UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

Scares For Sale: Diseño y desarrollo de un videojuego en Unity 3D. Inteligencia artificial y animación

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: David Fernández Molina

Tutor: Javier Lluch Crespo

2013/2014

Resumen

Este proyecto desarrolla la creación de la inteligencia artificial del videojuego Scares For Sale, así como, la implementación de una herramienta para el diseño e implementación de Inteligencias Artificiales de manera gráfica. Además añade optimizaciones tempranas en el desarrollo del juego a nivel de gestión de memoria y gráfico.

Palabras clave: inteligencia artificial, IA, estado, optimización, Unity3D, videojuego

Abstract

This project develops the creation of the artificial inteligence for the videogame Scares For Sale, together with the implementation of a tool for the graphic design and later implementación of Artificial Inteligence in other projects. Furthermore, it adds early memory and graphic optimizations in the development of the videogame.

Keywords : artificial inteligence, AI, state, optimization, Unity3D, videogame

Agradecimientos

A mi familia por su apoyo, comprensión y fuerza a lo largo de todos estos años.

A mis amigos y compañeros, sin los cuales no hubiese podido completar el desarrollo de este proyecto.

A Javier Lluch Crespo por su tutorización y paciencia a lo largo del proyecto.

Tabla de contenidos

Contenido

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.

El desarrollo del videojuego se divide en tres partes, siendo la parte encargada de los sistemas de entrada y audio (*Diseño y desarrollo de un videojuego 3D: Audio e introducción a LeapMotion*) realizada por Sergio Alapont Granero, las interfaces y trampas dentro de la mansión (*Diseño y desarrollo de un videojuego 3D: Interfaces y trampas)* por Agustín Esteve Aguinot y la inteligencia artificial y las optimizaciones (*Diseño y desarrollo de un videojuego 3D: Inteligencia Artificial)* por David Fernández Molina. La supervisión del proyecto se realizó bajo la aprobación de Javier Lluch Crespo.

La parte a desarrollar en este proyecto comprende el desarrollo de la inteligencia artificial (IA) de los enemigos, creando para ello una extensión gráfica del editor con la que facilitar la creación de IAs tanto en este videojuego como en otros. Pero también la aparición de estos enemigos en el escenario mediante la creación de un sistema de oleadas que permita a un diseñador de niveles especificar la configuración adecuada en cada pantalla.

Además, se implementarán diversas optimizaciones tanto de memoria como gráficas con las que obtener un mejor rendimiento del videojuego (Fotogramas por segundo) debiendo ser aplicadas en una fase temprana del desarrollo.

La estructura de la memoria comienza en la siguiente sección, *Estado del Arte*, donde se realiza una comparación entre los diferentes motores existentes en la industria y los lenguajes de programación que pueden ser utilizados en Unity3D.

Después, en la sección *Antecedentes* se presenta el motor y lenguajes utilizados para después dar una introducción a los fundamentos del motor Unity3D.A continuación se realiza una breve explicación de la programación de extensiones de editor dentro de Unity3D, así como, las aproximaciones más actuales a la programación de inteligencia artificial en los videojuegos.

El siguiente punto es la sección *Análisis y Diseño*, donde se presentan los problemas a resolver y se diseñan soluciones haciendo uso de las herramientas que *Unity3D* brinda a los desarrolladores.

Realizado el diseño de las herramientas y componentes, en el apartado *Implementación* se detalla la codificación dada a las funcionalidades exigidas así como las limitaciones surgidas a la hora de implementarlas.

Por último, en los apartados *Resultados* y *Conclusiones y trabajos futuros* se realiza una comparativa entre los objetivos establecidos cuando se inició el proyecto y los logros realmente conseguidos, así como, las posibles mejoras a realizar en las soluciones dadas.

2. Objetivos

Aunque Unity es uno de los motores de videojuegos más completos del mercado los equipos de desarrollo y en particular los programadores siempre requieren de simples scripts incluso herramientas propias que agilicen el proceso de desarrollo.

En el diseño de la inteligencia artificial (IA) de Scares For Sale se decidió ceñirse a la idea anterior para así facilitar la implementación de la IA en posteriores juegos. Para ello se establecieron los siguientes objetivos:

Creación de una herramienta gráfica con la que diseñar la inteligencia artificial en base a una máquina de estados

Ampliación de la herramienta gráfica para convertir la máquina definida a una serie de clases/*scriptso* donde un programador pueda completar la lógica restante

Implementación de la herramienta como plugin integrado dentro de Unity para así poder exportarlo como un package y poder ser utilizado en varios projectos cómodamente.

Uso de la herramienta para definir e implementar la IA de los enemigos del videojuego Scare For Sale.

Junto con la inteligencia artificial los modelados y animaciones dan personalidad y carisma a los personajes de los videojuegos. En Scares For Sale se hará uso del sistema *Mecánim* de Unity para definer las transiciones entre las animaciones aplicadas a los modelos de los enemigos:

Uso de *Mecánim* para la definición del funcionamiento de las animaciones y su activación mediante código.

Independientemente de la plataforma de ejecución, un videojuego actual debe ejecutarse a un ritmo fijo superior a los 30 *frames* por segundo (FPS), esto quiere decir que su lógica principal debe ejecutare como mínimo 30 veces por segundo. Con ello se consigue la fluidez necesaria para que la sensación de juego sea cómoda y satisfactoria para el ojo humano.

La eficiencia es incluso más importante cuando una de las plataformas objetivo son los dispositivos móviles, debido al carácter limitado de las baterías así como por las limitaciones de espacio en memoria y disco.

En el caso de los videojuegos la eficiencia resulta ser tan importante que se trata en una fase especial del desarrollo en la que se realizan optimizaciones tanto gráficas como de memoria y de código.

En Scares For Sale, al tratarse del desarrollo vertical de un videojuego y no una versión completa ser realizarán optimizaciones de memoria y gráficas definidas en estos objetivos:

Creación de un pool genérico de objetos con acceso desde el editor para reducir el coste de instanciación de entidades (enemigos).

Uso del pool de objetos para la implementación del sistema de oleadas con el que gestionar la aparición de enemigos en los diferentes niveles desde el editor.

Uso de *shaders* móviles con los que reducir el coste de renderización del entorno de la mansión.

3. Estado del arte

Durante la última década, con la aparición de los *smartphones*, se ha popularizado el género *casual* en el mercado de los videojuegos convirtiendo este sector en el más rentable dentro de la industria del entretenimiento. A esto se le ha añadido la aproximación de los distintos entornos de desarrollo a los creadores permitiendo un incremento tanto en la variedad como en el número de aplicaciones en los mercados.

En particular, se decidió realizar el juego *Scares For Sale* tomando como referencias algunos videojuegos de éxito del género *Toser Defense*. Estas referencias corresponden a: *Plants Vs Zombies* (2009), *Orcs Must Die* (2011) y *Luigi's Mansión: Dark Moon* (2013).

3.1. Comparativa con otros motores

Los entornos de desarrollo más empleados en la actualidad se agrupan en *Unreal Engine*, *Unity3D* y *Cry Engine*. A la hora de plantearse el desarrollo de un videojuego es necesario decidir si se va a realizar desde la nada o, por el contrario, se va a emplear un sistema como base para la implementación. En este último caso el usuario se enfrenta a la toma de una segunda decisión, la elección de dicho sistema. Para ello se debe conocer cuál es el abanico de posibilidades al alcance y escoger aquel cuyas prestaciones más se adecuen a las necesidades del juego. Por norma general todos los motores suelen ofrecer las mismas herramientas con mayor o menor rendimiento, en cuyo caso el usuario a de escoger aquel cuya usabilidad le sea más cómoda.

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.

En la tabla 1 puede verse una comparativa resumen sobre los pros y contras que Unreal Engine[1] presenta sobre Unity [2]:

	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. Gran comunidad [3] donde buscar ayuda y consejos.
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: Comparativa. Unreal VS UNity3D

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 Huntert 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.

En la tabla 2 se muestra una comparativa resumen sobre los pros y contras que Unreal Engine [4] 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, sufiente 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. Gran comunidad [3] donde buscar ayuda y consejos.
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: Comparativa. Cry Engine VS Unity3D

3.2. Comparativa de lenguajes disponibles en Unity

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.

Las tablas 3, 4 y 5 muestran las principales ventajas e inconvenientes de los lenguajes utilizados en Unity3D.

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: C#, ventajas e inconvenientes

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: UnityScript, ventajas e inconvenientes

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: Boo, ventajas e inconvenientes

4. Antecedentes

Previa a la realización del proyecto se tuvo que investigar acerca de las distintas tecnologías que iban a ser empleadas para el mismo, realizando, para ello, una revisión exhaustiva de las herramientas que dichas tecnologías ofrecían y un estudio acerca de la usabilidad de las mismas y la compatibilidad con el *framework* sobre el que está construido el entorno *Unity3D*.

4.1. Motor de videojuegos utilizado

Se ha elegido Unity 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 otro 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 como 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.

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.
- 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 articifial) por ellos de manera más profesional y similar a lo que haría una empresa de videojuegos que utilizase el motor.

4.2. Lenguaje utilizado

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.

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).

4.3. Fundamentos del motor Unity

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 son la ventana del proyecto, la ventana de la jerarquía, el inspector, la ventana de escena y la de juego.

La ventana del proyecto es como un explorador de archivos 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 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 una ventana que 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 mas comunes como los elementos primitivos, vectores 2D y 3D, curvas de animación, colores, y más.

La ventana de escena contiene una representación del mundo 3D que conforman todos los *assets* instanciados del videojuego. Esta ventana es muy útil sobretodo 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.

La ventana del juego muestra lo que se verá a la hora de iniciar la escena seleccionada. *Unity* puede ejecutar el juego dentro del editor y será en esta ventana donde simularemos que estamos jugando. 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 ejecutará el juego dentro de esta ventana, y el botón de pausa que pausará 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. Otras ventanas son la ventana de la consola, que muestra los fallos y te redirige a la línea de código correspondiente, y ventanas de configuración de algún *plugin*.

4.3.2. Escena

Una escena en *Unity* se trata de una composición de *assets*, *scripts [5]* 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 ademas 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, el jugador, puntuaciones, etc...) y no quisiéramos depender de una base de datos como intermediario, existe una instrucción (d*ontdestroyonload*(*gameobject*)) que le indica a *Unity* que ese objeto no debe destruirse al cambiar de escena. Evidentemente tendremos que tener en cuenta que objetos perdurarán entre escenas para poder configurarlas sin problemas.

Otra manera de intercambiar datos entre escenas es usar la clase p*layerprefs*. Esta clase se trata de una abstracción de una tabla hash para poder guardar números enteros, de coma flotante, y cadenas de texto. Para ello le asignaremos a cada valor una clave, que será una cadena de texto. Esta clase es persistente por lo que además también podemos usarla para diferentes sesiones de juego. También existen *plugins* que permiten guardar objetos enteros en *playerprefs*.

4.3.3. El gameobject y sus componentes

Un *gameobject* o instancia de objeto de juego es la unidad básica que conforma *unity*. Un *gameobject* va a almacenar una serie de componentes que dotarán de comportamiento a este objeto. Cada *gameobject* tiene 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 esta 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[6]. 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 iskinematic 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 audiolistener colocado en la escena y a una distancia determinada. Aqui en el audiosource 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 audiosources 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 [6]. 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 script. 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 start. 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 awake.
- **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 update. 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. Inteligencia Artificial en videojuegos

La inteligencia artificial para videojuegos se compone de una serie de técnicas y algoritmos que permiten aportar interactividad, variedad y rejugabilidad a los mismos a través de enemigos, aliados o el propio entorno.

Comúnmente estas técnicas proveen de un comportamiento adaptativo a las acciones del jugador, por lo que su estructura básica es la de procesar cierta información de entrada codificada y realizar una acción en respuesta a esa entrada. Realizando este ciclo continuamente cuando el jugador modifica el estado del juego consigue darse la sensación de interactividad e inteligencia al usuario.

La implementación de la IA (Inteligencia Artificial) de un videojuego puede llevarse a cabo mediante varias técnicas, las más *ad-hoc* utilizan una enumeración (*enumerate*) con la que codificar los posibles estados junto con un switch para variar el comportamiento dependiendo del estado. Otras técnicas más avanzadas son las máquinas de estados (*Finite State Machines* o *FSM*) y los árboles de comportamiento (*Behaviour Trees*), ambos serán explicados en el siguiente apartado realizando una comparativa entre ambos.

4.4.1. Máquinas de estados y Árboles de comportamiento

Las máquinas de estados son una de las técnicas más utilizadas en la informática a lo largo de la historia y esto se debe a la sencillez de su planteamiento y por tanto su comprensión.

En inteligencia artificial una máquina de estados (Imagen 1) está formado por un conjunto de nodos (estados), transiciones y datos de entrada. Los nodos contienen acciones o comportamientos a aplicar cuando se alcanza ese estado. Las transiciones permiten el desplazamiento entre los diferentes nodos comprobando el cumplimiento de ciertas condiciones sobre el estado actual y los datos de entrada.

Los datos de entrada pueden ser el estado del mundo o entorno del videojuego tras ser afectado por la acción del jugador o directamente las entradas físicas del usuario.

Las máquinas de estados permiten definir patrones de comportamiento para los desarrolladores de forma rápida, y a su vez permiten ser ampliadas a posteriori con sencillez. En cambio tienen un gran inconveniente, la escalabilidad, una inteligencia compleja y variable puede ser muy difícil de representar con estados y complicar aún más su posterior comprensión al tratarse finalmente de una gran red.

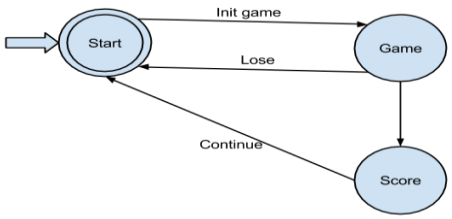

Imagen 1: Máquina de estados típica

Por otra parte, los árboles de comportamiento [7] está formado por una estructura de nodos que se recorre desde arriba a abajo, en la que los nodos intermedios son condiciones de control y los nodos hoja representan acciones, de esta forma un estado viene dado por un camino desde la raíz superior a una hoja. Un ejemplo de árbol de comportamiento de la herramienta Behave2 [8] (plugin disponible en la *Asset* Store de Unity3D [9]) para Unity3D puede verse en la imagen 2.

La curva de aprendizaje de los árboles de comportamiento es superior a la de las máquinas de estados debido a los diferentes tipos de nodos y sus comprobaciones, además su comprensión resulta ser más difícil debido a que se debe recorrer una rama completa para ver las posibles acciones de la inteligencia. Pero a diferencia de las máquinas de estados los árboles resultan ser mucho más sencillos de escalar y permiten crear comportamientos más complejos y variables.

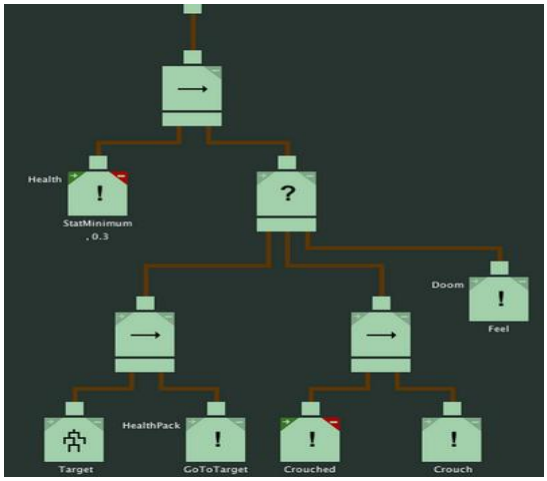

Imagen 2: Árbol de comportamiento en Behave2

4.5. Extensiones del editor

Al igual que un componente de Unity necesita heredar de Monobehaviour para poder sobrescribir ciertos métodos, como el *Start, Update y FixedUpdate*¸una extensión de editor [10] (ejemplo disponible en la imagen 3) es un script en c# que hereda de la clase *Editor* (con la que poder sobrescribir las *callbacks* del editor de Unity), importa la librería UnityEditor para poder crear interfaces dentro del propio editor y está localizado en una carpeta dentro del proyecto denominada "*Editor*".

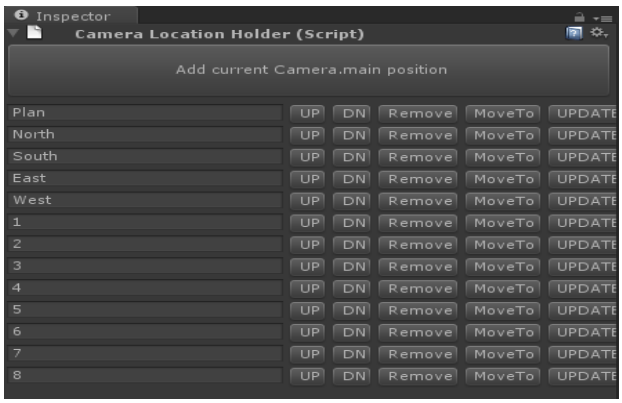

Imagen 3: Extensión de editor

De esta manera dispondremos de las siguientes funcionalidades con las que desarrollar la extensión requerida:

- **OnGUI:** *Callback* realizada por Unity cada vez que la interfaz del editor debe ser redibujada. En este método se de incluir toda la lógica que gestiona la interfaz de la extensión utilizando los *layouts* y elementos propios del editor de Unity. Los elementos de interfaz que se utilizan durante el juego también pueden ser utilizados aquí pero el uso de los componentes del editor aporta funcionalidades propias del Sistema Operativo a los mismos, como cambiar de uno a otro utilizando el tabulador.

  Es importante recordar que el orden de dibujado de los elementos se realiza en el orden en que se declaran dentro de este método, así un botón que declare antes que otro aparecerá por detrás del siguiente. En caso de querer volver a pintar todos los elementos inmediatamente se puede realizar una llamada *EditorWindwe.Repaint()*.
- **OnEnable** y **OnDisable:** Llamada que se realiza cuando la ventana de la extensión que se ha creado se muestra y se minimiza o cierra, respectivamente.
- **EditorWindow.GetWindow(typeof(nombreScriptExtensión)):** Con esta llamada se recupera la instancia existente del tipo pasado. Sirve para "reciclar" la instancia de una ventana en el editor o en caso de no existir crearla.
- **[MenuItem ("Window/My Window")]:** Utilizada para hacer que la nueva extensión aparezca en el menú de Unity en el Sistema Operativo utilizado. La nueva extensión aparecerá bajo la jerarquía especificada dado que es una EditorWindow [11].
- **BeginWindows() y EndWindows():** Estas funciones marcan el inicio y fin del área de ventanas no modales dentro de la propia ventana de la extensión. Entre el inicio y el fin se debe realizar tantas llamadas a GUILayour.Window como ventanas se quiera, pudiendo especificar una función propia de dibujado a cada una de las ventanas definidas, como puede verse en la imagen 4:

```
// Main GUI Function
function OnGUI () {
        // Begin Window
        BeginWindows ();

        // All GUI.Window or GUILayout.Window must come inside here
        windowRect = GUILayout.Window (1, windowRect, DoWindow, "Hi There");

        // Collect all the windows between the two.
        EndWindows ();
}

// The window function. This works just like ingame GUI.Window
function DoWindow () {
        GUILayout.Button ("Hi");
        GUI.DragWindow ();
}
```

Imagen 4: Ejemplo de código BeginWindows() y EndWindows()

- **GUILayout:** Clase con métodos para definir *layouts* dentro del método OnGui, como ares con *scroll* vertical u horizontal, botones, cajas, etiquetas, etc
- **EditorUtility:** Conjunto de métodos estáticos para dar un mejor acabado a las extensión del editor mediante diálogos, popup menus, barras de carga, lectura y guardado de ficheros en el Sistema Operativo, etc.

5. Análisis y diseño

En una etapa previa a la implementación de los distintos puntos del proyecto se realizó una fase de diseño en la que se analizaron los problemas a resolver así como las distintas formas de solventarlos para obtener una solución lo más eficiente y simple posible, con el fin de poder emplearla en posibles proyectos futuros.

Las soluciones planteadas se han ideado con el objetivo de que los usuarios ajenos al proyecto sean capaces de utilizarlo con el menor tiempo de aprendizaje posible, pudiendo facilitar así su uso a nuevos programadores y reducir el tiempo de desarrollo.

5.1. Editor de máquinas de estados

La programación de la inteligencia artificial para videojuegos constituye una de las partes del desarrollo con menos patrones o buenas prácticas a seguir debido a que las funcionalidades requeridas a una IA varían mucho de un proyecto a otro. Esta variabilidad dota a los programadores de una gran libertad a la hora de programarlas pudiendo utilizar los métodos más básicos para inteligencias sencillas (como los mencionados en el apartado 4.4) o haciendo totalmente necesario el uso de librerías o *plugins* externos para inteligencias multijugador donde la eficiencia y escalabilidad del número de agentes es imprescindible.

Tratando de abarcar el mayor rango posible de tipos de proyectos en este trabajo se ha realizado el diseño e implementación de un *plugin* o una extensión del editor para Unity haciendo uso de máquinas de estados, debido a que su fácil comprensión y uso permiten crear IAs para proyectos de pequeño y mediano tamaño rápidamente. Sin olvidar que el uso de estados en comportamientos para enemigos suele ser el más utilizado con el objetivo de permitir al jugador descubrir el patrón de comportamiento o actuación del enemigo.

Para lograr ese objetivo se diseñó una extensión de editor denominada FSM Editor (Finite State Machine Editor) con interfaz gráfica que contuviera las siguientes funcionalidades:

- **Creación y borrado**: Crear una máquina de estados desde cero destruyendo la actual mostrando una ventana modal si esta no se ha guardado previamente.
- **Añadir estados:** La adición de estados a la máquina permitiendo su libre colocación en el espacio proporcionado por su la ventana del editor.
- **Eliminación de estados:** Borrado de estados y de todas las transiciones que entren o salgan del mismo.
- **Añadir transiciones:** Añadir transiciones entre cualquier par de estados siendo fácilmente reconocible el sentido de la transición.
- **Borrar transiciones y cancelaciones:** Posibilidad de borrar transiciones, así como, cancelar la creación o borrado de una transición
- **Generación de la IA en clases:** Para un fácil uso de la inteligencia diseñada gráficamente el plugin permite la generación de una serie de clases equivalente

a los estados definidos junto con un controlador que limite las transiciones entre ellos y ofrezca un acceso completo a los estados/clases generados.

Uno de los principales objetivos de este diseño con interfaz gráfica es la de facilitar la colaboración de diseñadores y programadores en un mismo proyecto, dado que el plugin permite a un diseñador plasmar fácilmente su idea o la especificación de un cliente y su posterior implementación por parte de un programador tomando como base las clases generadas por el plugin y los comentarios dados por el trabajo previo del diseñador.

Para cumplir este objetivo y ofrecer las funcionalidades requeridas teniendo en cuenta las limitaciones del editor de Unity se decidió implementar un sistema dividido en dos partes: El editor y las clases padre de IA generada (Editor y AI Framework respectivamente, como puede apreciarse en la imagen 5), consiguiendo con ello dotar de suficiente modularidad a la IA generada como para que el futuro usuario del plugin pueda modificarla cómodamente y libremente.

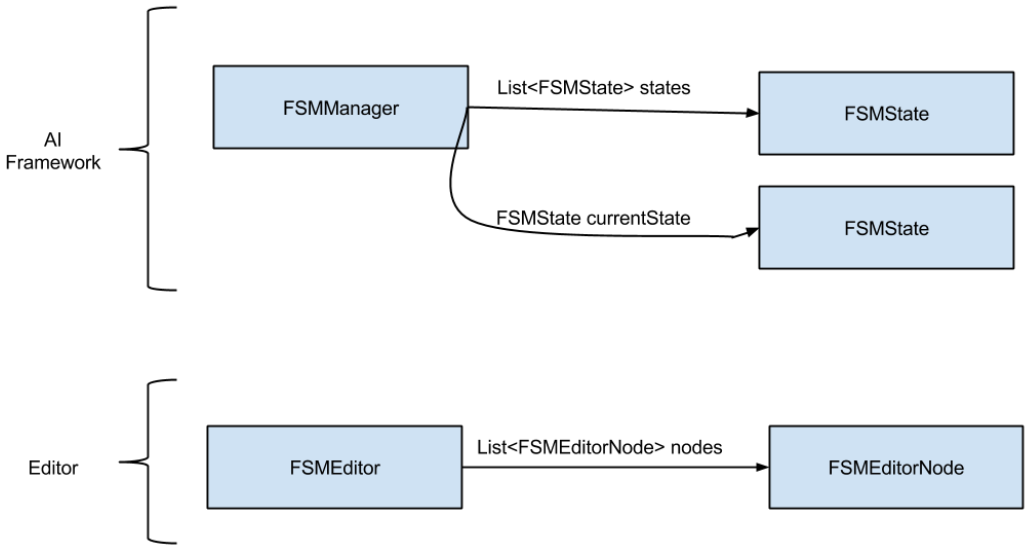

Imagen 5: Diseño FSMEditor y AI Framework

De esta manera cuando el usuario decida guardar la intelgencia diseñada generará con ello una serie de clases donde los nodos definidos en la interfaz serán transformados en un conjunto de clases que heredan de FSMState y que tendrán por nombre el dado al nodo que representan. El FSMManager será el encargado de permitir o prohibir las transiciones entre estados conforme a lo especificado en el editor.

Cabe mencionar que aunque el principal uso de la extensión de editor es la facilitar el diseño e implementación de inteligencias artificiales su funcionalidad permite crear máquinas de estados para todo tipo de trabajos como puede ser la implementación del mecanismo de un arma dentro de un videojuego y sus diferentes estados: Disparo y animación, tiempo de recarga junto con animación, tiempo de desenfundado, etc.

5.2. Inteligencia artificial de los enemigos de Scares For Sale

En Scares For Sale el jugador tiene el objetivo de ahuyentar a los visitantes de una antigua mansión en venta utilizando los diferentes objetos situados en las plantas de la mansión para así asustarlos cuando se sitúen cerca de los objetos. De esta manera los visitantes resultan ser los enemigos del videojuego necesitando una inteligencia artificial con la que el juego suponga un reto abordable para el jugador.

Tratando de crear una inteligencia artificial cuyos estados fuesen reconocibles por el jugador aun habiendo varios enemigos por pantalla se realizaron varios diseños basándose en el movimiento del personaje como base para los diferentes estados (andando, corriendo, caminando distraído, estacionario, huyendo, etc) pero finalmente se optó por un diseño basado en el estado anímico del personaje siendo así más fácil asociar el estado del mismo a una animación creada por el artista 3D del proyecto.

Comúnmente en los videojuegos el jugador derrota a sus enemigos cuando estos pierden la totalidad de sus puntos de vida o salud, representados a nivel de código por un entero o un valor decimal. En el videojuego creado en este proyecto los enemigos saldrán huyendo de la mansión cuando su valor de cordura llegue a cero. Este valor indica cuantos sustos han sufrido los visitantes a mano de las trampas del jugador y se recuperará poco a poco si el visitante no es asustado en un periodo de tiempo determinado, representando que su estado anímico se está recuperando.

Utilizando esta aproximación para diseñar a los enemigos el conjunto de estados resultante fue:

- ***Looking* (Mirando):** Estado inicial de la IA representando el estado "normal" o calmado del enemigo. En él los sustos del jugador merman la cordura del enemigo en la cantidad especificada por el susto a diferencia de otros estados donde los sustos afectarán más al enemigo debido a que está en un estado más nervioso o excitado.

- ***Interested* (Interesado):** En este estado el enemigo a pasado cerca de un objeto (trampa) del jugador y se queda observándolo mientras sigue caminado más lentamente hacia el final de la mansión. Al estar interesado en el objeto un susto proveniente del mismo disminuirá la cordura multiplicando por 2 el valor del susto original.
  El estado puede ser alcanzado desde el estado *looking* pues al ir caminando calmadamente puede pasar a estar atraído por un objeto de la mansión sin ningún tipo de miedo. Igualmente cuando el enemigo pierda interés volverá al estado *looking* o si por el contrario su cordura ha disminuido enormemente pasará al estado *Afraid,* el cual se explicará a continuación.

- ***Afraid* (Asustado):** Este estado representa al enemigo cuando su cordura está muy baja, estando por ello muy nervioso, alertado y preparado para cualquier susto, debido a ello el daño recibido por cualquier susto del usuario es reducido

a la mitad, consiguiendo con ello que el ahuyentar del todo a un visitante sea algo más costoso.
Además en él el enemigo recupera una cantidad determinada de cordura cada 0.5 segundos pudiendo recuperar la suficiente hasta volver al estado *looking* si el jugador no lo vuelve a asustar en el tiempo necesario.

El estado puede ser alcanzado desde el estado *looking* o el *interested* y cómo puede deducirse puede volver al estado *looking* si la cordura lo permite.

- ***BeingScared* (Siendo asustado):** En este estado el enemigo está parado mostrando la animación de susto y por lo tanto se impide la posibilidad de que otro susto reduzca su cordura y de la misma forma no se recupere mientras dure la animación.
  Puede ser alcanzado desde los estados *looking, interested y afraid* y vuelve al estado del que se transitó a él debido a que en él no se puede perder cordura ni recuperarla.

- ***Running-away* (huyendo):** Estado que representa al enemigo cuando su cordura a llegado a cero, no puede perder ni ganar más cordura y en él se reproduce la animación de huir despavorido hasta que sale de la casa donde el enemigo es destruido.

La máquina de estados resultante puede observarse en la imagen 6:

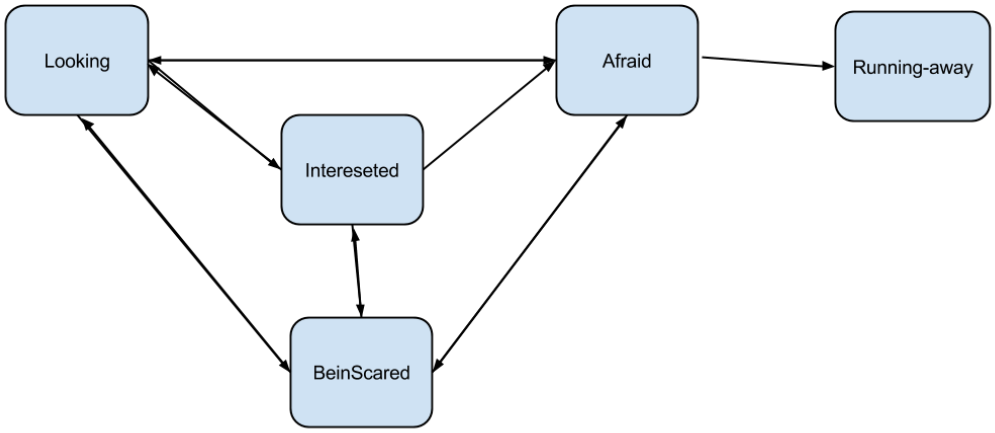

Imagen 6: Diseño IA del visitante

Para el movimiento del personaje por la mansión se planteó la posibilidad de implementar el algoritmo A* unido a una maya 3D dentro de la mansión pero más tarde se decidió utilizar la herramienta ***NavMesh*** de Unity junto con una serie de puntos o *waypoints* situados en el espacio 3D de la mansión por los que la IA debería pasar obligatoriamente hasta su camino al tejado de la mansión donde haría perder la partida al jugador al llegar.

La *NavMesh* de Unity, ilustrada en la imagen 7, puede ser traducida por una malla de navegación y permite delimitar de forma sencilla las zonas o superficies por las que un

agente que tenga adjuntado el componente *NavMeshAgent* puede caminar. Además el *NavMeshAgent* hace públicos una serie de métodos que permiten mover un agente por esa maya con solo especificar un punto del mundo en 3D (Vector3) esquivando obstáculos (*NavMeshObstacles*) y otros agentes en el cálculo de la ruta más óptima hacia el punto. El agente utilizará la NavMesh para ir viajando entre los diferentes puntos de visita obligados situados en el nivel por el diseñador.

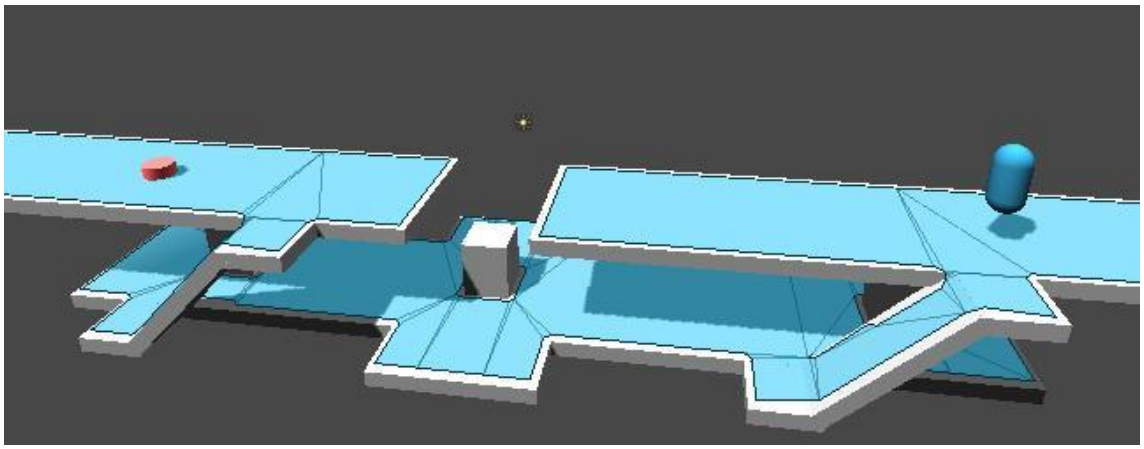

Imagen 7: Ejemplo de uso de NavMesh en Unity3D

Para las animaciones de los visitantes se estipularon 2 animaciones por enemigo, la animación de caminado y la de susto, dado que la animación de caminado sería reproducida a mayor velocidad para dar la sensación de correr en el estado de huida de la mansión.

Las transiciones entre animaciones y su activación fueron realizadas con el sistema ***Mechanim*** de Unity, sistema que permite editar dichas transiciones como si de una máquina de estados se tratase haciendo uso del componente *Animator*[17], cuyo uso y explicación se darán en el apartado de implementación (Apartado 6).

5.3. Oleadas enemigas

Una vez se diseñaron los enemigos básicos y la inteligencia artificial de estos, se comenzó el diseño del script encargado de la generación de enemigos en un nivel.

Debido a que el juego pertenece al género de los Tower Defense se desea que los enemigos se generen por lo que se conoce como oleadas. Una oleada está formada por varios grupos de enemigos diferentes cada uno con una cantidad de enemigos dada y cuya aparición esté espaciada por un tiempo tal que el jugador necesite calcular una estrategia mínima de uso de los objetos repartidos por la mansión sin que sea imposible derrotarlos acabar con todos los enemigos.

A su vez un nivel está formado por varias oleadas, donde una oleada comienza al finalizar la última y tras esperar un tiempo de fijado en el que mostrar la oleada actual y el número de oleadas totales del nivel que se está jugando.

Puesto que en una misma mansión se deben jugar varios niveles ofreciendo así al jugador la posibilidad de acostumbrase a la distribución de la mansión pudiendo plantear mejores estrategias, se decidió realizar un script en Unity3D modificable desde

el editor evitando la necesidad de codificar un nivel diferente cada vez que se necesite diseñar un nivel.

Así el script debería permitir especificar el número de oleadas de un nivel, así como, para cada tipo de enemigo dentro de la oleada, los enemigos que aparecerán y después de cuánto tiempo, desde el inicio de la oleada, deben aparecer.

5.4. Optimizaciones

La fase de optimización suele ser una de las últimas fases del desarrollo de videojuegos, debido a que una vez se ha completado el juego puede realizarse un estudio exhaustivo del rendimiento del mismo, encontrando cuellos de botella en las diferentes partes del mismo: Programación, modelados, luces, texturas, etc y planteando soluciones para ellos.

Dado el desarrollo vertical de Scares For Sale, esta fase no podrá ser aplicada dentro proyecto actualmente descrito pero si se aplicarán optimizaciones tempranas, es decir, optimizaciones que afectan a la base del desarrollo y deben hacerse cuando comienza la implementación del juego.

En el caso de Scares For Sale las optimizaciones aplicadas serán principalmente dos: Pool o piscina de objetos y el "bakeado" o lightmapping de luces.

5.4.1. Pool de objetos

Siendo el lenguaje de programación elegido para Scares For Sale C#, hay que recordar que su gestión dinámica de memoria, y por tanto el uso del recolector de basura (*Garbage Colector* o GC), puede ocasionar en ciertos momentos caídas de fotogramas o incluso un uso inapropiado de la RAM, algo crítico en dispositivos móviles.

Esta mala gestión tiene lugar al instanciar objetos y después dejar de usar su referencia esperando a que el recolector de basura pase a recogerlos para destruirlos y liberar así la memoria que utilizaban.

Por ello se diseñó un script que se encargara de optimizar la gestión de memoria en lo referente a los objetos instanciados dinámicamente dentro del juego, como es el caso de los enemigos. Este script debería trabajar de forma genérica para cualquier tipo de objetos y tendría que estar formado por tripletas Objeto – Número – Incremento , donde una tripleta indica el número de objetos de ese tipo que se utilizarán durante la ejecución de un nivel y en qué cantidad se debe aumentar el número de obejtos existentes si ya no quedan objetos disponibles y se realiza una petición al *pool*. De esta manera el script instanciaría los objetos en la inicialización del juego (mientras se muestra un menú de carga o mientras el usuario está realizando una tarea independiente), los desactivaría y los guardaría en una colección a la espera de que otros scripts le pidan uno de esos objetos. En el momento en que un script necesite uno de esos objetos el pool elegiría uno de ellos, lo activaría y lo entregaría al script para su uso. Posteriormente cuando se desee destruir ese objeto el script debería avisar al pool y este desactivaría el objeto reseteando todas sus variables estando preparado para un próximo uso.

Con este proceso se evita la instanciación de objetos durante la ejecución del *gameplay* del juego así como la ejecución del recolector de basura al estar todos los objetos siempre referenciados desde el pool y al hacer un uso responsable de ellos.

5.4.2. *Lightmapping* o Bakeo de luces

A diferencia de la optimización anterior el *lightmapping* [18] es una herramienta ofrecida por Unity y únicamente disponible en su versión PRO. Esta herramienta aumenta significativamente el rendimiento del videojuego al permitir quitar las luces del videojuego y aplicar sus efectos sobre las texturas de los objetos del entorno como si estas estuvieran activas o aún siguiesen en la escena.

Su uso implica la eliminación de las sombras arrojadas sobre los elementos del escenario debido a que la luz se "hornea" (bakea) sobre la textura previamente a la ejecución del juego. Aun así, si se quiere tener alguna sombra puntual en una zona del nivel con la que dar mayor dramatismo a la escena puede realizarse el lightmapping sobre el nivel y además dejar algunas luces activadas para que estas apliquen sombras sobre los modelados circundantes, siempre y cuando se tenga en cuenta el coste en tiempo de ejecución de estas luces y sombras.

Esta optimización suele realizarse cuando el nivel o entorno esté totalmente finalizado pero antes que el resto de optimizaciones para que los diseñadores de nivel y los artistas del proyecto puedan tener runa imagen previa del acabado visual del juego de manera temprana.

Para realizar este proceso deben marcarse todos los objetos de la escena que sean estáticos en cuanto a luces se refiere, es decir, que nunca se verán afectados por luces en tiempo real. De la misma forma numerosas opciones como la resolución de la sombras, el número de rayos a utilizar, el número de rebotes permitidos pueden ser especificadas, las cuales pueden aportar un mayor realismo a la escena pero también pueden aumentar el tiempo de "horneado" significativamente. La duración del mismo aumenta proporcionalmente con el número de luces, el de objetos y con el del número de rayos utilizado (máximo de 3). La pantalla de configuración del lightmapping puede verse en la imagen 8 mostrada a continuación:

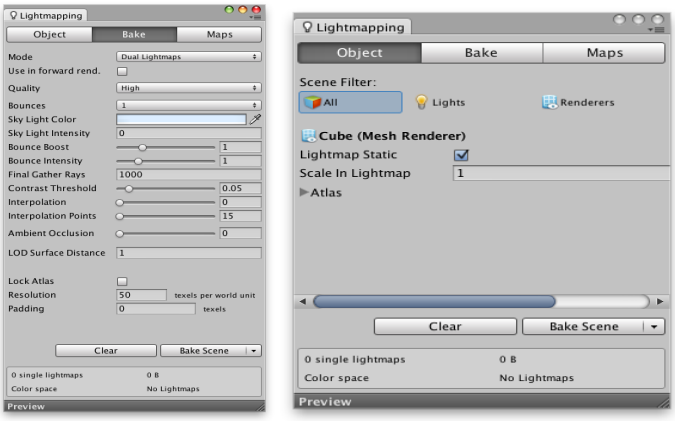

Imagen 8: Opciones de lightmapping en Unity3D

6. Implementación

Una vez se dispuso del diseño de los componentes necesarios, se procedió a la implementación del código de los mismos, haciendo uso en los casos necesarios de las herramientas ofrecidas por Unity tanto de la versión Free como de la Pro utilizando en este último caso la prueba gratuita de 30 días.

6.1. Editor de máquinas de estados

La programación del editor gráfico de máquinas de estados (FSMEditor) fue dividida en dos clases, como se puede observarse en el apartado de diseño 5.1., la clase FSMEditor y la clase FSMEditorNode.

La primera de estas clases contiene el código para dibujar el editor principal y para la gestión de las funcionalidades de generación de código, estados de la máquina y creación y destrucción de estados y transiciones.

Las variables contenidas en esta clase están descritas en la tabla 6:

Nombre	**Tipo**	**Descripción**
FSMPrefabName	String	Nombre base de la IA cuando esta sea transformada en un conjunto de scripts listos para ser modificados por el programador.
editorNodes	List<FSMEditorNode>	Lista de *Editor Nodes* que contiene el conjunto de estados de la máquina.
createTransitionOriginNode deleteTransitionOriginNode	FSMEditorNode	Ambas variables almacenan el nodo origen de creación o destrucción de una transición, en el momento en que el nodo de destino es seleccionado, la transición se crea o se destruye y estas variables pasan a tomar el valor null.
FSMSaved	Bool	Booleano que indica si el código que la representa ha sido generado o no desde la última modificación.
scrollPos	Vector2	Posición en el espacio 2D de la ventana. Indica el punto de origen de la venta de scroll de la extensión del plugin.
nextStateId	Int	Entero que identifica al próximo estado a crear con un valor único utilizado como nombre por defecto.
arrowTexture	Texture2D	Textura o imagen utilizada para indicar el sentido de una transición.

mouseRect	Rect	Rectángulo que indica la posición del ratón en el espacio 2D en todo momento. Es uso de un rectángulo y no de un Vector2D se debe a que el método que dibuja curvas toma como entrada dos rectángulos.

Tabla 6: C#, Variables clase FSMEditor

El método más importante de esta clase es *OnGUI*¸ este método (heredado de la clase de Unity Monobehabiour) es el encargado de dibujar todos los componentes en el orden correcto en cada frame además de comprobar cuando se ha pulsado o accionado alguno de ellos para llamar a la funcionalidad requerida. Para mostrar su implementación sin entrar en detalle se muestra a continuación una versión esquematizada de este método con la funcionalidad encapsulada en su interior:

1) Cancelar cualquier acción si el botón escape ha sido pulsado
2) Dibujar el botón de "nueva máquina de estados" y resetear todas las variables del editor si este botón es pulsado
3) Dibujar el botón de generación de código de la IA y generar las clases pertinentes si este es pulsado, preguntando previamente por el nombre base de la IA así como el directorio donde se guardarán los scripts generados.
4) Dibujar el botón de creación de estado y crear y añadir un estado (*FSMEditorNode*) al editor si este botón se pulsa
5) Comenzar a dibujar el área de *scroll* de 1000 x 1000 pixeles, con tamaño ajustable por el usuario, donde se mostrarán los nodos (ventanas) creados por el usuario
6) Comienzo del área de ventanas no modales dentro de la *scroll view*
7) Dibujar cada nodo dentro de la lista de estados del editor
8) Dibujar las curvas (transiciones) entre nodos
9) Fin del área de ventanas no modales dentro de la *scroll view*
10) Si se ha seleccionado un nodo como origen para crear una transición dibujar una línea azul entre ese nodo y el cursor
11) Si se ha seleccionado un nodo como origen para crear una transición dibujar una línea roja entre ese nodo y el cursor
12) Fin del área de *scroll*

Finalmente esta clase también incluye la gestión de la ventana del editor, es decir, es la encargada de realizar la carga de los recursos necesarios como la textura de la *arrowTexture* cuando se maximiza o se abre la ventana, así como la destrucción de todos los recursos utilizados cuando la ventana se cierra, llamando incluso al recolector de basura de C# para una mayor eficiencia dentro del editor.

De igual manera la clase *FSMEditorNode* representa a un nodo dentro de la máquina de estados y se encarga de la gestión visual de los nodos, así como de mantener una referencia a todos los nodos a los que este puede transitar.

Las variables y funcionalidad dentro de la clase *FSMEditorNode* son las siguientes:

Nombre	Tipo	Descripción
id	int	Entero que identifica de manera inequívoca en el sistema al nodo.
name	string	Nombre del nodo y por tanto nombre de la clase que será generada que represente a este nodo.
transitions	List<FSMEditorNode>	Conjunto de nodos a los que puede transitar el nodo actual.
window	Rect	Referencia al nodo visual o ventana dentro del marco del editor de máquinas de estados. Utilizado para dibujar los componentes necesario dentro de él.
editor	FSMEditor	Referencia al gestor de la extensión del editor para notificar de los cambios producidos sobre el nodo, como su destrucción y para la consulta de variables del editor como *createTransitionOriginNode* ó *deleteTransitionOriginNode*.

Tabla 7: Variables clase FSMEditorNode

En este caso la lógica de dibujo y de gestión del nodo queda encerrada en el método *DrawNodeWindow*, el cual es llamado por el FSMEditor dentro de una llamada a GUI.Window() pasado como un puntero a función, asegurándose de realizar esta llamada entre las instrucciones BeginWindows() y EndWindows() para hacer que el nodo sea una ventana no modal.

La lógica de un nodo se encarga de dibujar una *input label* que indique el nombre del nodo, pero también del dibujo de los botones Join y Delete, los cuales si el valor de la variable createTransitionOriginNode/deleteTransitionOriginNode es distinto de null crearan o destruirán una transición respectivamente hacia este nodo. En caso contrario dan valor a esas variables poniendo como nodo origen de creación o destrucción de una transición y activando por el dibujo de una línea entre el nodo y el ratón en el editor.

Finalmente un botón de destrucción del nodo es dibujado, el cual avisa al editor principal para que destruya cualquier referencia a este nodo desde cualquiera de los otros nodos existentes. El resultado de ello puede verse en la imagen número 9.

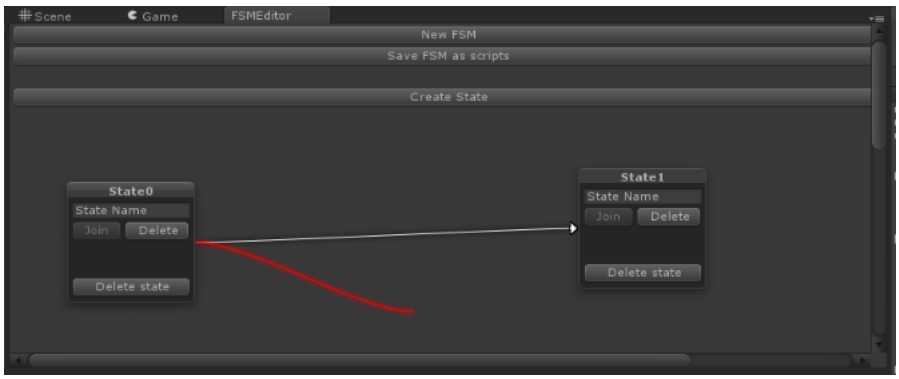

Imagen 9: Plugin FSMEditor, borrado de una transición

Por otra parte, como se ha explicado anteriormente en este apartado y en el de diseño del editor (5.1), la clase *FSMEditor* es la encargada de generar el código resultante de la IA diseñada gráficamente. Esta generación de código tiene lugar en el método *SaveScripts*() dado que transforma los nodos que tiene almacenados a sus equivalentes en la IA haciendo uso de las clase FSMState y de la clase FSMManager.

La generación dinámica [12] de código se lleva a cabo utilizando la librería System.IO de C# creando y escribiendo [13] un nuevo fichero/script/clase [14] por cada nodo, que hereda de la clase FSMState. De la misma forma se genera una clase que hereda de FSMManager encargada de la gestión de la IA, que contiene una referencia a todos los estados que la IA debe contener, con el nombre base de la IA especificada al generar los scripts.

Un segmento de la generación de código para los estados puede verse en la imagen 10 a partir de la línea 203:

```
178     public void SaveScripts(){
179         //Create gameobject and add FSM manager
180         FSMSaved = true;
181         //GameObject fsm = new GameObject ();
182         //fsm.AddComponent<FSMManagger> ();
183 
184         //Generate state scripts
185         string basePath = EditorUtility.SaveFilePanel ("Save FSM as scripts",
186                                                        "",
187                                                        "FSMBaseName",
188                                                        "cs");
189 
190         int fsmNameLength = basePath.Length - basePath.LastIndexOf ("/");
191         Debug.Log (fsmNameLength);
192         string fsmName = basePath.Substring (basePath.LastIndexOf ("/")+1, fsmNameLength - 4);
193         Debug.Log (fsmName);
194 
195 
196         foreach(FSMEditorNode node in editorNodes){
197             Debug.Log(basePath.Substring(0,basePath.Length-fsmNameLength+1));
198             string stateFileNameWithPath = basePath.Substring(0,basePath.Length-fsmNameLength+1)+node.name+"State.cs";
199             if( File.Exists(stateFileNameWithPath) == false ){ // do not overwrite
200                 using (StreamWriter outfile =
201                        new StreamWriter(stateFileNameWithPath))
202                 {
203                     outfile.WriteLine("using UnityEngine;");
204                     outfile.WriteLine("using System.Collections;");
205                     outfile.WriteLine("");
206                     outfile.WriteLine("public class "+node.name+"State : FSMState {");
207                     outfile.WriteLine(" ");
```

Imagen 10: Código FSMEditor, Generación de los estados FSMState

La clase FSMManager almacena el estado actual de la máquina y ejecuta el código contenido en el método *Act()* del mismo mientras el método *Reason()* del estado no indique la transición a otro estado. Si el método *Reason()* decide realizar una transición el manejador se encargará de comprobar si es una transición válida, cambiando el estado actual en caso afirmativo o mostrando un error en caso contrario.

La clase FSMState obliga al programador a encapsular la lógica del estado en dos métodos que debe sobrescribir: *Reason()* y *Act()*. El primero debe estar formado por la lógica encargada de decidir cuando este estado debe transitar a otro. El segundo debe contener la lógica a ejecutar mientras la máquina se encuentra en este estado. Ambos son llamados por el FSMManager al ser arrastrado como componente a un gameObject de la escena.

Además la clase FSMState contiene nuevos métodos que facilitan la implementación de la Inteligenciar Artificial como son *DoBeforeEntering()* y *DoBeforeLeaving()* los cuales están declarados como virtuales y pueden ser sobrescritos por el programador

mediante el modificador *override* para que sean llamado por el gesto de la IA (FSMManager) antes de entrar al estado y antes de transitar a otro respectivamente.

El acabado final puede observarse en la imagen 11 donde se muestra una máquina de estados diseñada con el editor:

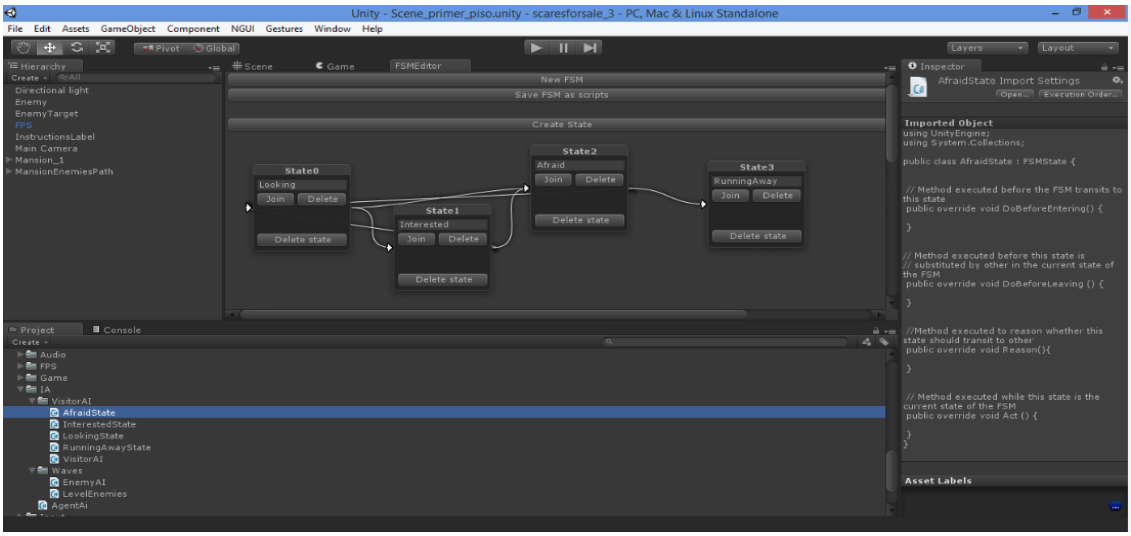

Imagen 11: Plugin FSMEditor, ejemplo de uso

6.2. Inteligencia artificial de los enemigos de Scares For Sale

En Scares For Sale se desarrolló una única IA enemiga que todos los diferentes enemigos compartirían variando la cantidad de cordura y la velocidad de regeneración de la misma durante el estado *Afraid*.

Para la implementación de la inteligencia artificial de los enemigos se utilizó el plugin descrito en el sub-apartado anterior. Su uso simplificó enormemente la codificación debido a que el paso del diseño escrito al virtual es casi inmediato dada su similitud con las máquinas de estados convencionales. El resultado dentro del editor puede verse en la imagen 12:

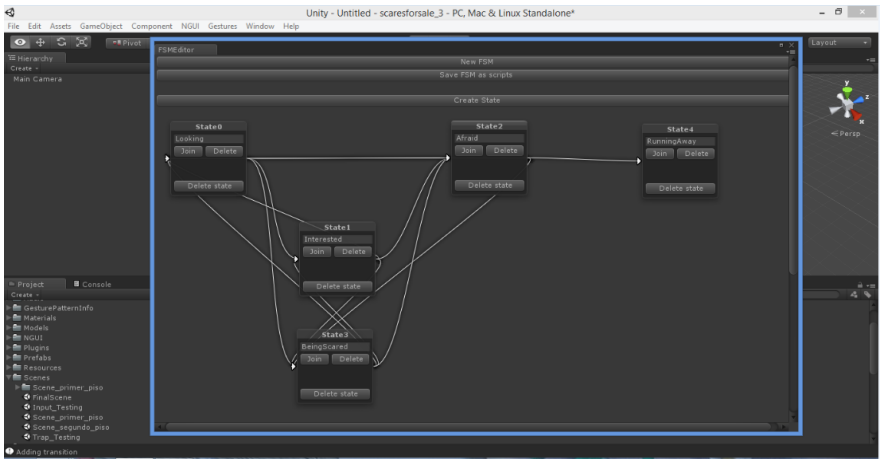

Imagen 12: Plugin FSMEditor: Máquina de estados de la IA del visitante

La IA fue generada con el nombre de VisitorAI por estar hecha para los visitantes y fue almacenada en el directorio del proyecto “Assets/IA/VisitorAI”. Cabe destacar que el uso de los scripts generados es muy sencillo, basta con arrastrar el manejador (VisitorAI en este caso) sobre un gameObject o prefab para que este añada el resto de estados al mismo gameObject cuando se inicie la ejecución, pero también se pueden añadir todos los componentes generados al gameObject si así se desea, en cuyo caso el manejador los detectará y no añadirá estado extra.

El código generado por la extensión fue completado con una serie de variables y métodos para gestionar la cordura de los enemigos. A continuación, en la tabla 8, se explicarán las variables contenidas en la clase VisitorAI por ser la más importante y de la que dependen el resto de estados:

Nombre	**Tipo**	**Descripción**
sanity	Float	Cantidad actual de cordura en este enemigo.
MaxSanity	float	Cantidad máxima de cordura permitida.
target	Transform	Objecto transform que indica el objetivo actual al que el enemigo ha de dirigirse.
agentNavMesh	NavMeshAgent	Acceso al componente NavMeshAgent del enemigo con el que iniciar o detener el movimiento del mismo hacia el objetivo.
agentPath	List<Transform>	Lista de puntos en el espacio 3D que el visitante debe alcanzar almenos una vez para poder llegar hasta el punto más alto dela mansión.
currentPathPoint	int	Entero que indica la posición en la lista agentPath del objetivo a perseguir actualmente. Es utilizado para poner el siguiente valor adecuado a la variable target.
exitPoint	Transform	Punto donde se debe dirigir el enemigo si su cordura ha llegado a cero. Al llegar a este punto el enemigo es destruido.

Tabla 8: Variables clase VisitorAI

Junto con las variables mencionadas se añadieron métodos para dar acceso a las trampas del entorno a la cordura del enemigo, en concreto dos: Scare(int amount) que resta la cordura pasada como argumento teniendo en cuenta el estado actual y RecoverSanity(int amount) utilizado por la propia IA para recuperar cordura durante el estado Afraid. El movimiento del enemigo también se gestionó desde dentro de esta clase con métodos como WalkToPoint(Vector3), StopWlaking() y WalkToNextWaypoint(), los cuales eran accedidos desde los estados generados.

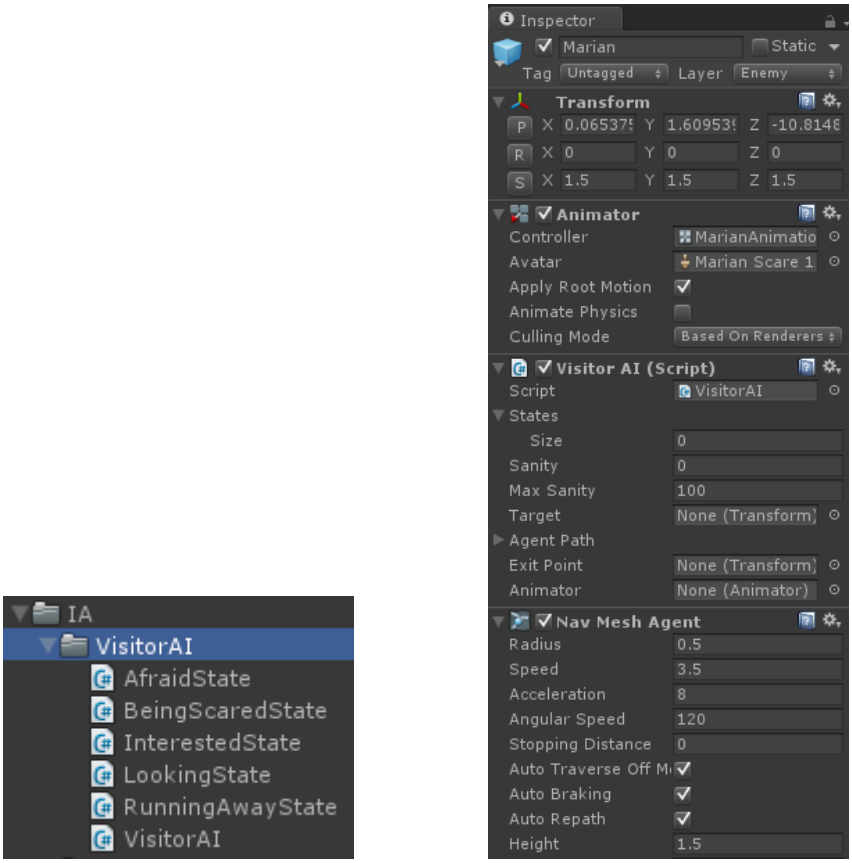

Imagen 13: Clases generadas e integración dentro del gameObejct visitante

La imagen 13 muestra los scripts generados par la IA diseñada dentro del editor de *Unity* así como el uso del Script *VisitorAI* dentro de un *gameObejct* en la escena.

6.3. Oleadas enemigas

El script encargado de gestionar las oleadas enemigas se denominó *LevelEnemies* y tal y como se diseñó está implementado para ser fácilmente modificable desde el editor de Unity, pudiendo configurar un nivel del videojuego sin tener que escribir una línea de código adicional.

Este script utiliza dos clases internas para definir la clase oleada (*EnemyWave)* y para definir el enemigo a instanciar junto con su tiempo de aparición, es decir, el momento en el que debe crearse (WEnemy).

Además de esas dos clases, el script LevelEnemies, incluye una lista de oleadas (EnemyWaves), una lista de waypoints que se asignará a la IA enemiga cuando esta sea instanciada para que recorra la mansión pasando por unos puntos prefijados, así como, variables para almacenar la oleada actual.

Haciendo uso de los elementos mencionados, LevelEnemies instancia la primera oleada utilizando el método *Start()* de Unity y comprobando constantemente en el método *Update()* si la oleada actual ha terminado llamando al método *IsWaveOver()* de la clase *EnemyWave*.

Una vez que una oleada se ha terminado se inicializa la siguiente de la misma forma que se instanció la primera oleada, llamando al método *LaunchWave(Enemy wave)*, esta función utiliza la corrutina *SpawnEnemy(Wenemy, Wave)* para realizar la instanciación del enemigo tras esperar el tiempo especificado en el objeto WEnemy e incrementar en uno el número de enemigos instanciados de la oleada actual Wave.

El script final debe añadirse en un gameObject de la escena para que este sea ejecutado cuando la escena se cargue. El resultado al arrastrar dicho script a un objeto puede verse en la imagen 14.

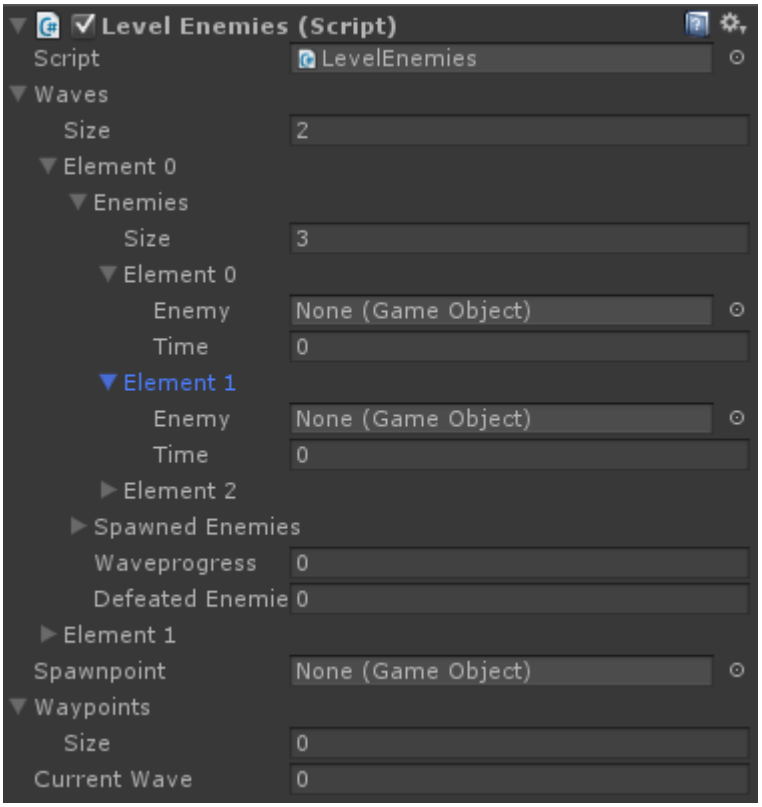

Imagen 14: Script LevelEnemies dentro de un gameObejct en la escena

6.4. Optimizaciones

6.4.1. Pool de objetos

Al igual que en el script de oleadas enemigas *LevelEnemie,* en el script *ObjectPool* se usó la definición de una clase interna para la creación del *pool* de objetos. Dicha clase se denomina ObjectPoolEntry y almacena la información del GameObject a instanciar, junto con la cantidad necesaria y el incremento del número de objetos de este tipo en caso de que no queden suficientes en el pool.

ObjectPool inicializa una lista de listas de Gameobjects dentro del método Start(). Para ello recorre un array de objectos *ObjectPoolEntry* y para cada uno de ellos crea una

lista instanciando *GameObjects* del tipo especificado en la variable *Prefab* de *ObjectPoolEntry* actual con un número igual al especificado en la variable count.

Posteriormente se implementaron dos métodos para coger objetos del pool, así como, para devolverlos al mismo:

El primero de ellos es *GetObejctForType(string objectType, bool onlyPooled)* y recibe el nombre del gameObject que se quiere recuperar y un valor lógico que indica si se quiere ampliar la talla del objeto requerido en caso de que no haya suficientes. Si este valor es igual a True, en caso de no quedar suficientes gameObejcts de ese tipo se devolverá el valor null. Si el valor es igual a False en caso de estar todos los objetos en uso, se ampliará la lista actual en *Increment* unidades y se devolverá una.

El segundo método es *PoolObject(GameObject obj)* y devuelve el objeto pasado como parámetro a su lista origina y desactivándolo, manteniendo así una referencia a él en todo momento.

Por otra parte para evitar que la instanciación de un gran número de GameObejcts llene la vista "Jerarquía" de la escena de Unity, todos los objetos que estén almacenados en el pool, y por tanto desactivados, son colocados como hijos del GameObject *ContainerObject*, un gameObject vacío que se instancia dentro del método *Start()* del pool para una mejor organización de la escena.

El resultado al arrastrar dicho script a un objeto dentro de la escena se muestra en la imagen 15.

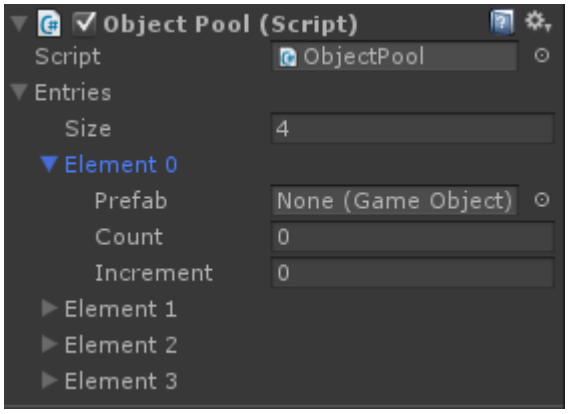

Imagen 15: Script ObjectPool dentro de un gameObejct en la escena

6.4.2. *Lightmapping* o Bakeo de luces

Para aumentar el rendimiento del videojuego Scares For Sale se realizó el *lithmapping* sobre la mansión desarrollada a lo largo de este proyecto. En este caso, debido a la naturaleza del videojuego se marcaron todos los objetos incluidos en la escena como estáticos a ojos del *lightmapping*, debido a que salvo contadas animaciones de rotura o objetos volando ninguno de ellos se movería haciendo que la luz siempre le afecte de la

misma forma, por ello alterar la textura de los modelados añadiéndole el efecto de a luz no varía en nada su aspecto durante la ejecución.

Las luces utilizadas para el lightmapping fueron:

- Luz direccional: Luz direccional general de la escena que imita a la luz proveniente del sol.
- Luces puntuales: Luces situadas en las lámparas del techo en cada uno de los pisos

Y se excluyeron las luces de candelabros de mesa, candelabros de pared y la chimenea dado que estas luces se utilizarán en el momento del susto del objeto y se decidió que su efecto anaranjado si afectase temporalmente a los modelos cercanos y al del enemigo.

Una vez completado el lightmapping puede observarse en la imagen 16 como tras desactivar las luces de la jerarquía de escena, la mansión sigue viéndose afectada por las luces y sombres proyectadas durante el proceso.

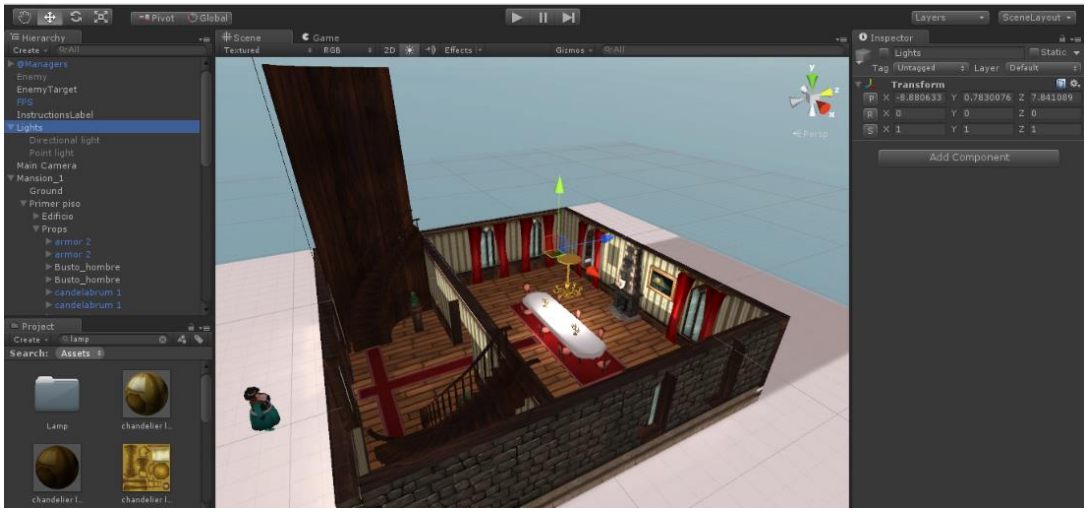

Imagen 16: Escena con lightmaps activados y luces desactivadas

La configuración del *lightmapping* aplicado a la mansión se muestra en la imagen 17:

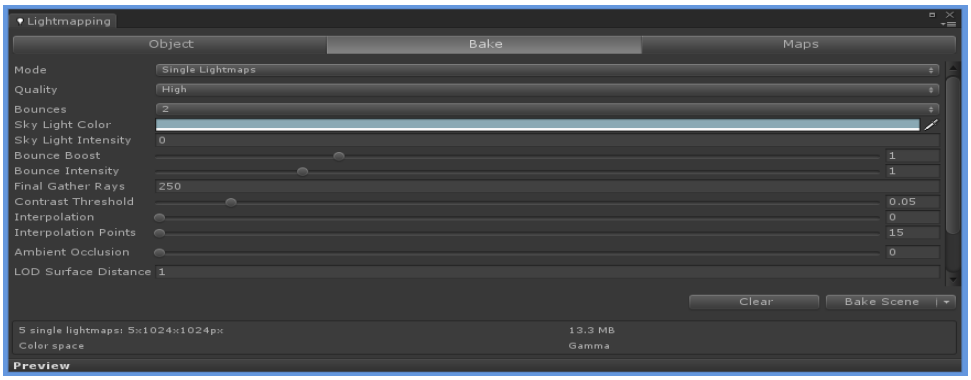

Imagen 17: Configuración del lightmapping en Scares For Sale

4. Resultados

Finalizado el diseño e implementación del editor, la IA, las optimizaciones y su integración dentro del videojuego se obtuvieron ciertas conclusiones sobre el uso y estado de los mismos.

La extensión de editor creada durante el desarrollo del proyecto permite crear máquinas de estados orientadas a la implementación de inteligencias artificiales mediante una interfaz gráfica donde crear nodos y transiciones. Después del diseño dicha máquina puede ser transformada a código de manera dinámica, eligiendo el nombre de la IA generada así como de las clases-estado creadas. Por ello el plugin ha demostrado ser de gran utilidad para el desarrollo de inteligencias artificiales debido a su simplicidad y a su funcionamiento basado en máquinas de estados. Además los scripts generados pueden ser ampliados fácilmente por un programador ajeno a la codificación de la herramienta.

La inteligencia artificial de los visitantes de Scares For Sale quedó finalmente formada por 5 estados: *Looking, Interested, BeingScared, Afraid* y *RunningAway*. Los cuales son los mismos para todo enemigo del juego a pesar de ser visualmente diferentes, por ejemplo el personaje Marian, una anciana de gran tamaño que representa uno de los enemigos más duros dado su curtido carácter y experiencia, utiliza la misma IA que Daniel, un niño de 8 años, pero se consigue cambiar la dificultad para derrotarlos variando los valores de cordura y de regeneración de cordura que pueden tener desde el propio editor, siendo muy cómodo para un diseñador equilibrar la sensación de juego sin la ayuda de un programador. Gracias a ello los enemigos sumados a su velocidad de aparición hacen que los niveles suponga un reto para el jugador.

Las optimizaciones realizas ayudaron a obtener buenos resultados de rendimiento del juego incluso en su temprana fase de desarrollo, consiguiendo, en la última versión estable, 240 *frames* o fotogramas constantes. Esto se debió al uso del *pool* o piscina de objetos evitando caídas de fotogramas al tener que instanciar gameObjects durante el juego y evitando las pasadas del colector de basura de C# al mismo tiempo. Pero por otro lado el *mapeado* de luces al aplicar las luces sobre las texturas y evitar el uso de luces dinámicas supuso una diferencia en el rendimiento de 90 *frames*, dao que se obtenian 150 *frames* en la versión original, perdiendo a costa de ello el efecto 3D que aportan las sombras en tiempo real.

El desarrollo vertical del videojuego Scares For Sale en este proyecto ha servido para que todos los integrantes del mismo hayan podido desarrollar herramientas prácticas y reutilizables en próximos proyecto. A causa de ello el juego se encuentra todavía en un estado temprano de desarrollo con sólo una mansión completa, un enemigo desarrollado (modelado, animaciones, comportamiento, etc) y sólo algunas trampas. Todos estos componentes fueron necesarios para comprobar el correcto funcionamiento de las mecánicas diseñadas, pero no fueron suficientes como para dar un acabado final al videojuego.

5. Conclusiones y trabajos futuros

Gracias a la forma en que se diseñó el proyecto se han obtenido varias herramientas y utilidades que, aunque mejorables, facilitarán el desarrollo de futuros videojuegos haciendo uso del motor *Unity3D* sin necesidad de la compra de extensiones de terceros reduciendo tiempo de desarrollo y coste.

La elección del motor Unity3D resultó ser muy satisfactoria gracias a su portabilidad y a la facilidad de uso de sus componentes así como el gran número tutoriales y soluciones a problemas presentados en los foros de *Unity*. La cantidad de herramientas incluidas en el motor (NavMesh, Colliders, Animator, etc) facilitó enormemente la implementación del movimiento de los enemigos por un escenario variable así como su animación e interacción con elementos del escenario.

La flexibilidad del motor para ampliar su funcionalidad mediante *plugins* resultó ser la característica más utilizada, dado que la posibilidad de exportar esa extensión para ser utilizada en otros proyectos como un *package* ahorrará mucho tiempo en ellos.

Debido a las limitaciones de tiempo impuestas por el trabajo de fin de grado el estado de desarrollo del videojuego es el mínimo para haber podido realizar la integración de los diferentes proyectos personales de los integrantes del proyecto. De esta manera el desarrollo del juego debería completar los siguientes puntos:

- Completar la funcionalidad de todas las trampas disponibles en la primera mansión, así como la adición de nuevas trampas pertenecientes a otros estilos de mansión o épocas.
- Adición de nuevos enemigos como los diferentes integrantes de una familia (niño, niña, padre y madre) así como otro personaje de dificultad elevada representado el marido de Marian.
- Creación de nuevas mansiones con diferentes diseños de niveles y estilos visuales.

Aunque los sistemas desarrollados han sido de gran utilidad para el videojuego desarrollado, durante su uso han surgido ciertas mejoras y ampliaciones que se implementarán en el futuro:

- Posibilidad de guardar la máquina de estados como un fichero con formato propio para su posterior edición desde el editor.
- Incluir los elementos necesarios de forma usable para que el usuario del plugin pueda seleccionar el estado inicial de la máquina y no solamente por código como en la versión desarrollada.
- Creación de una inteligencia artificial con diferentes comportamientos para los enemigos más difíciles del juego también conocidos como jefes finales.

6. Bibliografía

[1] UnrealEngine/Licencias. [Internet] Disponible en: < https://www.unrealengine.com/>

[2] Unity3D/Licencias. [Internet] Disponible en: < http://unity3d.com/ http://unity3d.com/unity/licenses>

[3] Unity3D/Comunidad. [Internet] Disponible en: < http://unity3d.com/community>

[4] UnrealEngine/Licencias. [Internet] Disponible en: < https://www.unrealengine.com/>

[5] Unity documentation (2014). Scripting API [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/>

[6] Unity documentation (2014). MonoBehaviour [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/MonoBehaviour.html>

[7] BehaviourTrees [Internet] Disponible en: < http://en.wikipedia.org/wiki/Behavior_Trees

[8] Behave2 for Unity3D [Internet] Disponible en: < http://blogs.unity3d.com/2014/04/28/tutorial-behave-2-for-unity/>

[9] Unity3D/AssetStore. [Internet] Disponible en: <https://www.assetstore.unity3d.com/en/>

[10] Unity Editor extension (2014). [Internet] Disponible en: < http://code.tutsplus.com/tutorials/how-to-add-your-own-tools-to-unitys-editor--active-10047>

[11] Unity documentation (2014). EditorWindow [Internet] Disponible en: <http://docs.unity3d.com/ScriptReference/EditorWindow.html>

[12] Unity community (2014). Create class dinamically [Internet] Disponible en: < http://answers.unity3d.com/questions/12599/editor-script-need-to-create-class-script-automati.html>

[13] MSDN Library. StreamWriter (Clase) [Internet] Disponible en: < http://msdn.microsoft.com/es-es/library/system.io.streamwriter(v=vs.110).aspx>

[14] MSDN Library. File (Clase) [Internet] Disponible en: <http://msdn.microsoft.com/es-es/library/system.io.file(v=vs.110).aspx>

[16] Unity3D/Navigation. [Internet] Disponible en: < http://unity3d.com/learn/tutorials/modules/beginner/navigation/navigation-overview>

[17] Unity3D/Animator. [Internet] Disponible en: < http://docs.unity3d.com/Manual/Animator.html>

[18] Unity3D/LightMapping. [Internet] Disponible en: < http://docs.unity3d.com/Manual/Lightmapping.html>