**OTROS SISTEMAS DE UNITY**

**1 - Pathfinding**

Pathfinding significa “Búsqueda de caminos”. Un algoritmo de pathfinding es una serie de pasos lógicos que nos permiten calcular cual es el camino más corto y eficiente entre dos puntos, sin atravesar ningún obstáculo.

Unity tiene su propio sistema de pathfinding. En este sistema los objetos que buscan caminos se llaman agentes y el área donde se puede caminar se llama mesh de navegación.

**1.1 - NavMesh**

NavMesh, o mesh de navegación, es un asset que representa una superficie tridimensional (un mesh) que define por donde pueden moverse nuestros agentes.

Crear un nav mesh es el primer paso para implementar pathfinding en nuestro juego.

Para que Unity pueda crear un mesh de navegación debe saber qué objetos en la escena son móviles y cuales son estáticos. Esto es porque el mesh de navegación no cambia, representa un área fija, y por lo tanto solo los objetos estáticos pueden considerarse para su creación.

Primero, entonces, debemos declarar qué objetos de nuestra escena son estáticos. Para esto seleccionamos todos los objetos que representen objetos quietos en nuestra escena (como las paredes, el piso y las rocas) y seleccionamos la opción “static” que se encuentra en la parte más alta del inspector.

Tras haber hecho esto abrimos la ventana de navegación (Window -> AI -> Navigation). En esa ventana, en la sub-pestaña “Bake”, hay un botón llamado “Bake” en la parte baja. Al apretar este botón Unity calculará y creará el NavMesh. Podremos verlo marcado en azul en la escena mientras la ventana de navegación permanezca abierta.

En la misma pestaña tenemos una serie de parámetros que pueden modificar el área de navegación. Muchos de ellos tienen que ver con qué podemos asumir acerca de los agentes de navegación. Esto quiere decir que, por ejemplo, el área para un agente muy ancho no tendrá disponible los pasillos estrechos, y que un agente muy alto puede pasar a través de pequeños obstáculos.

**1.2 - NavMeshAgent**

El NavMeshAgent es un componente que, añadido a un game object, permitirá que Unity lo desplace a través de nuestro nav mesh.

Cuando queramos que nuestro agente valla hacia un punto en el mapa, obtenemos una referencia al NavMeshAgent, que llamaremos agent, llamamos a la siguiente función:

agent.SetDestination (posicionDeDestino);

La posición de destino es un Vector3, cualquier punto que deseemos. El agente comenzará a marchar sólo hasta acercarse tanto a ese punto como el mesh de navegación lo permita.

La velocidad de movimiento del agente y la destreza o torpeza con que hará los giros en su camino, se pueden controlar modificando en el inspector las varias propiedades en este componente.

**1.3 - OffMeshLink**

Es posible que una pared sea muy empinada para subirla, pero también es posible que haya una cuerda que permita hacerlo. Si queremos conectar puntos normalmente desconectados en nuestro mesh de navegación, recurrimos a un componente llamado OffMeshLink.

Un OffMeshLink requiere una referencia a 2 transforms. Estos representarán el punto de origen y el punto de destino.

El hecho de tener un off mesh link en la escena hará que automáticamente los agentes lo consideren al calcular sus caminos.

Por defecto los agentes parecerán deslizarse de un punto a otro, lo cual suele verse extraño. Si queremos que, por ejemplo, el personaje corra una animación de subir una cuerda, debemos seguir estos pasos:

1 - Desactivar “auto traverse off mesh links” en el agente en cuestión. Esto evitará que el personaje intente recorrer automáticamente el link.

2 - En nuestro código, detectar que nuestro personaje entró a un off mesh link usando la variable booleana isOnOffMeshLink del agente.

3 - Ejecutar la animación correcta que el personaje debería utilizar en la transición (ejm: subir cuerda).

4 - Ejecutar lógica que haga que el personaje viaje del punto de origen al de destino de manera correcta. Podemos acceder a estos puntos usando agent.currentOffMeshLinkData.startPos y agent.currentOffMeshLinkData.endPos.

5 - Cuando el personaje haya llegado a su destino, llamamos a la función: agent.CompleteOffMeshLink() que liberará al agente para que siga su recorrido de manera automática.

**2 - Partículas**

Un efectos de partículas es una técnica que involucra la utilización de un montón de sprites que se mueven y transforman para simular efectos complejos como fuego, chispas y efectos especiales como rayos de luz o zonas de energía.

El sistema de partículas de Unity está basado en un solo componente, llamado ParticleSystem.

Este componente está compuesto de una serie de módulos, y cada módulo tiene una serie de propiedades relacionadas. Cada una de estas propiedades controla algún pequeño aspecto de los sprites del efecto, como por ejemplo la cantidad de ellos, su rotación, su color, su velocidad, etc.

**2.1 - Valor único o dinámico**

Muchas propiedades que se pueden encontrar en los módulos de ParticleSystem tienen más de una forma de ser asignadas.

Para entender esto, analizaremos la propiedad “start speed” y “start color” que se encuentran en el primer módulo.

Start speed es una propiedad numérica, y controla la velocidad que tendrá cada partícula en el momento en que nazca. Por defecto el start speed está definido por un solo número. Esto significa que todas las partículas del sistema comenzarán su vida con la misma velocidad fija. Esto no es siempre lo que queremos.

Si queremos que las partículas puedan nacer con distintas velocidades podemos escoger, con la flechita a la derecha de la propiedad, entre un número random entre dos valores y opciones de curvas.

Un random entre dos valores nos permitirá escribir dos números y hará que todas las partículas recién nacidas tengas una velocidad al azar entre ambos.

Una curva hará que, a lo largo de la vida del sistema, las partículas vayan naciendo con una velocidad distinta, según lo que indique la curva.

Finalmente tenemos la opción “random between two curves” en la que definimos dos curvas. A lo largo de la vida del sistema se irán escogiendo valores al azar entre los valores indicados por cada curva.

En el caso de la propiedad “start color” (el color con el que comienza cada partícula) tenemos opciones similares. En este caso, en lugar de elegir entre dos números elegimos entre dos colores y en lugar de crear una curva creamos una gradiente de colores.

Este es el nivel de detalle con el que podemos modificar cada propiedad del sistema de partículas, y tenemos cientos de ellas.

**2.2 - Start vs Lifetime**

En múltiples casos podemos encontrar dos versiones de una misma propiedad, la versión Start y la opción Over Lifetime. Por ejemplo, tenemos “start color” en el módulo principal y tenemos “color” en el módulo Color Over Lifetime. Estas dos propiedades modifican el color, pero de maneras distintas. Es muy importante conocer la diferencia.

Un efecto de partículas tiene una duración. Esta duración es el tiempo que el **sistema** estará vivo, emitiendo partículas. Además de esto cada partícula tiene un tiempo de vida. Este es el tiempo que la **partícula** estará viva.

El tiempo de vida de cada partícula y el tiempo de vida del sistema completo se definen ambos en el módulo Main (ver sección 2.3).

Entonces, cualquier propiedad que designemos con una curva o gradiente, como la velocidad o el color de una partícula, varía en uno de estos dos tiempos, el tiempo del sistema o el tiempo de la partícula.

Las propiedades “over lifetime” hacer que cada partícula cambie durante su propio tiempo de vida.

Las propiedades “start” hacer que cada partícula sea emitida con valores distintos, a lo largo del tiempo de vida del sistema.

**2.3 - Módulo Main**

**Duration:** Esta propiedad define el tiempo que durará el sistema de partículas. El tiempo en que estará emitiendo partículas. Esta duración es muy importante porque muchas propiedades de las partículas pueden tener curvas o gradientes que varíen a lo largo de este tiempo de vida.

**Loop:** Esta propiedad booleana decide si el sistema, una vez terminada su duración, comenzará desde el principio nuevamente.

Además de las propiedades recién descritas hay muchas que tienen que ver con las características iniciales de las partículas emitidas, como el color, la velocidad o la rotación con que las partículas son emitidas.

**2.4 - Módulo Emission**

Este módulo define tres aspectos sobre la emisión de partículas:

**Rate over time:** Cantidad de partículas emitidas por segundo.

**Rate over distance:** Cantidad de partículas emitidas por cada unidad de Unity que se desplace nuestro sistema de partículas. Útil cuando simulamos el polvo levantado por una rueda, por ejemplo.

**Bursts:** Esta lista define momentos en el tiempo de vida del sistema en que se producirán emisiones de muchas partículas en un solo instante. Por cada burst se puede definir cuántas partículas se emiten, cuantas veces se repetirá y el intervalo entre cada repetición. Útil, por ejemplo, si queremos simular señales de humo.

**2.5 - Módulo Shape**

Si este módulo se quita, todas nuestras partículas nacerán desde un único punto de emisión. Si queremos que se emitan en un área o volumen en particular, debemos seleccionar una forma de emisión en este módulo.

**2.6 - Módulo Renderer**

Este es el módulo que define cómo se verán nuestras partículas. La principal propiedad de este módulo es el material. Este material, que debe provenir de un shader apropiado, decide el aspecto de las partículas.

Hay muchos shaders hechos para partículas, y todos se pueden acceder en la subsección “Particles” al escoger el shader de un material. Estos shaders se diferencian en el modo en que se ven las partículas al estar sobre otros objetos y al solaparse entre ellas.

Cuando uno crea un material para partículas, con la mayoría de los shaders, se puede escoger una textura principal. Esta textura contendrá el o los sprites de las partículas.

**2.7 - Texture Sheet Animation y Otros Módulos**

Además de los recién descritos, hay módulos que controlan el color de cada partícula a lo largo de su vida, su tamaño, su velocidad, etc. También hay módulos que añaden ruido al movimiento de las partículas, haciendo que se muevan erráticamente, por ejemplo.

La mayoría de estos módulos se pueden entender intuitivamente dado su nombre y sus propiedades, pero hay uno que vale la pena describir mejor. Este es el módulo Texture Sheet Animation, que permite que las partículas cambien su sprite en el tiempo.

Lo que hace este módulo es tomar la textura principal del material y dividirla en pedazos. Para esto hay que asignar la cantidad de filas y columnas en que se dividirá el material.

Además de esto hay propiedades que deciden, por ejemplo, en cuál frame, de los recién definidos, comenzará cada partícula. Otras deciden qué tan rápido cada partícula cambiará de frame o cuantos ciclos de animación completos hará durante su vida.

**3 - Networking**

Networking quiere decir juegos en línea. El sistema de networking de Unity tiene la siguiente estructura:

* Cada computador conectado al juego corre una copia del juego de manera independiente. Esto es, a diferencia de otros modelos de networking en que solo un computador corre el juego y los otros solo lo observan.
* Debido a lo anterior hay que tomar medidas para que las copias independientes del juego se sincronicen. Esto es que, por ejemplo, si un personaje se mueve en un computador se mueva también en los otros.
* No es necesario sincronizar todo. Por ejemplo, la posición de la cámara de cada copia del juego debería mantenerse independiente en cada computador.
* Las cosas que necesitan sincronizarse le mandan mensajes a uno de los computadores conectados que hace las veces de líder. Este computador líder, conocido como servidor, se encarga de propagar los cambios a todos los demás computadores, para sincronizar las cosas que deben sincronizarse.

Las siguientes dos definiciones deben quedar claras antes de continuar:

**Server:** Computador que intermedia los mensajes entre todos los computadores conectados.

**Client:** Todos los computadores conectados, incluyendo el server, son clientes.

**3.1 - Network Identity**

El network identity es un componente de Unity. Ya hablamos de que hay ciertas cosas que queremos sincronizar en la red y ciertas cosas que no. Todos los game objects que contengan al menos un componente que queramos sincronizar deben tener un componente NetworkIdentity.

Un ejemplo lo debería dejar más claro.

Dijimos que cada computador corre su propia copia del juego. En cada una de estas copias existen diversos game objects, como por ejemplo una granada. Imaginemos que queremos sincronizar la posición de esa granada, para que esté en el mismo lugar en todos los computadores. Cuando la granada se mueve en uno de los computadores se le envía una notificación al server y el server busca todas las otras granadas en los otros computadores y les cambia su posición adecuadamente. Para que el server sea capaz de encontrar las granadas en todos los computadores las identifica por su componente network identity. Es decir, este componente es una etiqueta única que marca cada granada en la red de la misma forma, para que el server sepa que se trata del mismo objeto y sea capaz de propagar los cambios hechos a este.

**3.2 - Player Object**

Cada computador conectado tiene un objeto que lo representa, un avatar. Este avatar es literalmente un game object en la escena y se conoce como Player Object. Por lo general se trata de un objeto físico, con un modelo 3D, como una nave o un personaje, pero puede tratarse también de un gameobject abstracto, que tenga una serie de componentes que ejecuten sólo lógica y que no tenga una representación visual en la escena.

Dado que cada computador corre su propia copia del juego, cada avatar existe una vez en cada computador. Esto quiere decir, que si tengo un computador A y un computador B, en el juego que corre el computador A estará tanto el avatar de A como el de B. Lo mismo para el computador B, en ese computador estará también el avatar de A y el de B. Es decir, habrá un avatar A por cada computador conectado y un avatar B por cada computador conectado.

Necesitamos que cada computador sea capaz de reconocer cuál de los avatares en su copia del juego es el suyo y cuáles no lo son. Sabiendo esto podemos hacer que cada computador sepa a cuál de los avatares le corresponde controlar.

Para definir un Player Object necesitamos crear un game object y transformarlo en un prefab. Este prefab lo añadiremos a un script llamado NetworkManager, como veremos más adelante.

Este prefab debe tener un network identity, por las razones descritas en la sección anterior. Este network identity debe tener habilitado “local player authority”. Esta es la opción que le permitirá a cada computador diferenciar su avatar del resto de los que se están simulando.

El componente network identity con esa opción habilitada es lo único que necesitamos para definir un player object, pero en muchos juegos queremos además que tenga un renderer, un collider, un rigidbody, etc.

No debería quedar una copia activa de este prefab en la escena, porque los player objects serán instanciados dinámicamente. Si queremos conservar una copia en la escena para poder realizar cambios, tenemos que asegurarnos de desactivar su game object antes de correr nuestro juego.

**3.3 - Network Start Position**

El network start position es un componente que, añadido a un game object, lo marcará como un punto de aparición para los avatares de los jugadores en el juego. Podemos tener en la escena tantos como queramos. Se escogerá al azar entre ellos cada vez que aparezca el avatar de un jugador en la escena.

**3.4 - Network Manager**

El Network Manager es un componente que **debe** ser añadido a algún objeto en la escena en un juego en línea. Este componente es el que guarda la referencia al player object descrito más arriba. Un prefab se transforma en un player object en el momento en que se le añade al campo “player object” en este componente.

La propiedad **player spawn method** nos permite escoger entre “Random” y “Round Robin”. Esto decide si cada avatar aparecerá en un punto al azar o si sólo se seleccionará entre los puntos que no hayan sido utilizados aún, respectivamente (ver Network Start Position arriba).

Para poder probar nuestro juego en línea es útil añadirle al mismo objeto otro componente. Este componente es el **NetworkManagerHUD** y nos da una serie de botones de interfaz de usuario que nos permiten crear partidas en línea y unirnos a ellas.

Teniendo los elementos descritos hasta este punto ya se puede probar el juego. Para probar un juego en línea hay que correr al menos dos instancias de él. La manera más rápida de hacerlo es compilando nuestro juego y correrlo desde Unity y desde una aplicación independiente al mismo tiempo. Podemos hacer estas pruebas en el mismo computador. La interfaz de usuario que nos da el NetworkManagerHUD nos permitirá conectar los juegos entre ellos.

**3.5 - Scripting para juegos en línea**

El scripting para juegos en línea solo se diferencia del scripting en un juego normal en las partes donde hay que sincronizar cosas. La clase que contiene las funciones de sincronización es el NetworkBehaviour, que reemplazará al MonoBehaviour en los game objects que necesiten sincronizarse.

**3.5.1 - Network Behaviour**

Cuando queremos que un script sea sincronizado entre los computadores necesitamos 2 cosas:

1 - Debe heredar de NetworkBehaviour (que a su vez hereda de MonoBehaviour)

2 - Su game object debe tener un NetworkIdentity (ver más arriba).

Recordar que cada componente que exista en un computador tendrá sus homólogos en los otros computadores. El Network Behaviour tiene varios métodos y variables que nos permiten sincronizar las copias que existan de él. Las secciones que vienen describen estos métodos.

**3.5.2 - RPC**

RPC son las siglas de Remote Procedure Call, o Llamada a Procedimiento Remoto. Esto es cuando un computador llama a una función en otro computador. Un método (función) puede ser marcado como un RPC.

Un método RPC, **al ser llamado desde el servidor**, hace que se ejecute la función en **todos los clientes** (recordar que esto incluye el servidor).

Este es el principal método de sincronización que se utiliza en el sistema de networking de Unity.

Para hacer que un método sea un RPC hay que hacer tres cosas:

* Crear la función dentro de una clase que herede de NetworkBehaviour.
* Escribir el atributo [ClientRpc] arriba de la declaración de la función.
* Hacer que el nombre del método comience con “Rpc”.

Por ejemplo, si quiero activar una transición en el animator de todos los homólogos de un game object, en un NetworkBehaviour de ese objeto escribo:

[ClientRpc]

void RpcActivateTransition ()

{

GetComponent<Animator>().SetBool (“moving”, true);

}

Luego tengo que llamar a la función de manera normal:

RpcActivateTransition();

Lo importante acá es que debo asegurarme de que esta función sea llamada en la copia del juego que está en el servidor, no de cualquier cliente. Los commands son una manera de asegurarnos de esto, y se ven a continuación.

**3.5.3 - Command**

Un Command es una función que, no importa que cliente la llame, **será ejecutada en el servidor**, y en ninguna otra parte.

Para hacer que un método sea un Command hay que hacer tres cosas:

* Crear la función dentro de una clase que herede de NetworkBehaviour.
* Escribir el atributo [Command] arriba de la declaración de la función.
* Hacer que el nombre del método comience con “Cmd”.

Los commands vienen muy bien en conjunto con los RPC. A continuación un ejemplo de esto siguiendo el ejemplo de la sección anterior:

[ClientRpc]

void RpcActivateTransition ()

{

GetComponent<Animator>().SetBool (“moving”, true);

}

[Command]

void CmdActivateTransition ()

{

RpcActivateTransition();

}

Para activar la transición en todos los game objects homólogos hay que llamar a la función:

CmdActivateTransition();

Puede llamarse desde cualquier cliente, se ejecutará siempre en el servidor y hará que todos los clientes activen su transición de animación.

**3.5.4 - isLocalPlayer**

Cuando tenemos un NetworkBehaviour **dentro de un Player Object** (ver sección 3.2), la variable booleana isLocalPlayer nos indicará si se trata de nuestro avatar o no.

Esta es la manera en que diferenciamos entre nuestro avatar y el de otros.

Un ejemplo muy común es escribir algo como lo siguiente:

void Start ()

{

if (!isLocalPlayer) enabled = false;

}

Esto desactivará el script en cuestión en todos los avatares que no sean el nuestro. Muy útil si este script, por ejemplo, controla el movimiento de nuestro personaje.

**3.5.5 - isServer**

Por último, otra variable útil a la que se accede desde los NetworkBehaviour, es el booleano isServer. Como es de esperarse, isServer será true solo si el computador corriendo el script es el servidor.

**3.6 - Matchmaking**

El NetworkManagerHUD es muy útil para probar nuestro juego, pero es muy probable que queramos usar un método de matchmaking para la versión final de este.

Un sistema de matchmaking evita tener que conocer el IP del servidor para conectarse a un juego. Es una interfaz que nos permite crear y unirnos a juegos a través de **nombres**.

Esto quiere decir que, por ejemplo, un jugador puede crear un juego llamado “JoinMe”. Tras crearlo, otros jugadores pueden encontrarlo en una lista universal de todos los juegos creados, y pueden unirse él.

Un asset gratuito oficial de Unity nos permite hacer esto. Se llama Network Lobby y está disponible en el asset store.

Para utilizarlo hay que seguir los siguientes pasos:

1 - Crear una nueva escena. Será nuestro lobby.

2 - Añadir a esta escena el prefab LobbyManager.

3 - Asegurarse de que todas nuestras escenas estén en los build settings.

4 - Asignarle a este LobbyManager (que es en el fondo un NetworkManager) la referencia al Player Object.

5 - Asignarle al LobbyManager referencias a las escenas de lobby y de juego.

Esto es lo mínimo necesario para implementar matchmaking. El objeto Lobby Manager, y todos sus hijos y elementos de UI, pueden modificarse para adaptarse a la estética de nuestro juego.