Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Asset de inteligencia artificial en Unity

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Carlos Garcia Rodriguez

Tutor: Francisco José Abad Cerdá

Cotutor: Ramon Pascual Mollá Vayá

Septiembre 2018

Agradecimientos

Ante todo, quería empezar agradeciendo a mis padres por haberme dado la oportunidad de cursar una carrera universitaria y por todos los sacrificios que han hecho para que yo llegue hasta aquí y pueda, en un futuro, tener más oportunidades para trabajar de lo que yo quiera.

Por otro lado, también querría agradecer a todos los amigos que he hecho dentro de la carrera y con los que he tenido el placer de cursar ciertas asignaturas, haciendo que estas fuesen más llevaderas, así como más amenas y fáciles de aprender.

También me gustaría agradecer a mi tutor, Don Ramon Pascual Mollá Vayá, por darme la oportunidad de hacer un *TFG* relacionado con temas que me apasionan como son los videojuegos y la inteligencia artificial, así como la guía y consejos que me ha proporcionado durante la realización de este.

Por último, me gustaría hacer un agradecimiento especial a mi hermana y a toda esa gente que, en ciertos momentos, me han apoyado y me han animado para que continuase con la carrera y no la dejase, a pesar de todas las dificultados y complicaciones que iban apareciendo.

Resumen

Al programar una aplicación existe la posibilidad de hacer uso de una *API* para, por ejemplo, hacer uso de imágenes o audio. Esta práctica evita que el programador tenga que crear todas las funciones necesarias para poder utilizar dichas imágenes o audio. En la gestión de la Inteligencia Artificial, esta *API* es inexistente. Cuando se necesita hacer uso de cierta inteligencia artificial en una aplicación, todo el trabajo recae en el programador. Este *TFG* intentará solventar este vacío. El proyecto parte de una *API* de gestión de la Inteligencia Artificial en videojuegos basada en la teoría de máquinas de estados finitos, realizada anteriormente en formato de *TFM* por José Alapont Luján. El objetivo es poder generar un producto final en forma de *asset* (*Game Artificial Intelligence - GAI*) que sea integrado de una manera profesional en cualquier videojuego y emplee los canales comerciales habituales de la tienda en línea del entorno de desarrollo de videojuegos *Unity*. Este *asset* debe poder ser utilizado en diferentes proyectos, centrándose sobre todo en proyectos relacionados con el campo de los videojuegos. El *asset* presentará al programador de videojuegos un interfaz homogéneo de selección de tipo de comportamiento a asignar al personaje. Así mismo, habrá un gestor de *IAs* parametrizable por fichero *XML* en el que contendrá toda la información del tipo de *IA* y la estructura asociada al comportamiento. En el proyecto original se realizó un banco de pruebas muy sencillo que validaba el *API* original mínimamente. No se pudo verificar su uso en entornos de videojuegos reales. En este caso se ha escogido un videojuego real suministrado por *Unity* en el que se configurarán diferentes comportamientos de cada personaje del juego mediante el nuevo *API*. El objetivo es disponer de una caja de arena para comprobar el correcto funcionamiento del *API* de inteligencia artificial en condiciones reales de un videojuego. Por último, se solucionarán los posibles bugs y errores que pudieran surgir al someter al *API* al nuevo entorno, así como ampliar el número de tipos de máquinas de estados implementadas. Para finalizar se realizará un estudio del resultado obtenido para así saber la opinión de las posibles personas que hagan uso de ella y poder avanzar en el desarrollo del *MVP*. Para la realización del campo de pruebas se hará uso del motor gráfico de *Unity*, del lenguaje de programación C#, así como ficheros de configuración en *XML*.

Palabras clave: *FSM*, máquinas de estados, inteligencia artificial, *API*, videojuegos, *IA*, Unity.

Abstract

When you are programming an application, you have the possibility to use an API to, for example, make use of images or audio. This practice prevents the programmer from having to create all the necessary functions to be able to use images or audio. In the management of Artificial Intelligence, this API is non-existent. When you need to use some artificial intelligence in an application, all the work remains on the programmer. This TFG will try to solve this gap. The project is based on a management API of Artificial Intelligence in videogames based on the theory of finite-state machines, previously carried out in TFM format by José Alapont Luján. The objective is to generate a final product in the form of an asset (Game Artificial Intelligence - GAI) that

is integrated in a professional way, in any videogame, and uses the usual commercial channels of the online store of the Unity videogame development environment. This asset should be able to be used in different projects, focusing mainly on projects related to videogames. The asset will present the video game programmer with a homogeneous interface for selecting the type of behavior to be assigned to the character. Likewise, there will be an AI manager that can be parameterized by an XML file which t will contain all the information of the type of AI and the structure associated with the behavior. In the original project, a very simple test bank was made to validate the original API minimally. Its use in real videogame environments could not be verified. In this case, a real videogame provided by Unity has been chosen, in which different behaviors of each game character will be configured through the new API. The objective is to have a sandbox to check the correct functionality of the artificial intelligence API in real conditions of a video game. Finally, possible bugs and errors that could arise when submitting the API to the new environment will be solved, as well as expanding the number of machine types of implemented states. Finally, a study of the result obtained will be carried out to know the opinion of possible people who make use of it and be able to advance in the development of the MVP. For the realization of the test field, the UNITY engine, the C # programming language will be used, as well as configuration files in XML

Keywords: *FSM*, *finite state machines*, *artificial intelligence*, *API*, *videogames*, *AI, Unity*.

Tabla de contenidos

1. Introducción

Pese a que la inteligencia artificial es un campo relativamente moderno, ya que oficialmente se formalizó este término en 1956, es una de las ramas de las ciencias computacionales más desarrolladas en la actualidad.

Sin embargo, a la hora de hablar de inteligencia artificial se tiene que tener en cuenta que este término, dependiendo del campo en el que se esté empleando o incluso dependiendo de un autor u otro, puede tener diversos significados.

Para empezar, se puede hacer una división de la inteligencia artificial en 4 tipos, tal y como explican *Stuart Russel* y *Peter Norvig* en [1], según la función que se quiera que realice el sistema ya sea;

- **Pensar como los humanos** – Estos sistemas tratan de emular el pensamiento humano, y un ejemplo de ello podrían ser las redes neuronales. Las actividades que se pueden vincular con procesos de pensamientos humanos podrían ser actividades tales como la toma de decisiones, resolución de problemas y aprendizaje.
- **Actuar como humanos** – Estos sistemas tratan de actuar como humanos, es decir, imitar el comportamiento humano, como por ejemplo se hace en la robótica.
- **Pensar de forma racional** – Es decir, hacer uso de la lógica, tratando así de imitar o llegar a emular el pensamiento lógico y racional del ser humano; por ejemplo, esto se realiza en los sistemas expertos.
- **Actuar de forma racional** – Tratar de emular de forma racional el comportamiento humano, como se hace en por ejemplo agentes inteligentes.

Es por ello por lo que para simplificar el termino de inteligencia artificial se puede entender como las técnicas o métodos que puede emplear cierta entidad para así poder transmitir un comportamiento que podría clasificarse como lógico o inteligente.

Uno de los campos que más ha desarrollado la inteligencia artificial, además de la robótica, ha sido el campo de los videojuegos. Sin embargo, en este sector, el termino inteligencia artificial se relaja bastante ya que se considera como inteligencia artificial cualquier comportamiento en una entidad del juego que tienda a conseguir un objetivo, sin importar la técnica empleada.

Esta es la diferencia fundamental entre la Inteligencia Artificial académica y la inteligencia artificial en los videojuegos y, tal y como se puede encontrar en [3], mientras que se podría decir que la inteligencia artificial académica busca obtener una solución a un problema que se plantee de la forma más optima, sin importar el coste computacional o tiempo que esto conlleve, en la inteligencia artificial de los videojuegos no se tiene como objetivo que ante cierto problema siempre se llegue a la solución más optima o correcta. Esto se debe principalmente a que, aparte de que los recursos son limitados y no se puede, por tanto, hacer un gran uso del procesador, el tiempo de

respuesta debe ser el menor posible ya que la respuesta se debe de generar en tiempo real, lo cual también hace que el calculo que se realice deba ser el menor posible. Por otro lado, en un videojuego no se busca esa solución óptima de la que se ha hablado, si no proporcionar al jugador un entretenimiento. Si se lo proporciona por ejemplo a un enemigo del videojuego una inteligencia artificial muy optima, lo que pasará es que ese enemigo no fallará ningún golpe, siempre le dará al jugador y por tanto llegará a ser muy frustrante, así como muy aburrido para el jugador, ya que siempre perderá.

Sin embargo, esta inteligencia de la que se dota a un enemigo o un *NPC* (*palabra definida en el diccionario de términos*) debe de intentar no ser ni predecible ni tampoco monótona, puesto que esto hará que sea demasiado fácil o aburrido para el jugador.

En la industria del videojuego el método más común a la hora de emplear inteligencia artificial es el uso de *FSM* (*finite state machines*) o máquinas de estados finitos ya que además de ser muy fáciles de implementar y emplear, como se verá más adelante hay una gran variedad de ellas, ofreciendo así la posibilidad de elegir cual emplear ,dependiendo del tipo de inteligencia se necesite en el proyecto o videojuego a realizar, obteniendo así el resultado más acorde a las necesidades que se tengan.

1.1 Motivación

Para empezar, una de las principales razones por las cuales se ha elegido este trabajo de fin de grado es el hecho de que mientras que hay multitud de *APIs* (*palabra definida en el diccionario de términos*) de las cuales puedes hacer uso ya sea a la hora de emplear audio, imágenes o por ejemplo servicios web, cuando se va a hacer uso de inteligencia artificial para un proyecto, es el propio programador quien tiene que gestionarla, crear las funciones necesarias, decidir que usar para dotar a su proyecto o aplicación de dicha inteligencia así como asegurarse de que dicho funcionamiento sea el apropiado y óptimo.

Es por ello por lo que crear una herramienta que proporcione todas estas facilidades, como podría ser la creación de una *API* de inteligencia artificial, que pueda ayudar a que no se pierda tanto tiempo en un proyecto a la hora de crear dicha inteligencia, así como conseguir que sea tanto sencilla de implementar como de comprender, resulta ser una idea muy llamativa e interesante.

Además, otra de las principales razones es que cuando se está disfrutando de un videojuego, una de las cosas en las que los jugadores más se suelen fijar es en lo bien hechos que están los enemigos, *bosses* (*palabra definida en el diccionario de términos*) así como los *NPCs*, en lo difíciles que son ciertos enemigos, el cómo reaccionan a diversas acciones, como se comportan dependiendo de los cambios que haya en el terreno, las acciones que realizan algunos *NPCs* que están de fondo, como por ejemplo

cocinar o plantar alimentos, o como ciertos personajes consiguen mantener conversaciones con el jugador de forma tan fluidas y casi humanas.

Ejemplo de juegos cuya inteligencia artificial es brillante podría ser *Half Life,* juego que salió en 1998 desarrollado por *Valve Corporation*. En este juego la inteligencia artificial[1] esta tan lograda que pesa a ser tan antiguo hoy sigue teniendo en sus enemigos y *NPCs* una de las mejores inteligencias artificiales implementadas, ya que estos tenían en su código programado incluso un *"olfato"* con el cual eran reactivos frente a diversas situaciones que ocurriesen en su entorno. Si por ejemplo un guardia detectaba con este *"olfato"* el olor de un cadáver este se empezaba a alejar mientras soltaba frases de desagrado. Si por el contrario un enemigo *"olía"* la carne de este cadáver, aunque el jugador estuviese cerca no se acercaba a atacarle, ya que este solo quería alimentarse, y prefería ir a por la carne que a por el jugador. Además de este comportamiento tan fascinante, en este juego hasta las cucarachas poseían cierta inteligencia artificial, ya que, si por ejemplo el jugador se acercaba, estas salían corriendo lejos de él, y si la luz estaba apagada y de golpe se encendía, éstas empezaban a correr aleatoriamente e intentaban esconderse.

Figura 1: Enemigos del videojuego *Half Life*

Otro ejemplo más actual de un videojuego que haga una muy buena implementación de su inteligencia artificial[2] seria *The legend of Zelda Breath of the wild* juego desarrollado por *Nintendo EPD*. En este ejemplo hay algunos enemigos que se sitúan en campamentos y van armados. Si por ejemplo el jugador los desarma, irán corriendo en busca de un arma, si ven una cercana, y si no cogerán rocas o elementos del terreno para poder defenderse, e incluso con el paso del tiempo, cuando el jugador haya derrotado muchos de estos campamentos, en lugar de aparecer enemigos simples cada vez aparecerán enemigos más y más fuertes incrementando la dificultad, dando así la sensación de que al igual que el jugador experimenta una evolución, se va haciendo más

[1] Inteligencia Artificial *Half Life:* *https://areajugones.sport.es/2017/09/27/half-life-incluia-en-el-codigo-de-sus-npcs-la-capacidad-de-olfatear/*

[2] Inteligencia Artifial *Zelda Breath of the Wild*: https://www.mundogamers.com/noticia-asi-finge-zelda-breath-of-the-wild-la-ia-de-sus-enemigos.18397.html

fuerte y va aprendiendo, los enemigos también lo hacen. Por otro lado, los *NPCs* que aparecen también poseen una inteligencia muy bien lograda. Se pueden encontrar personajes en el pueblo realizando tareas, pero si por ejemplo ven un enemigo o empieza a llover, éstos irán corriendo a sus casas para protegerse. También se pueden encontrar viajeros los cuales, si son atacados, a no ser que vayan armados, en cuyo caso pelearán, empezaran a correr en busca de ayuda y si ven al jugador, le pedirán ayuda, dándole una recompensa en caso de que lo haga.

Figura 2: Enemigos del videojuego The legend of Zelda Breath of the wild

A modo de resumen se podría decir que el poder realizar un proyecto relacionado con este tipo de comportamientos, el poder llegar a comprender como consiguen llevar a cabo dichas reacciones que parecen tan reales y humanas, así como implementarlas en un ejemplo en el que sean visibles todas estas acciones, y poder facilitar el que otras personas puedan llegar a crear comportamientos similares en sus videojuegos o proyectos, es la razón principal por la que se ha elegido la realización de este trabajo de fin de grado.

1.2 Objetivos

Este proyecto parte con bastantes objetivos a cumplir, siento el primero de dichos objetos la realización del campo de pruebas en el que se comprobara el correcto funcionamiento de una *API* base que se realizó en 2014 como trabajo final de máster por el alumno José Alapont Luján.

Dicha *API*, tal y como se explica en [4], se implementó mediante el uso de máquinas de estado finitos, y se llegaron a implementar diferentes tipos de ellas, para así tener diversas opciones a la hora de emplear inteligencia en algún videojuego o aplicación. Es

por ello por lo que uno de los objetivos principales será, primero asegurarse del correcto funcionamiento de la *API* y cada uno de sus componentes, así como un comportamiento lógico y esperado en todas las *FSM* implementadas.

También será uno de los objetivos del proyecto la realización de un campo de pruebas más acorde con el objetivo principal de esta *API*, los videojuegos. Es por ello por lo que se realizará, mediante el uso de *Unity*, un nivel de pruebas en el cual se pueda comprobar tanto el correcto funcionamiento de las *FSM* cuando son empleadas por enemigos o *NPCs* como la diferencia entre emplear un tipo de *FSM* u otra.

Tras esto el siguiente objetivo será preparar la *API* para que se pueda subir al *asset store (palabra definida en el diccionario de términos)* de *Unity* para que así diversos usuarios puedan descargársela y hacer uso de ella. Es por esta la razón por la cual además de solventarlos posibles problemas que hayan o puedan surgir en la *API*, otro objetivo será intentar hacer que dicha *API* sea lo más simple, clara y fácil de utilizar por cualquier usuario.

Además, se realizará un estudio en el que se recogerán las opiniones de diversos usuarios, con diferentes conocimientos en el campo de la informática, sobre tanto la utilidad, como la facilidad al utilizar dicha *API*, así como su opinión global de la misma.

Por ultimo y para finalizar también están como objetivos generales tanto la ampliación de conocimientos relacionados con el mundo del videojuego e inteligencia artificial, como ampliar los conocimientos en el uso de base de datos, del lenguaje de programación *C#*, así como la creación de un videojuego utilizando el motor de videojuegos *Unity*.

1.3 Metodología

A la hora de establecer una metodología a seguir en este proyecto se estudiaron las diversas metodologías que se habían visto en la carrera, en asignaturas tales como Ingeniería del Software ,y se decidió que para este tipo de proyectos una metodología ágil seria la que más se ajustaría, puesto que, aunque no haya un cliente como tal, el tutor del trabajo de fin de grado actúa como uno, ya que hay que hacer reuniones periódicas con él, así como mostrarle los avances que se realizan, escuchar su opinión y consejo y realizar cambios, acciones que se realizarían con el cliente si lo hubiese.

Una vez establecida que la metodología a implantar para este proyecto iba a ser una metodología ágil, se decidió que, entre los diferentes tipos de metodologías agiles que existen ,se usarían 2 en este proyecto. La primera seria *SCRUM*, ya que este tipo de metodología parte de la idea de que ocurrirán procesos caóticos y en lugar de eliminarlos, la idea es gestionarlos antes de que ocurran gracias a realizar frecuentes reuniones para asegurar el cumplimiento de los objetivos. Este tipo de metodología sería de gran utilidad para asegurarse del cumplimiento de los plazos y además en caso de aparecer problemas o dudas en la realización del TFG, poder quedar con el tutor y solventarlas rápidamente. La segunda seria *KANBAN* la cual se centra en la idea de

organizar en 3 columnas las tareas que están pendientes, en proceso y terminas, idea que será de utilidad para mejorar la productividad y eficiencia, puesto que será de gran utilidad visualizar que actividades faltan por realizar y no tener que buscar que actividades faltan y cuales ya han sido realizadas.

Además de este tipo de metodologías a realizar se siguieron otros métodos para la realización de este trabajo de fin de grado. Para empezar en cuanto a la realización del campo de pruebas, debido a que se quería mantener el campo de pruebas lo más organizado posible, cada trozo de código fue comentando para, así en futuras ocasiones que se necesite repasar, sea más fácil poder entender que hace cada método y dentro de cada método cada línea de código.

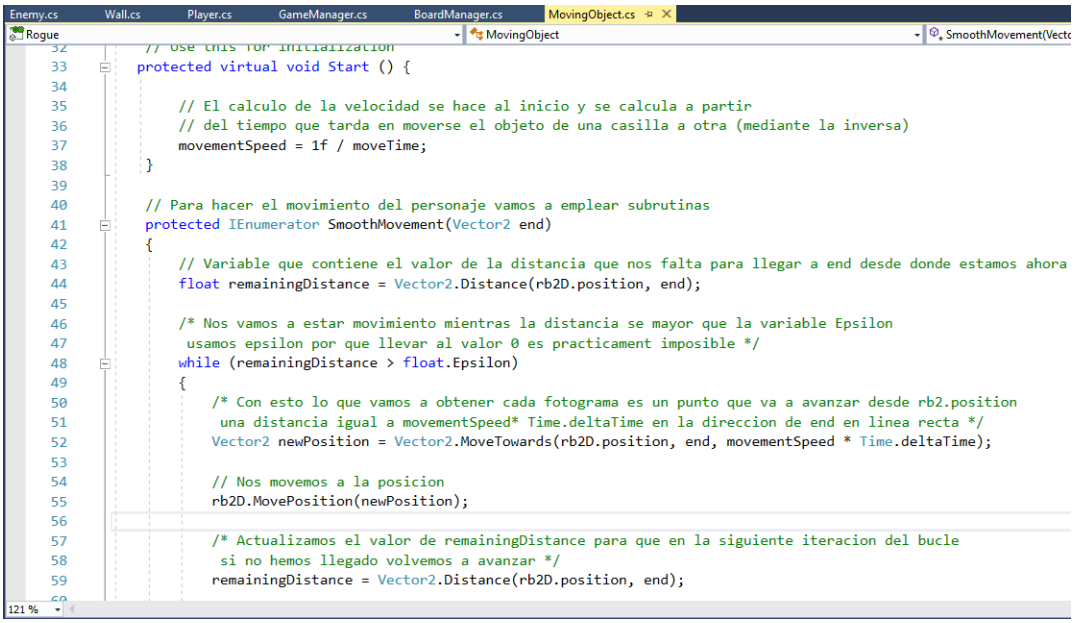


Figura 3: Código comentado

También se siguió este modelo cuando se realizaron los tutoriales explicativos de *Unity*, redactando como se hacen tareas tales como proporcionar movimiento a un personaje o como gestionar y crear las animaciones. Para ello lo que se hizo fue redactar a mano en libretas clasificadas por temáticas los pasos a seguir, así como el por qué hacerlas de esa manera.

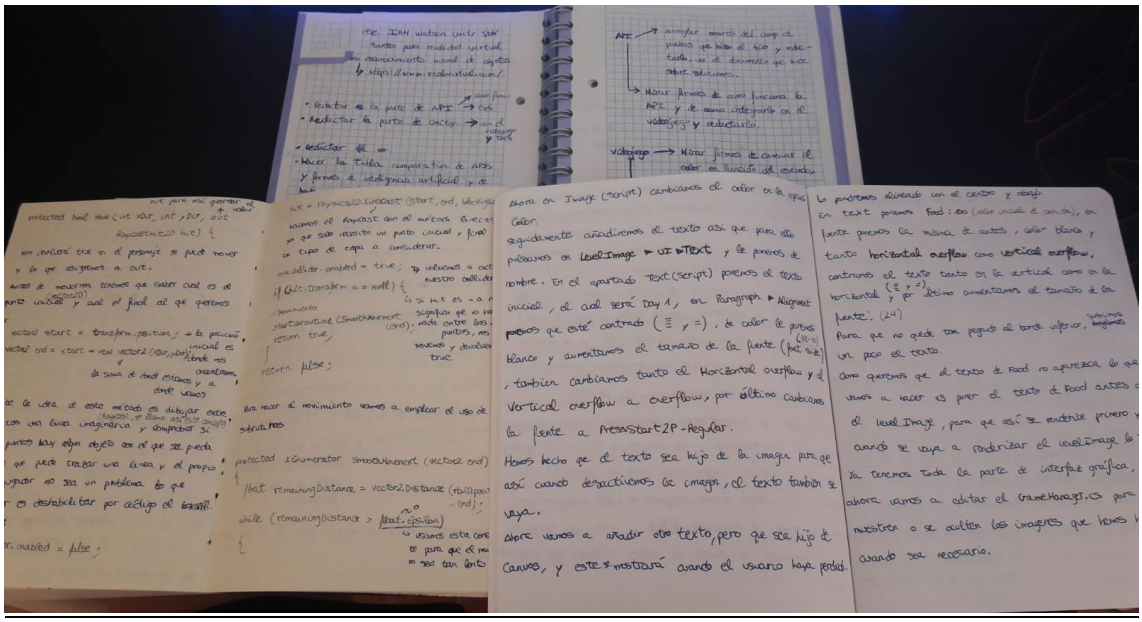


Figura 4: Libretas explicativas

Por último, y para finalizar, a la hora de redactar la memoria se decidió añadir ciertos puntos de interés para que tanto la lectura como la comprensión de esta fueran más fáciles. Estos puntos son un diccionario de términos, en los que se definen tanto palabras concretas de la herramienta *Unity*, como de las máquinas de estados finitos o del mundo de los videojuegos. También se añadió una tabla de figuras para así poder localizar con facilidad las imágenes, tablas o dibujos que se emplearon así como la página donde aparecen y un Anexo donde se añadirán documentos externos que serían de interés leer como por ejemplo las guías de usuario que aparecerán con la descarga de la *API* para tanto entender como emplear la *API*, así como elegir la *FSM* que al usuario le convenga mejor.

1.4 Convenciones

Para la redacción de esta memoria se siguieron una serie de convenciones para que así, de esta forma, el resultado final fuera más cómodo de leer.

La tipología empleada para la realización de esta memoria fue Georgia con un tamaño de fuente 11. Siempre se empleará esta tipología a la hora de redactar el contenido de cada punto, sin embargo, el tamaño de la fuente variará en las siguientes ocasiones. Cambiará a un tamaño 36 cuando se trate del titulo de uno de los bloques de la memoria, a un tamaño 26 si ese bloque contiene subapartados, a un tamaño 20 cuando se trate del título de un subapartado el cual ya era subapartado de uno de los bloques de la memoria y por último a un tamaño 9 cuando sea el titulo de una de las figuras.

Además de esto a lo largo de la memoria se resaltarán en negrita o en colores las palabras que sean importantes tales como etiquetas de los documentos *XML* de los que se esté hablando o el nombre de carpetas o documentos importantes. Además de esto algunas palabras aparecerán en color azul indicando que haciendo *click* a dicha palabra se irá o bien a la sección de diccionario de términos, al anexo o a la bibliografía, o bien a

la sección que indique. Así mismo se empleará la cursiva cada vez que se mencione una palabra que sean siglas, inglesas, que sea un método o trozo de código o que se trate del titulo de uno de los anexos o alguna obra.

También, y para finalizar, se entrecomillará cualquier titulo que se mencione en la memoria, ya sea una obra que forme parte de la bibliografía o bien el título de uno de los documentos que compongan el anexo

1.5 Planificación

Gracias a las metodologías que se eligieron seguir, a la hora de realizar tareas, se tuvo un orden en las que faltaban o se debían realizar con cierta prioridad, así como las que se debían volver a realizar si tras su revisión se comprobaba que o bien se podía mejorar el resultado o no se estaba satisfecho con lo obtenido.

Debido a que este trabajo de fin de grado parte de un trabajo de fin de máster ya realizado, la primera semana se decidió que serviría para leer la memoria de José Alapont Luján, para así intentar entender, no solo el funcionamiento de la propia *API*, si no como estaba estructurada y la razón de esta estructura. Seguidamente para comprender más en profundidad la *API*, y poder así trabajar de una forma más cómoda y productiva, lo que se hizo fue realizar un estudio tanto de las máquinas de estados finitos como de la inteligencia artificial en sí. Es por ello por lo que la siguiente semana lo que se hizo fue buscar tanto bibliografía como páginas webs que pudieran ayudar.

Una vez comprendida la *API* y conociendo un poco más en profundidad sobre, tanto las *FSM* como la inteligencia artificial, y con una base bibliográfica con la que poder hacer referencias y dar credibilidad a los argumentos que se hiciesen, la siguiente semana se decidió que serviría para conocer un poco la herramienta con la que se iba a trabajar para la realización del campo de pruebas, *Unity*.

Una vez ya se tenía tanto información de la *API* como del motor gráfico *Unity*, se decidió que la siguiente semana seria para ver el funcionamiento del campo de pruebas que realizó José Alpaont Luján en el 2014 así como estudiarlo para ver cómo fue realizado y el motivo de hacerlo de dicha forma.

La siguiente semana serviría tanto para redactar los primeros puntos de la memoria (introducción, objetivos y motivación), como para comenzar a crear el campo de pruebas en *Unity*.

La siguiente semana se decidió que serviría tanto para continuar con la creación del campo de pruebas como con la redacción de la memoria, añadiendo los puntos explicativos del campo de pruebas, así como los puntos de estado de arte y critica al estado de arte. Por último, también se integraría la *API* al campo de pruebas y se redactarían los resultados obtenidos y los cambios realizados.

Por último, la última semana serviría tanto para pulir la *API* y subirla a la *asset store* , como redactar la conclusión y hacer el análisis de las opiniones de los usuarios de la *API*.

A modo de resumen, y de forma gráfica, se mostrará un diagrama de *Gantt* ,en la figura 5, en el que se recoja dicha planificación que se ha establecido de forma previa, representando el eje Y las actividades realizadas, y en el eje X la duración en semanas, especificando los días, de dichas actividades, y, más adelante, se revisará e irá rellenando con el tiempo que realmente ha necesitado cada una de las tareas, para así comprobar si se siguió la planificación, y en caso de no ser así, analizar el motivo de éste en la conclusión de la memoria.

Etapas de Desarrollo	Semana 1 (9-15)	Semana 2 (16-22)	Semana 3 (23-29)	Semana 4 (30-5)	Semana 5 (6-12)	Semana 6 (13-19)	Semana 7 (20-26)	Semana 8 (27-30)
Lectura del TFM								
Bibliografía								
Herramienta Unity								
Campo pruebas TFM								
Redactar primeros puntos y campo de pruebas								
Campo de pruebas y cambios de la API								
API y redactar								
Subir API y conclusión								

Figura 5: Diagrama de *Gantt* inicial

2. Estado del arte

En esta sección de la memoria se va a hacer una explicación del estado de arte, tanto en la *API* de inteligencia artificial, realizada mediante máquinas de estado finitos o *FSM*, así como en el campo de pruebas, empleando *Unity* como motor gráfico *(palabra definida en el diccionario de términos)*.

La *API* que se va a emplear, tal y como se ha mencionado anteriormente, se trata de un trabajo de fin de máster que realizó el alumno José Alapont Luján y, tal y como se mencionó en el apartado de motivación, actualmente no existe ninguna *API* de la que se pueda hacer uso a la hora de emplear inteligencia artificial en un proyecto o videojuego.

Como bien se explica en [4] la *API*, se realizó mediante máquinas de estados finitos o *FSM*. En esta *API* lo que se hizo fue, crear ciertas clases que permitieran al usuario hacer uso de las FSM del tipo que eligiese proporcionándole además una forma más cómoda de hacer uso de ellas como es el uso de documentos *XML*. Mediante esta forma, el usuario debería rellenar unos documentos *XML* los cuales contienen apartados con los componentes de dicha *FSM* y ya podría hacer uso de dicha *FSM*. Si no quisiera hacerlo de esta forma, también podría, manualmente y llamando a los métodos oportunos ir creando los elementos de la *FSM*, así como realizar las conexiones oportunas. La estructura de la *API*, así como una explicación de cómo usarla se hará en el apartado de la *API*.

Pese a que se decidió realizar dicha *API* de inteligencia artificial mediante el uso de máquina de estados finitos , esta no es la única herramienta en el mercado con la que se pueda dotar a una estructura de cierta inteligencia y a continuación, se estudiarán las diferentes formas más usadas para dotar a un videojuego o proyecto de cierto comportamiento inteligente.

La primera de ellas son las redes neuronales, y tal y como se puede intuir a partir de su nombre, tienen la idea de imitar el funcionamiento de las redes neuronales de los organismos vivos, es decir, un conjunto de neuronas conectadas entre sí, y que trabajan en conjunto para llevar a cabo una tarea, pero cada neurona no tiene una tarea concreta a realizar, si no que se trata de una tarea conjunta y global. Con la experiencia, las neuronas van creando y reforzando ciertas conexiones para “aprender” algo que quede fijo en el tejido. Las redes neuronales parten de esta idea de aprendizaje y se basan en la idea de que dados unos parámetros hay una forma de combinarlos para predecir un cierto resultado. El cómo combinarlos es desconocido y serán las redes neuronales quienes deban encontrar un modelo para averiguar esta combinación y aplicarla al mismo tiempo. Para ello lo que se realiza es dotarlas de un aprendizaje, y tal y como se explica en [5] el más convencional es el modelo de las sinapsis, que consiste en modificar los pesos de cada neurona, siguiendo cierta regla de aprendizaje, construida a partir de la optimización de una función de error o coste, la cual mide la eficacia actual de la operación de la red. Este proceso de aprendizaje es usualmente iterativo, actualizando los pesos de cada neurona una y otra vez hasta que se alcanza el rendimiento que sea deseado.

Ejemplo de videojuegos que emplean este tipo de inteligencia en sus *enemigos* o *NPCs* podría ser, por ejemplo, *Shadow of Mordor*[3]. El sistema que se encarga de dicha inteligencia artificial implantada en sus enemigos se llama "*Nemesis System*", y consiste en que los enemigos recuerdan los encuentros que tienen con el jugador y dependiendo de las heridas que éste les causó o la forma en que fueron derrotados, en posteriores encuentros, modificaran su apariencia o forma de pelear. Un ejemplo de esto se puede observar cuando el jugador mata a unos orcos quemándoles, pues la próxima vez que vaya a ese campamento, se encontrará con orcos desfigurados por quemaduras y estos recordaran que fueron atacados con fuego, evitándolo en la medida de lo posible. También se puede observar este comportamiento cuando el jugador derrota un campamento a distancia con el arco, pues en los siguientes encuentros, el campamento estará formado por orcos con escudos, los cuales se protegerán de los ataques a distancia del jugador, obligándole a éste a tener que enfrentarse cara a cara con los orcos. Este sistema también beneficia a los enemigos, puesto que si estos derrotan al jugador adquirirán nuevas mejoras y habilidades, así como un ascenso, por lo que en futuros encuentros en lugar de ser un soldado orco podrá ser un lugarteniente o capitán orco.

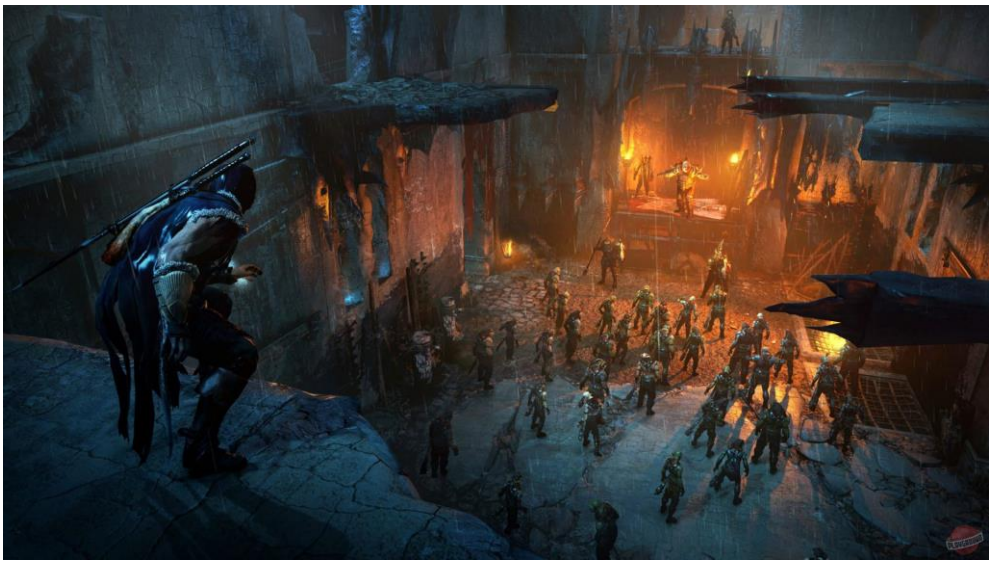


Figura 6: Videojuego *Shadow of Mordors*

Otra forma también muy usada para dotar a una entidad de inteligencia artificial son los árboles de toma de decisión *MinMax*. Este algoritmo se basa en la idea de elegir el mejor movimiento para el computador suponiendo que el jugador contrincante siempre elegirá un movimiento que perjudique a la máquina. Para escoger la opción más beneficiosa lo que hace este algoritmo es realizar un árbol de búsqueda con todos los posibles movimientos, luego recorre todo el árbol de soluciones del juego a partir de un estado dado y elige así el movimiento que más le convenga en ese instante. Este algoritmo se ejecutará cada vez que le toque a la inteligencia artificial del sistema realizar un movimiento. Para profundizar más en el funcionamiento de dicho algoritmo se recomiendo la lectura de [8].Tal y como puede intuirse este tipo de inteligencia artificial se emplea sobre todo en videojuegos por turnos en los que el jugador siempre se enfrenta a un contrincante. Ejemplos de juegos que implementan dicha forma de

3 Sistema *Némesis Shadow of Mordor:* http://sombrasdeguerra.wikia.com/wiki/N%C3%A9mesis

proporcionar inteligencia a la maquina serian, el ajedrez o las damas, juegos por turnos en los que el jugador se debe enfrentar a un adversario. Un ejemplo algo más actual podría ser la saga de videojuegos *Civilization*. En este videojuego de conquista el jugador se enfrenta a otras civilizaciones cuyo objetivo es el mismo que el suyo, ganar la partida. En este videojuego, el jugador puede realizar ciertas acciones en su turno y al acabar le cede el turno a la máquina para que controle las otras civilizaciones a las que se enfrenta y, ésta siempre intentará ponérselo difícil al jugador. Si por ejemplo el jugador empieza muy bien y va en cabeza ganando debido a que tiene muchos aldeanos y va aumentando la cultura de su civilización, lo que hace la *IA* para minimizar dicho crecimiento es enviarle tropas bárbaras para que tenga que dejar de avanzar para defenderse y pierda algún ciudadano o soldado. Si el jugador empieza a expandir sus tierras lo que se hace es que las otras civilizaciones empiezan a construir cerca de donde el jugador lo hace, limitando así la posibilidad de expansión.

Figura 7: Videojuego *Civilization VI*

Por último, como opción que se tuvo en cuenta, y que finalmente se eligió, están las máquinas de estados finitos o *FSM*, las cuales se definen como un conjunto de estados que sirven de intermediario entre una relación de entradas y salidas, siendo estos estados finitos por ser una *FSM*. La idea básica de este modelo, tal y como se menciona en [3], es descomponer un comportamiento objetivo amplio, en estados o trozos, que sean más pequeños y más manejables. Una explicación más exhaustiva de las máquinas de estado y sus diferentes tipos se realizará en el apartado de la *API*.

Muchos videojuegos emplean este tipo de herramientas para dotar de inteligencia artificial a sus videojuegos y un gran ejemplo de ello podría ser el videojuego *Hollow Knight*. En dicho videojuego se maneja a un personaje el cual explora un amplio número de escenarios, en los que se encontrará con un gran número de enemigos, cada uno con un comportamiento diferente, y por tanto a la hora de enfrentarlos, no lo hará de la misma forma. Cada enemigo presenta un comportamiento diferente, pero sí que se observa como por ejemplo ciertos enemigos están en modo *"patrullando"* y hasta que el jugador no está en su campo de visión no entra en modo *"atacar"*. Otros

enemigos tienen comportamientos los cuales no solo dependen de si el jugador es visible o no, si no también de lo que realiza el propio jugador o el resto de los enemigos, como, por ejemplo, ciertos enemigos que solo atacan al jugador si el resto de los enemigos lo hacen o si este tiene poca vida. Debido a que este videojuego surgió como Kickstarter los desarrolladores fueron publicando el progreso del videojuego, por lo que por ejemplo es posible ver como realizaron el comportamiento de algunos enemigos[4].

Figura 8: Videojuego *Hollow Knight*

Tal y como ha podido verse en estos videojuegos los desarrolladores decidieron emplear estas formas de implementar una *IA* para sus proyectos, sin embargo, no emplearon ninguna herramienta como podría ser una *API* o un *SDK* para crearla, teniendo, como consecuencia, que crearla desde 0. Esto como se verá a continuación, es debido a que principalmente no existe una herramienta como podría ser una API que recoja todas o alguna de estas formas de crear dicha IA para el proyecto a realizar.

A parte de estudiar diferentes formas de implantar inteligencia a proyectos o videojuegos, también se buscó *APIs* o paquetes proporcionados por *Unity* que fueran similares a la *API* implementada y aunque, en cuanto a inteligencia artificial no se encontró nada similar a la idea que se tiene en este proyecto, sí que se encontró un trabajo que se está realizando entre *IBM* y *Unity* sobre inteligencia artificial. Este proyecto conjunto entre estas dos empresas, consiste en un *SDK* llamado *IBM Watson Unity SDK* el cual, aunque es cierto que no persigue el mismo objetivo que la *API* presentada en este trabajo de fin de grado, sí que trabaja con inteligencia artificial, centrándose sin embargo en facilitar al usuario servicios tales como reconocimiento de objetos o formas, reconocimiento de voz y escritura de este en texto y por último reconocimiento de idioma. Además de esto lo más parecido a máquinas de estado que presenta Unity es en el propio *Animator Controller*, herramienta la cual sirve para crear las animaciones de los personajes que conformen el videojuego. El comportamiento de esta herramienta es el mismo que el de una máquina de estados. Cada animación sería un estado el cual realiza la acción de mostrar las imágenes que se quiera, las transiciones serían las conexiones que se realizan para pasar de una

[4] Lógica interna de los enemigos de *Hollow Knigth:* https://www.kickstarter.com/projects/11662585/hollow-knight/posts/1193649

animación a otra, y para pasar de una animación a otra se debe de cumplir cierta condición.

Fuera de *Unity* se buscaron también herramientas en forma de *API*[5] que pudieran ser utilizadas para proporcionar cierta *IA* a un proyecto o que pudieran emplearse en videojuego, y pese que al igual que ocurría en *Unity*, no se encontró nada similar a la idea de este *TFG* si se encontraron algunas *APIs* interesantes. Se encontraron, por ejemplo, bastantes *APIs*, las cuales integran al proyecto realizado cierto control por voz, para que estas interpreten lo que el usuario les pida y lleve a cabo la acción correspondiente. Algunas de ellas podrían ser *Jasper*, la cual permite al usuario emplear la voz para pedir cierta información a determinadas aplicaciones, manejar el hogar o actualizar datos. Otra de ellas podría ser *Wit.ai,* la cual es capaz de transformar el lenguaje a datos *JSON* para que así le sea más fácil al desarrollador de la aplicación trabajar. Por último, una muy interesante es *Meya Webhooks AP,* la cual sirve para crear una aplicación la cual avise al usuario de ciertas acciones que se realizan enviándole una notificación.

En cuanto a motores gráficos, en el mercado se encuentran varias opciones para utilizar en el campo de pruebas del proyecto, y a continuación se pasará a ver qué cualidades pueden ofrecer.

La primera opción que se puede encontrar es *GameMaker*, un motor que se centra en videojuegos de 2D, muy fácil de utilizar, ya que incluso te permite crear videojuegos sin tener que escribir ni una línea de código, pero que ofrece la opción de poder programar en un lenguaje propio del motor, dando al desarrollador así más control sobre el videojuego, además de incluir en el propio motor una herramienta interna que permite al desarrollador pintar *sprites* en *píxel art*. Pese a que pueda parecer que al ser tan fácil de utilizar y que no necesite tanta programación para realizar un videojuego, no va a ofrecer un gran resultado o no va a estar a la altura de crear un videojuego de tanta calidad, como otros motores gráficos, todo lo contrario. Además de todas las cualidades comentadas anteriormente otra muy importante es que, al ser un motor que lleva tantos años en el mercado, tiene una grandísima comunidad detrás, lo que implica que encontrar tutoriales o resolver dudas será muy fácil y sencillo, ya sea mediante videotutoriales explicativos o incluso en la misma página de la herramienta o en foros. Ejemplos de videojuegos creados con este motor gráfico muy reconocidos pueden ser *Undertale* o *Hyper Light Drifter,* los cuales fueron realizados por un pequeño equipo de desarrollo, razón por la cual utilizaron esta herramienta, pero dando como resultado videojuegos que pueden competir en el mercado y que son muy disfrutables y queridos por la comunidad.

[5] Lista de *APIs* de inteligencia artificial: https://www.programmableweb.com/category/artificial-intelligence/api

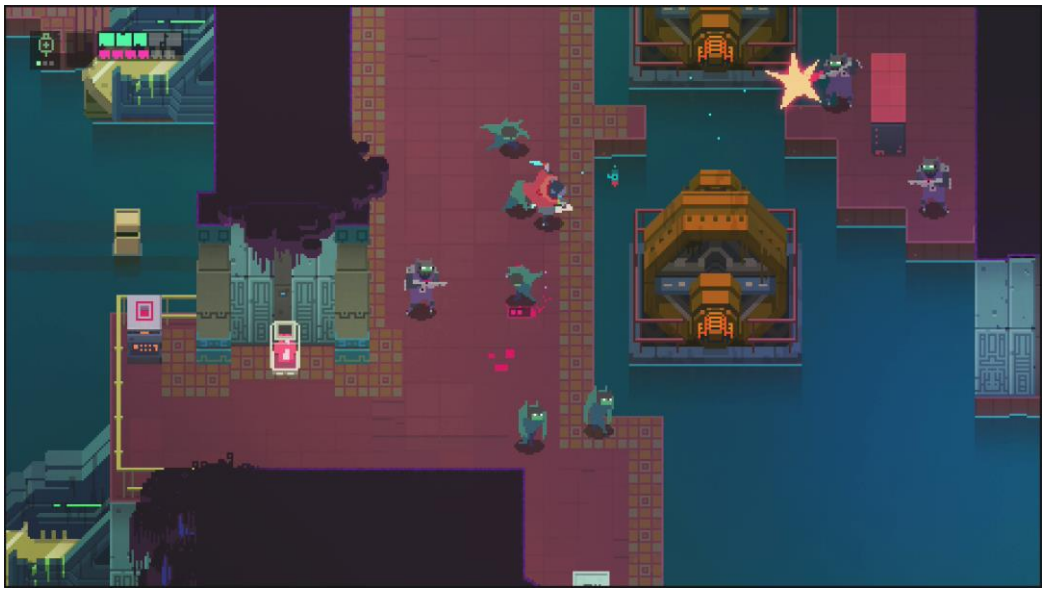


Figura 9: Videojuego *Hyper Light Drifter*

Como siguiente opción, se encuentra *Unreal Engine 4*, motor gráfico, muy potente, que presenta diversas ventajas que podrían ser muy beneficiosas. La primera de ellas es que al ser más profesional que, por ejemplo, *GameMaker*, como lenguaje de programación se emplea *C++*, un lenguaje bastante sólido y que, si en algún momento surgiese alguna duda durante la realización del proyecto, se podría buscar información al respecto sin ningún problema, debido a que es un lenguaje bastante utilizado y tiene una comunidad muy grande. Otra ventaja, y la principal por la que se suele escoger este motor gráfico para la realización de videojuegos, es el apartado de modelaje, tanto de personajes, *NPCs*, enemigos, terreno…, ya que está sumamente logrado, siendo muy cómodo realizar estas tareas y proporcionando un acabado excelente. Por último, en *Unreal Engine 4*, se encuentra la opción de descargar más herramientas mediante una tienda que viene incorporada en el propio motor gráfico, dando así la opción de poder subir la *API* a la tienda y permitiendo así a los usuarios que puedan hacer uso de ella libremente, objetivo que, como se ha visto, se quiere cumplir.

Muchos desarrolladores han utilizado *Unreal Engine* para la creación de sus videojuegos, siendo uno de los más conocidos *Bioshock,* videojuego que destaca tanto por su historia como por los diseños de los personajes, enemigos y *NPCs* que aparecen, los cuales fueron realizados gracias a esta herramienta.

Figura 10: Videojuego *Bioshock*

En cuanto a la herramienta *Unity*, una de sus características principales es que consigue unir de una forma intuitiva, fácil y sin saturar mucho al usuario, tanto una parte artística, en la que se puede tanto modelar personajes o escenarios, crear animaciones y establecer cómo se debe pasar de una a otra, como una parte más de código en la que se puede programar usando *scripts* en *Visual Studio* lo que se necesite y poder ponerle ese trozo de código a cualquier elemento del videojuego. En cuanto a lenguajes que se pueden emplear en este motor gráfico para programar se cuenta con 2 opciones, pudiendo usarse ambas en un mismo proyecto. Estas 2 opciones son, o bien *C#* o bien *UnityScript,* el cual se trata de un lenguaje diseñado específicamente para el uso con *Unity* y modelado tras *JavaScript*.

Por último, como ultima característica, siendo la más importante, *Unity* es una herramienta para desarrollar videojuegos más populares en el mundo de los videojuegos, cuyos creadores proporcionan una documentación muy bien organizada y fácil de consultar, por si se tiene alguna duda o surge algún problema durante la realización del trabajo. Además de la documentación, los propios creadores de esta herramienta proporcionan una serie de videojuegos base creados por ellos mismo, los cuales están explicados mediante videotutoriales como deberían de realizarse paso a paso, dando así una base para saber cómo se realizan ciertas acciones con esta herramienta. Sin embargo, si aparecen dudas muy concretas o no sé sabe cómo realizar ciertas acciones, al ser tan popular es muy fácil encontrar en foros soluciones a problemas similares o incluso encontrar en internet videotutoriales muy bien explicados para encontrarles una solución.

3. Critica al estado de arte

En este apartado se hará una revisión a las diversas opciones que se pueden encontrar en el mercado, las cuales se han comentado en el apartado de estado de arte, y se compararan con las herramientas con las que finalmente se ha decidido trabajar.

Empezando por la *API* de inteligencia artificial, tal y como se ha visto en el apartado anterior, existen varias formas de dotar a un videojuego de cierta inteligencia y autonomía, pero no se eligieron las formas anteriormente vistas por las siguientes razones.

Empezando por las redes neuronales, aunque la idea que hay detrás de ellas, el entrenarlas para que vayan aprendiendo y sean ellas mismas las que al final, con una eficiente autonomía puedan, a partir de ciertas condiciones, tomar decisiones y llevar a cabo ciertas acciones, presentan ciertos inconvenientes por los cuales no se eligieron para la creación de la *API* empleada en este proyecto. El primer inconveniente es que hay que entrenarlas, se les debe dar un aprendizaje para que sepan que elección es la correcta y cual no. Dicho aprendizaje debe cumplir con la condición de que sea el correcto, es decir tiene que estar validado y por tanto la fuente a partir de la cual vayan aprendiendo debe haberse comprobado que es la correcta y que no van a aprender de una fuente que puede llevar a un comportamiento erróneo. Dicha base de la cual deben aprender no se dispone, razón por la cual no se eligió esta forma de dotar a una entidad de cierta inteligencia, ya que si a la hora de comprobar si la *API* creada a partir de redes neuronales se observasen fallos costaría identificar si dichos errores son debidos o bien a un fallo en la propia *API* o bien que el aprendizaje que han obtenido no era el correcto.

Continuando con otros modos de dotar de inteligencia a un proyecto, igualmente válidos, tal y como se vio en el apartado anterior, podrían ser los árboles de toma de decisión *MinMax*. Estos tal y como se mencionó realizan grandes búsquedas para así elegir la opción que más les beneficie y pueda perjudicar al jugador. Estas búsquedas que se realizan en los arboles pueden llegar a ser muy largas puesto que, cuantos más nodos tenga el árbol, más grande será este y por tanto la búsqueda necesitará más tiempo. En juegos por turnos, esto no es un inconveniente, puesto que el jugador entiende que no es su turno y por tanto no tiene que realizar ninguna acción, solo esperar a que le toque otra vez. Sin embargo, en juegos en los que la respuesta debe ser lo más rápida posible, como por ejemplo en un *shooter,* este tipo de inteligencia artificial emplearía demasiado tiempo en calcular una respuesta, dándole tiempo al jugador a eliminar al enemigo sin esfuerzo alguno.

Por último, en cuanto a las máquinas de estado finitos, tal y como se explica en [3] el hecho de usar *FSM* para implementar esta inteligencia es debido a las siguientes razones. La primera es que son muy rápidas y fáciles de implantar mediante código. A diferencia de por ejemplo en arboles de toma de decisión de *MinMax* en los que el algoritmo para saber por qué nodo sé tiene que ir debe ser los más preciso posible, o programar el cómo realizan el aprendizaje las redes neuronales debe ser el más optimo,

en las *FSM* la programación resulta muy cómoda ya que, en última instancia, lo que se realizan son condiciones del tipo *if-else*. Por otro lado, no necesitan un coste computacional muy excesivo, ya que evalúan las condiciones del estado en el que se encuentran, y a raíz de eso deciden si ir al siguiente estado o no, a diferencia de las redes neuronales en las que se tienen en cuenta muchos más factores o los arboles de decisiones *MinMax* en los que el coste es tan grande como el tamaño del árbol, puesto que como se ha mencionado, hay que recorrerlo. Por último, al ser tan fáciles de programar, si se necesita hacer un cambio en el código, éste se puede hacer sin ningún problema, ya que localizar la parte de código que se desea modificar es bastante sencillo. Sin embargo, las *FSM* también presentan algunas desventajas como por ejemplo que los comportamientos que realizan pueden llegar a ser muy monótonos, dándole así al jugador, la sensación de que todo está realmente programado, sin embargo, esto se puede solventar empleando otros tipos de máquinas de estado para así, mejorar el comportamiento de ésta. También hay que tener cuidado a la hora de hacer modificaciones en el código, puesto que al ser tan sencillo el poder modificar el código, hay que intentar organizarlo ya que se puede llegar a tener un trozo bastante grande con estados repetitivos o redundantes.

Por otro lado, dado que no existe ninguna *API* de inteligencia artificial que tenga como idea la de este *TFG* con la que comparar, se centrará en la estructura de las diversas *APIs* encontradas y la del *TFM*. Mientras que las que se han encontrado presentan una estructura definida y organizada a la que se le añaden documentos explicativos de tanto la organización de la misma, como guías de utilización de la herramienta descargada, en la *API* del *TFM* del cual se parte, esta estructura no está del todo realizada, al igual que no existen documentos explicativos que puedan ayudar al usuario una vez quiera hacer uso de la *API*.

En cuanto al motor gráfico *Unity,* tal y como se ha comentado en el apartado de estado del arte, presenta muchas características favorables, sin embargo, no hay que olvidar que en el mercado hay varios motores gráficos los cuales pueden llegar a ofrecer características muy similares o en ocasiones incluso superiores a las que puede llegar a ofrecer *Unity*.

Empezando por la herramienta *GameMaker*, pese a que presenta características muy llamativas como las comentadas anteriormente, no se escogió este motor por unas cuantas razones secundarias como puede ser que el lenguaje de programación que emplea es propio de la herramienta, el cual se llama *Game Maker Language* (*GML*), y se basa en lenguajes de *scripts,* y pese a su gran simplicidad, habría que aprenderlo, quitando tiempo para realizar otras tareas del trabajo. Otra razón secundaria por la cual no se eligió este motor gráfico es que pese a que, poco a poco se va integrando la opción de realizar un videojuego en 3D, este motor gráfico, tal y como se ha comentado, se centra en videojuego de 2D y el intentar realizar cosas en 3D no es tan fácil ni intuitivo, y al intentar buscar información sobre dudas que puedan surgir no se van a encontrar tantísimas soluciones como ocurre en *Unity*. Pese a todo esto la razón fundamental por la que no se escogió este motor gráfico es que *GameMaker* no incluye en su herramienta de desarrollo una opción de *asset store* para bajarse herramientas para el proyecto que se esté realizando, no dejando así la posibilidad de subir la *API* final, como si fuera una *APP* y dar así al usuario la opción de que la pueda utilizar en sus videojuegos de una forma tan sencilla como podría hacerlo con *Unity*.

Continuando con la herramienta *Unreal Engine 4*, este motor gráfico también presenta ciertas cualidades negativas que hicieron que al final se optase por no elegir. La primera de ellas es que *Unreal Engine 4* sólo da la opción de realizar videojuegos en 3D, quitando la opción así de poder crear videojuegos en 2D. La segunda de ellas, y la más determinante, es que, pese a que *Unreal Engine 4* es bastante conocido y popular en el mundo de los videojuegos, no es tan fácil encontrar gente que sepa dominar este motor gráfico en cuanto a la programación se refiere, ya que lo que más se encuentra es gente que domina el apartado artístico, y pese a que tienen una grandísima documentación, con la cual puedes de forma autodidacta ir aprendiendo poco a poco, si surgen problemas y se les tiene que sacar una solución lo más rápido posible, es preferible que haya una comunidad detrás que pueda solventar estas dudas.

A modo de resumen se incluye una tabla comparativa, una para que sean más visible los motivos de elección de máquinas de estados finitos y la necesidad de mejora de la API y otra para que sea más visible el motivo de la elección de Unity como motor gráfico. En ella se ha marcado con un *tick* azul cuando dicha opción presenta la característica indicada en el eje Y, con una cruz roja si no presenta dicha característica, con una ralla negra si esa característica no tiene que ver con la opción seleccionada y por último una raya amarilla cuando dicha característica se cumple parcialmente.

	Redes Neuronales	MinMax	Máquinas de estados	Api de la se parte
Fáciles de programar	✗	✗	✓	—
Fáciles de depurar	✗	✓	✓	—
Facilidad para realizar cambios	✓	✓	✓	—
Respuesta rápida	✓	✗	✓	—
Se pueden validar fácilmente	✗	✓	✓	—
Inteligencia artificial que no parezca programada	✓	✗	✗	—
Estructura organizada	—	—	—	—
Documentos explicativos de la API	—	—	—	✗
Guías de uso para el usuario	—	—	—	✗

Figura 11: Tabla comparativa de modos de implantar IA

Es por todas estas razones por las que, al final se optó por utilizar máquinas de estados finitos o *FSM*, debido a las facilidades que estas proporcionan a la hora de programarlas, las facilidades que presentan para encontrar errores, así como la forma de poder usar diversos tipos de maquinas de estados para solventar problemas como la sensación de tener una IA programada y no con un comportamiento menos programado.

	GameMaker	Unreal	Unity
Lenguaje de programación conocido			
Facilidad para encontrar soluciones			
Posibilidad de trabajar en 2D y 3D			
Asset Store			
Facilidad de Uso			

Figura 12: Tabla comparativa de modos de implantar *IA*

Es por todas estas razones por las que al final se decidió usar la herramienta de Unity como motor gráfico para poder realizar el campo de pruebas y así validar el correcto comportamiento de la *API* de máquinas de estados de la cual se parte.

4. API implementada

Tal y como se ha comentado en apartados anteriores la *API* de la que se va a hacer uso es un trabajo de fin de máster que se realizó en 2014 por el alumno José Alapont Luján, la cual está basada en máquinas de estados finitos.
En este bloque de la memoria se hará, a lo largo de los diferentes apartados que la componen, una breve explicación de que son las máquinas de estados finitos, los diferentes tipos que se han implementado y como está estructurada la *API*, para así poder entenderla un poco más, y así saber cómo poder hacer un uso correcto de ella.
Por último, se comentarán los cambios que se han tenido que realizar o los elementos que se han añadido con el fin de que sea más útil, práctico y más sencillo para el usuario hacer un uso correcto de la *API*.

4.1 Introducción a las máquinas de estados finitos

En este apartado se pasará a hacer una explicación de que son las máquinas de estado, así como hacer un repaso de sus características, comentando sus puntos fuertes y sus puntos débiles.
Para empezar, tal y como se explica en el capítulo 4 de [3], se puede definir una máquina de estados como un modelo de comportamiento en un sistema con entradas y salidas, en donde las salidas dependen no sólo de las señales de entradas actuales sino también de las anteriores. Es por ello por lo que una máquina de estados estará formada por los estados que la compongan, las transacciones que conectan dichos estados unos con otros y por último se tendrán las acciones que se lleven a cabo.
Para empezar, se puede definir un estado, tal y como se hace en [1], como una configuración única de información, y si en concreto se trata de una máquina de estados finitos, un autómata de pilas o una máquina de Turing, se definirá un estado como un conjunto particular de instrucciones, las cuales serán ejecutadas en respuesta a la entrada de la máquina. Por otro lado, las transiciones serán lo que permita pasar de un estado de la maquina a otro, y normalmente están ligadas a condiciones o eventos que deben de cumplirse para pasar a otro estado. Para finalizar, cuando se habla de las acciones de una máquina de estados se tiene que tener en cuenta que dichas acciones pueden estar asociadas a una transición, y por ello solo se ejecutaran al pasar de un estado a otro, o también pueden estar asociadas a un estado, pudiendo realizar acciones al entrar o salir del estado, con lo cual solo se ejecutarían dichas acciones una vez, o bien se puede ejecutar dicha acción mientras se esté en el estado de la que forme parte y no se cumpla ningún evento que haga abandonar el estado actual.
Por último, las máquinas de estado finitos presentan un estado inicial o punto de inicio, el cual indica cual debe ser el estado del que se tiene que partir al comenzar con la ejecución de la máquina de estados.

En cuanto a las ventajas e inconvenientes que presentan, tal y como se mencionan en los libros [2] y [3] puede destacarse tanto la facilidad de implementación como de utilización ya que, en última instancia, lo que se está haciendo es comprobar en que estado se encuentra la máquina, y si se cumple alguna de las condiciones asociadas a las transiciones para pasar a otro estado. Esto también presenta una doble ventaja puesto que, al estar realizando comprobaciones, lo cual en código se traduce en el uso de *switch-case*, se obtendrá un código muy organizado en el que realizar cambios se traducirá en añadir un *case* mas o eliminar el que se desee. Sin embargo, el hecho de que sean tan fáciles de realizar dichas modificaciones puede presentar un inconveniente, ya que se tendrá que tener un especial cuidado en no añadir más apartados *case* de los necesarios para no generar así, ni estados redundantes, ni estados inalcanzables en la máquina de estados. También otra ventaja que presentan es que, debido a que se están realizando simplemente comprobaciones, el coste que esto presenta es bastante reducido y la velocidad de respuesta es muy elevada.
Por otro lado, tal y como se ha comentado anteriormente, una desventaja que presentan son los estados, ya que estos al ser finitos son limitados, y por tanto al tener un número limitado de estados esto puede provocar como resultado un comportamiento previsible, pese a que este hecho se pueda mitigar haciendo uso de otros tipos de máquinas de estados finitos. Por otro lado, otro inconveniente que presentan son las respuestas que pueda proporcionar una *FSM,* puesto que estas se basan en reglas simples, y por tanto bajo situaciones muy complejas no serán muy útiles, ya que se necesitarían demasiados estados, pudiendo dar como resultado una máquina de estados inestable o incontrolable.

A modo de resumen se incluye una tabla comparativa entre sus ventajas e inconvenientes, en la que con un *tick* azul se han marcado las ventajas que presentan y con una X en rojo se han marcado los inconvenientes que presentan.

	Ventajas	Inconvenientes
Facilidad de implantar	✓	
Útil bajo situaciones complejas		✗
Código sencillo	✓	
Comportamiento previsible		✗
Facilidad de uso	✓	
Código estructurado	✓	

Figura 13: Tabla de ventajas de las *FSM*

4.2 Tipos de máquinas de estados finitos de la API

En este apartado de la memoria se comentarán los diferentes tipos de máquinas de estados finitos implementadas en la *API*, y se realizará una explicación de cada una de ellas, así como de las condiciones que se deben de cumplir para poder hacer un uso de ellas lo más correcto posible.

Antes de empezar a comentar los tipos de *FSM* existentes en la *API*, hay que comentar que, entre las *FSM* existe una división entre las que son deterministas y las que son indeterministas. La diferencia principal entre ambas es que mientras que en las deterministas desde un estado del cual se parte, solo se puede ir a otro estado, en las indeterministas de un estado se puede ir a múltiples estados. Además de esto en las indeterministas las transiciones no tienen por qué tener un evento o condición que deba cumplirse para pasar de un estado al siguiente, pudiendo ir, por tanto, de un estado a otro sin necesidad de que se cumpla ningún requisito.

En la *API* se encuentran las siguientes máquinas de estados finitos:

- **Determinista** (*FA_Classic*): En este tipo de *FSM* además de cumplirse las condiciones propias de las *FSM* deterministas vistas anteriormente, también se cumple

que para ciertas entradas siempre se van a obtener las mismas salidas. Este tipo de *FSM* son las más simples y las que, a la hora de hacer uso de ellas, se tiene que tener en cuenta que debido a que de un estado solo se puede ir a otro, visualmente en el videojuego se podrá observar como el comportamiento del *NPC* o *boss* será muy intuitivo y predecible. Esto se puede solventar, y se suele realizar esta práctica, haciendo que la *FSM* Determinista sea la que rija el comportamiento general mientras que cada estado tenga asociada una *FSM* que se encargue de llevar a cabo acciones para que así el comportamiento tenga un acabado más detallado. Es por ello por lo que estas máquinas se deberán emplear cuando se quiera una *FSM* que controle, o bien comportamientos generales o muy sencillos. Sin embargo, hay que tener en cuenta ciertas condiciones cuando el desarrollador cree los estados de su *FSM*. El primero de ellos es que no se deben crear estados a los que no se pueda llegar, es decir que no estén conectados con otros estados. Además, en esta *FSM* sólo se acepta un evento por ciclo, por lo tanto, solo se podrá llevar a cabo una acción por ciclo.

- **Probabilística** (*FA_Classic*): Este tipo de *FSM* indeterminista se diferencia de la determinista en que, aparte de las diferencias vistas anteriormente, las transiciones tienen asociada una probabilidad, es decir, que, aunque se cumpla la condición para pasar de un estado a otro, si tiene asociada una probabilidad, se puede dar la condición de que no se llegue a ir a ese estado. Pese a que esto pueda parecer absurdo, es bastante útil, ya que esto proporciona un comportamiento más impredecible, al *NPC* o *boss* que haga uso de ella, dando como resultado un comportamiento no tan monótono y rutinario, proporcionándole así al jugador la sensación de que el comportamiento no está programado y que es el azar el que dicta las acciones que realizará el *NPC* o *boss* que haga uso de esta *FSM*. Este tipo de *FSM* comparte el mismo código que el de la *FSM* determinista, ya que en el caso de la *FSM* determinista la probabilidad es del 100%, puesto que siempre que se cumpla la condición se saltará al estado, mientras que en la probabilística se le asocia el porcentaje que se desee. Es por este motivo que las condiciones que se tienen que respetar son las mismas que en la *FSM* determinista.

- **Inercial** (*FA_Inertial*): Este tipo de *FSM* indeterminista tiene asocia una latencia en cada estado, lo cual hace que se haga una espera para comprobar si se cumple el evento o condición asociada a la transición, que conecta dicho estado con el siguiente. Este tipo de *FSM* presenta la misma estructura que la de la *FSM* Determinista, pero con una espera añadida a los estados para así poder solventar un problema muy recurrente en las *FSM* el cual es la oscilación de estados. Este problema es bastante común, y ocurre cuando nada más pasar a un estado se cumple el evento o condición asociada a la transición para pasar al siguiente estado, por lo que se realiza dicha transición, y en este nuevo estado también ocurre lo mismo, pasando al siguiente estado. Esto tiene como resultado que en pocos ciclos se hayan realizado muchas transiciones entre estados y se han llevado a cabo diversas acciones, lo cual provoca como resultado un comportamiento nervioso o muy rápido en el *NPC* o *boss*. Este problema se puede solventar realizando esta espera, para que asi cuando se haya finalizado este tiempo, se compruebe de nuevo si se cumple la condición de la transición, y entonces se realice el salto al siguiente estado. Al realizar esta espera lo que sucederá es que, el tiempo que se está en el estado aumente, por lo que, si se tiene asociada una acción por permanecer en dicho estado, esta acción se realizará un número de veces mayor. Además de tener en cuenta las condiciones de la *FSM* determinista, en la *FSM* inercial se podrá o no añadir latencia a los estados, y, por tanto, no será obligatorio que todos los estados la

tengan. Por último, la latencia se asociará a los estados, y no a las transiciones, y dichas latencias estarán en milisegundos.

- **Basada en pilas** (*FA_Stack*): Este tipo de *FSM* son bastante útiles ya que permiten crear un efecto de *"memoria"* en el *NPC* o *boss* que haga uso de esta *FSM*. Estas *FSM* son de las más útiles, pues proporcionan al personaje que haga uso de ellas, la posibilidad de poder, mientras está en un estado realizando las acciones que toquen, detener esa acción para ir a otro estado, que sea más importante, a realizar una nueva acción, y, tras finalizar esa nueva acción, poder volver al estado en el que se detuvo la ejecución continuando con su ejecución. Para ello deberá, de cierta forma recordar a donde tiene que volver y es por ello por lo que a los estados se les tendrá que añadir cierta prioridad. Este aspecto se añadió para establecer un orden a la hora de apilar los diferentes estados. De esta forma lo que se hace es activar los estados más prioritarios y así llevar cierto orden de apilado, y por otro lado se consigue que los estados que tengan la misma prioridad no se interrumpan entre ellos, a no ser que tengan una transición que los conecte.

- **Estados concurrentes** (*FA_Concurrent_States*): Este tipo de *FSM* otorga la posibilidad de ejecutar varios estados el mismo tiempo. El número de estados que se activaran será indicado por el desarrollador y esta *FSM* seguirá el modelo de redes de Petri. Una red de Petri, tal y como se explica en [6], es una representación matemática o grafica de un sistema a eventos discretos en el cual puede describir la topología de un sistema o bien distribuido, paralelo o concurrente. Por otro lado, como se explica en [7], el funcionamiento de este tipo de *FSM* se basa en *"créditos"* o *"monedas"*, asociados a cada estado, los cuales se generan en ciertos estados y son consumidos en otros. Si un estado tiene créditos de ejecución podrá estar activo, y cuando se salte de un estado a otro el primero le otorgará un crédito al segundo y si el primero se queda sin créditos entonces ese estado quedará inactivo. Estos créditos serán de utilidad para asegurar ese grado de paralelismo que se busca en este tipo de *FSM*. Es por ello por lo que serán de especial utilidad cuando se quieran ejecutar varios estados a la vez , los cuales sean de aspectos diferentes, como podría ser por ejemplo la IA, la música y las animaciones de un *boss* o un *NPC*, estados que podrían darse a la vez.

Estos son los tipos de máquinas de estados finitos que están definidas, y de las que se puede hacer uso, en la *API*. Para una mayor explicación y clarificación con ejemplos de comportamientos que se podrían implantar con estas *FSM*, así como ejemplos reales en máquinas de estados finitos creadas, se recomienda revisar el Anexo *"Guía de elección de FSM"*. Este mismo documento, que aparecerá en el Anexo, será el que se añada como *pdf* con el mismo nombre al *asset* que se descargue el usuario para que así, pueda disponer de toda la información necesaria, pueda visualizar ejemplos y pueda tener una idea más clara de los tipos de *FSM* existentes en la *API* , teniendo así una mayor facilidad para elegir el que más le convenga dependiendo del comportamiento que quiera implantar.

4.3 Estructura de la *API*

En este apartado de la memoria se explicará en detalle tanto como está estructura la *API,* como la funcionalidad de cada uno de sus componentes, para facilitar así la comprensión de ésta y así, más tarde ,en el apartado de utilización de la *API*, el lector tenga mas claro el motivo de los pasos que se siguen.

Inicialmente la *API* estaba parcialmente organizada, por lo que la primera acción que se realizó sobre ella fue organizar todos los componentes de la misma, y así el usuario, al descargársela obtuviera una herramienta organizada en la que si deseara consultar o modificar alguna cosa lo tuviera más fácil. Es por ello por lo que la primera acción que se realizó fue realizar un estudio sobre la *API* para así hacer una clasificación coherente de los componentes y poder realizar una correcta división de éstos.

Al querer hacer uso de la *API*, tras la descarga en el *asset store*, al usuario le aparecerá en el apartado de *Asset* una carpeta llamada *GAI (game artificial intelligence).* En dicha carpeta se encontrarán diversas subcarpetas, necesarias para el correcto funcionamiento de la *API*, una carpeta que contendrá un ejemplo de un videojuego en el cual se ha integrado la *API*, para que el usuario tenga un ejemplo base y 2 documentos *pdf* en los cuales se explica al usuario como hacer uso de la herramienta y como elegir la máquina de estados finito que más le convenga, según sus necesidades o los objetivos que desee cumplir en su proyecto. Dichos documentos se encuentran en el apartado de Anexo de la memoria como, "*Guía de utilización de la API*" y "*Guía de elección de FSM*". En la figura 14 se puede observar la estructura final con la que quedó el *asset.*

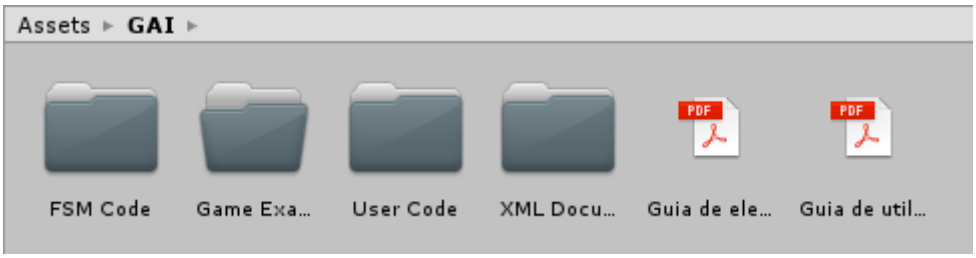


Figura 14: Estructura del *Asset GAI*

En la primera carpeta, llamada ***FSM Code***, se encontrará todo el código encargado del funcionamiento óptimo, tanto de la máquina de estados finitos que se elija para hacer uso, como del *Parser* encargado de trabajar con las plantillas *XML*, en caso de que se haga uso de ellas. Dentro de las clases encargadas del correcto funcionamiento de la *API* se tienen varias clases, cada una de ellas encargada de realizar una parte importante. La primera de ellas se llama ***FSM_Manager*** y esta clase se encarga principalmente de asignarle a la entidad que lo pida, la máquina de estados finito que desee. Cuando una entidad necesite hacer uso de una *FSM* lo que se hará es invocar el método *CreateMachine()* e internamente el *FSM_Manager* lo que hará será buscar en su diccionario la máquina de estados que haya sido solicitada e instanciará un objeto *FSM_Machine* que corresponde al autómata almacenado. La siguiente clase por

analizar es la clase ***FSM_Machine*** la cual, a diferencia de la clase *FSM_Manager*, cuya función se centra más en el almacenaje de *FSM* y asignaciones, se centra en recorrer de forma óptima la *FSM* elegida según su tipo. Además, también se tendrá en esta carpeta el código de cada una de las máquinas de estados finitos, ***FA_Clasic***, para la *FSM* determinista y la *FSM* probabilística, ***FA_Concurrent***, para la *FSM* basada en estados concurrentes, ***FA_Inertial***, para la *FSM* inercial y ***FA_Stack***, para la *FSM* basada en pilas. A parte de estas clases también se encontrarán la clase ***State*** y la clase ***Transition***, las cuales sirven para crear los estados y las transiciones respectivamente, según el tipo de *FSM* que se haya elegido, ya que como se ha visto en el punto de tipos de máquinas de estados finitos, al haber diferencias entre ellas, los estados y las transiciones también cambian, necesitando más o menos variables. Por último, y para finalizar con el contenido de esta carpeta, se encuentra la clase ***FSM_Parser*** cuyo uso, tal y como se ha comentado anteriormente, es totalmente opcional ,y su función se basa en cargar automáticamente las variables de la *FSM* desde un documento de texto *XML*. El contenido de dicha carpeta se puede visualizar en la figura 15.

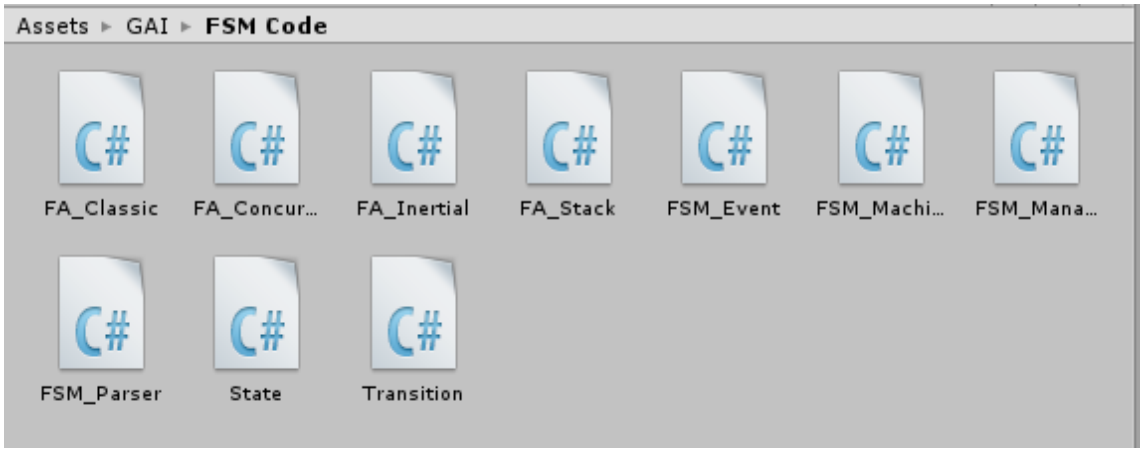


Figura 15: Contenido de la carpeta *FSM Code*

Tal y como se puede intuir todas las clases que forman esta carpeta no deben ser modificadas por el usuario, por lo que en el documento explicativo del funcionamiento de la *API*, se mencionan dichas clases y sus funciones, pero se le advierte al usuario de que no debe tocar o modificar ninguna línea de código de dichas clases, puesto que esto provocará fallos en el funcionamiento de la *API* que no podrá localizar o solventar.

En la siguiente carpeta, llamada Game Example, se encontrará el campo de pruebas que se ha empleado, junto con las plantillas *XML* de las máquinas de estados finitos rellenadas de las cuales se hizo uso, así como las clases en las que se realizó la integración de la *API*, para que el usuario vea como se haría la integración en un ejemplo real, como es uno de los videojuegos modelo que proporciona *Unity* para realizar, y que el propio usuario tenga la opción de comparar el código resultante tras la integración de la *API* con el código original y así ver las diferencias entre emplearla o no. También le servirá al usuario para, una vez leídas las dos guías que se proporcionan, poder ver en un ejemplo que comportamientos se obtendrían en un videojuego al cambiar entre las *FSM* que están implementadas. El contenido de dicha carpeta se puede visualizar en la figura 16.

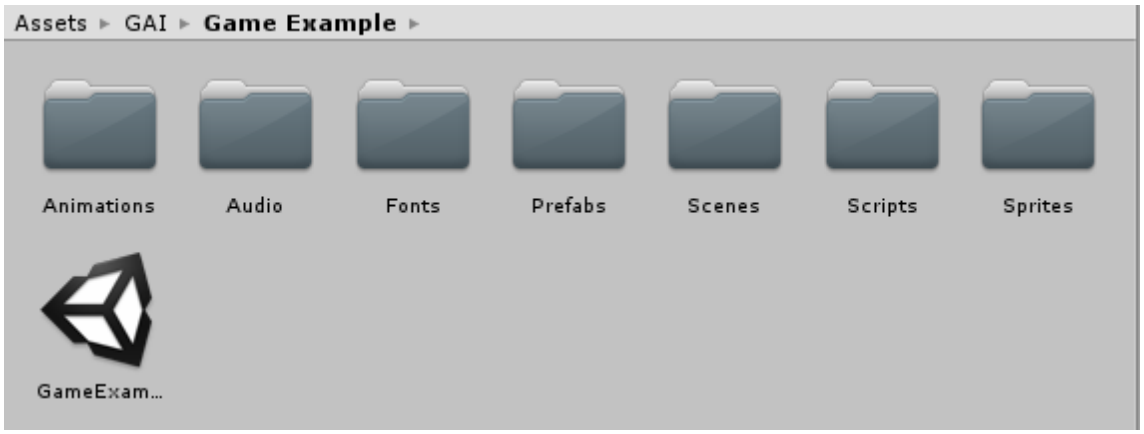


Figura 16: Contenido de la carpeta *Game Example*

La siguiente carpeta que se tiene se llama ***User Code***, y en esta carpeta será, junto con la carpeta que contiene los documentos *XML*, donde el usuario tendrá que hacer modificaciones. En esta carpeta se encontrarán 3 clases, la clase ***Tags***, la cual sirve tanto para definir las variables que representan los estados, acciones, transiciones, y eventos, como darles valor, y las clases ***ModelUsingTools*** y ***ModelWithoutUsingTools***, en las cuales el usuario tendrá un modelo con todas las variables, llamadas a métodos y valores a proporcionar, que tendrá que emplear para hacer uso de la *API*, así como comentarios explicando tanto que se está haciendo en cada línea de código como el orden a seguir, los posibles métodos que puede emplear dependiendo de la FSM empleada, y que valores dicho método debe recibir. En dichos modelos se muestra cómo usar la *API* tanto si se decide emplear la clase *FSM_Parser* como los documentos XML(***ModelUsingTools***), como si se decide no hacerlo(***ModelWithoutUsingTools)***. La clase ***Tags***, al igual que el uso de los documentos *XML* y el *FSM_Parser*, es opcional pese a que su uso es muy recomendado, ya que la función de dichas herramientas se basa sobre todo en la organización, evitando así que se tenga una clase muy extensa en la que gran parte del código son simples definiciones, asignaciones de variables y métodos para retornar valores. Delegando esta acción en otra clase se obtendrá un código mucho más limpio, al igual que haciendo uso de los documentos *XML*, ya que así se evitará realizar enormes invocaciones a métodos para crear los estados, las transiciones y los eventos o condiciones asociados a ellas, así como la conexión entre los estados. Simplemente rellenando dicho documento y haciendo uso del *FSM_Parser* se conseguiría el mismo resultado de una forma más limpia. El contenido de dicha carpeta se puede visualizar en la figura 17.

Figura 17: Contenido de la carpeta *User Code*

Por último, en la carpeta llamada ***XML Documents***, se encontrarán todos los documentos *XML*, los cuales sirven como plantilla para crear cada una de las máquinas de estados finitos, habiendo, por tanto, una plantilla por cada tipo de *FSM*. Cada plantilla contendrá los apartados correspondientes para crear los elementos necesarios que compongan cada una de las máquinas de estados finitos, así como una descripción de cada uno de los apartados, para que así el usuario sepa no solo la utilidad de dicho apartado si no también que valores están permitidos. Además de esto se proporciona al usuario una carpeta llamada *Hierarchical* para colocar allí las plantillas que cree para realizar máquinas anidadas. El uso de estos documentos es totalmente opcional, no siendo por tanto obligatorio para el correcto de la *API*, sin embargo, son muy recomendados para obtener tanto un código final más estructurado y claro, como para reducir así la aparición de errores y tener una facilidad mayor a la hora de depurar fallos. En estas plantillas el usuario encontrará apartados para darle nombre a los estados, transiciones, acciones y eventos, así como establecer desde que estado se puede ir a otro y las acciones que se llevaran a cabo. Tal y como se ha mencionado, el uso de estas plantillas es totalmente opcional y por tanto si el usuario decide no hacer uso de ellas lo que tendrá que hacer es realizar llamadas a los métodos que se encarguen de crear todos los elementos de la *FSM*, así como realizar las conexiones entre los estados.El contenido de dicha carpeta se puede visualizar en la figura 18.

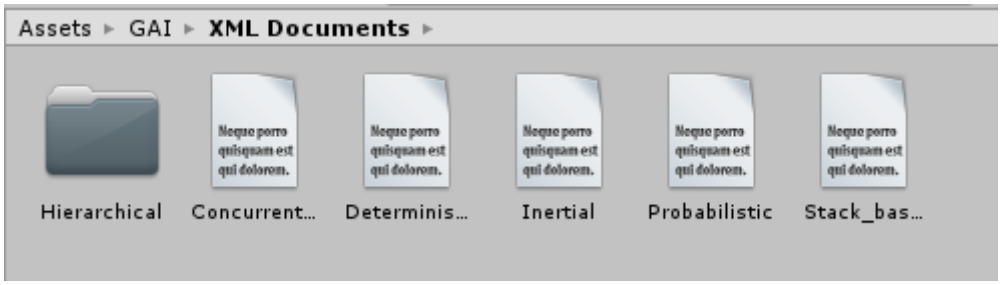


Figura 18: Contenido de la carpeta *XML Documents*

Por último, junto con estas carpetas, y tal y como se ha comentado anteriormente, se han añadido los documentos que sirven de guía para la elección de la máquina de estados finitos, así como una guía de utilización de la *API*, ambos en formato *pdf*.

4.4 Utilización de la *API*

Tras haber visto como está estructurada la *API*, en este apartado de la memoria se hará una explicación de como hacer un uso correcto de ésta, exponiendo el orden a seguir, así como el motivo de hacerlo de esa forma y en dicho orden.

Antes de empezar se debe recordar que tal y como se ha explicado en el apartado anterior, hay ciertas partes de la *API*, tales como los documentos *XML*, la clase *FSM_Parser* y la clase *Tags*, cuyo uso, aunque muy recomendado, es totalmente opcional, razón por la cual la explicación de cómo hacer uso de la *API* se dividirá en dos partes. En la primera de ellas se explicará como usarla utilizando todas las herramientas opcionales que se incluyen, y en la segunda sin hacer uso de dichas

herramientas, y por tanto siendo el desarrollador quien cree todos los estados, transiciones y la lógica de conexión entre estados.

Toda la explicación que se hará a continuación de forma más extendida así como con imágenes extraídas de las clases que se proporcionan como modelos, junto con una parte de la estructura de la *API*, para que así el usuario pueda ubicarse mejor entre las carpetas, se podrá visualizar en un documento en formato *pdf* que le aparecerá al usuario llamado *"Guía de utilización de la API"*, el cual aparecerá en el anexo con el mismo nombre.

Además de esto, para que el usuario no tenga que hacer el código desde 0, y pueda usar una base para su proyecto, se le proporcionará en la carpeta *User Code* una clase llamada ***ModelUsingTools*** en la cual estarán realizadas, paso a paso, todas las llamadas a métodos y todas las acciones necesarias para el correcto funcionamiento de la *API* haciendo uso de las herramientas propias de la *API* y otra clase llamada ***ModelWithoutUsingTools*** en la cual se hará los mismos pasos pero sin hacer uso de dichas herramientas.

4.4.1 Haciendo uso de las herramientas

Dado que, si se utilizan las herramientas que proporciona la *API*, se tendrá más comodidad a la hora de hacer uso de la máquina de estados finitos elegida, la mayoría de las acciones que se realizaran al hacer uso de estas herramientas serán rellenar los apartados señalizados con el nombre que se desee, en el caso de los documentos *XML*, y darles un valor en el caso de la clase *Tags*.

El primer paso que realizar, tras haber hecho la elección de la *FSM* de la que se quiera hacer uso, es ir a la carpeta ***XML Documents*** y, seleccionando la plantilla de la *XML* elegida, rellenar los apartados necesarios para crear tanto los estados, transiciones y eventos ligados a dichas transiciones. Pese a que cada plantilla de cada una de las *FSM* es diferente, mas o menos todos comparten los mismos apartados, puesto que todas parten de una *FSM* Determinista a la cual se le añaden los apartados que sean necesarios, razón por la cual los siguientes apartados los tendrán todas las *FSM*. Cada tipo de plantilla además de venir con las variables necesarias para cada tipo de *FSM*, viene con el apartado que indica el tipo de *FSM* ya rellenado, siendo 4 los tipos de *FSM*; *CLASSIC* en el caso tanto de la *FSM* Determinista (pero con el apartado *Probabilistic* marcado como NO) como de la *FSM* Probabilistica (pero con el apartado *Probabilistic* marcado como YES), *INERTIAL* en el caso de la *FSM* Inercial, *STACK_BASED* en el caso de la *FSM* basada en pilas y por último, *CONCURRENT_STATES* para el caso de la FSM basada en estados concurrentes.

Lo primero que deberá realizar el usuario es rellenar los apartados que corresponden con los elementos básicos que componen una *FSM*. Para empezar, el usuario deberá darle un nombre a la *FSM* de la cual vaya a hacer uso así como al *callback* al cual el

FSM_Manager llamará internamente cada ciclo para ver si ha ocurrido algún evento y por tanto se deba ir a un estado y realizar la acción que corresponda.

El nombre que se dé a dicho *callback* será un método que se deberá implementar en la clase que haga uso de la *API*, por lo que los nombres deberán coincidir. Tanto si al acceder a la *FSM* como cuando se realice el *callback*, los nombres no coinciden con los que se puso en el documento *XML,* se obtendrá un fallo de compilación, en concreto un *Null pointer Exception*, debido a que no se encontrará dicho método, por tanto es importante asegurarse de que dichos nombres coincidan en ambos documentos.

Para empezar, el usuario empezará rellenando las etiquetas que representen al estado que quiera crear. Para ello a dicho estado le tendrá que dar un nombre y unas acciones a realizar, pudiendo realizar dichas acciones al entrar al estado, al salir o mientras se esté en dicho estado. También se le podrá pasar al estado una ruta hacia otro documento *XML* que contenga otra *FSM* para que dicho estado sea realmente una sobaquina.Al rellenar estos campos el usuario tendrá 1 estado creado y para crear mas simplemente tendrá que duplicar esta estructura y rellenarla con los valores del nuevo estado.

Después de esto se procederá a rellenar las etiquetas de las transiciones y los eventos asociados a dichas transiciones. Para ello se deberá darle a dicha transición un nombre, así como el nombre del estado de origen del cual se parte, el nombre del estado destino al cual se dirige, una acción que se puede o no asociar a dicha transición(dándole el valor NULL no se tendría ninguna acción) y por último el evento o eventos que harán que esta transición se realice y se pase al estado destino. Estos eventos, como se verá más adelante, están ligados a condiciones a cumplir, y cuando esto sucede, a partir de dichos eventos la *FSM* obtendrá la acción que se deba realizar. Al igual que en el caso de los estados, si se quieren añadir más transiciones o eventos lo único que se tiene que hacer es añadir más bloques de este tipo.

Todas las demás *FSM* contendrán estos mismos apartados a rellenar en sus plantillas *XML*, pero tendrán algunos apartados más a rellenar, los cuales se verán a continuación. Empezando por la *FSM* Probabilística, esta presentará una etiqueta más para indicar la probabilidad que se desea que tenga cada transición, y deberá rellenarse con un valor entre 0, siendo la probabilidad más baja y por tanto dicho estado nunca será alcanzable, y 100, siendo la probabilidad más alta y por tanto dicho estado, siempre que se cumpla el evento asociado a dicha transición será alcanzable. La siguiente plantilla es la de la *FSM* Inercial, y en ella se añade una etiqueta a los estados, para que asi se puede indicar la latencia en milisegundos que quiera que tenga el estado. Seguidamente se tiene la *FSM* basada en pilas, en la que se añade una etiqueta a los estados, para así indicar la prioridad de estos, teniendo en cuenta que la prioridad más alta será la prioridad con el número más alto y 0 será la prioridad más baja. Además, en estas FSM habrá que indicar que estados pueden ser apilados y para ello se marcaran las transiciones que vayan de un estado menos prioritario a otro más prioritario como STACKABLE en lugar de BASIC, dejando como *BASIC* las transiciones entre estados igual de prioritarios o de estados más prioritarios y menos prioritarios.

Por último, se tiene la *FSM* de estados concurrentes en la que se añade una etiqueta a los estados, para indicar los créditos iniciales con los que empezará dicho estado.

Para ver los pasos mas detallados, que campos hay que rellenar, así como un ejemplo de una plantilla rellenada se recomienda la lectura del anexo "Guía de utilización de la *API*".

Una vez se hayan rellenado los apartados del documento *XML* de la *FSM* elegida, lo que se hará será rellenar la clase **Tags** ubicada en la carpeta **User Code.** En dicha clase lo que se hará será inicializar las variables que se han creado en el paso anterior, las cuales representan tanto los estados y transiciones como los eventos. Dicho valor entero no puede estar repetido si la variable que ya tenia dicho valor pertenece al mismo campo que la variable declarada, es decir, que si se están definiendo las variables que representarán los estados no se les dé el mismo valor a dos estados.

Por último, en esta clase se tendrá que completar un método ***StringToTag(string Word)*** el cual implementa un *switch-case* en el que se comprueba la primera letra que compone el *String Word* ,que recibe como parámetro, y busca en el *switch* el case que corresponda con dicha letra. Una vez se ha encontrado la letra, se procede a buscar a ver si coincide con alguna de las palabras definidas y si es así, devuelve el valor de esta variable. Para ello lo que tendrá que hacer el usuario será añadir la variable que haya inicializado, para que se pueda encontrar al comparar con la palabra actual, así como devolver su valor.

Una vez se hayan realizado estos 2 procesos de rellenado, ya se tendrá la *FSM* lista para hacer uso de ella, y para hacerlo de una forma más cómoda lo que se recomienda es que el usuario coja la clase ***ModelUsingTools***, ubicada en la carpeta ***User Code***, la cual contiene todas las llamadas a métodos necesarias, para poder hacer uso de la *FSM* elegida, y sea este *script* el que asocie al *NPC* o *boss* que quiera hacer uso de la *API*. En esta clase, además de aparecer junto con cada método una pequeña explicación, se especificará al usuario cuales de dichos métodos deberán ser modificados por él y cuales no debe modificar. A continuación, se pasará a explicar en detalle la clase ***ModelUsingTools*** para que así se entienda tanto el motivo de realizar las llamadas a métodos que se realiza, el orden de dichas llamadas y el por que es necesario crear algunos métodos.

La primera acción que se realiza es importar la librería *GAI* para así poder hacer uso tanto de los métodos contenidos en la clase ***FSM_Manager*** y **FSM_Parser** como las variables inicializadas en la clase ***Tags***. Para importar librerías o carpetas en Unity se utiliza la directiva *using*.

Seguidamente se definen un par de variables para tener el código mas estructurado y así sea más fácil de entender las acciones que se están realizando. Una vez se tengan estas variables definidas el primer paso que sé tendrá que hacer será inicializarlas y crear la máquina de estados, y para eso en un método ***Start()*** se llevaran a cabo dichas acciones. El método *Start* es propio de Unity, y lo que nos asegura es que la llamada a este método se realizará una única vez y al principio de la ejecución del programa, garantizando así que lo que se ponga en este método será lo primero a ejecutar. Lo primero que se realiza en dicho método es la inicialización de tanto la lista de eventos, como el *FSM_Parser* y el *FSM_Manager* pasándole como argumento el *parser* inicializado anteriormente. Una vez se haya realizado este paso, ya se podrá añadir el *FSM* que se haya creado mediante el documento *XML*, acción que se puede realizar llamando al método *AddFSM()* y pasándo como argumento de dicho método la ruta

donde se encuentra el documento *XML* que contiene la *FSM*. Cuando se haya realizado esto ya se podrá crear la *FSM* llamando al método *CreateMachine()* pasándole como argumentos el objeto que hará uso de la *FSM*, el tag del tipo de la *FSM* y por último el *FSMid* de dicha *FSM* contenido en el documento *XML*. Este método no deberá ser modificado por el usuario.

El siguiente método implementado se llama ***WaitForAction()***en él lo que se hará será ir actualizando la *FSM* para ver si hay acciones que se tengan que realizar. Si hay acciones, se añadirán a las lista *DoActions(),* se recorrerá dicha lista y se llamará al método *ExecuteAction()* para que ejecute la acción que corresponda.
Para actualizar la *FSM* lo que se hará será realizar una llamada al método *UpdateFSM()* el cual realiza diversas acciones sobre la FSM para llevar acabo la lógica del tipo de FSM seleccionada, pero la acción mas importante para el usuario será una llamada interna a un *callback* el cual fue definido en el documento *XML*. Este *callback* lo que hará será, a partir de los eventos asociados a las transiciones, elegir que acciones se tienen que ejecutar, es decir, a partir de la lista de eventos creada (*EventsList*) se obtendrá la lista de acciones (*DoActions*) a realizar.

El siguiente método será el *callback* mencionado anteriormente, el cual tendrá que tener el mismo nombre que se haya puesto en el documento *XML* de la *FSM* que se vaya a emplear. En este método lo que se hará será realizar un *switch case* en el que se comprueba el tipo de *FSM* que se esté empleando, y dependiendo de las condiciones que se cumplan o no se añadirá el evento que se desee a la lista de eventos. Esta acción se realiza debido a que a partir de la lista de eventos la clase *FSM_Machine* se obtiene la lista de acciones a realizar. Las comprobaciones que se realicen serán las que provoquen que, en caso de cumplirse, se lleve a cabo las acciones asociadas al estado al que se pase. Es por ello por lo que se recomienda, evitar en la medida de lo posible que varios eventos se puedan dar a la vez, ya que el primero que se añada a la lista de eventos será el que se ejecute y si continuamente se está cumpliendo dicho evento no se saldrá del estado por mucho que ocurran cambios en el videojuego. Para evitar esto se recomienda el uso de variables que activen otros estados o bien si se está comprobando la distancia del jugador con el *boss*, establecer rangos de distancia para que no se quede continuamente en un estado como se ha comentado. Además de esto se recomienda que si uno de los estados va a ser el que más se emplee y los otros sean muy puntuales, como podría ser un estado de movimiento y otro de ataque, ya que el de ataque solo se efectuará en ciertas ocasiones y se estará mas en el estado de movimiento, sea la condición del evento de este estado puntual la que se compruebe antes que la del estado más empleado.

Para finalizar el ultimo método de esta clase será el método mencionado anteriormente llamado *DoActions(),* el cual se encargará de recibir las acciones que se tengan que realizar, y mediante un *switch*- case buscará la acción con dicho nombre e invocará el método encargado de llevar a cabo dicha acción. Se recomienda emplear métodos y no poner el contenido de los métodos en este apartado para una mejor implementación, así como organización.

Una vez se hayan hecho todos estos pasos ya se tendrá una clase la cual hace uso de la *API*, en concreto empleando la *FSM* que el usuario haya creado, con la lógica de la *FSM* elegida, pero con el nombre de los estados, transiciones, acciones y eventos que el usuario desee. Pese a que parezcan muchos pasos a seguir en última instancia lo único

que el usuario realizará será rellenar una plantilla dando los nombres que el desee, rellenar una clase con los mismos nombres dándoles un valor y por último poner las condiciones a los eventos y e invocar los métodos que lleven a cabo dichas acciones.

4.4.2 Sin hacer uso de las herramientas

Dado que, si no se utilizan ni los documentos *XML* ni la clase *FSM_Parser* se tendrán que realizar todas las llamadas internas para crear cada uno de los elementos del *FSM* elegido. Debido a que se tendrán que hacer muchos pasos y puede ser bastante fácil cometer un fallo se le proporciona al usuario un modelo de ejemplo en la clase ***ModelWithoutUsingTools*** de modo que solo tenga que copiar el esquema proporcionado para crear cada uno de los componentes de un *FSM* y rellenarlo con los datos correspondientes.

Para empezar las variables que se necesitaran serán las mismas que fueron empleadas en el apartado anterior, excepto si tampoco se hace uso de la clase Tags en cuyo caso en este apartado se añadirán las variables que se desee dándoles un valor.

Pese a que la forma de hacer uso de la API será idéntica si se hace uso de la clase Tags y simplemente se tendrá que sustituir por las variables que se hayan definido en el apartado de variables de esta clase si no se hace uso de ella, el método *Start()* se verá modificado completamente ya que es aquí donde se realiza la creación del *FSM* que el usuario desee emplear.Para empezar el usuario tendrá que crear un objeto de tipo *FSM_Manager* e inicializarlo ,y a continuación crea un objeto del tipo de la FSM que se desee y llamar al método que corresponda para así crear la *FSM* de dicho tipo, dándole las variables que sean necesarias. Cada *FSM* necesitará unas variables y su método tendrá un nombre diferente, razón por la que se buscaron y se pusieron todos los métodos para que le usuario no tenga que hacerlo.

A continuacion el usuario procederá con la creacion de los estados en los que nuevamente deberá crear un objeto de tipo State, inicializar dicho estado llamando al metodo que corresponda según el tipo de FSM elegido, y por ultimo añadirlo al fsm. Para que usuario no tuviera que buscar que metodo le corresponde a cada tipo de FSM se pusieron todos los metodos.Seguidamente se continuará con la creación de las transiciones que conecten los estados y para ello lo primero que deberá hacer el usuario será crear un objeto de tipo *Transition* y dos objetos de tipo *State* para así conectarlos. Seguidamente se creará el evento de dicha transición y creamos la transición, la cual dependiendo de la *FSM* tendrá que recibir más o menos variables. Por último, añadimos al estado origen y a la *fsm* creada dicha transición. Al igual que como ocurría anteriormente debido a que las transiciones cambian dependiendo del FSM a emplear se pusieron en comentarios el método a llamar dependiendo de la FSM a emplear.

Por último, y para finalizar, lo ultimo que tendrá que hacerse será llamar al método *Start()* del tipo de la FSM que se haya creado, para acabar de crear la FSM, así como

añadir al FSM_MANAGER la *fsm* creada y crear la *FSM,* la cual es de tipo *FSM_Machine*, del mismo modo que se hacía empleando las herramientas. Al finalizar todos estos pasos ya se tendrá una *FSM* de la que pueda hacerse uso, y como se ha comentado anteriormente, los pasos para poder emplearla serán los mismos que los vistos si se hiciera uso de las herramientas.

Debido a que como ha podido verse, los pasos a seguir son invocaciones a métodos ,así como realizar una correcta elección de los valores que se les pasa, el hacer uso de la *API* sin las herramientas puede tener como resultado un mayor numero de errores. Por ese motivo se decidió añadir este Modelo también, para que así el usuario simplemente tenga que sustituir los métodos con los valores correctos y ya pueda hacer uso de ella.

Para una mayor explicación y clarificación con ejemplos gráficos sacados del código, se recomienda revisar el Anexo *"Guía de utilización de la API"*. Este mismo documento, que aparecerá en el Anexo, será el que se añada como *pdf* con el mismo nombre al *asset* que se descargue el usuario para que así, pueda disponer de toda la información necesaria, pueda visualizar ejemplos y pueda tener una idea más clara de como hacer un correcto funcionamiento de la *API* tanto sin usar las herramientas como usándolas.

5. Campo de pruebas

Tal y como se ha mencionado anteriormente el campo de pruebas, que se ha realizado para verificar el correcto funcionamiento de la *API*, se ha hecho sobre la herramienta *Unity*, utilizando como lenguaje de programación *C#*.

Para el campo de pruebas se tomó la decisión de, en lugar de crear un videojuego propio, seguir los tutoriales guiados que proporcionaba la propia herramienta *Unity* para así, al finalizarlos, no sólo obtener un resultado tanto con unas mecánicas como con un apartado artístico mas llamativo que el que podría una persona, en el tiempo tan reducido que se tenía para la realización del trabajo de fin de grado, sino también un resultado validado del que con seguridad se sabe que no va a producir errores, eliminando así las posibles dudas que podrían surgir a la hora de realizar la integración de la *API* al videojuego sobre si el error ocurrido es por el videojuego o la propia *API*, facilitándo así tanto la identificación como la posible corrección de dicho fallo. Es por ello por lo que tras tomar esta decisión se procedió con la elección del videojuego mas apropiado, de los que *Unity* ofrecía, que se ajustase mejor como campo de pruebas para poder validar la *API* realizada. Tras estudiar las varias opciones que ofrecía la herramienta *Unity* se decidió que el videojuego elegido seria "*2D Roguelike tutorial*".[6]

La elección de este videojuego para ser realizado se hizo debido a que, a parte de que se trata de un videojuego en 2D con el que se aprenderá a trabajar tanto con imágenes 2D como a animarlas, en él aparecen no solo diversos enemigos con los que probar las

[6] Tutorial del videojuego 2D *Roguelike* de Unity: https://unity3d.com/es/learn/tutorials/s/2d-roguelike-tutorial

diversas *FSM* y poder ver varios enemigos con diversas *FSM,* si no que también el propio escenario es cambiante. Con cada carga del nivel, se produce una generación aleatoria del escenario, teniendo así por tanto en cada nivel un mapa nuevo, condición que resultó de sumo interés ya que así, además de añadir cierta inteligencia a los enemigos, se podrá ver si el comportamiento de éstos cuando el terreno es cambiante, y por tanto las condiciones que disparan los eventos van cambiando, es la esperada o no. Por último como aliciente, en este videojuego el comportamiento de los enemigos que aparecen es muy simple, por lo que también se querría mostrar al usuario que con pocos pasos y con el empleo de la *API* creada, se puede conseguir una mejora notable en la reacción de dicho enemigos frente al usuario. En los siguientes puntos se verá paso a paso como se realizó dicho videojuego, viendo tanto el apartado artístico como los *scripts* que fueron necesarios crear, y las modificaciones que se hicieron en estos para que así se pudiera visualizar los efectos de la *API* de una forma más cómoda. Además de esto, para que el lector que no tenga nociones de Unity pueda entender un poco la organización de dicha herramienta, se añadió en el anexo llamado "*Introducción a al motor de videojuegos Unity*" una explicación sobre las diversas ventanas que componen la herramienta, para que así al leer los siguientes puntos sepa el por que se accede a dicha ventana o donde se localiza el objeto del cual se está hablando.

5.1 Jugador

Para la creación de nuestro jugador lo primero que se tuvo que hacer fue ir a la *Hierarchy* y crear un objeto de tipo *GameObject* y nombrarlo como Player, para así poder empezar a crear a el jugador. El primer paso que se realizó fue darle el aspecto al jugador y para ello lo que se hizo fue, en la ventana *Sprites* , seleccionar las imágenes que correspondan con el jugador y arrastrarlas hasta el objeto *Player*.

Figura 19: *Sprites* que representan la animación *PlayerIdle*

Al hacer esta acción *Unity* preguntará donde se quieren guardar dichas imágenes y lo que se hizo fue crear una carpeta llamada *Animations* donde se guardaran las animaciones. Estas imágenes formaran parte de la animación *PlayerIdle*, las cuales se pueden observar en la figura 19. Tras esto se podrá observar como se han creado 2 objetos en dicha carpeta la animación llamada *PlayerIdle* y el controlador de las animaciones del jugador llamado *Player*. Para añadir las demás animaciones se realizo el mismo paso, elegir las imágenes que formen parte de la animación, arrastrarla al objeto Player y darles el nombre que se desee. Además de la animación de *PlayerIdle* se añadió la animación *PlyaerHit*, figura 21,para cuando el jugador es golpeado y la

animación *PlayerChop*, figura 20,para cuando el jugador destruya algún elemento del terreno.

Figura 20: *Sprites* que representan la animación *PlayerChop*

Figura 21: *Sprites* que representan la animación *PlayerHit*

Una vez se realizaron estos pasos, se procedió a realizar la lógica para pasar de una animación a otra, la cual se guardará en el controlador de animaciones que se creó anteriormente llamado *Player*. En dicho controlador lo que se hizo fue establecer como estados las animaciones que se habían creado y como transiciones crear las condiciones para pasar de una a otro al igual que la duración de cada animación. Se estableció como animación inicial la animación *PlayerIdle* y como condiciones *playerChop* para pasar a la animación *PlayerChop* y *playerHit* para pasar a la animación *PlayerHit* como puede verse en la figura 22. El cuándo realizar dichas transiciones se verá mas adelante en el código que se asociará al jugador.

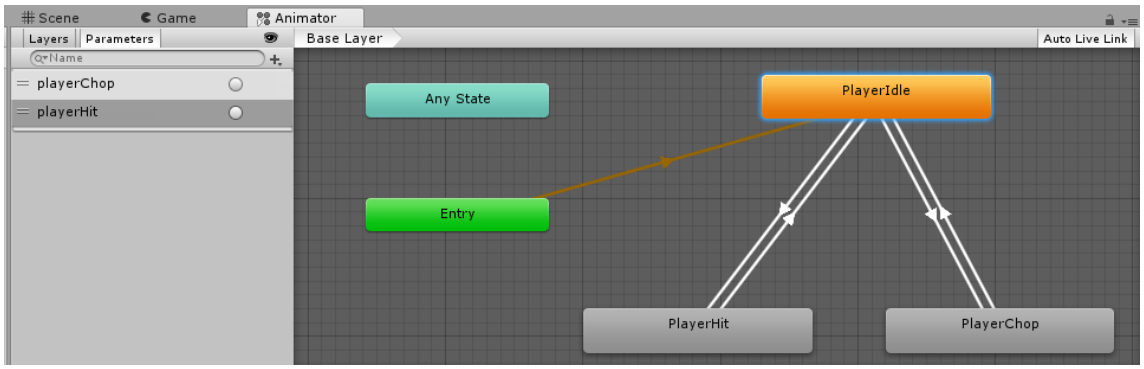


Figura 22: *AnimatorController* del jugador

Para finalizar y dejar el objeto jugador listo para guardarlo como *Prefab,* y así poder usarlo más adelante, se realizarán una serie de modificaciones en el Inspector las cuales se pueden observar en la figura 22. Lo primero será etiquetar al jugador como un objeto de tipo Player, acción que se realiza seleccionando la etiqueta Player del campo Tag. Seguidamente se indicará que dicho objeto será de tipo *BlockingLayer*, lo cual significa que dicho objeto será un objeto bloqueante, el cual puede ser bloqueado por otros objetos que formen parte de la escena. Dentro del apartado *Sprite Renderer,* en la opción *Sorting Layer*, se deberá elegir la opción *Units* la cual lo que le indicará a *Unity* es que el objeto jugador debe ser el ultimo objeto en renderizarse. Esto se hará para que al empezar a ejecutar el videojuego lo primero que se haga sea generar el mapa, luego

los objetos encima del mapa y luego el jugador. Seguidamente, lo que se realizará será añadir físicas al objeto jugador, por lo que se deberá añadir un componente *RigidBody 2D* e indicar en la opción *Body Type* que el objeto será de tipo *Kinematic,* lo cual quiere decir que a dicho objeto le afectaran todas las físicas menos la de la gravedad (ya que si le afectase la gravedad el jugador se caería fuera del mapa puesto que la vista es cenital), y también se añadirá un componente de tipo Box Collider 2D para así delimitar la zona del jugador, y se ajustará al tamaño del *sprite* del mismo. Para finalizar se añadirá en el apartado *Animator*, en la opción *Controller* el controlador de animaciones que se creó anteriormente llamado Player.

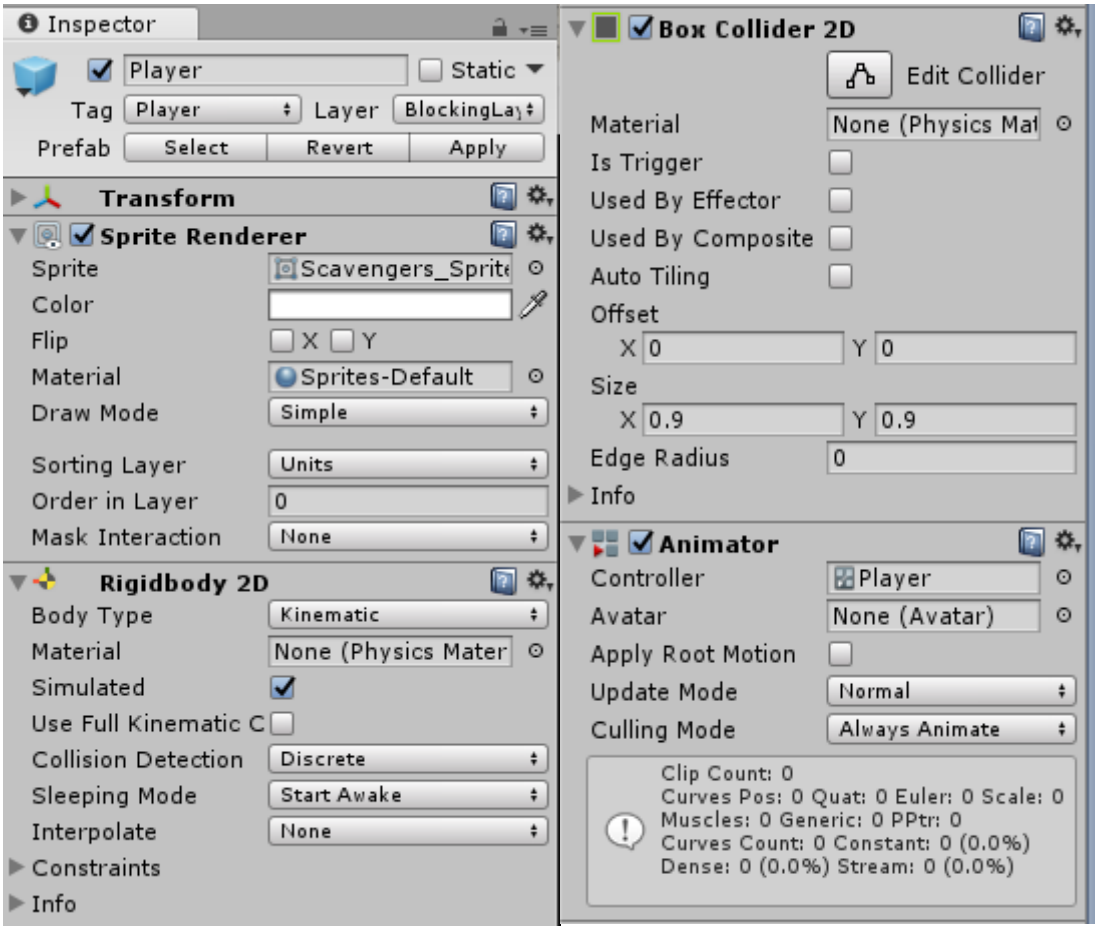


Figura 23: Inspector del Jugador

Una vez se completaron estas acciones ya se obtuvo un objeto jugador y para completarlo lo único que hacía falta era asociarle el *script* que representase su comportamiento.

Debido a que tanto los enemigos como el jugador iban a compartir métodos, tales como comprobar que objeto tienen delante o si se pueden mover, se creó una clase llamara *MovingObject* de la cual heredarían tanto el jugador como los enemigos. En esta clase se tiene un método *Awake()* que sirve para obtener el objeto que está haciendo uso de la clase y un método *Start()* para decirle la velocidad de movimiento. También se tienen una serie de métodos encargados de comprobar que objetos tienen delante y realizar el movimiento hasta la posición indicada en caso de que se pueda. Estos

métodos son *Move(),* el cual empleando el concepto de *Raycast*, el cual consiste en trazar una línea invisible entre dos puntos y comprobar que objeto hay en medio, devolverá false si hay un objeto bloqueando el paso o true y realizará el movimiento llamando al método *SmoothMovement(),* el cual en cada fotograma se encarga de acercar el objeto desde el punto de inicio hasta el punto final. Estos dos métodos serán llamados desde otro método llamado *AttemptMove*() el cual también llamará a un último método abstracto llamado *OnCantMove*() el cual, en el caso del jugador comprobará si no se puede mover por que el objeto que lo bloquea es un muro interno, en ese caso lo podrá destruir, y en el caso del enemigo si es el jugador quien lo bloquea le golpeará.

Una vez se realizó este *script* se procedió con la creación del *script Player* el cual como se ha comentado heredaría del script *MovingObject*. En este *script* lo que se hizo fue, implementar los métodos de la clase *MovinObject* pero adaptándolos al caso del jugador, ya que este cuando se moviese tendría que perder puntos e ir mostrándolos en un texto en la escena, además de comprobar cuando colisione con un objeto si dicho objetos es un muro interno, en cuyo casi le hará daño al muro y ejecutará la animación de *PlayerChop*. También se añadieron algunos métodos más tales como *OnTriggerEnter2D*(),el cual se encarga de comprobar con que objeto ha colisionado, si el objeto es la salida se ha superado el nivel y se carga una nueva escena y si es comida o soda se suman puntos, *LoseFood*(), el cual hace que el jugador pierda puntos, los muestra en un texto en la escena y ejecuta la animación de *PlayerHit*.

Para finalizar también se añadió a este *script* una variable que almacenaría el objeto jugador en cada escena y en se modificó el método *Awake*() para que comprobase si esa variable ya tenia un valor, si es así lo elimine y guarde el nuevo valor, para que si se ha pasado a otra escena sea el nuevo objeto jugador y no el antiguo el que almacene. Esto se hizo para desde el enemigo acceder a los métodos del jugador.

Cuando ya se tuvo el *script* finalizado se le asoció al *prefab* del jugador y se le dieron valor a alguna de las variables tales como el daño que se le haría a los muros interiores y los puntos que recibiría por la comida y la bebida.

5.2 Enemigos

Los pasos para crear las animaciones de los enemigos fueron los mismos que los seguidos con el jugador, pero cambiando los *sprites*, y con una animación menos ya que los enemigos solo presentan dos animaciones *EnemyIdle* y *EnemyAttack*, las cuales se pueden observar en las figuras 24 y 25 respectivamente. También el controlador de animaciones siguió el mismo procedimiento de creación solo que solo había una condición para pasar de la animación *Enemy1Idle* a la animación *Enemy1Attack*, la cual se llamó *enemyAttack* como puede verse en la figura 26.

Figura 24: *Sprites* que representan la animación *Enemy1Idle*

Figura 25: *Sprites* que representan la animación Enemy1Attack

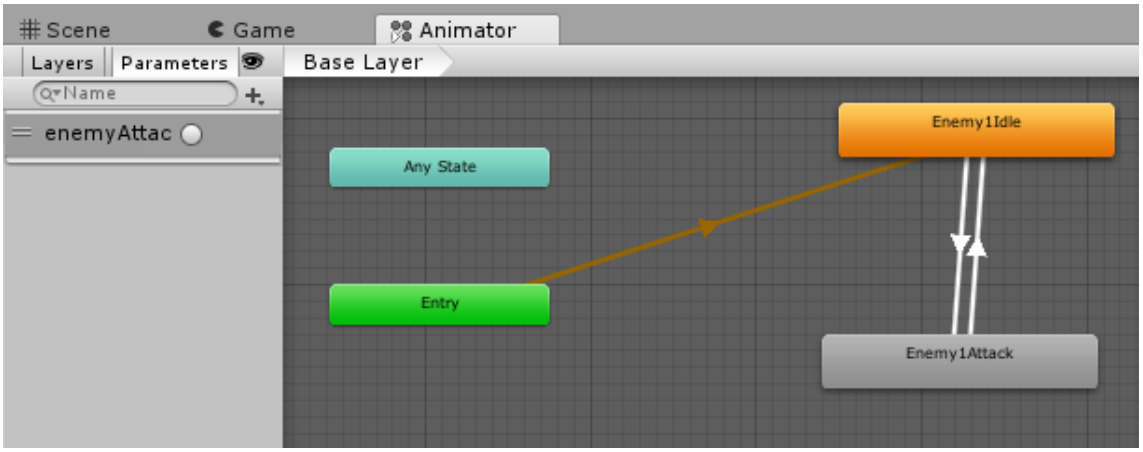


Figura 26: *AnimatorController* del enemigo 1

Por último, como paso final se realizaron los mismos cambios en el Inspector que se hicieron en el jugador, pero en esta ocasión la etiqueta en lugar de ser *Player* se cambió a *Enemy*.

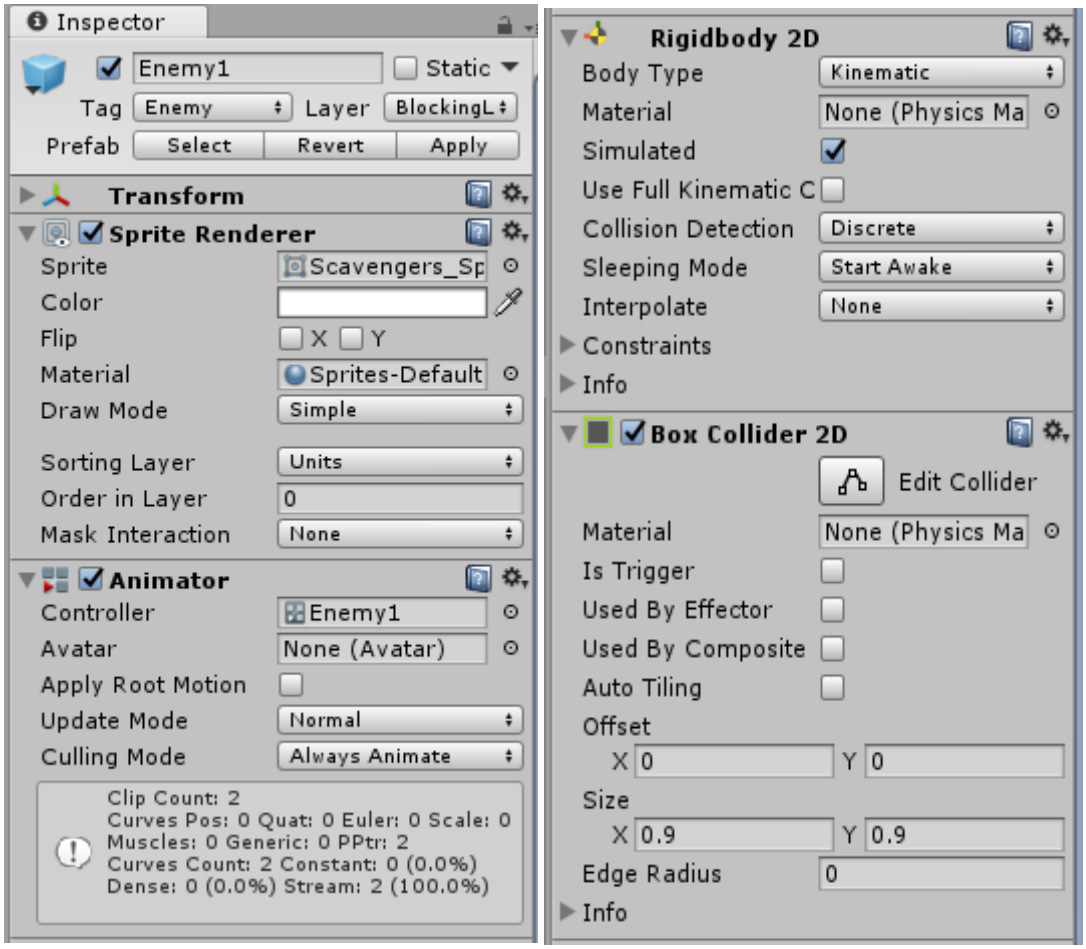


Figura 27: Inspector del enemigo 1

Debido a que los dos tipos de enemigos tienen las mismas animaciones y el mismo comportamiento simplemente con el *prefab* de *Enemy1* , después se duplicó y se cambiaron los *sprites* para tener las animaciones *Enmy2Idle* y *Enemy2Attack*, las cuales se pueden ver en las figuras 27 y 28 respectivamente, y el controlador de animaciones junto con el inspector como se puede observar en las figuras 29 y 30 respectivamente.

Figura 28: *Sprites* que representan la animación *Enemy2Idle*

Figura 29: *Sprites* que representan la animación *Enemy2Attack*

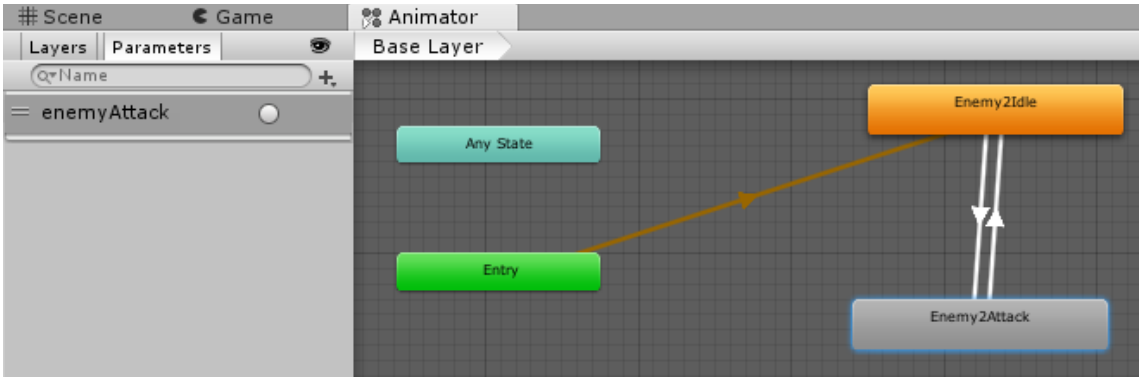


Figura 30: *AnimatorController* del enemigo 2

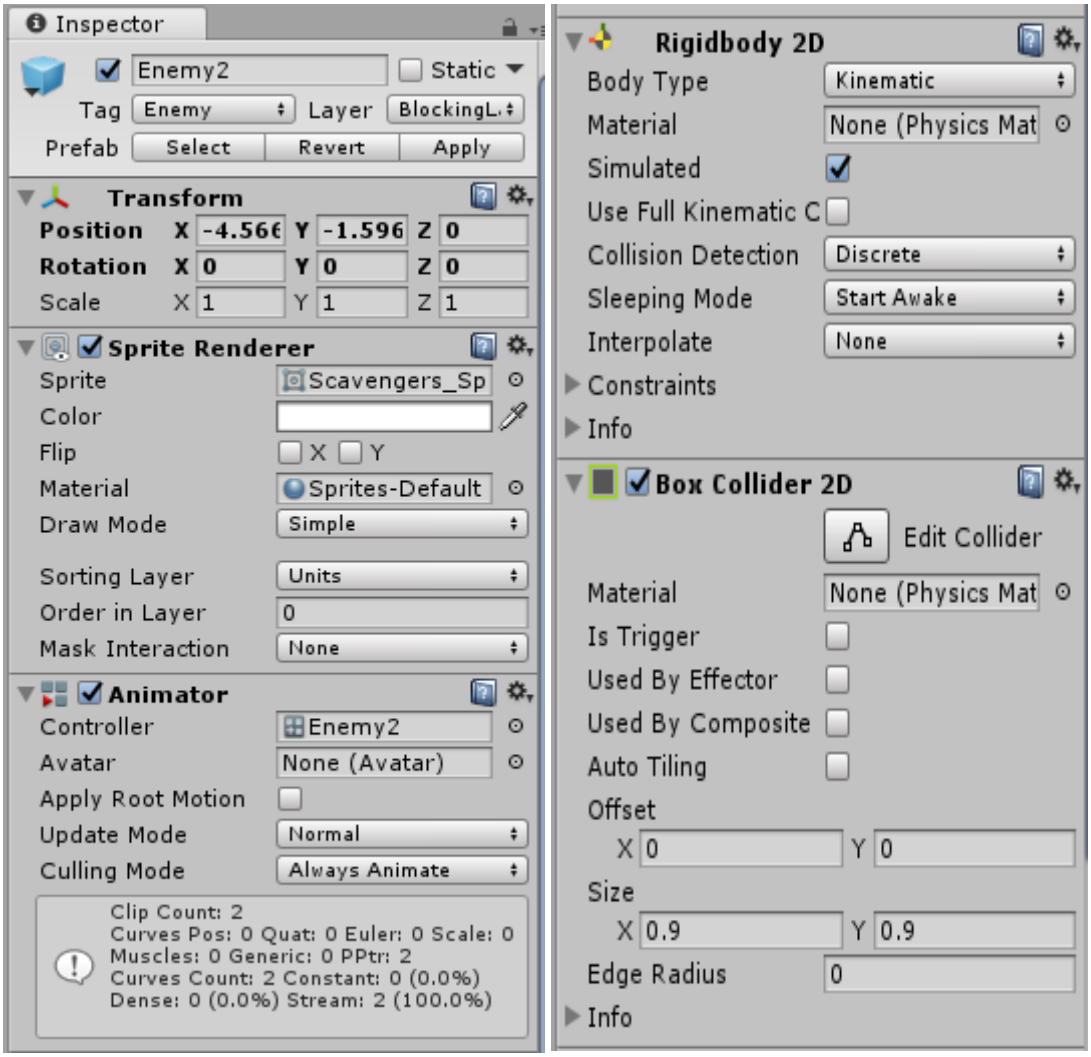


Figura 31: Inspector del enemigo 2

Una vez ya se realizaron los 2 *prefabs* de los enemigos que aparecerían en la escena se procedió con la creación del *script* *Enemy* el cual al igual que el *script* de *Player*, heredará de *MovingObject* y por tanto implementará los métodos de dicha clase pero adaptándolos al caso del enemigo, ya que si colisiona con el jugador este deberá perder puntos y el enemigo deberá ejecutar la animación de *EnemyAttack*. Además de esto, al ser esta clase la que va a hacer uso de la *API* se añadieron todos los métodos y se realizaron todas las acciones necesarias para realizar un correcto funcionamiento de la misma, tal y como se ha visto en el apartado de Utilización de la *API*. En los únicos métodos que se hicieron modificaciones fueron en el método *AttemptMove(),* del cual heredan tanto *Enemy* como *Player*, para que en el caso de *Enemy* no se llamase a *OncantMove()* ya que dicho método hace que se golpee al jugador, y en los métodos de la API que debían ser modificados. Este cambio en la clase de *MovingObject()* se hizo así para poder separar la acción de moverse de la de atacar. El *callback Events()* se modificó para que se comprobase, en el tipo de *FSM* que se estuviera haciendo uso, si se cumplían las condiciones para añadir el evento correspondiente a la lista de eventos, y llevar a cabo así la acción que tocase. Por último, en el método *ExecuteAction()* se añadieron todas las acciones a realizar, así como los métodos a los que tendría que llamar para llevar a cabo dicha acción, así como añadió un apartado para obtener una referencia al jugador y poder llamar al método *LoseFood()* de la clase Player al recibir el jugador un golpe.

5.3 Escenarios

Para la realización del escenario donde estarán tanto el jugador como los enemigos se llevó a cabo el mismo procedimiento que el visto anteriormente, pero debido a que no tendrán ninguna animación asociada y cada objeto solo tendrá un único *sprite* asociado lo que se hará para hacerlo de una forma más rápida es coger el *sprite* que represente el objeto que se desee y se suelta en la ventana *Hierarchy*. Esto lo que provocará es que se cree un objeto de tipo *GameObject* el cual tenga un componente de tipo *Sprite Renderer* con el *sprite* que se ha soltado asociado directamente. Esta acción se tendrá que hacer con los 8 sprites que representan al suelo, los cuales se pueden ver en la Figura 32, cambiándole a cada uno el nombre por floor 1 hasta floor 8 e indicando en la opción *Sorting Layer*, dentro de *Sprite Renderer*,tal y como puede observarse en la Figura 33, que dicho objeto es de tipo *Floor*. Esto lo que le indicará a Unity es que este objeto tiene que ser lo primero que se genere en el juego.

Figura 32: *Sprites* que representan el suelo del escenario

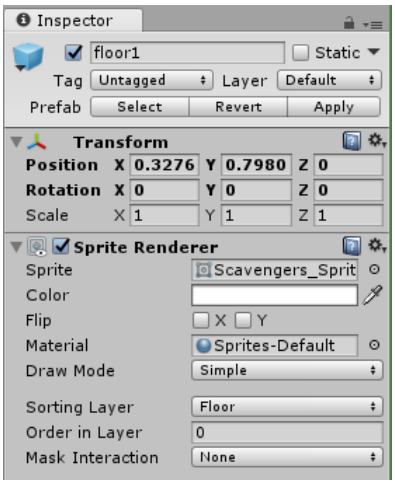


Figura 33: Inspector del suelo del escenario

Una vez se ha hecho esto con el suelo, ya se podrá guardar como *prefab* y se continuará con la creación de los muros que rodeen el mapa, así como la comida y bebida que le otorgaran puntos al jugador y la salida que le permitirá al jugador pasar al siguiente nivel. Para aprovechar lo que ya se ha realizado lo que se hará será coger uno de los *prefabs* de suelo terminados, cambiarle el *sprite* por el deseado y el nombre y añadirle un componente *Box Collider 2D*, ya que tanto el jugador como los enemigos si deberán colisionar con estos objetos. Es por ello por lo que para empezar se añadirá a uno de estos *prefabs* el *sprite* que represente a la salida, el cual se puede ver en la figura 34, se le añadirá el componente mencionado anteriormente y se marcará la opción IsTriger ya que sí que se quiere que le jugador colisione con la salida, pero que la pueda atravesar. Para finalizar se pondrá como tag *Exit*, y en *Sprite Renderer*, en *Sorting Layer*, ponemos en lugar de *Floor Items* para que así se genere después del suelo, y ya se podrá guardar este objeto como *prefab Exit*. Una vez se haya hecho el *prefab* de la salida, tanto el de la comida como el de la bebida será exactamente el mismo, solo que en el caso de la comida se marcará como Tag Food y se guardará dicho *prefab* como Food y en el caso de la bebida se marcará como Tag Soda y se guardará ducho *prefab* como Soda. En la figura 34 se podrán ver los *sprites* de estos 3 objetos y en la figura 35 se podrá ver el inspector de uno de dichos objetos debido a que los otros 2 objetos son muy similares.

Figura 34: *Sprite* de los objetos *Soda*, *Food* y *Exit*

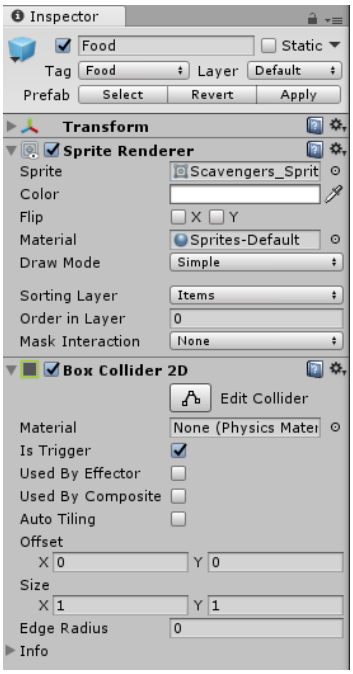


Figura 35: *Inspector* del objeto *Food*

Por último, para acabar con la creación de los *prefabs* necesarios para la creación del escenario, se crearán los muros. Dentro de los muros tenemos unos que irán por fuera, los cuales servirán para delimitar la escena, y otros que se generarán dentro del escenario. La única diferencia entre ambos tipos de muros es que mientras que los exteriores en su *Sprite Renderer* se seleccionará, en la opción de *Sorting Layer,* Floor, en los muros interiores se seleccionará *Items* para que así se generen estos muros por encima del suelo. Una vez hechas las modificaciones guardaremos los muros exteriores como OuterWall, como puede verse en la figura 36, y los muros internos como Wall, como puede observarse en la figura 37.

Figura 36: *Sprites* del objeto *OuterWall*

Figura 37: *Sprites* del objeto *Wall*

Una vez se hayan realizado todos los *prefabs* necesarios para la creación del escenario, se crearán 2 *scripts*, uno llamado ***BoardManager***, el cual se encargará de crear cada nivel y otro llamado ***GameManager***, el cual se encargará de gestionar el videojuego. Estos 2 *scripts* irán asociados a un objeto de tipo *GameObject* al cual se llamará *GameManager*.

Lo primero que se hará en el *script BoardManager* será definir las variables que nos serán de utilidad para generar el nivel. Los *arrays* que se crean de los objetos de comida, enemigos… se ponen públicos para que en el objeto *GameManager* pongamos los *prefabs* de cada uno de dichos objetos. También como variables necesarias se tendrá una lista de las posiciones de los objetos que ya se han generado en el mapa, y así evitar que varios objetos ocupen la misma posición, dos variables para delimitar el tamaño del tablero y variables para representar tanto los muros interiores como los exteriores y la salida.

Este *script* constará de un método el cual servirá para inicializar el escenario con los *prefabs* del suelo, crear los muros exteriores, interiores y colocar sobre el suelo tanto la comida como los enemigos y la salida. Este método se llama *SetUpScene(int level),* el cual a partir del nivel que sea creará un nuevo nivel con un escenario generado de forma aleatoria. Para que este método no presente un código muy largo y diverso lo que se hará es organizarlo en diferente método y que *SetUpScene()* los llame conforme los vaya necesitando. El primer método del cual hará uso será el método *BoardSetUp()* ,lo que hará será ir recorriendo las columnas y las filas del tablero, si se encuentra en el borde del tablero lo que hará será coger de forma aleatoria uno de los *prefabs* de los muros exteriores y colocarlo en la escena, pero si no estamos en los bordes se cogerá un *prefab* aleatorio de suelo y se colocará en la escena. Para obtener los prefabs aleatorios simplemente se creó un método llamado *GetRandomInArray(GameObject[] array)* el cual devuelve uno de los elementos de dicho array empleando la directiva *Random.Range*(valor inicial,longitud del array)

El siguiente método que empleará será *Initializelist*(), el cual lo único que realiza es, primero vaciar la lista de las posiciones de los objetos y después ir rellenándola para asi saber que objeto ocupa cada posición. Los siguientes métodos son el mismo solo que con valores diferentes el cual es el método *LayOutObjectAtRandom(GameObject,numero mínimo,numero máximo),* el cual lo que hace es colocar en una posición aleatoria un numero aleatorio del tipo de objeto que se le pase, asegurando que como mínimo habrá el número mínimo que se le indique y como máximo el numero máximo que se le indique. Para ello, lo que se hace es comprobar el tipo de objeto que se le pasa, para en la escena organizar los objetos que se van creando por tipo y no tener todos los objetos de la escena en la *Hierarchy* desordenados, luego se obtiene una posición y un *prefab* del objeto aleatorio y se coloca en escena.

Por último, lo que hará el método *SetupScene*(int level) será colocar la salida, la cual como tiene que estar arriba a la derecha no hace falta colocarla de forma aleatoria, simplemente indicando el objeto y la posición en la que se quiera que aparezca es suficiente.

Una vez realizados estos métodos se rellenó el *script GameManager* el cual presenta un método llamado *InitGame()* que se encarga de mostrar los puntos de comida del

jugador y mostrar una pantalla inicial en la que se muestra el día o nivel en el que se está, así como inicializar la escena y la lista de enemigos. También existen diversos métodos como *HideLevelImage()* que se encarga de activar o desactivar la imagen inicial o *GameOver()* que muestra una imagen al llevar a 0 puntos de comida. A parte de estos también existen los métodos como *OnEnable()* o *OnDisable()* que se encargaran de añadir o quitar la escena y el método *OnLevelFinishedLoading()* que se encarga de ir incrementando el nivel. Por último, en esta clase se encuentra el método *MoveEnemies()* método el cual se encarga de realizar mediante corrutinas el movimiento de los enemigos cuando no sea el turno del jugador. Este método se modificó para que, en lugar de llamar al método creado en el *script enemies* llamado *Move()* el cual además del movimiento realizaba un ataque al jugador si este estaba próximo, se llamase al método que realiza la actualización de la lista de acciones a realizar.

Para finalizar con el apartado de la generación del escenario lo que se realizó fue un *script* más llamado *Loader* el cual lo único que hace es comprobar si existe alguna instancia del objeto *gameManager* y este script se le asoció a la cámara de la escena. Esto se hizo para que, cuando comience la ejecución del juego, el único objeto que hay en la escena son el jugador y la cámara, la cámara busca si existe este objeto *gameManager* y como no existe lo crea, el cual comenzará a generar el escenario con todos los objetos y enemigos.

5.4 Videojuego final

En este apartado de la memoria se verá cómo quedó el videojuego tras la finalización de éste, así como una explicación de las acciones que lleva a cabo el videojuego y el jugador cuando juega a él.

Nada más empezar el videojuego le aparecerá al jugador un texto con el título Day X siendo x el nivel en el que se encuentre. Conforme vayan aumentando los niveles este número se verá modificado. Tras mostrar este texto se comenzará con el videojuego, generándose tanto el mapa y rellenándolo con el jugador, la salida donde tiene que llegar y los muros internos, la comida y los enemigos. Tanto el jugador como la salida se generan siempre en la misma posición, generándose el jugador en la primera casilla de la última fila y generándose la salida en la última casilla de la primera fila. Por otro lado, todos los demás objetos que aparecen se generan siempre en una posición aleatoria, comprobando previamente que ningún objeto se ha generado en una posición ocupada. Todo este escenario y objetos se generan de forma ordenada gracias a como se etiquetaron dichos objetos tal y como se vio anteriormente. El número de objetos que aparecerán en el mapa también es aleatorio, menos en los enemigos, puesto que estos van creciendo exponencialmente, ya que así se generan menos enemigos y el juego dura más niveles, ya que de no ser así en pocos niveles habría demasiados enemigos, dificultando así la jugabilidad. Una vez se han generado todos los componentes que conforman el videojuego se le permite al jugador que éste se mueva por el mapa

mediante las flechas direccionales del teclado. Cada dos movimientos que realice el jugador, los enemigos procederán a evaluar la ubicación de éste e intentaran moverse hacia su posición. El jugador podrá recolectar comida, la cual aumentará el indicador de comida el cual aparece fuera del mapa. Dicho indicador también disminuirá conforme el jugador se vaya moviendo, ya que consume comida para poder llevar a cabo dicha acción y a su vez también perderá comida cuando sea golpeado por un enemigo. Además, el jugador podrá romper los bloques que se encuentre dentro del mapa para así poder pasar por cierta zona que este bloqueada, aunque esto le hará perder comida. Cuando el jugador llegué a la salida, se cargará un nuevo nivel totalmente aleatorio y el texto inicial en el que aparece Day X, aparecerá con dicha idea incrementado, puesto que ya se habrá superado dicho nivel. Por ultimo cuando el jugador haya consumido mucha comida, ya sea por haberse movido mucho, no haber cogido comida o por haber recibido muchos golpes de los enemigos, y el contador de *Food* llegue a 0, el juego finalizará y se procederá a mostrar un texto en el que aparezca En las siguientes imágenes se resume de forma grafica las diversas acciones que se llevan a cabo en el videojuego, así como las que puede realizar el jugador dentro de él.

Figura 38: Ejecución del videojuego final

6. Errores y cambios realizados

En este apartado de la memoria se comentarán los errores que se obtuvieron al integrar la *API* al campo de pruebas que se ha realizo, la causa de dichos errores y como se solventaron.

Tal y como se comentó anteriormente la *API* de la cual se partía tenía una estructura parcialmente organizada por lo que los primeros pasos que se hicieron fue organizar la *API* para que quedase como se mostró en el apartado de Estructura de la *API*.

Una vez establecida la nueva estructura, para hacer uso tanto del código de las *FSM* como de los documentos *XML* había que ,previamente, indicar que se estaba haciendo uso de ellas mediante la instrucción *using* y el nombre del *namespace* que se le hubiera dado a cada *FSM*. Para que tuviera sentido con el *asset* que se estaba realizando este *namespace* se modificó para que fuera el de *GAI* en todas las FSM, en la *FSM_Machine y FSM_Manager ,* así como en el *FSM_Parser*, ya que originalmente para hacer uso de los métodos de la *FSM _Machine* o *FSM_Manager* había que hacer un *using* y para hacer uso de los *XML* había que hacer otro *using* más, esto se unificó y se llamó a todo *GAI*. Debido a esto al hacer uso del *parser* ya no era necesario indicar que se estaba accediendo al *namespace xmltest*(*xmltest.FSM_Parser*)ya que todos pertenecían al mismo *namespace* y como consecuencia la clase *FSM_Manager* tuvo que ser modificada para que el constructor recibiese elementos *FSM_Parser*.

Inicialmente al intentar integrar la *API* al campo de pruebas que se había realizado se obtuvieron bastantes errores, por lo que se decidió empezar solventando los errores relacionados primero con los documentos *XML* y la clase *Tags*, luego con el *Parser* y por último con el código de las *FSM*. Esto se decidió de esta forma ya que se pensó que las partes de la API que más se modifican, serían las que más fáciles de producir errores serian.

Debido a esto se decidió ir paso a paso y mediante el uso de la instrucción *Debug.Log* de *Unity* ir viendo en que tramos de la ejecución ocurría el error. El primer error que se detectó surgió al intentar crear la *FSM* mediante la instrucción *CreateMachine*() ya que esta siempre devolvía *NULL*. Debido a que este método recibe 3 parámetros, uno el objeto que hace uso de la *FSM*, otro el tipo de *FSM* y por último el nombre que se le puso en el apartado *FSMid* del documento *XML* que se está empleando, la primera fuente de error que se revisó fue el documento *XML*. Pese a que primera vista las plantillas de las *FSM* en los documentos *XML* de los que se partió parecían correctos, tras un estudio más en profundidad se observó que algunas de las plantillas tenían algunas etiquetas mal escritas. Una vez solventado este problema con las plantillas surgió otro más a la hora de actualizar la lista de acciones a realizar, problema bastante difícil de averiguar su fuente de error ya que, como se ha visto, para obtener la lista de acciones lo que se hace es llamar al método *UpdateFSM*(), el cual llama a un *callback* que actualiza la lista de eventos y a partir de ellos se obtienen las acciones a ejecutar.

Como primera hipótesis se pensó que el error estaría al poner un nombre diferente en el *callback* del documento *XML* y el método de la clase a implementar la *API*, sin embargo, esto no fue así. Tras hacer *Debug* para ver si cada una de las partes que formaban las *FSM* se estaban creando de la forma correcta y comprobar que esto era así se comprobó que todos los estados juntos con sus acciones se creaban, las transiciones se creaban, pero cuando se llegaba a los eventos asociados a dichos estados estos no se creaban y eran la fuente del problema. Dado que se observan las plantillas y todo parecía correcto en una prueba se descubrió que el error residía en los comentarios que se habían dejado en las plantillas. Justo al finalizar la etiqueta de un evento y al cerrar la etiqueta de todos los eventos se había añadido el siguiente comentario: ***</Event> <!--Add another <Event> if you want--> </Events>***, esto lo que provocaba es que el *parser* no interpretase bien la etiqueta de finalización de los eventos y no dejaba actualizar bien la lista de eventos mediante el método *UpdateFSM()*.Para solventar esto se modificó dicho comentario y se movió al principio de la etiqueta de los eventos para que el usuario siguiese sabiendo que puede añadir varios eventos a una misma transición y que este comentario no provoque más errores.
<Events> <!--Add another <Event> if you want between <Events> *** </Events> **--></Events>**

Por último, otro error que surgió a raíz del *parser* ocurrió cuando ya se habían solventado todos los errores y se estaba probando con los usuarios la API. Éstos al intentar crear una *FSM* desde o haciendo uso de las XML , que se proporcionaban como modelos , obtenían siempre el mismo error; *NullPointerException*. Tras mucha investigación se descubrió que el error procedía al añadir al *FSM_Manager* la *FSM* mediante el *path* donde estaba ubicado dicho documento. Algo curioso de esto era que mientras que los *XML* con las *FSM* que ya se habían creado si que funcionaban, y si se copiaba su contenido en otro documento si se podía hacer uso, pero si se realizaba la más mínima modificación se obtenía el mismo error. Debido a que ya no era un error ni de las plantillas ni de los usuarios al crear las FSM se pasó a analizar los posibles errores tanto del *FSM_Manager* como del *FSM_Parser*. Tras descargar que el error fuera del *script FSM_Manager* se pasó a analizar el *script FSM_Parser* y en el se encontró la fuente del error, el cual se daba al internar hacer una escritura en un documento *.txt* para así saber que elementos de la *FSM* se estaban creando. Al hacer la escritura de dicho documento se estaba accediendo a un *path* erróneo por lo que se tuvo que modificar. Aun realizando este cambio, el error persistía y debido a que se consideró que dicha acción de control sobre que se estaba creando se podría llevar a cabo realizando invocaciones al método *Debug.log(),* se eliminó de la *API*.

Una vez se realizaron estos cambios se pudieron crear las *FSM* empleando tanto los documentos *XML* como la clase *FSM_Parser* y realizar las pruebas en el campo de pruebas creado sin problemas excepto cuando se quiso probar la *FSM* basada en estados concurrentes la cual producía el siguiente error:

ArgumentException: An element with the same key already exists in the dictionary.

System.Collections.Generic.Dictionary`2[System.Int32,System.Int32].Add (Int32 key, Int32 value) (at /Users/builduser/buildslave/mono/build/mcs/class/corlib/System.Collections.Generic/Dictionary.cs:404)

Este tipo de error en este caso se estaba produciendo por que el diccionario que tiene asociada la *FSM* basada en estados concurrentes se estaba añadiendo el mismo tag varias veces. Este no era el error original, puesto que haciendo *Debug* se comprobó que

el *Tag* que se estaba agregando repetidas veces era el *Tag UNKNOWN* indicando así que la creación del *FSM* no se estaba realizando correctamente. Sin embargo, se corrigió este error agregando, antes de añadirle los créditos correspondientes a dicho Tag, una comprobación para ver si ya pertenecía una entrada al diccionario con el mismo valor, indicando así que esa etiqueta ya estaba agregada y por tanto no se deberia añadir otra. Debido a que esta clase tenia una gran cantidad de código comentado que no se empleaba, localizar el fallo fue bastante tedioso por lo que la primera acción fue, tras comprobar que realmente el código comentado no era útil, eliminarlo para así poder buscar el error. Tras esto, el error que apareció indicaba que el fallo estaba en el método *UpdateFSM*() por lo que el error se encontraba en el *callback* que se había creado. Por ello se volvió a copiar el esquema del XML del FSM y se rellenó, dando esta vez un buen resultado. Sin embargo, este fallo ayudó a depurar un poco la clase, hacerla más fácil de visualizar y realizar cambios, así como solucionar un problema secundario.

Al solventar estos errores ya se podía crear una *FSM* del tipo que se eligiera, sin embargo, en el caso de las *FSM* basadas en pilas el comportamiento tal y como se verá en el siguiente apartado no fue el esperado, puesto que al etiquetar como *STACKABLE* las transiciones que iban hacia estados mas prioritarios estas no llegaban a ejecutarse nunca, dando como resultado, que no se realizase este apilado que se busca en este tipo de *FSM*.

Por otro lado, al crear las *FSM* basada en estados concurrentes estas dejaron de dar errores al intentar crearlas sin embargo cuando se creaban, la FSM aparecía vacía, razón por la cual se comprobó si el *fsm_manager* estaba haciendo un uso correcto del *parser*. Por ello se comprobó que el método *AddFSM()* estaba añadiendo la *FSM* que se había definido mediante los documentos *XML* de forma correcta. También se comprobó que los valores que internamente recibía el método eran los correctos.Puesto que el error residía en el método *CreateMachine()* se comprobaron que todos los valores que se le pasaban a dicho método fueran los correctos y que internamente dicho método hiciera uso de dichos valores y crease una máquina del tipo *FA_Concurrent_States*. Al estar este motodo correcto se procedió a buscar en el método encargado de la creación de dicha máquina, el método *FSM_Machine()* de la clase *FSM_Machine*. Dicho método tenia un caso especial para cuando la FSM fuese de este tipo por lo que se empezó a comprobar si cada variable recibía correctamente su valor y se inicializaban correctamente. Con esto se vio que eran correctos todos los parámetros que recibía y que empleaba por lo que comenzó a estudiarse la lógica de la *FSM* por si esta presentaba algún fallo. Tras hacer *Debug* en todos los métodos se comprobó que cuando tenia que asignar los créditos que le correspondían al estado inicial en lugar de hacer eso se le otorgaba como valor <i--Initial has to be "YES" or "NO" !--> etiqueta que aparecía como comentario al inicio de la creación de los estados para indicarle al usuario que valor puede colocar y cual no.Pese a que encontrar que originaba este error fue una tarea muy tediosa y larga, sirvió para ver que los documentos *XML* podían llegar a ser una enorme fuente de falles, por lo que se decidió eliminar todos los posibles errores, eliminando los comentarios que aparecían en los XML, y en el apartado de guía de utilización de la API añadir ahí los posibles valores que se le pueden dar a cada etiqueta.

En cuanto al comportamiento de las *FSM* basada en pilas se observó como cuando se actualizaba la *FSM* para obtener las acciones a realizar, si se cumplía el evento para ir a la transición marcada como *STACKABLE*, en lugar de devolver el valor de la acción a llevar a cabo se devolvía 1000 cuya etiqueta correspondía con el valor de la etiqueta *UNKNOWN*. Es por ello por lo que se comenzó comprobando si al emplear el documento *XML*, la etiqueta se clasificaba como *STACKABLE* de forma correcta, acción que se llevaba a cabo. Seguidamente se pensó que posiblemente el error estaría en el método *FSMUpdate()* y que este cuando se tenia dicha etiqueta no se llevaba a cabo dicha acción. Esto era en parte lo que ocurría, ya que observando el código se pudo comprobar que para que el comportamiento de la FSM basada en pilas fuese el correcto, era necesario que todos los eventos, independientemente de que pudieran o no ser apiladas, tuvieran una transición que fuese a ellos mismos. Al añadir a cada uno de los estados dicha transición, ya se llevaba a cabo el comportamiento deseado.

Por último, los últimos fallos que aparecieron y se solucionaron surgieron cuando se quiso crear un *FSM* sin hacer uso ni de la clase *Tags* ni de los documentos *XML*. Para ello se intentó seguir un ejemplo que aparecía para crear una *FSM* en [4], sin embargo, éste resultó ser bastante confuso principalmente por que usaba variables que tenían el mismo nombre que las que se emplearían después como por ejemplo *EventList*. Es por ello por que realizar esta tarea generó muchísimos errores y debido a esto se decidió que se añadiría un modelo de ejemplo más para que el usuario no tuviera que pasar por todo este trabajo, el cual puede generar muchos fallos. Para que fuera más fácil de entender se cambio el nombre de *EventList* por *ListofEvents* y se explicó el código que seria necesario crear para generar las distintas *FSM,* así como se añadió el método encargado de crear la *FSM,* ya que aunque al principio se cree un *fsm* esta no es la FSM sobre la que se realizará el método *UpdateFSM(),* por lo que antes se deberá añadir al *FSM_Manager* y más adelante al *FSM*. Además, se añadieron algunos métodos a este modelo, así como se quitaron otros que no eran necesarios.

7. Resultados obtenidos

Una vez se solventaron todos estos problemas ya se podía hacer uso de la API junto con el campo de pruebas sin que estos fallasen o dieran lugar a comportamientos raros en los enemigos. Para facilitar aún más la asimilación de los estados y que condiciones se iban cumpliendo se empleó la instrucción *Debug.Log* para así poder ir mostrándolo por consola. Debido a que, con estas demostraciones, a parte del correcto funcionamiento, también se quería mostrarle al usuario un ejemplo visual en el que pudiera ver el comportamiento de dicha *FSM* en un enemigo, al finalizar con la demostración de dicha *FSM* se realizaran observaciones y justificaciones para realizar dicho comportamiento con la *FSM* elegida, así como formas de poder mejorarla.

La primera FSM que se probó fue la FSM Determinista, la cual como se puede observar en la figura 39, está formada por 3 estados *SPAWN*, el cual puede llevar a cabo la acción *A_SPAWN* que consiste en pintar de amarillo al enemigo, *MOVE* el cual puede llevar a

cabo la acción A_*MOVE* que consiste en pintar al enemigo de verde así como invocar el método encargado de realizar el movimiento del mismo , y por último *ATTACK* el cual puede llevar a cabo la acción A_*ATTACK* que consiste en colorear de rojo al enemigo así como de invocar al método encargado de realizar un ataque al jugador. Dichos estados están conectados entre si mediante las transiciones *MOVING*, que conecta el estado *SPAWN* y *MOVE,* y *ATTACKING*, que conecta el estado *MOVE* y *ATTACK*. Estas transiciones están ligadas a los eventos *E_MOVING* y *E_ATTACK* las cuales para que se añadiesen en la lista de eventos y así se ejecutase la acción correspondiente deben de cumplir la condición de que el jugador esté a cierta distancia, ya que cuando el jugador está alejado el enemigo simplemente se mueve por la escena, pero cuando el jugador está a una distancia próxima, se entra en estado *ATTACK*.

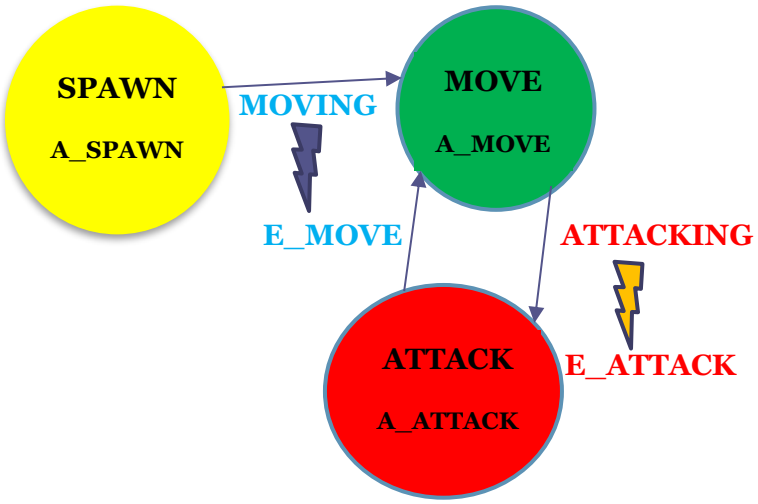


Figura 39: *FSM* Determinista del campo de pruebas

Se comprobó el comportamiento de dicha *FSM* para ver si realmente se comportaba como debía y reaccionaba a la presencia del jugador y esto ocurrió así, tal y como puede verse en la figura 40. Tras añadir los elementos que componían la *FSM* tanto al documento *XML* como a la clase *Tags* se comprobó el funcionamiento de este tipo de *FSM* el cual fue el esperado ya que inicialmente el enemigo se encontraba en estado *SPAWN*, y al empezar el jugador a moverse se cumplía la condición para pasar al estado *MOVE*. Dicha condición para pasar al estado *MOVE* era que ya se hubiera cargado el enemigo en pantalla, por tanto, al ocurrir esto se añadiría a la lista de eventos el evento *E_MOVE*, y se pasaría al estado *MOVE* ejecutando por tanto la acción A_*MOVE*, la cual llama al método *MoveEnemy()*.Mientras el jugador no se acercase al enemigo, éste continuaba moviéndose por la escena, hasta que estuviera a cierta distancia. Cuando se cumpliese dicha condición de proximidad, la cual era la siguiente *Vector3.Distance(this.transform.position, GameObject.Find("Player").transform.position) < 1.2* , en el que mediante el método *Distance()* se mide la distancia entre dos objetos, siendo estos objetos *this.transform.position*, es decir el objeto de la clase y por tanto el enemigo, el jugador, el cual mediante su *Tag* y el método *Find()* se podría encontrar y obtener su posición, se añadiría a la lista de eventos el evento *E_ATTACK*. Tras añadir este evento se pasaría al estado *ATTACK*, el cual realiza la acción A_ATTACK en la que se llama al

método ATTACK(), que avisa que se está ejecutando dicha acción, pinta al enemigo de rojo y llama al método *LoseFood()* de la clase *Player* para que éste realice la animación *PlayerHit* y pierda vida, en concreto 5.

Figura 40: Comportamiento de la *FSM Determinista*

Inicialmente el comportamiento de esta *FSM* no fue el esperado, pero esto fue debido a que como la condición de MOVE siempre se daba y estaba antes que la de ATTACK, se añadía el evento de MOVE a la lista de eventos y se ejecutaba la acción asociada a dicho estado, no dejando así que se pasase al estado ATTACK, aunque se cumpliese la condición. Es por ello por lo que tanto en la guía como en la memoria en el apartado de utilización de la *API* se recalca el hecho de comprobar primero los eventos que no se ejecuten de forma tan repetida frente a los que se ejecuten de forma continua.

Para asegurar que el comportamiento era el correcto se comprobó si, aunque se cumpliesen condiciones como la de atacar estando en el estado *SPAWN* se ejecutaban dichas acciones, acto que no debería hacer ya que dicho estado no es alcanzable tal y como se ha definido en el *XML*. Pese a que se cumplía dicho evento y se añadía a la lista de eventos, este no se llevaba a cabo, asegurando así un correcto funcionamiento de esta *FSM*.

Debido a que en este *FSM* de un estado solo se puede ir a otro, los comportamientos que se quieran realizar con este tipo de *FSM* no deberán de ser muy complejos ya que esto además hará ver al jugador que dicho comportamiento es muy monótono. Para solucionar este problema se pueden emplear las FSM que se ven a continuación.

Tras esto se pasó a comprobar el funcionamiento de la FSM Probabilista mediante el comportamiento que puede observarse en la figura 41.

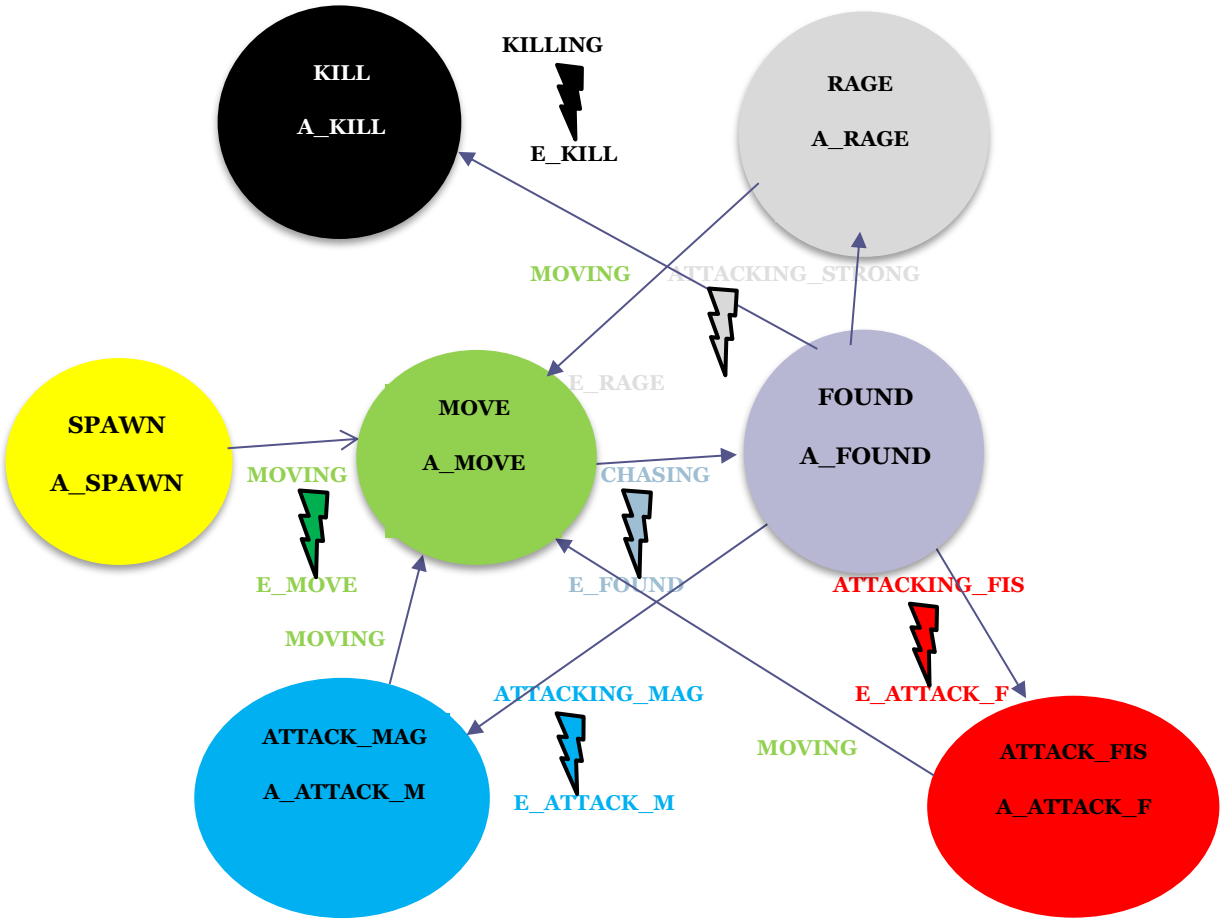


Figura 41: *FSM* Probabilística del campo de pruebas

El comportamiento de dicha *FSM* es similar al de la *FSM* determinista, pero añadiendo unos cuantos matices. Primero debido a que se trata de una *FSM* indeterminista, de un estado se podrá ir a varios con lo, ante cambios en el videojuego, se verán mas reacciones por parte del enemigo, y segundo al ser probabilística no siempre se realizan las acciones de un estado, aunque si se cumpla la condición para ir a éste. Tras añadir los elementos que componían la *FSM* tanto al documento *XML* como a la clase *Tags* se comprobó el funcionamiento de este tipo de *FSM* en el que inicialmente el enemigo se encontraba en estado *SPAWN*, y al empezar el jugador a moverse se cumplía la condición para pasar al estado *MOVE*. Dicha condición para pasar al estado *MOVE* era que ya se hubiera cargado el enemigo en pantalla, por tanto, al ocurrir esto se añade a la lista de eventos el evento *E_MOVE*, y se pasa al estado MOVE ejecutando por tanto la acción *A_MOVE*, la cual llama al método *MoveEnemy()*.Mientras el jugador no se acercase al enemigo, éste continuaba moviéndose por la escena, hasta que estuviera a

cierta distancia. Cuando se cumpliese dicha condición de proximidad, la cual se comprobaba de la misma forma que en la *FSM* Determinista, pero con otros valores, se añade a la lista de eventos el evento *E_FOUND* y por tanto se pasa al estado *FOUND* el cual realiza la acción *A_FOUND* que simplemente pinta al enemigo e inicializa ciertas variables que se comprobaran para poder pasar a los siguientes estados. En este momento dependiendo de si el jugador esta en un rango cercano o lejano se realizará un ataque de cerca o de lejos. Si el jugador está lejos se añadiría el evento *E_ATTACK_M* y por tanto se pasará al estado *ATTACK_MAG*, en el que se llevará a cabo la acción *A_ATTACK_M*, donde se llama al método *ATTACKMAG()* el cual notifica que se está realizando dicha acción, se pinta al enemigo de azul, se realiza la animación *EnemyAttack* del enemigo y por último se llama al método de la clase *Player LoseFood()* para que el jugador realice la animación *PlayerHit* y pierda comida, en concreto 3. Si por el contrario el jugador se encuentra cerca se añadiría el evento *E_ATTACK_F* y por tanto se pasará al estado *ATTACK_FIS*, en el que se llevará a cabo la acción *A_ATTACK_F*, donde se llama al método *ATTACKFIS()* el cual realiza lo mismo que ATTACKMAG() pero coloreando al enemigo de rojo y quitándole al jugador 5 de comida. Por último, si ya se habían realizado previamente un numero de ataques se añadiría a la lista de eventos el evento *E_RAGE* y por tanto se pasará al estado RAGE, en el que se llevará acabo la acción A_RAGE, donde se llama al método ATTACKRAGE() cual realiza lo mismo que ATTACFIS() pero coloreando al enemigo de negro y quitándole al jugador 10de comida. Por último, cuando ya se hayan realizado mas ataques y se esté en el estado RAGE, se añadirá el evento E_KILL y por tanto se pasará al estado KILL, en el que se llevará acabo la acción A_KILL, donde se llama al método KILL() el cual realiza el mismo que ATTACKRAGE() pero quitando 100 de comida y pintando de negro al enemigo. Siempre que se realice algún ataque se volverá al estado MOVE, donde se buscará al jugador para realizar el siguiente ataque. Tras definir el comportamiento de dicha *FSM* se comprobó si realmente se comportaba como debía y reaccionaba a la presencia del jugador y esto ocurrió así, tal y como puede verse en la figura 42.

Figura 42: Comportamiento de la *FSM Probabilistica*

Para comprobar el funcionamiento correcto de esta FSM lo primero que se hizo fue comprobar si todos los estados eran alcanzables al cumplirse las condiciones asociadas a las transiciones de dichos estados, por lo que probabilidad de todas las transiciones se puso al 100%. Con esto lo que se obtuvo era un comportamiento similar al de una FSM determinista, y se verificó que, efectivamente, todos los estados eran alcanzables. A continuación, se fueron dando probabilidades a las transiciones para comprobar si, aunque se cumpliesen las condiciones, a veces no se realizan y si estas realmente se veían afectadas por el número de probabilidad que se les daba, siendo las transiciones con números de probabilidades más bajas las que se realizaban menos veces.

A continuación, se siguió con la comprobación de la *FSM* Inercial y para ello puesto que esta *FSM* es idónes para solventar problemas relacionados con la oscilación entre los estados, se planteo un comportamiento de *FSM* en la que ocurriese este hecho, para aplicando esta *FSM*, ver si realmente se solventaba de forma satisfactoria este problema.

El comportamiento que se planteó realizar surgió a raíz de intentar crear el comportamiento anterior, ya que en un primer momento se pensó que se realizasen los ataques del enemigo como si fuesen una serie de golpes encadenados, sin embargo, al intentar esto lo que sucedió es que se realizaban los golpes muy seguidos, no dando la

posibilidad al jugador de escapar. El comportamiento de dicho *FSM* se puede observar en la Figura 43.

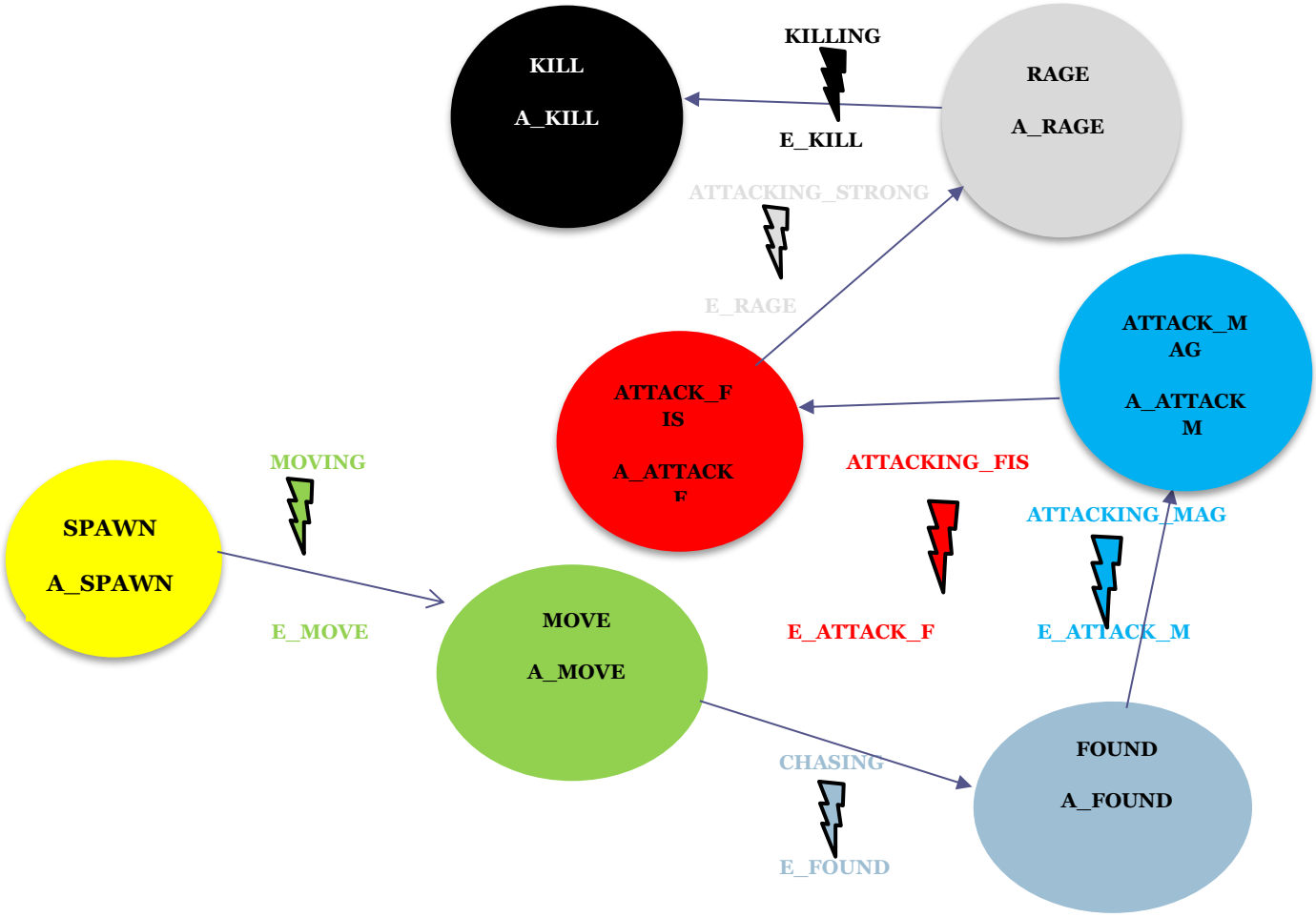


Figura 43: *FSM Inercial* del campo de pruebas

Debido a que los estados son los mismos que los del *FSM* anterior, se llevan a cabo las mismas acciones y lo único que cambia son los eventos que permiten que se pase a otro estado. Para que ocurriese este problema de oscilación entre estado lo que se hizo fue asociar a estos eventos la comprobación del valor de una variable que cambiaba en el estado anterior para que así, siempre que se compruebe si se puede ir al siguiente estado, se cumpliese dicha condición y se pasase al siguiente estado. Efectivamente al ocurrir esto, se pasaba de un estado al siguiente en cuanto le tocaba al enemigo realizar una acción y se hacía . Cuando se hizo uso de la *FSM* Inercial lo que sucedió es que se retrasaba un poco este paso al siguiente estado, teniendo como consecuencia que el estado en el que se encontraba la *FSM* actualmente se ejecutaba más veces, por lo que si se quisiese que una acción se realizase más veces se podría emplear esta FSM incrementándole el tiempo de espera, o si por el contrario ,como ocurre en este caso, se quiere simplemente que no hagan tan seguidas las acciones se podría simplemente asociar la acción al entrar al estado. Por último, para comprobar que el valor que se le daba a la espera realmente influenciaba el videojuego se fueron probando valores más y más altos para ver si realmente la espera era más larga y se comprobó que esto era realmente así.

Figura 44: Comportamiento de la *FSM Inercial*

Para continuar con las pruebas se siguió con la *FSM* basada en pilas cuyo comportamiento se puede observar en la figura 45.

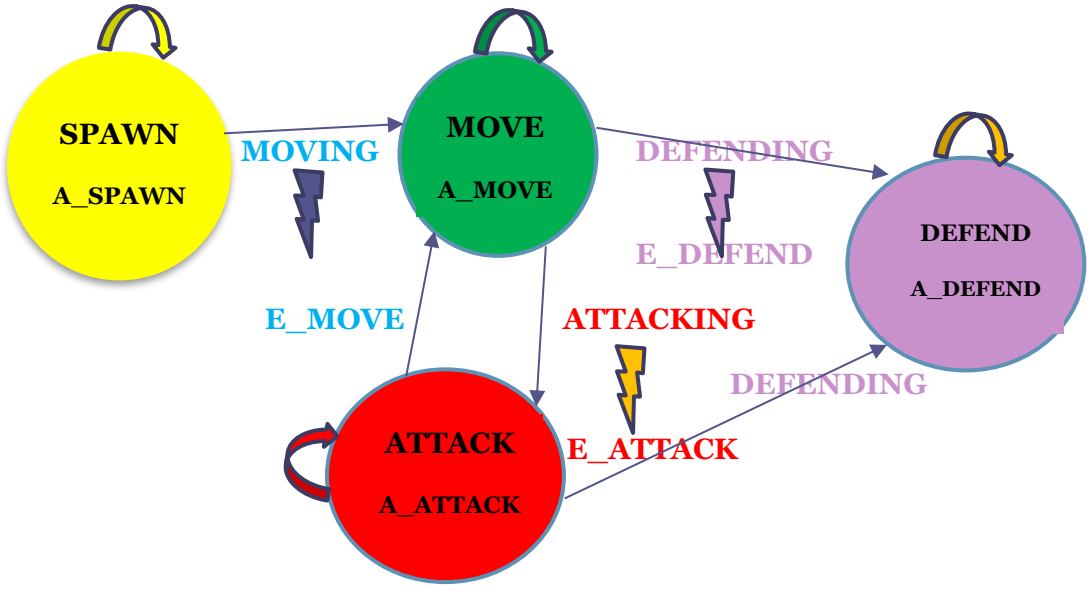


Figura 45: *FSM* Basada en Pilas del campo de pruebas

En esta *FSM* lo que se hizo fue añadir a la *FSM* determinista un estado más, el estado *DEFEND*. Este estado seria el más prioritario de dicho *FSM* por lo que todas las transiciones que vayan hacia dicho estado se tendrán que marcar como *STACKABLE* y no como *BASIC,* para que así se indique que son apilables y por tanto al salir del estado *DEFEND* se pueda volver al estado del cual se realizó la transición. En este *FSM* el enemigo lo que hace es moverse por el mapa y atacar cuando el jugador está cerca, ya que los estados que parten de la *FSM* determinista siguen el mismo comportamiento. Sin embargo, a las acciones de dichos estados se les hacía incrementar una añadiría el evento *E_DEFEND* a la lista de eventos y se pasaría a ejecutar la acción *A_DEFEND* la cual pondría dicha variable a 0 otra vez. Este comportamiento lo que quiere emular es que tras realizar ciertas actividades el enemigo se cansaba y descansaba estando en modo defensa.

Al intentar añadir este comportamiento al enemigo ocurrieron bastantes problemas ya que, aunque se solventaron los problemas para que se pudiera crear la FSM con sus componentes, el comportamiento no era el esperado. Tal y como se explicaba en *[4]* las transiciones que iban hacia un estado con mayor prioridad se debían de etiquetar en lugar de *BASIC* como *STACKABLE*, pero si se etiquetaban como *STACKABLE* no llegaban a ejecutarse nunca, a pesar de que se añadiese dicho evento a la lista de eventos. Tras una revisión más exhaustiva del código se encontró el problema tal y como se ha comentado en el apartado anterior de la memoria, el cual era que todos estados tenían que tener una transición que fuera a ellos mismos. Tras añadir estas nuevas transiciones el comportamiento fue el esperado ya que lo que mas interesaba ver en este tipo de *FSM* es que siempre que se cumpliese la condición para el estado mas prioritario, se fuese a ese estado, se llevase a cabo la acción que tocase, y cuando se finalizase se volviese al estado el cual se apiló, continuando con el funcionamiento correcto. Este comportamiento se puede visualizar en la figura 46.

Figura 46: Comportamiento de la *FSM* basada en Pilas

Por último, se realizo la ultima prueba con la *FSM* basada en estados concurrentes en la que la idea es que varios estados, los que el usuario desee, se puedan ejecutar a la vez de forma concurrente. Para validar esta *FSM* lo que se quiso hacer fue intentar separar la *FSM* Determinista vista anteriormente, en partes, teniendo por un lado las animaciones que el enemigo tenia que hacer, y por otro lado las acciones que tenia que llevar a cabo. El comportamiento que se implantó se puede visualizar en la figura 47.

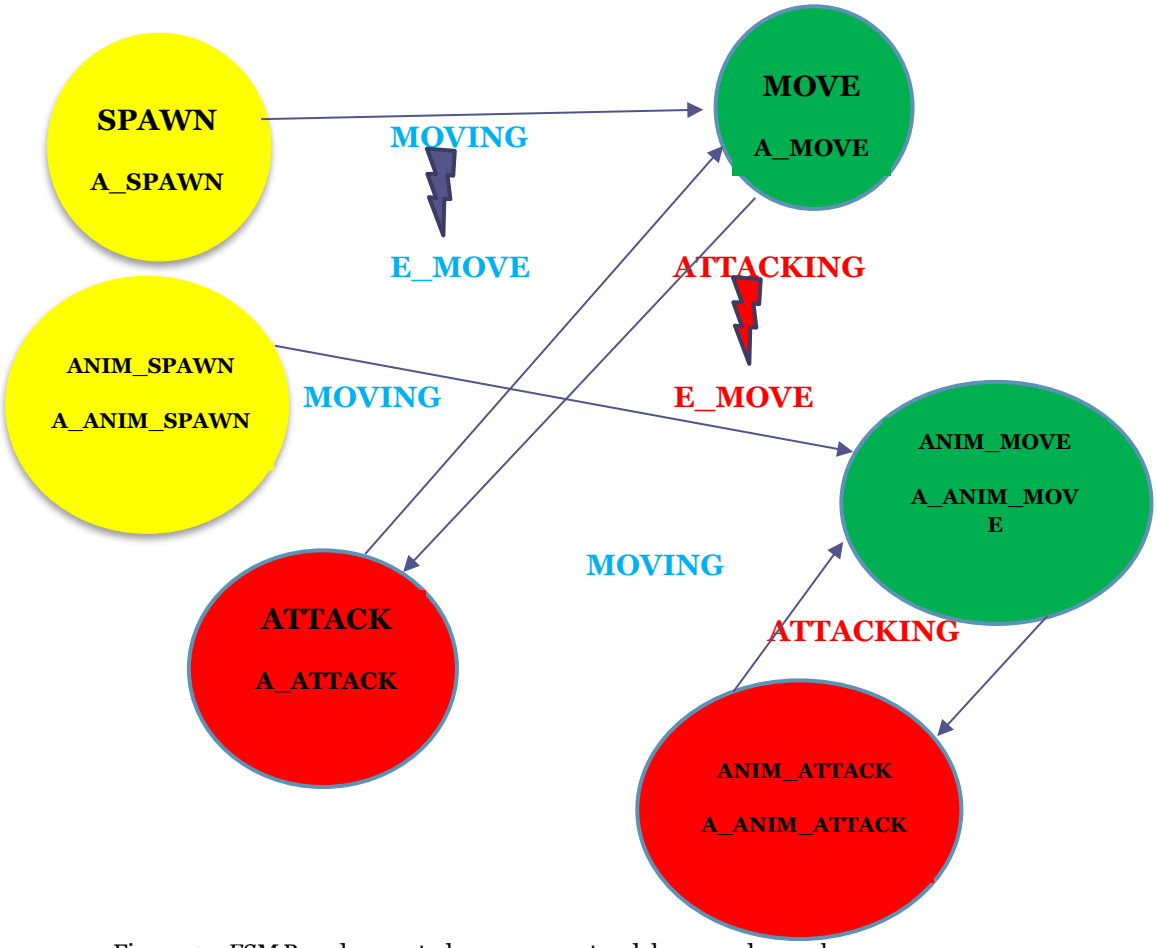


Figura 47: *FSM* Basada en estados concurrentes del campo de pruebas

Esta FSM presenta los mismos estados que la FSM determinista, pero se le añaden 3 estados más los cuales son; ANIM_SPAWN, el cual lleva a cabo la acción A_ANIM_SPAWN, donde se colorea al enemigo de amarillo, el estado ANIM_MOVE, la cual realiza la acción A_ANIM_SPAWN, donde se colorea de verde al enemigo y por ultimo ANIM_ATTACK, la cual realiza la acción A_ANIM_ATTACK, donde se realiza la animación del enemigo *enemyAttack* y se colorea al enemigo de rojo. Estos nuevos estados se conectan mediante las transiciones que ya estaban creadas, *MOVING* y *ATTACKING*, razón por la cual el evento que permite pasar de un estado al otro es E_MOVE y E_ATTACK respectivamente. En la figura 48 se puede ver dicho comportamiento en el videojuego.

Figura 48: Comportamiento de la *FSM* basada en estadas concurrentes

Pese a que el comportamiento al principio no fue el esperado, tras solventar el problema que se ha visto con esta *FSM* en el apartado de Errores y cambios realizados, se llevaban a cabo las dos acciones a la vez dando como resultado la ejecución de las animaciones y las acciones a la vez, confinando así el correcto funcionamiento de dicha *FSM*

8. Publicación y valoración del público de la API

Tras acabar con las modificaciones y validación de la *API* se decidió subir el resultado final obtenido a la *asset store* de *Unity* y, a partir de tanto las valoraciones obtenidas como las descargar realizadas de los usuarios, realizar una valoración del público para así ver si se debería modificar alguna parte de la estructura de la *API* o modificar las

guías ofrecidas para así clarificar y ayudar en la medida de lo posible al usuario en la utilización de la *API*.

Para no dejar solo la opinión y valoración del trabajo realizado a los usuarios que realizasen dichas descargas y diesen su opinión, se decidió realizar un estudio sobre como emplearían la *API* diversos usuarios, cada uno con un nivel diferente en el campo de la informática. Para la realización de este estudio participaron 5 personas cuyo nivel en el campo de la informática son:

- Usuario 1 -> Nulo, nunca ha dado ninguna materia relacionada con la informática y no tiene mucha idea
- Usuario 2 -> Nulo, entiende ciertos conceptos de la informática y se interesa por ciertos temas del mundo de la informática
- Usuario 3 -> Medio, cursó un grado medio de informática y entiende como hacer uso de estas herramientas.
- Usuario 4 -> Medio, cursó ciertas asignaturas del grado de ingeniería informática.
- Usuario 5 -> Alto, actualmente está cursando su último año en el grado de ingeniería informática.

Dado que los niveles en cuanto a conocimiento no solo del campo de la informática si no de Unity o de utilización del ordenador eran tan diferentes, lo que se decidió hacer es, a partir del campo de pruebas empleado para la validación de la *API,* descargarse de la *asset store* la *API* e intentar integrarla en el videojuego que se proporcionaba simplemente leyendo las guías para así ver si estaba explicado de forma correcta o no.

La primera observación que se hizo fue que mientras que los usuarios de nivel medio y alto comprendían perfectamente el objetivo de la herramienta, que era una *FSM* y para que la emplearían, los usuarios de mas bajo nivel no comprendían que era ni que les iba a proporcionar dicha herramienta, razón por la cual se decidió que en la "*Guía de elección de FSM*" además de los ejemplos de comportamientos y *FSM* creadas para usarlas como referencias, se añadiría el punto de la memoria en el cual se explicaba lo básico de una *FSM*.

Además de esto, otra observación que se realizó es que a la hora de hacer uso de la *API* cuando los usuarios empleaban las herramientas como los documentos *XML*, el *FSM_Parser* y la clase *Tags*, los usuarios de más nivel rápidamente entendieron que lo que estaban evitándose era el tener que ir creando las transiciones, los estados etc… y que simplemente rellenando el campo **<State></State>** por ejemplo ya tenían un estado creado y con la clase *Tags* se le daría un valor más adelante. Este concepto de *parser* que se realiza mediante el uso de estas herramientas solo lo comprendieron una vez tuvieron que hacer uso de la *API* sin las herramientas, momento en el que entendieron que rellenar estos documentos les evitaba el tener que crearlo mano a mano y todos los posibles errores que se generaban. Debido a esto se decidió que en la "*Guía de funcionamiento de la API*" se añadiría un apartado mas en el que se explicaría que ventajas presenta usar las herramientas que ofrece la *API* y por qué emplearlas.

A parte de estas observaciones, que se sacaron a partir de observar como empleaban el *asset* cada uno de los usuarios, también se les pidió al finalizar el tiempo de uso de *API* que hicieran una valoración general de la *API*, el nivel de complejidad que esta les había

supuesto y que dijeran algo positivo y negativo de la misma. La mayoría de ellos coincidieron en que el nivel de complejidad era medio, ya que ciertos conceptos, así como conocimientos debían de tenerse para poder hacer un uso correcto de la *API*. Por otro lado, algo positivo que todos valoraron fue el añadir la opción de poder crear las *FSM* mediante documentos *XML* ya que por el contrario algo negativo que se valoró de la *API* fue tener que crear a mano todos y cada uno de los componentes de la *FSM* ya que, si se producía un error, depurarlo era una tarea muy costosa.

9. Trabajos futuros

Una vez se finalizó con la realización de este trabajo de fin de grado, se consiguió validar la *API* basada en máquinas de estado finitos de la cual se partió. Durante la realización de esta tarea se tuvieron diversas ideas que se podrían añadir a la *API* para en un futuro para facilitar aún más la creación de las *FSM*.

La primera idea que se tuvo fue añadir a la *API* un compilador para detectar errores cuando se están escribiendo las variables *Strings* que representan el identificador de la *FSM* o el nombre de algún archivo o componente de la *FSM*. Esto se pensó debido a que mientras que cuando se inicializaban las variables de la clase *Tags* y se accedía a dicha variable, pero se escribía mal, se nos indicaba que no existía dicha variable, cuando se escribía el nombre de por ejemplo algún *String* como el nombre de la *FSM* donde se encuentra el documento *XML* de la *FSM* a emplear y se escribía de forma errónea, o no coincidía o no se había creado se producía un *NullPointerException*, un error tan amplio y con tantas posibles fuentes que la labor de encontrar que había originado dicho error era muy tediosa.

La segunda idea que se tuvo fue, al igual que *Unity* presenta una herramienta que se ha visto anteriormente para gestionar las animaciones como es el *Animator Controller,* el cual tiene un funcionamiento idéntico al de las *FSM,* se pensó en realizar una herramienta similar para poder crear todos los componentes de la *FSM,* así como las conexiones de esta, de una forma gráfica, de modo que se evite toda la parte de rellenar los documentos *XML*.

Además de esto también se podría aumentar el número de máquinas de estado finitos presentes en la *API* puesto que al final esta acción no se llevó a cabo.

Por otro lado al tener validada esta *API* un nuevo abanico de posibilidades se abre para realizar futuros trabajos o haciendo uso de esta *API* o sobre esta *API*. Tal y como se vio en el apartado de Estado del arte y Critica al estado el arte una forma muy empleada para crear un ente que presente inteligencia artificial es mediante las redes neuronales, las cuales presentan muchas ventajas, pero es cierto que se les debe de proporcionar gran cantidad de información y un aprendizaje. Teniendo una *API* de inteligencia artificial que se sabe que funciona y es válida, se podría emplear ésta como fuente de aprendizaje para dicha red neuronal y una vez ya haya aprendido de esta ya se tendría una red neuronal altamente capaz. También se podrían añadir funcionalidades extra a esta *API,* tales como podrían ser añadir la funcionalidad de *matching learning* o de

reconocimiento de voz. Estas funcionalidades, como se ha visto anteriormente, están creadas ya en *APIs* por lo que añadirlas a la *API* que se ha creado, o crearlas desde 0 con ciertas modificaciones, no seria muy complejo, ya que se tiene una base de la cual se puede aprender o consultar en caso de que surjan problemas.

10. Conclusión

Tal y como se ha visto en el apartado de planificación, al inicio del trabajo de fin de grado se realizó un reparto de tareas y se estimó que tiempo iban a durar cada una de ellas. Esto se vio totalmente modificado conforme se acercaba la fecha de entrega debido a diversos motivos. El primero de ellos es que, a pesar de que se creía que tanto la comprensión como la realización de una prueba de la *API* iba a ser sencillo esto no fue para nada así. Entender como se empleaban los eventos para que estos te dieran la lista de acciones a realizar costó muchísimo, principalmente por que el *callback* no se ejecutaba como tocaba. Además de esto el comprender la funcionalidad de cada clase, junto con la forma de hacer uso de ellas tanto si se empleaban los documentos XML como no llevo bastante tiempo. Esto junto con que al principio se quería probar el *API* en el banco de pruebas que se tenía inicialmente y este presentaba muchos errores me hizo perder mucho tiempo. Al final decidí centrarme directamente en mi campo de prueba e intentar crear las maquinas de estado de cada uno de los tipos para probarlas y verificar así el correcto funcionamiento de la *API*. Al principio también se empleó mucho del tiempo en leer mucha cantidad de información y varias veces la memoria del *TFM* del cual se partió para poder entender todas las acciones que se hicieron en su momento, y esto costó bastante, principalmente porque todo el tema de inteligencia artificial prácticamente no se había visto en la carrera y es en el máster donde más se profundiza. También en la realización del campo de pruebas, ya que su realización fue más lenta de lo normal puesto que todo era muy nuevo y ,emplear una herramienta como la de *Unity*, supuso un gran esfuerzo y tiempo. Sin embargo, donde más se empleó tiempo fue en hacer que la *API* funcionase nuevamente como lo hacía, puesto que al intentar hacer uso de ella las primeras veces, todo el apartado de los documentos *XML* producía muchos errores, los cuales fueron muy tediosos de localizar. Además de ello una vez solventado estos errores se comprobó que el comportamiento de algunas no era el indicado y este motivo también se tuvo que localizar. Por último, se realizaron muchos cambios conforme se iban aprendiendo cosas tanto de *Unity* como de las *FSM,* teniendo como resultado que durante la realización del *TFG* el campo de pruebas cambiase continuamente.

Además de esto una vez se finalizó con la creación de la API, cuando se mostró a los usuarios seleccionados para obtener un *feedback* y así saber las opiniones de los usuarios, estos me hicieron darme cuenta de que se podría facilitar mucho mas el uso de la API añadiendo unas mejores explicaciones o añadiendo comentarios o ejemplos en ciertos puntos, por lo que algunos puntos se vieron nuevamente modificados. Una

tabla con el diagrama de Gantt, el cual contiene la duración que se planificó de las actividades frente a la duración real de éstas se verá a continuación:

Etapas de Desarrollo	Semana 1 (9-15)	Semana 2 (16-22)	Semana 3 (23-29)	Semana 4 (30-5)	Semana 5 (6-12)	Semana 6 (13-19)	Semana 7 (20-26)	Semana 8 (27-30)	Semana 9 (3-7)
Lectura del TFM									
Bibliografía									
Herramienta Unity									
Campo pruebas TFM									
Redactar primeros puntos y campo de pruebas									
Campo de pruebas y cambios de la API									
API y redactar									
Subir API y conclusión									
Opinión de los usuarios									
Corrección de la memoria									

Figura 49: Diagrama de *Gantt* final

En este nuevo diagrama de Gantt, tal y como puede observar, se representa la duración de todas las actividades que se vio en el diagrama del apartado de planificación, las cuales además de su duración representada en color azul, se añadió el colo negro para representar aquellas actividades cuya realización acabó antes de tiempo. Además de esto algunas actividades que en un primer momento no se tuvieron en cuenta, como el enseñar la *API* y la valoración de los usuarios, o la corrección de la memoria se añadieron a este nuevo diagrama.

Tal y como se puede observar la planificación inicial fue muy idílica ya que se supuso que todo iría bien, que no surgirían problemas y que rápidamente en simplemente una semana cada una de las actividades estaría completada. Esto no ocurrió así, tal y como se ha visto en los apartados anteriores, ya que surgieron bastantes, algunas tareas se realizaron no solo una vez, si no varias veces a lo largo de la realización del TFG tales como la lectura del TFM del cual se partía o la realización del campo de pruebas, puesto que este conforme se solventaban problemas, se realizaban modificaciones.

Por ultimo y pese a que haya sido bastante complicado y haya requerido mucho tiempo a emplear para la realización del *TFG* estoy bastante contento con el resultado y con haber podido acabar la carrera haciendo algo tan útil como subir una *API* a una herramienta tan reconocida como es *Unity*. Además, al finalizar este proyecto me di cuenta de que había aprendido muchísimas cosas no solo de programación sino también de depuración de código, así como a estructurar el código y sobre todo a cómo explicarle al usuario medio como debe emplear una herramienta y a proporcionarle atajos para que lo haga de una forma más cómoda al igual que saber que modificar de la herramienta proporcionada atendiendo a sus propuestas o reacciones o formas al hacer uso de ésta.

Bibliografía

[1] *RUSSELL, Stuart J. & NORVIG, Peter (2014) Artificial Intelligence, a modern approach.*

[2] *Schwab, Bryan (2004) AI Game Engine Programming.*

[3] *Buckland,Mat (2005) Programming Game AI by Example.*

[4] *José Alapont Luján (2014) Memoria TFM API de gestión de Inteligencia Artificial basada en las Maquinas de Estados Finitos en C #*

[5] *Bonifacio Martin del Brio & Alfredo Sanz Molina (2001), Redes Neuronales y Sistemas Borrosos.*

[6] *Reisig,Wolfgang (1985) Petri Nets – An Introduction.*

[7] *Reisig,Wolfgang (2013) Understanding Petri Nets.*

[8] Funcionamiento del algoritmo *Min-Max*: *Introduction to Artificial Intelligence with Java*

Tabla de figuras

Diccionario de términos

En este apartado se definirán algunos términos que se emplearán a lo largo del TFG.

API: Este término son las siglas de *application programming interface* (interfaz de programación de aplicaciones) y se trata de un conjunto de subrutinas, funciones y procedimientos o métodos si se trata de un lenguaje orientado a objetos, que ofrece cierta biblioteca para ser utilizado por otro software como una capa de abstracción. Generalmente son usadas en las bibliotecas de programación.

Motor gráfico o motor de videojuego: Este término hace referencia a una serie de rutinas de programación que permiten el diseño, creación y representación de un videojuego. Existen tanto motores de juegos para consolas como para sistemas operativos. La funcionalidad básica de un motor es proveer al videojuego de un motor de renderizado para los gráficos 2D y 3D, de un motor de físicas o detección de colisiones, la posibilidad de integrar sonidos y de animar, así como poder hacer *scripting*, emplear inteligencia artificial y hacer uso de redes, *streaming*, administración de memoria y un escenario gráfico.

NPCs: Este término se emplea en videojuegos para referirse a aquellas identidades que no están bajo el control directo de los jugadores. Es frecuente que este termino se reserve para aquellos personajes que son o bien neutrales con el jugador o bien aliados. El comportamiento de dichas identidades viene prescrito y es automático, siendo activado por acciones o diálogos del jugador.

Bosses: Este término se emplea en videojuegos para referirse a aquellos enemigos que, por sus mecánicas, mayor vida o fuerza, son más difíciles que los enemigos comunes que aparecen por el videojuego, suponiendo un reto extra para el jugador. Normalmente el eliminar estos enemigos es crucial para poder avanzar en el videojuego, ya que muchas veces estos enemigos no dejan al jugador pasar al siguiente nivel o zona del videojuego, o proporcionan un objeto que más adelante se necesitará.

Asset y Asset Store: Un *asset* es un elemento que será introducido en un videojuego. Estos elementos que se pueden incluir son muy variados, desde modelos en 3D, *prefabs*, texturas, materiales, animaciones hasta scripts, sonidos o elementos específicos de cada motor. En el caso concreto de Unity se tiene una opción llamada *Asset Store*, en la cual se puede ver todo tipo de *assets* que la gente sube, los cuales pueden ser o no de pago, y se puede descargar para utilizar en nuestro videojuego.

Script: Un *script,* también llamado archivo de procesamiento por lotes, es un programa usualmente simple que por lo regular se almacena en un archivo de texto plano y realizan diversas tareas tales como combinar componentes, interactuar con el sistema operativo o interactuar con el usuario. En el caso concreto de *Unity* estos scripts suelen contener el acceso a variables que pueden o no modificarse desde la ventana del Inspector, el comportamiento que se quiera que tenga el objeto al que se asocie dicho *spript*, así como contener la lógica para pasar de una animación a otra.

Sprite: Un *sprite* en el mundo de los videojuegos se suele emplear para referirse a aquellas imágenes unidas en un mismo archivo, unas al lado de las otras y que representan a un mismo personaje u objeto en distintas posiciones. A diferencia de un archivo *GIF* animado, todas las imágenes son visibles al mismo tiempo y usualmente están en un formato de imagen mas ligero como por ejemplo *PNG*. Este tipo de imágenes agrupadas se emplean sobre todo para la creación de animaciones que representen comportamientos de los personajes.

Prefabs: Se trata de un tipo de *asset* en el cual se almacena un objeto *GameObject* con componentes que lo conforman, así como las propiedades que el creador del videojuego quiere que tenga. El *prefab* actúa como una plantilla a partir de la cual se pueden crear nuevas instancias del objeto en la escena. Si el creador modifica algo de este *prefab* esto se reflejará en todas las instancias producidas por él. Esto es muy útil cuando, por ejemplo, ya se ha hecho un enemigo y ya tiene todo lo que se necesita; una animación, tiene un script que gestione su comportamiento, etc…., como ene l videojuego aparecerán más de estos enemigos es mucho más cómodo tener uno ya creado y solo tener que ponerlo donde se quiera a tener que ir creándolo cada vez que sea necesario.

Parser: Este término hace referencia a un módulo, biblioteca o programa el cual se encarga de transforma un archivo de texto en una representación interna. En el caso de este proyecto, el *parseado* se realiza mediante la utilización de documentos XML, por tanto, lo que se hace es transformar el texto que se ponga en dichos documentos, en variables de las que se pueda hacer uso.

Anexo

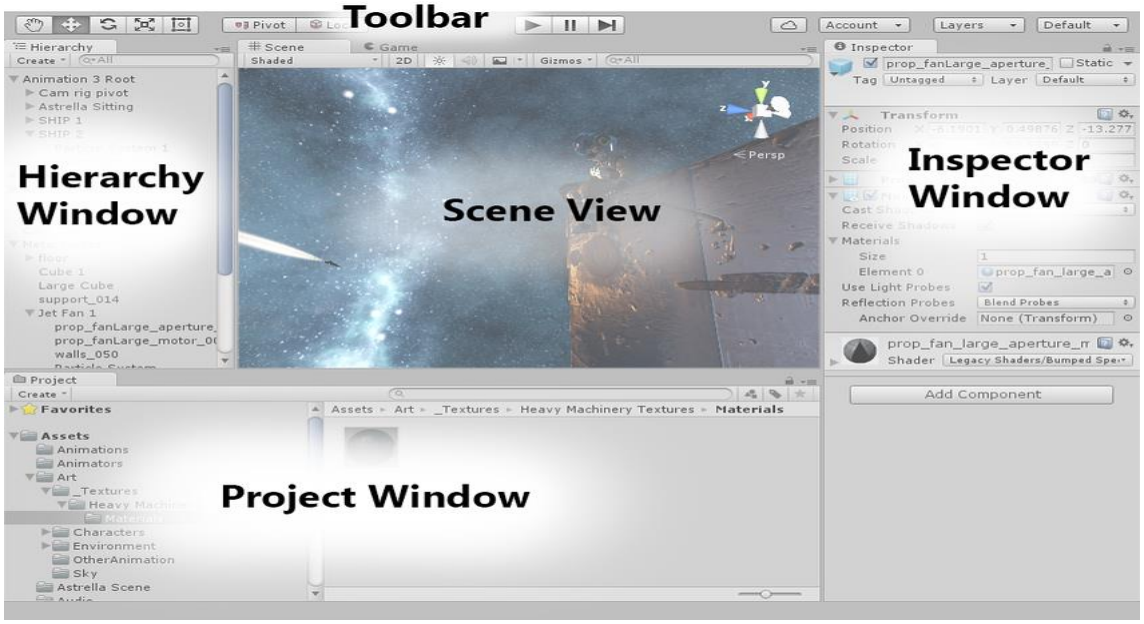
Introducción al motor de videojuegos Unity

En este apartado se pasará a hacer una pequeña explicación de las diferentes ventanas que componen *Unity* para la realización del videojuego, donde se encuentran y como se deberían de utilizar, para que así al ser mencionadas más adelante el lector pueda tanto ubicarse como entender lo que se está haciendo.

Unity está compuesto por diversas ventanas, las cuales agrupan elementos diferentes que forman parte del videojuego y que tienen diversas funciones.

Pese a que dichas ventanas pueden moverse y por tanto cambiar de posición, cuando se abre *Unity* se observan las siguientes ventanas que lo componen:

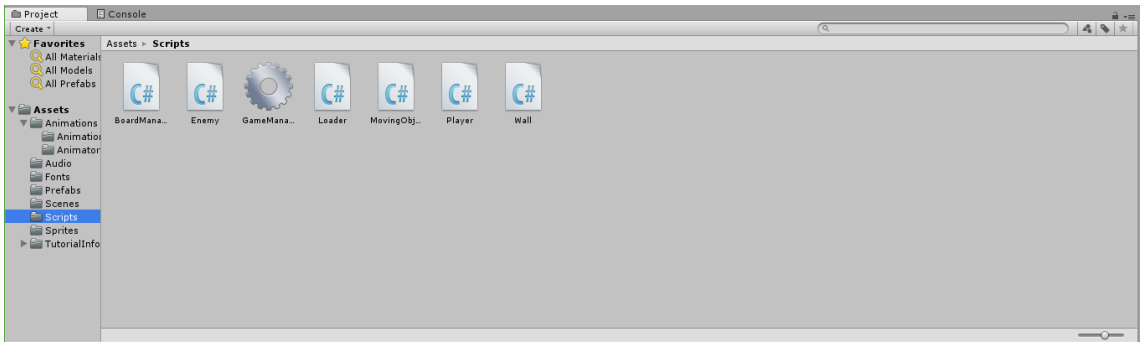
- Ventana de proyecto *(Project)*.
- La vista de escena *(Scene View)*.
- La ventana de jerarquía (Hierarchy).
- La ventana del inspector *(Insperctor)*.
- La barra de herramientas (Toolbar).



Interfaz básica de Unity

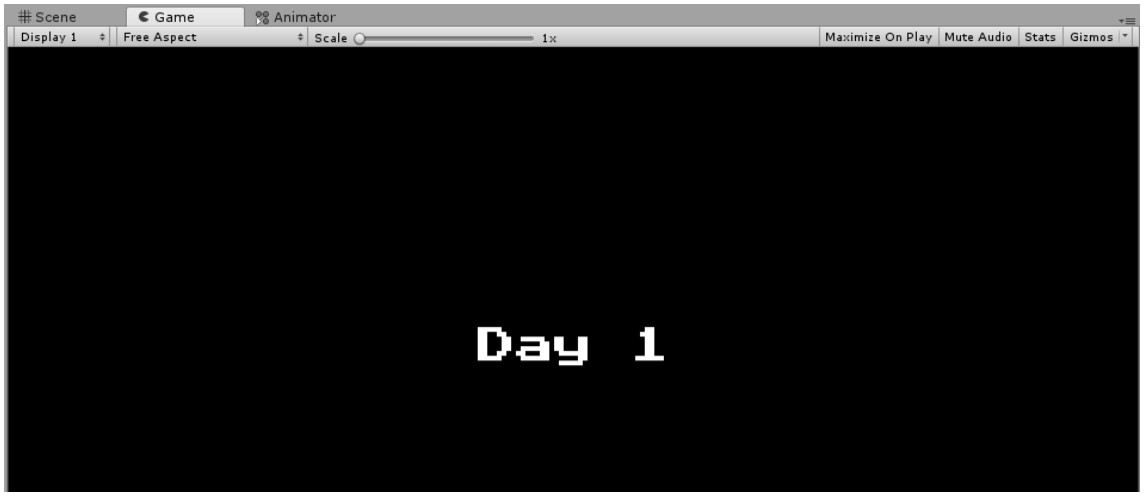
En la **ventana de proyecto** se encuentran todas las *assets* que formen parte del videojuego, pudiendo crear el número de carpetas que sean necesarias, así como clasificarlas de la forma que se quiera o necesite. Además de tener una sección en la que el desarrollador pueda guardar los *assets* que con más frecuencia emplee como favoritos para así tener un rápido acceso a ellos, también se puede ver arriba una “ruta

de navegación” en la que se puede ver en que carpeta se está viendo actualmente y de que carpeta se viene. Por último, en esta ventana se encuentra una barra para poder realizar búsquedas por el nombre del *asset* que el desarrollador esté buscando, evitando así que se tenga que ir carpeta por carpeta buscando si se sabe lo que se está buscando por su nombre.



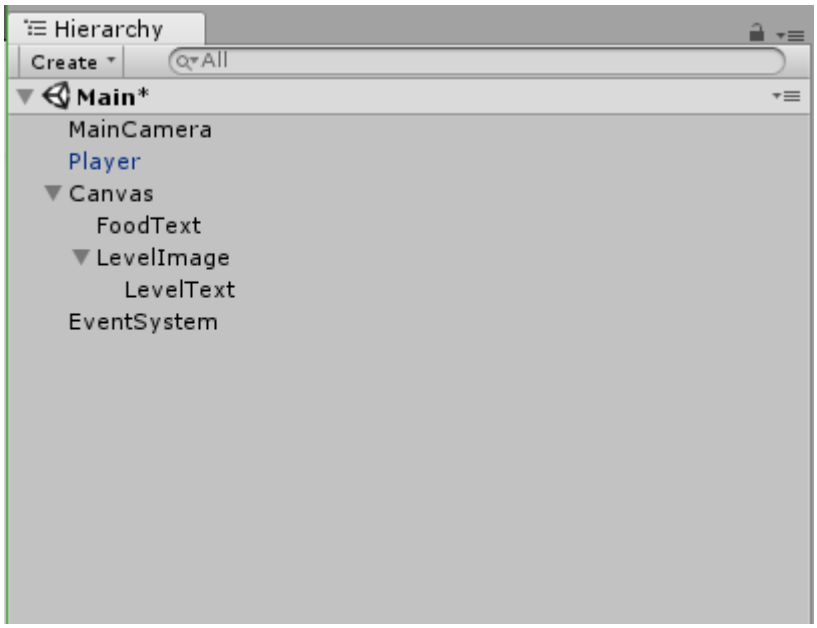
Ventana de proyecto de Unity

En la **ventana de escena** se encuentra el lugar donde el desarrollador del videojuego pueda ver qué forma va tomando el videojuego. *Unity* proporciona diferentes perspectivas ya sea en *2D* o *3D*. En esta ventana se encuentra una división entre la vista de escena y la vista de juego. Mientras que, en la vista de escena, el creador puede hacer modificaciones como por ejemplo donde situar enemigos o al jugador, así como rotarlos o cambiar su tamaño, en la vista de juego solo se ve el resultado que verá el jugador. También es posible añadir a estas dos visiones una tercera llamada *Animator*, la cual sirve principalmente para gestionar las animaciones de los diferentes personajes, enemigos o *NPCs*. Dichas animaciones se gestionan como máquinas de estados en donde los estados son las propias animaciones y las transiciones son las condiciones que cumplir para pasar de una animación a otra



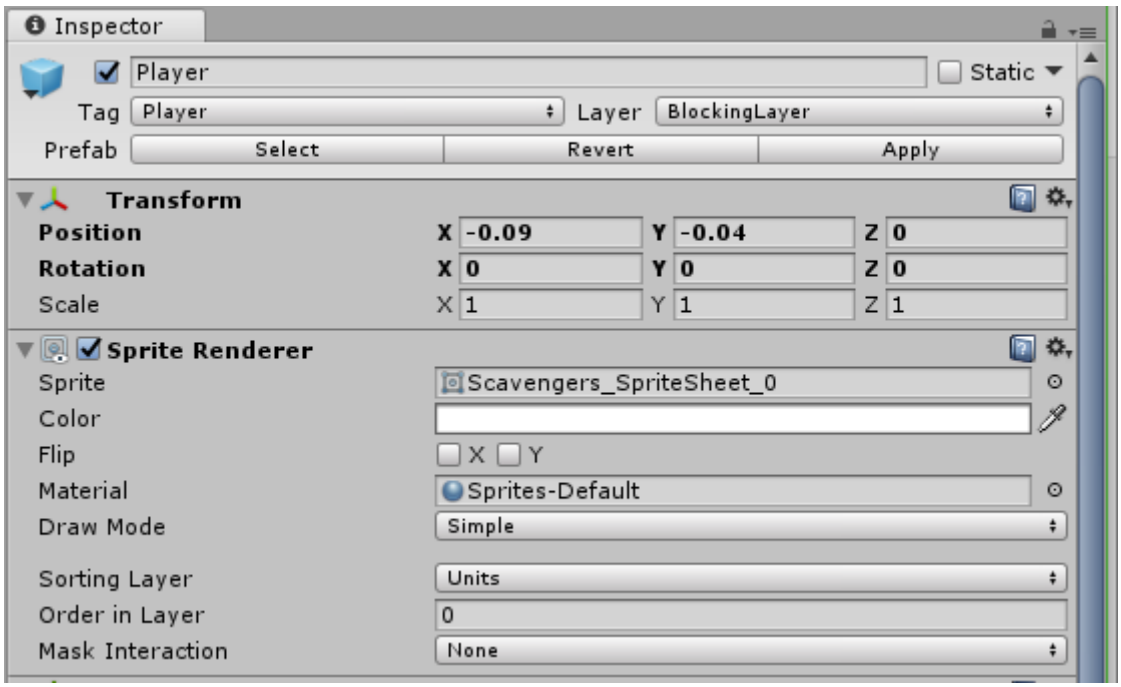
Ventana de escena de Unity

En la **ventana de jerarquía** se encuentran todos los objetos que forman parte de la escena. En la jerarquía el desarrollador del videojuego puede ver fácilmente como están estructurados ciertos objetos, si un objeto es hijo o padre de otro objeto.



Ventana de jerarquía de Unity

En la **ventana del inspector** se le permite al desarrollador tanto visualizar como editar todas las propiedades del objeto que haya seleccionado ya sea la posición, tamaño, materiales, los *scripts* que lo componen, que *prefabs* tiene etc.



Ventana del inspector de Unity

En la **barra de herramientas** es donde el desarrollador encuentra acceso a las características más esenciales para trabajar. A la izquierda del todo se encuentran una

serie de herramientas útiles para poder manipular la vista de escena y los objetos que la componen, seguidamente se tiene otro par de herramientas, situadas en el centro, las cuales son 3 botones (reproducción, pausa y pasos). Estos botones serán de gran ayuda para el desarrollador puesto que permiten empezar a reproducir la escena para ver cómo se vería el videojuego, pausarlo para poder hacer modificaciones y seguir con la ejecución o bien adelantarla, y tras haber realizado los cambios si se sale del modo ejecución, los cambios realizados no se harán, volviendo todo a como estaba antes de la ejecución, lo cual es bastante útil para hacer cambios y ver qué ocurriría. Por último, a la derecha del todo se encuentran botones relacionadas con la cuenta que se tenga en *Unity* y un botón muy útil llamado *Layers* el cual sirve para poder por ejemplo clasificar a ciertos objetos y así poder establecer diferencias entre ellos.

Para finalizar aclarar que esta barra será la única que el desarrollador no podrá modificar o cambiar de posición puesto que es fija y por tanto no se puede reajustar.

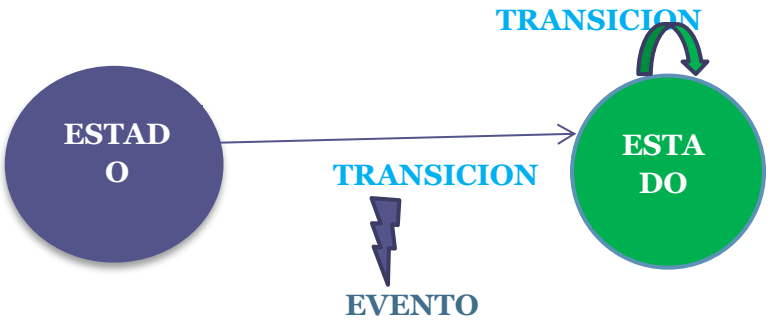
Barra de herramientas de Unity

Guía de elección de FSM

En esta guía de elección de *FSM* se le proporcionará una ayuda a la hora de elegir una de las *FSM* que están implementadas en la *API GAI(game artificial inteligence)*, para que así escoja la FSM que más le sea útil y que vaya acorde con sus necesidades. Para ello se explicarán las cualidades que proporciona cada *FSM,* así como comportamientos que se podrían implantar con dichas *FSM* y ejemplos prácticos con *FSM* creadas con sus estados, transiciones y eventos para que así tenga tanto ejemplos como ideas para crear sus *FSM*.

1.¿Qué es una FSM?

Para empezar una *FSM* o máquina de estados finitos se puede definir como un modelo de comportamiento en un sistema compuesto por entradas y salidas, en donde las salidas dependen no sólo de las señales de entradas actuales si no de las anteriores también. Es por ello por lo que una *FSM* estará formada por los estados que la compongan, las transiciones que conectan dichos estados unos con otros y por último se tendrán las acciones que se lleven a cabo.
Para empezar, se puede definir un estado, como un conjunto particular de instrucciones, las cuales serán ejecutadas en respuesta a la entrada de la máquina. Por otro lado, las transiciones serán lo que permita pasar de un estado de la maquina a otro, y normalmente están ligadas a condiciones o eventos que deben de cumplirse para pasar a otro estado. Para finalizar, cuando se habla de las acciones de una máquina de estados se tiene que tener en cuenta que dichas acciones pueden estar asociadas a una transición, y por ello solo se ejecutaran al pasar de un estado a otro, o también pueden estar asociadas a un estado, pudiendo realizar acciones al entrar o salir del estado, con lo cual solo se ejecutarían dichas acciones una vez, o bien se puede ejecutar dicha acción mientras se esté en el estado de la que forme parte y no se cumpla ningún evento que haga abandonar el estado actual.
Por último, las máquinas de estado finitos presentan un estado inicial o punto de inicio, el cual indica cual debe ser el estado del que se tiene que partir al comenzar con la ejecución de la máquina de estados.
A continuación, se puede observar el aspecto de una *FSM* en la que se tienen 2 estados y 2 transiciones, teniendo una de ellas un evento ligado a dicha transición, lo cual hace que, a no ser que se cumpla dicho evento, no se pueda pasar del estado azul al estado verde.



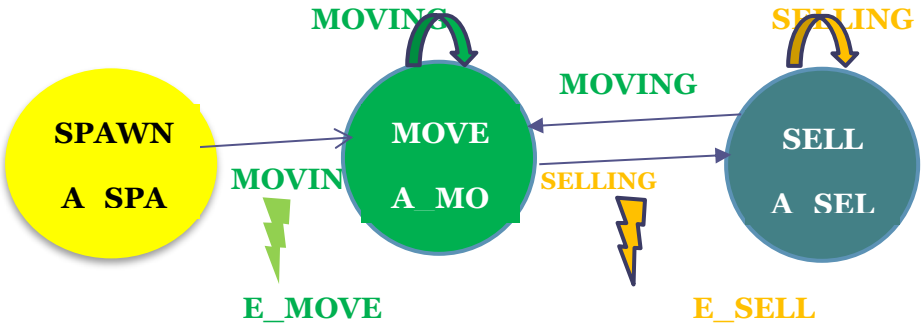
2. FSM Determinista

El primer tipo de *FSM* que se puede encontrar en la *API* es el determinista. Este *FSM* es el más básico de todos y la base de los demás *FSM,* razón por la cual no será apropiado cuando se desee implantar un comportamiento muy complejo o muy sofisticado. Este

tipo de FSM por otro lado será perfecto cuando se quiera controlar un comportamiento general, un comportamiento básico o muy simple o bien un comportamiento con pocos estados.

Generalmente este tipo de FSM se emplea en *NPCs* que salen de fondo en un videojuego, como por ejemplo en animales que simplemente tengan un comportamiento asustadizo frente al jugador huyendo de él cuando esté cerca y cuando no esté busquen comida. También se suele emplear en *NPCs* que están en ciudades o aldeas y simplemente realizan acciones como moverse por el mapa y pararse a realizar cierta tarea.

Un ejemplo práctico de este tipo de *FSM* podría ser un *NPC* vendedor el cual esté continuamente moviéndose por la tienda y cuando el jugador llegue al mostrador y pulse el timbre, el vendedor vaya hasta él para darle la opción de comprar objetos. El *FSM* resultante de dicho comportamiento podría ser el siguiente:



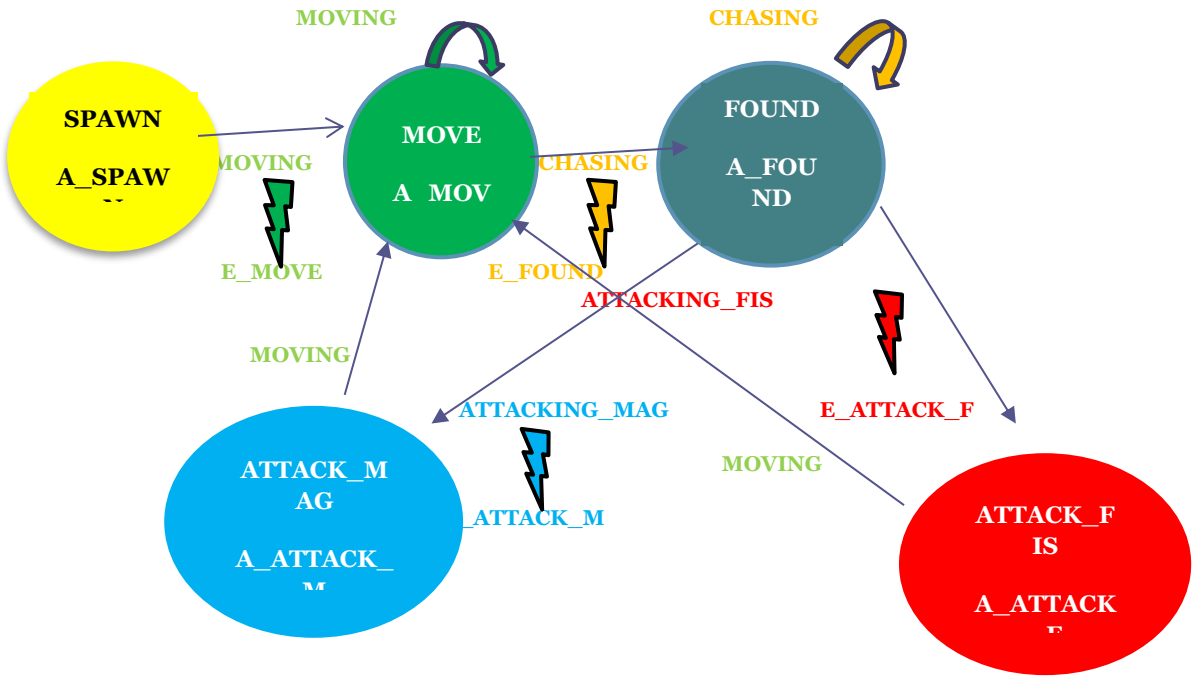
Dicho FSM consta de 3 estados; un estado inicial llamado **SPAWN**, el cual realiza la acción **A_SPAWN**, que consiste básicamente en posicionar al vendedor en la tienda, otro estado **MOVE** el cual realiza la acción **A_MOVE**, que se encarga de realizar el movimiento del *NPC* por la tienda, y por último un estado llamado **SELL**, el cual realizará la acción **A_SELL**, que ofrecerá la opción de mostrarle al jugador lo que puede comprar y venderle los objetos que este desee. También consta de 2 transiciones **MOVING**, la cual está ligada al evento o condición **E_MOVE**, de modo que cuando se cumpla dicha condición se pasará al estado **MOVE** y se realizará la acción **A_MOVE.** Mientras no se cumpla el evento o condición **E_SELL**, el vendedor seguirá dando vueltas por la tienda, hasta el momento que se cumpla dicho evento, instante en el cual el vendedor irá al mostrador y realizará la acción **A_SELL**. El primer evento podría darse cuando por ejemplo ya se haya puesto al vendedor en escena, momento en el que ya podrá comenzar a moverse y el segundo evento podría darse cuando el jugador pulsase el timbre del mostrador, por ejemplo.

Como se puede ver este tipo de *FSM* son muy apropiados para personajes secundarios que salen de fondo y cuyo comportamiento no sea muy sofisticado y no necesiten muchos estados. Para *bosses* o enemigos, estas *FSM* no son muy recomendadas ya que el comportamiento sería muy predecible y monótono, dando como resultado un enemigo cuyos movimientos sean muy fáciles de seguir y por tanto será una tarea muy sencilla para el jugador acabar con esta amenaza.

3. FSM Probabilístico

Este tipo de *FSM* añade al *FSM* determinista cierto grado de aleatoriedad, ya que lo que se hace es añadir una probabilidad a cada estado, de modo que, aunque se cumpla el evento o condición asociado a dicha transición cabe la posibilidad de que no se vaya al estado siguiente. Esto pese a que pueda parecer que no ofrece ninguna ventaja no es así, ya que como resultado se obtendrá una *FSM* en la que se tendrá la sensación de que el azar rige dicha máquina de estados proporcionando una solución a ese problema de comportamiento rutinario y monótono que puede llegar a dar la *FSM* determinista.

Por ejemplo, el problema que se tenía antes de añadir una *FSM* determinista a un enemigo y que este tuviera un comportamiento previsible, con esta *FSM* se solucionaría. Se podría tener por tanto un enemigo el cual se esté moviendo por la escena y cuando vea al jugador vaya a golpearle. Para que no siempre vaya a por él se le podría poner cierta probabilidad y así el jugador no siempre espere que vayan a por él, teniendo así la sensación de que no lo han detectado o que ha conseguido escapar. Cuando se tenga al jugador un poco más cerca se podría realizar un ataque a distancia o mágico y cuando se tenga cerca se realice un ataque físico. Sin embargo, para que no siempre se realice el ataque mágico se le podría poner una probabilidad. Esto como resultado tiene que el jugador no sepa con que lo van a atacar en todo momento y por tanto no tenga una forma de protegerse siempre y a veces el enemigo lo sorprenda y le golpee. El FSM de dicho comportamiento podría ser el siguiente:



En este FSM se tienen 5 estados; un estado inicial llamado **SPAWN** el cual realiza la acción **A_SPAWN**, que consiste en poner al enemigo en escena, otro estado llamado

MOVE, el cual realiza la acción **A_MOVE**, que consiste en mover al enemigo por el mapa de forma aleatoria, otro estado llamado **FOUND** el cual realiza la acción **A_FOUND,** que consiste en acercarse al jugador a una distancia lo suficiente cerca como para realizar un ataque, otro estado llamado **ATTACK_MAG**, el cual realiza la acción **A_ATTACK_M,** el cual realiza un ataque a distancia con el que daña al jugador y por ultimo se tiene el estado **ATTACK_FIS**, el cual realiza la acción **A_ATTACK_F**, el cual realiza un ataque cuerpo a cuerpo con el que daña al jugador. Este FSM también consta de varias transiciones las cuales tienen asociados diversos eventos o condiciones. La primera de ellas es la transición **MOVING** la cual tiene asociada el evento **E_MOVE** , para cumplir dicho evento por ejemplo y que el enemigo comenzase a moverse bastaría con comprobar si ya está en escena. La siguiente transición es **CHASING** la cual tiene asociada el evento **E_FOUND** el cual se podría cumplir cuando el jugador estuviera en el rango de visión del enemigo. La siguiente transición es **ATTACKING_FIS** la cual tiene asociada el evento **E_ATTACKING_F,** evento que se podría cumplir cuando el jugador estuviese al lado del jugador. Por último, se tiene la transición **ATTACKING_MAG** la cual tiene asociada el evento **E_ATTACKING_M,** evento que se podría cumplir cuando el jugador estuviese a pocas casillas del enemigo.

A los estados que componen esta FSM se les ha dado un porcentaje para que así haya ese grado de aleatoriedad del que se estaba hablando. Para que nada más encontrar no se pase al estado FOUND, y a veces el jugador tenga la sensación de que o ha escapado o no ha sido detectado, se le dio a dicho estado un porcentaje del 75 %. En cuanto a los ataques, para que no siempre se realice un ataque mágico, aunque el jugador este lo suficientemente cerca, se le dio un porcentaje al estado ATTACK_MAG del 35 %, para que asi no ocurra con frecuencia este ataque puesto que será más poderoso que el ataque físico. Sin embargo, para que cuando el jugador este al lado del enemigo no se vaya sin recibir ningún daño el estado de ATTACK_FIS tendrá un porcentaje del 100% por lo que siempre realizará un ataque cuando el jugador esté cerca.

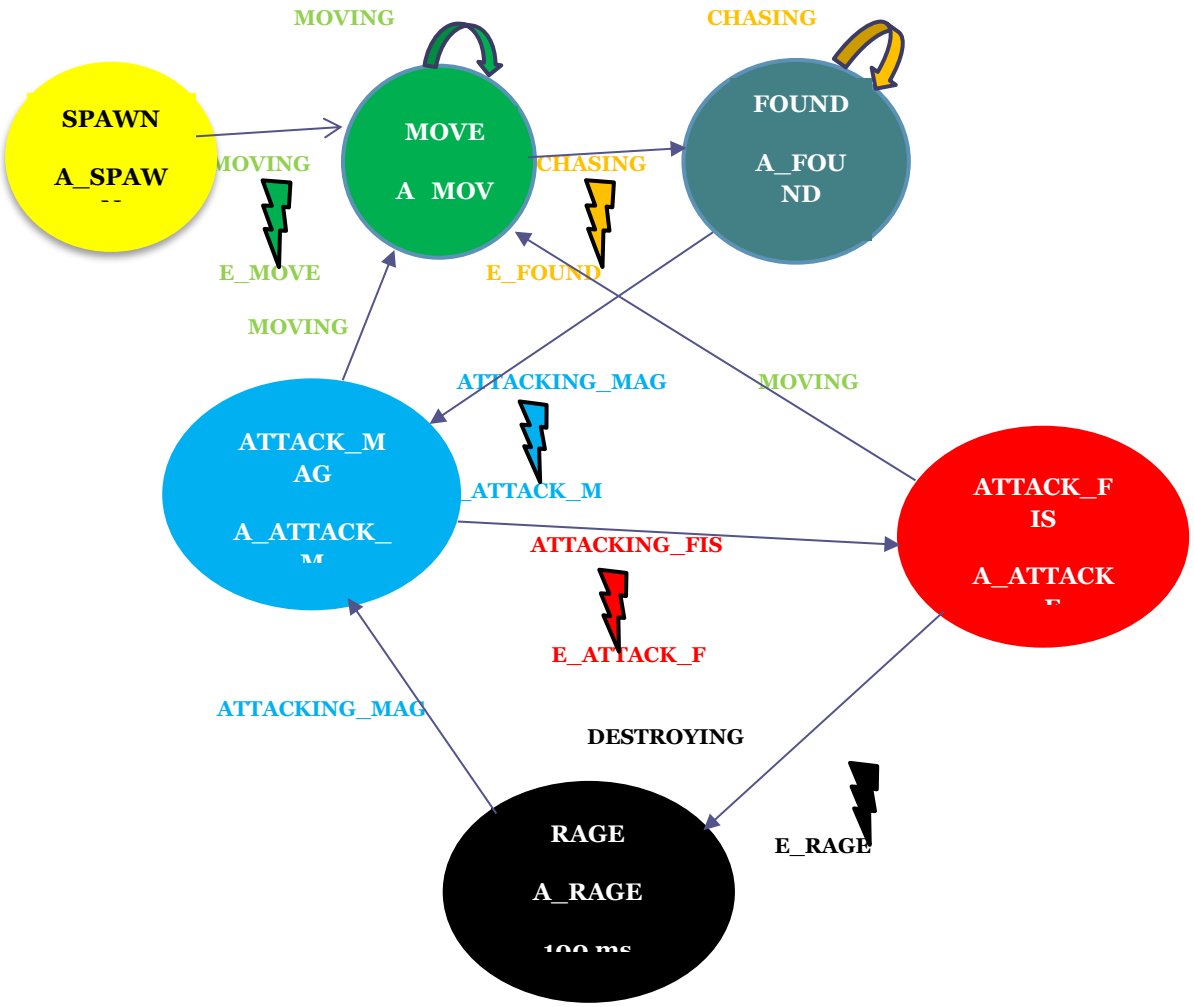
Además, se han realizado transiciones entre diversos estados para asi poder dar lugar a un comportamiento más fluido, sin que el enemigo se quede permanentemente en un mismo estado. Por ejemplo, una vez el enemigo hay realizado un ataque, se vuelve al estado de MOVE y si el jugador sigue aún cerca puede que se vaya al estado de FOUND, o puede que no se vaya con lo que se tendrá que volver a buscar al jugador.

4. FSM Inercial

Este tipo de *FSM* indeterminista añade al FSM determinista una latencia en cada estado, para que así al llegar a un estado se realice una espera (en milisegundos) antes de comprobar si se cumple la condición o evento asociada a la transición de dicho estado. Esta espera se hace para solventar un problema bastante común en las FSM el cual es la oscilación entre estados. Este problema ocurre cuando al llegar a un estado se cumple la condición asociada a la transición de dicho evento, con lo cual se pasará al siguiente estado, en el cual también ocurre lo mismo por lo que se pasará a otro estado.

Esto lo que provoca es que en pocos ciclos se hayan pasado por diversos estados y por tanto se hayan realizado demasiadas acciones en un corto periodo de tiempo. Visualmente lo que se observaría sería un *boss* o *NPC* el cual presenta un comportamiento muy rápido o nervioso. Es por ello por lo que, al añadir esta espera en milisegundos a los estados, lo que se consigue no es solo retrasar el paso de un estado a otro, si no también que se vuelva a comprobar la condición asociada a dicho estado para ver si aún se tiene que pasar al siguiente estado.

Si por ejemplo se quiere hacer el comportamiento de un *boss* en el que se lleven a cabo una serie de acciones seguidas, como podría ser el realizar un combo de golpes, esta sería una FMS idónea para elegir. Un ejemplo de dicho comportamiento podría ser el de un enemigo el cual , una vez ya haya encontrado al jugador, y cuando éste esté cerca se realice un ataque a distancia , si el jugador sigue cerca se seguirá golpeándolo con un ataque físico y si el jugador todavía no se ha alejado se realizará un golpe en modo *rage* el cual aumenta tu daño. Si en algún momento el jugador se alejase lo que se haría seria, si se completó el combo comenzar otro nuevo, y si en mitad del combo se aleja comenzar a moverse para encontrarlo de nuevo. El FSM de dicho comportamiento podría ser el siguiente:



En este FSM se tienen 6 estados; un estado inicial llamado **SPAWN** el cual realiza la acción **A_SPAWN**, que consiste en poner al enemigo en escena, otro estado llamado **MOVE**, el cual realiza la acción **A_MOVE**, que consiste en mover al enemigo por el mapa de forma aleatoria, otro estado llamado **FOUND** el cual realiza la acción **A_FOUND,** que consiste en acercarse al jugador a una distancia lo suficiente cerca como para realizar un ataque, otro estado llamado **ATTACK_MAG**, el cual realiza la acción **A_ATTACK_M,** el cual realiza un ataque a distancia con el que daña al jugador, otro estado **ATTACK_FIS**, el cual realiza la acción **A_ATTACK_F**, el cual realiza un ataque cuerpo a cuerpo con el que daña al jugador, y por ultimo se tiene el estado **RAGE** el cual realiza la acción **A_RAGE**, el cual realiza un ataque muy potente al jugador. Este FSM también consta de varias transiciones las cuales tienen asociados diversos eventos o condiciones. La primera de ellas es la transición **MOVING** la cual tiene asociada el evento **E_MOVE** , para cumplir dicho evento por ejemplo y que el enemigo comenzase a moverse bastaría con comprobar si ya está en escena. La siguiente transición es **CHASING** la cual tiene asociada el evento **E_FOUND** el cual se podría cumplir cuando el jugador estuviera en el rango de visión del enemigo. La siguiente transición es **ATTACKING_FIS** la cual tiene asociada el evento **E_ATTACKING_F,** evento que se podría cumplir cuando el jugador estuviese al lado del jugador. Por otro lado, se tiene la transición **ATTACKING_MAG** la cual tiene asociada el evento **E_ATTACKING_M,** evento que se podría cumplir cuando el jugador estuviese a pocas casillas del enemigo. Por último, se tiene la transición **DESTROYING** la cual tiene asociada el evento **E_DESTROY,** evento que se podría cumplir cuando, por ejemplo, se hubiesen realizado todos los golpes anteriores que conforma el combo.

Si esta FSM se aplicase de forma que no se añadiese ninguna latencia a los estados, y el jugador estuviera cerca del personaje en todo momento sin alejarse lo que pasaría es que se realizarían todos los golpes de gorma muy rápida, dando la sensación al jugador de que no es justo, puesto que no se le ha dado ninguna oportunidad para evitar o esquivar dichos golpes.

Además, se han realizado transiciones entre diversos estados para así poder dar lugar a un comportamiento más fluido, sin que el enemigo se quede permanentemente en un mismo estado. Por ejemplo, una vez el enemigo hay realizado un ataque, si el jugador se ha alejado en lugar de quedarse en el estado actual lo que se hace es pasar al estado MOVE para así poder moverse y comenzar a buscar de nuevo al jugador.

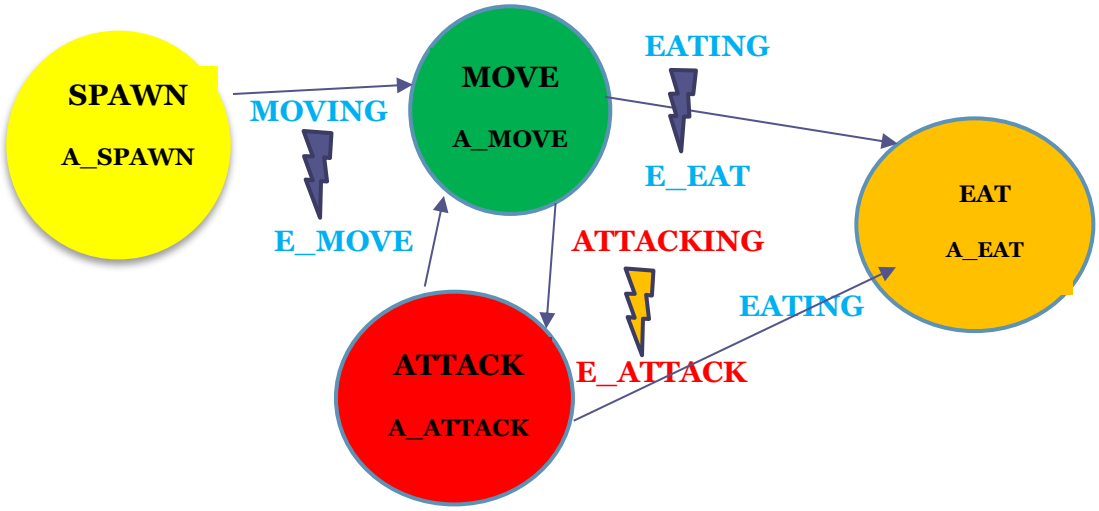
5. FSM Basada en Pilas

Este tipo de *FSM* son bastante útiles ya que permiten crear un efecto de *“memoria”* en el *NPC* o *boss* que haga uso de esta *FSM*. Estas *FSM* son de las más útiles, pues proporcionan al personaje que haga uso de ellas, la posibilidad de poder, mientras está en un estado realizando las acciones que toquen, detener esa acción para ir a otro estado, que sea más importante, a realizar una nueva acción, y, tras finalizar esa nueva acción, poder volver al estado en el que se detuvo la ejecución continuando con su

ejecución. Para ello deberá, de cierta forma recordar a donde tiene que volver y es por ello por lo que a los estados se les tendrá que añadir cierta prioridad. Este aspecto se añadió para establecer un orden a la hora de apilar los diferentes estados. De esta forma lo que se hace es activar los estados más prioritarios y así llevar cierto orden de apilado, y por otro lado se consigue que los estados que tengan la misma prioridad no se interrumpan entre ellos, a no ser que tengan una transición que los conecte.

Ejemplos de comportamientos que podrían emplear este *FSM* podria ser el comportamiento de un *boss* en el que se lleven a cabo una serie de acciones seguidas,

como podría ser el realizar una serie de golpes, pero que cada golpe consuma una energía y cuando esta se agote se tenga que descansar o se quiere llevar a cabo ciertas acciones, pero más tarde se tiene que volver a una ubicación del mapa. El FSM con uno de estos comportamientos podría ser el siguiente:



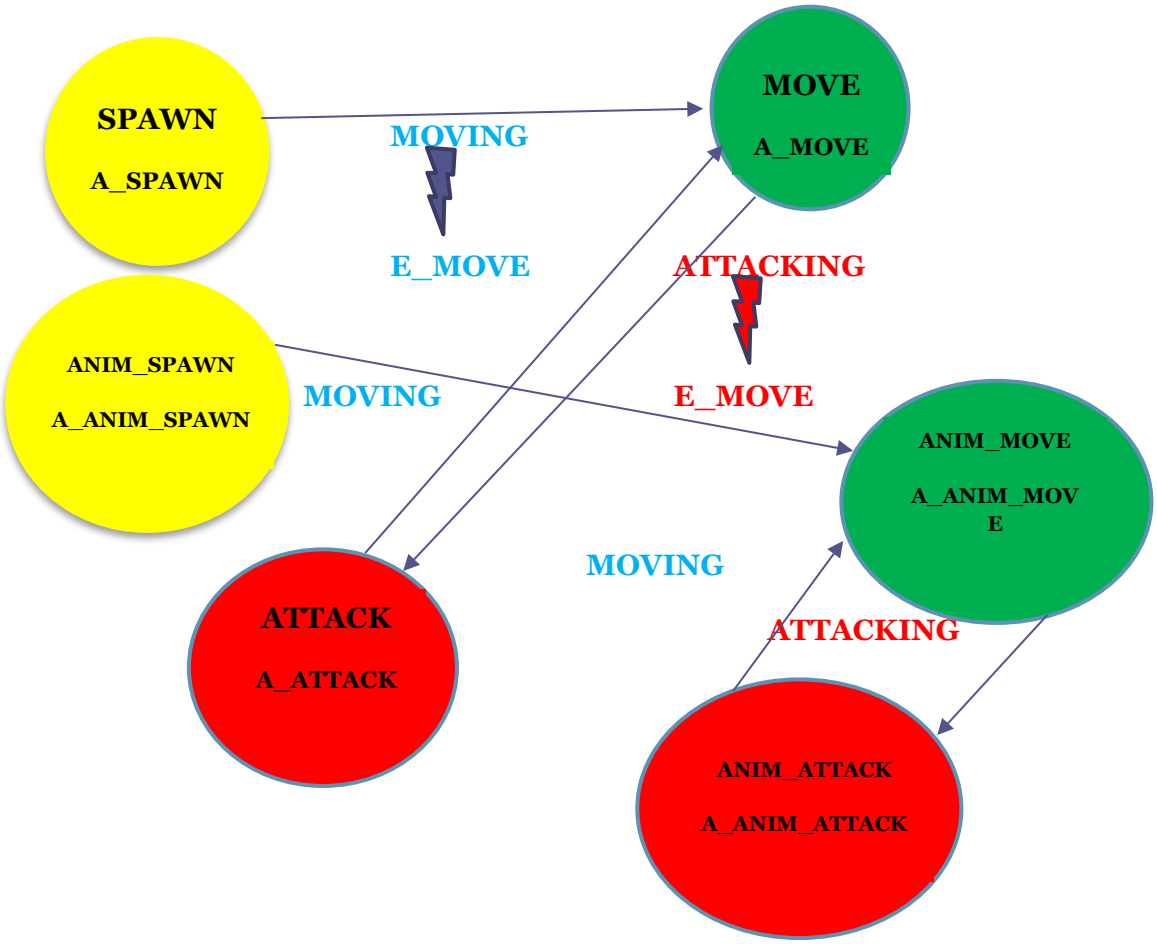
En este FSM se tienen 4 estados; un estado inicial llamado **SPAWN** el cual realiza la acción **A_SPAWN**, que consiste en poner al enemigo en escena, otro estado llamado **MOVE**, el cual realiza la acción **A_MOVE**, que consiste en mover al enemigo por el mapa de forma aleatoria, , otro estado **ATTACK**, el cual realiza la acción **A_ATTACK**, el cual realiza un ataque cuerpo a cuerpo con el que daña al jugador, y por último se tiene el estado **EAR** el cual realiza la acción **A_EAT**, el cual realiza una espera para que el enemigo pueda alimentarse y continuar así atacando y moviéndose. Este FSM también consta de varias transiciones las cuales tienen asociados diversos eventos o condiciones. La primera de ellas es la transición **MOVING** la cual tiene asociada el evento **E_MOVE** , para cumplir dicho evento por ejemplo y que el enemigo comenzase a moverse bastaría con comprobar si ya está en escena. La siguiente transición es **EATING** la cual tiene asociada el evento **E_EAT** el cual se podría cumplir cuando el enemigo ya hubiese consumido toda la energía que le quedase y por tanto debería descansar.

Esta *FSM* es especialmente potente debido a que cuando se cumpla la condición se irá al estado **EAT** pero al salir de este se sabrá perfectamente de cual se vino y por tanto se volverá a este, siendo esto un comportamiento muy importante. También son

especialmente útiles estas FSM cuando se quiere que el enemigo o NPC recuerde ciertos lugares a visitar o que cuando sucedan algunos eventos se vaya a otro lugar, pero al acabarse dichos eventos se continúe con lo que se estaba haciendo.

6.FSM Basada en Estados Concurrentes

Este tipo de *FSM* otorga la posibilidad de ejecutar varios estados el mismo tiempo. El número de estados que se activaran será indicado por el desarrollador. El funcionamiento de este tipo de *FSM* se basa en *"créditos"* o *"monedas",* asociados a cada estado, los cuales se generan en ciertos estados y son consumidos en otros. Si un estado tiene créditos de ejecución podrá estar activo, y cuando se salte de un estado a otro el primero le otorgará un crédito al segundo y si el primero se queda sin créditos entonces ese estado quedará inactivo. Estos créditos serán de utilidad para asegurar ese grado de paralelismo que se busca en este tipo de *FSM*. Es por ello por lo que serán de especial utilidad cuando se quieran ejecutar varios estados a la vez , los cuales sean de aspectos diferentes, como podría ser por ejemplo la IA, la música y las animaciones de un *boss* o un *NPC*, estados que podrían darse a la vez. El FSM con uno de estos comportamientos podría ser el siguiente:



Esta *FSM* presenta los mismos estados que la FSM determinista, pero sustituyendo el estado **SELL** por un estado **ATTACK**, y se le añaden 3 estados más los cuales son; **ANIM_SPAWN**, el cual lleva a cabo la acción **A_ANIM_SPAWN**, donde se colorea al enemigo de amarillo, el estado ANIM_MOVE, la cual realiza la acción **A_ANIM_SPAWN**, donde se colorea de verde al enemigo y por ultimo **ANIM_ATTACK**, la cual realiza la acción **A_ANIM_ATTACK**, donde se realiza la animación del enemigo *enemyAttack* y se colorea al enemigo de rojo. Estos nuevos estados se conectan mediante las transiciones que ya estaban creadas, ***MOVING*** y ***ATTACKING***, razón por la cual el evento que permite pasar de un estado al otro es **E_MOVE y E_ATTACK** respectivamente.

Pese a que pueda parecer poco útil esta *FSM*, el separar aspectos diferentes del comportamiento inteligente es muy beneficioso no solo para obtener un código más estructurado, sino también para poder, a la hora de depurar errores, poder ubicarlo mucho más fácil.

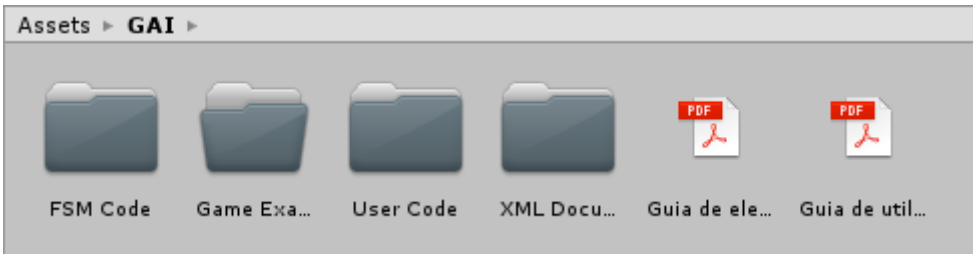
Guía de utilización de la API

En esta guía de utilización de la API se explicará paso a paso como hacer un uso correcto de la API, tanto si se decide hacer uso de las herramientas que se proporcionan siendo estas las plantillas de las *FSM* en formato *XML,* situadas en la carpeta *XML Documents,* y las dos clases *Tags* y *FSM_Parser*, situadas en la carpeta *User Code,* y en la carpeta *FSM Code* respectivamente.

Para que le sea más fácil entender cómo utilizar la *API* lo primero que se hará será empezar con una explicación de cómo está estructurada la *API* y seguidamente se explicaran los pasos a seguir para hacer uso de ésta con o sin las herramientas opcionales que se proporcionan.

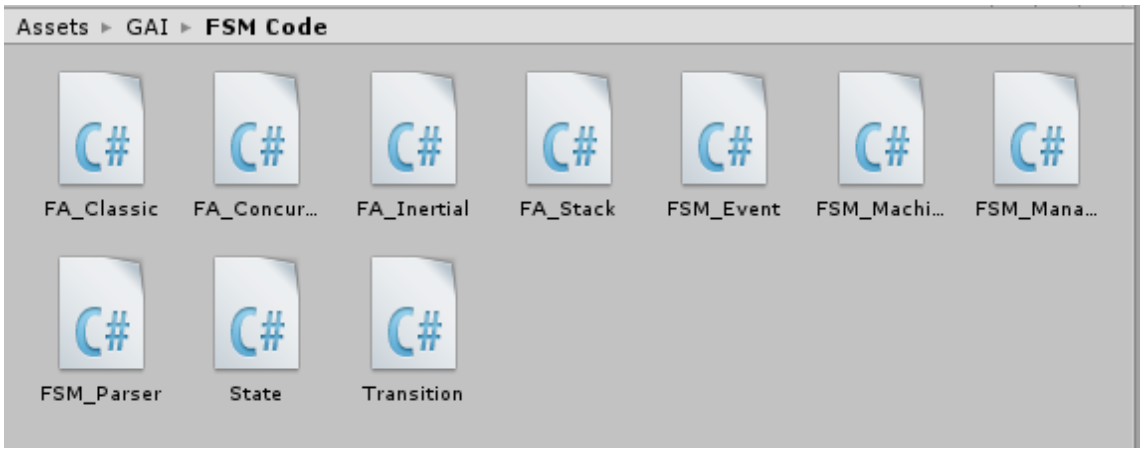
1. Estructura de la *API*

Como puede ver, en el apartado de *Asset*, le aparecerá una nueva carpeta llamada *GAI (game artificial intelligence)*. En dicha carpeta se encontrarán diversas subcarpetas, necesarias para el correcto funcionamiento de la *API*, una carpeta que contendrá un ejemplo de un videojuego en el cual se ha integrado la *API*, para que así tenga un ejemplo base y 2 documentos *pdf* en los cuales se explica cómo hacer uso de la herramienta, el cual es este documento, otro para saber como elegir la máquina de estados finito que más le convenga, según sus necesidades o los objetivos que desee cumplir en su proyecto, lectura la cual también se recomienda realizar.



Estructura del *Asset GAI*

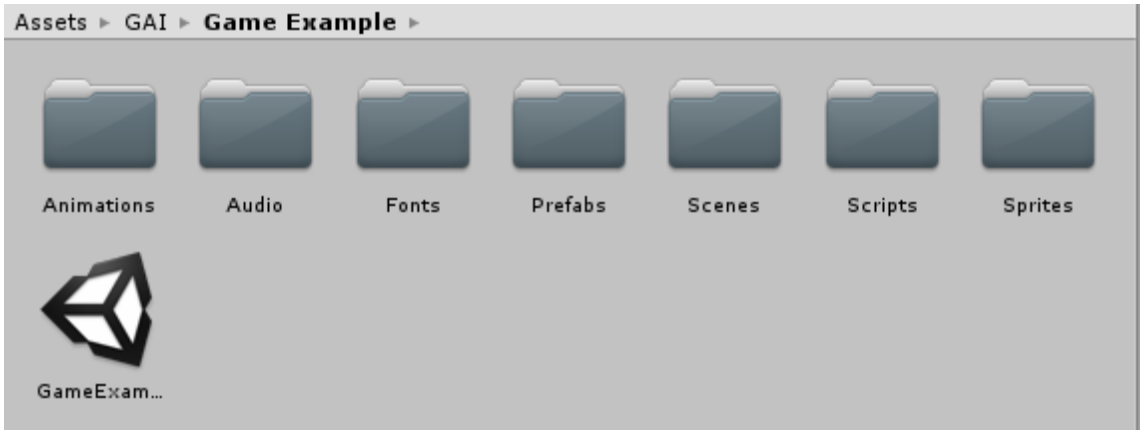
En la primera carpeta, llamada ***FSM Code***, se encuentra todo el código encargado del funcionamiento óptimo, tanto de la máquina de estados finitos que quiera hacer uso, como del *parser* encargado de trabajar con las plantillas *XML*, en caso de que haga uso de ellas. Dentro de las clases encargadas del correcto funcionamiento de la *API* se tienen varias clases, cada una de ellas encargada de realizar una parte importante. La primera de ellas se llama ***FSM_Manager*** y esta clase se encarga principalmente de asignarle a la entidad que lo pida, la máquina de estados finito que desee. Cuando una entidad necesite hacer uso de una *FSM* lo que se hará es invocar el método *CreateMachine()* e internamente el *FSM_Manager* lo que hará será buscar en su diccionario la máquina de estados que haya sido solicitada e instanciará un objeto *FSM_Machine* que corresponde al autómata almacenado. La siguiente clase por analizar es la clase ***FSM_Machine*** la cual, a diferencia de la clase *FSM_Manager*, cuya función se centra más en el almacenaje de *FSM* y asignaciones, se centra en recorrer de forma óptima la *FSM* elegida según su tipo. Además, también se tendrá en esta carpeta el código de cada una de las máquinas de estados finitos, ***FA_Clasic***, para la *FSM* determinista y la *FSM* probabilística, ***FA_Concurrent***, para la *FSM* basada en estados concurrentes, ***FA_Inertial***, para la *FSM* inercial y ***FA_Stack***, para la *FSM* basada en pilas. A parte de estas clases también se encontrarán la clase ***State*** y la clase ***Transition***, las cuales sirven para crear los estados y las transiciones respectivamente, según el tipo de *FSM* que se haya elegido, ya que como podrá ver en la guía de elección de FSM, al haber diferencias entre ellas, los estados y las transiciones también cambian, necesitando más o menos variables. Por último, y para finalizar con el contenido de esta carpeta, se encuentra la clase ***FSM_Parser*** cuyo uso, tal y como se ha comentado anteriormente, es totalmente opcional ,y su función se basa en cargar automáticamente las variables de la *FSM* desde un documento de texto *XML*. El contenido de dicha carpeta se puede visualizar a continuación:



Contenido de la carpeta *FSM Code*

Tal y como se puede intuir todas las clases que forman esta carpeta no deben ser modificadas, ya que tocar o modificar cualquier línea de código de dichas clases, que provocará fallos en el funcionamiento de la *API* que serán muy difíciles de localizar así como de solventar una vez realizados.

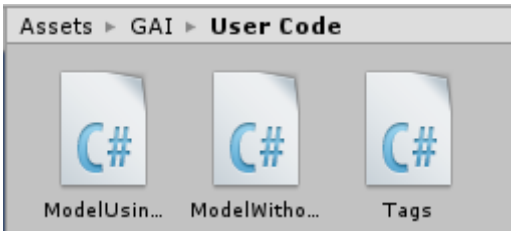
En la siguiente carpeta, llamada Game Example, se encuentra un ejemplo de videojuego el cual hace uso de la *API*, junto con las plantillas *XML* de las máquinas de estados finitos rellenadas las cuales se emplearon en dicho ejemplo, así como las clases en las que se realizó la integración de la *API*, para que vea como se haría la integración en un ejemplo real. El videojuego que se ha elegido es uno de los videojuegos tutoriales que *Unity* proporciona para su realización llamado "*2D Roguelike tutorial*", y una práctica que se recomienda hacer es comparar el comportamiento de dicho videojuego con el comportamiento que se puede añadir haciendo uso de la *API*. El contenido de dicha carpeta se puede visualizar a continuación:



Contenido de la carpeta *Game Example*

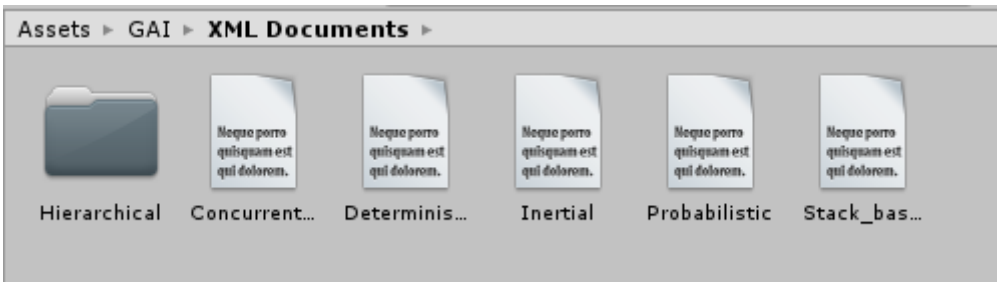
La siguiente carpeta que se tiene se llama ***User Code***, y en esta carpeta será, junto con la carpeta que contiene los documentos *XML*, donde tendrá que hacer modificaciones. En esta carpeta se encuentran 3 clases, la clase ***Tags***, la cual sirve tanto para definir las variables que representan los estados, acciones, transiciones, y eventos, como darles valor, y las clases ***ModelUsingTools*** y ***ModelWithoutUsingTools***, en las cuales tendrá un modelo con todas las variables, llamadas a métodos y valores a proporcionar,

que tendrá que emplear para hacer uso de la *API*, así como comentarios explicando tanto que se está haciendo en cada línea de código como el orden a seguir, los posibles métodos que puede emplear dependiendo de la FSM empleada, y que valores dicho método debe recibir. En dichos modelos se muestra cómo usar la *API* tanto si se decide emplear la clase *FSM_Parser* como los documentos XML(***ModelUsingTools***), como si se decide no hacerlo(***ModelWithoutUsingTools)***. La clase ***Tags***, al igual que el uso de los documentos *XML* y el *FSM_Parser,* es opcional pese a que su uso es muy recomendado, ya que la función de dichas herramientas se basa sobre todo en la organización, evitando así que se tenga una clase muy extensa en la que gran parte del código son simples definiciones, asignaciones de variables y métodos para retornar valores. Delegando esta acción en otra clase se obtendrá un código mucho más limpio, al igual que haciendo uso de los documentos *XML*, ya que así se evitará realizar enormes invocaciones a métodos para crear los estados, las transiciones y los eventos o condiciones asociados a ellas, así como la conexión entre los estados. Simplemente rellenando dicho documento y haciendo uso del *FSM_Parser* se conseguiría el mismo resultado de una forma más limpia. El contenido de dicha carpeta se puede visualizar a continuación:



Contenido de la carpeta *User Code*

Por último, en la carpeta llamada ***XML Documents***, se encontrarán todos los documentos *XML,* los cuales sirven como plantilla para crear cada una de las máquinas de estados finitos, habiendo, por tanto, una plantilla por cada tipo de *FSM*. Cada plantilla contendrá los apartados correspondientes para crear los elementos necesarios que compongan cada una de las máquinas de estados finitos, así como una descripción de cada uno de los apartados, para que así sepa no solo la utilidad de dicho apartado si no también que valores están permitidos. Además de esto se proporciona una carpeta llamada *Hierarchical* para colocar allí las plantillas que cree para realizar máquinas anidadas. El uso de estos documentos es totalmente opcional, no siendo por tanto obligatorio para el correcto de la *API*, sin embargo, son muy recomendados para obtener tanto un código final más estructurado y claro, como para reducir así la aparición de errores y tener una facilidad mayor a la hora de depurar fallos. En estas plantillas encontrará apartados para darle nombre a los estados, transiciones, acciones y eventos, así como establecer desde que estado se puede ir a otro y las acciones que se llevaran a cabo. Tal y como se ha mencionado, el uso de estas plantillas es totalmente opcional y por tanto si el usuario decide no hacer uso de ellas lo que tendrá que hacer es realizar llamadas a los métodos que se encarguen de crear todos los elementos de la *FSM*, así como realizar las conexiones entre los estados. El contenido de dicha carpeta se puede visualizar a continuación:



Contenido de la carpeta *XML Documents*

Por último, junto con estas carpetas, y tal y como se ha comentado anteriormente, se han añadido los documentos que sirven de guía para la elección de la máquina de estados finitos, así como una guía de utilización de la *API,* ambos en formato *pdf*.

2. Utilización de la API con las herramientas

Dado que, si se utilizan las herramientas que proporciona la *API*, se tendrá más comodidad a la hora de hacer uso de la máquina de estados finitos elegida, la mayoría de las acciones que se realizaran al hacer uso de estas herramientas serán rellenar los apartados señalizados con el nombre que se desee, en el caso de los documentos *XML*, y darles un valor en el caso de la clase *Tags*.

El primer paso que realizar, tras haber hecho la elección de la *FSM* de la que se quiera hacer uso (para una elección de esta lo más correcta posible según sus necesidades se recomienda la lectura de la guía de elección de FSM), es ir a la carpeta ***XML Documents*** y, seleccionando la plantilla de la *XML* elegida, rellenar los apartados necesarios para crear tanto los estados, transiciones y eventos ligados a dichas transiciones. Pese a que cada plantilla de cada una de las *FSM* es diferente, más o menos todos comparten los mismos apartados, puesto que todas parten de una *FSM* Determinista a la cual se le añaden los apartados que sean necesarios, razón por la cual los siguientes apartados los tendrán todas las *FSM*. Cada tipo de plantilla además de venir con las variables necesarias para cada tipo de *FSM*, viene con el apartado que indica el tipo de *FSM* ya rellenado, siendo 4 los tipos de *FSM*; *CLASSIC* en el caso tanto de la *FSM* Determinista (pero con el apartado *Probabilistic* marcado como NO) como de la *FSM* Probabilistica (pero con el apartado *Probabilistic* marcado como YES), *INERTIAL* en el caso de la *FSM* Inercial, *STACK_BASED* en el caso de la *FSM* basada en pilas y por último, *CONCURRENT_STATES* para el caso de la FSM basada en estados concurrentes. Por lo que lo primero que debe comprobar es que este tag corresponde con el tipo de *FSM* que haya elegido.

Una vez ya tenga seleccionada su plantilla y sea la correcta, se podrán comenzar a rellenar los apartados necesarios, los cuales se explicarán en el orden de aparición.

Para empezar, se tendrá que darle un nombre a la *FSM* de la cual se vaya a hacer uso rellenando el campo *FSMid,* el cual se puede ver a continuación, y para ello se tendrá

que borrar el comentario que hay entre las etiquetas *FSMid* y colocar el nombre que se desee.

```
<FSMid><!--put here the name of this machine--></FSMid>
```

Campo *FSMid* del documento *XML*

Esta acción es importante ya que a raíz de este nombre se accederá, más adelante, a las variables de dicha *FSM*, mediante el *FSM_Parser* y, junto con la clase *Tags,* se les dará el valor, realizando así un *parseado*. Seguidamente se le dará nombre al *callback* al cual el *FSM_Manager* llamará internamente cada ciclo para ver si ha ocurrido algún evento o condición, y por tanto se deba ir a un estado y realizar la acción que corresponda. Para ello lo que se deberá hacer es rellenar la etiqueta *callback*, borrando el comentario entre dichas etiquetas y colocando el nombre que se desee para dicho *callback*.

```
<Callback><!--put here the name of events routine--></Callback>
```

Campo *Callback* del documento *XML*

El nombre que se dé a dicho *callback* será un método que se deberá implementar en la clase que haga uso de la *API*, por lo que los nombres deberán coincidir, y si tanto al acceder a la *FSM* como cuando se realice el *callback,* los nombres no coinciden con los que se puso en el documento *XML,* se obtendrá un fallo de compilación, en concreto un *Null pointer Exception*, debido a que no se encontrará dicho método, por tanto es importante asegurarse de que dichos nombres coincidan en ambos documentos.

A continuación se procederá a rellenar los campos necesarios para la creación de los estados, los cuales se pueden ver en la siguiente imagen, y para ello primero se indicará si el estado es inicial(rellenando con YES) o no(rellenando con NO), después el nombre que se le quiere dar al estado, rellenando la etiqueta ***<S_Name>,*** y las acciones asociadas a dicho estado, pudiendo tener acciones de entrada (etiqueta ***<S_InAction>***), de salida (etiqueta ***<S_OutAction>****)* y por estar en el estado (*S_Action*). Si se desea, un estado puede no tener acciones asociadas, asignándole el valor *NULL* a la etiqueta de la acción correspondiente. Además de esto si se desea se puede rellenar el Apartado ***<S_Fsm> </S_Fsm>*** con la ubicación de otra *FSM* por si se quiere que dicho estado contenga una máquina que lleve a cabo las acciones. Dicha FSM se recomienda que esté contenida en la carpeta *Hierarchy* para así tener una mayor organización.

Al rellenar todos estos campos se habrá creado 1 estado y para crear más estados lo único que se deberá de hacer será copiar el bloque desde ***<State Initial =””>*** hasta ***</State>***,copiarlo a continuación, dentro del bloque ***<States> </States>***, y rellenarlo con los nuevos datos que formaran el siguiente estado.

```
<States>

  <State Initial="">
    <S_Name><!--Put here the name of this state (no quotes)--></S_Name>
    <S_Action><!--Put here the name of this state action (no quotes, it can be NULL)--></S_Action>
    <S_inAction><!--Put here the name of this state IN action (no quotes, it can be NULL) --></S_inAction>
    <S_outAction><!--Put here the name of this state OUT action state (no quotes, it can be NULL) --></S_outAction>
    <S_Fsm><!--Put here the PATH to a submachine (this could be empty) --></S_Fsm>
  </State>

</States>
```

Estado de la *FSM* determinista del documento *XML*

Después de esto se procederá a rellenar los campos necesarios para la creación de las transiciones y los eventos asociados a dichas transiciones. Una transición estará compuesta por el nombre de la transición (etiqueta ***T_Name***), el nombre del estado de origen del cual se parte(etiqueta ***T_Origin***), el nombre del estado destino al cual se dirige(etiqueta ***T_Destination***), una acción(etiqueta ***T_Action***), la cual se puede o no asociar a dicha transición, puesto que dándole el valor de *NULL* no se realizará ninguna acción, y por último los eventos que harán que esta transición se realice y se pase al estado destino. Estos eventos, como se verá más adelante, están ligados a condiciones a cumplir, y cuando esto sucede, a partir de dichos eventos la *FSM* obtendrá las acciones que se deban realizar. Un evento estará compuesto por un nombre (etiqueta ***<ID>***) y un tipo de evento(etiqueta ***<Type>***) pudiendo tener este campo como valor o bien *BASIC* o bien *STACKABLE*. Por último, si se desean añadir más eventos a dicha transición lo único que deberá hacerse será copiar el esquema del evento y pegarlo entre las etiquetas ***<Events> </Events>***.

Al igual que en el caso de los estados, si se quieren añadir más transiciones lo único que se tiene que hacer es añadir más bloques ***<Transition> </Transition>*** entre las etiquetas ***<Transitions></Transitions>.*** En la siguiente imagen se podrá ver todas las etiquetas comentadas anteriormente:

```
<Transitions>

  <Transition>
    <T_Name><!--Put here the name of this transition (no quotes)--></T_Name>
    <T_Origin><!--Put here the name of the origin state (no quotes)--></T_Origin>
    <T_Destination><!--Put here the name of destination state (no quotes)--></T_Destination>
    <T_Action><!--Put here the name of this state (no quotes, it can be NULL)--></T_Action>

    <Events>

      <Event>
        <ID><!--Put here the name of this Event (no quotes)--></ID>
        <Type><!--Put here the type of this Event (no quotes), BASIC or STACKABLE--></Type>
      </Event>

    </Events>

  </Transition>

</Transitions>
```

Transiciones y Eventos de la *FSM* determinista del documento *XML*

Todas las demás *FSM* contendrán estos mismos apartados a rellenar en sus plantillas *XML*, pero tendrán algunos apartados más a rellenar, los cuales se verán a continuación.

Empezando por la *FSM* Probabilística, se deberá indicar la probabilidad que se desea que tenga cada transición, rellenando la etiqueta ***<Probability></Probability>*** con un valor entre 0, siendo la probabilidad más baja y por tanto dicho estado nunca será alcanzable, y 100, siendo la probabilidad más alta y por tanto dicho estado, siempre que se cumpla el evento asociado a dicha transición será alcanzable. Dicha etiqueta se añadirá en el bloque de las transiciones tal y como puede verse a continuación:

```
<Transitions>

  <Transition>
    <T_Name><!--Put here the name of this transition (no quotes)--></T_Name>
    <T_Origin><!--Put here the name of the origin state (no quotes)--></T_Origin>
    <T_Destination><!--Put here the name of destination state (no quotes)--></T_Destination>
    <T_Action><!--Put here the name of this state (no quotes, it can be NULL)--></T_Action>

    <Events>

      <Event>
        <ID><!--Put here the name of this Event (no quotes)--></ID>
        <Type><!--Put here the type of this Event (no quotes)--></Type>
        <!--Type can be BASIC or STACKABLE-->
      </Event>

    </Events>

    <Probability><!--put here the probability of execution of this Transition (between 0 and 100)--></Probability>

  </Transition>


</Transitions>
```

Transiciones y Eventos de la *FSM* Probabilística del documento *XML*

La siguiente plantilla es la de la *FSM* Inercial, y en ella se añade la etiqueta ***<S_Latency></S_Latency>*** a los estados, para que así se puede indicar la latencia que quiera que tenga el estado. Tal y como se indica en la plantilla este valor estará en milisegundos y dicha etiqueta se añadirá al bloque de los estados, tal y como puede verse a continuación:

```
<States>

  <State Initial="">
    <S_Name><!--Put here the name of this state (no quotes)--></S_Name>
    <S_Action><!--Put here the name of this state action (no quotes, it can be NULL)--></S_Action>
    <S_inAction><!--Put here the name of this state IN action (no quotes, it can be NULL) --></S_inAction>
    <S_outAction><!--Put here the name of this state OUT action state (no quotes, it can be NULL) --></S_outAction>
    <S_Fsm><!--Put here the PATH to a submachine (this could be empty) --></S_Fsm>
    <S_Latency><!--Put here LATENCY of this state(in milliseconds) --></S_Latency>
  </State>

</States>
```

Estados de la *FSM* Inercial del documento *XML*

Seguidamente se tiene la *FSM* basada en pilas, en la que se añade la etiqueta ***<S_Priority></S_Priority>*** , para así indicar la prioridad de los estados, teniendo en cuenta que la prioridad más alta será la que contenga el número más alto y 0 será la prioridad más baja. Además, en estas *FSM* habrá que indicar que estados pueden ser apilados y para ello se marcaran las transiciones que vayan de un estado menos prioritario a otro más prioritario como *STACKABLE* en lugar de *BASIC*, dejando como *BASIC* las transiciones entre estados igual de prioritarios o de estados más prioritarios y menos prioritarios. Además de esto para que se dé este correcto comportamiento de apilado cada estado deberá tener una transición que vaya a sí mismo. Esta etiqueta se añadirá al bloque de los estados, tal y como puede verse a continuación:

```
<States>

  <State Initial="">
    <S_Name><!--Put here the name of this state (no quotes)--></S_Name>
    <S_Action><!--Put here the name of this state action (no quotes, it can be NULL)--></S_Action>
    <S_inAction><!--Put here the name of this state IN action (no quotes, it can be NULL) --></S_inAction>
    <S_outAction> <!--Put here the name of this state OUT action state (no quotes, it can be NULL) --></S_outAction>
    <S_Fsm><!--Put here the PATH to a submachine (this could be empty) --></S_Fsm>
    <S_Priority><!--Put here the priority of this state (>=0) the greater, the more priority --></S_Priority>
  </State>

</States>
```

Estados de la *FSM* basada en pilas del documento *XML*

Por último, se tiene la *FSM* de estados concurrentes en la que se añade el apartado ***<Fsm MaxConcurrent*** = “”> , para indicar mediante un valor entero, el número de estados que se puedan dar a la vez, y también la etiqueta ***<S_Credits></S_Credits>*** la cual se añadirá al bloque de los estados, para indicar los créditos iniciales con los que empezará dicho estado. Es por esto por lo que ésta será la única *FSM* que pueda tener mas de un estado inicial, pudiendo tener diversos estados cuya etiqueta ***<State Initial>*** sea *YES*, teniendo las demás *FSM* solo un estado con dicho valor. Las etiquetas de esta *FSM* podrán verse a continuación:

```
<Fsm MaxConcurrent="">

  <Callback> <!--put here the name of events routine--> </Callback>

  <States>
    <State Initial="">
      <!--Initial has to be "YES" or "NO" -->
      <S_Name> <!--Put here the name of this state (no quotes)--></S_Name>
      <S_Action><!--Put here the name of this state action (no quotes, it can be NULL)--></S_Action>
      <S_inAction><!--Put here the name of this state IN action (no quotes, it can be NULL) --></S_inAction>
      <S_outAction><!--Put here the name of this state OUT action state (no quotes, it can be NULL) --> </S_outAction>
      <S_Fsm><!--Put here the PATH to a submachine (this could be empty) --> </S_Fsm>
      <S_Credits><!--Put here initial execution credits (>=0, no quotes) --></S_Credits>
    </State>

  </States>
```

Estados de la *FSM* basada en estados concurrentes del documento *XML*

Por último, como consejo se recomienda sustituir los comentarios que aparezcan ente las etiquetas por los valores que se les quiera dar y no añadir etiquetas a dichos documentos, puesto que esto, aunque parezca inofensivo puede originar errores a la hora de hacer el *parseado*, siendo muy difíciles de localizar dichos errores. Para que pueda ver un ejemplo de cómo quedaría una plantilla *XML* rellenada se mostrará a

continuación una de las plantillas las cuales se pueden ver en el videojuego que se proporciona que hace uso de la *API*, el cual se ubica en la carpeta ***Game Example*** :

```
<FSM_Machine>
<!--FSM specification -->
<FSMtype Probabilistic="NO">CLASSIC</FSMtype>
<FSMid>FSM_Determinista</FSMid>

<Fsm>
<Callback>Events</Callback>

<States>

<State Initial="YES">
<S_Name>SPAWN</S_Name>
<S_Action>A_SPAWN</S_Action>
<S_inAction>NULL</S_inAction>
<S_outAction>NULL</S_outAction>
<S_Fsm></S_Fsm>
</State>

<State Initial="NO">
<S_Name>MOVE</S_Name>
<S_Action>A_MOVE</S_Action>
<S_inAction>NULL</S_inAction>
<S_outAction>NULL</S_outAction>
<S_Fsm></S_Fsm>
</State>

<State Initial="NO">
<S_Name>ATTACK</S_Name>
<S_Action>A_ATTACK</S_Action>
<S_inAction>NULL</S_inAction>
<S_outAction>NULL</S_outAction>
<S_Fsm></S_Fsm>
</State>


</States>

<Transitions>

<Transition>
<T_Name>MOVING</T_Name>
<T_Origin>SPAWN</T_Origin>
<T_Destination>MOVE</T_Destination>
<T_Action>NULL</T_Action>
<Events>
<Event>
<ID>E_MOVE</ID>
<Type>BASIC</Type>
</Event>
</Events>
</Transition>

<Transition>
<T_Name>MOVING</T_Name>
<T_Origin>ATTACK</T_Origin>
<T_Destination>MOVE</T_Destination>
<T_Action>NULL</T_Action>
<Events>
<Event>
```

```
<ID>E_MOVE</ID>
<Type>BASIC</Type>
</Event>
</Events>
</Transition>

<Transition>
<T_Name>ATTACKING</T_Name>
<T_Origin>MOVE</T_Origin>
<T_Destination>ATTACK</T_Destination>
<T_Action>NULL</T_Action>
<Events>
<Event>
<ID>E_ATTACK</ID>
<Type>BASIC</Type>
</Event>
</Events>
</Transition>

</Transitions>
</Fsm>
</FSM_Machine>
```

Ejemplo de plantilla *XML* rellenada

Una vez se hayan rellenado los apartados del documento *XML* de la *FSM* elegida, lo que se hará será rellenar la clase ***Tags*** ubicada en la carpeta ***User Code*.** En dicha clase lo que se hará será inicializar las variables que representan los componentes de dicha *FSM* y que se han creado en el paso anterior, es decir, los estados, acciones, transiciones y eventos. Dicho valor entero no puede estar repetido si la variable que ya tenía dicho valor pertenece al mismo campo que la variable declarada, es decir, que si se están definiendo las variables que representarán los estados no se les puede dar el mismo valor a dos estados. Para que no se parta de una clase vacía se le proporcionaran algunos estados, transiciones, eventos y acciones tanto definidas como inicializadas, tal y como puede verse la siguiente figura, para que así tenga unos ejemplos, así como unas etiquetas que no deberán de borrarse pues son las que identifican al tipo de *FSM* que se está empleando.

```
//FSM type TAGS (DO NOT change or delete)
public const int CLASSIC            = 0;
public const int INERTIAL           = 1;
public const int STACK_BASED        = 2;
public const int CONCURRENT_STATES  = 3;
public const int UNKNOWN            = 1000;

//State tags
public const int SPAWN = 0;
public const int MOVE = 1;
public const int ATTACK = 3;

//Transition tags
public const int MOVING = 0;
public const int ATTACKING = 1;
```

Variables de la clase *Tags*

Por último, en esta clase se tendrá que completar un método ***StringToTag(string Word)*** el cual implementa un *switch-case* en el que se comprueba la primera letra que compone el *String Word* ,que recibe como parámetro, y busca en el *switch* el case que corresponda con dicha letra. Una vez se ha encontrado la letra, se procede a buscar a ver si coincide con alguna de las palabras definidas y si es así, devuelve el valor de esta variable. Para ello lo que tendrá que hacer es añadir la variable que haya inicializado, para que se pueda encontrar al comparar con la palabra actual, así como devolver su valor.

```
public static int StringToTag (string word)
    {
        switch (word[0]) {
        case 'A':
            if(word.Equals("ATTACK")) return Tags.ATTACK;
            if (word.Equals("ATTACKING")) return Tags.ATTACKING;
            if (word.Equals("A_ATTACK")) return Tags.A_ATTACK;
            if (word.Equals("A_MOVE")) return Tags.A_MOVE;
            if (word.Equals("A_SPAWN")) return Tags.A_SPAWN;
            break;
        case 'B':
            break;
```

Método *StringToTag(String Word)* de la clase *Tags*

Tanto para los documentos *XML* como para la clase *Tags,* se recomienda que al dar un nombre a las variables que vayan a representar uno de los componentes de las *FSM* se haga en mayúsculas para así poder evitar errores al hacer *maching* por haber puesto una letra en mayúscula cuando era minúscula.

Una vez se hayan realizado estos 2 procesos de rellenado, ya se tendrá la *FSM* lista para poder hacer uso de ella, y para hacerlo de una forma más cómoda lo que se recomienda es que coja la clase ***ModelUsingTools***, ubicada en la carpeta ***User Code***, la cual contiene todas las llamadas a métodos necesarias, y sea este script el que asocie al *NPC* o *boss* que quiera hacer uso de la *API*. En esta clase, además de aparecer junto con cada método una pequeña explicación, se le especificará cuales de dichos métodos deberán ser modificados por usted y cuales no debe modificar. A continuación, se pasará a explicar en detalle la clase ***ModelUsingTools*** para que así se entienda el motivo de realizar las llamadas a métodos que se realiza, el orden de dichas llamadas y el por qué es necesario crear algunos métodos.

La primera acción que se realiza es importar la librería *GAI* para así poder hacer uso tanto de los métodos contenidos en la clase ***FSM_Manager*** y **FSM_Parser** como las variables inicializadas en la clase ***Tags***. Para importar librerías o carpetas en Unity se utiliza la directiva *using*.

```
using GAI; // In order to use all the methods and definitions made on the classes
           // that form part of the API you have to import the GAI
```

Librerías necesarias en la clase *Model*

Seguidamente se recomienda definir un par de variables para tener el código más estructurado y así sea más fácil de entender las acciones que se están realizando. Estas variables tal y como se puede ver en la siguiente figura son; una lista llamada ***DoActions,*** la cual se utilizará para almacenar las acciones de los estados que se tengan que realizar, una lista llamada ***EventsList,*** la cual almacenará la lista de eventos que se cumplan, una variable de tipo ***FSM_Machine,*** la cual servirá para almacenar la *FSM* que se elija, y por último una variable de tipo ***FSM_Manager*** que servirá para almacenar el manejador de *FSM*.

```
//////////////////////////// Variables of GAI - GAME ARTIFICIAL INTELIGENCE /////

/// DO NOT MODIFY THIS VARIABLES ///
// List of accions to be done
List<int> DoActions;

// Lis of events
List<int> EventsList;

// FSM Machine that you are going to use
FSM_Machine FSM;

// Manager of FSM Machines
FSM_Manager fsm_manager;

//////////////////////////////////////////////////////////////////////////////////
```

Variables definidas en la clase *ModelUsingTools*

Una vez se tengan estas variables definidas, el primer paso que sé tendrá que realizar será inicializarlas y crear la máquina de estados, y para eso en un método ***Start()*** se llevaran a cabo dichas acciones. El método *Start* es propio de Unity, y lo que asegura es que la llamada a este método se realizará una única vez, y al principio de la ejecución del programa, garantizando así que lo que se ponga en este método será lo primero a ejecutar. En dicho método lo primero que se realiza es la inicialización de tanto la lista de eventos, como el *FSM_Parser* y el *FSM_Manager* pasándole como argumento el *parser* inicializado anteriormente. Una vez se haya realizado este paso, ya se podrá añadir el *FSM* que se haya creado mediante el documento *XML*, acción que se puede realizar llamando al método *AddFSM()* y pasando como argumento de dicho método la ruta donde se encuentra el documento *XML* que contiene la *FSM*. Cuando se haya realizado esto ya se podrá crear la *FSM* llamando al método *CreateMachine(),* pasándole como argumentos el objeto que hará uso de la *FSM(*si el objeto que va a hacer uso de ella es la propia clase se puede emplear *this*), el tag del tipo de la *FSM* y por último el *FSMid* de dicha *FSM* contenido en el documento *XML*. Este método no deberá ser modificado por el usuario. El contenido del método explicado anteriormente se puede observar en la siguiente imagen:

```
/// DO NOT MODIFY THIS METHOD ///
// At the beginning what we will do is to initialize the variables defined before and call the FSM_Parser
//in order to use de FSM created in the XML documents as wll as create the FSM of the type we want
private void Start()
{

    // We initialize the list of events which will be used to see the events that had happened
    //and depending on them we will add actions to be done on the list Doactions.
    EventsList = new List<int>();

    // We create the Parser
    FSM_Parser parser = new FSM_Parser();

    // We create the Manager of the FSM using the parser created before
    fsm_manager = new FSM_Manager(parser);

    // We add  the manager to the FSM created on the XML Document.
    // To do that you have to use the path where the XML of the FSM is.
    fsm_manager.AddFSM("Assets/GAI/XML Documents/NameOfTheFSMYouHaveCreate.xml");

    // We create the FSM of the type you have chosen using the XML. You have to put the object that
   // is going to use the FSM, on the Tags.the tag of the FSM you are using (CLASSIC,INERTIAL,STACK_BASED or CONCURRENT_STATES)
   // and then the name of the FSM that you have put on the XML Document in the paragraph FSMid
    this.FSM = this.fsm_manager.CreateMachine(this, Tags.CLASSIC, "NameOfTheFSMYouHavePutOnTheFSMid");
}
```

Método *Start*() de la clase *ModelUsingTools*

El siguiente método implementado se llama ***WaitForAction()*** y tal y como se puede ver en la siguiente imagen, en él lo que se hará será ir actualizando la *FSM* para ver si hay acciones que se tengan que realizar. Si hay acciones, se añadirán a las lista *DoActions(),* se recorrerá dicha lista y se llamará al método *ExecuteAction()* para que ejecute la acción que corresponda.
Para actualizar la *FSM* lo que se hará será realizar una llamada al método *UpdateFSM()* el cual realiza diversas acciones sobre la *FSM* para llevar acabo la lógica del tipo de *FSM* seleccionada, pero la acción más importante que deberá tener en cuenta será una llamada interna a un *callback* el nombre del cual se proporcionó en el documento *XML*. Este *callback* lo que hará será, a partir de los eventos asociados a las transiciones, elegir que acciones se tienen que ejecutar, es decir, a partir de la lista de eventos creada (*EventsList*) se obtendrá la lista de acciones (*DoActions*) a realizar.

```
/// DO NOT MODIFY THIS METHOD ///
/// IF ANOTHER CLASS WANTS TO USE THE ACTIONS OF THE STATES OF THE FSM
/// USE THIS METHOD TO BE CALLED ///
// This method is going to be called in order to actualize
// the list of actions, and if there is an action to be done
// we will call a method called ExecuteAction(actiontobedone)
public void WaitForAction()
{
    // UpdateFSM() is a method from the clas FSM_Manager where there are a lot of internal calls.
    // One of the most important is a callback you have defined in the XML document of the FSM you are using. in which from
    // the list of events we have the list of actions to be done. This callback has to be in the code where the API is used.
    DoActions = FSM.UpdateFSM();

    // We go through the list of actions and if we have an action to be done we called
    // the method ExecuteAction(actiontobedone) so that this action is done.
    foreach (int action in DoActions)
    {
        if (action != Tags.UNKNOWN) ExecuteAction(action);
    }
}
```

Método *WaitForAction()* de la clase *ModelUsingTools*

El siguiente método será el *callback* mencionado anteriormente, el cual tendrá que tener el mismo nombre que se haya puesto en el documento *XML* de la *FSM* que se vaya a emplear. En este método lo que se hará será realizar un *switch-case* en el que se comprueba el tipo de *FSM* que se esté empleando, y dependiendo de las condiciones que se cumplan se añadirá el evento que se desee a la lista de eventos. Esta acción se realiza debido a que a partir de la lista de eventos la clase *FSM_Machine* se obtiene la

lista de acciones a realizar. Las comprobaciones que se realicen serán las que provoquen que, en caso de cumplirse, se lleve a cabo las acciones asociadas al estado al que se pase. Es por ello por lo que se recomienda, evitar en la medida de lo posible que varios eventos se puedan dar a la vez, ya que el primero que se añada a la lista de eventos será el que se ejecute, y si continuamente se está cumpliendo dicho evento no se saldrá del estado por mucho que ocurran cambios en el videojuego. Para evitar esto se recomienda el uso de variables que activen otros estados o por ejemplo si se está comprobando la distancia del jugador con el ente que emplee la *API*, establecer rangos de distancia para que no se quede continuamente en un estado como se ha comentado. Además de esto se recomienda que si uno de los estados va a ser el que más se emplee y los otros sean muy puntuales, como podría ser un estado de movimiento y otro de ataque, ya que el de ataque solo se efectuará en ciertas ocasiones y se estará más en el estado de movimiento, sea la condición del evento de este estado puntual la que se compruebe antes que la del estado más empleado, dejando dicha comprobación como ultima a evaluar.

```
/// MODIFY THIS METHOD ///
/// YOU WILL HAVE TO ADD THE EVENTS YOU HAVE CREATED AS WELL AS THE CONDITIONS IN ORDER TO PASS FROM ONE STATE TO OTHER ///
// We check the type of FSM and depending on the conditions we will add the Tag from the appropiate event to the EventsList.
// You will have some examples to get ideas.
public List<int> NameOfTheCallBackYouPutOntheXMLDocument()
{

    EventsList.Clear();
    //MAKE A SWITCH of events depending of the FSM
    switch (this.FSM.getFSM().getTag())
    {
        case Tags.CLASSIC:
            // Probabilistic
            if (this.FSM.getFSM().isProbabilistic())
            {
                // if (HaveMOVE == true && Vector3.Distance(this.transform.position, GameObject.Find("Player").transform.pos
                // if (HaveSpawn == true) EventsList.Add(Tags.E_MOVE);

            }
            // Deterministic
            else
            {

            }
            break;
        case Tags.INERTIAL:
            ///
```

Callback de la clase *ModelUsingTools*

Para finalizar, el ultimo método de esta clase será el método mencionado anteriormente llamado ***DoActions(),*** el cual se encargará de recibir las acciones que se tengan que realizar, y mediante un *switch*- case buscará la acción con dicho nombre e invocará el método encargado de llevar a cabo dicha acción. Se recomienda emplear métodos y no poner el contenido de los métodos en este apartado para una mejor implementación, así como organización. El contenido de dicho método se puede visualizar a continuación:

```
/// MODIFY THIS METHOD ///
/// YOU WIL HAVE TO PUT THE NAME OF THE ACTIONS YOU PUT IN CLASS TAGS AS WELL AS THE METHODS THAT WILL BE EXECUTED///
// You will have some examples to get ideas.
//Method that executes an action.
public void ExecuteAction(int action)
{
    switch (action)
    {
        case Tags.A_SPAWN:
           // SPAWN();
            break;
        case Tags.A_MOVE:
         //  MOVE()
            break;
        case Tags.A_ATTACK:
            // ATTACK()
            break;
            ///
        default: break;
    }
}
```

Método *ExecuteAction(action)* de la clase *ModelUsingTools*

Una vez se hayan hecho todos estos pasos ya se tendrá una clase la cual hace uso de la *API*, en concreto empleando la *FSM* que haya creado, con la lógica elegida, pero con el nombre de los estados, transiciones, acciones y eventos que desee. Pese a que parezcan muchos pasos a seguir en última instancia lo único que se realizará será rellenar una plantilla dando los nombres que desee, rellenar una clase con los mismos nombres dándoles un valor y por último poner las condiciones a los eventos y e invocar los métodos que lleven a cabo dichas acciones. Además, para evitarle el tener que crear los métodos de nuevo lo que se recomienda, tal y como se ha mencionado anteriormente es simplemente usar esta plantilla como script de comportamiento el cual se asocio a la entidad que quiera hacer uso de la *API*.

3. Utilización de la API sin las herramientas

Debido a que, si no se utilizan ni los documentos *XML* ni la clase *FSM_Parser*, se tendrán que realizar todas las llamadas internas para crear cada uno de los elementos del *FSM* elegido, y como consecuencia se tendrán que hacer muchos pasos y puede ser bastante fácil cometer un fallo se le proporciona un modelo de ejemplo en la clase ***ModelWithoutUsingTools*** de modo que solo tenga que copiar el esquema proporcionado para crear cada uno de los componentes de un *FSM* y rellenarlo con los valores correspondientes.

Para empezar las variables que se necesitaran serán las mismas que fueron empleadas en el apartado anterior y que, excepto si tampoco se hace uso de la clase *Tags* en cuyo caso en este apartado se añadirán las variables que se añadirían a dicha clase, dándoles un valor entero y recordando que, si pertenecen al mismo tipo, es decir si estamos definiendo los estados, no se les debe dar el mismo valor.

Pese a que la forma de hacer uso de la *API* será idéntica si se hace uso o no de la clase Tags, ya que simplemente se tendrá que definir las variables en el apartado de variables

de esta clase, como de los documentos *XML*, el método *Start()* se verá modificado completamente, ya que es aquí donde se realiza la creación del *FSM* que desee emplear.

Para empezar lo primero que tendrá que hacer es crear un objeto de tipo ***FSM_Manager***, inicializarlo ,y a continuación crear un objeto del tipo de la *FSM* que se desee y llamar al método que corresponda para así crear la *FSM* de dicho tipo, dándole las variables que sean necesarias. Cada *FSM* necesitará unas variables y su método tendrá un nombre diferente, razón por la que se buscaron y se pusieron todos los métodos posibles para que simplemente vea el método de la FSM que haya elegido, lo copie y lo rellene con los parámetros oportunos, tal y como puede verse en la imagen siguiente:

```
private void Start()
{
    ////// To create the FSM //////////////

    // We create an object of the type FSM_Manager and we initialize it
    FSM_Manager fsm_manager = new FSM_Manager();
    // We create an object of the type of FSM we want to use(FA_Classic,FA_Inertial,FA_Concurrent_States,FA_Stack)
    FA_Classic fsm;
    // We initialize this FSM with the variables that are needed
    // Deterministic --> FA_Classic(string ID, int tag, string CallbackName, bool FlagProbabilistic) FlagProbabilistic = false
    // Probabilistic --> FA_Classic(string ID, int tag, string CallbackName, bool FlagProbabilistic) FlagProbabilistic = false
    // Inertial --> FA_Inertial(string ID, int tag, string CallbackName, bool FlagProbability)
    // Stack --> FA_Stack(string ID, int tag, string CallbackName, bool FlagProbability)
    // ConcurrentStates --> FA_Concurrent_States(string ID, int tag, int num, string CallbackName, bool FlagProbability)
    fsm = new FA_Classic("NameOfTheFSM", Tags.CLASSIC, "Name of the callbacks", false);
    ////////////////////////////////////////
```

Método *Start()* de la clase *ModelWithoutUsingTools* (1)

A continuacion se procederá con la creacion de los estados en los que, nuevamente, deberá crear un objeto de tipo ***State***, inicializar dicho estado llamando al metodo que corresponda según el tipo de *FSM* elegido, y por ultimo añadirlo al *fsm*(objeto del tipo de la *FSM* que se haya elegido). Para que no tenga que buscar que metodo le corresponde a cada tipo de FSM se han puesto todos los metodos como puede verse en la siguiente imagen:

```
///////// To create the STATES of the FSM /////////////////
// We create an object of type State
State aux;
// We initialize this State with the variables that needs
// Deterministic --> State(string ID, string FlagInitial, int state_tag, int action_tag, int in_action_TAG, int out_action_TAG)
// Probabilistic --> State(string ID, string FlagInitial, int state_tag, int action_tag, int in_action_TAG, int out_action_TAG)
// Inertial --> State(string ID, string FlagInitial, int state_tag, int action_tag, int in_action_TAG, int out_action_TAG, int latency)
// Stack --> State(string ID, string FlagInitial, int state_tag, int action_tag, int in_action_TAG, int out_action_TAG, uint priority)
// ConcurrentStates --> State(string ID, string FlagInitial, int state_tag, int action_tag, int in_action_TAG, int out_action_TAG, short credits)
aux = new State("State ID", "Initial (YES or NO)", Tags.StringToTag("ID tag"), Tags.StringToTag("StateAction(it can be NULL)"), Tags.StringToTag("StateINAction(it can be NULL)"), Tags.StringToTag("StateOu
// We add the state created to the FSM
fsm.addState(aux);
// If you want to create more states you just have to copy the same structure
////////////////////////////////////////////////////////////
```

Método *Start()* de la clase *ModelWithoutUsingTools* (2)

Seguidamente se continuará con la creación de las transiciones que conecten los estados y, para ello, lo primero que deberá hacer es crear un objeto de tipo *Transition* y dos objetos de tipo *State,* que representaran los estados que se conectaran entre sí. Además, se creará una lista para contener los eventos asociados a dicha transición. Para obtener los estados que se vayan a conectar se empleará la función *getStateByID()* la cual obtendrá el estado que se desee conectar a partir del nombre de dicho estado. Seguidamente se añadirá el evento o eventos de dicha transición a la lista de eventos, y se creará la transición, la cual dependiendo de la *FSM* tendrá que recibir más o menos

parámetros. Por último, se añadirá al estado origen y a la *fsm* creada dicha transición, para que así se pueda realizar esta conexión entre los estados. Al igual que como ocurría anteriormente debido a que las transiciones cambian dependiendo del *FSM* a emplear se pusieron en comentarios el método a llamar dependiendo de la *FSM* a emplear, tal y como puede verse en la siguiente imagen:

```
////////// To create the TRANSITIONS of the FMS ////////////////
// We create an object Transition
Transition trans;
// We create the origin State and the destination State
State origin, destination;
// We create an object List<FSM_Events> in order to Add the events that the transition will have
List<FSM_Event> ListOfEvents = new List<FSM_Event>();
// We get the states we want to conect
origin = fsm.getStateByID("NameOfTheStateOrigin");
destination = fsm.getStateByID("NameOfTheStateDestination");
// We create the Event that activate this transition and we add it to the EventsList(we can create more)
ListOfEvents.Add(new FSM_Event("NameOfTheEvent",Tags.StringToTag("NameOfTheEvent"), "TypeOfTheEvent(BASIC or STACKABLE"));
// We create the Transition with the variables that needs
// Deterministic --> Transition(string ID, State A, State B, int transition_tag, int action_tag, List<FSM_Event> EventsList)
// Probabilistic --> Transition(string ID, State A, State B, int transition_tag, int action_tag,  List<FSM_Event> EventsList, int probability)
// Inertial --> Transition(string ID, State A, State B, int transition_tag, int action_tag, List<FSM_Event> EventsList)
// Stack --> Transition(string ID, State A, State B, int transition_tag, int action_tag, List<FSM_Event> EventsList)
// ConcurrentStates --> Transition(string ID, State A, State B, int transition_tag, int action_tag, List<FSM_Event> EventsList)
trans = new Transition("Transition ID",origin,destination,Tags.StringToTag("Transtion ID"), Tags.StringToTag("TransitionAction(it can be NULL)"), ListOfEvents);
// We add the transition to the origin State
origin.addTransition(trans);
// We add the transition to the destination State
fsm.addTransition(trans);
// We load the event or events that activate this transition
ListOfEvents.Add(new FSM_Event(("NameOfTheEvent"),Tags.StringToTag("NameOfTheEvent"), "TypeOfTheEvent(BASIC or STACKABLE"));
// We clear the List of Events
ListOfEvents.Clear();
// If you want to create more transitions and connect them with other states you have to copy the same structure
```

Método *Start()* de la clase *ModelWithoutUsingTools* (3)

Por último, y para finalizar, lo último que tendrá que hacerse será llamar al método *Start()* del tipo de la *FSM* que se haya creado, para acabar de crear la *FSM*, así como añadir al *FSM_MANAGE*R la *fsm* creada y crear la *FSM,* la cual es de tipo *FSM_Machine*, del mismo modo que se hacía empleando las herramientas en el apartado anterior.

```
////////////////// To add the FSM  to the manager and be able make use of it ////////////////////////////////

// We call the method Start from the class fsm in order to make the operations that are needed to make the FSM.
fsm.Start();


// We add the fsm to the FSM_Manager
fsm_manager.AddFSM(fsm);

// We add to the FSM_Machine created the final fsm created //
this.FSM = fsm_manager.CreateMachine(this, Tags.CLASSIC, "FSM_Determinista");

////////////////////////////////////////////////////////////////////////////////////////////////////////////
```

Método *Start()* de la clase *ModelWithoutUsingTools* (4)

A partir de este momento ya se tendrá la FSM lista para emplear y los pasos a seguir para poder hacer uso de ella serán los mismos que los que se han visto en el apartado anterior.

4. ¿Por qué emplear las herramientas?

Para finalizar, y a modo de resumen final, se hará una pequeña explicación, así como valoración para que así pueda decidir por que emplear las herramientas y evitar a toda costa el tener que crear la *FSM* a mano.

Pese a que pueda parecer que el emplear las herramientas puede ser más extensa que sin hacer uso de ellas y por tanto es una mejor opción esto no es del todo así. Para empezar el emplear las herramientas, aunque se hayan proporcionado los métodos y un modelo para poder emplear la *API*, tendrá que ser usted el que proporcione los parámetros de dicho método. Al haber creado tantas variables, tener que crear cada estado y cada transición, así como todos los eventos, es muy fácil que se acabe obteniendo un trozo de código bastante extenso en el que todo es bastante similar. Esto tiene como peligro que se realizan mal las asignaciones, que se creen transiciones duplicadas o conexiones que no se querrían realizar, teniendo como resultado un final un comportamiento que no se deseaba en la entidad que haga uso de la API.

Todo esto se puede evitar haciendo uso de las herramientas puesto que el código de la clase tendrá la misma longitud, simplemente se aumentarán los apartados del documento XML, siendo más fácil de visualizar si se ha cometido un error, así como teniendo que preocuparte menos ya que las asignaciones de los parámetros de los métodos, así como la llamada a dichos métodos se harán de forma trasparente al usuario, mitigando muchísimo más la posible aparición de errores.

Además, para concluir, pese a que al hacer uso de ambas herramientas es posible obtener errores, por una mala asignación o por crear conexiones o estados que no se deberían, es mucho más fácil poder encontrar el error en un documento en el que solo se han dado nombres y todo está organizado en etiquetas, que un trozo mucho más largo de código en el que hay llamadas muchos métodos, y muchas variables, las cuales se tendría que ir comprobando una por una si tienen el valor correcto o si ese parámetro es el que tocaba pasarle a ese método,