**CÁMARAS Y AUDIO**

**1 - Cámara, Oclusión y Batching**

**1.1 - El Componente Camera**

A continuación se analizan en detalle las propiedades más importantes de este componente que todos los juegos deben tener. Los desafío a crear un juego en que no haya ninguna cámara (¿quizás un juego en que eres ciego y te guias por el audio?).

**1.1.1 - Clear Flags**

Esta primera propiedad del componente cámara decide **que dibujar cuando no veo nada**. Esto quiere decir que, si la cámara observa una zona del juego donde no hay absolutamente ningún objeto que dibujar, rellenará esos pixeles en la pantalla con algo.

Por defecto lo que se dibujará en esas zonas es el skybox, es decir el cielo, pero también tenemos la opción de dibujar un color sólido **o de no dibujar nada**. Esto último cobrará sentido más adelante cuando hablemos de múltiples cámaras.

**1.1.2 - Culling Mask**

Esta propiedad es una lista de layers. Podemos escoger acá si la cámara ignorará objetos que se encuentren en ciertos layers. Si una cámara ignora un layer significa que verá a través de ella, como si esos objetos no existieran.

**1.1.3 - Projection**

Hay dos tipos de cámara, definidas por esta propiedad: la cámara en perspectiva y la cámara ortográfica. Una cámara ortográfica ve el mundo como aparece en los sims: los objetos se ven del mismo tamaño sin importar lo lejos que estén. Una cámara en perspectiva nos muestra el mundo como lo ven nuestros ojos: un dedo cerca de la pupila se verá más grande que una montaña.

**1.1.4 - Field of View y Orthographics size**

Si nuestra cámara es ortográfica aparecerá en el inspector la propiedad “orthographic size”. Si es en perspectiva, aparecerá “field of view”. Ambas propiedades controlan a su manera el zoom de la cámara.

**1.1.5 - Clipping planes**

Estas son dos propiedades, dos distancias, medidas desde la posición del objeto cámara. Representan la posición de dos planos de corte. Solo serán visibles para la cámara los objetos que se encuentren entre estos dos planos de corte.

**1.1.6 - Viewport Rect**

El viewport rect define qué zona de la pantalla ocupará la cámara. Nos podemos imaginar una pantalla dividida en 2 en un juego como Mario Kart. Tendríamos 2 cámaras, cada una con un diferente viewport rect.

**1.1.7 - Depth**

Cuando tenemos múltiples cámaras en la escena, hay un orden en que las cámaras dibujan en la pantalla. La cámara con el menor depth dibujará lo que ve de la escena en la pantalla primero. La cámara con el mayor depth dibujará en la pantalla al final, quizás sobre lo que las cámaras anteriores hayan dibujado. Tocaremos el tema de las múltiples cámaras a continuación.

**1.2 - Múltiples Cámaras**

Se pueden conseguir muchos efectos interesantes utilizando más de una cámara en la escena usando las propiedades descritas en la sección anterior. A continuación dos ejemplos.

**1.2.1 - Minimapa**

Para crear un minimapa necesitamos lo siguiente:

* Una cámara que observe el mundo desde lo alto. Esta cámara es preferiblemente ortográfica, para que las cosas se dibujen planas y sin profundidad.
* El viewport rect de la cámara del minimapa debe ocupar sólo un trozo pequeño del total de la pantalla. Idealmente en alguna esquina.
* La cámara del minimapa debe dibujarse sobre la cámara principal, por lo que su depth debe ser mayor.

**1.2.2 - Portrait**

Podríamos querer que la cara de algún personaje no jugador con el que estamos interactuando apareciera más grande en alguna esquina baja de la pantalla. Necesitaríamos lo siguiente:

* Una cámara posicionada frente a ese NPC (personaje no jugador, siglas en inglés).
* La cámara deberá dibujar solamente al NPC, por lo que filtrará los otros layers (propiedad culling mask, sección 1.1.2).
* No queremos que la cámara del NPC dibuje el skybox detrás de este, así que escogemos “Depth Only” en la propiedad Clear Flags (sección 1.1.1). Depth only hace que no se dibuje nada en las zonas vacías.
* Queremos que el NPC aparezca sobre la cámara principal, así que nos aseguramos de que su depth sea mayor que el de la cámara principal.
* El viewport rect debe estar ajustado a la esquina donde queremos mostrar el NPC.

**1.3 - Oclusión**

Por defecto, Unity solo renderiza (dibuja) los objetos que estén dentro del espectro de la cámara (entre sus dos planos de corte). Pero aún con esta optimización podemos estar haciendo trabajo de sobra. Si tenemos, por ejemplo, un objeto que está detrás de un muro, no tenemos por qué dibujarlo.

Para habilitar la oclusión debemos decirle a Unity qué objetos son inmóviles. De esta manera el motor podrá precalcular qué objetos cubren que áreas desde qué ángulos. Para esto debemos seleccionar los objetos inmóviles de la escena y habilitar la propiedad booleana “static” en la parte de arriba del inspector.

Tras definir los objetos estáticos debemos hacer el precálculo mencionado. Para esto primero abrimos la ventana de oclusión (Window -> Rendering -> Occlusion Culling). Desde acá hacemos click en el botón “Bake”, en la pestaña con el mismo nombre.

En la ventana de oclusión podemos ajustar algunos parámetros para conseguir resultados más precisos. Por ejemplo, podemos decirle qué tamaño tendrá el “smallest occluder”, es decir el objeto más pequeño que cubre la visión, para que la información de oclusión generada al hacer bake sea lo más pequeña posible.

**1.4 - Batching**

Hay dos computadores dentro de cada computador, un computador lógico y un computador gráfico, conocidos como CPU y GPU respectivamente.

Cuando el CPU decide dibujar un objeto en la pantalla, junta un montón de información en un paquete que contiene el mesh, el material, la posición, etc, y se lo envía al GPU para que lo resuelva y lo dibuje.

Mientras menos de estos paquetes enviemos, más rápido va a correr nuestro juego. Batching, o procesamiento por lotes, es una técnica que junta la información de los meshes y materiales de varios objetos a la vez y los envía al GPU como si se tratara de un solo objeto.

El batching ocurre automáticamente siempre que la cámara debe dibujar más de un objeto **estático** que **compartan materiales**. Es decir, si tenemos objetos inmóviles en la escena que ocupan el mismo material (como varios muros de un laberinto), conviene marcarlos como estáticos en la parte alta del inspector para que nuestro juego corra mejor.

**2 - Scripting de Cámara**

En esta sección se tocan dos temas: el movimiento gradual de cámaras y la detección de objetos que estén bajo el mouse.

**2.1 - Lerp para movimiento gradual de cámaras**

Es muy común querer que nuestra cámara observe un objeto desde cierta altura e inclinación. Conseguir esto es fácil, simplemente le decimos a la posición y rotación de la cámara que tomen ciertos valores. Esto último hará que la cámara instantáneamente se posicione y observe nuestro objeto. Ahora, si queremos que la cámara parezca moverse gradualmente hasta su posición de destino tenemos que hacer un poco más de trabajo.

Para conseguir estos movimientos graduales utilizamos algo conocido como interpolación lineal, o **lerp** por sus siglas en inglés.

La interpolación lineal es un método que toma un valor de origen y uno de destino y escoge un punto entre ellos dado un tercer valor. Este tercer valor representa una posición normalizada entre los dos primeros. Que la posición esté normalizada significa que 0 representa el punto de origen y 1 el punto de destino. Veamos un ejemplo.

Tengo los valores 4 y 10 y quiero ubicarme a la mitad de la distancia entre ellos. Para esto uso un método de lerp con estos valores, dándole una posición normalizada de 0.5. El resultado será la mitad entre 4 y 10, es decir 7 en este caso.

Hay funciones de lerp para números, vectores y rotaciones. Las últimas dos son relevantes para movimientos de cámaras.

**2.1.1 - Lerp para posiciones**

La clase Vector3 tiene una función Lerp que funciona como en el siguiente ejemplo:

posicionMedia = Vector3.Lerp (posOriginal, posFinal, 0.5f);

Esta línea de código encuentra el punto medio entre dos puntos usando Lerp.

Para conseguir cambios de posición graduales tenemos que aplicar repetidamente esta función. Una manera en que podemos hacerlo es dentro de Update:

void Update ()

{

transform.position = Vector3.Lerp (transform.position, posFinal, 0.3f);

}

Este código, en cada Update, acercará nuestra posición actual ⅓ de la distancia a nuestra posición final. Ese tercio de distancia cada vez será menor, lo que producirá un movimiento más lento mientras más cerca estemos del objetivo.

**2.1.2 - Lerp para rotaciones**

La clase Quaternion tiene una función Lerp que funciona como en el siguiente ejemplo:

rotacionMedia = Quaternion.Letp (rotOriginal, rotFinal, 0.5f);

Esta línea de código encuentra la rotación media entre dos rotaciones usando Lerp.

Para conseguir cambios de rotación graduales tenemos que aplicar repetidamente esta función. Una manera en que podemos hacerlo es dentro de Update:

void Update ()

{

transform.rotation = Quaternion.Lerp (transform.rotation, rotFinal, 0.3f);

}

Este código, en cada Update, hará que nuestra rotación esté ⅓ más cerca de nuestra rotación final. Ese tercio de rotación representará un ángulo cada vez menor, lo que producirá un movimiento más lento mientras más cerca estemos de la rotación objetivo.

**2.2 - Raycasts desde la cámara para detectar la posición del mouse**

El componente cámara tiene una función llamada ScreenPointToRay, que recibe como parámetro una posición en la pantalla y crea un rayo a partir de ella. Esta función es muy útil para detectar sobre qué objeto se encuentra nuestro cursor, pues si lanzamos un rayo desde la cámara pasando por la posición del mouse en la pantalla, ese rayo intersectará exactamente con lo que sea que nuestro mouse esté apuntando.

Revisemos el siguiente código:

Ray ray = cam.ScreenPointToRay (Input.mousePosition);

RaycastHit hitInfo;

if (Physics.Raycast (ray, out hitInfo))

{

// En este punto encontramos el objeto bajo el mouse.

}

La función ScreenPointToRay retorna un rayo que va desde la posición de la cámara “cam” pasando por la posición del mouse. Pasamos ese rayo a la función Raycast, que funciona tal como la describimos previamente en el curso, para detectar el objeto bajo nuestro cursor.

**2 - Audio**

Esta sección revisa los componentes y assets más importantes para manejar audio en Unity.

Los componentes más esenciales para manejar audio son el AudioListener y el AudioSource.

El AudioListener es un componente que, añadido a un game object, lo transforma en **nuestras orejas**. Es decir, para que Unity decida qué tan lejos se escucha un sonido, utiliza la distancia desde la fuente del sonido hasta nuestro componente AudioListener. En la escena debiera haber solo un componente AudioListener. Si hay más de uno Unity nos lanzará una advertencia. Este único AudioListener suele estar en la cámara principal, que representa la posición de los ojos del jugador (que suele corresponderse con la posición de sus orejas).

El AudioListener no tiene ninguna propiedad visible en el inspector.

El AudioSource es un componente que, añadido a un game object, lo transforma en un emisor de sonidos. No hay límite para la cantidad de AudioSource que pueden haber en una escena.

**2.1 - AudioSource**

A continuación se presenta un glosario con las propiedades más importantes del componente AudioSource.

**AudioClip:** Este será el sonido que el audio source tocará. Cualquier archivo de sonido que importemos a Unity puede ser arrastrado en este campo.

**Mute:** Decide si el sonido se escucha o no.

**Play On Awake:** Si esta propiedad es verdadera el sonido comenzará a sonar desde que la escena empieza. Si es falsa hay que decirle al audio source con código que comience a sonar.

**Loop:** Si esta propiedad es verdadera el sonido se repetirá indefinidamente.

**Volume:** El volumen del audio.

**SpatialBlend:** Esta propiedad decide si nuestro sonido será afectado por la distancia que lo separa de nuestro AudioListener. Si el slider está en el extremo 2D el sonido se escuchará igual de fuerte sin importar a qué distancia se encuentre el audio source. Si está en el extremo 3D el sonido se escuchará más despacio mientras más lejos se encuentre. Podemos escoger un punto intermedio entre los dos extremos.

**Doppler Level:** El efecto doppler es un fenómeno físico que cambia la frecuencia de un sonido según se aleja o acerca a nosotros la fuente de este. Con este slider controlamos la intensidad de este efecto.

**Gráfico:** Este gráfico puede editarse para controlar cómo varían algunas propiedades, como el volumen y el spatial blend, de acuerdo a la distancia que tiene este audio source con respecto al audio listener. Una línea vertical roja indica la posición actual del audio listener.

**2.2 - AudioReverbZone**

Este componente, al ser añadido a un game object, crea dos volúmenes esféricos alrededor de él. Si nuestro audio listener está dentro de estas esferas el sonido se verá distorsionado de alguna manera.

**2.2.1 - Min y Max Distance**

Las dos primeras propiedades de este componente definen los radios de las dos esferas. Cuando nuestro listener cruce el umbral de la primera esfera, el sonido comenzará a distorsionarse. Cuando nuestro listener cruce el umbral de la esfera interior, el volumen habrá alcanzado su nivel máximo de distorsión.

**2.2.2 - Reverb Preset**

El efecto de distorsión que nuestro reverb zone producirá se selecciona de esta lista. Tenemos opciones para simular cuevas, hangares, montañas, llanuras e incluso efectos psicóticos.

**2.3 - AudioMixer**

A diferencia de lo descrito hasta el momento, el AudioMixer no es un componente sino un asset. Es decir, no existe en nuestra escena sino en nuestro proyecto.

El audio mixer puede ser entendido como un filtro ubicado entre un audio source y el audio listener, que modifica propiedades del sonido antes de que lleguemos a escucharlos.

Para crear un AudioMixer vamos a Create -> AudioMixer en la ventana de proyecto. Si le hacemos doble click a este asset abriremos la ventana del audio mixer.

**2.3.1 - Grupos**

Un audio mixer está compuesto de audio mixer groups. Cada uno de estos grupos puede recibir audio y modificarlo de alguna forma. Estos grupos están jerarquizados, al igual que nuestros objetos están jerarquizados en nuestra escena.

Cuando un grupo recibe un sonido lo modifica y luego se lo pasa a su grupo padre. El padre hace lo mismo y el proceso se repite hasta que llegamos al único grupo que es padre de todos, el tronco de la jerarquía de grupos. Este último grupo aplica también sus efectos y luego, finalmente, le pasa el sonido, modificado por todos estos filtros, al audio listener para que lo escuche.

En la ventana del audio mixer creamos grupos añadiéndolos a la lista que se encuentra a la izquierda. Podemos jerarquizar y nombrar estos grupos de la misma forma en que lo hacemos con los game objects en la escena.

Los grupos que creemos se visualizan como rectángulos grandes en la parte central de la ventana. Al seleccionar uno de ellos el inspector nos mostrará los efectos que este grupo aplica a los sonidos que recibe. Podemos añadir efectos de una manera homóloga a cómo añadimos componentes en un game object. Hay un botón en el inspector que nos permite hacerlo.

Los efectos que se pueden aplicar varían enormemente. Van desde efectos de cambio de frecuencia hasta efectos de eco pasando por todo tipo de distorsiones. Cada uno de estos efectos tiene sus propios parámetros que podemos editar en el inspector.

En el componente AudioSource (sección 2.1) hay una propiedad llamada output. Es ahí donde seleccionamos a que grupo este audio source irá a parar. Si un audio source no va a ningún grupo el sonido irá directamente al audio listener.

**2.3.2 - Modificar parámetros con código**

Es posible que queramos usar código para cambiar algún parámetro de algún efecto de algún grupo de nuestro audio mixer. Para hacer esto debemos seguir los siguientes pasos:

1 - Exponer la propiedad que queremos cambiar. Para hacerlo hacemos click secundario en la propiedad y seleccionamos “expose to script”.

2 - Hecho esto, arriba a la derecha en la ventana de mixer podemos seleccionar el parámetro expuesto y ponerle un nombre para referenciarlo.

3 - Finalmente en nuestro código, usando una referencia a nuestro AudioMixer, podemos llamar a la función SetFloat para asignarle el valor que queramos a la propiedad.

El código se vería así:

mixer.SetFloat (“masterVolume”, 200);

Además, para poder tener una referencia al mixer debemos escribir:

using UnityEngine.Audio;

en la parte más alta del script.