**BASES DEL 3D EN UNITY**

**1 - Elementos de la gráfica 3D**

Para dibujar un objeto tridimensional en Unity son necesarias 3 piezas: Un mesh, un material y un renderer.

**Mesh:** Un mesh puede ser imaginado como una figura de alambres. Es un conjunto de vértices y triángulos que define un objeto tridimensional.

**Material:** Un material puede ser imaginado como una manta que cubre un mesh. Un mesh por si solo se podría decir que es completamente invisible, necesitamos cubrirlo con un material para que sea visible.

**Renderer:** El renderer es el componente que recibe un mesh y un material y los junta. Es el componente que, puesto en un GameObject, hace que sea visible en la escena.

**1.1 - Meshes**

Un mesh es el conjunto de toda la información acerca de la forma tridimensional de un objeto, más algunas cosas adicionales. A continuación se describen las más importantes.

**Vertices:** Esta es una lista de todos los vértices del modelo. Puede argumentarse que es la parte más importante de un mesh.

**Triangles:** Es un conjunto de datos que definen desde qué lado será visible cada triángulo que compone el mesh. Es muy común en los juegos 3D que los modelos sean visibles solamente vistos desde cierto ángulo. Un ejemplo típico es el terreno. En más de una oportunidad nos habremos topado con que la cámara del juego queda detrás de un muro y el muro se vuelve mágicamente invisible. Esto se hace para que el computador no tenga que dibujar ambas caras de un objeto, utilizando inútilmente el doble de recursos en el proceso.

**Rig:** No todos los meshes tienen uno, pero es esencial en muchos de ellos. Un rig es un esqueleto que le permite al mesh deformarse y ser animado.

**UV Coordinates:** Si queremos cubrir el mesh con una imagen, las coordenadas UV nos indican qué posición tiene cada pixel de la imágen en la superficie del mesh.

**1.2 - Materiales**

No se puede hablar de materiales sin hablar de shaders. Los shaders son scripts que realizan cálculos matemáticos y algoritmos para calcular el color de cada pixel que se renderiza. Un shader puede requerir como input para funcionar ciertos valores numéricos (como qué tan metálico queremos que sea el objeto) y texturas (que son básicamente imágenes cuadradas).

Un material es un asset que utiliza un shader en particular y le asigna valores y texturas concretas.

Podemos crear un material haciendo click secundario en la ventana de proyecto y escogiendo Create -> Material.

Una vez tengamos un material creado podremos ver en el inspector sus propiedades. La primera de ellas es el shader que utiliza, y será la que tenga mayor impacto en cómo será nuestro material.

Hay muchos tipos de shaders para crear diferentes tipos de materiales con ellos: cartoon, realistas, efectos de energía, etc.

Cuando creamos un material en Unity, el shader que utiliza por defecto es el Standard Shader. Este shader crea materiales que se comportan físicamente de manera realista. Esto quiere decir que el material, una vez aplicado a un objeto, hará que este produzca brillos, refleje la luz y proyecte sombras como un objeto físico real lo haría.

A continuación se describen algunos campos del Standard Shader. Esta lista pretende mostrar qué tipo de campos podemos esperar de un shader y, en particular, como un shader puede utilizar las texturas que se le den como input.

**Albedo:** Esta propiedad cuenta con 2 campos: un color y una textura. La textura que le asignemos cubrirá el objeto 3D, y el color será añadido sobre esta para darle un tinte adicional. Esta es la utilización más obvia de una textura: como una manta que cubre un mesh.

**Metallic:** Esta propiedad cuenta con un slider y con un campo de textura. El slider controla que tan metálico será el objeto que utilice este material (que se traduce en como refleja la luz). La textura hace lo mismo, pero permite más flexibilidad. Si utilizamos una textura en esta propiedad, podemos imaginarnos que, al igual que con el Albedo, nuestro mesh será envuelto en esta. A diferencia del Albedo, eso si, esta textura no aporta color sobre nuestro mesh, sino que cambia qué tan metálico es cada punto de nuestro objeto. Esto lo hace de acuerdo a un código de colores: mientras más rojo sea un píxel en un punto de la textura, más metálico será el punto del mesh que ese pixel cubra.

**Normal Map:** Esta propiedad cuenta solamente con un campo para una textura. Al igual que con la textura metálica, la textura de normales cubre nuestro mesh. La textura de normales cambia **la dirección en la que refleja la luz** cada punto de nuestro objeto. El efecto que produce una textura de normales es que nuestro objeto parecerá tener profundidades y elevaciones a lo largo de su superficie que no están realmente ahí. Esta es una manera muy económica de conseguir detalles en un modelo 3D. Si quisiéramos, por ejemplo, tener un personaje con muchos pliegues en su ropa, es mucho más eficiente aplicarle un mapa de normales que tener que crear un mesh complejo que dé cuenta de cada pliegue.

**1.2 - Renderers**

Un Renderer es un componente de Unity que le aplica un material a un mesh y lo hace visible en nuestro juego. En realidad los renderers son una familia de componentes que dibujan cosas en la pantalla. Los que nos interesan en esta sección son los renderers que tratan con meshes y materiales, y de estos hay 2:

**1.2.1 - MeshRenderer**

Un MeshRenderer necesita ir en conjunto con un componente MeshFilter. El MeshRenderer tiene una referencia al material que se utilizará y el MeshFilter contiene el mesh que se utilizará. En conjunto estos dos componentes son todo lo que necesitamos para que un objeto 3D sea visible en nuestro juego.

El MeshRenderer lo utilizamos para dibujar objetos 3D **sin esqueleto**. Esto quiere decir cualquier modelo estático, como un grifo, un edificio o una roca.

**1.2.2 - SkinnedMeshRenderer**

El SkinnedMeshRenderer nos sirve para dibujar modelos 3D cuyos meshes tengan esqueletos. A diferencia del MeshRenderer, el SkinnedMeshRenderer no necesita un MeshFilter, la referencia al mesh la tiene el mismo componente. Un SkinnedMeshRenderer necesita 3 referencias para cumplir con su función: Un mesh, un material y un **esqueleto**. El esqueleto es una jerarquía de objetos, cada uno de los cuales representa un hueso del rig de nuestro mesh. Cuando la posición o rotación de uno de estos huesos cambie, nuestro mesh de deformará apropiadamente.

**2 - Scripting 3D**

Esta sección trata diversas herramientas que podemos utilizar para crear la lógica de nuestros juegos 3D.

**2.1 - Rotación**

En los juegos 3D, mucho más que en los juegos 2D, nuestros objetos en la escena suelen cambiar de dirección (rotar). La rotación de un objeto está contenida en su componente Transform y está guardada como un quaternion. Un quaternion es una estructura matemática compuesta por 4 valores que en conjunto describen una rotación. Unity utiliza quaternions en lugar de vectores de 3 valores para describir sus rotaciones porque los últimos producen comportamientos erráticos cuando se interpola entre ellos. Esto quiere decir que si fuera de un vector de rotación A a uno B, la curva que seguiría mi objeto no sería limpia, sino que sinusoidal.

No necesitamos saber cómo funcionan los Quaternions para tratar con rotaciones en Unity. Nos basta con saber utilizar funciones que nos permiten utilizarlos.

**2.1.1 - Quaternion.Euler**

Para escribir en código una rotación en 3 ejes, lo hacemos de la siguiente manera:

Quaternion.Euler (90, 15, 180);

Esa función nos retorna un quaternion y toma como input valores de rotación normales en 3 ejes.

Si queremos que nuestro transform tenga una rotación de 90 grados alrededor del eje y, podemos escribir los siguiente:

transform.rotation = Quaternion.Euler (0, 90, 0);

**2.1.2 - Multiplicación de Quaternions**

Cuando queremos sumar dos rotaciones, no lo hacemos con el símbolo ‘+’, sino que con el símbolo de multiplicación ‘\*’.

Si queremos que nuestro transform sea rotado 90 grados **más** de lo que ya tiene alrededor del eje y, podemos escribir lo siguiente:

transform.rotation \*= Quaternion.Euler (0, 90, 0).

Esto hará que nuestro objeto haga un giro de 90 grados a partir de su posición actual.

**2.1.3 - Multiplicación de quaternions por vectores.**

Si queremos que un vector rote según una cierta rotación, podemos multiplicar esa rotación por el vector. Esta es una propiedad muy útil de los quaternions.

Si queremos que un vector que apunta hacia arriba gire 90 grados alrededor del eje x, podemos escribir lo siguiente:

Quaternion.Euler (90, 0, 0) \* Vector3.up;

Para utilizar esta propiedad necesitamos que el quaternion esté escrito a la izquierda del vector.

**2.2 - Vectores**

Los vectores se utilizan para todo tipo de cosas, pueden representar una posición en el espacio, una dirección de movimiento o una fuerza aplicada a un objeto.

**2.2.1 - Dirección entre dos puntos**

Si tenemos un vector que representa una posición A y uno que representa una posición B y queremos obtener un vector que indique la dirección desde A hacia B, lo hacemos restando ambos vectores: B - A.

Es decir, para obtener la dirección desde un punto a otro restamos **al** punto final el punto inicial.

**2.2.2 - Normalización de vectores**

Imaginemos que acabamos de obtener el vector que separa un punto A de uno B. Este vector tiene un largo igual a la distancia entre los dos puntos. En muchas situaciones queremos un vector que apunte de un punto a otro pero, al mismo tiempo, queremos que este vector tenga un largo de 1. La razón por la que querríamos esto se explica en la siguiente sección, pero esta es la forma de hacerlo:

vector.normalized

Es decir, podemos acceder a la propiedad “normalized” de cualquier vector y obtendremos el mismo vector, con la misma dirección y sentido, pero con largo 1.

**2.2.3 - Multiplicación de vectores**

Si multiplicamos un vector por un número ‘n’ obtendremos un vector con la misma dirección y sentido pero ‘n’ veces más largo. Por ejemplo:

V \* 3 nos dará un vector 3 veces más largo que V.

V \* 0.5f nos dará un vector de la mitad del largo de V.

Si el vector V tiene un largo de 1, el número por el que lo multipliquemos indicará el largo en el que quedará el vector. Es decir V \* 500 en este caso nos dará un vector de largo 500.

Juntemos lo que hemos visto en las secciones anteriores con un ejemplo: Imaginemos que queremos aplicarle a un objeto una fuerza de 200 hacia otro objeto. Podríamos seguir los siguientes pasos:

1 - Calculamos el vector dirección entre ambos objetos D = B - A.

2 - Normalizamos el vector resultante R = D.normalized.

3 - La fuerza que debemos aplicarle al objeto es R \* 200. Es decir, un vector que apunta del objeto A hacia el B de un largo exacto de 200.

**2.2.4 - Ejes globales**

A través de la clase Vector3 podemos acceder a las propiedades: Vector3.up, Vector3.down, Vector3.right, Vector3.left, Vector3.forward y Vector3.back.

Estas propiedades son variables estáticas que guardan vectores de largo 1 que apuntan en las direcciones básicas a lo largo de los ejes del mundo.

Vector3.up, por ejemplo, es un vector con los valores (0, 1, 0). Es decir, apunta exactamente hacia arriba en el mundo y tiene un largo de 1.

Estos vectores son muy útiles. Por ejemplo, si quisiéramos tener un vector que midiera 300 y que apuntara hacia arriba en el mundo, podemos escribir Vector3.up \* 300.

**2.2.5 - Ejes locales**

Es muy importante tener clara la distinción entre coordenadas locales y globales. Cada objeto que creamos en una escena en Unity tiene su propia noción de “hacia arriba”, “hacia abajo”, “hacia el fondo”, etc. Si tenemos un personaje humanoide, por ejemplo, la dirección de su mano derecha será su derecha **local**. Pero la mano derecha de ese personaje no tiene por qué apuntar hacia la derecha del mundo: la derecha **global**.

La derecha del mundo es Vector3.right y la derecha del objeto es transform.right.

Es decir, si queremos que el objeto camine hacia la derecha del mundo le decimos que valla hacia Vector3.right, pero si queremos que el objeto camine hacia **su** derecha, le decimos que vaya a transform.right.

Al igual que con Vector3.right, transform.right tiene un largo de 1.

Una de las propiedades más útiles de estos vectores locales es que podemos escribir cosas como esta:

transform.right = Vector3.up;

Esto obligará a que la mano derecha del personaje quede apuntando hacia arriba en el mundo, lo que hará que nuestro personaje quede acostado. Es decir, podemos utilizar los vectores locales como un método más para rotar nuestros objetos.

**2.2.6 - Proyección de vectores**

La proyección de un vector puede visualizarse como la sombra que deja un vector sobre una superficie o sobre otro vector.

Si queremos saber la sombra que deja un vector A sobre un vector B escribimos:

Vector3.Project (A, B);

Si queremos saber la sombra que proyecta un vector sobre una superficie el código sería:

Vector3.ProjectOnPlane (A, N);

Donde N es la normal de la superficie, es decir, un vector perpendicular a la superficie.