Jumping Robot 3D Hektor Profe

https://www.hektorprofe.net/

Proyecto

Nuevo proyecto 3D “Jumping Guy 3D”
Renombrar la escena a “Juego”
Acelerar Unity (Edit > Project settings > Editor > Enter Play Mode Settings > Marcar Mode Options y Scene
Descargar los paquetes (o instalar el adjunto en recursos):

- Jammo Character
- Polygon Started Pack
- Simple Sky Cartoon Assets
- Simple FX Cartoon Particles

Mover los assets externos a _Assets

Escena

Abrir la escena **SimpleSky > _Demo** y copiar los objetos **SkyDome y Clouds**
Pegarlos en la escena **Juego** y cambiar el cielo del **SkyDome > Materials < PolygonStarterSimpleSky_01**
Añadir un suelo Float_01 (0,0,0) 50x1x50
Añadir una montaña Mountains_Grass_02 en (0,0,16) con rotacion (0,180,0) y escala 2x3x2
Crear un GameObject vacio llamado Entorno, meter el suelo, la montaña, el cielo y las nubes.
Hacer que el entorno no se pueda seleccionar.
Hacer que el suelo y la montaña no se puedan seleccionar

Personaje

Añadir una instancia de **Jammo_Player** en (0,0.25,-1)
Jammo tiene un controlador por defecto AWSD (bajarle la velocidad para hacer pruebas)
Nuestro Jammo no va a correr lbremente, la jugabilidad del juego será muy básica, únicamente saltará para esquivar rocas que irán cayendo.
Posicionar la cámara en (5, 1, -3) con rotación (-15, -35, 0)
Cambiar su Character Controller de la siguiente forma:

Roca

Añadir “Roca” como Sphere_01P en (0,30,13.5) 1.5x1.5x1.5, cambiar su material a Mat_03 (gris)
Añadirle un rigidbody y reducir su drag y angular drag a 0.
Comprobar qué ocurre al dejarla caer.
Congelar posición X en la roca, añadir un material físico “Roca” sin resistencias, 0.25 de bouncing y otorgárselo.
Llevarse le material a la carpeta “Materials”
Aumentar la gravedad del juego al doble (*2)

probar el resultado.

Salto

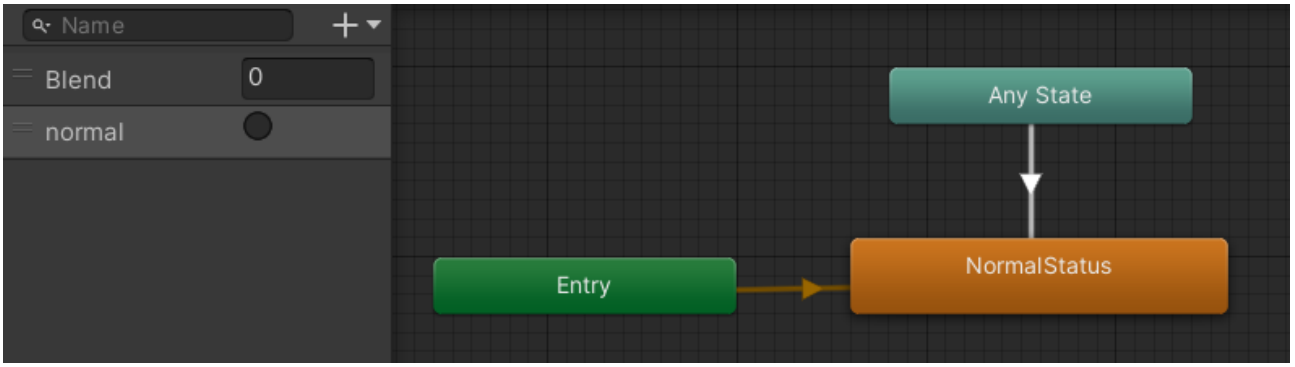
Desactivar los scripts actuales de jammo
Crear script **JammoController:**

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class JammoController : MonoBehaviour
{
    public float jumpSpeed;  // 12f
    public float gravity;    // 32f
    public Animator anim;
    public CharacterController controller;
    public bool isJumping;
    public Vector3 moveDirection = Vector3.zero;

    void Update()
    {
        handleMovement();
        handleGravity();
        handleJump();
    }

    void handleMovement()
    {
        controller.Move(moveDirection * Time.deltaTime);
    }

    void handleJump()
    {
        if (Input.GetButtonDown("Jump"))
        {
            if (!isJumping && controller.isGrounded)
            {
                isJumping = true;
                moveDirection.y = jumpSpeed;
            }
```

```
        }
    }

    void handleGravity()
    {
        if (!controller.isGrounded)
        {
            moveDirection.y -= gravity * Time.deltaTime;
        }
        else
        {
            isJumping = false;
        }
    }
}
```

Animación

Borrar estados y parametros del animador de Jammo, dejar solo:

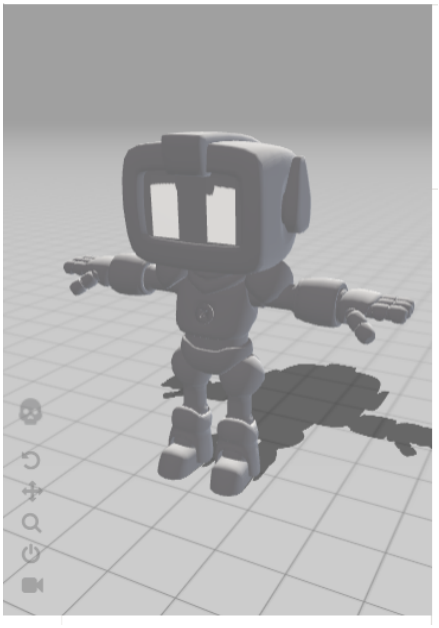


A partir del minuto 19:30, mucha atención https://www.youtube.com/watch?v=h2r3_KjChf4

Jammo tiene una animación por defecto de salto, pero no nos sirve porque tiene movimiento incorporado.

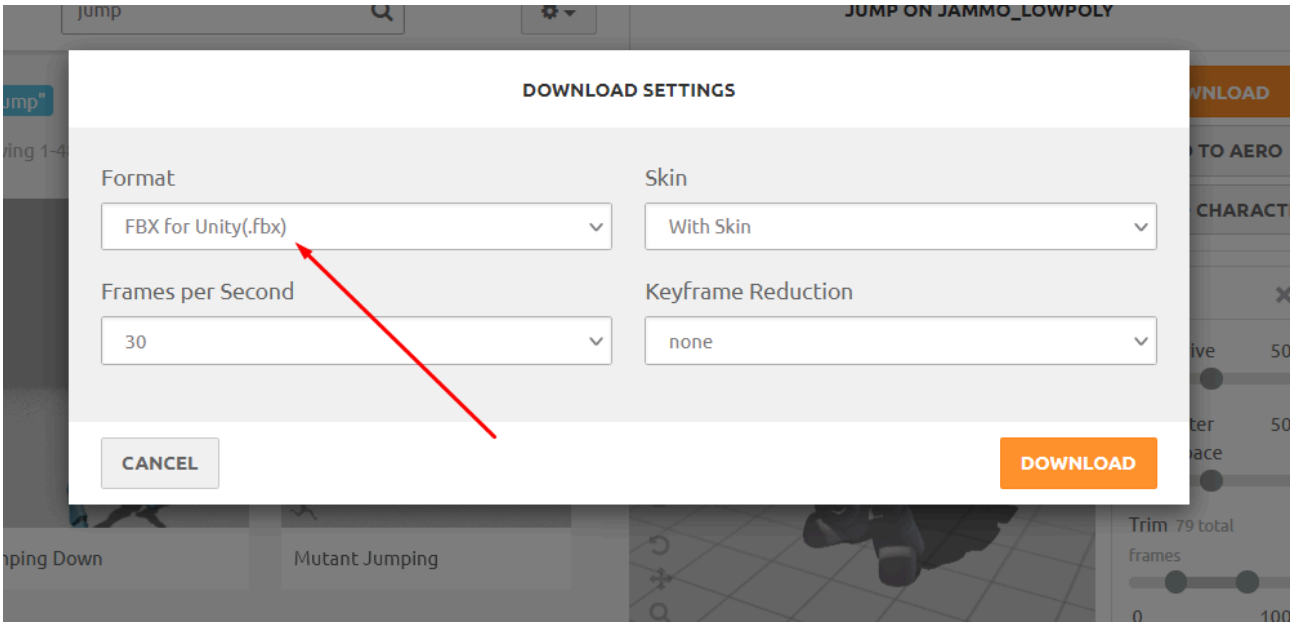
Vamos a generar una animación gracias a Mixamo, un servicio online de Adobe donde encontraremos multitud de animaciones.

Empecemos creando un cuenta en mixamo.com, aqui debemos subir el modelo de Jammo.



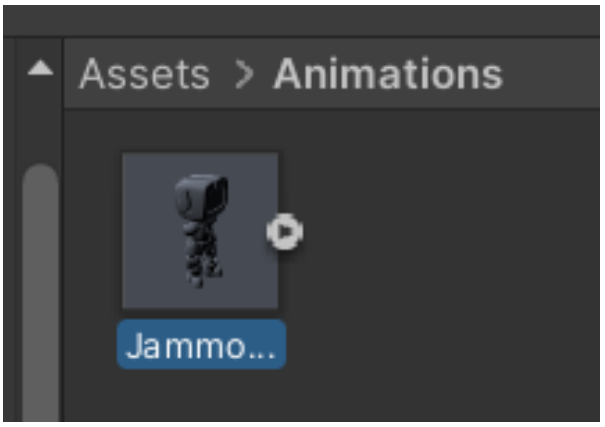
Una vez lo tenemos buscamos la animación Jump → *Female rosa* y la activamos para ver como la replica Jammo:

Descargamos la animación para Unity con el modelo de jammo (esta vez con SKIN, al final del curso lo haremos sin skin importandolo del avatar):



Os dejo una copia de seguridad de la animación en los recursos de la lección.

Arrastramos el fichero a un nuevo directorio ***Animations*** en Unity.

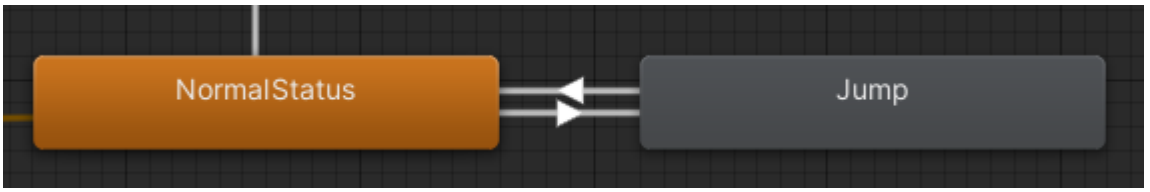


Ahora en la animación nos asegurarnos de que clip y nombre de la animación concuerden (jump y jump):

De vuelta al animator creamos un nuevo **State → Jump**

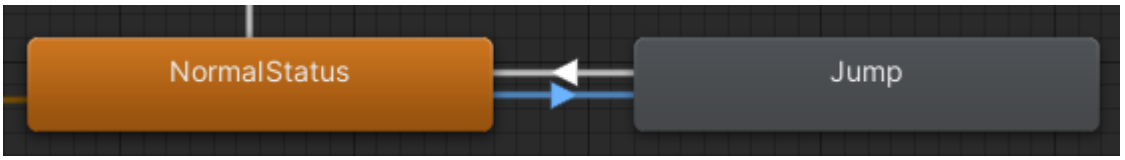
Seleccionamos para el state Jump la animación que hemos creado (no confundir con la original):

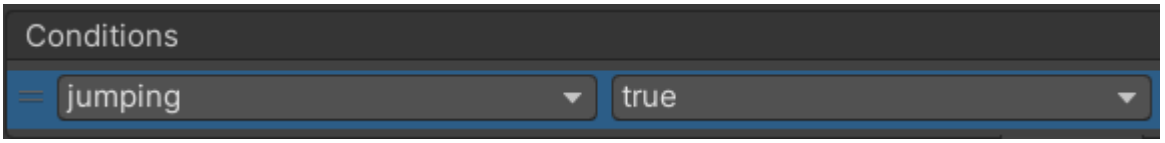
Creamos la transición a Jump de ida y vuelta:



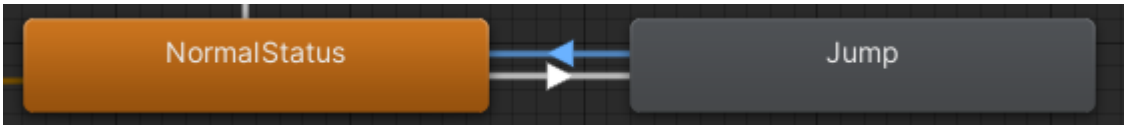
Desactivamos **en todas ellas** *Has Exit Time* para que la animación no tenga que acabar antes de reproducirse la siguiente animación.

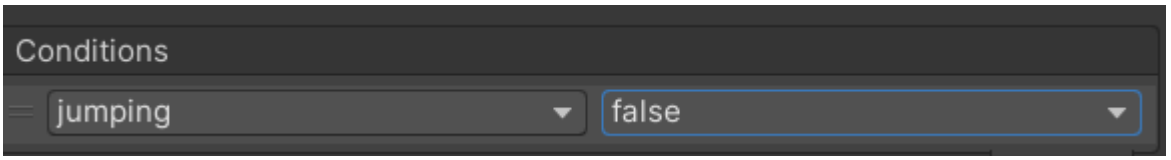
Creamos un parametro booleano jumping y la añadimos como condición de entrada:





Y la vuelta en false:



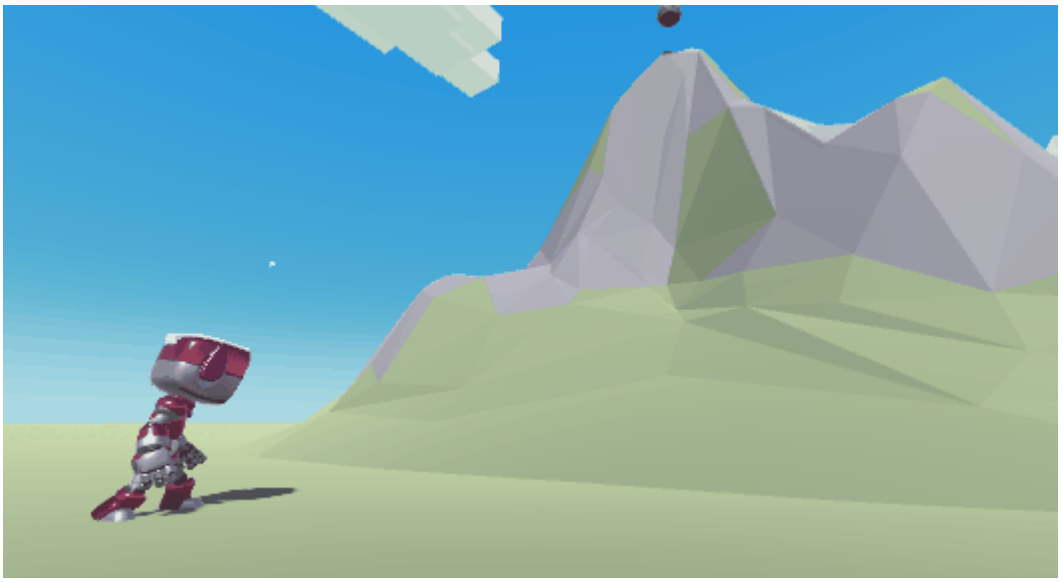


Establecemos los parametros de animación en el script, al empezar a saltar y al tocar el suelo:

```
    void handleJump()
    {
        if (Input.GetButtonDown("Jump"))
        {
            if (!isJumping && controller.isGrounded)
            {
                isJumping = true;
                moveDirection.y = jumpSpeed;
                anim.SetBool("jumping", true);  // empezar animacion
            }
        }
    }

    void handleGravity()
    {
        if (!controller.isGrounded)
```

```
        {
            moveDirection.y -= gravity * Time.deltaTime;
        }
        else
        {
            isJumping = false;
            anim.SetBool("jumping", false);  // finalizar animacion
        }
    }
```

Ya debería funcionar la animación, podemos ajustar las transiciones si lo deseamos o dejarlas así:



Generador

Creamos un directorio Prefabs, arrastramos la roca de la escena como Prefab Variant y la llamamos simplemente Roca y borramos la instancia de la escena.

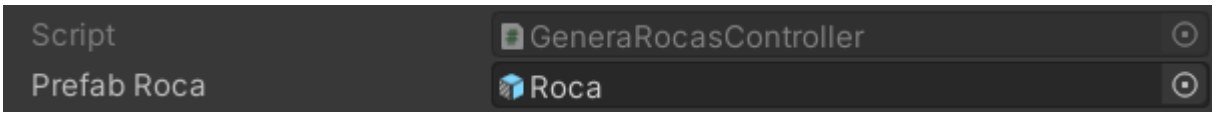
Establecemos la posición del prefab roca en (0,0,0) por defecto y lo guardamos.

Creamos un objeto vacío llamado GeneraRocas con un script GeneraRocasController:

```
public class GeneraRocasController : MonoBehaviour {
```

```
    public GameObject prefabRoca;

    void Start() {
        GenerarRoca();
    }
    void GenerarRoca() {
        Instantiate(prefabRoca, transform.position, transform.rotation);
    }
    void OnDrawGizmos() {
        Gizmos.color = Color.yellow;
        Gizmos.DrawWireSphere(transform.position, 0.5f);
    }
}
```

Posicionamos el generador de rocas en (0,30,13.5) y configuramos el prefab:



Configuramos un temporizador para repetir el método cada cierto tiempo:

```
    public float startTimer = 0f;
    public float repeatTimer = 2f;

    void Start() {
        InvokeRepeating("GenerarRoca", startTimer, repeatTimer);
    }
```

Para evitar que se queden las rocas en el escenario vamos a hacer que se borren automáticamente unos segundos después de instanciarlas:

```
void GenerarRoca() {
   GameObject roca = Instantiate(prefabRoca, transform.position, transform.rotation);
   Destroy(roca, 5);  // Destruir después de 5 segundos
}
```

Colisión

Un player controller detecta colisiones solo contra objetos que se mueven en dirección opuesta hacia donde él se mueve, como si entrase en ellos. En nuestro caso solo ejercemos una fuerza en el eje Y para saltar, por

lo que solo podemos detectar rocas saltando sobre ellas. Así que en lugar de detectar las colisiones en el jugador lo haremos en las rocas.

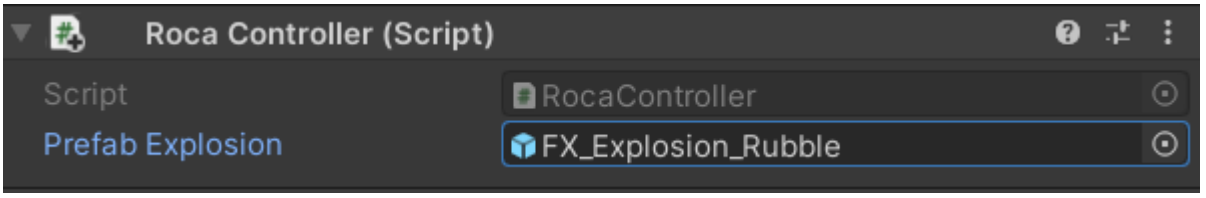
Creamos un script ***RocaController*** en el prefab de la roca.

```
public class RocaController : MonoBehaviour

void OnCollisionEnter(Collision collision)  {
    if (collision.gameObject.name == "Jammo_Player") {
        print("Colisión contra el jugador");
    }
}
```

Ahora que sabemos cuando ocurre el choque vamos a crear justo en ese lugar unas particulas:

```
public class RocaController : MonoBehaviour {
    public GameObject prefabExplosion;
    void OnCollisionEnter(Collision collision) {
        if (collision.gameObject.name == "Jammo_Player") {
            Instantiate(prefabExplosion, collision.transform.position,
                                         collision.transform.rotation);

        }
    }
}
```

El prefab es ***FX Explosion Rubble***:



El efecto resultante es muy chulo.

En este punto, justo cuando sucede la colisión vamos a:

- Borrar el personaje.
- Desactivar el generador de rocas.
- Borrar todas las rocas del escenario.

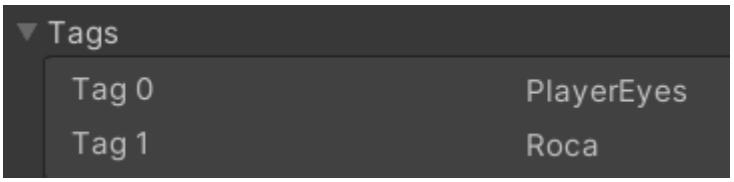
Borrar el personaje:

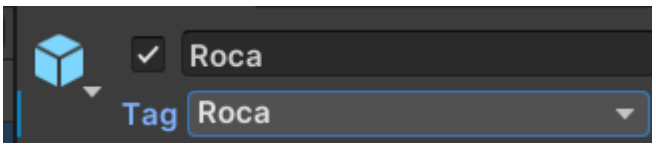
```
 // Destruir el jugador tras un instante
Destroy(collision.gameObject, 0.02f);
```

Desactivar el generador:

```
// Buscar el generador y cancelar todos los invokes
GameObject generador = GameObject.Find("GeneraRocas");
generador.GetComponent<GeneraRocasController>().CancelInvoke();
```

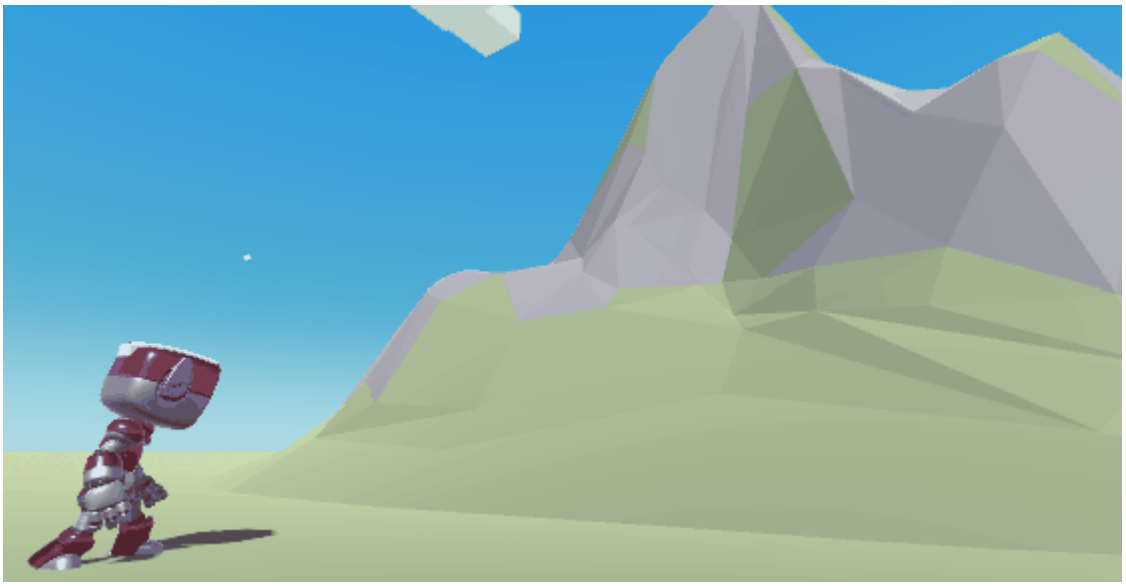
Para borrar todas las rocas debemos recuperarlas dinámicamente, para ello vamos a crear un tag en el Prefab que las identifique:





Ahora a través de código, con un simple foreach:

```
// Recorremos todas las rocas buscando a partir del tag
GameObject[] rocas = GameObject.FindGameObjectsWithTag("Roca");
foreach (GameObject roca in rocas) {
    // Destruimos las rocas al cabo de un instante
    Destroy(roca, 0.02f);
}
```



Partículas

Vamos a darle un toque más atractivo al juego generando unas partículas durante el salto.

Empecemos por unas partículas justo en el momento de saltar, el prefab ***FX_Fireworks_Blue_Small*** así que definamos una referencia a él en JammoController:

```
public GameObject prefabFXSalto;
```

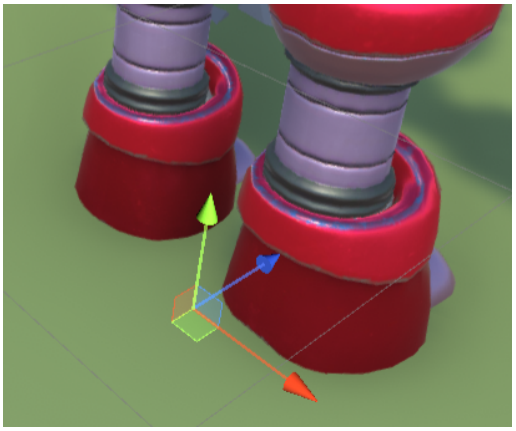
Y lo asignamos:

Ahora, vamos a crear la instancia justo al saltar:

```
if (!isJumping && controller.isGrounded) {
    ...
    Instantiate(prefabFXSalto);
}
```

En este punto tenemos que solucionar varias cosas. Primero, las partículas no aparecen en el sitio correcto, no tienen el color ideal y no se borran automáticamente.

Empecemos por cambiar la posición. Para esto podemos hacer un truco muy sencillo. Creamos un nuevo objeto dentro de **Jammo** y le llamamos **PosicionParticulas**. Este por defecto se crea en (0,0,0) respecto a Jammo, en sus pies, el lugar ideal para crear las partículas:



Ahora en el controlador de Jammo creamos una referencia a su transform:

```
public Transform posicionParticulas;
```

Y le asignamos este objeto:

Para crear las particula ahí simplemente usamos su transform:

```
Instantiate(prefabFXSalto, posicionParticulas.position, posicionParticulas.rotation);
```

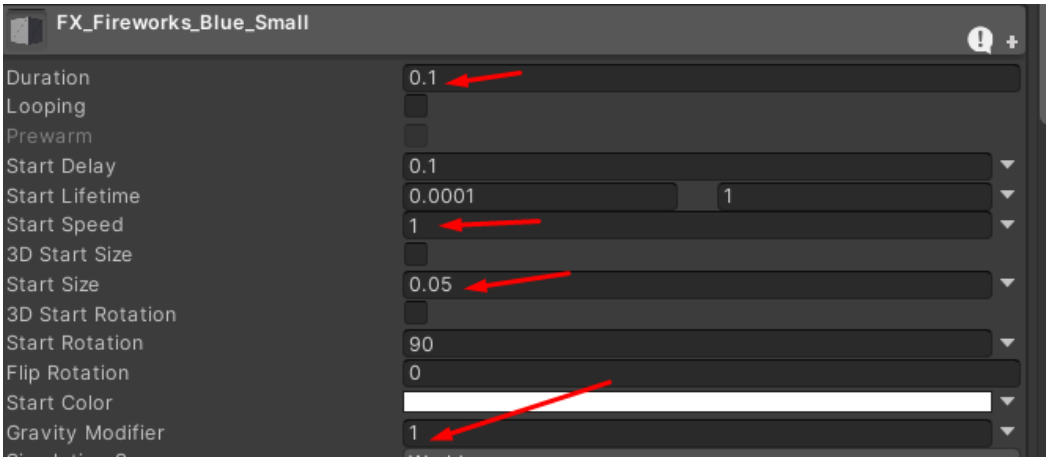
Primer problema resuelto.

Ahora debemos hacer que se borren automáticamente las instancias:

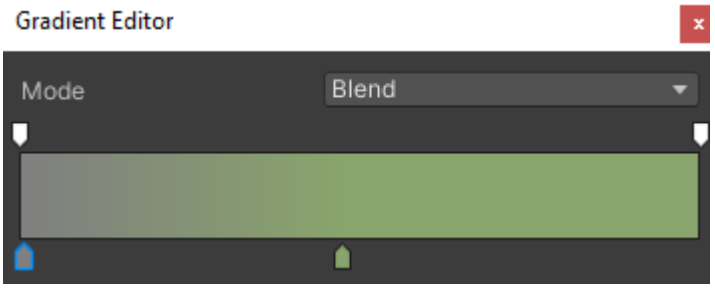
```
GameObject fxSalto = Instantiate(
   prefabFXSalto, posicionParticulas.position, posicionParticulas.rotation);
Destroy(fxSalto, 2);
```

Finalmente configuramos el prefab original a nuestro gusto:

La duración, la velocidad inicial, el tamaño inicial, y que les afecten la gravedad a modo de piedrecitas...



Para el ***Color over Lifetime***, empezamos con un color a piedras y lo difuminamos hacia la mitad a un color verde como el suelo:



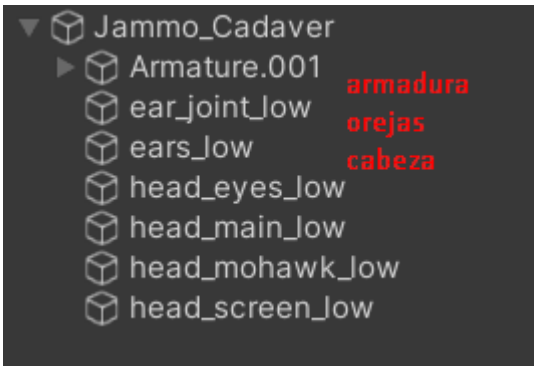
Experimentad vosotros mismos hasta conseguir el resultado que más os guste:

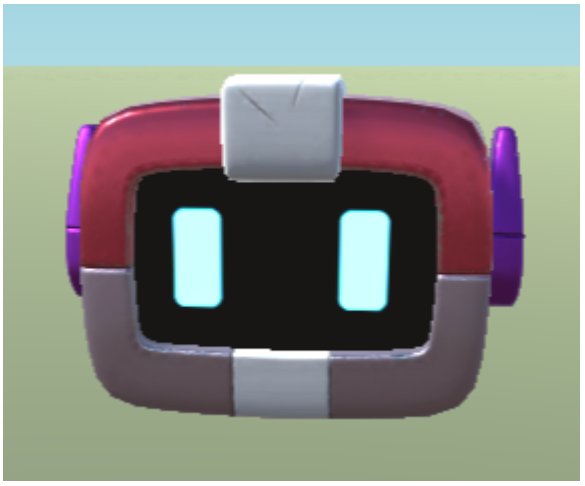
Restos

Vamos a crear los restos del pobre Jammo después de colisionar contra la roca, y que mejores restos que una cabeza de robot!.

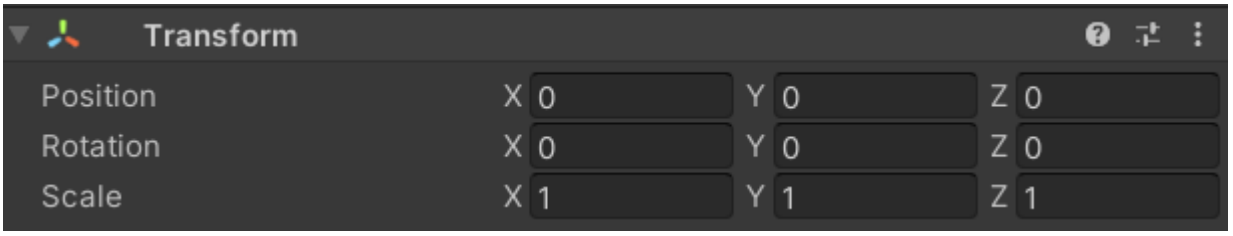
Vamos a crear una nueva instancia del prefab Jammo, haremos clic derecho en la jerarquía Prefab > Unpack.

Ahora del objeto resultante, que ya no será una instancia de prefab, vamos a ir borrando partes hasta dejar solo la cabeza con las orejas:

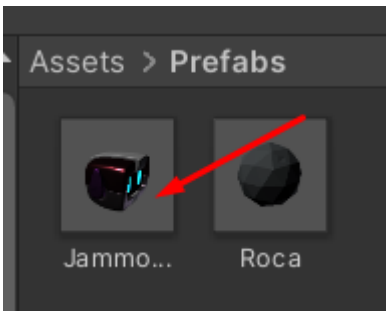




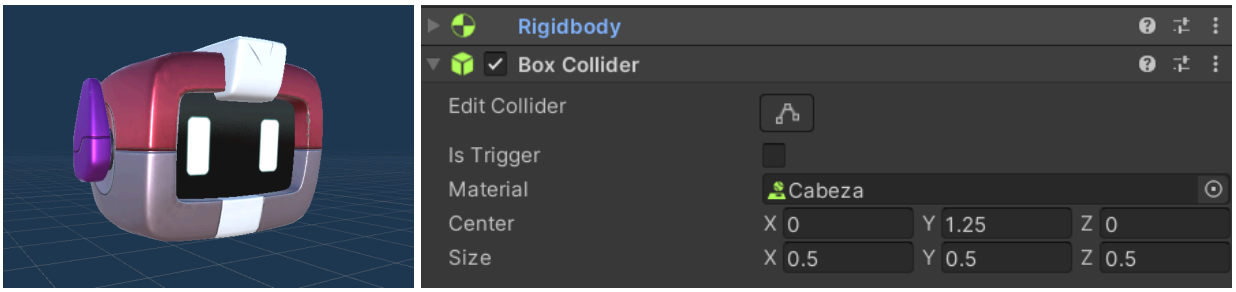
Es muy importante no rotarlo y dejar todos sus componentes con la posición y rotación a cero:



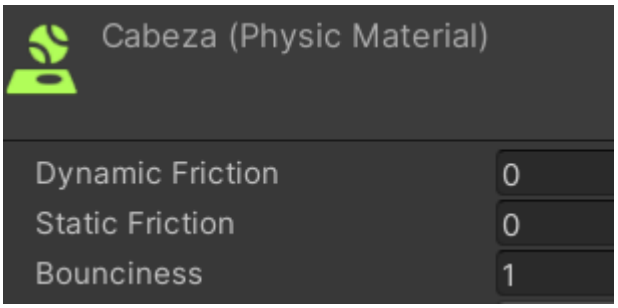
La cambiaremos el nombre al objeto a **Jammo_Cadaver** y lo arrastraremos a nuestra carpeta de **Prefabs**:



Ahora para que la cabeza caiga sobre el suelo le vamos a añadir un rigidbody y un BoxCollider:



Y para darle un toque vamos a añadirle un material físico con rebote:



Ahora vamos a crear una referencia a este Prefab en el script ***RocaController*** y lo asignaremos:

```
public GameObject prefabCadaver;
```

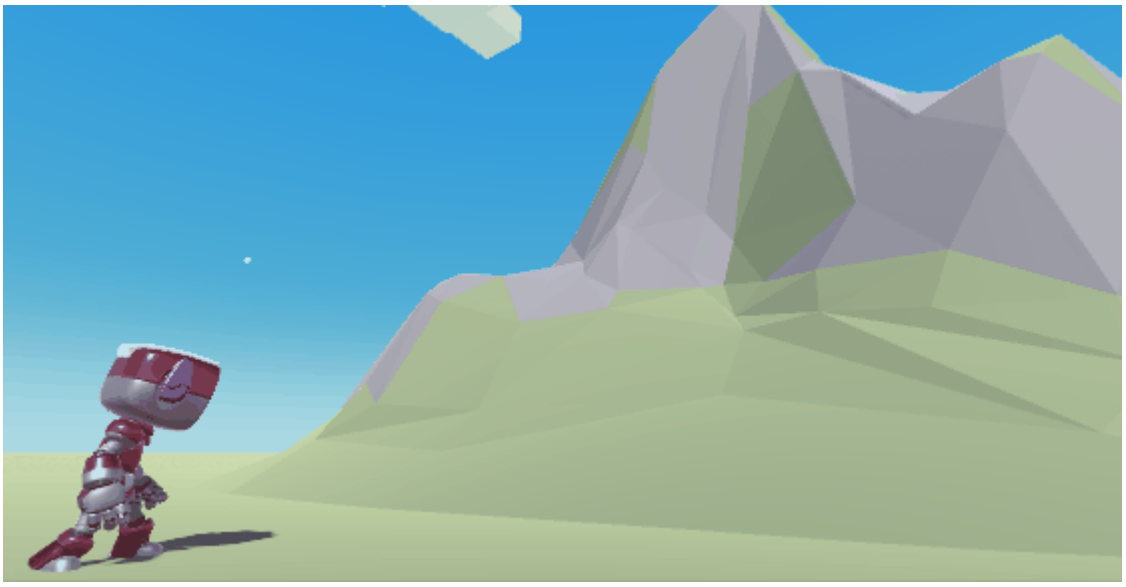
Ya solo nos falta crear la instancia de este prefab justo después de borrar las rocas:

```
// Instanciamos el cadaver en el lugar de la explosion
Instantiate(prefabCadaver, collision.transform.position,
                           collision.transform.rotation);
```

Incluso podríamos aplicar una pequeña fuerza a la cabeza para generar un efecto más interesante:

```
// Instanciamos el cadaver en el lugar de la explosion
GameObject cadaver = Instantiate(prefabCadaver,
          collision.transform.position, collision.transform.rotation);

// Le aplicamos un impulso vertical
// cadaver.GetComponent<Rigidbody>().AddForce(Vector3.up * 5, ForceMode.Impulse);
// cadaver.GetComponent<Rigidbody>().AddForce(Vector3.back * 2f, ForceMode.Impulse);
cadaver.GetComponent<Rigidbody>().AddForce(new Vector3(0f, 5f, 2f), ForceMode.Impulse);
```

Y listo… pobre Jammo xD



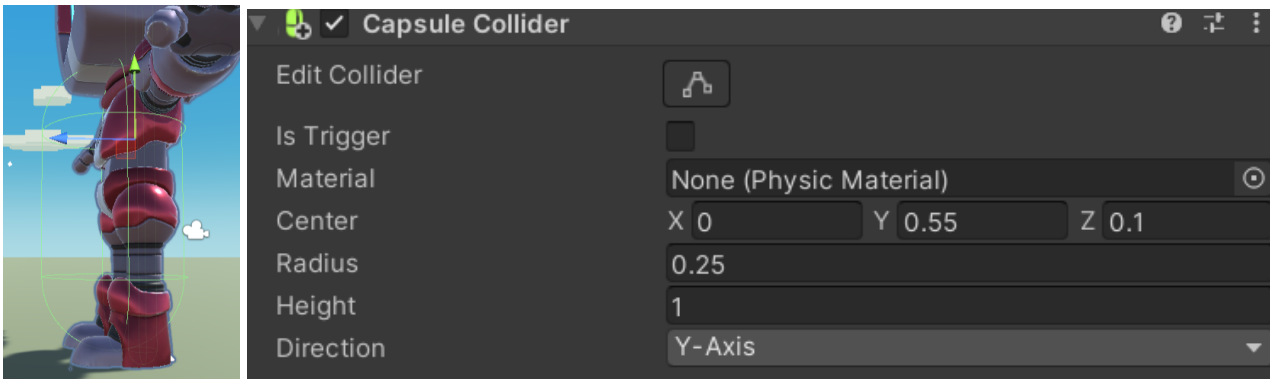
Bug

Tenemos un bug.

A veces podemos saltar sobre la roca y caminar sobre ella sin que ocurra la colisión. Es un breve momento pero no me gusta nada! La pregunta es, ¿porqué sucede esto? La forma de calcular las colisiones físicas entre el PlayerController, que es un cilindro y el rigidbody de la roca depende de la fuerza opuesta. Al caer hay un momento en que no se ejerce prácticamente fuerza y eso significa que no se detecta colisión porque no se llega a penetrar la roca.

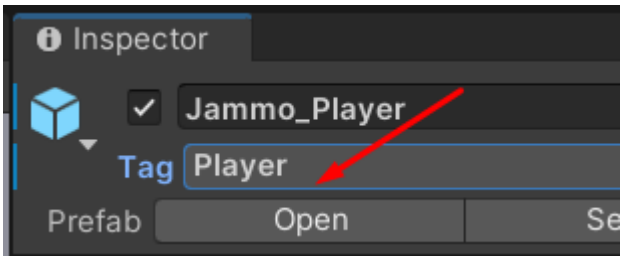
Para solucionar el bug vamos a hacer dos cosas.

En primer lugar crear un collider alternativo en nuestro personaje, este collider nos permitirá asegurar las colisiones donde el PlayerController no acaba de funcionar bien. Será un capsule collider:



Ahora, en segundo lugar, vamos a desactivar el collider de la roca cuando ésta se encuentre por detrás del jugador en el eje Z. Esto debería evitar el efecto de caminar por encima.

Para detectar el objeto jugador fácilmente podemos asignarle el tag Player que viene por defecto:



Y en el Update haremos la comprobación para desactivar el collider vía el script ***RocaController***:

```
void Update()  {

    GameObject player = GameObject.FindGameObjectWithTag("Player");

    if (transform.position.z < player.transform.position.z) {
        GetComponent<MeshCollider>().enabled = false;
    }
}
```

Y con estas dos rectificaciones el bug ya no debería ocurrir, o en caso de ocurrir será mucho menos perceptible.

Reinicio

Necesitamos una forma de reiniciar el juego una vez finaliza, así que vamos a ello.

Vamos a crear un objeto para manejar las acciones relacionadas con la escena, lo llamaremos Manager.

Dentro añadiremos un script llamado ***ManagerController***. Para reiniciar una escena debemos hacer uso del gestor de escenas, que debemos importar del módulo SceneManagement

```
using UnityEngine.SceneManagement;
```

Vamos a crear un método para cargar una escena a partir del nombre, así podremos reutilizarlo en el futuro:

```
void CargarEscena(string nombre) {
      SceneManager.LoadScene(nombre);
}
```

Tan simple como esto.

Ahora para reiniciar la escena actual podemos detectar, por ahora, si presionamos la tecla R, así que desarrollemos la lógica en el Update:

```
void Update() {
    if (Input.GetKeyDown(KeyCode.R)) { CargarEscena("Juego"); }
}
```

Con esto podemos reiniciar la escena Juego presionando R. Si lo hacemos, seguramente tendáis un problema y es que la luz de la escena deja de funcionar. La solución es muy simple, debemos generar las luces para la escena desde ***Window > Rendering > Lightihng > Generate Lightning:***

Listo.

Dificultad

Ahora mismo el juego no ofrece un reto, saltar sobre las rocas es bastante simple. Lo que vamos a hacer es incrementar la velocidad del juego paulatinamente para que la misma mecánica de saltar sobre una roca sea cada vez más difícil.

En el script GeneraRocasController,

```
public float scaleInterval = 6f;    // cada X segundos
public float scaleSeconds = .25f;  // % de incremento en base 1

void Start() {
    InvokeRepeating("EscalarDificultad", startTimer, scaleInterval);
}

void EscalarDificultad() {
    Time.timeScale += scaleSeconds;
}
```

Con esto tenemos listo el bucle de juego, cada vez irá más y más deprisa.

Solo debemos asegurarnos de restaurar la velocidad de juego a 1 cuando el jugador pierde, así que de vuelta al script ***RocaController***, justo al colisionar contra el bueno de Jammo, abajo del todo vamos a establecer de nuevo el escalado a 1:

```
// Reestablecemos la velocidad de juego
Time.timeScale = 1;
```

Perfecto.

HUD

En esta lección vamos a añadir un HUD con la puntuación actual y la más alta.

Para dibujar el HUD vamos a hacer uso del paquete ***TextMeshPro***, así que debemos instalarlo desde el ***Package Manager*** si no lo tenéis. Además debéis instalar las dependencias esenciales:

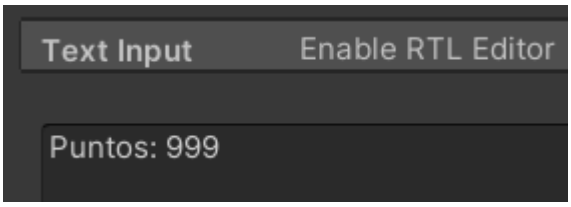
Luego en la jerarquía clic derecho ***UI > Text - TextMeshPro***.

Se crearán tres objetos, uno llamado Canvas con un Text (TMP) y otro llamado EventSystem.

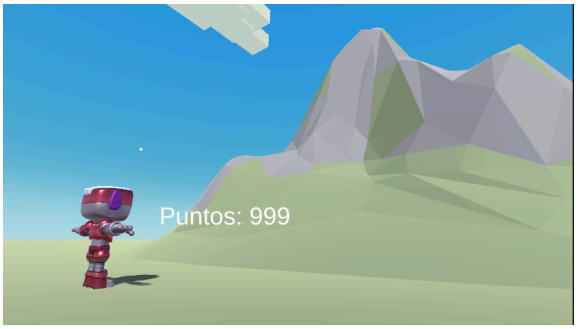
Vamos a renombrar el objeto **Canvas** → ***HUD*** y **Text (TMP)** → ***Puntos,*** eventsystem lo dejaremos tal cual***.***

Si hacemos doble clic en ***Interfaz*** veremos el área que ocupa el lienzo donde se dibujarán los elementos gráficos, de ahí que tenga asignada la ***Layer UI*** por defecto.

Si hacemos doble clic en el objeto Puntos podemos escribir un texto:



Sin embargo lo más posible es que este texto no visualice correctamente al probar el juego:

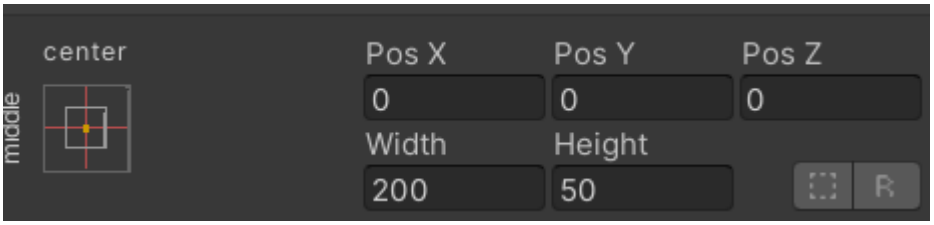


Hay dos cosas a tener en cuenta.

Primero el sistema de renderizado del lienzo, éste puede ser respecto a la vista de juego, a la vista de la cámara o dentro del espacio de juego, como si fuera otro elemento tridimensional.

Aquí entra en juego la segunda cosa, el escalado.

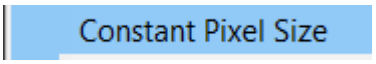
Tened en cuenta que los elementos de interfaz tienen un tamaño en píxeles:



Incluso la propia fuente:

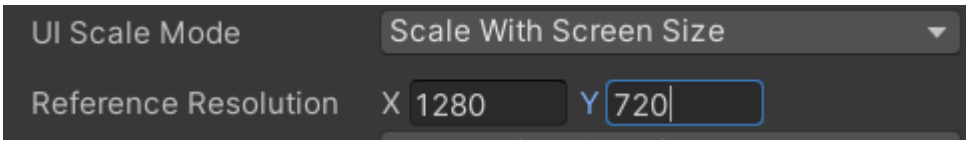
Esto significa que este tamaño debe escalarse en conjunto a alguna referencia, a alguna proporción.

Por defecto no hay ninguna proporción, es un tamaño de píxel constante:



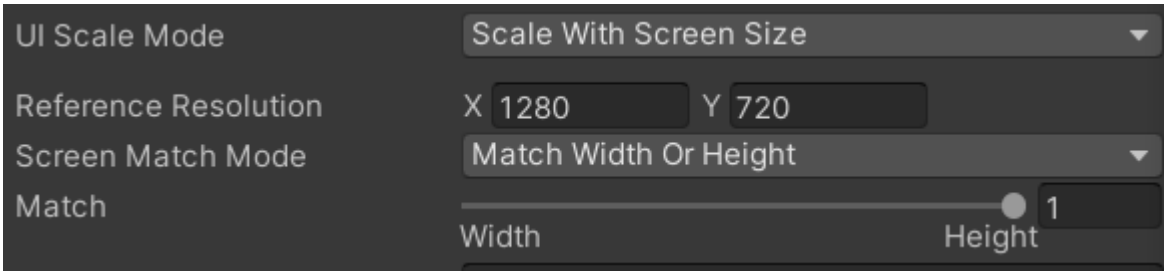
Por eso independientemente del tamaño de la vista de juego la interfaz conserva el suyo propio, es constante.

Pero podemos hacer que se escala tomando como referencia un tamaño de ejemplo, por ejemplo una pantalla de 1280x720, que es una proporción 16/9 (la seleccionamos también en la pantalla de Game):

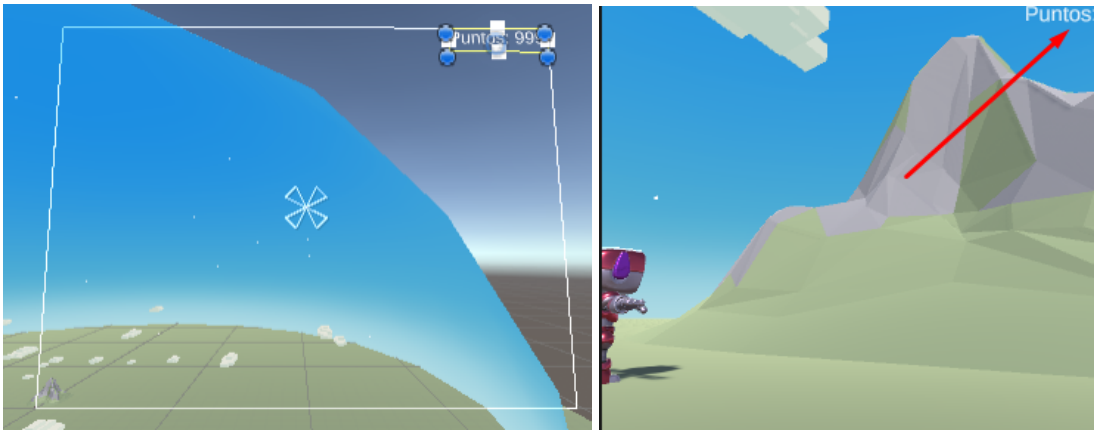


Ahora debemos decidir si esta interfaz se adaptará en proporción a la altura o ancho de la pantalla, se escalará o se encogerá.

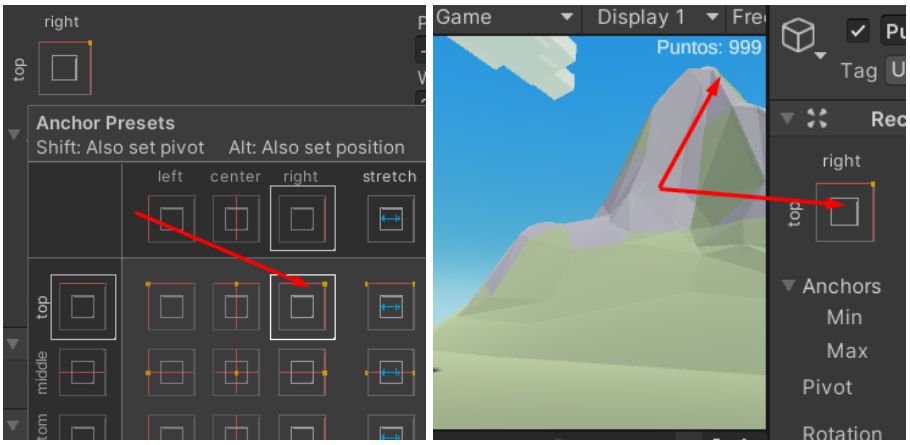
Para que el escalado se realice en función del alto de la pantalla seleccionaremos ***Match Height 1***. Con esto no importa que una pantalla sea muy estrecha, mientras tenga 720px de alto el tamaño del texto será el que estamos viendo en la previsualización:



Ahora nos llevaremos el texto a la esquina superior derecha del lienzo y probaremos qué ocurre si la pantalla es muy estrecha:

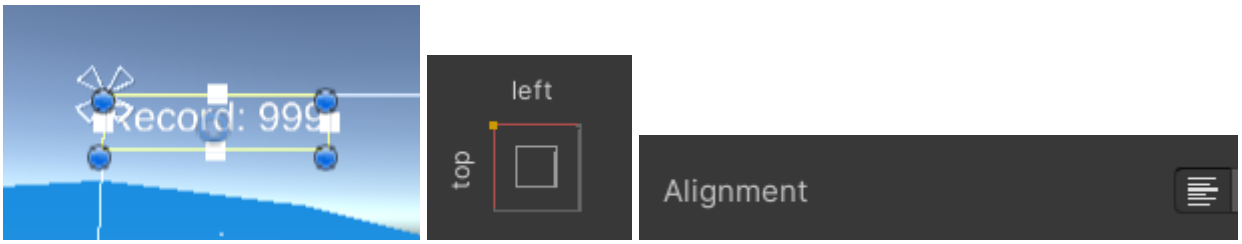


Veréis que desaparece. Eso es debido a la alineación con respecto a donde se escala, por defecto está puesto el centro, por lo que se intenta conservar esa alineación. Pero si lo alineamos a la esquina superior derecha sempre se conservará ese anclaje y se dibujará correctamente la interfaz independientemente del ancho de la pantalla:

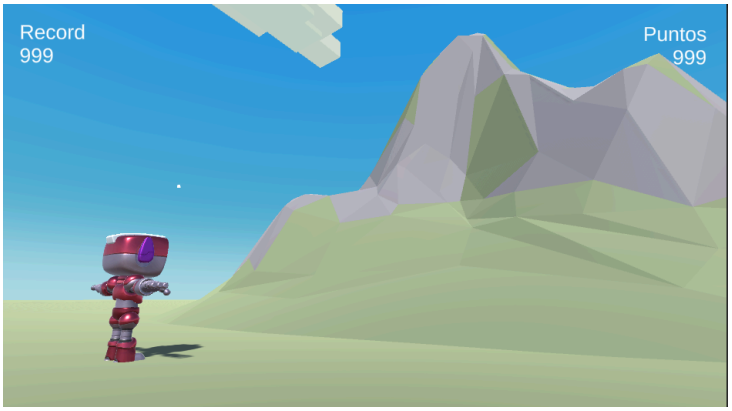


Con el ancla a la derecha os sugiero también alinear el texto a la derecha:

Ahora clonamos el objeto Puntos, le llamamos Record y nos lo llevamos al lado izquierdo, anclado arriba a la izquierda y alineado a la izquierda:



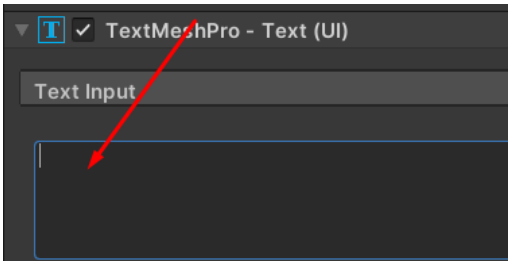
Jugueteando un poco con los márgenes y el tamaño del texto el resultado final está bastante logrado:



Solo nos faltaría mostrar un mensaje al morir que pusiera algo como “Presiona R para jugar de nuevo”.

Así que vamos a añadir un nuevo UI > TextMeshPro Text llamado *Reiniciar*, tamaño 72, etc.

Lo que vamos a hacer es borrar su contenido y establecerlo dinámicamente:



Ahora, justo cuando Jammo explota en el script ***RocaController***, buscaremos el objeto y cambiaremos su texto. Como el componente del texto se encuentra en la librería TextMeshPro tenemos que importarla arriba:

```
using TMPro;
```

Y ya simplemente al ocurrir la colisión, abajo del todo establecemos el texto:

```
// Creamos el mensaje de reinicio
GameObject.Find("Reiniciar").GetComponent<TMP_Text>().text = "Pulsa R para reiniciar";
```

Listo.

Marcador

Tenemos listo el HUD, es hora de configurar el sistema para detectar puntos y guardarlos en los marcadores.

¿Cuándo podemos sumar un punto? Evidentemente después de que Jammo haya saltado la roca. Tenemos un momento concreto que es perfecto, justo al desactivar el collider de la roca.

Ya sabemos el cuándo, ahora nos falta el dónde. ¿Dónde almacenamos las puntuaciones?

Posiblemente se os ocurra que el HUD sería un buen lugar, a fin de cuentas es quien muestra los puntos.

Sin embargo, os recomiendo no mezclar funcionalidades. Cada objeto tiene su utilidad y el HUD está para dibujar, no para gestionar. Y sí que tenemos un objeto pensado para gestionar, nuestro Manager.

Así que vamos a crear dos variables en ***ManagerController*** para la puntuación actual y el récord.

```
private int puntos, record;
```

El caso es que necesitamos dibujar estas variables en la interfaz, así que podemos crear un método que se encargue de ello:

```
using TMPro;

private TMP_Text puntosHUD, recordHUD;

void Start(){
    puntosHUD = GameObject.Find("Puntos").GetComponent<TMP_Text>();
    recordHUD = GameObject.Find("Record").GetComponent<TMP_Text>();
}

void DibujarPuntos(){
    puntosHUD.text = $"Puntos\n{puntos}";
    recordHUD.text = $"Récord\n{record}";
}
```

Y lo llamamos desde el principio:

```
void Start() {
    DibujarPuntos();
}
```

Lo siguiente será como no ir aumentando el marcador, para ello podemos crear un método en el propio manager:

```
void SumarPunto() {
    puntos++;
    if (puntos > record) { record++; }
    DibujarPuntos();
}
```

Eso sí, este método debe ser público para que podamos ejecutarlo desde otros scripts:

```
public void SumarPunto() { ... }
```

Para llamarlo podemos recuperar el objeto ***Manager*** justo al desactivar el collider de la roca en ***RocaController***:

```
void Update() {
        GameObject player = GameObject.FindGameObjectWithTag("Player");
```

```
        if (transform.position.z <= player.transform.position.z) {
            GetComponent<MeshCollider>().enabled = false;
            // Incrementamos el marcador
            GameObject.Find("Manager").GetComponent<ManagerController>().SumarPunto();
    }
}
```

Y aqui viene un truco. Si queremos llamar a un método público de un Script, no necesitamos recuperar su componente, podemos simplemente usar el método SendMessage del objeto:

```
// Incrementamos el marcador
GameObject.Find("Manager").SendMessage("SumarPunto");
```

Si ejecutamos el juego, efectivamente se incrementarán las puntuaciones… ¡Pero ojo! Como este código se encuentra dentro del Update, y este se ejecuta cada fotograma pues se llama un montón de veces.

¿Se os ocurre alguna forma de evitar este pequeño error y limitarlo a una sola vez? La respuesta se encuentra en el propio código. Recordad que estamos desactivando el ***MeshCollider*** de la roca, pero éste inicialmente está activado. Podríamos incrementar el marcador solo cuando el ***MeshCollider*** esté activo y luego desactivarlo:

```
if (GetComponent<MeshCollider>().enabled) {
    //Incrementamos el marcador una vez, antes de desactivar el collider
    GameObject.Find("Manager").SendMessage("SumarPunto");
}
GetComponent<MeshCollider>().enabled = false;
```

Una cosa menos.

Persistencia

Nos falta un detalle en nuestro marcador, y es que el récord debe guardarse en la memoria y mostrarse.

Por suerte para nosotros Unity incluye un módulo llamado PlayerPrefs que permite almacenar y recuperar datos en el disco duro.

El funcionamiento es extremadamente sencillo.

Antes de dibujar el récord por primera vez vamos a intentar recuperarlo del disco mediante una clave (en el ***ManagerController***):

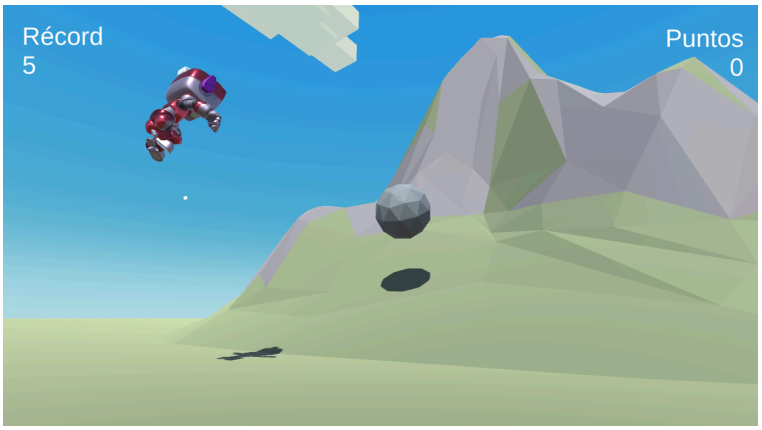
```
// Recuperamos el record del disco duro o establecemos 0 por defecto
record = PlayerPrefs.GetInt("Record", 0);
```

Luego lo escribiremos en el disco siempre que se supere el record actual:

```
if (puntos > record) {
    record++;
```

```
    PlayerPrefs.SetInt("Record", record);
}
```

Y con esto tenemos listo el sistema de récord persistente.



Sonidos

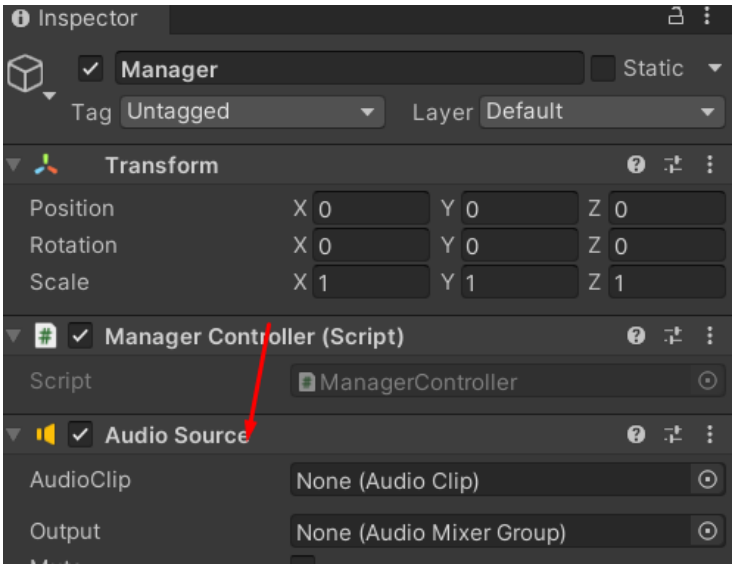
¿Qué sería de un videojuego sin los efectos de audio y una buena canción de fondo?

En esta lección os proporciono un paquete llamado ***Audios*** que podéis descargar en los recursos e instalarlo en el proyecto, contiene :

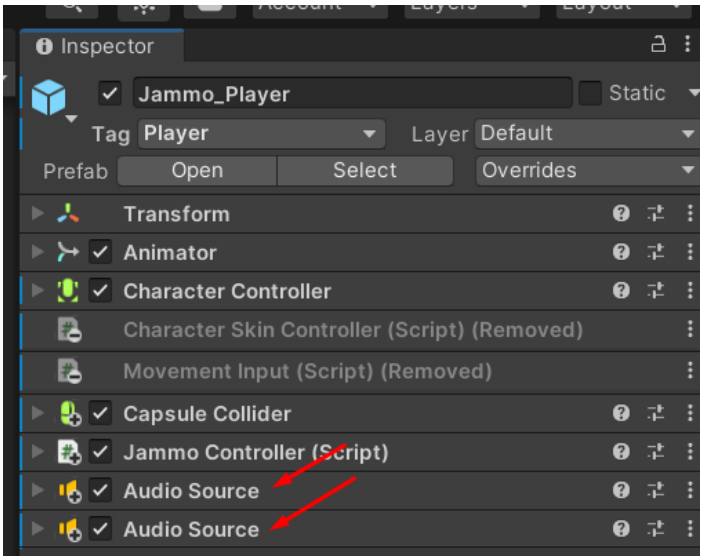
- Una música de fondo
- Una melodía al morir
- Un efecto de salto
- Un efecto de punto

Para reproducir sonido necesitamos un componente llamado ***AudioSource***. Este componente puede reproducir un único sonido a la vez. Necesitamos por lo menos dos audio sources, uno para la música y la melodía, y otro para los efectos.

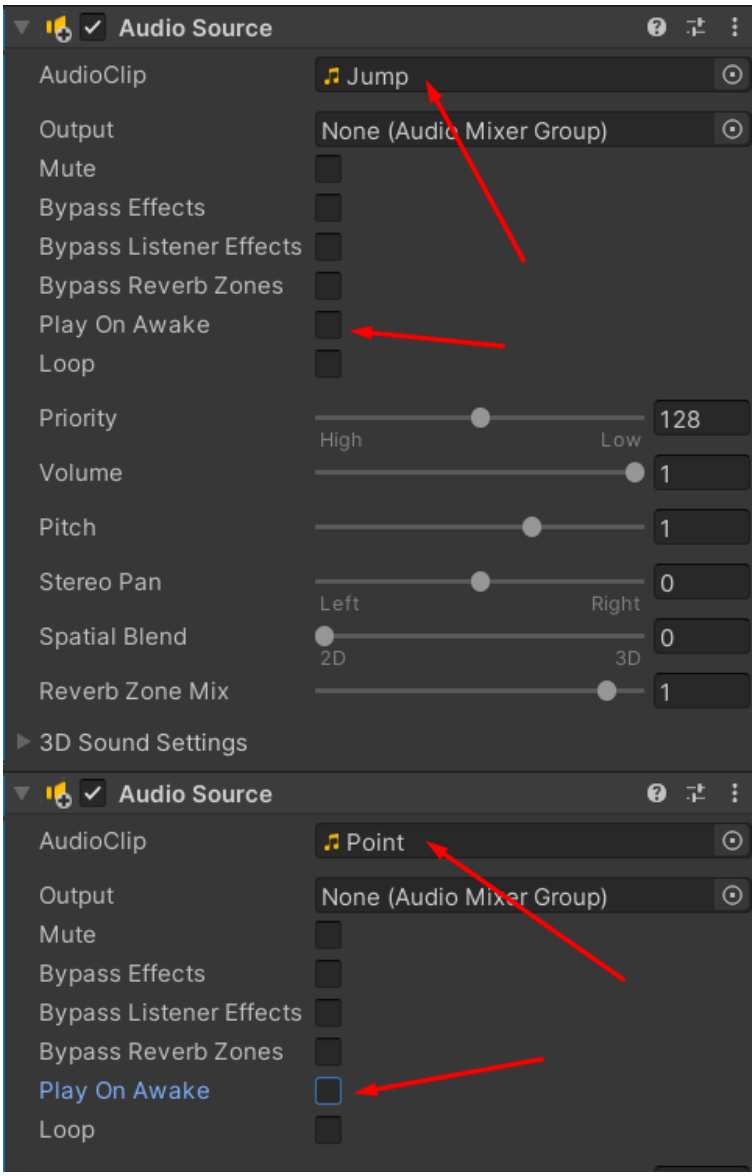
La música y la melodía no se van a solapar, la idea es que suene la música y al morir se pare y suene la melodía. Podemos reproducir ambos audios desde el ***Manager***:



En cuanto a los efectos, ambos vienen determinados por acciones del jugador (saltar y conseguir punto), por lo que podemos añadirle otro audiosource. El problema es que, como saltar y conseguir punto son acciones encadenadas, eventualmente sonarán casi a la vez, por lo que necesitamos en realidad dos audiosources, uno para cada efecto:



Podemos establecer el clip de audio manualmente o mediante código. Para los efectos asignaremos uno a cada audiosource y desmarcamos la opción Play on Awake, ya que eso reproduce inicialmente los sonidos:



Ahora en el script ***JammoController*** vamos a crear dos referencias para utilizar estos audioSources

```
public AudioSource jump, point;
```

Y los asignamos manualmente:

Podemos reproducir el audio de este componente mediante su método Play, es muy fácil. Buscamos el momento idóneo y lo llamamos.

El de salto:

```
if (!isJumping && controller.isGrounded){
      isJumping = true;
      moveDirection.y = jumpSpeed;
      nim.SetBool("jumping", true);

      jump.Play();  // reproducimos el sonido
}
```

Y el de punto justo al incrementar el marcador en ***RocaController***, a través de componente audiosource:

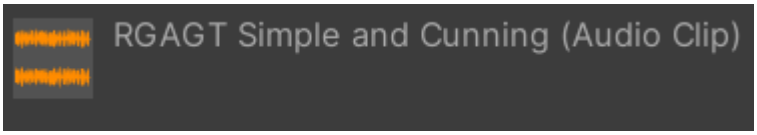
```
if (GetComponent<MeshCollider>().enabled){
    // Incrementamos el marcador una vez, antes de desactivar el collider
    GameObject.Find("Manager").SendMessage("SumarPunto");

   // Reproducimos el sonido de punto del jugador
   player.GetComponent<JammoController>().point.Play();

}
```

Podríamos haber creado el audiosource directamente en la roca, pero he preferido hacerlo así.

En cualquier caso los efectos ya estarán funcionando perfectamente.

Respecto a la música de fondo y el audio os voy a enseñar como reproducir ambos con el mismo audiosource.

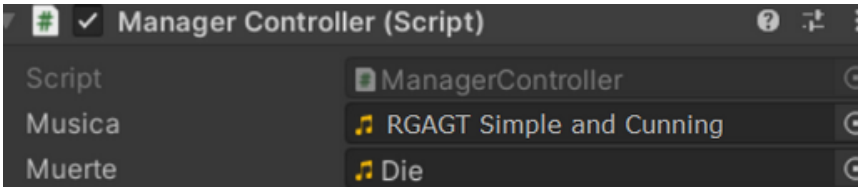
De vuelta al ***ManagerController*** vamos a reproducir la canción al empezar el juego, ésta tiene el nombre "*RGAGT Simple and Cunning*" y si la seleccionamos en el proyecto apreciamos que es de tipo ***Audio Clip***:



Precisamente ***AudioClip*** es un tipo de dato que podemos establecer como propiedad, así que vamos a añadir dos:

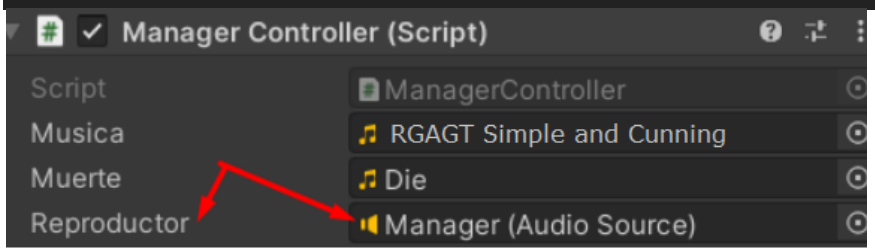
```
public AudioClip musica, muerte;
```

Los asignamos:



Ahora lo que debemos hacer es cargar el clip en el audiosource y empezar a reproducirlo, para ello creamos una referencia:

```
public AudioSource reproductor;
```



Y en el start:

```
// Cargamos e iniciamos la música
reproductor.clip = musica;
reproductor.Play();
```

Solo nos falta hacer lo propio en el momento de morir, abajo del todo del ***RocaController***:

```
// Cambiamos el clip del reproductor y reproducimos la música de muerte
ManagerController managerController = GameObject.Find("Manager").GetComponent<ManagerController>();
managerController.reproductor.clip = managerController.muerte;
managerController.reproductor.Play();
```

Con la música cambiará perfectamente y habremos acabado esta parte.

Refactorización

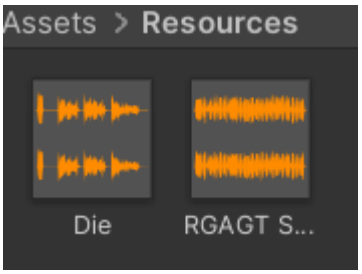
En programación, refactorizar el código es optimizar el funcionamiento sin afectar al comportamiento.

Quiero que hagamos una pequeña refactorización del **Manager** para reproducir sonidos de forma más sencilla, puesto que ahora mismo tenemos código redundante, vamos a simplificarlo.

En primer lugar, vamos a crear un método ***ReproducirClip*** en el ***ManagerController***

```
public void ReproducirClip(string nombre){

}
```

Fijaros que vamos a recibir una cadena y no un clip, eso es porque vamos a utilizar el buscador de recursos para cargar los clips a partir del nombre, pero para ello necesitamos llevarnos ambos sonidos a un lugar especial, una nueva carpeta debemos llamar ***Resources***:



El código para buscar recursos en esta carpeta y cargarlos es el siguiente:

```
public void ReproducirClip(string nombre){
    AudioClip clip = Resources.Load<AudioClip>(nombre);
    reproductor.clip = clip;
    reproductor.Play();
}
```

Con esto podemos ahorrarnos las dos propiedades con los clips:

```
public AudioClip musica, muerte;
```

Y solo debemos llamar al método con el nombre del recurso:

```
// Cargamos e iniciamos la música
ReproducirClip("RGAGT Simple and Cunning");
```

De la misma forma ya nos avisa que debemos actualizar lo propio para reproducir la melodía de muerte en ***RocaController*** a partir de enviar un SendMessage con el método y el argumento:

```
// Cambiamos el clip del reproductor y reproducimos la música de muerte
ManagerController managerController = GameObject.Find("Manager").GetComponent<ManagerController>();
managerController.reproductor.clip = managerController.muerte;
managerController.reproductor.Play();
```

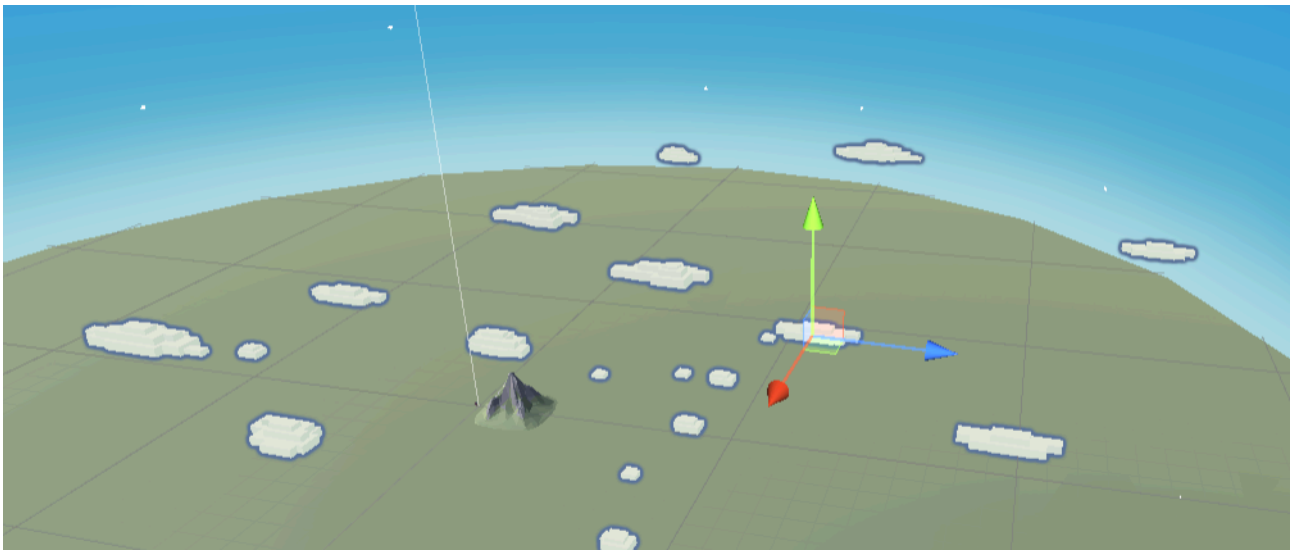
```
// Reproducimos la música de muerte
GameObject.Find("Manager").SendMessage("ReproducirClip", "Die");
```

Y así hemos refactorizado el código, mismo comportamiento pero diferente funcionamiento.

Nubes

En esta lección vamos a programar un sistema que mueva las nubes para añadir dinamismo.

Las nubes tienen un recorrido que si observamos desde el punto de vista de la cámara podemos considerar que comienza en el punto (25, 25, 295) y finaliza en (4, 27, -260) al moverla por su eje Z. Ese punto final es justo cuando la última nube está por detrás del personaje.



Vamos a posicionar la nube en su punto inicial (25, 25, 295) e incrementaremos su eje X en un nuevo script **CloudsController**:

```
public float speed = 6f;
void Update() {
    transform.position += Vector3.back * speed * Time.deltaTime;
}
```

Con esto ya se mueve, el problema es que eventualmente la nube desaparece. Simplemente debemos comprobar si se ha superado la posición Z y reiniciarla a su inicio. Para ello podemos capturar su posición inicial en una variable y reestablecerla:

```
public float speed = 6f;
private Vector3 inicio, final;

void Start() {
    inicio = transform.position;
    final = new Vector3(4, 27, -260);
}

void Update() {
    transform.position += Vector3.back * speed * Time.deltaTime;
    if (transform.position.z < final.z) {
        transform.position = inicio;
    }
}
```

Si aumentamos mucho la velocidad de las veremos como se va reiniciando la posición:

Pero este sistema es demasiado monótono, por eso vamos a asignar una posición Z aleatoria de inicio entre el mínimo y máximo. Para ello necesitamos generar un número entre ese rango, y eso lo conseguiremos utilizando la función Random.Range de Unity:

```
void Start() {
    inicio = transform.position;
    final = new Vector3(4, 27, -260);

    // Generamos un lugar aleatorio inicial para el eje Z
    float zRandom = Random.Range(final.z, inicio.z);
    transform.position = new Vector3(inicio.x, inicio.y, zRandom);
}
```

Escenario

En esta lección el objetivo es hacer más atractivo el escenario. Tenéis vía libre para ser creativos y añadir elementos a vuestro gusto, montañas de fondo, adornos en el suelo, piedras, árboles… Lo que se os ocurra, eso sí, debéis añadirlos dentro del objeto ***Entorno*** para tenerlos bien organizados:

Postprocesado

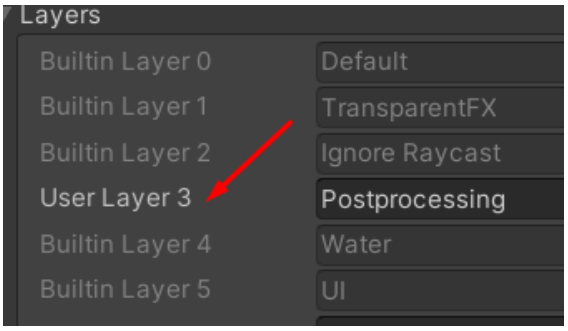
Con el escenario acabado vamos a añadir unos detalles de postprocesado para darle el toque final.

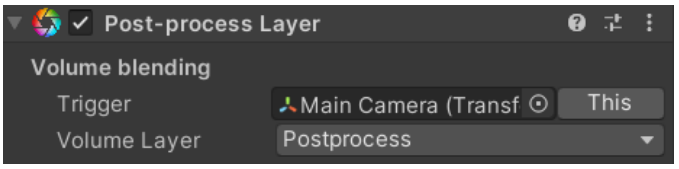
Esto ya enseño a hacerlo en mi curso de fundamentos, así que vamos a ir por faena.

Para activar el postprocesado debemos instalar el paquete del registro ***Post Processing***

En la cámara vamos a añadir un componente ***Post-process Layer***:

Vamos a crear una capa ***Postprocess antes que la UI*** y la asignaremos al ***Volume Layer***, básicamente no queremos que el postprocesado afecte a la interfaz:

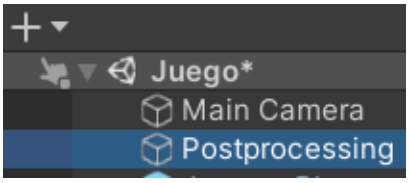


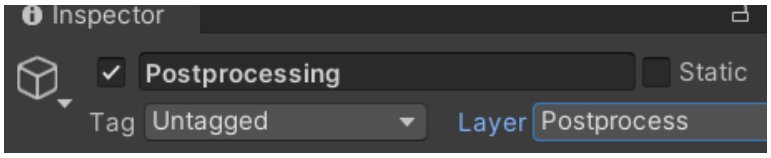


Activamos el antialising FXAA.

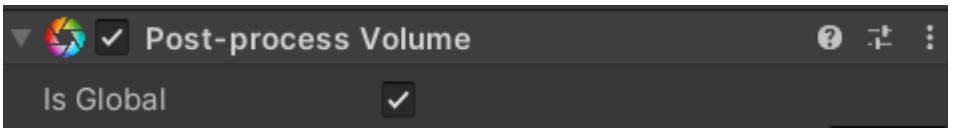
Ahora vamos a por la corrección de color para hacer que el escenario se vea más vivio.

Crearemos un objeto llamado ***Postprocessing*** y le otorgaremos la capa ***Postprocess***:

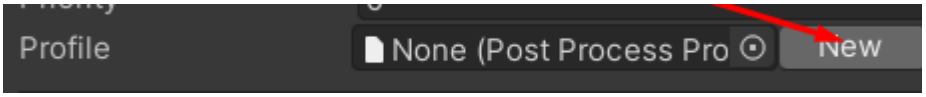




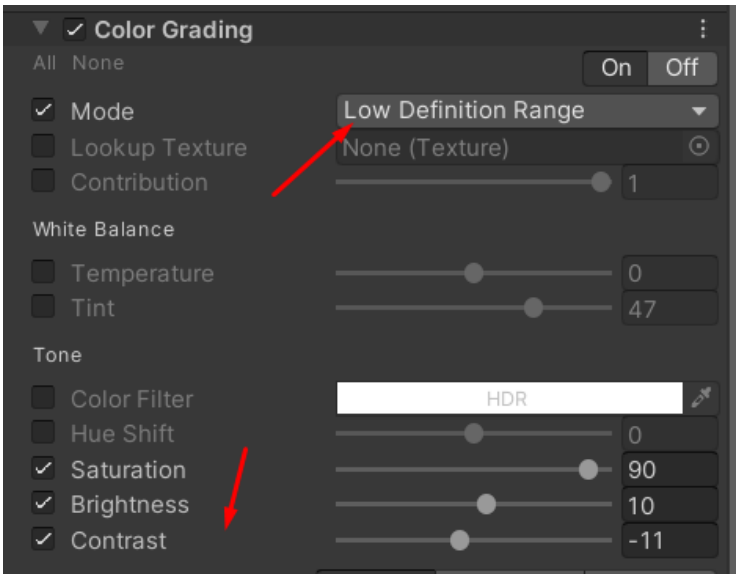
Añadiremos un componente ***Post Process Volume*** y marcaremos la casilla ***Is Global***:



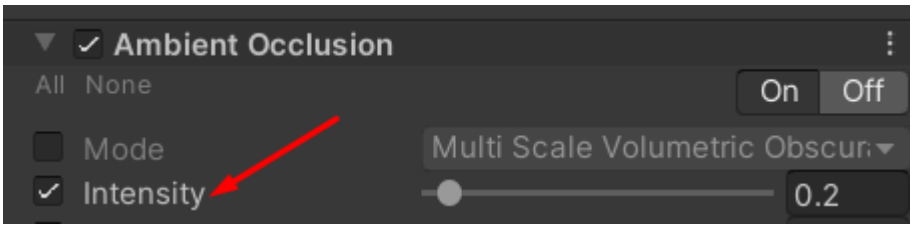
Generaremos un nuevo ***Profile***:



Y añadiremos un efecto ***Color Grading***: con un ***Low Definiton Range*** y configuramos la saturación, brillo, contraste y mezclas de color al gusto:

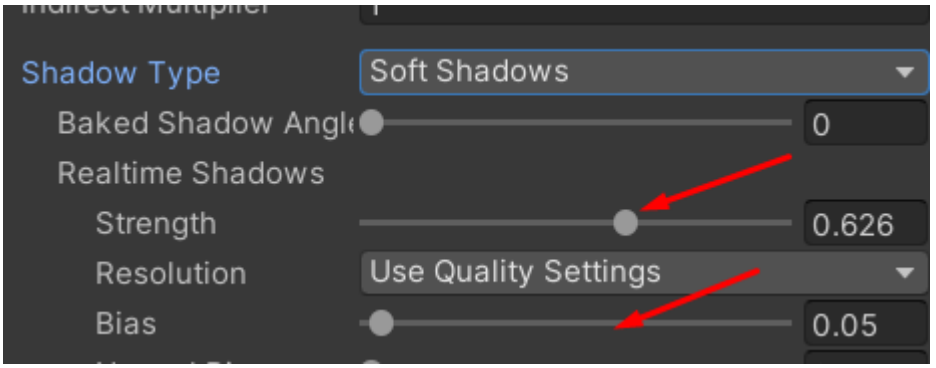


Con otro filtro de Oclusión ambiental podemos remarcar ligeramente las sombras del escenario:



Sin abusar demasiado del postprocesado el resultado es mucho mejor.

El último detalle gráfico para mí sería suavizar un poco la sombra:



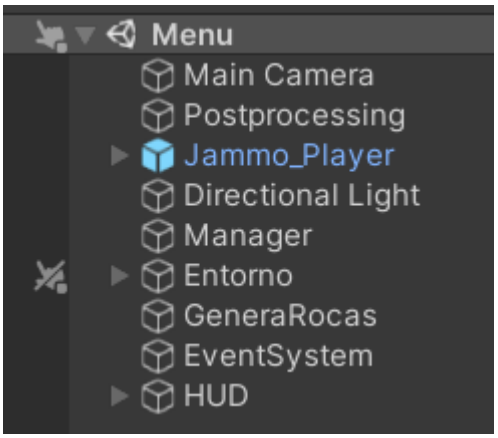
Y juego listo:

Portada

Si generamos el ejecutable del videojuego en este punto y lo exportamos notaremos que empieza a ejecutarse en el mismo momento que la pantalla de crédito de Unity.

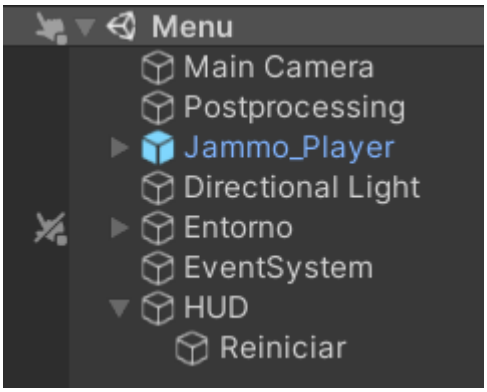
Para evitar esta situación podemos crear una escena de portada, de ahí podemos poner el título del videojuego, un botón para empezar a jugar y nuestro nombre como creadores.

Así que vamos a hacer una cosa, vamos a duplicar la escena Juego guardando como (Save As) y el nombre ***Portada***:

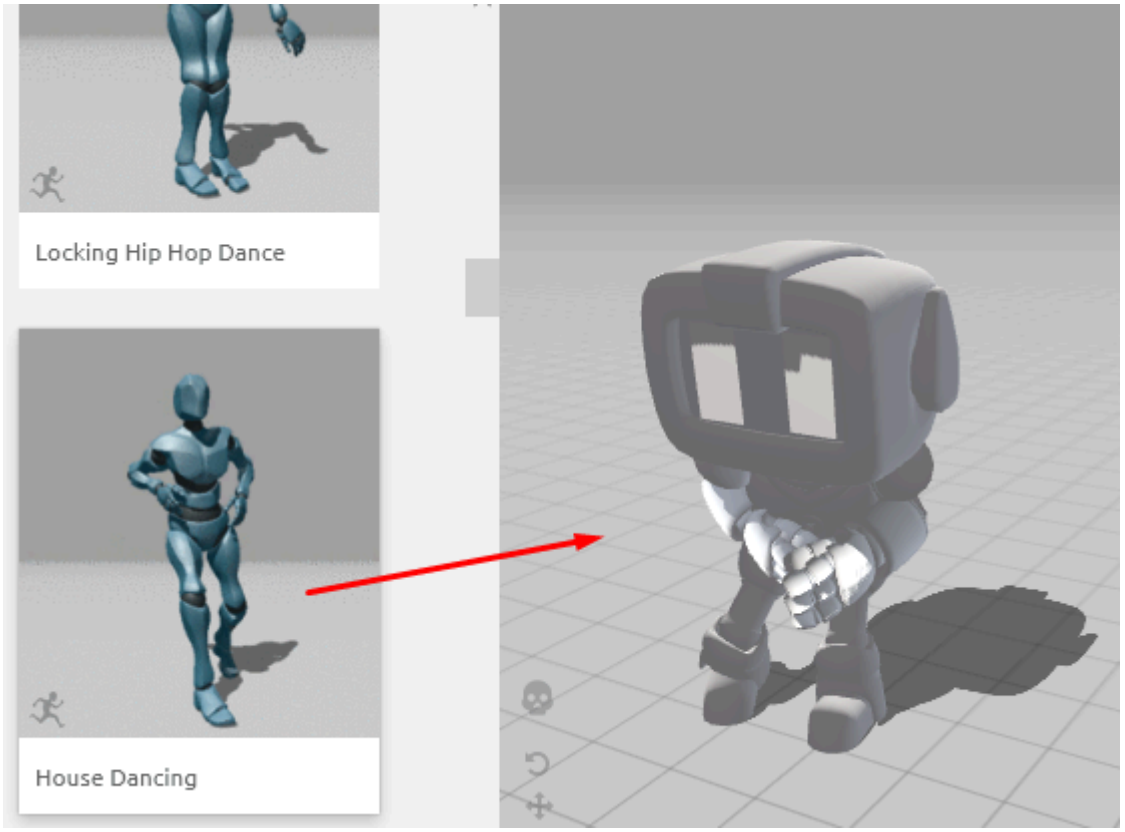


En esta escena vamos a enfocar la cámara justo en la cima de la montaña:

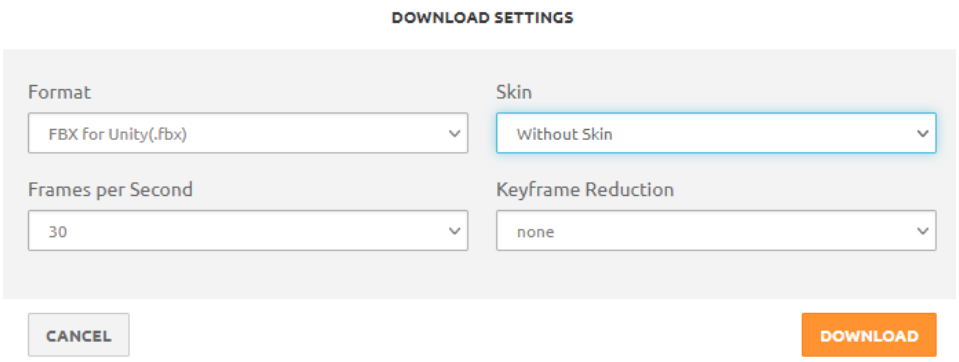
Y vamos a borrar los objetos del **manager**, el **generador** de rocas y del HUD **Puntos** y **Record**:



Vamos a proceder a descargar una nueva animación de Mixamo, donde el personaje se encuentre bailando, os la dejaré en los recursos por si en algún momento se pierde (House Dancing)e:

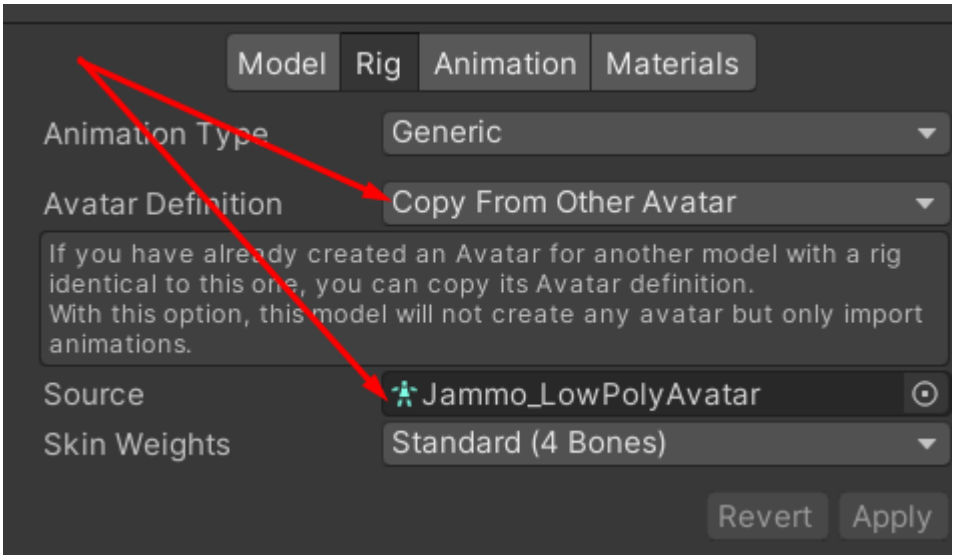


La descargamos en formato Unity sin Skin:

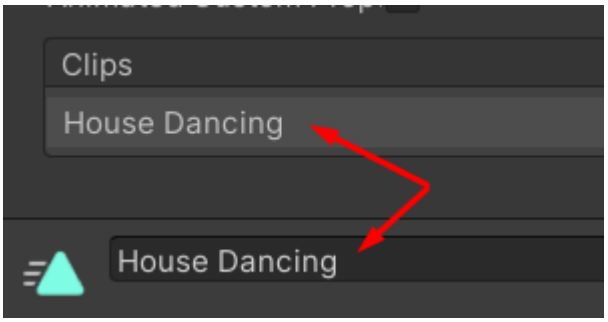


Y arrastramos el fichero a la carpeta Animations:

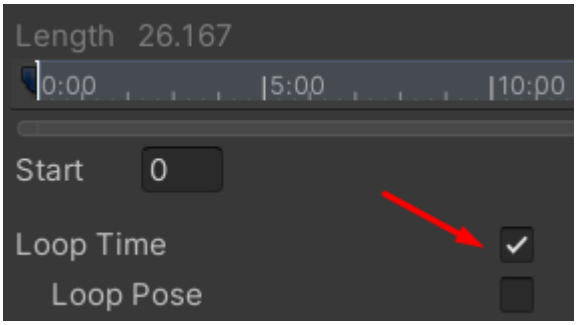
En la pestaña Rig seleccionaremos la definición del avatar "**copy from other**" y buscaremos **JammoLowPolyAvatar** y daremos a Apply:



Nos aseguramos de que la pestaña ***Animation*** el clip y la ***animación*** concuerden:

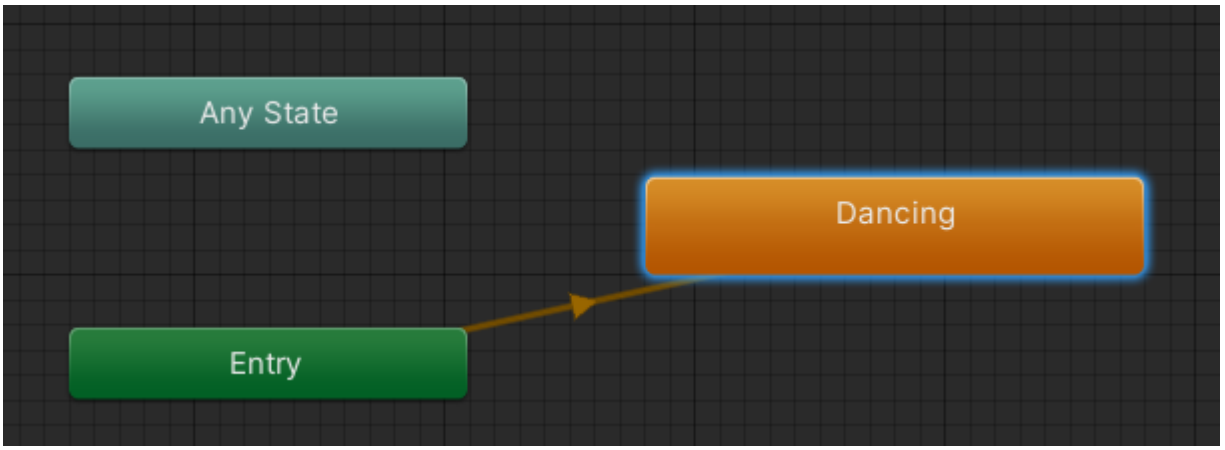


Activaremos el **loop** y presionamos **Apply** para que nunca pare de bailar:



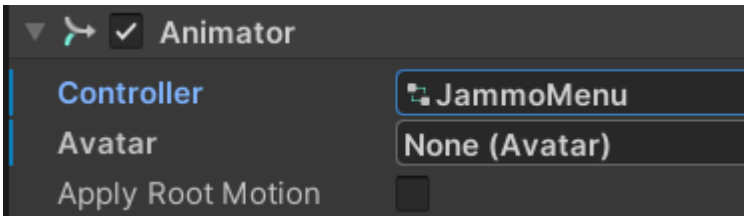
Y procederemos a crear un nuevo **Animation Controller** al que llamaremos **JammoMenu**:

Le hacemos doble clic y creamos un nuevo estado Dancing, le asignamos la animación de bailar ***House Dancing***:



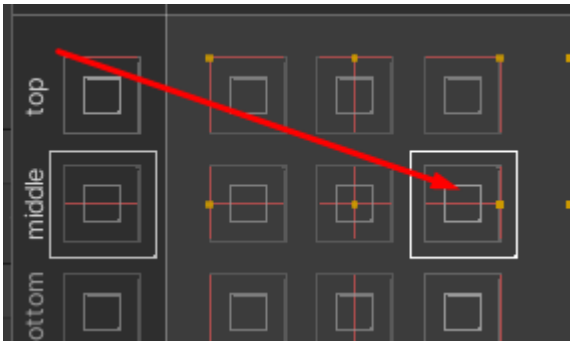
Traemos a Jammo más al frente y lo rotamos para que mire hacia la cámara:

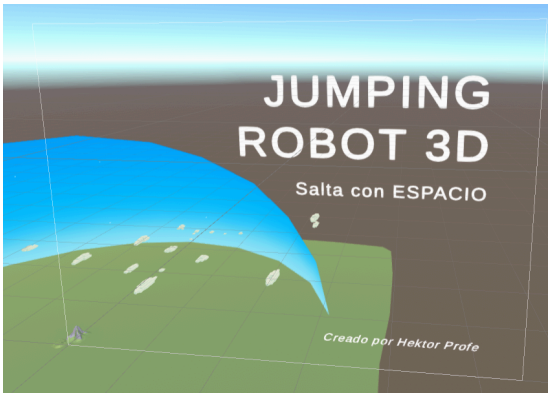
Cambiamos el controlador de la animación al nuevo:



Y ahí lo tendremos bailando graciosamente:

Solo falta cambiar jugar con el objeto reiniciar, lo clonamos un par de veces y generamos un interfaz inicial alineando los elementos a un lado (y el texto), importante:



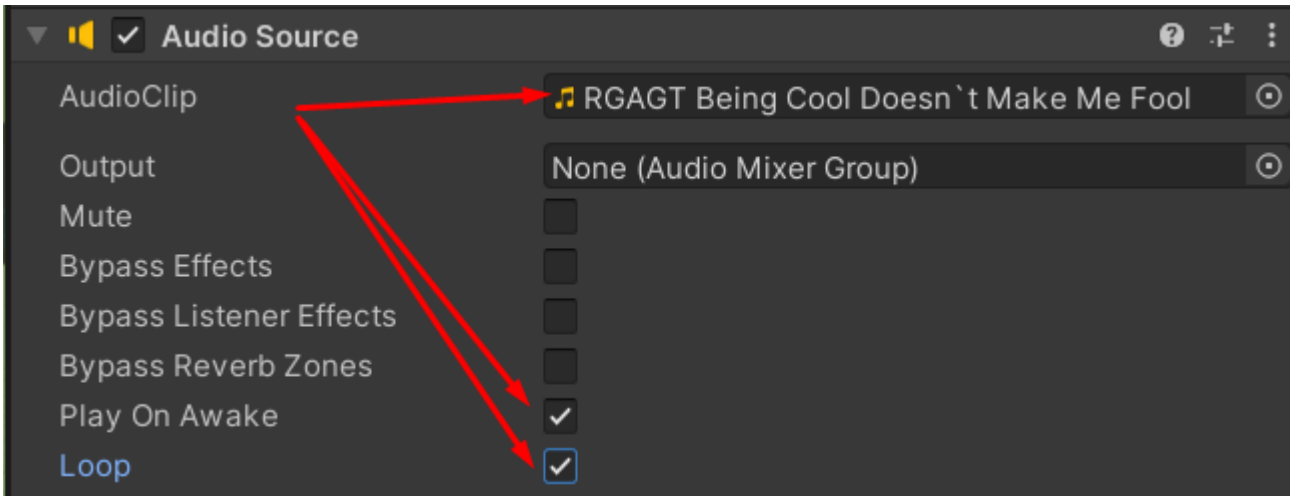


Solo nos falta crear un script para cambiar de escena y si queremos incluso reproducir una canción… ¿Pero eso no lo teníamos en el objeto ***Manager***? Así es, pero ese objeto está pensado para manejar el juego, no el menú.

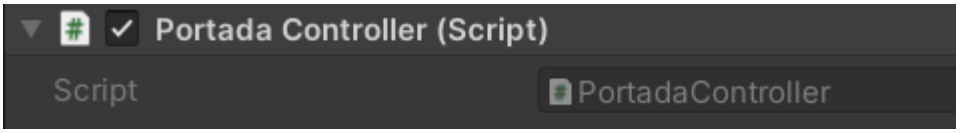
Así que vamos a crear un objeto **PortadaManager**:

Podemos añadir una canción en un AudioSource, cualquier canción que queráis o importar la que os dejo en los recursos.

Marcaremos ***Play on Awake*** para sonar desde el principio y ***Loop*** para que se repita al acabar:

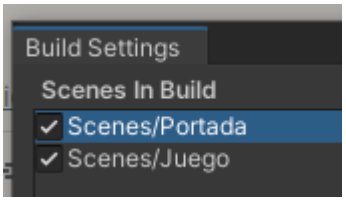


Ahora en un nuevo script ***PortadaController*** detectamos si se presiona espacio y cambiar de escena:



Si ponemos el juego en marcha y presionamos espacio nos saltará un error:

Debemos añadir ambas escenas a los **BuildSettings**, fácilmente desde **File** arrastramos las escenas a la pantalla, poniendo primero **Portada** y luego **Juego** para que se ejecuten en ese orden:



Con esto ya podemos probar el juego y debería funcionar perfectamente.

Transición

Una transición es un efecto que añade un suavizado de movimiento al cambiar de una imagen a otra para que el cambio no sea tan brusco. En la assets store encontramos multitud de paquetes para este propósito, vamos a descargar e importar uno llamado **Simple Fade Scene Transition System** *ref*, en caso de que no lo encontraréis os lo adjunto en los recursos.

Este paquete nos permite generar una transición Fade, de las que desaparecen y aparecen progresivamente entre escenas.

Una vez instalado el paquete lo único que deberemos hacer es llamar a un método en lugar de cambiar de escena manualmente:

```
public class PortadaController : MonoBehaviour {
      void Update() {
            if (Input.GetKeyDown(KeyCode.Space)) {
```

```
                // Escena, color transición, duración
                Initiate.Fade("Juego", Color.black, 0.5f);
                // SceneManager.LoadScene("Juego");
            }
        }
}
```

Y listo, aunque yo prefiero desactivar el HUD antes de la transición para generar un efecto más suave:

```
GameObject.Find("HUD").SetActive(false);
Initiate.Fade("Juego", Color.black, 2f);
```

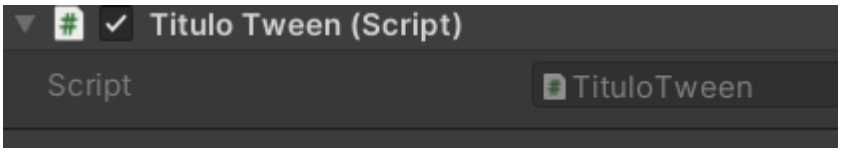
Tweening

Rizando el rizo de los elementos que podemos conseguir para hacer aún más atractivo el juego tenemos el tweening, una técnica de interpolación de propiedades para generar efectos muy logrados.

La interpolación es un cálculo que dado un valor inicial y otro final, permite calcular el valor intermedio en un instante del tiempo. Esto se realiza mediante funciones matemáticas pero nosotros vamos a ahorrarnos los cálculos y en su lugar vamos a utilizar un paquete muy famoso de la Asset Store llamado ***DoTween*** *ref*, os dejo un paquete de backup en los recursos.

Al importarlo nos pedirá configurarlo, y nos mostrará su panel donde encontrar documentación y ejemplos.

La versión gratuita no incluye editor gráfico así que vamos a realizar los tweens mediante código. Vamos a empezar creando un script llamado ***TituloTween*** al ***TextMeshPro*** del canvas de la portada:

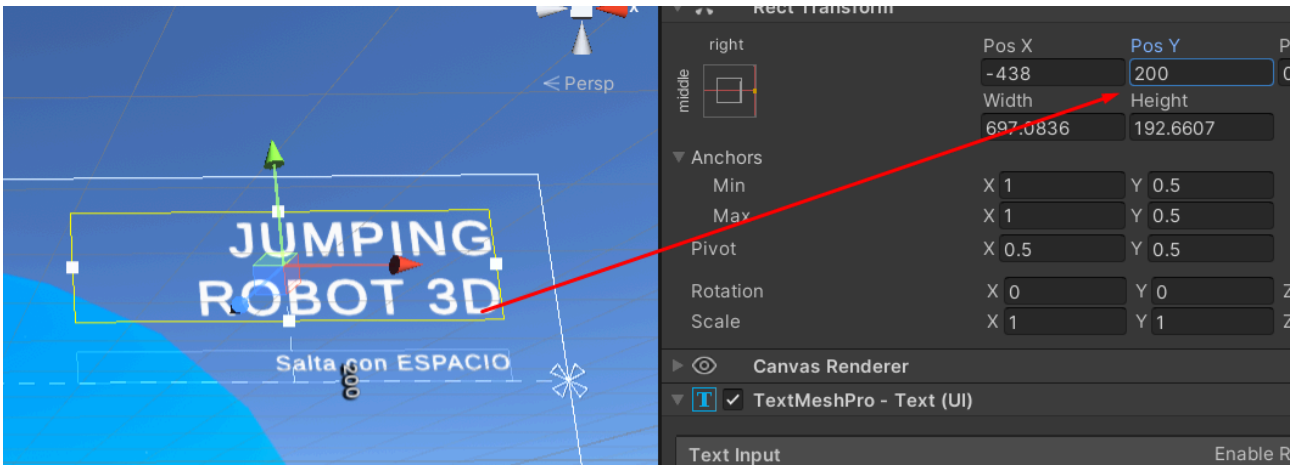


Vamos a mover el título del videojuego 100 píxeles hacia abajo a lo largo de un 1 segundo:

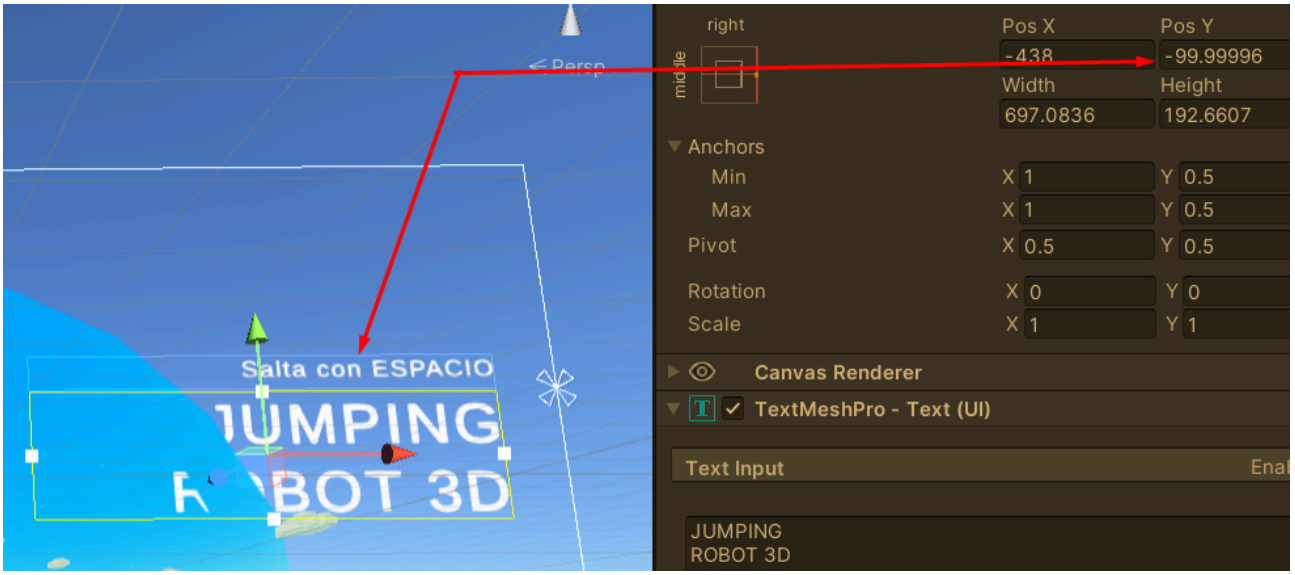
```
using DG.Tweening;

public class TituloTween: MonoBehaviour
{
    void Start()

        transform.DOLocalMoveY(-100, 1f);
    }
}
```

Tened en cuenta que estos 100 píxeles son respecto al canvas. Si inicialmente éste se encuentra en la

posición Y 200:



Al final de la animación estará en Y ⇒ 100:



Lo mejor del tweening es que se le pueden añadir ***easing functions*** para cambiar el comportamiento del efecto, por ejemplo un rebote:

```
transform.DOLocalMoveY(-100, 1f).SetEase(Ease.OutBounce);
```

Para ver representaciones visuales de easing tenéis a vuestra disposición la web https://easings.net/.

Pero no solo podemos configurar easing, sino también un delay para que se espere un tiempo antes de empezar el movimiento:

```
transform.DOLocalMoveY(-100, 1f).SetEase(Ease.OutBounce).SetDelay(2f);
```

Vamos a añadir un tween diferente para el texto de los créditos ***CreditosTween***. En lugar de un movimiento utilizaremos un fade para que aparezca progresivamente el texto. Esto lo podemos hacer gracias a la transparencia del color del texto.

Por defecto no tenemos transparencia, podemos hacer que sea totalmente transparente (alpha 0%) así:

```
using UnityEngine;
using DG.Tweening;
using TMPro;

public class CreditosTween : MonoBehaviour {
    void Start() {
        GetComponent<TMP_Text>().alpha = 0f;
    }
}
```

Ahora la parte buena, con un tween ***DOFade*** le establecemos que cambie a alpha 100% en un segundo:

```
 GetComponent<TMP_Text>().alpha = 0f;
 GetComponent<TMP_Text>().DOFade(1f, 1f);
```

Si le añadimos un delay podemos enlazar los efectos del título rebotando y los créditos apareciendo:

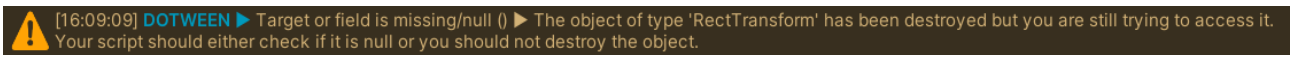
```
GetComponent<TMP_Text>().DOFade(1f, 1f);
```

Finalmente para el texto de Iniciar vamos a hacer algo aún más divertido ***IniciarTween***:

```
public class IniciarTween : MonoBehaviour {
   void Start(){
      GetComponent<TMP_Text>().alpha = 0f;
      GetComponent<TMP_Text>().DOFade(1f, 1f).SetDelay(2f);
      transform.DOPunchPosition(
            Vector3.up * 25, .5f, 0, 0).SetDelay(3.5f).SetLoops(-1);
     }
}
```

El resultado final es simple pero elegante:

Solo una cosa, si cambiamos de escena DOTween nos mostrará un mensaje:



Esto es porque hemos borrado un objeto con un Tween activo, no pasa nada, pero podemos evitar la situación si finalizamos correctamente la librería justo antes de cambiar de escena en ***PortadaManager***:

```
using DG.Tweening;
public class PortadaController : MonoBehaviour {
    void Update() {
        if (Input.GetKeyDown(KeyCode.Space)) {
            // Finalizamos los posibles tweens
            DOTween.Clear(true);
```

Ahora sí, todo está perfecto.

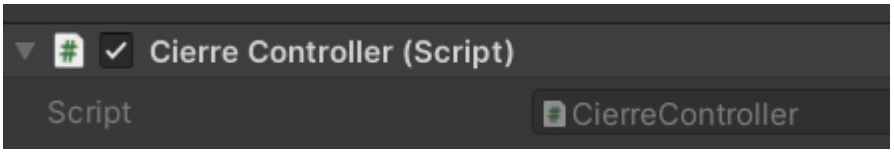
Cierre

Hemos llegado casi casi al final pero todavía no hemos programado una forma de cerrar el juego más allá de **Alt + F4**.

Instintivamente para salir siempre se utiliza la tecla Escape así que vamos a capturarla y a ejecutar la instrucción de salida **Application.Quit()**.

Pero si os lo paráis a pensar debemos capturar la tecla ESC en las dos escenas. Podemos modificar los scripts **PortadaController** y **ManagerController** pero estaríamos repitiendo código. Quiero aprovechar esta situación para despedir el curso introduciendo una instrucción que nos permitirá conservar un objeto entre escenas.

Para mantener la explicación sencilla vamos a crear en la portada un objeto **Cierre** con el script **CierreController**.




```
public class CierreController : MonoBehaviour {
    void Update() {
        if (Input.GetKeyDown(KeyCode.Escape)) {
            print("Saliendo del juego");
            Application.Quit();
        }
    }
}
```

Con esto ya podemos cerrar el juego desde la portada, aunque en el modo editor la instrucción no hace nada en el ejecutable final sí, podemos probarlo generando de nuevo el distribuible con **Control+B**.

He puesto el **print** para que veamos que efectivamente se ejecuta el código en el modo editor:

Aún así os voy a enseñar un truco para cerrar el juego en el editor mediante una condición de compilación:

```
using UnityEditor;

if (Input.GetKeyDown(KeyCode.Escape)){
...
#if UNITY_EDITOR
EditorApplication.isPlaying = false;
#endif
}
```

Con este ***if*** tan especial podemos comprobar si estamos en el modo edición y cerrar el juego. La condición es a nivel de compilación porque el editor no existe en el ejecutable.

En cualquier caso, tenemos el objeto ***Cierre***, pero este al cambiar a la escena de Juego se borra, podemos verlo en la jerarquía.

Para que un objeto no se borre al cargar una escena tenemos a nuestra disposición un método llamado ***DontDestroyOnLoad*** al que debemos pasarle la instancia del objeto que no queremos destruir.

Este código suele ejecutarse en el método Awake, antes de iniciar la escena:

```
public class CierreController : MonoBehaviour{
    void Awake() {
        // this es la palabra clave que sustituye a la instancia del script
        DontDestroyOnLoad(this);
    }
```

Con esta modificación la instancia del objeto ***Cierre*** ya no se borrará y podremos salir con **Escape** también en la escena **Juego**:

Con esto ya lo tenemos, pero os quiero avisar de algo.

En nuestro juego en ningún momento volvemos a la portada, pero si lo hiciésemos, el objeto ***Cierre*** se duplicaría. Es lo que tiene que no se borre y se vuelva a crear junto con la escena. Si en algun momento necesitáis un **objeto persistente y único**, os dejo un tutorial que hice para mi canal de Youtube donde enseño a implementarlo mediante el patrón Singleton: https://www.youtube.com/watch?v=RtCOomI_txo

Puntero

Finalmente el último detalle del juego, esconder el puntero ratón. Esto es por una cuestión de practicidad, como nuestro juego no hace uso de él, mejor no mostrarlo.

Para esconderlo fácilmente desde el script ***PortadaController*** lo podemos desactivar en el Start:

```
void Start(){
   Cursor.visible = false;
}
```

Y ya está, no más puntero molestando y videojuego oficialmente terminado.