

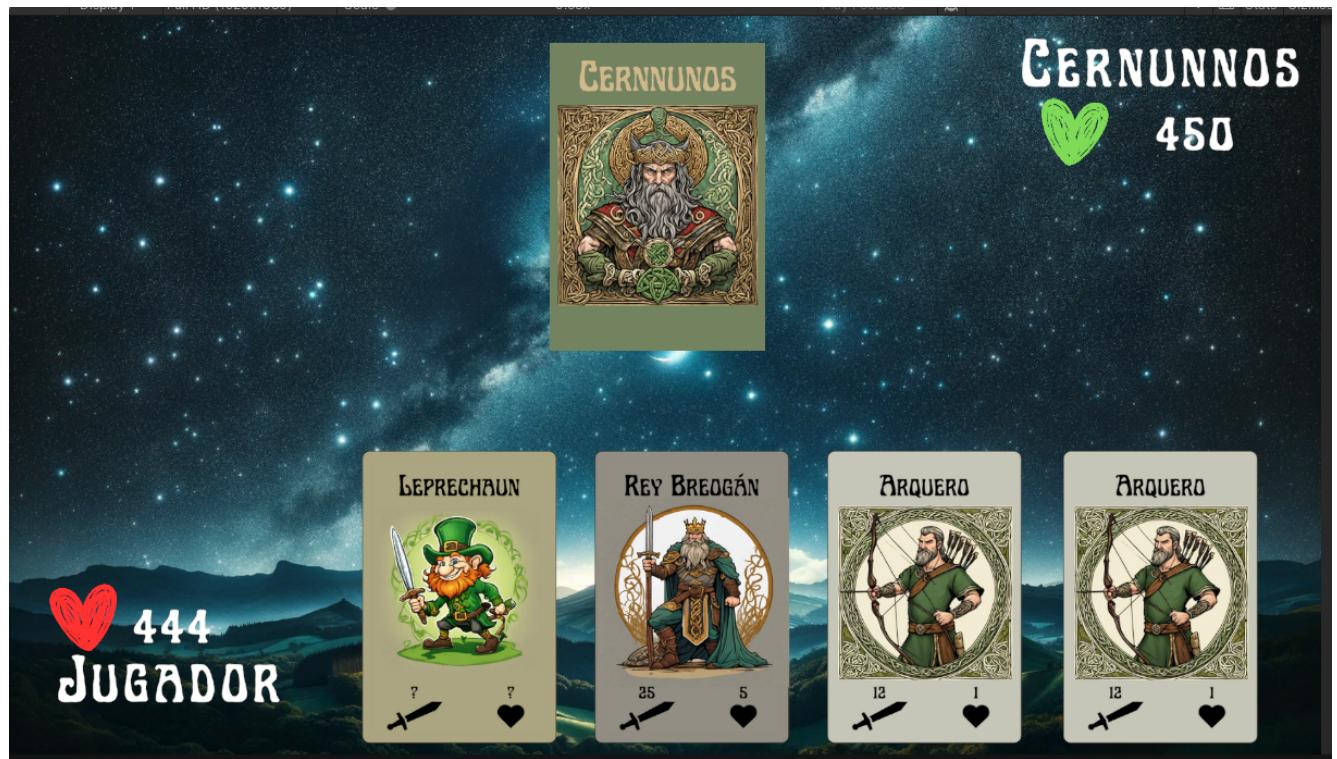
Clase 07.2 - Desarrollo Juego Cartas RPG



1- Cómo será la dinámica del juego.....	2
2- Cómo empezar el proyecto.....	3
2.1- EscenaInicial.....	6
2.2- EscenaGanar.....	14
2.3- EscenaPerder.....	15
2.4- Cambiar el tipo de Fuente.....	16
2.5- EscenaBatalla.....	19
3- Dinámica de Juego.....	21
3.1- Plantilla Cartas.....	21
3.2- Plantilla Dios.....	24
3.3- Game Manager.....	26
3.3.1- Inicio de la Partida.....	29
3.3.2- Audio de la Partida.....	37
3.3.3- Animaciones de Nivel y de Cartas.....	40
3.3.4- Programar onClick de las Imágenes.....	44
3.3.5- Programar Leprechaun.....	47
4- Código de la clase PartidaCartas.cs.....	49
5- Posibles Mejoras del Juego.....	63
Anexo 1- Manual y Clases de Unity que necesitamos.....	65
0- Manual de Unity.....	65
Coroutine.....	65
yield return null.....	67
Font.....	68
MonoBehaviour.....	69
Métodos de MonoBehaviour.....	70
ScriptableObject.....	71

1- Cómo será la dinámica del juego

El juego es un estilo RPG de cartas por turnos donde jugaremos contra los dioses celtas. Tendremos 4 cartas a usar y que se irán renovando cada vez que las usemos.



Tenemos que derrotar al dios haciendo que su vida sea menor de 0 y antes de que acabe con nuestra vida (jugador).

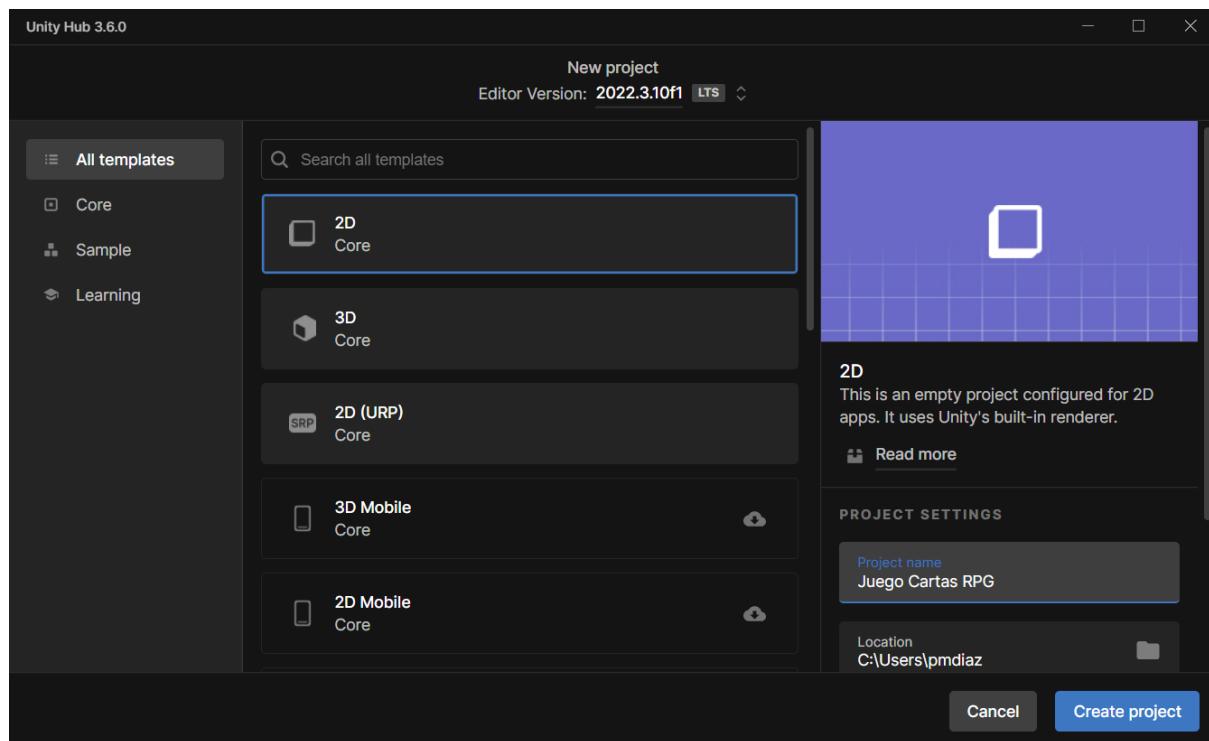
Hay 3 niveles de Dioses en los que iremos derrotando a cada uno de ellos, si llegamos al nivel 3 y ganamos => ganamos la partida.

Si nos quedamos sin vida antes de vencer al Dios => perdemos la partida.

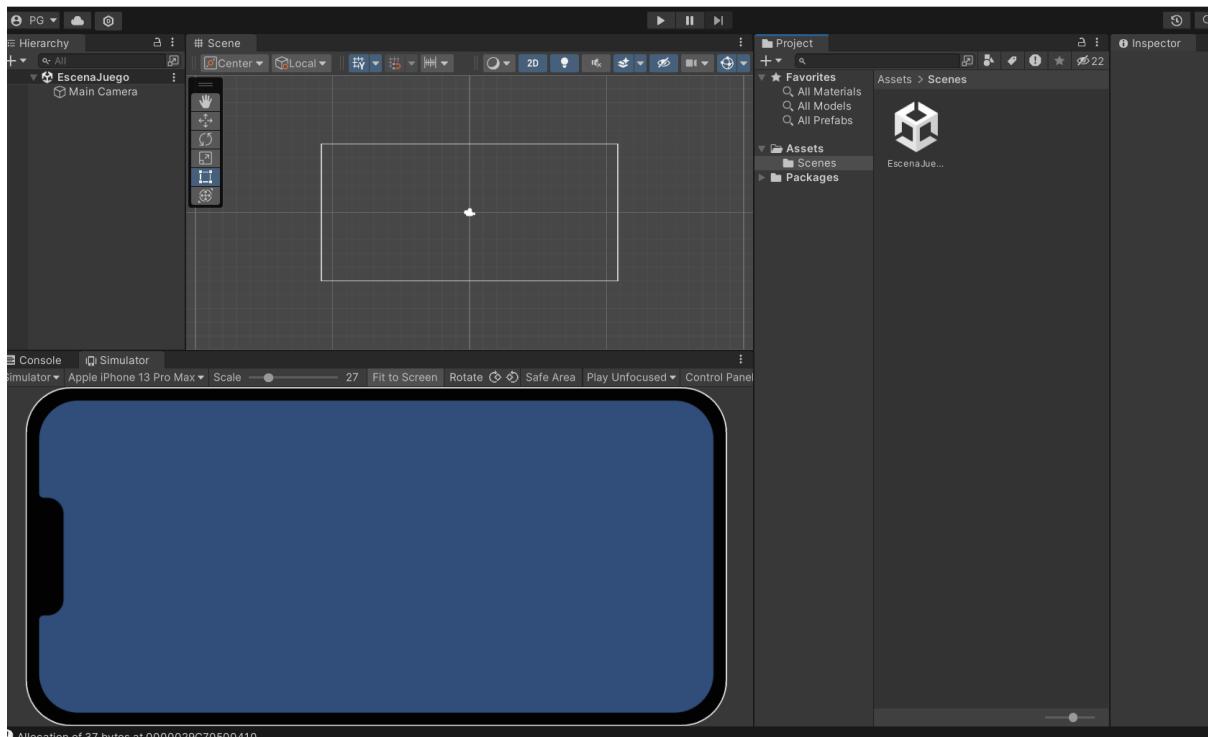
2- Cómo empezar el proyecto

En este proyecto vamos a empezar de cero, es por eso que al abrir Unity Hub escogemos:

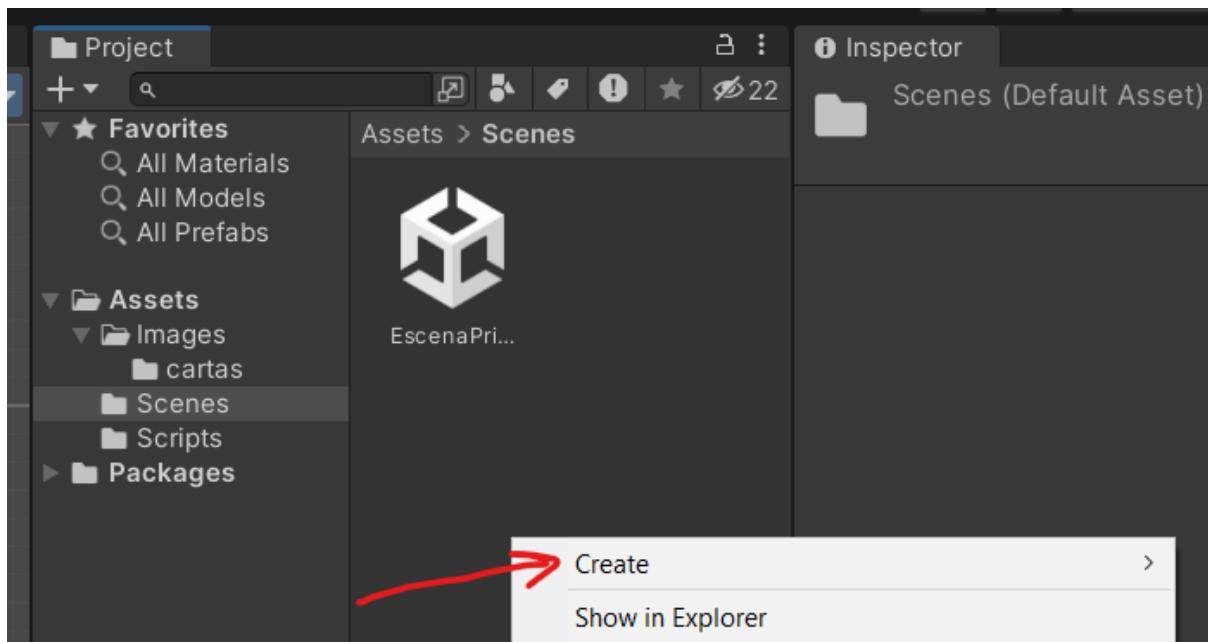
All templates > 2D Core



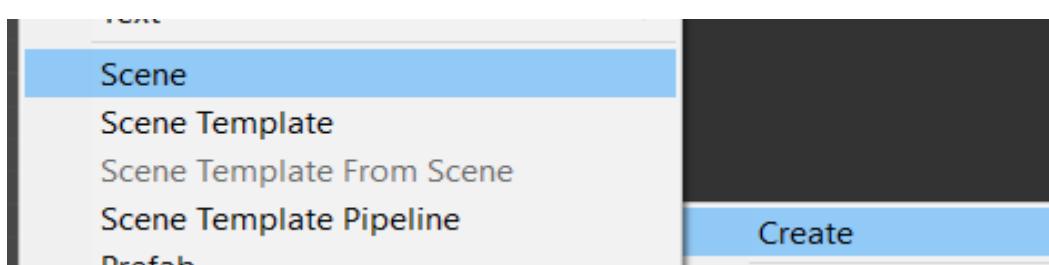
Configuramos nuestro entorno de trabajo y cambiamos el nombre a la escena.



Vamos a crear las escenas del juego, para ello desde la vista proyecto hacemos clic con el botón derecho en la opción create.



Create > Scene



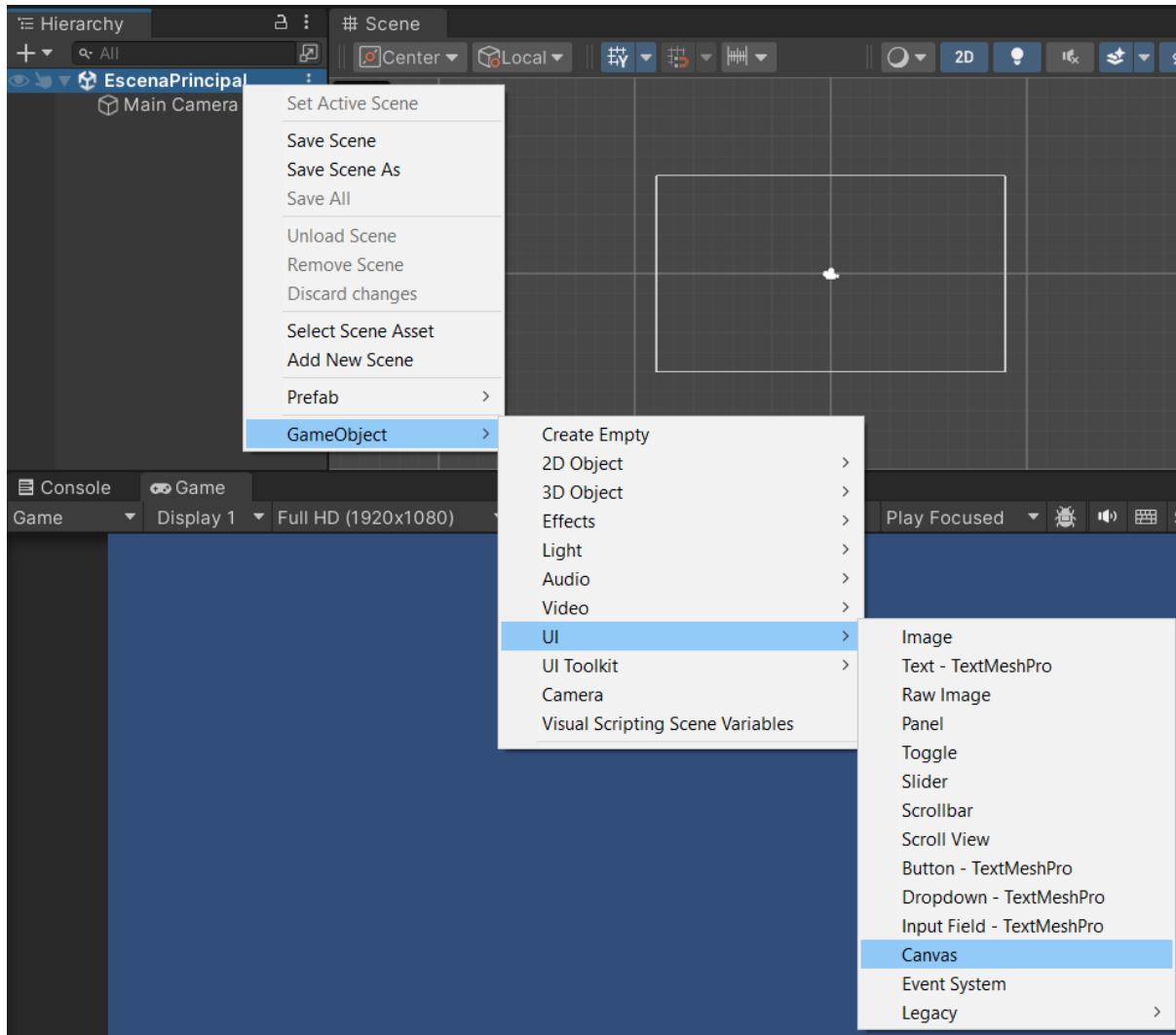
Creamos 4 escenas:

- EscenaPrincipal => donde estará el juego
- EscenaInicial => donde podremos hacer clic en "Empezar Partida"
- EscenaGanar => escena final donde indicará que ganamos
- EscenaPerder => escena final donde indicaremos que perdimos

2.1- Escena Inicial

Ya tenemos diseñada la escena inicial en el documento anterior lo único que vamos a hacerla dentro de Unity para ello vamos a seguir los siguientes pasos:

Añadir un Canvas (Lienzo) en la Escena Inicial en ventana Hierarchy:



Vamos a ver cómo podemos indicar que ese Canvas con la UI se adapte al tamaño de pantalla que tenemos (Responsive Design) y nos mantenga las proporciones:

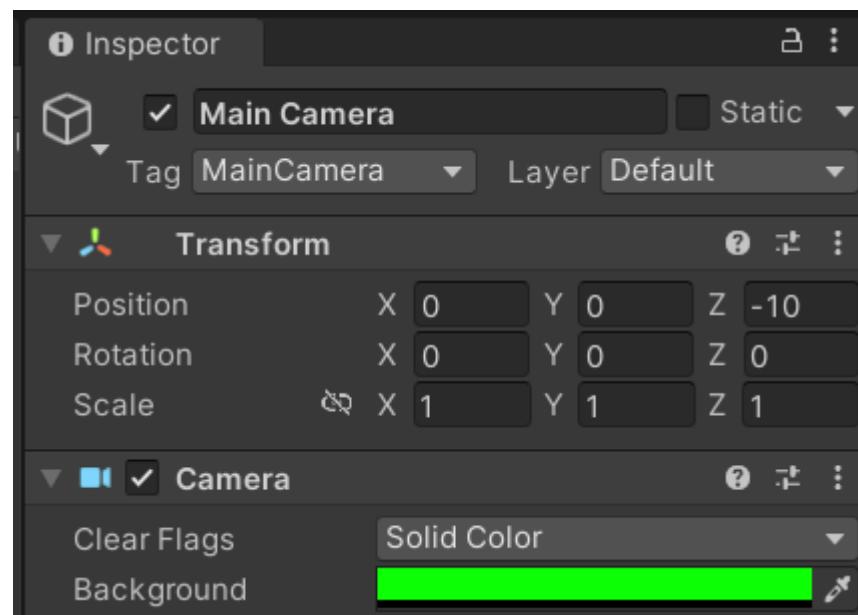
En el Inspector de Escena> Canvas vamos a la parte que pone "Canvas Scaler" e indicamos "Scale with Screen Size", x=1920, y=1080



Vamos a recordar el diseño que teníamos para la EscenaIncial:



Para el color del fondo lo vamos a cambiar en la “Main Camera” de Hierarchy. La seleccionamos y en Inspector cambiamos el “Background”



El siguiente paso para construir la UI de Inicio es añadir los diferentes elementos gráficos:

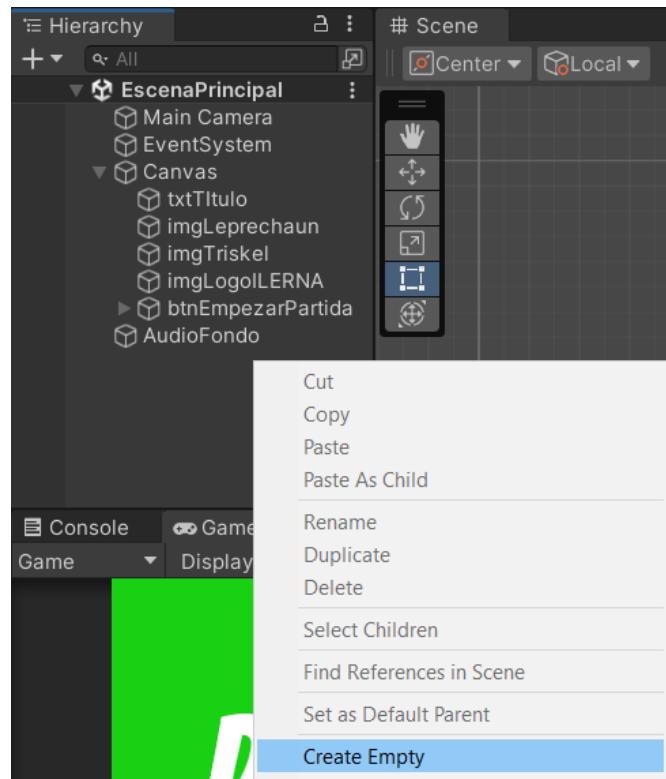
- Dibujo leprechaun
- Dibujo triskel
- Logo de Ilerna
- TextMeshPro: el título
- Botón empezar partida.

Trabajando con estos elementos nos debe quedar la EscenaIncial de esta forma:

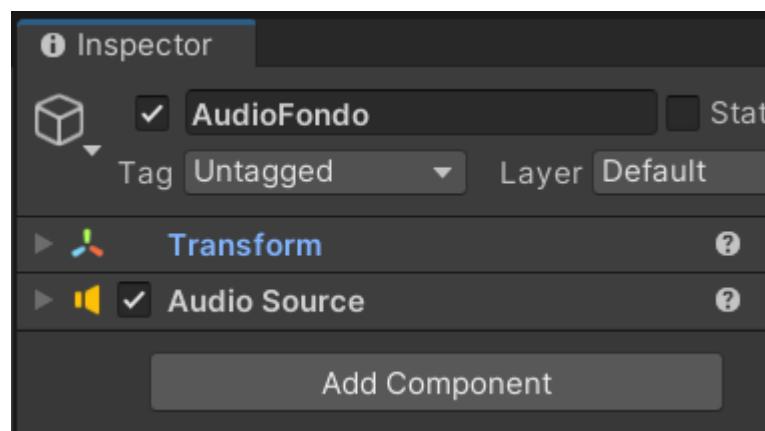


Como es la pantalla de Inicio vamos a aprovechar a ponerle una música de inicio al estilo “música celta” para que nos anime el inicio del juego.

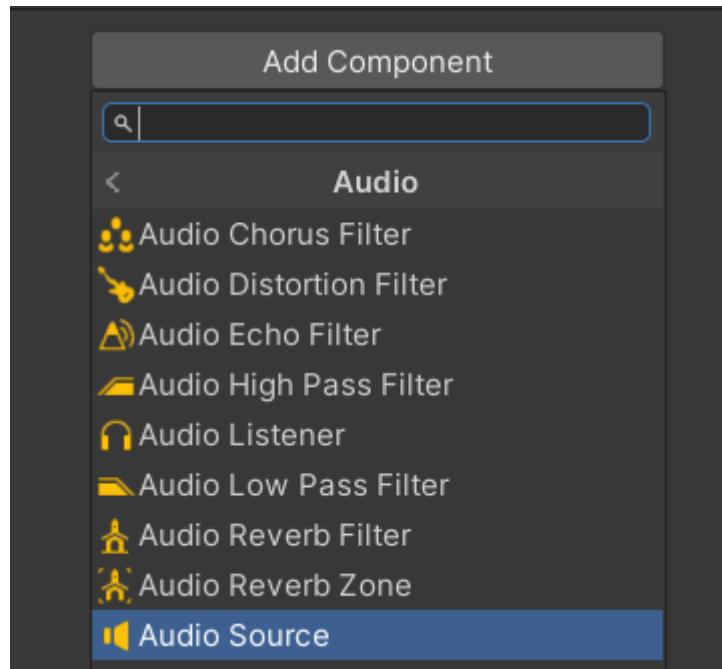
Para ello creamos un elemento vacío (Create Empty) en la EscenaIncial y lo renombramos como AudioFondo



Desde el inspector de AudioFondo le damos a Add Component

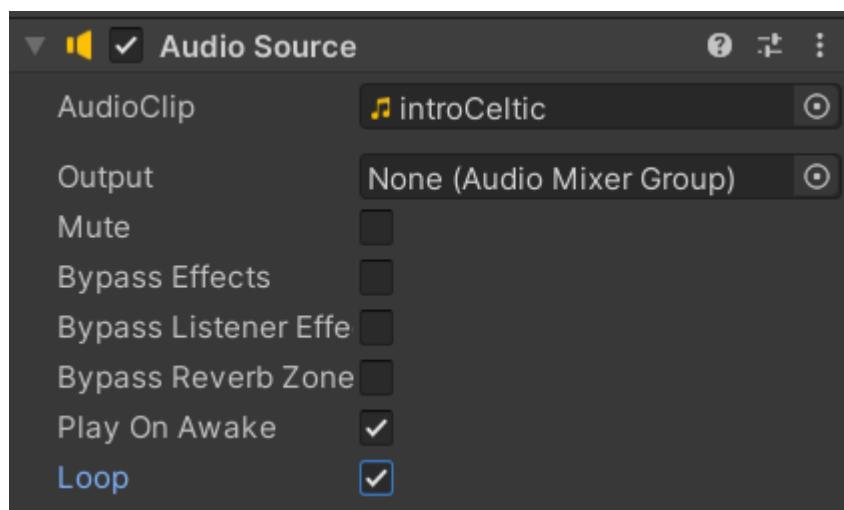


Escogemos Audio > AudioSource



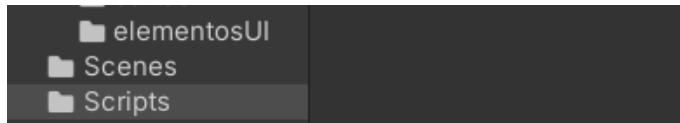
Configuramos el componente añadiendo el archivo de audio en AudioClip y marcando estas opciones

- Play On Awake (sonar desde el principio)
- Loop (que se repita en bucle)



Lo que nos falta es lanzar la EscenaPrincipal al hacer clic en el botón para ello hacemos los siguientes pasos:

En Project > Scripts > Create C# Script con el nombre “CambiarEscena.cs”

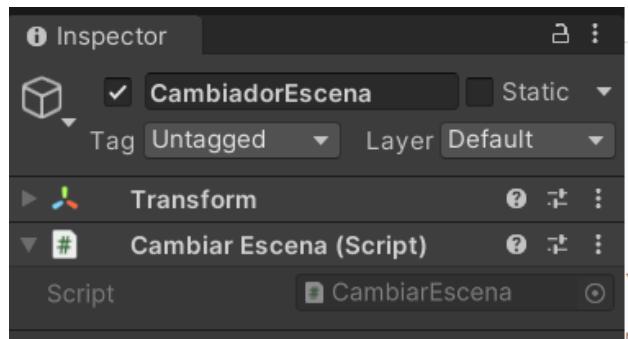


El código de ese Script va a ser el siguiente:

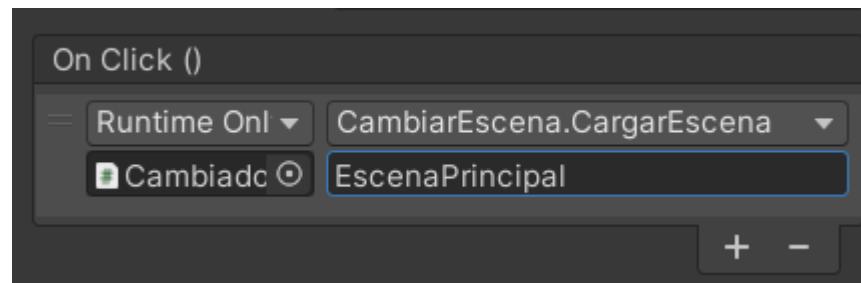
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.SceneManagement;

public class CambiarEscena : MonoBehaviour
{
    /**
     * Método para cargar una escena
     */
    public void CargarEscena(string nombreEscena)
    {
        SceneManager.LoadScene(nombreEscena);
    }
}
```

Para configurarlo en el botón tenemos que hacer un CreateEmpty en Hierarchy de la EscenaInicial (Crear un GameObject vacío) al que le asociamos el script de CambiarEscena.



Ahora vamos al botón de la “Empeza Partida”. Vamos a la parte de On Click y arrastramos el objeto creado “CambiadorEscena” al objeto y seleccionamos el método CargarEscena al cual le podemos poner el nombre de EscenaPrincipal



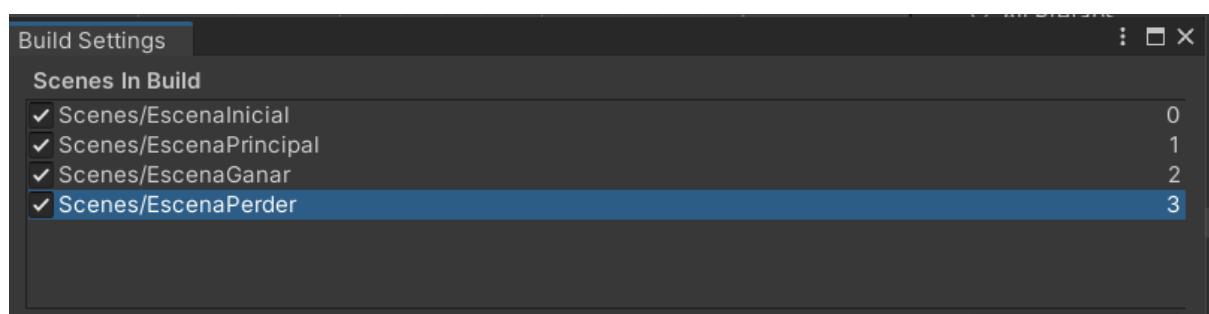
Si probamos el botón Empezar Partida nos va a salir este error:

 Scene 'EscenaPrincipal' couldn't be loaded because it has not been added to the build settings or the AssetBundle has not been loaded.

“Scene ‘EscenaPrincipal’ couldn’t be loaded because.... “

Tenemos que preparar las escenas. Asegúrate de que ambas escenas (la actual y la a la que quieras cambiar) están agregadas en el "Build Settings" de tu proyecto:

- Ve a File > Build Settings.
- Arrastra tus escenas desde el Explorador de Proyectos al cuadro de "Scenes In Build" en la ventana de Build Settings.
- Cierra la ventana de Build Settings.



2.2- EscenaGanar

Vamos a recordar cómo es la escena ganar del diseño:



Vamos a crearla con los siguiente elementos:

- Título (TextMeshPro)
- Imagen del Leprechaun contento
- Botón de volver a jugar (que no aparece)
- Audio de fondo “celtic_win.mp3”

Con lo visto en la EscenaInicial puedes hacer esta escena.



2.3- EscenaPerder

Vamos a recordar cómo es la escena perder del diseño:



Vamos a crearla con los siguiente elementos:

- Título (TextMeshPro)
- Imagen del Leprechaun contento
- Botón de volver a jugar (que no aparece)
- Audio de fondo “celtic_loose.mp3”

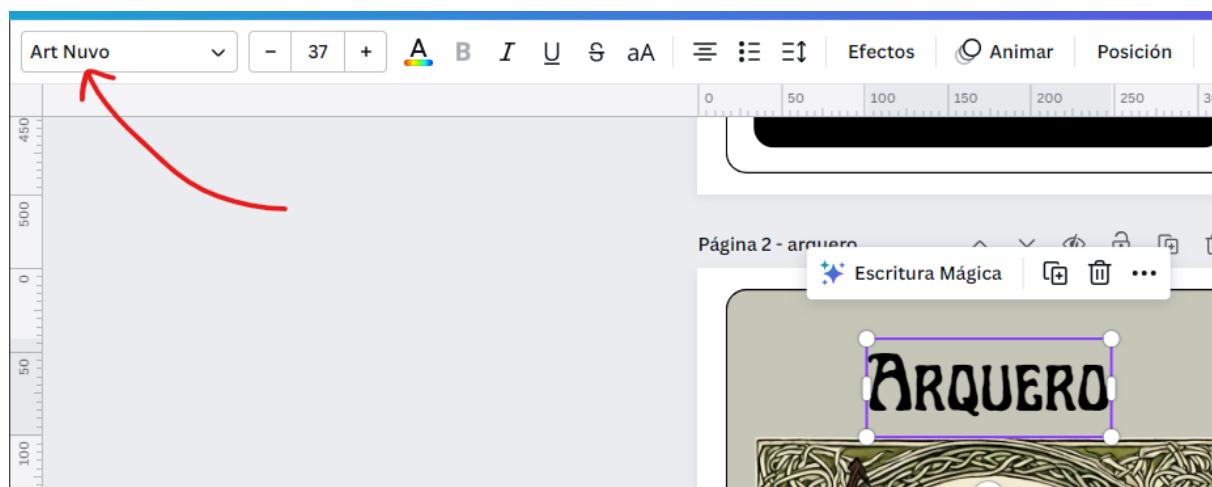
Con lo visto en la EscenaInicial puedes hacer esta escena.



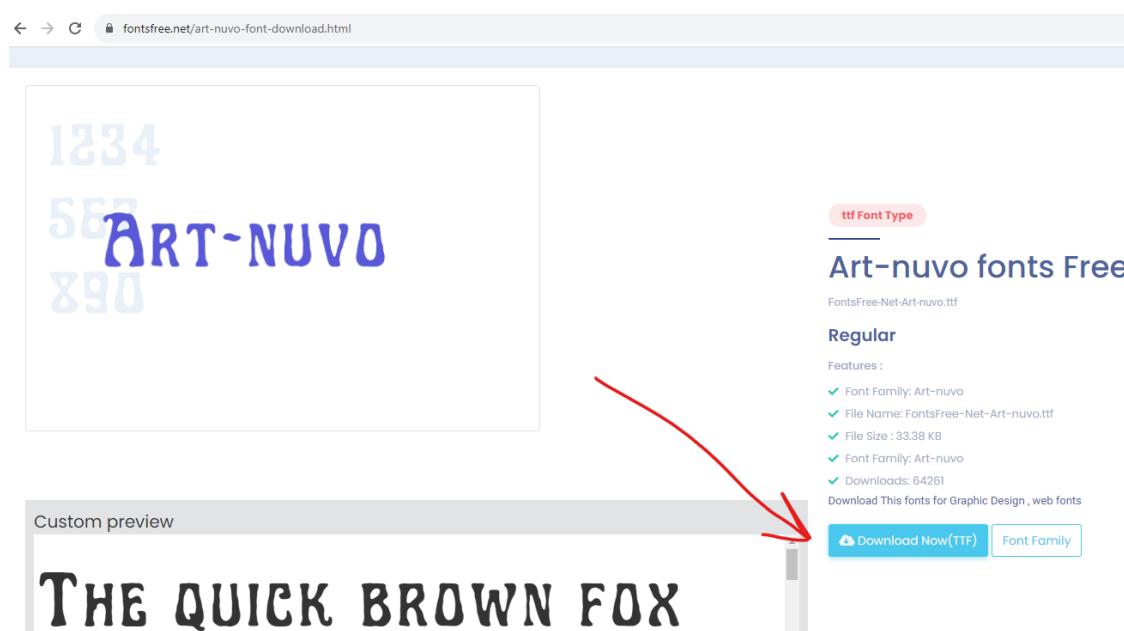
2.4- Cambiar el tipo de Fuente

Vamos a utilizar otro recurso gráfico que aún no hemos utilizado y es muy habitual en las aplicaciones multimedia. Vamos a añadir el tipo de fuente que usamos en el diseño de las cartas para darle uniformidad al juego con el mismo tipo de fuente

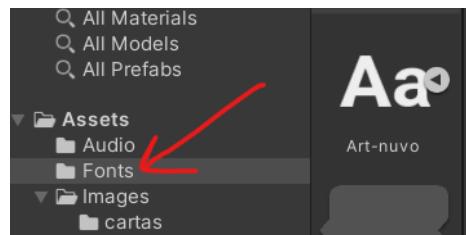
Lo primero que vamos a hacer es identificar y descargar el tipo de fuente que usamos en Canva para el diseño de las cartas.



La fuente se llama “Art Nuvo”. Buscamos en google “font art nuvo” y nos lleva a la página <https://fontsfree.net/> desde donde podemos descargar el archivo .ttf

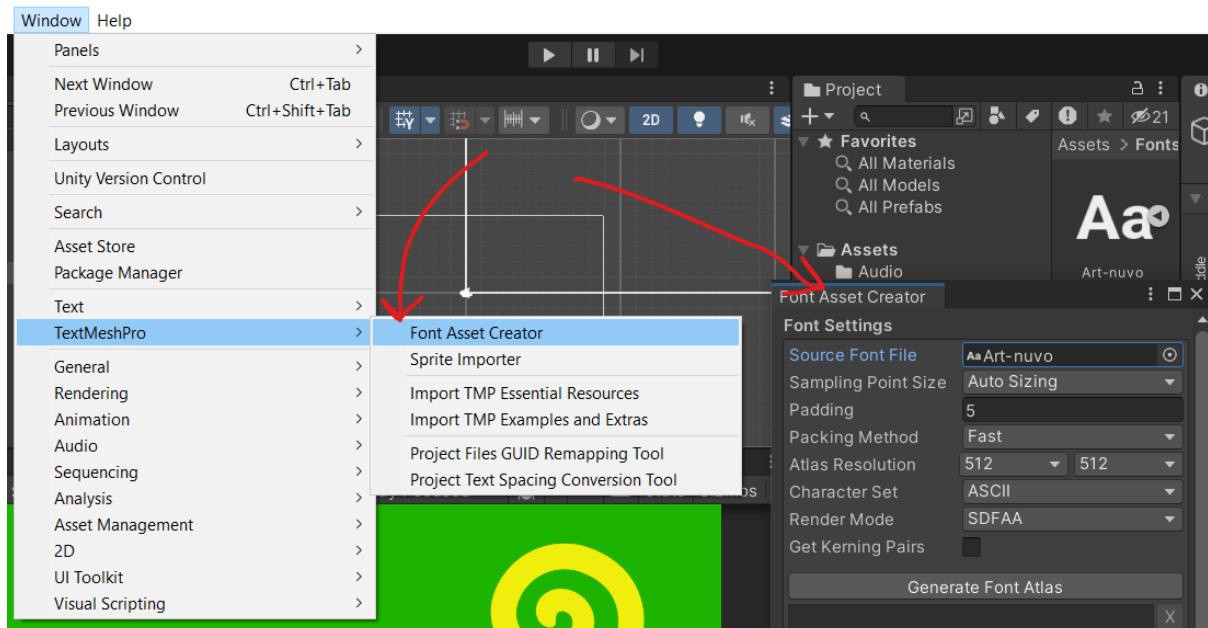
A screenshot of the FontsFree.net website. The URL in the address bar is "fontsfree.net/art-nuovo-font-download.html". The main content area shows a preview of the "ART-NUVO" font, displaying various characters and numbers. To the right, there's a sidebar with the title "Art-nuovo fonts Free" and a download button labeled "Download Now(TTF)". A red arrow points from the text "Podemos descargar el archivo .ttf" to the "Download Now(TTF)" button.

Una vez descargado el archivo .ttf lo movemos a nuestro proyecto dentro de Assets > Fonts (una carpeta que hemos de crear)

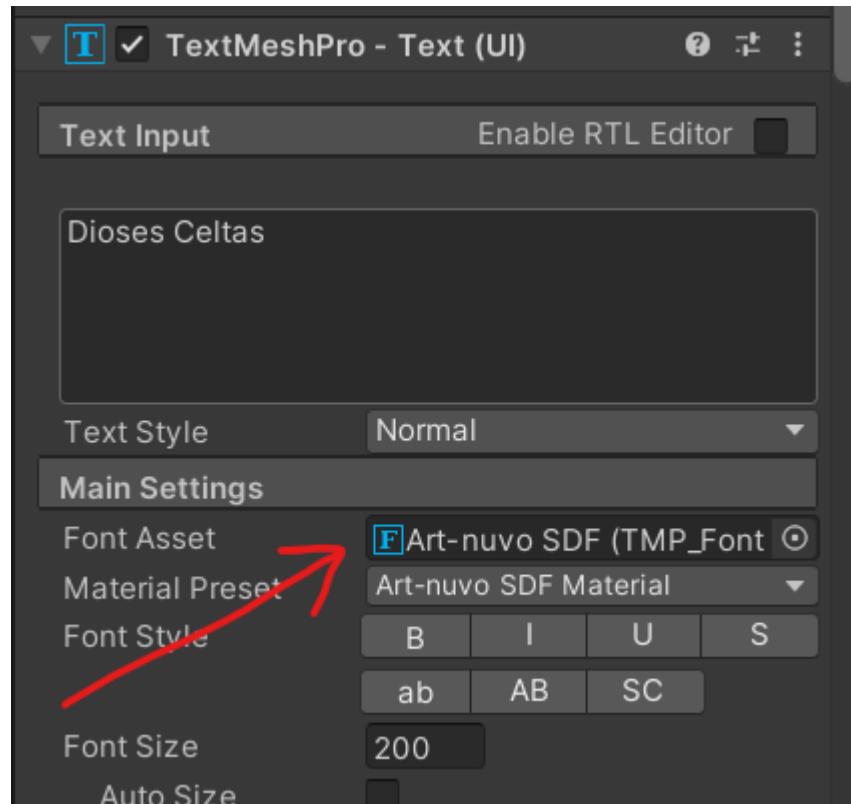


Una vez importado el archivo .ttf, necesitas convertirlo en un asset de fuente compatible con TextMeshPro:

- Abrir el Window de TextMeshPro: Ve a Window > TextMeshPro > Font Asset Creator.
- Seleccionar la Fuente: En el Font Asset Creator, selecciona tu fuente .ttf importada como 'Source Font File'.
- Ajustar Configuraciones: Configura las opciones según tus necesidades (tamaño de la fuente, caracteres incluidos, etc.).
- Crear el Asset de Fuente: Haz clic en 'Generate Font Atlas' y luego en 'Save As...' para guardar el nuevo asset de fuente de TextMeshPro en tus Assets.



Tras añadirlo ya podemos usar esa fuente en los TextMeshPro



Aquí vemos un ejemplo de cómo quedaría la EscenaIncial con esos cambios



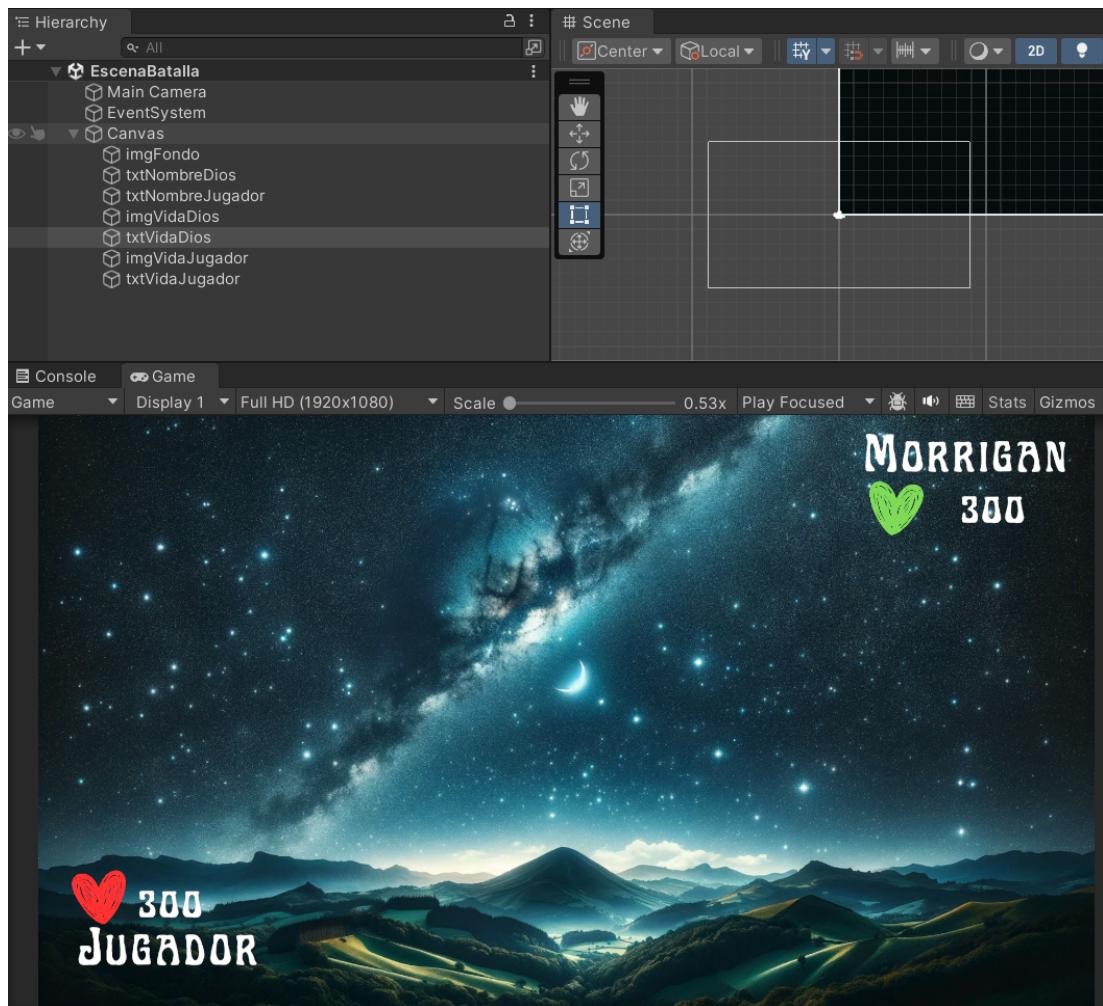
2.5- EscenaBatalla

Vamos a usar una imagen de fondo creada con IA con DALL-E de OpenAI, y esta será la imagen a utilizar:

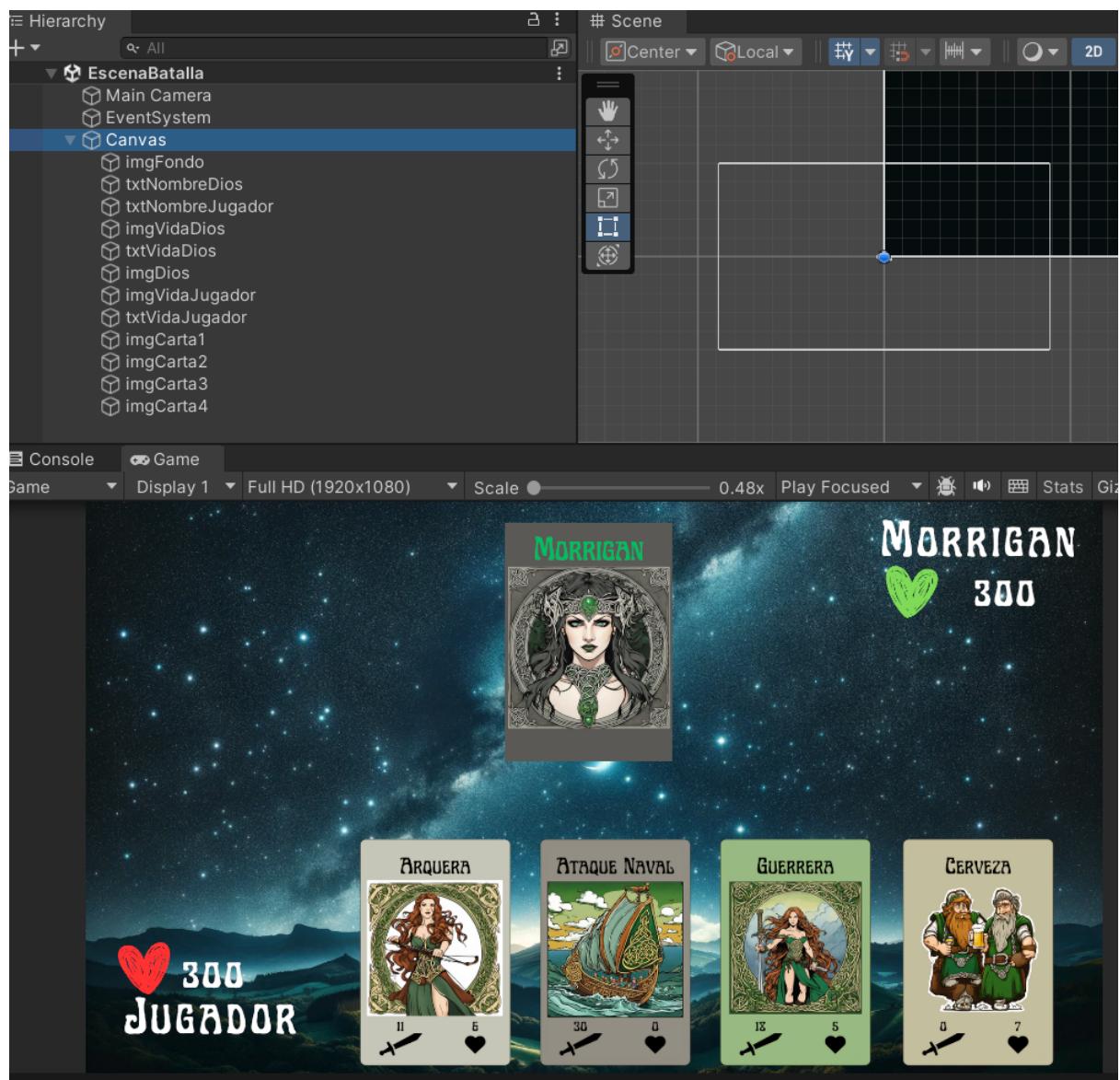


Un fondo que invita a pensar que sería un campo de batalla contra los dioses.

Lo primero que haremos será añadir los nombres (dios y jugador), img de vida y puntos de vida



Nos queda por poner la imagen del dios y las 4 cartas que usaremos en la partida



Aunque usaremos más elementos de UI como mostrar el nivel cuando cambiemos de nivel eso lo iremos viendo en cada parte del proceso.

3- Dinámica de Juego

3.1- Plantilla Cartas

Vamos a hacer la plantilla de las cartas, para ello vamos a crear la clase **PlantillaCartas**, pero vamos a hacerlo utilizando una de las clases más versátiles de Unity.

La clase **ScriptableObject** en Unity es una herramienta poderosa para los desarrolladores, utilizada principalmente para guardar y almacenar datos. A diferencia de los scripts de MonoBehaviour que están ligados a los objetos de juego y dependen del estado de la escena, los ScriptableObjects permiten crear múltiples instancias de un script con sus propios valores únicos, y no necesitan estar adjuntos a un objeto de juego.

Los principales casos de uso para [ScriptableObjects](#) incluyen:

1. Guardar y almacenar datos durante una sesión del Editor: Esto permite a los desarrolladores conservar información mientras trabajan en el Editor de Unity, sin la necesidad de ejecutar la escena o el juego.

2. Guardar datos como un activo en tu proyecto para usar en tiempo de ejecución:

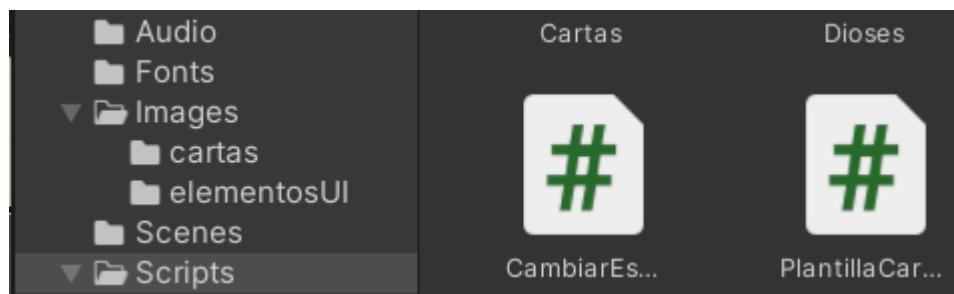
Puedes crear ScriptableObjects para almacenar datos que luego se utilizarán cuando el juego se ejecute, como configuraciones, diálogos, inventario y más.

Además de actuar como contenedores de datos versátiles que pueden almacenar los mismos tipos de datos disponibles para los MonoBehaviours, los ScriptableObjects ofrecen beneficios significativos para la arquitectura del código y la eficiencia del flujo de trabajo. Ayudan a crear una mejor arquitectura de código al implementar patrones de diseño, acelerar el flujo de trabajo en Unity y mejorar la colaboración entre programadores y artistas.

Para definir un ScriptableObject, se crea una clase C# que hereda de la clase base ScriptableObject y se le agregan campos y propiedades para los datos que se desean almacenar. Esto significa que puedes utilizar ScriptableObjects para separar los datos del

juego de la lógica, lo cual facilita el mantenimiento del proyecto y promueve la reutilización del código.

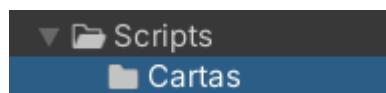
Carpeta Scripts > Create Script > "PlantillaCarta"



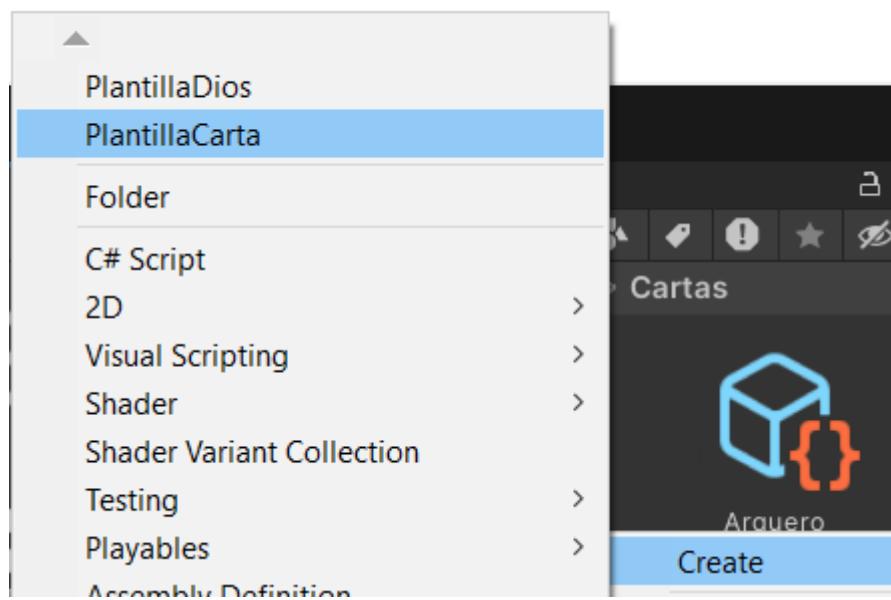
```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

//Esta línea nos permite crear el menú en el Asset
[CreateAssetMenu(fileName = "ModeloCarta", menuName =
"PlantillaCarta")]
public class PlantillaCarta : ScriptableObject
{
    public Sprite imagenCarta;
    public string nombreCarta;
    public int ataque;
    public int vida;
    public bool esAtaque;
}
```

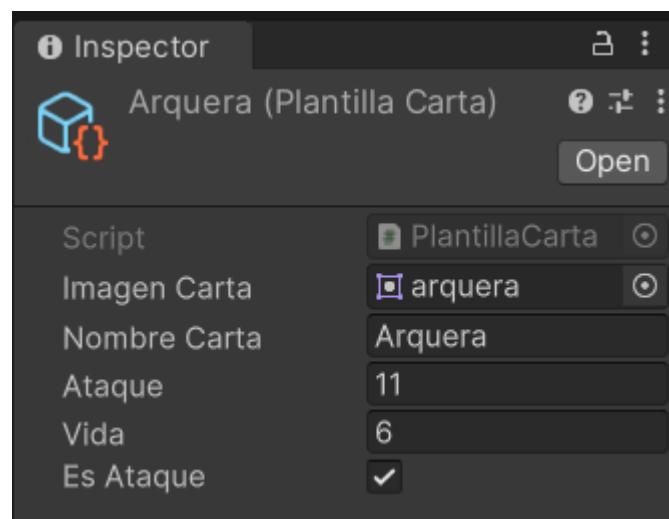
Creamos una carpeta llamada "Cartas" dentro de Scripts



En ella vamos a crear cada una de las cartas que necesitamos Create > PlantillaCarta



Ahora podemos crear los objetos de las cartas que vamos a usar:



De esta forma tenemos ya en Unity los datos de cada tipo de carta y la imagen de la misma. Ahora mismo ya podríamos usar las 12 cartas que tenemos para poder jugar.

3.2- Plantilla Dios

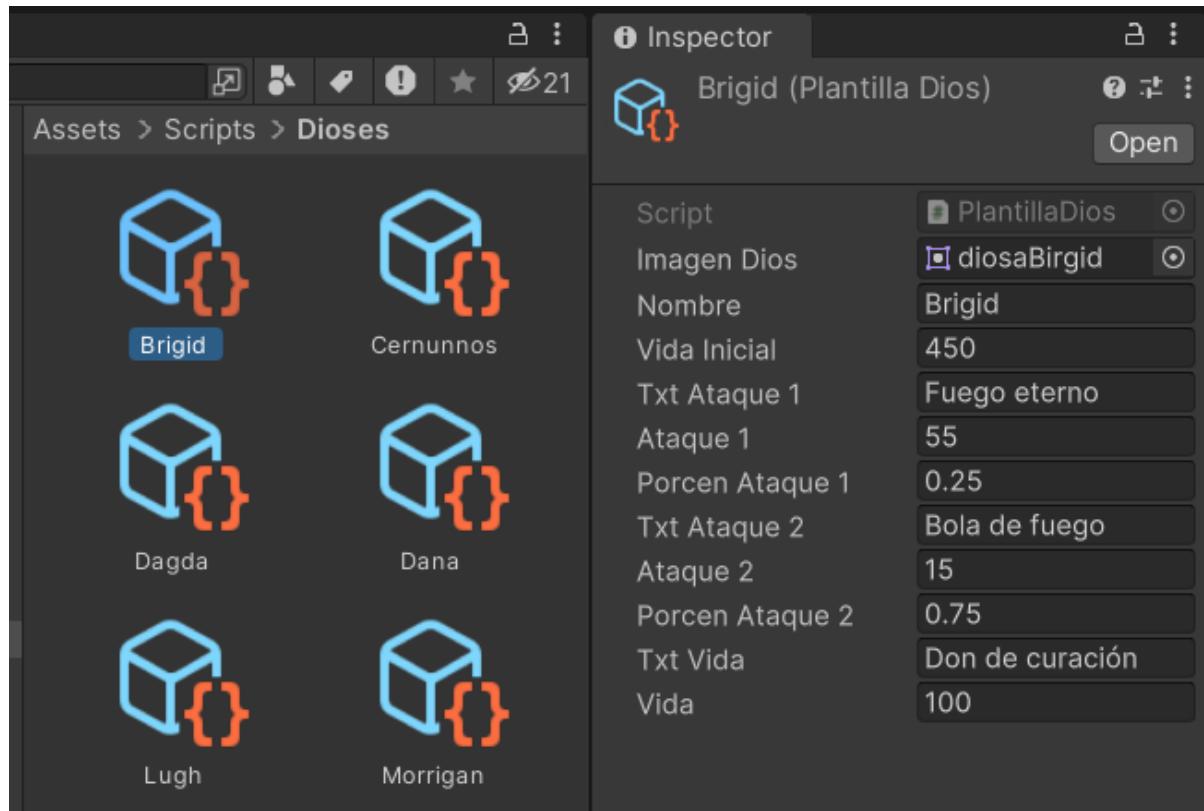
Vamos a hacer la plantilla de los dioses, para ello vamos a crear la clase **PlantillaDios** y repetimos el proceso con la clase ScriptableObject

Carpeta Scripts > Create Script > “PlantillaDios”

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

//Esta linea nos permite crear el menú en el Asset
[CreateAssetMenu(fileName = "ModeloDios", menuName = "PlantillaDios")]
public class PlantillaDios : ScriptableObject
{
    public Sprite imagenDios;
    public string nombre;
    public int vidaInicial;
    // Datos del ataque 1
    public string txtAtaque1;
    public int ataque1;
    public double porcenAtaque1;
    // Datos del ataque 2
    public string txtAtaque2;
    public int ataque2;
    public double porcenAtaque2;
    // Recuperar vida
    public string txtVida;
    public int vida;
}
```

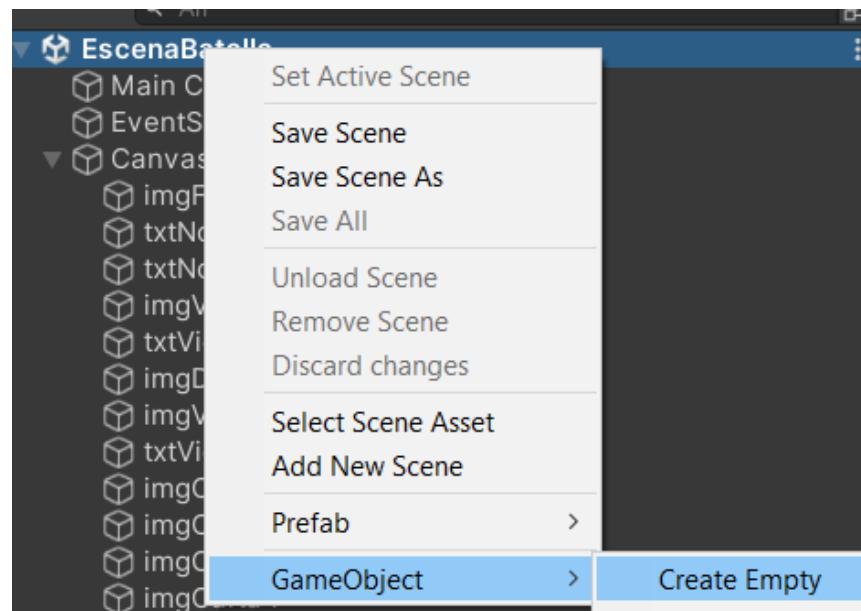
Ahora podemos crear los objetos de los dioses que vamos a usar:



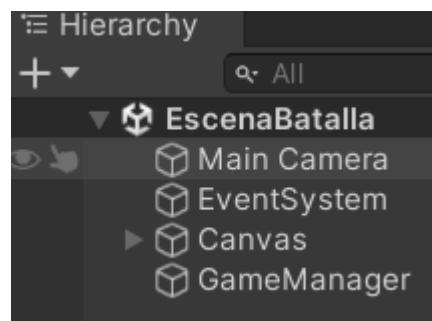
De esta forma tenemos ya en Unity los datos de cada tipo de Dios y la imagen de cada uno. Ahora mismo ya podríamos usar los 6 dioses que tenemos para poder jugar.

3.3- Game Manager

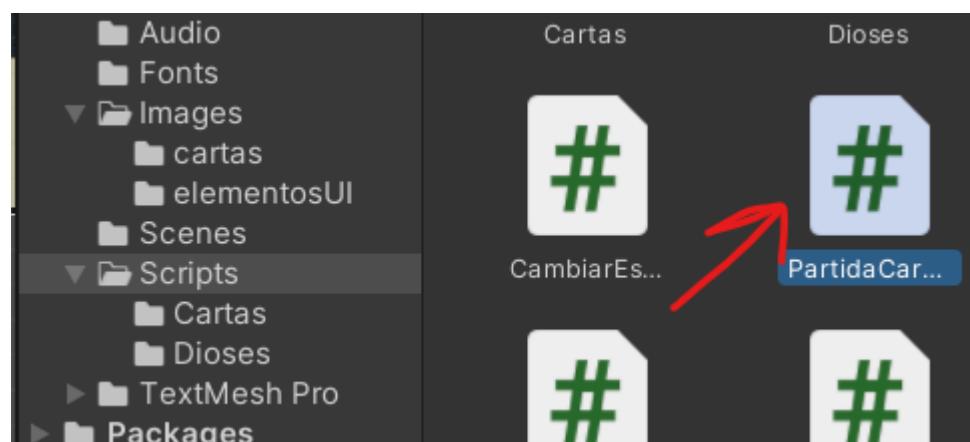
Ya nos estamos acercando a la creación de la dinámica del juego y para ello vamos a crear un GameObject Vacío (Empty) donde podremos añadir el script de juego para realizar todo el trabajo de gestionar la dinámica del juego.



Lo llamaremos **GameManager**



Crearemos un Script llamado **PartidaCartas** para gestionar toda la partida



Vamos ver la primera parte de la clase PartidaCartas, para ello nos vamos a fijar que estamos identificando como atributos de la clase los elementos que vamos a utilizar en la dinámica de juego

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

public class PartidaCartas : MonoBehaviour
{
    // Mostrar atributos privados en el Inspector
    [SerializeField]
    private Image imagenDios;
    // Si los hacemos públicos se muestran por defecto en el Inspector
    public TextMeshProUGUI txtNombreDios;
    public TextMeshProUGUI txtNombreJugador;
    public TextMeshProUGUI txtVidaDios;
    public TextMeshProUGUI txtVidaJugador;
    // Imágenes de las vidas
    public Image vidaDios;
    public Image vidaJugador;
    // Recogemos las 4 cartas a jugar
    public Image carta1;
    public Image carta2;
    public Image carta3;
    public Image carta4;

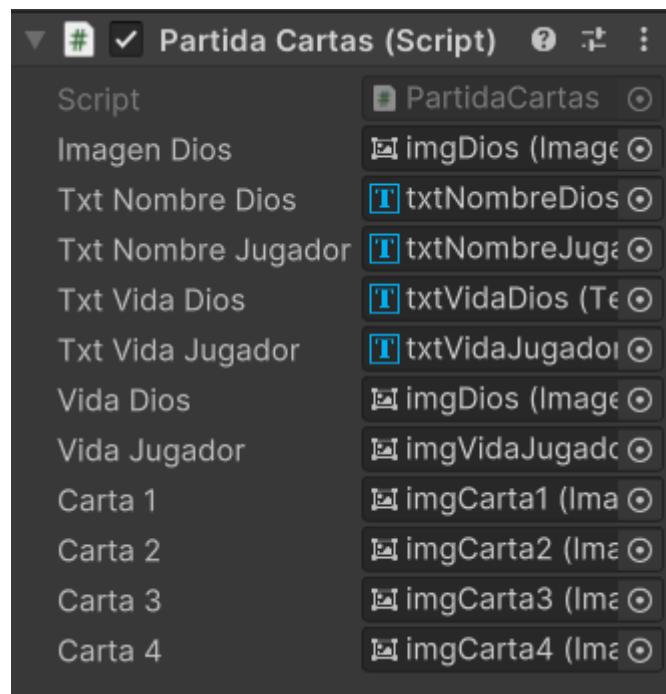
    ....
```

Si usamos `SerializeField` para los atributos privados o si usamos `public` en este Script podremos verlos en el inspector.

Primero vamos a añadir el componente al GameObject con este Script. Y nos fijamos que aparecen los componentes que especificamos.



Arrastramos los componentes al panel del inspector para poder asociarlos a nuestro GameManager.



3.3.1- Inicio de la Partida

Vamos a crear el inicio de la partida. Lo primero que vamos a hacer es poner como no visibles los elementos tanto del Jugador como del Dios. Sólo veremos la imagen de fondo.

Para ello en el método Start vamos a usar este código

```
// Start is called before the first frame update
void Start()
{
    //Primero ocultamos los elementos del Dios y del Jugador
    ocultarElementosDios();
    ocultarElementosJugador();
}
```

Y el código de estos métodos será el siguiente

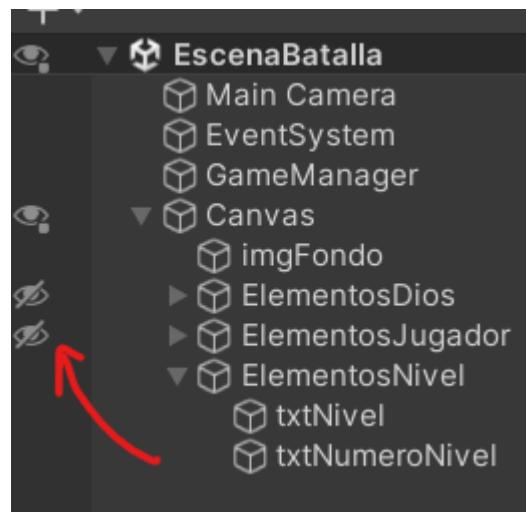
```
// Ocultamos los elementos del Dios
private void ocultarElementosDios()
{
    imagenDios.enabled = false;
    txtNombreDios.enabled = false;
    txtVidaDios.enabled = false;
    vidaDios.enabled = false;
}

// Ocultamos los elementos del Jugador
private void ocultarElementosJugador()
{
    txtNombreJugador.enabled = false;
    txtVidaJugador.enabled = false;
    vidaJugador.enabled = false;
    carta1.enabled = false;
    carta2.enabled = false;
    carta3.enabled = false;
    carta4.enabled = false;
}
```

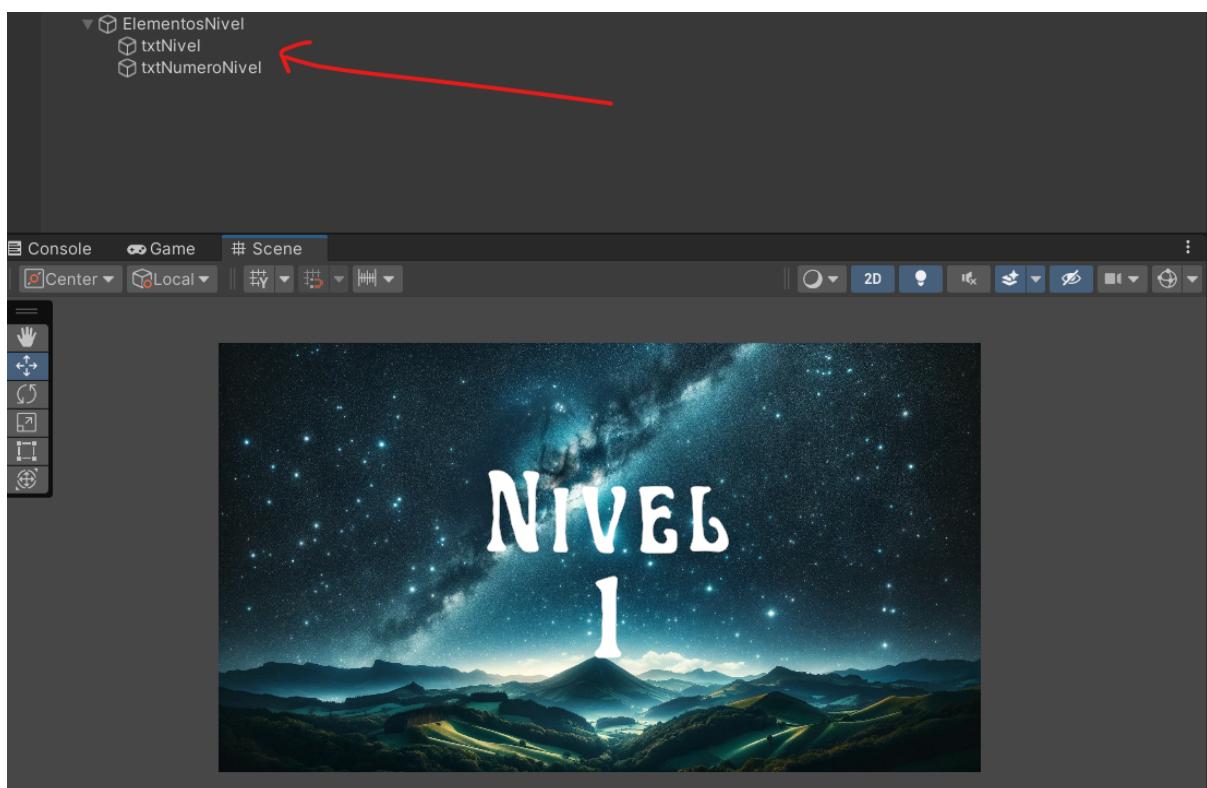
Si nos fijamos establecemos “enabled=false” a cada componente.

La siguiente parte va a ser tener la información de mostrar el Nivel en el que estamos, para ello vamos a hacer lo siguiente:

Vamos a ordenar la Escena por elementos del Dios y elementos del Jugador para poder ocultarlos cuando estemos trabajando en la vista Scene usamos “Empty objects” que nos servirán para ordenar la Jerarquía como si fueran carpetas



Creamos un ElementoNivel y le añadimos 2 textos: txtNivel y txtNúmeroNivel



Añadimos a nuestro Script el txtNivel y el txtNumNivel para poder modificarlo:

```
public class PartidaCartas : MonoBehaviour
{
    // Mostrar atributos privados en el Inspector
    [SerializeField]
    private Image imagenDios;
    // Si los hacemos públicos se muestran por defecto en el Inspector
    public TextMeshProUGUI txtNombreDios;
    public TextMeshProUGUI txtNombreJugador;
    public TextMeshProUGUI txtVidaDios;
    public TextMeshProUGUI txtVidaJugador;
    public Image vidaDios; // Imágenes de las vidas
    public Image vidaJugador;
    public Image carta1; // Recogemos las 4 cartas a jugar
    public Image carta2;
    public Image carta3;
    public Image carta4;
    public int vidaInicialJugador=300; // Vida Inicial del jugador
    public TextMeshProUGUI txtNivel;
    public TextMeshProUGUI txtNumNivel;
    private int nivelJuego=1;
}
```

Los asociamos desde el Inspector:



Ahora vamos a crear la **animación para mostrar el nivel**, para ello lo hacemos en el Script:

PartidaCartas.cs

```
private void MostrarNivel()
{
    txtNivel.enabled = true;
    txtNumNivel.enabled = true;
}

// Ocultamos los elementos del Nivel
private void OcultarNivel()
{
    txtNivel.enabled = false;
    txtNumNivel.enabled = false;
}

// Método para la Animación de Cambiar Nivel
private IEnumerator AnimarCambioNivel()
{
    float halfDuration = tiempoAnimacionNivel / 2f;
    float currentDuration = 0f;
    MostrarNivel();

    while (currentDuration < tiempoAnimacionNivel)
    {
        currentDuration += Time.deltaTime;

        float alpha;
        if (currentDuration <= halfDuration)
        {
            alpha = currentDuration / halfDuration;
        }
        else
        {
            alpha = 1 - (currentDuration - halfDuration) / halfDuration;
        }

        SetAlphaNivel(alpha);
        yield return null;
    }

    SetAlphaNivel(0); // Asegurar que el texto sea invisible al final
    OcultarNivel();
    StartCoroutine(AparecerCartas()); // Hacemos la animación de aparecer las cartas
}
```

```
// Establecemos el Alpha del Nivel
private void SetAlphaNivel(float alpha)
{
    SetAlphaTextMesh(txtNivel, alpha);
    SetAlphaTextMesh(txtNumNivel, alpha);
}

// Establecemos el Alpha de un TextMeshPro
private void SetAlphaTextMesh(TextMeshProUGUI text, float alpha)
{
    Color color = text.color;
    color.a = alpha;
    text.color = color;
}

// Start is called before the first frame update
void Start()
{
    //Primero ocultamos los elementos del Dios y del Jugador
    OcultarElementosDios();
    OcultarElementosJugador();
    StartCoroutine(AnimarCambioNivel()); // Hacemos la animación de Cambio de Nivel
}
```

Para hacer las animaciones vamos a usar Corrutinas ([Coroutine](#)), si pensamos en la asignatura DAM_M09 Programación de Procesos y Servicios viene a ser un pequeño Hilo que nos permite lanzar ese “mini-proceso” para realizar la animación.

También vamos a realizar la **animación para mostrar las cartas del nivel** y así poder empezar la partida en el nivel actual en el que nos encontramos.

Para ello vamos a crear otros métodos que nos van a ayudar a realizar esta nueva animación.

```
// Mostramos los elementos del Dios
private void MostrarElementosDios()
{
    imagenDios.enabled = true;
    txtNombreDios.enabled = true;
    txtVidaDios.enabled = true;
    vidaDios.enabled = true;
}

// Ocultamos los elementos del Dios
private void OcultarElementosDios()
{
    imagenDios.enabled = false;
    txtNombreDios.enabled = false;
    txtVidaDios.enabled = false;
    vidaDios.enabled = false;
}

// Mostrar los elementos del Jugador
private void MostrarElementosJugador()
{
    txtNombreJugador.enabled = true;
    txtVidaJugador.enabled = true;
    vidaJugador.enabled = true;
    carta1.enabled = true;
    carta2.enabled = true;
    carta3.enabled = true;
    carta4.enabled = true;
}

// Ocultamos los elementos del Jugador
private void OcultarElementosJugador()
{
    txtNombreJugador.enabled = false;
    txtVidaJugador.enabled = false;
    vidaJugador.enabled = false;
    carta1.enabled = false;
    carta2.enabled = false;
    carta3.enabled = false;
    carta4.enabled = false;
}
```

```
// Animación para que aparezcan las cartas
private IEnumerator AparecerCartas()
{
    float duration = 5f; // Duración en segundos de la Animación
    float halfDuration = duration / 2f;
    float currentDuration = 0f;

    MostrarElementosDios();
    MostrarElementosJugador();
    SetAlphaCartas(0); // El alpha de las cartas a 0

    while (currentDuration < duration)
    {
        currentDuration += Time.deltaTime;
        float alpha;
        alpha = currentDuration / halfDuration;
        SetAlphaCartas(alpha);
        yield return null;
    }
}

// Animación para que desaparezcan las cartas
private IEnumerator DesaparecerCartas()
{
    float duration = 5f; // Duración en segundos de la Animación
    float halfDuration = duration / 2f;
    float currentDuration = 0f;

    MostrarElementosDios();
    MostrarElementosJugador();
    SetAlphaCartas(0); // El alpha de las cartas a 0

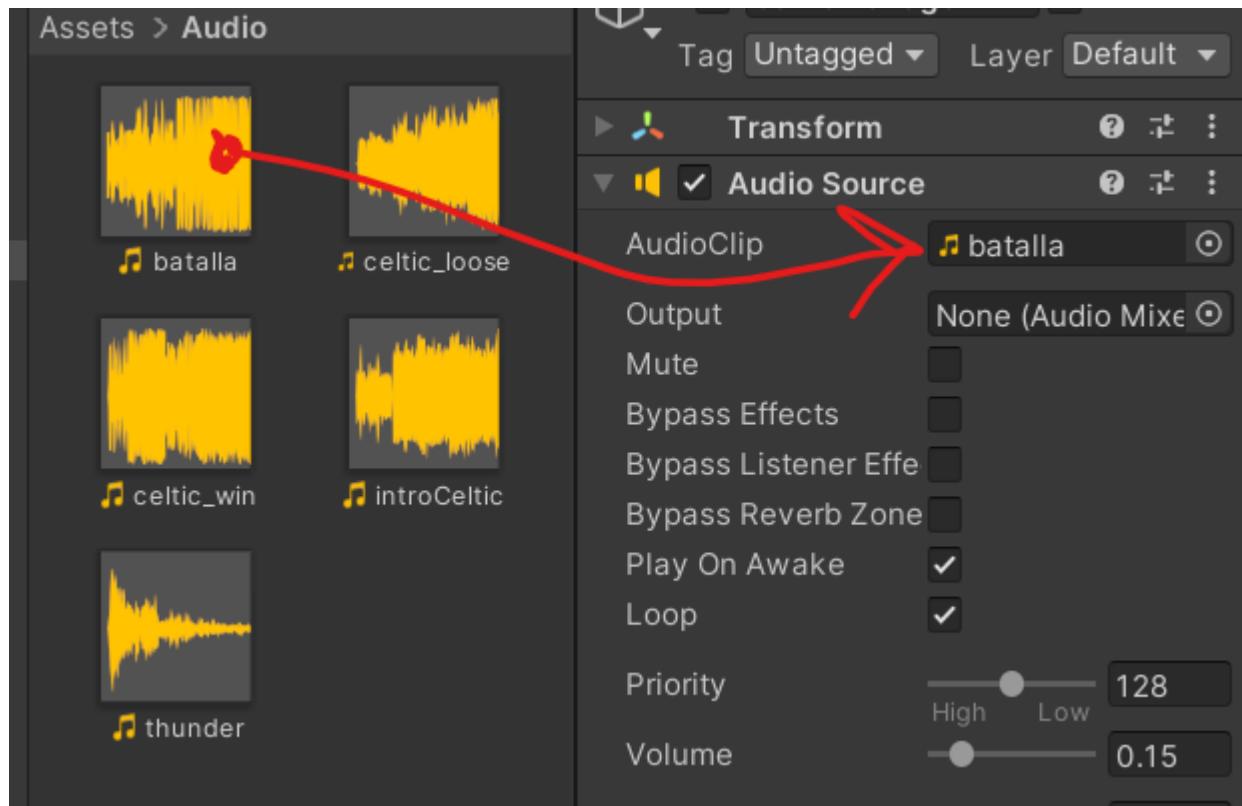
    while (currentDuration < duration)
    {
        currentDuration += Time.deltaTime;
        float alpha;
        alpha = 1 - (currentDuration - halfDuration) / halfDuration;
        SetAlphaCartas(alpha);
        yield return null;
    }
}
```

```
// Establece el Alpha de las cartas y de la imágenes de la partida
private void SetAlphaCartas(float alpha)
{
    SetAlphaImagen(imagenDios, alpha);
    SetAlphaImagen(vidaDios, alpha);
    SetAlphaImagen(vidaJugador, alpha);
    SetAlphaImagen(cartal, alpha);
    SetAlphaImagen(carta2, alpha);
    SetAlphaImagen(carta3, alpha);
    SetAlphaImagen(carta4, alpha);
    SetAlphaTextMesh(txtNombreDios, alpha);
    SetAlphaTextMesh(txtNombreJugador, alpha);
    SetAlphaTextMesh(txtVidaDios, alpha);
    SetAlphaTextMesh(txtVidaJugador, alpha);
}

// Establece el Alpha de una imagen
private void SetAlphaImagen(Image imagen, float alpha)
{
    imagen.color = new Color(imagen.color.r, imagen.color.g,
    imagen.color.b, alpha);
}
```

3.3.2- Audio de la Partida

Vamos a utilizar Audio en la partida, el primero será el audio de fondo de la Escena, para ello añadimos un componente AudioSource a Game Manager con el archivo que tenemos en Assets llamado batalla.mp3



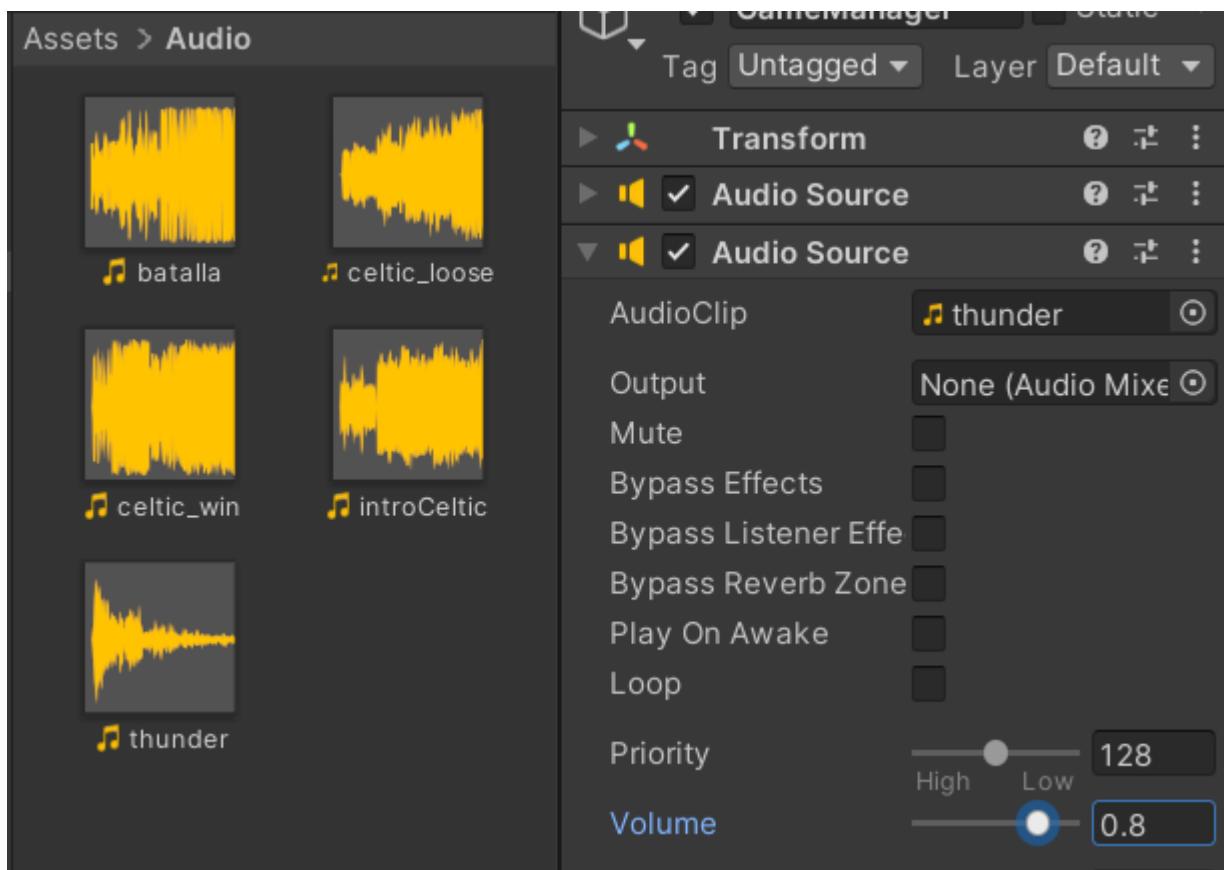
Recordar ponerlo con el PlayOnAwake (empezar a sonar solo) y el Loop (bucle).

Vamos a usar más archivos de audio en la batalla, es por eso que vamos a referenciarlos con este array

```
// Archivos de audio cargados
private AudioSource[] audios = GetComponents< AudioSource>();
```

Luego podremos llamarlos según la posición en la que se encuentren en el GameManager. Ej: audios[0] será el Audio de fondo que cargamos anteriormente.

Vamos a cargar el archivo de audio "thunder" que es el audio de un rayo para cuando cambiamos de nivel en la animación.



Tendremos que añadir en el código de la animación lo siguiente:

```
// Archivos de audio cargados
private AudioSource[] audios;

// Start is called before the first frame update
void Start()
{
    //Cogemos los archivos de audio
    audios = GetComponents<
```

En la animación de cambio de nivel vamos a añadir la reproducción de la archivo de audio del rayo:

```
// Método para la Animación de Cambiar Nivel
private IEnumerator AnimarCambioNivel()
{
    float halfDuration = tiempoAnimacionNivel / 2f;
    float currentDuration = 0f;
    MostrarNivel();
    audios[1].Play(); // Reproducir el audio del rayo

    while (currentDuration < tiempoAnimacionNivel)
    {
        currentDuration += Time.deltaTime;

        float alpha;
        if (currentDuration <= halfDuration)
        {
            alpha = currentDuration / halfDuration;
        }
        else
        {
            alpha = 1 - (currentDuration - halfDuration) / halfDuration;
        }

        SetAlphaNivel(alpha);
        yield return null;
    }

    SetAlphaNivel(0); // Asegurar que el texto sea invisible al final
    OcultarNivel();
    // Hacemos la animación de aparecer las cartas
    StartCoroutine(AparecerCartas());
}
```

3.3.3- Animaciones de Nivel y de Cartas

Uno de los efectos más interesantes que podemos tener en los juegos es dar dinamismo a las escenas y tener animaciones que generen ese movimiento dentro del mismo.

En este caso vamos a hacer la animación de cambio de Nivel:

- Aparece el texto “Nivel N” poco a poco y desaparece poco a poco
- Suena archivo de audio del Rayo

Para ello tenemos ya el siguiente código:

```
// Start is called before the first frame update
void Start()
{
    //Cogemos los archivos de audio
    audios = GetComponents< AudioSource >();
    // Empezamos el nivel del Juego
    EmpezarNivelJuego();
}

// Establece el comienzo de un nivel de juego
private void EmpezarNivelJuego()
{
    SetDiosPorNivel();
    EmpezarNivelCartas();
}

// En base al nivel de juego tenemos un/a Dios/a u otro/a
private void SetDiosPorNivel()
{
    int numAleatorio = aleatorio.Next(2);
    switch(nivelJuego)
    {
        case 1:
            if(numAleatorio==0)
                diosActual=dioses[0]; // Brigid
            else
                diosActual=dioses[1]; // Cernunnos
            break;
    }
}
```

```
        case 2:
            if(numAleatorio==0)
                diosActual=dioses[2]; // Lugh
            else
                diosActual=dioses[3]; // Morrigan
            break;
        case 3:
            if(numAleatorio==0)
                diosActual=dioses[4]; // Dagda
            else
                diosActual=dioses[5]; // Dana
            break;
    }
    // Una vez elegido el dios actualizamos la información en la UI
    imagenDios.sprite = diosActual.imagenDios;
    txtNombreDios.text = diosActual.nombre;
    txtVidaDios.text = diosActual.vidaInicial.ToString();
}

// Establecemos un nivel de vida al jugador + cartas para jugar
private void EmpezarNivelCartas()
{
    //Primero ocultamos los elementos del Dios y del Jugador
    OcultarElementosDios();
    OcultarElementosJugador();
    // Ocultamos los ataques del dios y del Leprechaun
    OcultarTextAtaqueDios();
    OcultarTextLeprechaun();
    // Hacemos la animación de Cambio de Nivel
    StartCoroutine(AnimarCambioNivel());
    // Insertamos 4 cartas en la baraja de la partida
    for(int i=0; i<numCartas; i++)
    {
        cartasActuales[i]=GetCartaAleatoria();
    }
    // Sumamos entre 0 y 200 x Nivel a la vida del jugador
    int vidaASumar = aleatorio.Next(200);
    int vidaJugador = int.Parse(txtVidaJugador.text);
    txtVidaJugador.text = (vidaJugador + vidaASumar).ToString();
    ActualizarCartasUI();
}
```

```
// Método para la Animación de Cambiar Nivel
private IEnumerator AnimarCambioNivel()
{
    float halfDuration = tiempoAnimacionNivel / 2f;
    float currentDuration = 0f;
    MostrarNivel();
    audios[1].Play();
    while (currentDuration < tiempoAnimacionNivel)
    {
        currentDuration += Time.deltaTime;

        float alpha;
        if (currentDuration <= halfDuration)
        {
            alpha = currentDuration / halfDuration;
        }
        else
        {
            alpha = 1 - (currentDuration - halfDuration) / halfDuration;
        }

        SetAlphaNivel(alpha);
        yield return null;
    }

    SetAlphaNivel(0); // Asegurar que el texto sea invisible al final
    OcultarNivel();
    // Hacemos la animación de aparecer las cartas
    StartCoroutine(AparecerCartas());
}

// Animación para que aparezcan las cartas
private IEnumerator AparecerCartas()
{
    float duration = 5f; // Duración en segundos de la Anim
    float halfDuration = duration / 2f;
    float currentDuration = 0f;
    MostrarElementosDios();
    MostrarElementosJugador();
    SetAlphaCartas(0); // El alpha de las cartas a 0
    while (currentDuration < duration)
    {
        currentDuration += Time.deltaTime;
```

```
        float alpha;
        alpha = currentDuration / halfDuration;
        SetAlphaCartas(alpha);
        yield return null;
    }
}
```

Para crear las animaciones de los objetos vamos a usar [Coroutines](#) como elementos para que podamos realizar esa animación. En Unity, una coroutine es una función que puede pausar su ejecución y reanudarla en el próximo frame o después de un cierto tiempo, permitiendo realizar tareas a lo largo del tiempo sin bloquear el resto del juego.

Para ello tenemos que hacer un par de pasos:

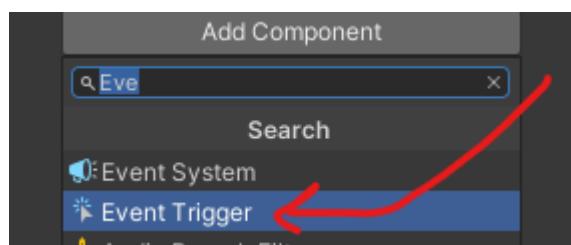
- Crear método “private/public **IEnumerator** Animacion”
 - Tiene que devolver un objeto IEnumerator
- Usar la línea [`yield return null;`](#), cuando en una coroutine encuentras la línea `yield return null;`; significa lo siguiente:
 - **Pausar la Coroutine:** En el punto donde aparece `yield return null;`; la ejecución de la coroutine se pausa.
 - **Esperar hasta el Próximo Frame:** La coroutine continuará su ejecución en el siguiente frame del juego. Esencialmente, esto indica a Unity que espere hasta que se haya completado el frame actual, incluyendo el renderizado y otros procesos del motor, antes de continuar con el siguiente paso de la coroutine en el próximo frame.
 - **Reanudar la Ejecución:** En el próximo frame, la coroutine reanuda su ejecución justo después de la línea `yield return null;`.

Si pensamos en la asignatura de DAM_M09 Servicios y Procesos una Coroutine puede ser considerado como un pequeño Hilo que nos permite actualizar elementos gráficos del juego teniendo el foco del programa.

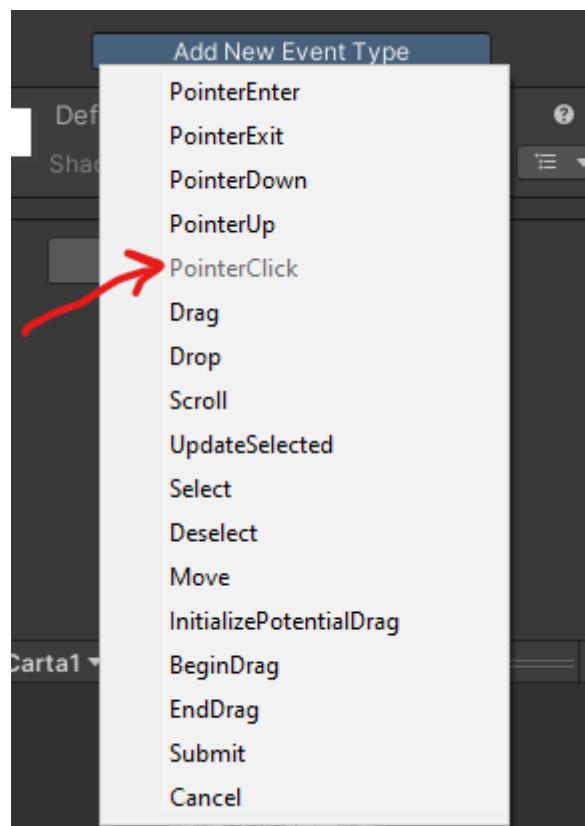
3.3.4- Programar onClick de las Imágenes

La dinámica del juego de cartas nos lleva a realizar un tipo de programación basada en eventos, en este caso nos lleva a programar las imágenes de las cartas de forma similar a que sean botones y podamos usar el evento OnClick con la función que estimemos oportuna utilizar. Vamos a seguir los siguientes pasos:

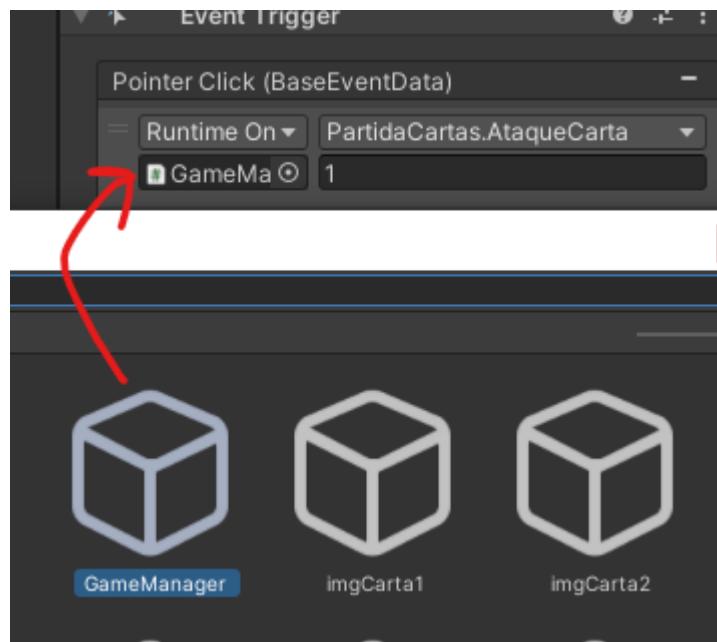
1. Elegimos una de las cartas del juego y añadimos el componente EventTrigger



2. Hacemos clic en "Add New Event Type" > "PointerClick"



3. Añadimos al “Ponter Click” el objeto de GameManager y usamos el método del archivo PartidaCartas.cs llamado **AtaqueCarta(int numero)**



Vamos a revisar el método AtaqueCarta en el fichero PartidaCartas.cs:

```
// El jugador ataca / recupera vida con la carta que tiene escogida
public void AtaqueCarta(int numero)
{
    int ataque=GetAtaqueCarta(numero);
    int vida=GetVidaCarta(numero);
    // Actualizamos la UI para Dios y Jugador
    txtVidaDios.text=(int.Parse(txtVidaDios.text)-ataque).ToString();
    txtVidaJugador.text=(int.Parse(txtVidaJugador.text)+vida).ToString();
    // Si la carta es el Leprechaun => usamos sus poderes
    if(GetNombreCarta(numero)=="Leprechaun")
    {
        int ataqueLeprechaun=GetAtaqueLeprechaun();
        int vidaLeprechaun=GetVidaLeprechaun();
        txtVidaDios.text=(int.Parse(txtVidaDios.text)-
                           ataqueLeprechaun).ToString();
        txtVidaJugador.text=(int.Parse(txtVidaJugador.text)-
                           vidaLeprechaun).ToString();
        StartCoroutine(MostrarAtaqueLeprechaun(ataqueLeprechaun,
                                                vidaLeprechaun));
    }
    // Una vez usada la carta la cambiamos
    ActualizarCarta(numero);
    ActualizarCartasUI();
}
```

```
// Después de un Ataque de una Carta viene el Ataque del Dios
AtaqueDios();
// Si muere el dios => cambiamos de nivel
if(GetVidaDios()<0)
{
    Debug.Log("Cambio de Nivel");
    nivelJuego++;
    EmpezarNivelJuego();
}
// Si muere el jugador => EscenaPerder
if(GetVidaJugador()<0)
    SceneManager.LoadScene("EscenaPerder");
// Si el Nivel de Juego es > 3 => Ganamos la partida
if(nivelJuego>3)
    SceneManager.LoadScene("EscenaGanar");
}
```

Este método lleva el peso de la dinámica del juego ya que cada ataque de cada carta va a hacer que se muevan los datos de vida de Dios y llama al Ataque del Dios después del ataque del jugador.

3.3.5- Programar Leprechaun

La carta más juguetona del juego a favor del jugador es el Leprechaun y vamos a usarla para atacar con fuerza y recuperar un buen nivel de vida.

Para programar su funcionalidad vamos a ver cómo la usamos en el método AtaqueCarta(int numero) de PartidaCartas:

```
// Si la carta es el Leprechaun => usamos sus poderes
if(GetNombreCarta(numero)=="Leprechaun")
{
    int ataqueLeprechaun=GetAtaqueLeprechaun();
    int vidaLeprechaun=GetVidaLeprechaun();
    txtVidaDios.text=(int.Parse(txtVidaDios.text)-
        ataqueLeprechaun).ToString();
    txtVidaJugador.text=(int.Parse(txtVidaJugador.text)+ 
        vidaLeprechaun).ToString();
    StartCoroutine(MostrarAtaqueLeprechaun(ataqueLeprechaun,
        vidaLeprechaun));
}
```

Vamos a ver los métodos que afectan al Leprechaun:

```
// Ataque del Leprechaun entre 100 y 200
private int GetAtaqueLeprechaun()
{
    int ataqueBase=100;
    int ataqueAleatorio=aleatorio.Next(100);
    return ataqueBase+ataqueAleatorio;
}

// Vida a sumar del Leprechaun entre 100 y 200
private int GetVidaLeprechaun()
{
    int vidaBase=100;
    int vidaAleatorio=aleatorio.Next(100);
    return vidaBase+vidaAleatorio;
}
```

```
private IEnumerator MostrarAtaqueLeprechaun(int ataque, int vida)
{
    float duration = 1f; // Duración en segundos de la Anim
    float halfDuration = duration / 2f;
    float currentDuration = 0f;
    MostrarTextLeprechaun();
    txtLeprechaun.text="Leprechaun"+"\nAtaque(\"+ataque.ToString() +\")\nVida(\""
                        +vida.ToString()+"\")";
    SetAlphaTextMesh(txtLeprechaun, 0);
    while (currentDuration < duration)
    {
        currentDuration += Time.deltaTime;

        float alpha;
        if (currentDuration <= halfDuration)
        {
            alpha = currentDuration / halfDuration;
        }
        else
        {
            alpha = 1 - (currentDuration - halfDuration) / halfDuration;
        }
        SetAlphaTextMesh(txtLeprechaun, alpha);
        yield return null;
    }
    OcultarTextLeprechaun();
}
```

4- Código de la clase PartidaCartas.cs

Este es el código principal del juego que usamos en GameManager y podéis ver toda la programación este apartado:

```
using TMPro;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;
using UnityEngine.SceneManagement;

public class PartidaCartas : MonoBehaviour
{
    // Mostrar atributos privados en el Inspector
    [SerializeField]
    private List<PlantillaDios> dioses;
    [SerializeField]
    private List<PlantillaCarta> cartas;
    // Variables del Dios del juego Actual y de las cartas del jugador
    private PlantillaDios diosActual;
    private PlantillaCarta[] cartasActuales = new PlantillaCarta[4]; // 4
    cartas necesitamos
    // Si los hacemos públicos se muestran por defecto en el Inspector
    public TextMeshProUGUI txtNombreDios;
    public TextMeshProUGUI txtNombreJugador;
    public TextMeshProUGUI txtVidaDios;
    public TextMeshProUGUI txtVidaJugador;
    public Image imagenDios;
    public Image vidaDios; // Imágenes de las vidas
    public Image vidaJugador;
    private int numCartas=4;
    private PlantillaCarta carta1Actual; // PlantillasCarta actuales de la
    baraja
    private PlantillaCarta carta2Actual;
    private PlantillaCarta carta3Actual;
    private PlantillaCarta carta4Actual;
    public Image carta1; // Recogemos las 4 cartas a jugar
    public Image carta2;
    public Image carta3;
    public Image carta4;
    public int vidaInicialJugador=300; // Vida Inicial del jugador
```

```
public TextMeshProUGUI txtNivel; // Información del Nivel
public TextMeshProUGUI txtNumNivel;
public TextMeshProUGUI txtAtaqueDios; // Texto del ataque o vida del
dios
public TextMeshProUGUI txtLeprechaun; // Texto para el ataque y defensa
del Leprechaun
private float tiempoAnimacionNivel=10f; // Tiempo en segundos
private int nivelJuego=1;
// Archivos de audio cargados
private AudioSource[] audios;
// Para generar números aleatorios
private System.Random aleatorio = new System.Random();

// Start is called before the first frame update
void Start()
{
    //Cogemos los archivos de audio
    audios = GetComponents<();
    // Empezamos el nivel del Juego
    EmpezarNivelJuego();
}

/*
 * De aquí para abajo vamos a crear los métodos que necesitamos para
nuestro juego
 */

// Mostramos los elementos del Dios
private void MostrarElementosDios()
{
    imagenDios.enabled = true;
    txtNombreDios.enabled = true;
    txtVidaDios.enabled = true;
    vidaDios.enabled = true;
}

// Ocultamos los elementos del Dios
private void OcultarElementosDios()
{
    imagenDios.enabled = false;
    txtNombreDios.enabled = false;
    txtVidaDios.enabled = false;
```

```
vidaDios.enabled = false;
}

// Mostrar los elementos del Jugador
private void MostrarElementosJugador()
{
    txtNombreJugador.enabled = true;
    txtVidaJugador.enabled = true;
    vidaJugador.enabled = true;
    carta1.enabled = true;
    carta2.enabled = true;
    carta3.enabled = true;
    carta4.enabled = true;
}

// Ocultamos los elementos del Jugador
private void OcultarElementosJugador()
{
    txtNombreJugador.enabled = false;
    txtVidaJugador.enabled = false;
    vidaJugador.enabled = false;
    carta1.enabled = false;
    carta2.enabled = false;
    carta3.enabled = false;
    carta4.enabled = false;
}

// Mostramos los elementos del Nivel
private void MostrarNivel()
{
    txtNivel.enabled = true;
    txtNumNivel.text = nivelJuego.ToString();
    txtNumNivel.enabled = true;
}

// Ocultamos los elementos del Nivel
private void OcultarNivel()
{
    txtNivel.enabled = false;
    txtNumNivel.enabled = false;
}
```

```
private void MostrarTextAtaqueDios()
{
    txtAtaqueDios.enabled = true;
}

private void OcultarTextAtaqueDios()
{
    txtAtaqueDios.enabled = false;
}

private void MostrarTextLeprechaun()
{
    txtLeprechaun.enabled = true;
}

private void OcultarTextLeprechaun()
{
    txtLeprechaun.enabled = false;
}

// Método para animar el texto del Ataque del Dios
private IEnumerator AnimarAtaqueDios()
{
    float duration = 1f; // Duración en segundos de la Anim
    float halfDuration = duration / 2f;
    float currentDuration = 0f;
    SetAlphaTextMesh(txtAtaqueDios, 0);
    while (currentDuration < duration)
    {
        currentDuration += Time.deltaTime;

        float alpha;
        if (currentDuration <= halfDuration)
        {
            alpha = currentDuration / halfDuration;
        }
        else
        {
            alpha = 1 - (currentDuration - halfDuration) / halfDuration;
        }
    }
}
```

```
        SetAlphaTextMesh(txtAtaqueDios, alpha);
        yield return null;
    }

}

// Método para la Animación de Cambiar Nivel
private IEnumerator AnimarCambioNivel()
{
    float halfDuration = tiempoAnimacionNivel / 2f;
    float currentDuration = 0f;
    MostrarNivel();
    audios[1].Play();
    while (currentDuration < tiempoAnimacionNivel)
    {
        currentDuration += Time.deltaTime;

        float alpha;
        if (currentDuration <= halfDuration)
        {
            alpha = currentDuration / halfDuration;
        }
        else
        {
            alpha = 1 - (currentDuration - halfDuration) / halfDuration;
        }

        SetAlphaNivel(alpha);
        yield return null;
    }
    SetAlphaNivel(0); // Asegurar que el texto sea invisible al final
    OcultarNivel();
    // Hacemos la animación de aparecer las cartas
    StartCoroutine(AparecerCartas());
}

// Establecemos el Alpha del Nivel
private void SetAlphaNivel(float alpha)
{
    SetAlphaTextMesh(txtNivel, alpha);
    SetAlphaTextMesh(txtNumNivel, alpha);
}
```

```
// Establecemos el Alpha de un TextMeshPro
private void SetAlphaTextMesh(TextMeshProUGUI text, float alpha)
{
    Color color = text.color;
    color.a = alpha;
    text.color = color;
}

// Animación para que aparezcan las cartas
private IEnumerator AparecerCartas()
{
    float duration = 5f; // Duración en segundos de la Anim
    float halfDuration = duration / 2f;
    float currentDuration = 0f;
    MostrarElementosDios();
    MostrarElementosJugador();
    SetAlphaCartas(0); // El alpha de las cartas a 0
    while (currentDuration < duration)
    {
        currentDuration += Time.deltaTime;
        float alpha;
        alpha = currentDuration / halfDuration;
        SetAlphaCartas(alpha);
        yield return null;
    }
}

// Animación para que desaparezcan las cartas
private IEnumerator DesaparecerCartas()
{
    float duration = 5f; // Duración en segundos de la Anim
    float halfDuration = duration / 2f;
    float currentDuration = 0f;
    MostrarElementosDios();
    MostrarElementosJugador();
    SetAlphaCartas(0); // El alpha de las cartas a 0

    while (currentDuration < duration)
    {
        currentDuration += Time.deltaTime;
        float alpha;
        alpha = 1 - (currentDuration - halfDuration) / halfDuration;
```

```
        SetAlphaCartas(alpha);
        yield return null;
    }

}

// Establece el Alpha de las cartas y de la imágenes de la partida
private void SetAlphaCartas(float alpha)
{
    SetAlphaImagen(imagenDios, alpha);
    SetAlphaImagen(vidaDios, alpha);
    SetAlphaImagen(vidaJugador, alpha);
    SetAlphaImagen(cartal, alpha);
    SetAlphaImagen(carta2, alpha);
    SetAlphaImagen(carta3, alpha);
    SetAlphaImagen(carta4, alpha);
    SetAlphaTextMesh(txtNombreDios, alpha);
    SetAlphaTextMesh(txtNombreJugador, alpha);
    SetAlphaTextMesh(txtVidaDios, alpha);
    SetAlphaTextMesh(txtVidaJugador, alpha);
}

// Establece el Alpha de una imagen
private void SetAlphaImagen(Image imagen, float alpha)
{
    imagen.color = new Color(imagen.color.r, imagen.color.g,
    imagen.color.b, alpha);
}

// En base al nivel de juego tenemos un/a Dios/a u otro/a
private void SetDiosPorNivel()
{
    int numAleatorio = aleatorio.Next(2);
    switch(nivelJuego)
    {
        case 1:
            if(numAleatorio==0)
                diosActual=dioses[0]; // Brigid
            else
                diosActual=dioses[1]; // Cernunnos
            break;
        case 2:
            if(numAleatorio==0)
```

```
        diosActual=dioses[2]; // Lugh
    else
        diosActual=dioses[3]; // Morrigan
    break;
case 3:
    if(numAleatorio==0)
        diosActual=dioses[4]; // Dagda
    else
        diosActual=dioses[5]; // Dana
    break;
}
// Una vez elegido el dios actualizamos la información en la UI
imagenDios.sprite = diosActual.imagenDios;
txtNombreDios.text = diosActual.nombre;
txtVidaDios.text = diosActual.vidaInicial.ToString();
}

// Obtenemos una carta aleatoria de las que tenemos cargadas
private PlantillaCarta GetCartaAleatoria()
{
    int numCartas = cartas.Count;
    int numAleatorio = aleatorio.Next(numCartas);
    return cartas[numAleatorio];
}

// Actualizamos la UI de las cartas en el juego
private void ActualizarCartasUI()
{
    carta1Actual = cartasActuales[0];
    carta1.sprite = cartasActuales[0].imagenCarta;
    carta2Actual = cartasActuales[1];
    carta2.sprite = cartasActuales[1].imagenCarta;
    carta3Actual = cartasActuales[2];
    carta3.sprite = cartasActuales[2].imagenCarta;
    carta4Actual = cartasActuales[3];
    carta4.sprite = cartasActuales[3].imagenCarta;
}

// Establecemos un nivel de vida al jugador + cartas para jugar
private void EmpezarNivelCartas()
{
    //Primero ocultamos los elementos del Dios y del Jugador
```

```
OcultarElementosDios();  
OcultarElementosJugador();  
// Ocultamos los ataques del dios y del Leprechaun  
OcultarTextAtaqueDios();  
OcultarTextLeprechaun();  
// Hacemos la animación de Cambio de Nivel  
StartCoroutine(AnimarCambioNivel());  
// Insertamos 4 cartas en la baraja de la partida  
for(int i=0; i<numCartas; i++)  
{  
    cartasActuales[i]=GetCartaAleatoria();  
}  
// Sumamos entre 0 y 200 x Nivel a la vida del jugador  
int vidaASumar = aleatorio.Next(200);  
int vidaJugador = int.Parse(txtVidaJugador.text);  
txtVidaJugador.text = (vidaJugador + vidaASumar).ToString();  
ActualizarCartasUI();  
  
}  
  
// Establece el comienzo de un nivel de juego  
private void EmpezarNivelJuego()  
{  
    SetDiosPorNivel();  
    EmpezarNivelCartas();  
}  
  
private int GetVidaDios()  
{  
    return int.Parse(txtVidaDios.text);  
}  
  
private int GetVidaJugador()  
{  
    return int.Parse(txtVidaJugador.text);  
}  
  
// Obtenemos los puntos de ataque de la carta que buscamos  
private int GetAtaqueCarta(int numero)  
{  
    int ataque=0;  
    switch(numero)  
    {
```

```
        case 1:
            ataque=carta1Actual.ataque;
            break;
        case 2:
            ataque=carta2Actual.ataque;
            break;
        case 3:
            ataque=carta3Actual.ataque;
            break;
        case 4:
            ataque=carta4Actual.ataque;
            break;
    }
    return ataque;
}

// Obtenemos los puntos de ataque de la carta que buscamos
private int GetVidaCarta(int numero)
{
    int vida=0;
    switch(numero)
    {
        case 1:
            vida=carta1Actual.vida;
            break;
        case 2:
            vida=carta2Actual.vida;
            break;
        case 3:
            vida=carta3Actual.vida;
            break;
        case 4:
            vida=carta4Actual.vida;
            break;
    }
    return vida;
}

// Obtenemos el nombre de la carta
private string GetNombreCarta(int numero)
{
    string nombre="";

```

```
switch(numero)
{
    case 1:
        nombre=carta1Actual.nombreCarta;
        break;
    case 2:
        nombre=carta2Actual.nombreCarta;
        break;
    case 3:
        nombre=carta3Actual.nombreCarta;
        break;
    case 4:
        nombre=carta4Actual.nombreCarta;
        break;
}
return nombre;
}

// Ataque del Leprechaun entre 100 y 200
private int GetAtaqueLeprechaun()
{
    int ataqueBase=100;
    int ataqueAleatorio=aleatorio.Next(100);
    return ataqueBase+ataqueAleatorio;
}

// Vida a sumar del Leprechaun entre 100 y 200
private int GetVidaLeprechaun()
{
    int vidaBase=100;
    int vidaAleatorio=aleatorio.Next(100);
    return vidaBase+vidaAleatorio;
}

// Después de usar una carta se actualiza a una nueva
private void ActualizarCarta(int numero)
{
    cartasActuales[numero-1]=GetCartaAleatoria();
    ActualizarCartasUI();
}

// El Dios ataca / recupera vida según lo que le corresponde
```

```

private int AtaqueDios()
{
    txtAtaqueDios.enabled=true; // Activamos la vista del ataque del
Dios

    int ataqueDios=0;
    // 0 y 1 => Ataques del Dios/a; 2 => recuperar vida el Dios/a
    int numAleatorio = aleatorio.Next(3);
    float probabilidad = Random.Range(0f, 1f); // Genera un número
aleatorio entre 0 y 1
    // Ataque
    if(numAleatorio>=0 && numAleatorio<=1)
    {
        if (probabilidad < diosActual.porcenAtaque1) // % de
probabilidad de ataque1
        {
            ataqueDios=diosActual.ataque1;
            txtAtaqueDios.text = diosActual.txtAtaque1 +
" ("+ataqueDios.ToString()+" )";
        }
        else // % restante => ataque2
        {
            ataqueDios=diosActual.ataque2;
            txtAtaqueDios.text = diosActual.txtAtaque2 +
" ("+ataqueDios.ToString()+" )";
        }
        txtVidaJugador.text = (int.Parse(txtVidaJugador.text) -
ataqueDios).ToString();
    }
    else // Defensa
    {
        int defensaDios=diosActual.vida;
        txtAtaqueDios.text=diosActual.txtVida + " ("+defensaDios+" )";

txtVidaDios.text=(int.Parse(txtVidaDios.text)+defensaDios).ToString();
    }
    StartCoroutine(AnimarAtaqueDios());// Animamos el Ataque del Dios
para verlo claro
    return ataqueDios;
}

private IEnumerator MostrarAtaqueLeprechaun(int ataque, int vida)
{

```

```
float duration = 1f; // Duración en segundos de la Anim
float halfDuration = duration / 2f;
float currentDuration = 0f;
MostrarTextLeprechaun();

txtLeprechaun.text="Leprechaun"+`\nAtaque("+ataque.ToString()+"")\nVida("+vida.ToString()+"")`;
SetAlphaTextMesh(txtLeprechaun, 0);
while (currentDuration < duration)
{
    currentDuration += Time.deltaTime;

    float alpha;
    if (currentDuration <= halfDuration)
    {
        alpha = currentDuration / halfDuration;
    }
    else
    {
        alpha = 1 - (currentDuration - halfDuration) / halfDuration;
    }
    SetAlphaTextMesh(txtLeprechaun, alpha);
    yield return null;
}
OcultarTextLeprechaun();
}

// El jugador ataca / recupera vida con la carta que tiene escogida
public void AtaqueCarta(int numero)
{
    int ataque=GetAtaqueCarta(numero);
    int vida=GetVidaCarta(numero);
    // Actualizamos la UI para Dios y Jugador
    txtVidaDios.text=(int.Parse(txtVidaDios.text)-ataque).ToString();
    txtVidaJugador.text=(int.Parse(txtVidaJugador.text)+vida).ToString();
    // Si la carta es el Leprechaun => usamos sus poderes
    if(GetNombreCarta(numero)=="Leprechaun")
    {
        int ataqueLeprechaun=GetAtaqueLeprechaun();
        int vidaLeprechaun=GetVidaLeprechaun();
        txtVidaDios.text=(int.Parse(txtVidaDios.text)-ataqueLeprechaun).ToString();
        txtVidaJugador.text=(int.Parse(txtVidaJugador.text)+vidaLeprechaun).ToString();
        StartCoroutine(MostrarAtaqueLeprechaun(ataqueLeprechaun, vidaLeprechaun));
    }
    // Una vez usada la carta la cambiamos
```

```
ActualizarCarta(numero);
ActualizarCartasUI();
// Después de un Ataque de una Carta viene el Ataque del Dios
AtaqueDios();
// Si muere el dios => cambiamos de nivel
if(GetVidaDios()<0)
{
    Debug.Log("Cambio de Nivel");
    nivelJuego++;
    EmpezarNivelJuego();
}
// Si muere el jugador => EscenaPerder
if(GetVidaJugador()<0)
    SceneManager.LoadScene("EscenaPerder");
// Si el Nivel de Juego es > 3 => Ganamos la partida
if(nivelJuego>3)
    SceneManager.LoadScene("EscenaGanar");
}
```

5- Posibles Mejoras del Juego

Aquí tienes varias ideas de mejoras para este juego de cartas RPG que podrían enriquecer la experiencia de juego y añadir más profundidad a la dinámica:

1. Añadir más animaciones

Hay animaciones que pueden mejorar la sensación del juego como son los ataques de las cartas o los ataques de los Dioses. Te recomiendo que eches un vistazo al juego [Marvel Snap](#) que es un juego de cartas de este estilo y que tiene una cantidad de animaciones muy interesantes.

2. Desacoplar el código

Tras programar estos días el juego tengo la sensación de que, sin usar patrones de diseño o una planificación bien pensada del diseño, es muy fácil que nos quede un código del juego muy acoplado y, por tanto, difícil de mantener. Un patrón MVC (Modelo Vista Controlador) nos podría ayudar a separar la parte de la UI de la parte de la lógica del juego. Sería un apartado interesante de desarrollar para crear un tipo de juego que se pueda reutilizar de forma sencilla.

3. Sistema de Habilidades y Mejoras

Incorpora un sistema de habilidades o mejoras que los jugadores puedan aplicar a sus cartas. Esto podría incluir potenciadores temporales, habilidades especiales, o mejoras permanentes que se adquieren a lo largo del juego. Esto añadiría una capa estratégica adicional, ya que los jugadores tendrían que decidir cuándo y cómo usar estas habilidades.

4 Modos de Juego Variados

Añade diferentes modos de juego para mantener el interés y ofrecer nuevas formas de jugar. Esto podría incluir modos de tiempo limitado, desafíos específicos, o modos de historia con objetivos únicos. Estos modos pueden ofrecer recompensas especiales y ayudar a mantener el juego fresco y emocionante.

5. Sistema de Progresión y Recompensas

Implementa un sistema de progresión donde los jugadores puedan desbloquear nuevas cartas, habilidades, o contenido a medida que avanzan. Las recompensas podrían estar basadas en logros, completar ciertos niveles o desafíos, y ayudarían a mantener a los jugadores comprometidos a largo plazo.

6. Interacciones y Sinergias entre Cartas

Desarrolla interacciones y sinergias entre diferentes cartas para fomentar la creación de estrategias y barajas. Las cartas podrían tener efectos que se potencian o cambian cuando se juegan junto a ciertas otras cartas, creando una capa más profunda de estrategia y planificación.

7. Elementos Narrativos y Mundo Expansivo

Integra elementos narrativos y construcción de mundo en el juego. Esto podría incluir historias para cada carta, misiones secundarias, y un mundo expansivo que los jugadores pueden explorar a través del juego. La inclusión de una narrativa fuerte puede aumentar el compromiso emocional y el interés en el juego.

Estas mejoras no solo aumentan la profundidad y complejidad del juego, sino que también ofrecerían a los jugadores una experiencia más rica y variada, manteniendo el interés y la participación a lo largo del tiempo.

Anexo 1- Manual y Clases de Unity que necesitamos

0- Manual de Unity

<https://docs.unity3d.com/es/current/Manual/index.html>

The screenshot shows the Unity Documentation website for version 2019.4 LTS. The left sidebar contains a navigation menu with sections like Manual, Unity User Manual (2019.4 LTS), Packages, New in Unity 2019, Working in Unity, Importing, Input, 2D, Gráficos, Física, Scripting, Multijugador y Networking (redes), Audio, Descripción general de video, Animación, User interfaces (UI), Navegación y Pathfinding (Búsqueda de Caminos), Servicios de Unity, XR, Repositorios Open-source, Asset Store Publishing, and Platform development. The main content area displays the Unity User Manual (2019.4 LTS) with a title, introduction, and several sections: Nuevo, Packages, and Mejores prácticas y guías de expertos.

Coroutine

En Unity, una **Coroutine** es una función especial que permite pausar su ejecución y reanudarla en un punto posterior. Esto es especialmente útil para tareas que se extienden a lo largo del tiempo, como animaciones, esperas, o procesos que ocurren durante varios frames. Las coroutines son una parte fundamental de Unity para manejar el tiempo y las operaciones asincrónicas sin bloquear el flujo principal del juego.

Características Clave de una Coroutine en Unity

- Pausar y Reanudar la Ejecución: Las coroutines pueden interrumpir su ejecución en un punto específico, devolver el control al motor de Unity, y luego continuar desde ese punto en un momento posterior.
- Control del Tiempo y Asincronía: Son ideales para acciones que necesitan un retraso o que deben ocurrir a lo largo de varios frames, como retrasos temporizados, animaciones suaves, o la carga de recursos en segundo plano.

- Uso de yield: Las coroutines utilizan la palabra clave yield para indicar cuándo y cómo deben pausar su ejecución. Por ejemplo, yield return null pausa la coroutine hasta el siguiente frame, mientras que yield return new WaitForSeconds(1f) la pausará durante un segundo.

Inicio con StartCoroutine: Para ejecutar una coroutine, se utiliza el método StartCoroutine() de un MonoBehaviour.

```
using System.Collections;
using UnityEngine;

public class EjemploCoroutine : MonoBehaviour
{
    void Start()
    {
        // Iniciar la coroutine
        StartCoroutine(MiCoroutine());
    }

    IEnumerator MiCoroutine()
    {
        // Hacer algo inmediatamente
        Debug.Log("Inicio de la coroutine");

        // Esperar 2 segundos
        yield return new WaitForSeconds(2f);

        // Hacer algo después de 2 segundos
        Debug.Log("2 segundos después");
    }
}
```

yield return null

La línea yield return null; en una coroutine de Unity tiene un significado especial y es un elemento clave para entender cómo funcionan las coroutines.

En Unity, una coroutine es una función que puede pausar su ejecución y reanudarla en el próximo frame o después de un cierto tiempo, permitiendo realizar tareas a lo largo del tiempo sin bloquear el resto del juego.

Cuando en una coroutine encuentras la línea yield return null;, significa lo siguiente:

1. **Pausar la Coroutine:** En el punto donde aparece yield return null;, la ejecución de la coroutine se pausa.
2. **Esperar hasta el Próximo Frame:** La coroutine continuará su ejecución en el siguiente frame del juego. Esencialmente, esto indica a Unity que espere hasta que se haya completado el frame actual, incluyendo el renderizado y otros procesos del motor, antes de continuar con el siguiente paso de la coroutine en el próximo frame.
3. **Reanudar la Ejecución:** En el próximo frame, la coroutine reanuda su ejecución justo después de la línea yield return null;.

Por ejemplo, en una animación o en un proceso que necesitas que ocurra a lo largo de varios frames, usarías yield return null; dentro de un bucle para asegurarte de que cada paso de la animación o proceso se lleva a cabo frame a frame, en lugar de todo de una vez.

Este mecanismo permite crear animaciones suaves, temporizadores, retrasos y otras funcionalidades que dependen del tiempo o de los frames, sin la necesidad de complicados sistemas de seguimiento del tiempo o múltiples funciones y actualizaciones.

Font

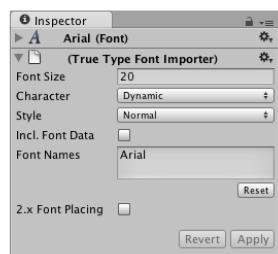
Font (Fuente)

Los Fonts pueden ser creados o importados para uso ya sea desde [GUI Text](#) o el [Text Mesh Components](#).

Importando archivos de Fuente (font)

Para agregar una fuente a su proyecto usted necesita colocar el archivo fuente en su carpeta de Assets. Unity luego automáticamente lo va a importar. Los formatos de fuentes soportados son TrueType Fonts (archivos .ttf) y OpenType Fonts (archivos .otf).

Para cambiar el **Size** (tamaño) de la font (fuente), resalte la en el [Project View](#) y usted tendrá un número de opciones en los **Import Settings** (ajustes de importación) en el [Inspector](#).



MonoBehaviour

Resumen de su Función Principal:

MonoBehaviour es la clase base de la cual todos los scripts de Unity que se adjuntan a los objetos del juego (GameObjects) derivan. Proporciona acceso a funciones importantes del ciclo de vida del juego, como Start, Update, FixedUpdate, y más. Estas funciones son automáticamente llamadas por el motor de Unity en diferentes puntos del ciclo de vida del juego.

Cómo las Usamos en los Juegos:

Los scripts que heredan de MonoBehaviour se pueden adjuntar a GameObjects para darles comportamiento. Son esenciales para la mayoría de las funcionalidades interactivas en Unity, desde controlar la entrada del jugador hasta mover objetos en la escena y gestionar la lógica del juego.

Ejemplos de su Uso:

- Un script PlayerController que maneja la entrada del jugador para mover un personaje.
- Un script EnemyAI que define la inteligencia artificial de un enemigo.
- Un script GameManager que controla el flujo y las reglas del juego.

Métodos de MonoBehaviour

Tenemos en esta imagen resumen ordenados los métodos de MonoBehaviour por orden de ejecución y con los usos habituales de cada uno de ellos.

```
void Awake()
{
    /* Referencias entre Scripts, inicialización de variables
     * Referencia a componentes: anim = GetComponent<Animator>();
     * ¡Importante! Se ejecuta aunque esté el script deshabilitado*/
}

@ Mensaje de Unity | 0 referencias
void OnEnable()
{
    //Se ejecuta cuando se activa el objeto
}

@ Mensaje de Unity | 0 referencias
void Start()
{
    /* Se ejecuta antes del primer frame y solo si el script está habilitado
     * Meter aquí: delays, permitir movimiento de enemigos..
     * Puede definirse como una coroutine: IEnumerator Start()*/
}

@ Mensaje de Unity | 0 referencias
void FixedUpdate()
{
    /* Se ejecuta cada X segundos (0.02 normalmente)
     * No depende de la máquina donde se esté ejecutando
     * Meter aquí: movimientos por físicas*/
}

@ Mensaje de Unity | 0 referencias
void Update()
{
    /* Se ejecuta una vez por frame
     * La llamada entre updates no es estable y depende de
     * donde se esté ejecutando el juego
     * Meter aquí: Input y actualización de variables*/
}

@ Mensaje de Unity | 0 referencias
void LateUpdate()
{
    /* Se ejecuta después de todos los Update
     * Suele usarse para controlar movimiento de la cámara*/
}

@ Mensaje de Unity | 0 referencias
void OnDisable()
{
    //Se ejecuta cuando se desactiva el objeto
}
```

ScriptableObject

Resumen de su Función Principal:

ScriptableObject es una clase de Unity usada para almacenar datos que no necesitan estar vinculados a un GameObject. Los objetos de este tipo son útiles para guardar configuraciones, estados, o información que debe ser accesible desde varios lugares sin la necesidad de copiarlos o recrearlos.

Cómo las Usamos en los Juegos:

Los ScriptableObjects son ideales para almacenar datos como configuraciones, tablas de estadísticas, diálogos, o cualquier otra información que sea parte del juego pero que no requiera estar directamente atada a la lógica de los GameObjects. Pueden ser creados y editados dentro del editor de Unity y luego referenciados en los scripts de MonoBehaviour.

Ejemplos de su Uso:

- Un objeto de datos GameSettings que almacena configuraciones del juego como volumen, dificultad, etc.
- Un InventoryItem ScriptableObject que define las propiedades de un ítem en un juego de rol.
- Un Dialogue ScriptableObject que contiene la información de diálogos y secuencias de texto en un juego narrativo.

OJO:

- Si usas los datos del ScriptableObject directamente => modifica los datos en tu objeto almacenado
 - Si es una configuración inicial => **modifica una copia** de ese valor!!!!

