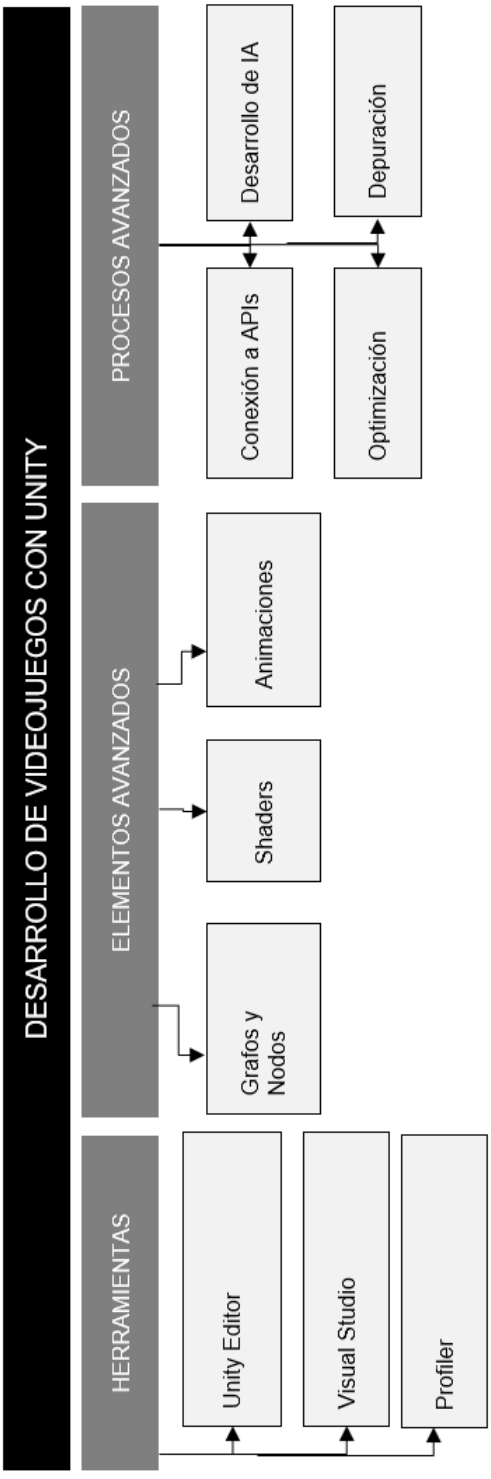
Programación multimedia y dispositivos móviles

Desarrollo de videojuegos en Unity

Índice

Esquema


DESARROLLO DE VIDEOJUEGOS CON UNITY
HERRAMIENTAS
Unity Editor
Visual Studio
Profiler
ELEMENTOS AVANZADOS
Grafos y Nodos
Shaders
Animaciones
PROCESOS AVANZADOS
Conexión a APIs
Optimización
Desarrollo de IA
Depuración

Material de estudio

10.1. Introducción y objetivos

Los objetivos que se persiguen en el siguiente tema son:

- Conocer las distintas arquitecturas de software que tenemos disponibles para el desarrollo de videojuegos.
- Aprender qué son los motores gráficos de videojuegos.
- Comparar distintos motores gráficos de videojuegos.
- Conocer Unity, sus características y su importancia en el mercado.

10.2. Scripting: C#

El scripting en Unity utilizando C# es una de las principales formas de personalizar y programar el comportamiento de los objetos en tus proyectos de videojuegos o aplicaciones interactivas. Unity eligió C# como su lenguaje de scripting principal debido a su facilidad de uso, potencia, y compatibilidad con las herramientas y entornos modernos.

Principales conceptos del scripting en Unity con C#

1. Scripts como componentes:

En Unity, cada script que escribes en C# es un componente que puedes adjuntar a un GameObject en tu escena. Los scripts controlan el comportamiento del GameObject al que están asignados.

```
using UnityEngine;

public class ExampleScript : MonoBehaviour
{
```

```
    // Se ejecuta al iniciar el juego
    void Start()
    {
        Debug.Log("Hola, mundo!");
    }

    // Se ejecuta cada frame
    void Update()
    {
        Debug.Log("Actualizando cada frame");
    }
}
```

En este ejemplo:

- Start() se ejecuta al inicio del juego o cuando el GameObject con este script es activado.
- Update() se ejecuta una vez por frame.

2. Ciclo de vida de un script:

Unity ofrece varios métodos específicos que se ejecutan en diferentes puntos del ciclo de vida de un GameObject. Algunos de los más comunes son:

- Awake(): Se llama al cargar el objeto, incluso antes de que la escena comience.
- Start(): Se llama justo antes de la primera actualización del frame.
- Update(): Se llama en cada frame.
- FixedUpdate(): Se llama en intervalos de tiempo fijos, ideal para física.
- OnDestroy(): Se llama cuando el objeto se destruye.

3. Uso del sistema de componentes:

Los scripts trabajan en conjunto con otros componentes, como Rigidbodies, Colliders o AudioSources. Puedes acceder y manipular otros componentes desde el script:

```
public class ControladorMovimiento : MonoBehaviour
{
    private Rigidbody rb;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
    }

    void Update()
    {
        if (Input.GetKey(KeyCode.W))
        {
            rb.AddForce(Vector3.forward * 10f);
        }
    }
}
```

En este caso:

GetComponent<T>() obtiene un componente específico (como el Rigidbody) del GameObject.

4. Manipulación de objetos y transformaciones:

Puedes controlar las propiedades de un GameObject, como su posición, rotación y escala, a través de su componente Transform.

```
public class RotarObjeto : MonoBehaviour
{
    void Update()
    {
        transform.Rotate(0, 50 * Time.deltaTime, 0);
    }
}
```

5. Eventos y colisiones:

Unity permite reaccionar a interacciones entre objetos a través de métodos específicos como:

- OnCollisionEnter(): Detecta colisiones entre objetos con colliders.
- OnTriggerEnter(): Detecta cuando un objeto entra en un trigger.

```
void OnCollisionEnter(Collision col)
{
    Debug.Log("Colisión con: " + col.gameObject.name);
}
```

6. Interacción con el jugador:

Unity proporciona herramientas como Input para capturar las interacciones del usuario.

```
void Update()
{
    if (Input.GetKeyDown(KeyCode.Space))
    {
        Debug.Log("¡Se presionó la barra espaciadora!");
    }
}
```

Métodos del ciclo de vida de un script

Los métodos del ciclo de vida de un script en Unity son funciones específicas que Unity llama automáticamente en momentos clave durante la ejecución del juego. Estos métodos están diseñados para ayudarte a interactuar con los eventos que ocurren en la escena, como inicialización, actualizaciones por frame, interacciones físicas y destrucción.

1. Inicialización

- Awake(): Es el primer método que se ejecuta cuando un script es cargado. Se llama antes de que el objeto esté activado en la escena. Se utiliza para inicializar variables o configurar estados iniciales independientemente de otros objetos.

```
void Awake(){
    Debug.Log("Awake: Este script se ha cargado.");
}
```

- OnEnable(): Se llama cada vez que el GameObject o el script es activado. Es útil si necesitas ejecutar lógica cada vez que un objeto vuelve a estar activo.

```
void OnEnable()
{
    Debug.Log("OnEnable: Este script está activo.");
}
```

- Start(): Se llama una vez, justo antes de que comience la primera actualización de frame (Update). Se usa para inicializar lógica que depende de otros objetos en la escena.

```
void Start()
{
    Debug.Log("Start: El juego ha comenzado.");
}
```

2. Actualización:

- Update(): Se llama en cada frame. Se usa para manejar lógica continua, como controles de jugador o animaciones.

```
void Update()
{
    Debug.Log("Update: Este mensaje aparece cada frame.");
}
```

- FixedUpdate(): Se llama en intervalos de tiempo fijos. Es ideal para cálculos relacionados con la física, ya que se sincroniza con el motor de física de Unity.

```
void FixedUpdate()
{
    Debug.Log("FixedUpdate: Se ejecuta en intervalos fijos.");
}
```

- LateUpdate(): Se llama al final de cada frame. Se usa para lógica que debe ejecutarse después de que todos los cálculos de otros Update() hayan finalizado. Por ejemplo, ajustar la cámara después de que los objetos se hayan movido.

```
void LateUpdate()
{
    Debug.Log("LateUpdate: Se ejecuta después de Update.");
```

```
}
```

3. Eventos de física

- OnCollisionEnter(Collision collision): Se llama cuando el GameObject colisiona con otro objeto que tiene un Collider.

```
void OnCollisionEnter(Collision collision)
{
    Debug.Log("Colisión con: " + collision.gameObject.name);
}
```

- OnTriggerEnter(Collider other): Se llama cuando un objeto entra en un área de Trigger.

```
void OnTriggerEnter(Collider other)
{
    Debug.Log("Entró al trigger de: " + other.gameObject.name);
}
```

Métodos relacionados:

- OnCollisionExit(Collision collision): Cuando un objeto deja de colisionar.
- OnTriggerExit(Collider other): Cuando un objeto sale de un Trigger.
- OnCollisionStay(Collision collision) y OnTriggerStay(Collider other): Cuando un objeto permanece en contacto o dentro de un Trigger.

4. Eventos de desactivación

OnDisable(): Se llama cada vez que el script o el objeto se desactiva. Es útil para detener lógica o liberar recursos temporalmente.

```
void OnDisable()
{
    Debug.Log("OnDisable: Este script ha sido desactivado.");
}
```

5. Eventos de destrucción

OnDestroy(): Se llama cuando el GameObject se destruye o la escena cambia. Es útil para liberar recursos, detener corutinas o guardar datos.

```
void OnDestroy()
{
    Debug.Log("OnDestroy: Este objeto ha sido destruido.");
}
```

Flujo general del ciclo de vida

1. Awake()
2. OnEnable()
3. Start()
4. (Cada frame):
    - Update()
    - FixedUpdate() (en intervalos fijos)
    - LateUpdate()
5. (Eventos físicos si ocurren):
    - OnCollisionEnter(), OnTriggerEnter(), etc.
6. OnDisable() (cuando el objeto se desactiva)
7. OnDestroy() (cuando el objeto se destruye)

Este flujo te permite controlar cómo y cuándo ocurren los eventos en tu juego, permitiendo crear comportamientos dinámicos y estructurados.

- Orden de ejecución: Aunque Awake() siempre se llama antes de Start(), el orden de ejecución entre diferentes scripts puede configurarse en el Project Settings > Script Execution Order.
- Uso eficiente: Usa FixedUpdate() únicamente para cálculos relacionados con física. Evita lógica costosa dentro de Update().

10.3. Uso de shaders

Los shaders en Unity son programas que controlan cómo los gráficos se procesan y se renderizan en la pantalla. Son esenciales para crear efectos visuales personalizados y se ejecutan principalmente en la GPU, lo que los hace altamente eficientes para operaciones gráficas.

Unity utiliza el lenguaje de sombreado HLSL (High-Level Shading Language) a través de su sistema de shaders. Además, proporciona diferentes formas de trabajar con shaders, desde los más básicos hasta los avanzados, utilizando herramientas como el Shader Graph.

Tipos de Shaders en Unity

1. Shaders estándar (Standard Shader):

Son shaders integrados que Unity proporciona por defecto. Ofrecen una configuración versátil para materiales PBR (Physically Based Rendering). Son fáciles de usar para materiales comunes como metal, plástico, madera, etc.

2. Surface Shaders:

Es una forma simplificada de escribir shaders en Unity. Permiten definir cómo interactúa una superficie con la luz, ideal para sombreado avanzado. Abstraen gran parte de la complejidad de los cálculos de iluminación.

3. Vertex y Fragment Shaders:

Se tratan de shadersmás avanzados y personalizados. Requieren controlar directamente las operaciones en los vértices y fragmentos (pixeles). Ofrecen mayor flexibilidad para crear efectos únicos.

4. Shader Graph:

Una herramienta visual para crear shaders sin escribir código. Permite diseñar shaders complejos mediante nodos, ideal para artistas y desarrolladores que prefieren interfaces gráficas.

Ejemplo: Shader simple escrito directamente en Unity para cambiar el color de un objeto.

```
Shader "Custom/SimpleColorShader"
{
    Properties
    {
        _Color ("Color", Color) = (1, 0, 0, 1) // Propiedad para el color
    }
    SubShader
    {
        Tags { "RenderType"="Opaque" }
        Pass
        {
            CGPROGRAM
            #pragma vertex vert
            #pragma fragment frag
            struct appdata_t
            {
                float4 vertex : POSITION;
            };

            struct v2f
            {
                float4 pos : SV_POSITION;
            };

            fixed4 _Color;
            v2f vert (appdata_t v)
            {
                v2f o;
                o.pos = UnityObjectToClipPos(v.vertex);
                return o;
            }

            fixed4 frag (v2f i) : SV_Target
            {
                return _Color;
            }
            ENDCG
        }
    }
}
```

5. Partes clave del shader:

- Properties: Define las variables expuestas en el inspector (por ejemplo, _Color). Estas pueden ser configuradas directamente en el editor o mediante scripts.
- SubShader: Contiene las instrucciones sobre cómo renderizar el material. Incluye una o más Pass que representan etapas del proceso de renderizado.
- CGPROGRAM / HLSL: Bloque de código donde defines los cálculos del shader. Se compone de dos funciones principales:
    - Vertex Shader (vert): Procesa los vértices de un objeto.
    - Fragment Shader (frag): Calcula el color de cada píxel.

Shader Graph

Unity incluye el Shader Graph, una herramienta visual que permite diseñar shaders complejos sin necesidad de escribir código. Es parte del sistema de renderizado SRP (Scriptable Render Pipeline), compatible con HDRP y URP.

Características principales:

- Nodos visuales: Crea efectos visuales arrastrando y conectando nodos.
- Previsualización en tiempo real: Ve los resultados mientras diseñas.
- Compatibilidad con SRP: Integrado con URP (Universal Render Pipeline) y HDRP (High Definition Render Pipeline).

10.4. Animaciones

Unity utiliza un sistema de animación basado en componentes donde los clips de animación se aplican a los objetos en la escena mediante el Animator Controller. Esto permite una gestión avanzada de las transiciones entre animaciones y el control de su reproducción.

Componentes principales del sistema de animación

1. Animator:

Es un componente que se asigna a un GameObject para controlar sus animaciones. Funciona en conjunto con un Animator Controller para gestionar qué animación se reproduce y cuándo.

2. Animator Controller:

Define un flujo de animaciones mediante una máquina de estados. Permite gestionar las transiciones entre clips de animación usando condiciones y parámetros.

3. Clips de animación:

Archivos que contienen datos de animación, como posiciones de huesos, rotaciones, escalas, o cambios en propiedades. Estos clips pueden ser creados dentro de Unity o importados desde software como Blender, Maya, o 3ds Max.

4. Avatar:

Es un recurso utilizado para mapear las animaciones en personajes humanoides. Permite reutilizar animaciones en diferentes modelos humanoides.

5. Timeline (línea de tiempo):

Herramienta avanzada para crear secuencias cinemáticas que combinan animaciones, cámaras, efectos visuales, y más.

Tipos de animaciones en Unity

1. Animaciones de personajes (Rigging):

Utilizan un esqueleto para animar modelos humanoides. Compatible con el sistema de Mecanim, que facilita el uso de avatares para compartir animaciones entre modelos humanoides.

2. Animaciones basadas en propiedades:

Permiten animar cualquier propiedad de un GameObject, como posición, rotación, escala o material. Se crean directamente en el Animation Window de Unity.

3. Animaciones de objetos físicos:

Utilizan físicas en tiempo real para crear animaciones dinámicas, como cuerpos rígidos moviéndose o reacciones a fuerzas.

Control de animaciones con Animator

El Animator Controller organiza y controla las animaciones usando una máquina de estados:

1. Estados:

Cada animación es un estado dentro del Animator Controller. Ejemplo: "Idle", "Run", "Jump".

2. Transiciones:

Conectan los estados y definen cuándo cambiar de una animación a otra.

Las transiciones pueden tener condiciones basadas en parámetros (bool, float, int, trigger). Ejemplo: Transición de "Idle" a "Run" cuando el parámetro isRunning es true.

3. Parámetros:

Variables que controlan las transiciones. Se pueden configurar desde scripts:

```
Animator animator = GetComponent<Animator>();

void Update()
{
    if (Input.GetKey(KeyCode.W))
    {
        animator.SetBool("isRunning", true);
    }
    else
    {
        animator.SetBool("isRunning", false);
    }
}
```

Crear animaciones en Unity

1. Usando el Animation Window:

- Selecciona un GameObject y abre la ventana Animation (Window > Animation).
- Crea un nuevo clip de animación y comienza a agregar fotogramas clave (keyframes) para definir cambios en las propiedades del objeto.

Ejemplo: Animar un cubo para que suba y baje.

```
1. Selecciona el cubo en la escena.
2. Abre la ventana de Animation.
3. Crea un nuevo clip llamado "SubeBaja".
4. En el primer fotograma, establece la posición inicial.
5. En el fotograma 30, mueve el cubo hacia arriba.
6. Reproduce la animación.
```

2. Importar animaciones desde un software 3D:

Unity soporta archivos de animación en formatos como .fbx, .dae, o .blend. Los clips de animación se importan junto con el modelo y se pueden configurar desde el inspector.

3. Usando Timeline:

- Abre la ventana Timeline (Window > Sequencing > Timeline).
- Crea un nuevo timeline para tu GameObject.
- Añade pistas (tracks) para animaciones, audio, o eventos.

10.5. Grafos y nodos

Los grafos y nodos en Unity son conceptos esenciales en varios sistemas visuales y arquitecturas dentro del motor. Unity utiliza grafos y nodos como interfaces visuales para representar y gestionar lógicas, comportamientos o flujos complejos de forma intuitiva. Estos sistemas son ampliamente utilizados tanto por desarrolladores como artistas.

¿Qué son los grafos y nodos?

- Nodos: Representan puntos individuales dentro de un flujo o red. Cada nodo realiza una tarea específica, como ejecutar un cálculo, aplicar un efecto o definir una acción.
- Grafo: Es una colección de nodos conectados entre sí, que forman una red lógica o visual. Representa un flujo de datos o lógica.

En Unity, los nodos se conectan mediante líneas o cables, que definen cómo fluye la información o los eventos entre ellos.

Aplicaciones de grafos y nodos en Unity

1. Shader Graph

Es una herramienta de diseño visual que permite crear shaders mediante un sistema de nodos. En lugar de escribir código en HLSL, puedes conectar nodos para definir la apariencia de materiales.

Componentes clave:

- Nodos de entrada: Datos como tiempo, posición del objeto o UVs.
- Nodos de procesamiento: Operaciones matemáticas, combinaciones de colores o efectos personalizados.
- Nodos de salida: Determinan el resultado final, como el color o la opacidad.

Ejemplo: Un grafo que usa un nodo Time y un nodo Sin para animar el color de un material.

2. Visual Scripting (antiguamente Bolt)

Herramienta de programación visual basada en nodos para diseñar comportamientos y lógica sin necesidad de escribir código. Ideal para artistas o desarrolladores sin experiencia en programación.

Componentes clave:

- Nodos de eventos: Detectan interacciones o cambios (como OnClick o OnTriggerEnter).
- Nodos de acción: Ejecutan comandos (como mover un objeto o cambiar su color).

- Flujo: Las conexiones entre nodos definen el orden de ejecución.

Ejemplo: Crear un sistema de movimiento donde un objeto se desplace hacia adelante al presionar una tecla.

3. State Machine Graphs (Máquinas de estados)

Representan los diferentes estados de un objeto o personaje (por ejemplo, correr, saltar, caminar). Se utilizan principalmente en el sistema de Animator para controlar animaciones.

Componentes clave:

- Estados: Cada nodo representa un estado, como "Idle" o "Run".
- Transiciones: Conexiones que definen cómo y cuándo cambiar de un estado a otro.
- Parámetros: Variables que controlan las transiciones, como isRunning o jumpTrigger.

Ejemplo: Un personaje que pasa de "Idle" a "Run" cuando se presiona una tecla y vuelve a "Idle" cuando se suelta.

4. Visual Effect Graph

Una herramienta para crear efectos visuales avanzados como partículas, explosiones o simulaciones de fluidos. Funciona mediante grafos que controlan cómo las partículas nacen, se comportan y mueren.

Componentes clave:

- Nodos de emisión: Determinan cómo se generan las partículas.
- Nodos de comportamiento: Controlan propiedades como la velocidad o la dirección.
- Nodos de salida: Especifican cómo se renderizan las partículas.

Ejemplo: Crear un efecto de niebla donde las partículas se desplacen de forma aleatoria.

5. GraphView (Sistema Personalizado de Grafos)

Unity proporciona herramientas para desarrollar sistemas de grafos personalizados, como el sistema de diálogos o lógica específica. Se basa en GraphView API, que permite construir editores visuales.

Casos de uso:

- Sistemas de diálogo: Representar diálogos entre personajes mediante nodos de texto.
- Árboles de comportamiento: Controlar el flujo lógico de IA en NPCs.

10.6. Uso de APIs

Conectar Unity con APIs REST externas es una práctica común cuando necesitas integrar tu juego o aplicación con servicios web, como bases de datos remotas, sistemas de autenticación, almacenamiento en la nube o APIs de terceros. Unity proporciona herramientas integradas para realizar solicitudes HTTP, como UnityWebRequest, que es la clase principal para interactuar con servicios RESTful.

La clase UnityWebRequest es la forma principal de enviar y recibir datos desde una API REST, por lo tanto, proporciona un sistema modular para componer peticiones HTTP y manejar respuestas HTTP. El objetivo principal del sistema de UnityWebRequest es permitirles a los juegos de Unity interactuar con backends Web modernos. También soporta características de alta demanda tal como solicitudes HTTP fragmentadas, operaciones de streaming POST/PUT y un control completo sobre encabezados HTTP y verbs.

Ejemplo básico: Hacer una solicitud GET

```
using UnityEngine;
using UnityEngine.Networking;
using System.Collections;

public class APIClient : MonoBehaviour
```

```
{
    private string apiUrl = "https://jsonplaceholder.typicode.com/posts";

    void Start()
    {
        StartCoroutine(GetDataFromAPI());
    }

    IEnumerator GetDataFromAPI()
    {
        UnityWebRequest request = UnityWebRequest.Get(apiUrl);
        yield return request.SendWebRequest();

        if (request.result == UnityWebRequest.Result.ConnectionError ||
request.result == UnityWebRequest.Result.ProtocolError)
        {
            Debug.LogError("Error: " + request.error);
        }
        else
        {
            Debug.Log("Respuesta: " + request.downloadHandler.text);
        }
    }
}
```

- UnityWebRequest.Get(apiUrl): Realiza una solicitud HTTP GET.
- request.SendWebRequest(): Envía la solicitud y espera la respuesta.
- request.downloadHandler.text: Contiene la respuesta en formato de texto (generalmente JSON).

Para funcionalidades más avanzadas, puedes usar librerías como RestSharp o Newtonsoft.Json:

- RestSharp: Simplifica las solicitudes HTTP.
- Newtonsoft.Json: Ofrece mayor flexibilidad para trabajar con JSON.

10.7. Análisis de ejecución. Optimización del código

El análisis de ejecución y la optimización del código en Unity son procesos cruciales para garantizar que tu juego o aplicación funcione de manera eficiente en diferentes plataformas. Unity ofrece herramientas y mejores prácticas para identificar cuellos de botella, reducir el consumo de recursos y optimizar el rendimiento.

Fases del análisis y optimización

1. Medir el rendimiento actual

Antes de optimizar, identifica las áreas problemáticas. Usa herramientas como el Profiler, Frame Debugger o scripts personalizados para obtener métricas detalladas.

2. Identificar cuellos de botella

Determina qué partes del código o activos están afectando el rendimiento: CPU, GPU, memoria, física, o red.

3. Optimizar

Aplica técnicas específicas según el área problemática, como reducir llamadas de dibujo (draw calls), optimizar scripts, o comprimir texturas.

4. Probar

Verifica si las optimizaciones mejoraron el rendimiento sin comprometer la calidad o la funcionalidad.

Herramientas de análisis en Unity

1. Unity Profiler

Permite analizar el rendimiento en tiempo real.

Métricas clave:

► CPU Usage: Procesamiento de lógica y scripts.

- GPU Usage: Renderizado y procesamiento gráfico.
- Rendering: Draw calls, triángulos y vértices procesados.
- Memory: Uso de RAM y GC (Garbage Collection).

Acceso: Window > Analysis > Profiler.

2. Frame Debugger

Inspecciona cada paso del proceso de renderizado. Útil para identificar problemas como draw calls innecesarios.

Acceso: Window > Analysis > Frame Debugger.

3. Stats Window

Muestra estadísticas rápidas sobre el rendimiento gráfico.

Acceso: Haz clic en el botón "Stats" en la esquina superior derecha de la Game View.

4. Deep Profiling

Ofrece un análisis más detallado del código y la ejecución de funciones. Actívalo en el Profiler cuando necesites evaluar scripts.

5. Third-Party Tools

- Rider, Visual Studio: Para analizar y depurar código.
- RenderDoc: Para análisis de renderizado avanzado.

A fondo

UnityWebRequest

https://docs.unity3d.com/es/530/Manual/UNet.html

Documentación en línea. (2024)

Documentación oficial sobre UnityWebRequest.

Scripting en Unity

https://learn.unity.com/search?k=%5B%22q%3AScripting%22%5D

Documentación en línea. (2024)

Portal de learning oficial sobre el scripting en Unity.

Godot Engine

https://unity.com/features/shader-graph

Documentación en línea. (2024)

Documentación oficial sobre la herramienta Shader-graph de Unity.

Entrenamientos

Entrenamiento 1

- Realiza el itinerario oficial de scripting para principiantes.
- Desarrollo paso a paso y solución: https://learn.unity.com/project/scripting-para-principiantes

Entrenamiento 2

- Realiza el itinerario oficial de scripting de nivel intermedio.
- Desarrollo paso a paso y solución: https://learn.unity.com/project/scripting-de-nivel-intermedio

Entrenamiento 3

- Práctica con el pathway oficial de programador-junior de Unity.
- Desarrollo paso a paso y solución: https://learn.unity.com/learn/pathway/programador-junior

Entrenamiento 4

- Desarrolla un videojuego en Unity 3D.

- Desarrollo paso a paso y solución: https://learn.unity.com/project/john-lemon-s-haunted-jaunt-3d-para-principiantes/?tab=overview

Entrenamiento 5

- Desarrolla un videojuego 2D de plataformas en Unity.

- Desarrollo paso a paso y solución: https://www.youtube.com/playlist?list=PLNEAWvYbJJ9kZpaIg2RfzAc_KZixBgchT