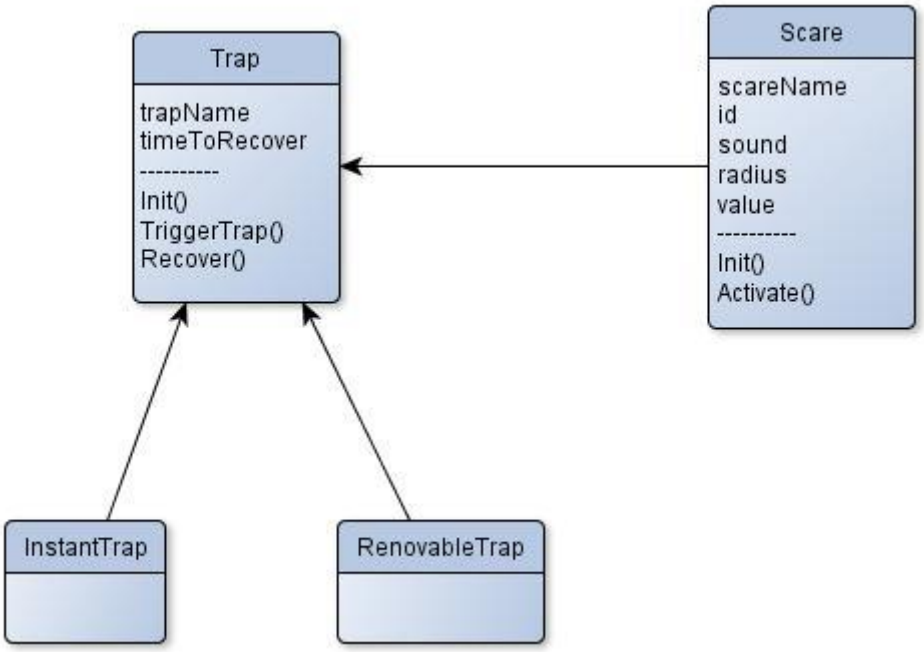

Imagen 5: Esquema de las trampas

Cada trampa tendrá un nivel determinado entre 1 y 3. Este nivel se puede aumentar desde la tienda en la escena del menú principal. Cuando una trampa se activa se accederán a los sustos disponibles para una trampa de ese nivel y escogeremos uno al azar.

Al activarse una trampa instantánea tendremos un tiempo en el que esa trampa este deshabilitada, y después volverá a estar disponible. Cuando se acaba el susto de una trampa renovable esta quedará deshabilitada hasta que el usuario la rearme otra vez.

Tendremos un *script* genérico *trap*, del cual no habrá ninguna instancia directa, pero si de las clases *InstantTrap* y *RenovableTrap*, que heredan de el. Sin embargo en *trap* gestionaremos el funcionamiento básico de las trampas con su ciclo básico. También almacenaremos información básica de las trampas, como la lista de los sustos por nivel. Todos los métodos de esta clase serán sobreescribibles, para poder moldear los métodos principales en caso de querer ampliar la funcionalidad.

Por otra parte el *script scare* modelará todo lo necesario para el susto. Es decir, cual es el nivel de susto, y cuanto radio de efecto tendrá. También modula el tiempo que pasa asustando y la animación que ejecutará el objeto para asustar.

5.3. Análisis y diseño de las interfaces

Para este juego vamos a necesitar diferentes menús navegables y un panel de *HUDs* para dentro del juego. El juego empezara con una pantalla de cargado, probablemente con una animación de la casa y un relámpago de fondo. Una vez terminado este proceso de carga, pasaremos a un menú principal que nos permitirá acceder al menú de opciones, a la tienda o a la selección de mansiones.

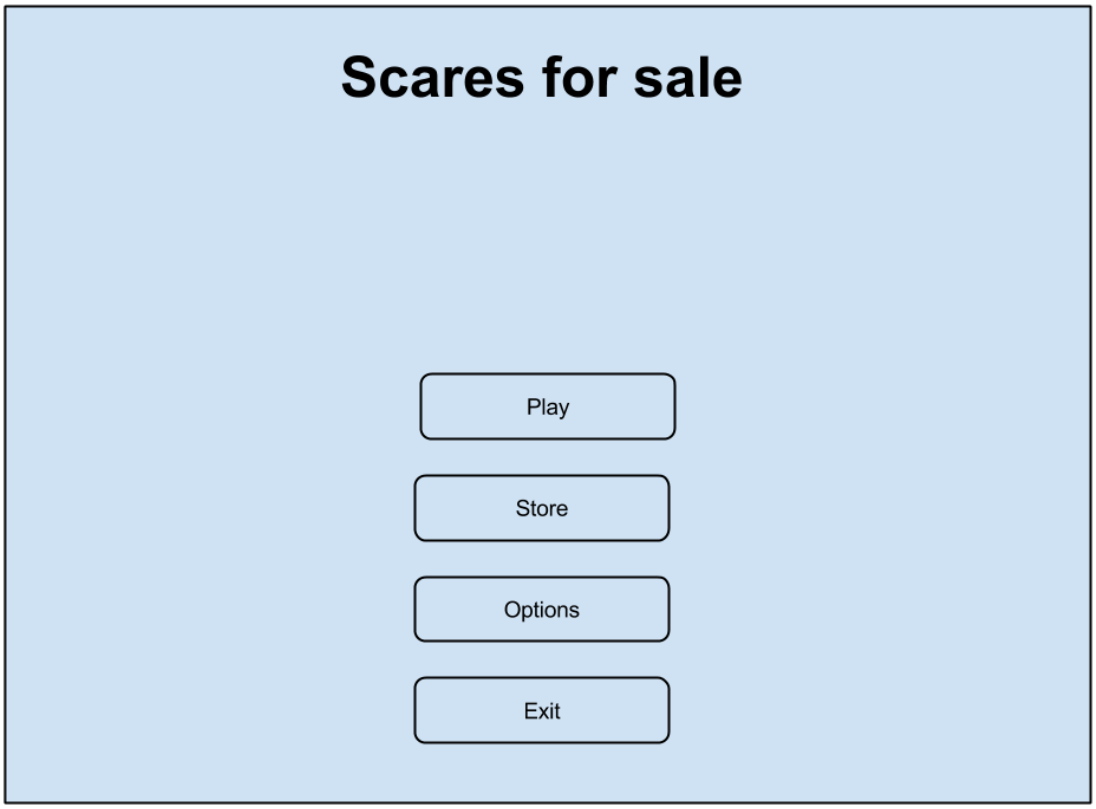

Imagen 6: Menú principal

El selector de mansiones tendrá un panel deslizante con una pequeña previsualización de la mansión. Esta previsualización además actuará como un botón para poder pasar a la escena correspondiente.

La tienda tendrá varias pestañas para cada una de las mansiones disponibles. Cada una de estas pestañas mostrará un elemento por cada trampa disponible en esa mansión. Estos elementos tienen un icono de la trampa, una etiqueta con su nombre, su nivel y su coste. Al hacer *click* en el elemento se ampliará este elemento mostrando una descripción, el coste de mejora y un botón de compra.

Si no se tiene suficiente dinero o se hace click sobre el indicador de monedas que tenemos en la esquina superior derecha, aparecerá una ventana emergente para realizar una compra de monedas con dinero real.

En el menú opciones encontraremos una barra de progreso que permitirá modelar el volumen de la música y de los sonidos del juego.

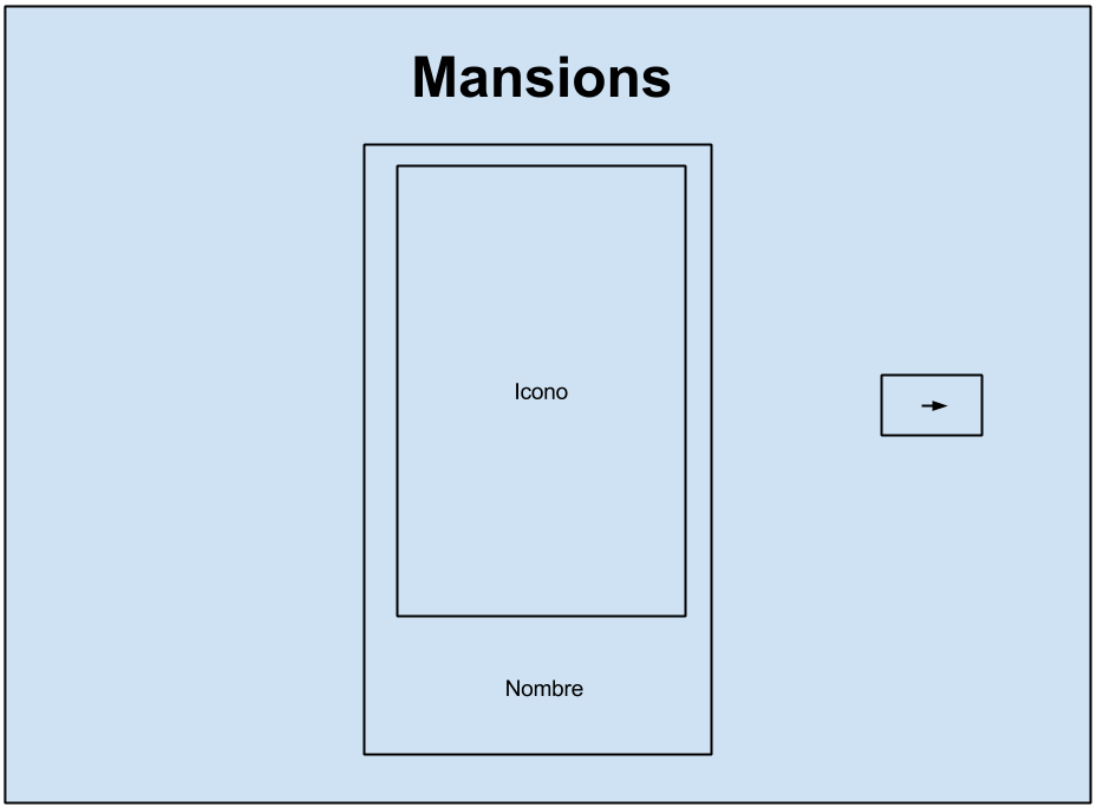

Imagen 7: Selector de mansiones

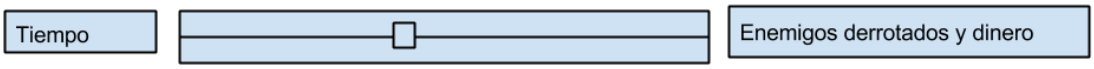

Imagen 8: HUDs

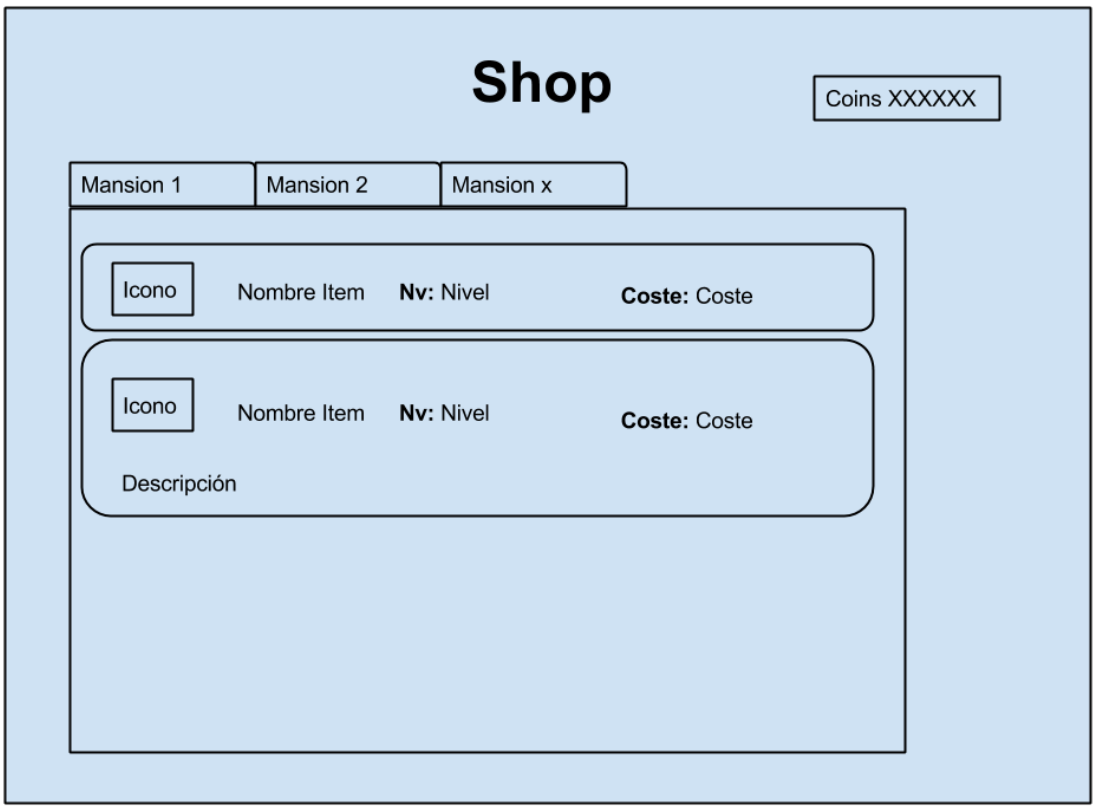

Imagen 9: Tienda

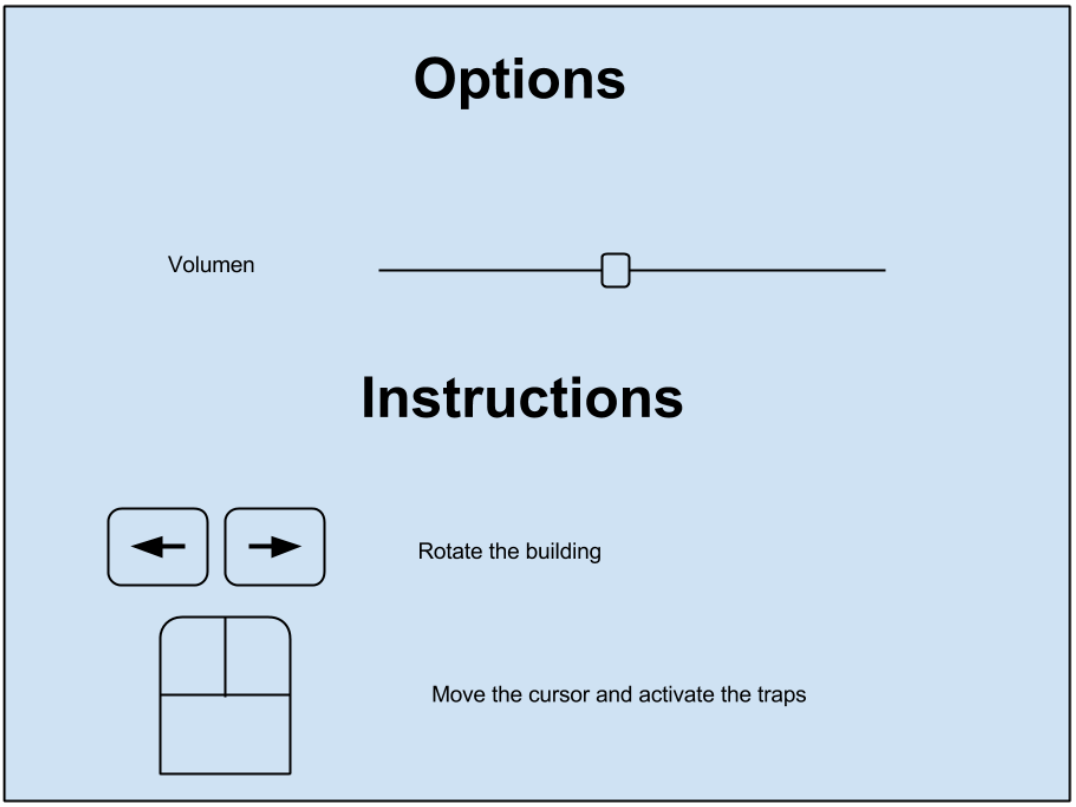

Imagen 10: Opciones

6. Implementación

A lo largo de este apartado vamos a ir *script* a *script* comentando como hemos implementado tanto el código que se va a ejecutar en tiempo de editor como el código que se ejecutará cuando la aplicación empiece su ciclo de vida.

6.1. MenuManager

Este será el eje central del gestor. Este fragmento de código será el que almacene los paneles de la escena y gestione como deben tratarse los movimientos entre los mismos. A su vez tendrá ciertos métodos para poder trabajar con estos componentes.

6.1.1. Editor

Para poder cumplir las principales necesidades del gestor de menús vamos a necesitar crear un inspector personalizado. Este inspector nos permitirá ejecutar código en tiempo de editor y por tanto configurar el gestor de menús para el juego en cuestión. Para más información se puede consultar los tutoriales de su página web **[5]**.

Para personalizar el inspector tenemos que añadir un script que hereder de *Editor* e indicar que vamos a usar las librerias *UnityEditor*. Ademas vamos a añadir una clave para indicarle al compilador el inspector de qué *script* vamos a editar.

Una vez creado este *script* tenemos que sobrecargar la función *OnInspectorGUI*. Si llamamos a la función *base* dentro de este nuevo módulo tendremos el inspector por defecto. En cambio a nosotros nos interesa mostrar una lista de los paneles que tenemos dentro de la escena y un menú desplegable para elegir el panel inicial que mostraremos. Además en tiempo de ejecución mostraremos el contenido de la pila de paneles.

Lo primero que vamos a hacer en esta función es asegurarnos de que la instancia del gestor de menús ésta asignada. Después vamos a buscar en la jerarquía todos los scripts del tipo *CustomPanel* y los guardaremos en la instancia de *MenuManager*. Usando estos paneles nos guardaremos el índice del panel inicial, si este panel es nulo cogeremos el primero en la lista.

Ahora mostramos el panel desplegable para asignar el panel inicial de entre todas las instancias con la llamada *EditorGUILayout.PopUp,* que nos devuelve el índice del desplegable seleccionado. Como hemos utilizado la lista de paneles para mostrarlo este índice coincidirá con el del panel seleccionado. Indicamos que los valores del *script* al que estamos modificando el inspector han cambiado, para que se guarden.

Para acabar con el inspector, vamos a recorrer todos los elementos de la pila de movimientos para mostrarlos. Evidentemente en tiempo de editor no habrá ningún movimiento, pero cuando ejecutemos la previsualización del juego y empecemos a movernos entre paneles podremos observar cómo va cambiado la pila de estado.

6.1.2. Código

Este *script* sigue el patrón *singleton* y por tanto tendrá una referencia estática a su instancia. Las demás propiedades són.

- *Panels* – Lista de los paneles que existen en la escena, adquiridos desde el editor modificado.
- *CurrentPanel* – El panel actual que se está mostrando por encima de todos. Al inicio tendrá el valor del panel inicial y conforme hagamos movimientos irá cambiando. Esta propiedad será muy útil, ya que supondremos que todos los movimientos partirán de este panel.
- *CurrentPanelIndex* – Solo lo utilizaremos para poder mostrar en el editor el panel inicial.
- *Movements* - Se trata de una pila de elementos de la clase *PanelMovement* que nos permite almacenar los movimientos entre paneles. La clase panel movement contiene el panel de origen, el panel destino y el tipo de movimiento que se ha realizado (de panel a panel, de panel a ventana emergente, o el cierre de una ventana emergente).
- *OnMovement* – Este evento se disparará justo antes de realizar un movimiento, de esta manera se puede crear un *script* que se subscriba a este evento para poder reaccionar ante los movimientos entre los paneles.

Solo hemos utilizado los métodos *Awake* y *Start* de los ofrecidos por *Monobehabiour*. En el primero solo nos aseguramos de que la instancia del patrón singleton esta asignada y en el segundo recorremos los paneles para mostrar el panel inicial y asegurarnos de que el resto de paneles están ocultos.

Todos los demás métodos serán servicios que ofreceremos para gestionar los movimientos básicos entre los paneles. Empezaremos por el principal: *MakeMovement*. Este método recibe un objeto de la clase *PanelMovement* y un booleano que indica si debe guardarse el movimiento en la pila o no.

El primer paso en este método es disparar el evento *OnMovement*. Ahora vamos a observar que tipo de situaciones se pueden dar. La mayoría de estos casos se comprobarán gracias a la propiedad *type* de la clase *PanelMovement* que nos han pasado, pero hay movimientos que producen movimientos más complejos.

El más común de los movimientos es el tipo *Panel* que tiene como origen y destino un panel. En este caso ocultamos el origen y mostramos el destino. Si el destino es una ventana emergente no ocultaremos el panel origen.

Si por el contrario estamos haciendo una transición desde una ventana emergente a un panel, vamos a ocultar todas la ventanas emergentes y crear un nuevo movimiento desde el primer panel que nos encontremos en la pila hasta el panel destino. Para ello vamos a recorrer la pila de movimientos mientras el origen de estos movimientos sea una ventana emergente, e iremos eliminando esos movimientos de la pila, ocultando además las ventanas emergentes. Cuando salgamos del bucle, nos encontraremos en el primer movimiento desde un panel a una ventana que hicimos, por tanto ocultamos la

ventana emergente y creamos un nuevo movimiento desde el panel origen de este movimiento al que nos encargaron movernos.

6.2. CustomButton

Después de *MenuManager* este es el componente más importante. Permitirá ejecutar ciertos fragmentos de código ya programados con funcionalidades básicas, aunque no todos los movimientos se tienen que hacer desde este *script*. Los métodos de *MenuManager* són públicos y se pueden llamar desde cualquier método.

6.2.1. Editor

Este *script* también necesita que personalizemos el inspector mostrado en tiempo de editor en la interfaz de *Unity*. Esta vez utilizaremos el editor no solo para configurar este *script* sino también para acceder a la información expuesta en *MenuManager*.

Este *script* de editor no va a heredar directamente de la clase *Editor* sino que lo hará de la clase que modifica el inspector de los *UIButtons* de *NGUI*. De esta manera podremos seguir utilizando las diferentes funcionalidades que nos ofrece.

Una vez mostrado todos los parámetros de los botones de *NGUI* vamos a añadir un menú desplegable para cambiar el tipo de botón que queremos. Dependiendo del tipo de botón mostraremos más o menos detalles de configuración como bien explicamos en el análisis.

Si el botón tendrá como finalidad transitar entre paneles, vamos a necesitar saber el panel objetivo, ya que el panel origen se presupone que es el actual. Para seleccionarlo pedimos al gestor de menús la lista de los nombres de los paneles y nos quedamos una referencia a este panel en la propiedad *target*.

Si seleccionamos la función de *toggle* mostraremos una caja para que se pueda soltar una instancia de un *GameObject* que activaremos o desactivaremos, junto a tres botones que nos darán las posibilidades de activar, desactivar o cambiar de estado el objeto seleccionado. Para mostrar que objeto tiene configurado la instancia que estamos configurando, utilizamos la instrucción *ObjectField*, que devuelve el objeto recibido desde el campo.

A continuación mostraremos la configuración actual del botón como una etiqueta mostrando el texto "*Toggle current"* si vamos a cambiar el estado actual o *"Toggle to: "* y si lo vamos a desactivar o no. Con esto tendremos *feedback* de la configuración y ayudará a la hora de configurarlo.

Después para mostrar los tres botones utilizaremos la instrucción *GUILayout.Button* que nos devolverá un booleano dependiendo de si se ha presionado o no, y configuraremos el *script* adecuadamente.

Si el botón va a ser del tipo *Group* vamos a mostrar un campo de entrada de enteros para indicar el identificador del grupo con la instrucción *EditorGUILayout.IntField* y con *EditorGUILayout.Toggle* un botón circular para indicar si este botón estará activo

al inicio o no. Todo esto modificará las propiedades *groupID* y *first* del botón para su posterior uso.

Las opciones *Back*, *On/Off* y *None* no necesitan mostrar nada por el editor, pero servirán para diferenciar las funcionalidades que se deben realizar.

6.2.2. Código

La clase *CustomButton* hereda de la clase *UIButton*, la que configura los botones del *plugin* de *NGUI* que utilizamos. Vamos a sobreescribir la función *OnClick* para poder ejecutar nuestro código en respuesta a los *clicks* del ratón.

Para cada funcionalidad hemos creado las propiedades y los métodos necesarios para cumplir los requisitos. Todas estas propiedades se asignan desde el *script* del editor. En el método *Awake* necesitaremos realizar unas comprobaciones. Si el botón es del tipo *On/Off* forzaremos el estado a normal y si el botón es de tipo *Group* comprobaremos si es el botón inicial de su grupo le cambiamos su estado a presionado y sino forzamos el estado normal. Además, es en este método donde registraremos los botones agrupados en una lista de listas de botones.

- *Transition* – Las propiedades necesarias son el panel objetivo y un entero necesario para mostrar en el menú desplegable el panel elegido. Cuando se haga *click* en el botón ejecutaremos el método *Move*. Este método comprueba si el objetivo es una ventana emergente o un panel normal y llama al método correspondiente de *MenuManager* para ejecutar el movimiento.

- *Back* – Ejecuta la función *Back* que avisa al gestor de que debe deshacer el último movimiento.

- Toggle – Vamos a necesitar dos *booleanos* y un objetivo. Uno de los *booleanos* indicara si debe cambiar su estado actual y si no es así utilizaremos el segundo *booleano* para cambiar el estado del objetivo.

- *On/Off* – Ejecutará el método *Switch*. Esta función accede a un *booleano* para comprobar el estado actual del botón y lo intercambia, a continuación llamamos al método *SetState* de los botones de *NGUI* para cambiar el estado según corresponda.

- *Group* – Para poder hacer efectiva esta funcionalidad tendremos un par de métodos, un para registrar a un botón en la lista de listas, asegurando que no haya más de uno activo, y otro para cambiar el estado de otro botón. Este último buscará el botón que esté actualmente activado y lo desactivara. Seguidamente activará el botón presionado. Estas activaciones y desactivaciones se realizarán llamando al método *switch*.

6.3. Paneles

Sobrescribir los paneles será de gran utilidad a la hora de diferenciar los paneles que sirven para organizar contenido, de los paneles que contienen interfaces.

6.3.1. Editor

El editor de los paneles que utiliza el gestor también será modificado pero sólo levemente. Al igual que el editor de los botones este también heredará de una clase de *NGUI*, la clase panel. El editor forzara el nombre del *script* al del campo de texto que vamos a mostrar justo después del editor predeterminado de los paneles de *NGUI*.

6.3.2. Código

Este *script* expondrá dos métodos principales, *Show* y *Hide*. Estos métodos gestionarán la activación y desactivación de los paneles. Actualmente es una simple llamada al método *SetActive* para activar o desactivar, pero lo mantendremos así para hacer el gestor más modulado y poder implementar nuevas funcionalidades más fácilmente.

6.4. Trampas

El *script trap* va a proporcionar ciertos métodos para que las clases que hereden de él, las que luego utilizaremos para configurar las trampas, solo tengan que sobreescribir los métodos necesarios y definir el comportamiento. Tendremos una propiedad llamada *state* que irá informando de en qué estado se encuentra la trampa.

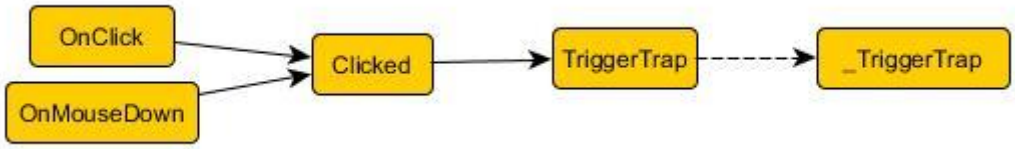

Imagen 11: Secuencia de activación de las trampas

El esquema básico es el propuesto en la imagen. Tendremos dos métodos lanzadera, si nos encontramos en la jerarquía de *NGUI* se llamará a *OnClick* y si no a *OnMouseDown*. Básicamente esta distinción nos permite probar las trampas sin necesidad de tener activos los *HUDs* del juego y además permitiría en un futuro añadir nuevos modos de entrada, ya que solo se debería llamar al método *Clicked*.

Una vez activado empezaremos a ejecutar la trampa. Como la ejecución de la trampa no es instantánea sino que tiene asociada una animación nos tendremos que esperar a que esta termine. Para ello vamos a lanzar una corutina (*_TriggerTrap*) que puede parar su ejecución sin afectar a los *FPS*. Primero cambiaremos el estado de la trampa a asustando y después elegiremos el susto aleatoriamente en base al nivel de la trampa, y lo activaremos llamando a su método *Activate*. Desde la corutina entraremos en un par de bucles de espera, uno para esperar a que empiece la animación y otro para esperar a que termine, y después de cambiar el estado a activada, dispararemos el evento *OnScareFinished*.

Por otro lado tenemos el método *Recover,* que inicilizará un *float* privado para contar el tiempo necesario hasta recuperarse. Este contador se irá actualizando con el tiempo que pasa entre *frames* dentro de un bucle *while*. Cuando el tiempo haya pasado, la trampa volverá al estado *idle* y disparará el evento *OnScareRecovered*.

6.4.1. Instantáneas

La trampa instantánea deberá empezar a recuperarse una vez que haya terminado de asustar. Para esto sobrescribiremos el método *Init* y una vez inicializada la trampa registraremos el método *OnScareFinishedCallback* al evento *OnScareFinished*. En la *callback* llamaremos al método *Recover* y empezaremos la recuperación.

6.4.2. Renovables

Para las trampas renovables sobrescribiremos el método *Clicked* para que en vez de llamar siempre al método, comprobaremos antes si la trampa se encuentra en el estado *triggered* empezaremos la recuperación, en caso contrario comprobaremos si podemos activar la trampa y si ninguna de las dos opciones es posible ignoraremos el *click*;

7. Resultados

Por la parte del gestor de menús, hemos conseguido crear un sistema que recoge en tiempo real todos los paneles que se encuentren en la escena permitiéndonos elegir el panel inicial y mostrar en tiempo de ejecución el estado de la pila de movimientos.

Por otra parte hemos conseguido que este sistema no requiera escribir código nuevo para los movimientos básicos entre los paneles. Evidentemente sigue siendo necesario la creación de códigos personalizados para gestionar la lógica de cada aplicación.

Todos los *scripts* realizados que hereden de las funcionalidades del plugin *NGUI* siguen manteniendo las propiedades, métodos y por tanto funcionalidades que ya proporcionaban.

Las interfaces que hemos diseñado cumplen con todos los requisitos que hemos prefijado. Sin embargo no hemos validado todavía las texturas que necesitaremos para las mismas, y por tanto el acabado no es definitivo.

El sistema de trampas que se ha implementado permite configurar cada una de las trampas diseñadas. Sin embargo la unión de los sustos y de las animaciones aun no aun quedado totalmente definidas. Se ha conseguido implementar para un caso en concreto con éxito, la trampa de la armadura. Las demás trampas no se han podido probar ya que todavía están realizándose.

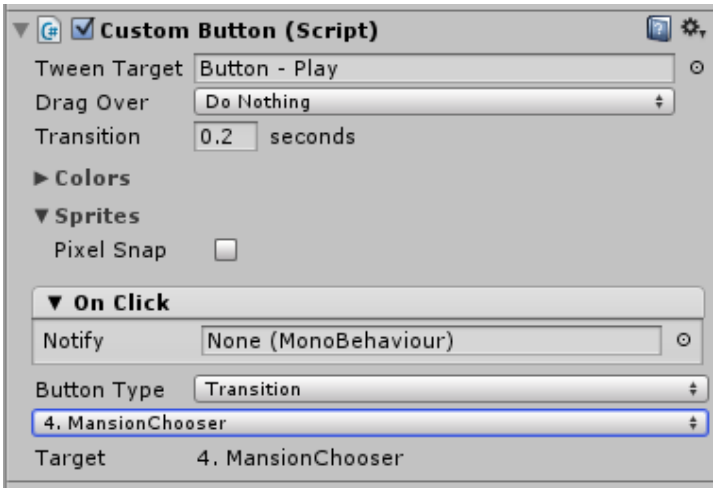

Imagen 12: Inspector del CustomButton

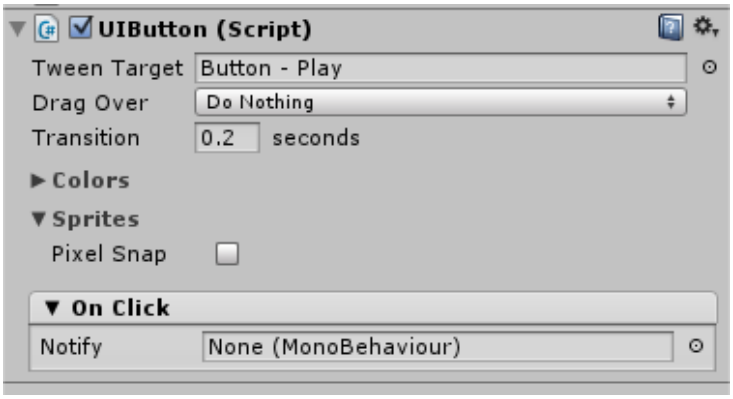

Imagen 13: Inspector del UIButton

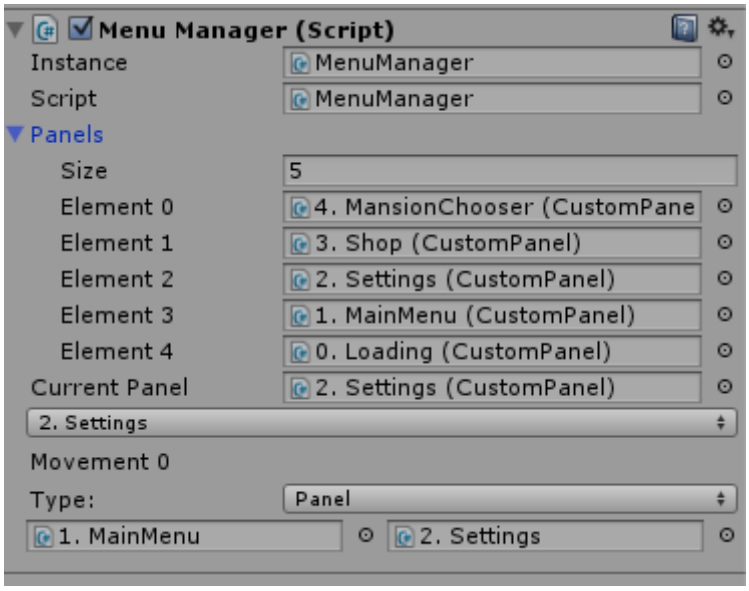

Imagen 14: Inspector del MenuManager

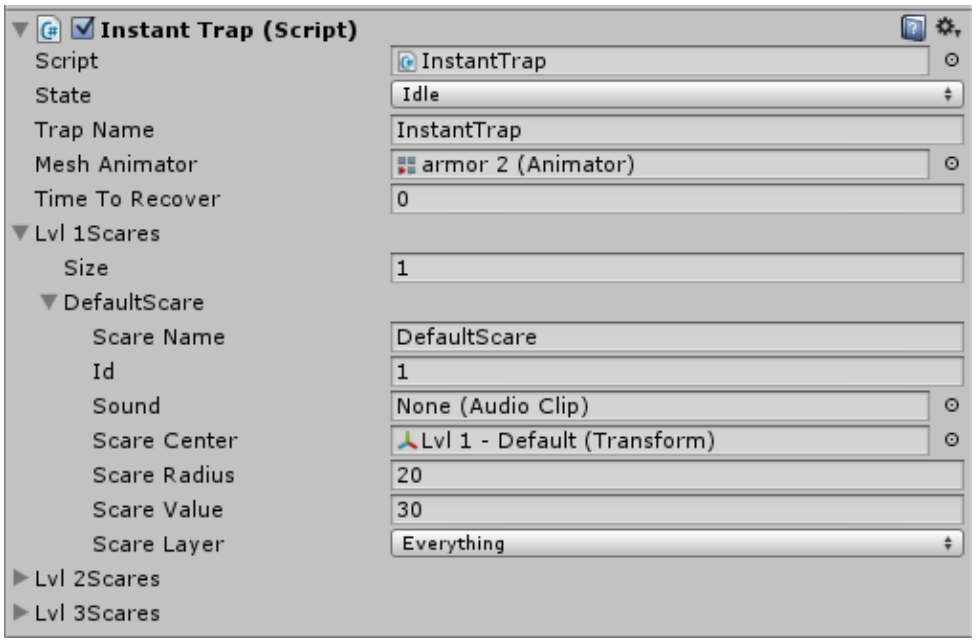

Imagen 15: Inspector de una trampa instantánea

8. Conclusiones y trabajos futuros

Durante la realización de este trabajo hemos adquirido experiencia y una serie de herramientas que sin duda adelantarán la creación de futuros proyecto. Todos los videojuegos requieren de unas interfaces aunque solo sean menús muy simples y con este trabajo se agilizarán los movimientos entre los paneles. Además sin la necesidad de crear nuevos *scripts* ayudaremos a llevar un orden dentro de la jerarquía del proyecto.

Debido a que la tecnología avanza a pasos desmesurados, y no solo de tecnología, sino también de las necesidades de los videojuegos o de los clientes que los demandan. Por ello vamos a realizar una lista de tareas que podrían mejorar estos sistemas:

1. Dar opción de diferentes movimientos entre paneles
2. Crear nuevos *prefabs* que permitan tener configuraciones básicas ya programadas y facilitar el inicio de los poryectos.
3. Recientemente han salido muchas mejoras dentro de la interfaz de Unity3D, que con su versión 4.6 añade un sistema de interfaces que mejora en rendimiento y compatibilidad lo proporcionado por *NGUI*. Por eso otro campo de ampliación sería pasar o ampliar este gestor de menús para usar esta tecnología.
4. Para el sistema de trampas se debería mejorar la interacción con las animaciones y se podrían inventar nuevas trampas con implementaciones más diversas.

9. Bibliografia

[1] Unity3D. [Internet] Disponible en: http://Unity3d.com/es

[2] API de Unity [Internet] Disponible en: <http://docs.Unity3d.com/ScriptReference/

[3] Descarga gratuita de Unity. [Internet] Disponible en: http://unity3d.com/es/unity/download

[4] Tienda de *plugins* de Unity. [Internet] Disponible en: https://www.assetstore.unity3d.com/en/

[5] Tutoriales de extensión de editor Unity. [Internet] Disponible en: https://unity3d.com/es/learn/tutorials/modules/beginner/editor

[6] Web de NGUI [Internet] Disponible en: http://www.tasharen.com/?page_id=140/

[7] API de NGUI [Internet] Disponible en: http://docs.Unity3d.com/ScriptReference/

[8] Descarga de NGUI. [Internet] Disponible en: http://www.tasharen.com/get.php?file=NGUI

[9] Tutoriales de NGUI. [Internet] Disponible en: http://www.tasharen.com/forum/index.php?topic=6754.0