PROGRAMACION DE JUEGOS EN UNITY

Introducción

El objetivo de la práctica es la aplicación de los conceptos del paradigma de programación orientado a objetos para **programar un videojuego en 2D utilizando un motor gráfico para crearlo y desarrollarlo**.

El motor gráfico del juego es el elemento más importante de todo proyecto de videojuego ya que es la herramienta que proporciona facilidades al usuario a la hora de crear sus propios juegos, facilitando la integración de gráficos, así como su uso y modificación, la introducción de sonidos en el juego, IA, etc.

La funcionalidad básica de un motor es proveer al videojuego de un motor de renderizado ("render") para gráficos 2D y 3D, un motor físico que detecte la colisión de objetos y la respuesta a dicha colisión, sonidos y música, scripting, animación, inteligencia artificial, comunicación con la red para juegos multijugador, streaming, administración de memoria y un escenario gráfico, de forma que el desarrollador sólo tiene que centrarse en la mecánica del juego y no en la forma en la se representa, haciendo posible que el desarrollador pueda crear un mundo dentro de juego combinando las capacidades del motor con los modelos que hayan podido crear los artistas responsables.

Los motores gráficos de juego permiten a los diseñadores controlar la lógica del juego utilizando lenguajes de programación de alto nivel o lenguajes de scripts, lo que disminuye la curva de aprendizaje y mejora el flujo de trabajo. Hoy en día existen una gran variedad de motores completos y motores gráficos como OGRE 3D, Doom Engine, Quake Engine, y GoldSrc, Source, BLAM! Engine, Source Engine, Unreal Engine, Unity, etc. Las capacidades gráficas del motor gráfico son una de las claves para su elección, destacando motores gráficos como CryEngine, pero también es importante la facilidad de desarrollo y la plataforma para la que se va a desarrollar.

Aunque hay una gran variedad de motores de juegos, Unity es uno de los pocos que proporciona un gran número de herramientas y técnicas que simplifican el proceso de desarrollo, facilitando la creación de juegos de alta calidad gracias a que permite abordar muchos aspectos del desarrollo del juego, incluyendo un potente entorno integrado de desarrollo (IDE), inteligencia artificial (AI), animaciones o iluminación.

Motor Gráfico Unity

El videojuego será programado utilizando **Unity** (https://unity.com/es), un **motor gráfico de videojuegos** multiplataforma empaquetado como una herramienta para crear juegos, aplicaciones interactivas, visualizaciones y animaciones en 2D y 3D creado por Unity Technologies que está disponible como plataforma de desarrollo para Windows, OS X y Linux.

Esta herramienta nos permite, mediante un potente editor visual y scripting, desarrollar videojuegos con un acabado profesional de forma fácil y sencilla para múltiples plataformas como Windows, OS X, Linux, Xbox, PlayStation, Wii, iPad, iPhone, Android y Windows Phone. El motor también puede publicar juegos basados en web (videojuegos de navegador) para Windows y Mac usando el plugin Unity web player.

El contenido del juego es construido desde el editor y el gameplay se programa usando un lenguaje de scripts. Unity tiene integrado un editor visual muy útil y completo que permite con unos pocos clicks importar modelos 2D y 3D, texturas, sonidos, etc. para después trabajar con ellos. Además incluye la herramienta de desarrollo MonoDevelop con la que se puede crear scripts en JavaScript, C# y Boo (un dialecto de Python) con los que extender la funcionalidad del editor, utilizando las API que provee y la cual se encuentra documentada junto a tutoriales y recursos en su web oficial.

Aunque Unity proporciona una gran versatilidad (en cuanto a la mecánica del videojuego, las físicas y detección de colisiones, algunos elementos gráficos como la iluminación, texturas, etc.), no es un programa que proporcione funcionalidades de modelado de objetos. Por ello los modelos que se utilizan en un videojuego Unity normalmente se importan desde otros programas de modelado y animación. A modo de ejemplo, Unity puede usarse junto con 3D Studio Max, Maya, Softimage, Blender, Modo, ZBrush, Cinema 4D, Cheetah3D, Adobe Photoshop, Adobe Fireworks y Allegorithmic Substance, de forma que los cambios realizados a los objetos creados con estos productos se actualizan automáticamente en todas las instancias de ese objeto durante todo el proyecto sin necesidad de volver a importar manualmente.

El motor gráfico utiliza Direct3D (en Windows), OpenGL (en Mac y Linux), OpenGL ES (en Android y iOS), e interfaces propietarias (Wii). Tiene soporte para mapeado de relieve, reflexión de mapeado, mapeado por paralaje, sombras dinámicas con mapas de sombras, render a textura y efectos de post-procesamiento de pantalla completa.

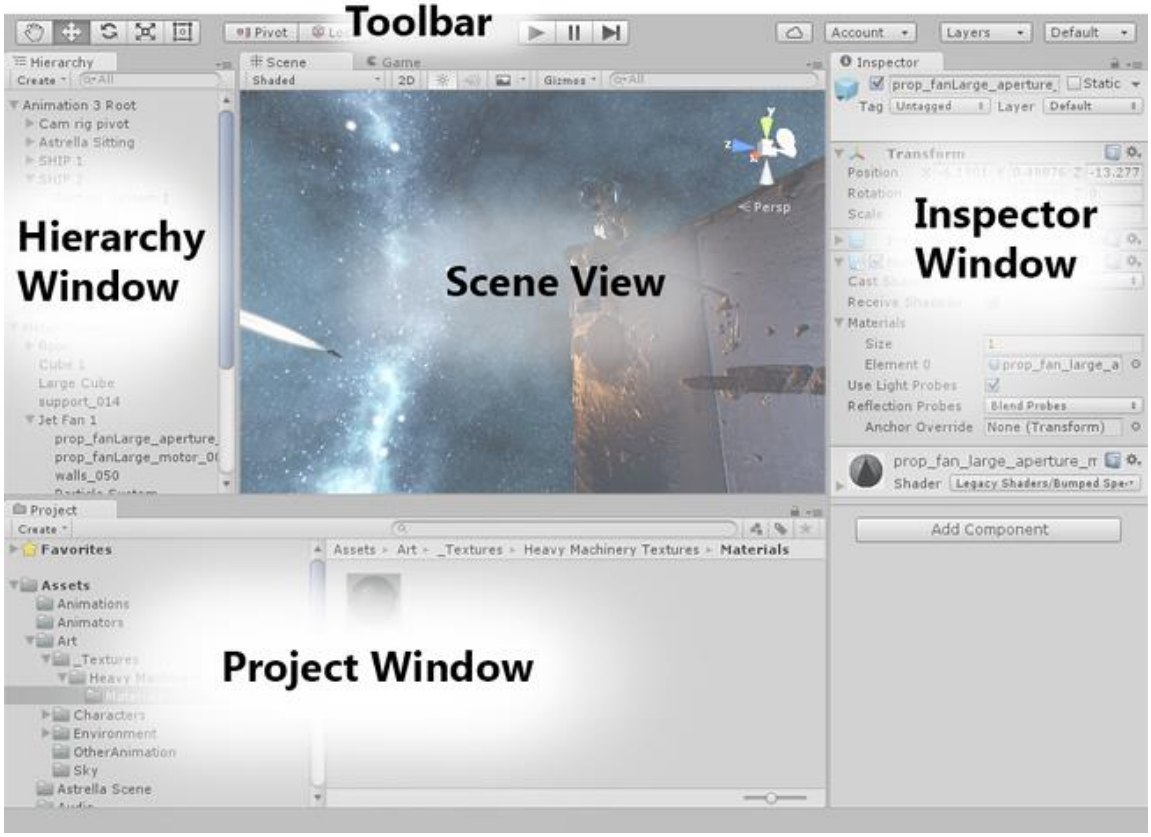
Además, si no tenemos tiempo o necesitamos recursos para nuestros juegos, en la propia aplicación podemos acceder a una tienda, donde encontraremos multitud de recursos gratuitos y de pago que podemos descargarnos. Incluso podremos extender la herramienta mediante plugins que obtendremos en dicha tienda. El Unity Asset Store (https:/assetstore.unity.com/) dispone de una extensa colección de paquetes de Assets en una amplia gama de categorías, texturas y materiales, sistemas de partículas, música y efectos de sonido, tutoriales y proyectos, paquetes de scripts, extensiones para el editor y servicios en línea.

Breve descripción de Unity

Debido a la potencia y complejidad de Unity, aprender toda su funcionalidad a nivel experto es bastante complicado de conseguir. No obstante, es un programa bastante intuitivo, por lo que empezar a desarrollador proyectos sencillos es una tarea fácil y cómoda. Además, en la web de Unity hay una sección dedicada exclusivamente al aprendizaje, donde se puede encontrar tutoriales y todo lo relacionado con la documentación a utilizar, dependiendo del lenguaje elegido para programar el videojuego (JavaScript, C# o Boo).

La ventana principal del editor está compuesta por varios paneles tipo pestañas conocido en Unity como Views. Hay varios tipos de Views preestablecidos en Unity (todas tienen funciones y funcionalidades específicas), y el usuario los puede modificar y recolocar como quiera, además de poder crear su propia View.

Los principales paneles se muestran en la siguiente figura:



La **Ventana de Proyecto** (Project Window) muestra los assets que pertenecen al proyecto (los assets que importamos aparecen aquí), incluyendo texturas, audio, otras escenas, prefabs, fuentes o scripts. Nuestro juego en realidad está formado por muchos ficheros diferentes que se encuentran guardados en alguna carpeta de nuestro ordenador. En este apartado podemos ver qué ficheros tiene nuestro

proyecto, y en qué carpeta se encuentran guardados. Si queremos utilizar algo en nuestro juego, lo primero de lo que tenemos que asegurarnos es que esté en alguna de las carpetas que podemos ver en esta ventana. El panel izquierdo muestra la estructura de carpetas del proyecto como una lista de jerarquía, mientras que en el panel derecho se muestra el contenido de la carpeta del panel izquierdo seleccionada (los assets individuales son mostrados como iconos que indican su tipo: script, material, sub-carpeta, etc.)

Un asset es una representación de cualquier ítem que puede ser utilizado en su juego o proyecto. Puede ser tanto un archivo creado fuera de Unity (un archivo de audio, una imagen, o cualquiera de los otros tipos de archivos que Unity soporta) como uno creado dentro de Unity (tal como un Animator Controller, un Audio Mixer o una Render Texture).

Aunque Unity trae incorporado una colección de assets estándar que son utilizados frecuentemente (2D, Cámaras, Personajes, CrossPlatformInput, Efectos, Entorno, ParticleSystems, Prototyping, Utilidad, Vehículos, etc.), para facilitar el desarrollo de videojuegos Unity proporciona un Asset Store, donde hay una gran cantidad de assets comerciales y gratuitos disponibles, desde texturas, modelos y animaciones hasta ejemplos de proyectos completos, tutoriales y extensiones del editor. Estos assets son accesibles desde una interfaz simple dentro del Editor Unity y son descargados e importados directamente en sus proyectos.

La **Barra de Herramientas** (Toolbar) proporciona acceso a las características más esenciales del trabajo y contiene las herramientas básicas para la manipulación de la vista escena y de los objetos que hay en ella, los botones para reproducir y ejecutar el juego paso a paso y los botones de visibilidad de capas (layers) que permiten renderizar e iluminar sólo una parte de la escena (la contenida en la capa seleccionada).

La **Vista Escena** (Scene View) nos permite navegar y editar la escena, la cual puede ser mostrada en perspectiva 2D o 3D, dependiendo del tipo de proyecto que creemos. La utilizaremos para seleccionar y posicionar todos los *GameObjects* (jugador, cámara, enemigos, etc.) que componen el juego. Las escenas son equivalentes a los niveles del juego e incluye todos los objetos relevantes y el entorno.

La **Ventana de Jerarquía** (Hierarchy Window) es una lista jerárquica de todos los objetos presentes en la escena (contiene cada *GameObject* de la escena actual), de forma que cada vez que añadimos/eliminamos un objeto de la escena se añade/elimina también de la jerarquía. La jerarquía nos permite conocer cómo los objetos están relacionados con respecto a los otros (parentesco). Cuando un objeto lo hacemos hijo de otro en la ventana jerarquía, hereda el movimiento y rotación de su padre, de forma que cuando el padre rota o se desplaza el hijo lo hace automáticamente también.

La **Ventana de Inspector** (Inspector Window) permite ver y editar todas las propiedades del objeto actualmente seleccionado, así como preferencias y ajustes dentro de Unity.

Cuando se selecciona un *GameObject* en la Ventana de Jerarquía o en la Vista Escena, en el Inspector se muestra las propiedades (Properties) de todos los componentes (Components) y materiales (Materials) del objeto de forma que es posible editarlos y modificarlos. En los componentes de tipo Script, las variables públicas de ese script son mostradas en el Inspector y pueden ser vistas y editas como las propiedades de los componentes integrados de Unity, permitiendo establecer parámetros y valores por defecto en sus scripts fácilmente sin modificar el código.

Las propiedades de los componentes pueden ser en general categorizadas en referencias (punteros a objetos y assets) o valores (números, cajas de verificación, colores…). Las referencias son asignadas arrastrando el objeto o asset a la propiedad en el Inspector, mientras que los valores son editados usando cajas de texto, casillas de verificación y menús).

Cuando se quiere editar varios objetos a la vez, basta con seleccionarlos de forma simultánea: los componentes que tienen en común aparecerán en el Inspector y sus valores podrán ser visualizados y modificados.

Creación en Unity de Juegos en 2D

Cada juego que creemos estará contenido en un proyecto de Unity. Cada proyecto se divide en una o varias escenas: la pantalla de títulos, cada uno de los niveles... Iremos construyendo distintas escenas dependiendo del número de niveles y pantallas diferentes que compongan nuestro proyecto. Cada escena se guarda en archivos independientes. Cuando cargamos una escena todos los objetos de esta se cargan en el panel de la jerarquía.

En cada escena se puede crear objetos o importarlos desde otros programas de modelado. Cada escena se compone de *GameObject*: la cámara, las luces, los enemigos, el jugador… El *GameObject* es la unidad básica dentro del motor, por lo que es importante entender qué es y cómo puede ser usado.

Los *GameObject* son contenedores que no son capaces de hacer nada por sí mismos. Para que un *GameObject* tenga funcionalidad es necesario añadirle componentes. Cada componente le proporciona una funcionalidad distinta: física, movimiento, gráficos, daños…

Cada *GameObject* consta, por tanto, de una serie de componentes (*Components*), que se pueden añadir o quitar para caracterizar el objeto que representan (dependiendo del objeto que se quiera crear).

Una vez creado un *GameObject* concreto, tras añadirle sus componentes y propiedades, si pensamos reutilizarlo en la escena varias veces modificándolo ligeramente o utilizarlo en otros juegos, es posible almacenarlo en disco guardándolo como un Prefab, de forma que se puede instanciar el *GameObject* creado cuantas veces queramos arrastrándolo a la escena. Cualquier edición hecha a un Prefab asset

será inmediatamente reflejado en todas las instancias producidas por él, mientras que las modificaciones y componentes añadidos/eliminados en las instancias sólo afectan a cada una de ellas de forma individual (esto es útil cuando se quiere crear varios objetos similares, pero con pequeñas variaciones: se crea un Prefab que se instancia varias veces y en cada instancia se modifica, elimina o añade el componente deseado).

Los componentes añadidos a un *GameObject* se pueden modificar desde la interfaz de Unity, pero también se puede acceder a ellos desde código, para cambiar sus características en tiempo de ejecución si así se requiere.

Algunos de los componentes más importantes, que muchos de los *GameObjects* de un juego suelen tener son:

- ***Transform:*** Es el componente más básico, por lo que todos los *GameObject* lo tienen y no puede ser eliminado. Este componente determina la posición del *GameObject* en la escena, su rotación y su tamaño. Cuando un objeto es hijo de otro (parentesco) los valores de posición, rotación y escala de su *Transform* son relativos al *Transform* del objeto padre.

- ***MeshRenderer:*** Este componente se encarga de renderizar un objeto en tiempo de ejecución. Por ello todo objeto que queremos que se vea en una escena debe contar con este componente.

- ***RigidBody:*** Asocia al *GameObject* algunas características que tendría un cuerpo sólido en la realidad: masa, gravedad, fricción, etc. Es el componente principal que permite el comportamiento físico para un objeto, permitiendo que dicho objeto sea afectado por colisiones, gravedad y otras fuerzas. Es muy útil para hacer que objetos de la escena interaccionen entre sí.

- ***Colliders*** (*Colisionadores*)***:*** Son áreas que envuelven al GameObject, que son utilizadas para comprobar las colisiones entre los distintos objetos del juego. Los colliders definen la forma del objeto para los propósitos de colisiones físicas, son invisibles y no tienen por qué tener la misma forma exacta que el mesh del objeto (generalmente se usa una aproximación a dicha forma). Hay varios tipos de colisionadores:

  - ***BoxCollider:*** Colisionador de caja. Se crea un área con forma de cubo alrededor del objeto.
  - ***SphereCollider:*** Colisionador de esfera. Se crea un área con forma de esfera alrededor del objeto.
  - ***CapsuleCollider:*** Colisionador de cápsula. Se crea un área con forma de cápsula alrededor del objeto.
  - ***MeshCollider:*** Colisionador de malla. El colisionador se acopla a la malla del objeto al que se le asigna. Es una forma de obtener colisiones de manera muy precisa, pero consume muchos recursos.

  Los colliders pueden ser agregados a un objeto sin un componente Rigidbody para crear pisos, paredes y cualquier otro elemento sin movimiento en la

escena. Estos son referenciados como static colliders. Los Colliders en un objeto que tiene un Rigidbody son conocidos como dynamic colliders. Los Colliders estáticos pueden interactuar con los colliders dinámicos pero debido a que estos no tienen Rigidbody, no se moverán en respuesta a las colisiones.

- ***Physic Material*** (Material de Física)***:*** Cuando los colliders interactúan, sus superficies necesitan simular las propiedades del material que supuestamente representan (ej. un hielo será resbaloso mientras que una goma deberá ofrecer mucha fricción y mucho rebote). Aunque la figura de los colliders no es deformada durante las colisiones, su fricción y rebote puede ser configurado utilizando Physics Materials.

- ***CharacterController*** (Controladores de Personaje)***:*** Normalmente se le añade este componente tan sólo al personaje principal, que será controlado por el jugador. Se utiliza cuando queremos que el objeto al que va asociado no tenga un comportamiento físicamente realista, en el sentido de que queremos que las colisiones si le afecten, pero la aceleración, frenado y cambio de dirección sea casi instantáneo sin ser afectado por el momentum (cantidad de movimiento). Este componente le da al personaje un simple collider (colisionador) en forma de capsula que siempre está en posición vertical. El controlador tiene sus propias funciones especiales para establecer la velocidad y dirección del objeto, pero a diferencia de los verdaderos colliders, un rigidbody no es requerido y los efectos del momentum no son realistas.

- ***Animation:*** Todo *GameObject* que contenga animaciones debe tener este componente añadido. En él se especifican el número de animaciones y el rango de fotogramas que ocupa cada animación. Además, sirve de referencia a la hora de controlar dichas animaciones desde código.

  El sistema de animación de Unity, llamado Mecanim proporciona un flujo de trabajo y configuración de animaciones para todos los elementos de Unity, incluyendo objetos, personajes y propiedades, además de soportar clips de animación importadas y animaciones creadas desde el propio motor que permite animar diferentes partes del cuerpo con diferente lógica, indicando como debería cambiar su posición y rotación en el tiempo.

- ***AudioSource:*** Cualquier *GameObject* que emita sonidos debe tener este componente asociado. Funciona como un almacén de sonidos y proporciona opciones para la reproducción de dichos sonidos.

- ***AudioListener:*** Este componente actúa como un oído, capta los sonidos emitidos por los objetos que tienen asociado un componente de tipo *AudioSource* y los reproduce por el altavoz del dispositivo. En una escena sólo puede haber un único objeto que contenga este componente, normalmente es la cámara principal la que lo contiene.

- ***Script:*** Un *Script* es una pieza de código añadida como componente a un *GameObject*. Están explicados a continuación.

Ciclo de vida de un Script

La manera que tiene Unity de coordinar el código del videojuego con los modelos es mediante componentes de tipo *Script*. Un *Script* es una pieza de código que puede asignarse a uno o varios *GameObjects* como un componente más. Los Scripts pueden estar escritos en C#, JavaScript o Boo, siendo posible escribir unos en un lenguaje y otros en otro distinto, es decir, se puede crear dos Scripts en lenguajes distintos y asignárselos a un mismo *GameObject*, los dos funcionarán.

Unity proporciona un amplio conjunto de librerías de clases específicas y funciones principales para el desarrollo de juegos por lo que muchas de las tareas comunes en el desarrollo de videojuegos ya están implementadas. Estas funciones o métodos heredan de la clase principal de Unity: MonoBehaviour. La herencia de MonoBehaviuor en JavaScript está implícita, mientras que en C# y BOO se tiene que especificar para cada clase.

Estas son algunas de los métodos más importantes presentes en las clases que heredan de MonoBehaviour:

- **Start():** A este método se le llama sólo al inicio del juego. Es muy útil para inicializar variables, posiciones, estados, etc.

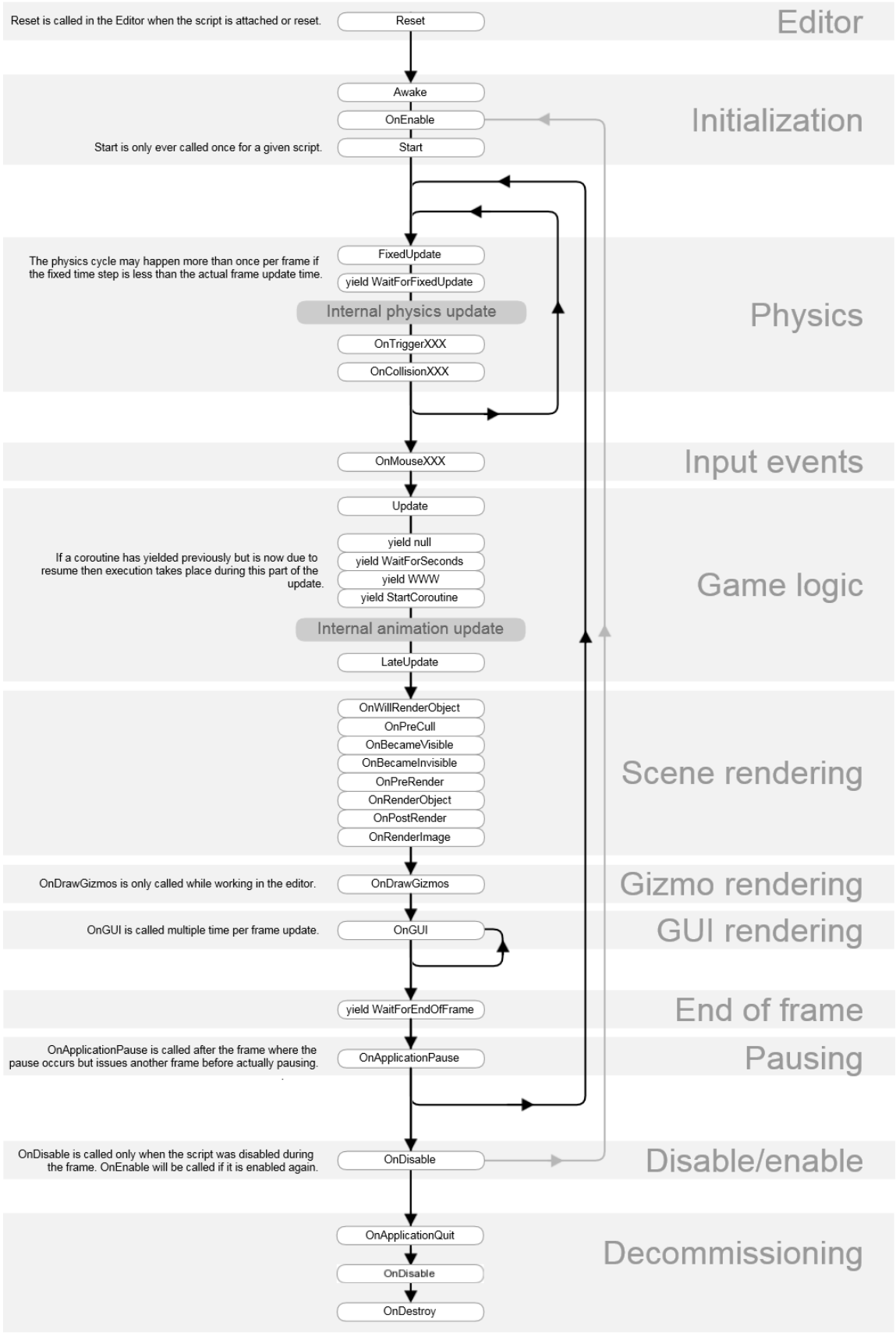
- **Awake():** Este método es llamado justo un fotograma antes que Start() y también justo después de que un prefab es instanciado (si un GameObject está inactivo durante el comienzo, Awake no es llamado hasta que se vuelva activo), por lo que también es práctico para inicializar variables, sobretodo variables que puedan tener conflictos de tiempo con Start().

- **Update():** Este método es llamado una vez por cada fotograma del juego, en tiempo de ejecución, antes de que el fotograma sea renderizado y las animaciones calculadas. Dentro de él se debe incluir todo el código que necesita actualizarse una vez por fotograma, como el movimiento o la detección de cambios. Por ello es el más importante a la hora de desarrollar el videojuego.

- **OnCollisionEnter():** A este método se le llama cada vez que el objeto que lleva este *Script* colisiona con otro objeto de la escena. Además existen los métodos OnCollisionStay() y OnCollisionExit() que se llaman mientras la colisión continúa y finaliza respectivamente.

- **OnTriggerEnter():** Un componente de tipo Collider puede ser marcado como *Trigger*. Esto significa que los objetos que colisionen con él no reaccionarán como si de una colisión real se tratase, sino que pasarán de largo. No obstante quedará registrado que un objeto ha entrado en ese área definida por el colisionador. Este método detecta esa intrusión en dicha área. Es muy útil para crear sistemas de detección. Al igual que antes OnTriggerStay() y OnTriggerExit() se llaman cuando la colisión sigue y finaliza.

Aparte de estos métodos, existen muchos otros. El siguiente diagrama resume el orden y repetición de funciones de evento durante la vida de script:



En la siguiente tabla se muestra una breve descripción de cada método:

Awake	Se llama cuando el script está siendo cargado.
FixedUpdate	Se llama cada cierto tiempo fijo (puede ser llamado varias veces por frame o puede no ser llamado entre frames, dependiendo de la velocidad del frame).
LateUpdate	Se llama una vez por frame, despues de que Update haya finalizado. Cualquier cálculo que sea realizado en Update sera completado cuando LateUpdate comience.
OnApplicationFocus	Sent to all game objects when the player gets or loses focus.
OnApplicationPause	Se envia a todos los game objects cuando el juego se pausa.
OnApplicationQuit	Se llama en todos los game objects antes de que se salga de la aplicación.
OnAudioFilterRead	If OnAudioFilterRead is implemented, Unity will insert a custom filter into the audio DSP chain.
OnBecameInvisible	Se llama cuando un objeto se vuelve invisible a cualquier cámara.
OnBecameVisible	Se llama cuando un objeto se vuelve visible.
OnCollisionEnter	Se llama cuando el Collider o Rigidbody entra en contacto con otro Collider o Rigidbody.
OnCollisionExit	Se llama cuando el contacto finaliza.
OnCollisionStay	Se llama una vez por frame mientras la colisión continua produciéndose.
OnConnectedToServer	Se llama cuando el cliente se conecta al servidor.
OnControllerColliderHit	OnControllerColliderHit is called when the controller hits a collider while performing a Move.
OnDestroy	Se llama cuando el objeto es destruido.
OnDisable	Se llama cuando el comportamiento se vuelve inactive o se desabilita.
OnDisconnectedFromServer	Se llama cuando la conexión se pierde o se desconecta del servidor.
OnDrawGizmos	Utilizado para dibujar Gizmos en la vista de escena por propósitos de visualización.

OnDrawGizmosSelected	Implement this OnDrawGizmosSelected if you want to draw gizmos only if the object is selected.
OnEnable	Se llama cuando el objeto activado.
OnFailedToConnect	Se llama cuando una conexión falla por alguna razón.
OnGUI	Llamado múltiples veces por frame en respuesta a los eventos GUI. El Layout y los eventos de Repaint son procesados primero, seguidos por un evento de Layout y de teclado/mouse para cada evento input.
OnJointBreak	Called when a joint attached to the same game object broke.
OnLevelWasLoaded	This function is called after a new level was loaded.
OnMasterServerEvent	Called on clients or servers when reporting events from the MasterServer.
OnMouseDown	Se llama cuando el usuario pulsa el botón del ratón mientras está sobre un GUIElement o Collider.
OnMouseDrag	Se llama cuando el usuario arrastra el ratón.
OnMouseEnter	Se llama cuando el mouse entra en un GUIElement o Collider.
OnMouseExit	Se llama cuando el raton sale.
OnMouseOver	Se llama en cada frame mientras el ratón está sobre el GUIElement o Collider.
OnMouseUp	Se llama cuando el usuario suelta el boton del ratón.
OnMouseUpAsButton	OnMouseUpAsButton is only called when the mouse is released over the same GUIElement or Collider as it was pressed.
OnNetworkInstantiate	Called on objects which have been network instantiated with Network.Instantiate.
OnParticleCollision	OnParticleCollision is called when a particle hits a collider.
OnPlayerConnected	Called on the server whenever a new player has successfully connected.
OnPlayerDisconnected	Called on the server whenever a player disconnected from the server.
OnPostRender	Se llama después de que una cámara finalice de renderizar la escena.

OnPreCull	Se llama antes de que la cámara corte (culls) la escena.
OnPreRender	OnPreRender se llama antes de que la cámara comience a renderizar la escena.
OnRenderImage	Se llama después de que toda la renderización de escena esté completa para renderizar la imagen de la pantalla.
OnRenderObject	Se llama después de que la cámara renderice la escena.
OnSerializeNetworkView	Used to customize synchronization of variables in a script watched by a network view.
OnServerInitialized	Called on the server whenever a Network.InitializeServer was invoked and has completed.
OnTransformChildrenChanged	Se llama cuando la lista de hijos del transform GameObject cambia.
OnTransformParentChanged	Se llama cuando la propiedad parent del transform del GameObject se cambia.
OnTriggerEnter	Se llama cuando un Collider marcado como Trigger entra en contacto con otro.
OnTriggerExit	Se llama cuando la colision finaliza
OnTriggerStay	Se llama una vez por frame mientras la colisión continua produciéndose.
OnValidate	Se llama cuando el script es cargado o un valor es cambiado en el inspector.
OnWillRenderObject	Se llama una vez por cada camara si el objeto es visible.
Reset	Se llama para inicializar las propiedades del script cuando es por primera vez adjunto al objeto y cuando el comando Reset es utilizado.
Start	Se llama cuando el objeto se carga, justo antes de que cualquier método update sea llamado por primera vez.
Update	Se llama en cada frame, si el objeto está enabled.

Formas de programar IA en los juegos

En un inicio, cuando empezaron a aparecer los primeros juegos y antes de las generalizaciones del uso de técnicas de IA, los personajes que no eran controlados por un ser humano sino por la propia máquina (NPCs) se comportaban de una manera fija, realizando desplazamientos según una ruta predeterminada o al azar, disparando de forma aleatoria, incluso siendo incapaces de evaluar situaciones de peligro. Estos comportamientos carentes de IA alguna hacían que el juego resultase monótono y predecible llegando a aburrir al jugador dado que éste no se sentía retado por el propio juego.

El intentar resolver dichos problemas para poder hacer los juegos más atractivos de cara al jugador hace que se empiecen a investigar y evolucionar las técnicas existentes para poder programar una IA para los NPCs que sea más competitiva. Esta IA se puede implementar de diversas formas, dependiendo el tipo de técnica que se siga: Máquinas de Estado (FSM, Finite-State Machines), Búsqueda de Caminos, Heurísticas, Algoritmos Genéticos, Redes Neuronales, Lógica Difusa.

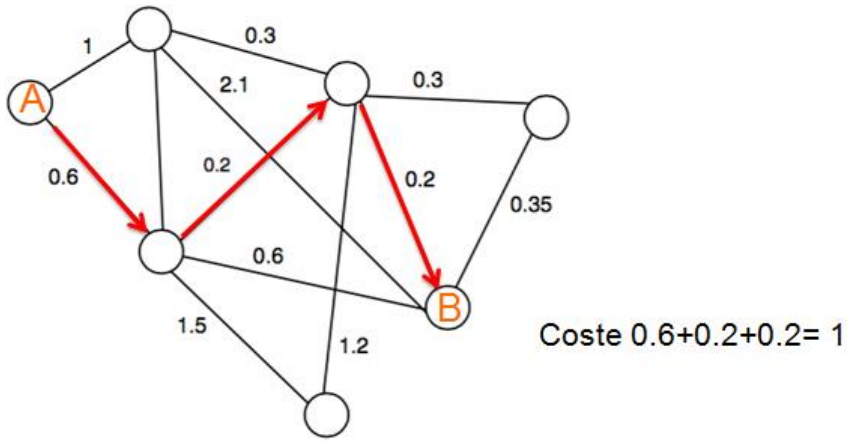
Cada una de estas técnicas tiene una aplicabilidad específica. Así para dar movimiento inteligente a los personajes se utilizan técnicas como la Búsqueda de Caminos, mientras que para hacer que el personaje tome decisiones en base a la información de que dispone para generar una acción se utilizan técnicas tales como árboles de decisión, máquina de estados y lógica difusa.

BÚSQUEDA DE CAMINOS (Pathfinding)

Los juegos normalmente necesitan que los personajes puedan moverse a través de la escena y en ocasiones, los creadores lo predeterminan, fijando el movimiento a una ruta fija. Esto es útil, por ejemplo, en un juego en el que un personaje debe patrullar una zona siguiendo un recorrido fijo. El problema es que si el personaje en cuestión encuentra un objeto en su camino que no estaba previsto que hubiese sido movido ahí, se queda bloqueado. Para evitar esto se pueden utilizar técnicas sencillas tales como movimientos aleatorios para evitar obstáculos (si los obstáculos son pequeños se pueden esquivar realizando un cambio de dirección lateral aleatorio y temporal en el avance) o avanzar alrededor de los obstáculos (si los obstáculos son grandes se pueden salvar siguiendo su frontera hasta que sea alcanzable de nuevo el destino sin el obstáculo) o técnicas más complejas y sofisticadas de pathfinding que determinen en qué dirección se ha de mover el personaje para llegar al destino deseado (desde una determinada posición).

Se trata pues de problemas de navegación en un escenario, que se representan como un grafo, en el que se desea ir desde un nodo (inicial) a otro nodo (final) con el menor coste posible (entre nodos hay un peso/coste: distancia, dificultad, obstáculos, etc.). Similar al problema de ir de una ciudad a otra dado un mapa, con muchos caminos posibles, acaso ciudades intermedias, zonas inalcanzables, etc.

El coste total de ir de uno a otro, sería la suma de costes:



Aunque a priori se podrían utilizar técnicas sencillas como los algoritmos voraces para su resolución, éstos podrían terminar en soluciones de coste muy elevado, por lo que generalmente se resuelven adaptando algoritmos tales como los clásicos Dijkstra, escalada (hill climbing), primero el mejor, y sobre todo A* (el cual hace uso de heurística para mejorar la eficiencia).

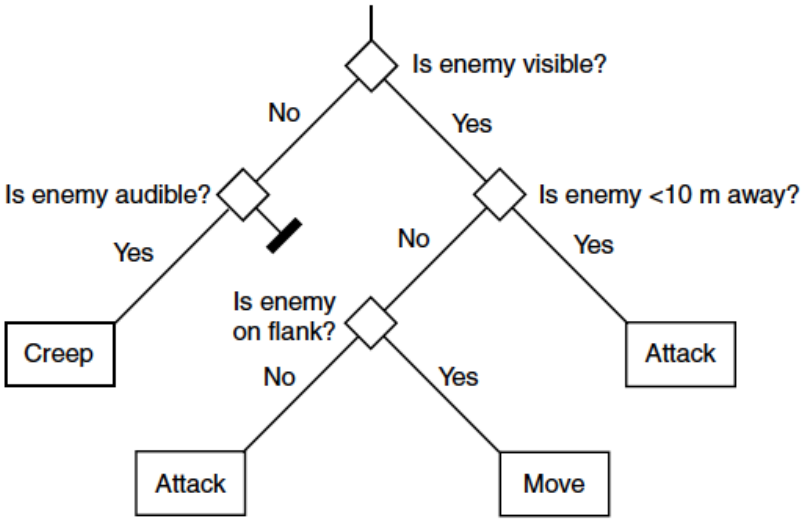
Por suerte Unity incorpora una herramienta que facilita bastante este trabajo, el NavMesh. NavMesh es un componente que se le puede asignar a cada objeto del juego. Una vez asignado, se le pasa al objeto (por ejemplo, el enemigo) una lista de posiciones ordenadas a gusto del programador. El *GameObject* empezará a dirigirse de manera ordenada a cada una de las posiciones indicadas, rodeando los obstáculos que encuentre, y que hayamos marcado previamente como obstáculos.

NavMesh contiene una serie de parámetros que debemos ajustar para lograr el comportamiento deseado. Por ejemplo cada *GameObject* que contiene un componente NavMesh ha de tener asignado un número de prioridad. De esa manera si dos o más de esos objetos se cruzan decidirán, basándose en ese parámetro de prioridad, cuál de ellos evita primero la colisión y rodea a los demás.

ÁRBOLES DE DECISIÓN (Decision Tree)

Los árboles de decisión es un mecanismo sencillo de implementar, rápido y fácil de entender y es utilizado en los videojuegos tanto para tomar decisiones como para el control de animaciones. Aunque sean el tipo de técnicas de toma de decisión más sencillas, pueden resultar bastante sofisticados con algunas extensiones/variaciones.

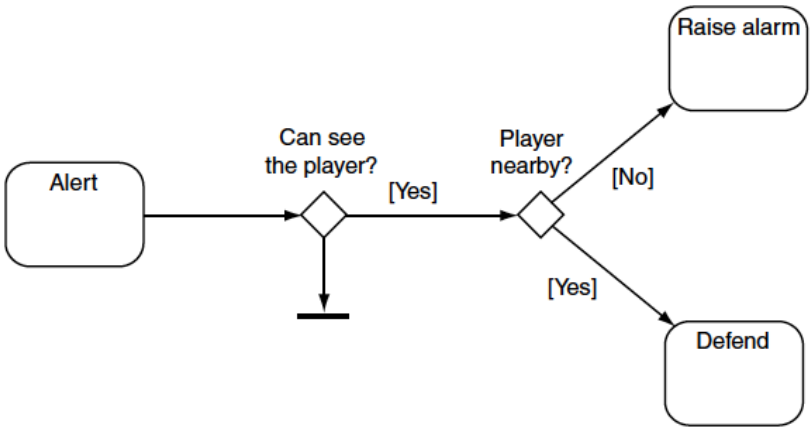
Como vemos en la siguiente figura, está formado por un punto de decisión, que se denomina la raíz. Cuando llegamos a un punto de decisión, la bifurcación que elegimos depende del estado de entrada, esto es, de la información de la que se dispone en el momento en el que se ha iniciado el proceso de toma de decisión.



Cuando el algoritmo de decisión llega a una determinada acción (hojas, en la figura), entonces esa acción es llevada a cabo inmediatamente (la misma acción puede utilizarse en distintas decisiones, de forma que la acción puede ser ejecutado desde diferentes condiciones previas, como ocurre con la acción Attack).

El árbol mostrado en la figura anterior es muy sencillo con opciones de tipo binario (YES/NO), pero se podría complicar con opciones numéricas, vectores... Incluso añadir una componente aleatoria mediante estados alternativos a los que se acceden con una determinada probabilidad.

Se pueden combinar árboles de decisión con máquinas de estados finitos: Autómatas con árboles de decisión entre los estados.

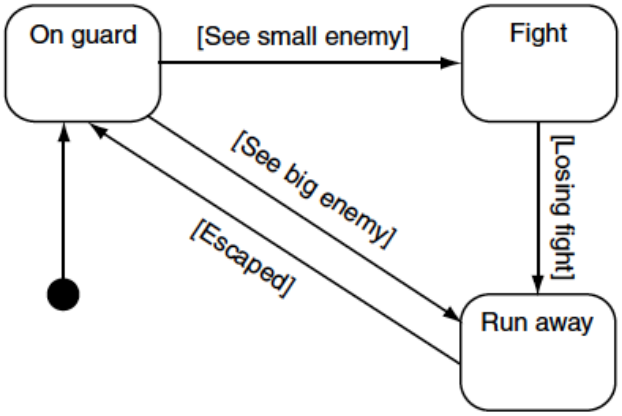


MAQUINAS DE ESTADOS (Finite State Machines o FSM)

Una máquina finita de estados (en inglés FSM) se puede considerar como una de las formas más simples de modelar IA y se utiliza comúnmente en la mayoría de los juegos por ser muy fáciles de implementar (los FSM simples pueden ser implementados usando instrucciones if else o instrucciones switch). Una FSM básicamente se compone de un conjunto de estados (vértices) y de transiciones (aristas) entre dichos estados.

En estas máquinas a cada estado se le asocia un tipo de acción o comportamiento (movimiento, huir, perseguir, esconderse, etc.). Por tanto, un personaje mientras permanezca en un mismo estado estará realizando una misma acción. Los distintos

estados están conectados mediante transiciones, que se producen solamente si se cumplen unas determinadas condiciones asociadas para el paso de un estado a otro (nos disparan, trascurre determinado tiempo, etc.) que dependen del exterior y opcionalmente del estado actual.



Por lo general, las FSM se usan para manejar unidades a nivel individual. Teniendo en cuenta lo que queremos que hagan dichas unidades y de qué manera lo realicen, agrupando acciones por un lado y condiciones de activación por otro. La idea es que el personaje se vaya moviendo de estado en estado estado, teniendo en cada estado una acción que realizar del tipo: moverse, atacar, perseguir, curarse, etc. Estas acciones que realiza son excluyentes, no interfieren unas con otras, dado que cada estado es independiente de los demás. El movimiento entre estos estados se da gracias a las transiciones. Estas transiciones se cumplen cuando se activan ciertas condiciones de activación como son: ningún enemigo a la vista, enemigo a la vista, enemigo perdido, salud baja, etc.

Las máquinas de estados tienen la ventaja de poder descomponer una acción compleja en subacciones de carácter más simples, lo que permite diseñar y estructurar complejos comportamientos de forma sencilla, siendo muy eficaz el empleo de esta técnica cuando queremos construir máquinas pequeñas y simples (alrededor de 10-15 estados por máquina). La gran desventaja que tiene una máquina de estados es cuando empiezan a incrementarse el número de estados y transiciones a causa de intentar construir comportamientos más complejos. Este inconveniente de las máquinas de estados se puede intentar solucionar mediante los Árboles de Comportamiento. Los cuales siguen manteniendo las ventajas que ofrece los FSM pero a su vez permiten una fácil implementación de IAs complejas.

ÁRBOLES DE COMPORTAMIENTO (Behavior Trees o BTs)

Los árboles de comportamiento (BTs) se han convertido en una herramienta muy popular para programar la IA de los NPCs y se usan para organizar el comportamiento de agentes y facilitar así el trabajo colaborativo. Halo 2 fue el primer videojuego de alto standing que implementó y detalló el uso de BTs para su IA.

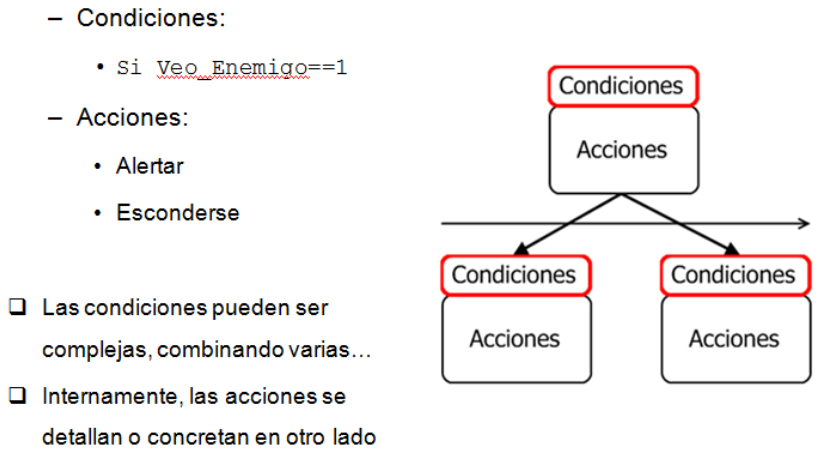
Los BTs simplifican el trabajo de creación de los comportamientos ya que permiten hacer más sencilla la inclusión de nuevos comportamientos, añadiendo complejidad a

las respuestas de los personajes. Los BTs son una síntesis de unas cuantas técnicas de que son utilizadas para el desarrollo de IAs: máquinas de estado, organización, planificación y acción ejecución.

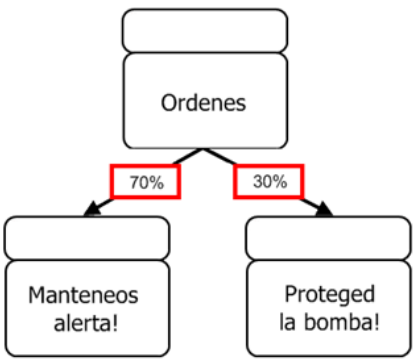
Los BTs tienen mucho en común con las FSM, como hemos podido apreciar en el apartando anterior, aunque en lugar de estados, los BTs tienen como bloque principal una tarea. Esta tarea puede ser tan sencilla como comprobar el valor de una variable en un estado del juego, o ejecutar una animación.

Las tareas pueden estar compuestas por sub-árboles para poder representar acciones más complejas. A su vez, estas acciones complejas pueden ser utilizadas para componer un comportamiento que tenga una mayor abstracción. En esta composición de distintos niveles de abstracción es donde reside la fuerza de los BTs. Dado que todas las tareas tienen una interfaz en común y pueden contenerse unas en otras, se puede construir fácilmente una jerarquía de tareas que permita despreocuparse de los detalles de implementación de las sub-tareas.

En los BTs los nodos representan cada comportamiento con sus acciones y precondiciones y los arcos marcan la jerarquía de comportamientos y su evolución. Se pasa de un nodo a otro nodo descendiente si se cumplen sus precondiciones completamente (las precondiciones son evaluadas de izquierda a derecha), deteniéndose el procesamiento cuando se alcanza un nodo hoja y se realizan sus acciones.



A veces, para aportar más realismo, las condiciones no se evalúan de izquierda a derecha sino que se hace aleatoriamente, e incluso con pesos diferentes, etc., para que no parezca sistemático.



Requisitos del juego a desarrollar

Como comentamos al inicio, el objetivo de la práctica es programar un videojuego en **2D** utilizando **Unity** (https://unity.com/es), un **motor gráfico de videojuegos** multiplataforma empaquetado como una herramienta para crear juegos, aplicaciones interactivas, visualizaciones y animaciones en 2D y 3D creado por Unity Technologies que está disponible como plataforma de desarrollo para Windows, OS X y Linux.

El juego a desarrollar será de libre elección por parte del alumno, al igual que el lenguaje utilizado para programar los scripts (se puede usar indistintamente C#, JavaScript o Boo), aunque independientemente del juego elegido este debe cumplir los siguientes requisitos mínimos:

- **El juego a desarrollar ha de tener varias pantallas o escenas (Scene)**, teniendo como mínimo una pantalla de presentación, otra de juego, otra de finalización y otra de configuración.
- **Los recursos (assets) utilizados en el juego** (sonidos, imágenes, texturas y materiales, música, etc.) **han de ser de libre distribución**, de forma que no se vulnere derechos de copyright (en la documentación habrá que indicar su procedencia y página de donde se han descargado, aun cuando sean de la propia Unity Asset Store).
- **La aplicación debe permitir guardar las preferencias más comunes del juego**, tales como volumen, records personales, partidas jugadas y cualquier cosa que necesitemos que se mantenga en memoria una vez se cierre nuestro juego.
- **La aplicación debe tener una pantalla de configuración** donde el jugador puede modificar aspectos del juego tales como número de vidas, tiempo para conseguir el objetivo del juego, etc.
- **Al finalizar el juego la aplicación debe mostrar la tabla de records personales**, permitiendo al jugador que inserte su nombre, en caso de que haya batido el record o quedado entre los 10 primeros.
- **En la medida en que sea posible, el juego debe hacer uso de alguna de las técnicas de IA enumeradas en el apartado anterior** (Búsqueda de Caminos, Árboles de Decisión, Máquinas de Estado…) para modelar la animación del personaje o las decisiones y comportamientos que éste pueda tomar.
- **Cuando esto no sea posible o no sea necesario** debido a las características del juego, **al menos deberá utilizar alguna técnica heurística** para que el comportamiento de los actores aparente cierta inteligencia **y/o alguna técnica aleatoria** para que el comportamiento no sea predecible.
- **Además del juego se deberá entregar una documentación** donde se explique las características del juego desarrollado, las clases y recursos utilizados para su elaboración y las direcciones web desde donde se han descargado dichos recursos, así como cualquier otro aspecto relevante que se desee considerar.

Evaluación

En la nota de la práctica se tendrá en cuenta tanto **el proyecto de programación** como el resultado de la entrevista y la defensa de las prácticas, **la calidad de la documentación entregada, la complejidad del juego desarrollado y la/s técnica/s de IA utilizada/s**, así como cualquier otro aspecto que deba ser tenido en cuenta como puede ser el número de integrantes del proyecto o el hecho de que los objetos del juego tales como música, sonidos, personajes, efectos especiales, etc. hayan sido creados por el propio alumno utilizando herramientas de modelado externas (aunque esto último no es necesario y no es objetivo de la asignatura se debe valorar el esfuerzo que el alumno realiza).

Tras la entrevista de prácticas, si la práctica está aprobada, se dará opción a mejorar la nota de la práctica corrigiendo los errores.

Las prácticas que no implementen TODA la funcionalidad requerida no serán evaluadas o tendrán una penalización en la nota final.

La copia de prácticas, aun siendo parcial, será sancionada con el suspenso de las prácticas tanto para el grupo que copia como para el que deja copiar. Cada grupo es responsable de la custodia de sus prácticas.

Entrega de prácticas

La entrega de prácticas se realizará en moodle. El representante del grupo deberá subir la documentación de la práctica y el proyecto de programación en un fichero comprimido .zip o .rar (no olviden indicar los nombres de los integrantes del grupo).

La práctica será realizada en **grupos de dos alumnos** como máximo y será desarrollada en una entrega. Una vez finalizada la práctica se realizará la entrevista de revisión y la defensa de prácticas.

El plazo de finalización de la práctica y presentación en clase finaliza el lunes **18 de abril** de 2022, disponiendo el alumno de una semana adicional para entregar y subir la documentación a moodle (hasta el lunes **25 de abril**).

El plazo para corregir la DOCUMENTACION, subsanar los errores y añadir las sugerencias propuestas tras la presentación de los videojuegos finaliza el **LUNES 2 DE MAYO**.

Nota:
Moodle no admite ficheros de tamaño mayor de 50Mb, por lo que si el fichero a subir pesa más no será posible hacerlo. En ese caso podrán entregarlo a través del servicio de consigna de la uhu (https://consigna.uhu.es), de dropbox (https://www.dropbox.com/) o de cualquier otro repositorio, subiendo al moodle un .zip o .rar con la documentación y un fichero .txt con el enlace al repositorio donde se haya subido (si el enlace está protegido por contraseña no olviden indicarla).