**Comparación C++ y C#**

C++ y C# son dos lenguajes de programación ampliamente utilizados, pero tienen diferencias significativas en términos de diseño, uso y características. A continuación, se presentan algunas de las diferencias clave entre C++ y C#, junto con ejemplos prácticos para ilustrarlas.

**1. \*Paradigma de Programación\***

**C++:**

- Lenguaje de propósito general que soporta programación procedural, orientada a objetos y genérica.

- Permite un control detallado sobre el hardware y la gestión de memoria.

**C#:**

- Lenguaje de programación orientado a objetos desarrollado por Microsoft.

- Enfocado en la simplicidad y seguridad, con características que facilitan el desarrollo de aplicaciones de Windows y web.

**Ejemplo de Clase Básica**

**C++:**

#include <iostream>

using namespace std;

class Person {

public:

string name;

int age;

Person(string n, int a) {

name = n;

age = a;

}

void display() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

Person p("Alice", 30);

p.display();

return 0;

}

**C#:**

using System;

class Person {

public string Name { get; set; }

public int Age { get; set; }

public Person(string name, int age) {

Name = name;

Age = age;

}

public void Display() {

Console.WriteLine($"Name: {Name}, Age: {Age}");

}

}

class Program {

static void Main() {

Person p = new Person("Alice", 30);

p.Display();

}

}

**2. Gestión de Memoria**

**C++:**

- Utiliza gestión de memoria manual con `new` y `delete`.

#include <iostream>

using namespace std;

int main() {

int\* ptr = new int; // Asignación dinámica de memoria

\*ptr = 10;

cout << \*ptr << endl;

delete ptr; // Liberación de memoria

return 0;

}

**C#:**

- Utiliza recolección de basura (garbage collection) para la gestión automática de memoria.

using System;

class Program {

static void Main() {

int? num = new int(); // Asignación dinámica de memoria

num = 10;

Console.WriteLine(num);

// No es necesario liberar memoria explícitamente

}

}

**3. Compatibilidad y Plataforma**

**C++:**

- Es multiplataforma y puede ser compilado en una variedad de sistemas operativos.

- Amplio uso en sistemas embebidos, desarrollo de juegos y software de alto rendimiento.

**C#:**

- Principalmente utilizado en el ecosistema de Microsoft, aunque con .NET Core/.NET 5+ es multiplataforma.

- Ideal para aplicaciones de Windows, servicios web, y aplicaciones de empresa.

**4. Librerías y Marco de Trabajo**

**C++:**

- No tiene un marco de trabajo estandarizado, pero cuenta con la Standard Library (STL) y muchas otras bibliotecas externas.

**C#:**

- Utiliza el .NET Framework (o .NET Core/.NET 5+), que proporciona una vasta colección de bibliotecas para casi cualquier tarea.

**5. Seguridad y Manejo de Errores**

**C++:**

- Menos seguro en términos de acceso a memoria, ya que permite punteros y aritmética de punteros.

- Manejo de errores a través de excepciones, pero es más propenso a errores como desbordamientos de búfer y accesos inválidos a memoria.

#include <iostream>

#include <stdexcept>

using namespace std;

void riskyFunction() {

throw runtime\_error("An error occurred");

}

int main() {

try {

riskyFunction();

} catch (const runtime\_error& e) {

cout << e.what() << endl;

}

return 0;

}

**C#:**

- Más seguro en términos de acceso a memoria gracias a su manejo de punteros restringido y recolección de basura.

- Manejo de errores a través de excepciones, con una sintaxis más limpia y segura.

using System;

class Program {

static void RiskyFunction() {

throw new InvalidOperationException("An error occurred");

}

static void Main() {

try {

RiskyFunction();

} catch (InvalidOperationException ex) {

Console.WriteLine(ex.Message);

}

}

}

**6. Sintaxis y Facilidad de Uso**

**C++:**

- Más complejo y detallado, adecuado para desarrolladores que necesitan un control fino sobre el hardware y el rendimiento.

**C#:**

- Más simple y accesible, con características modernas que facilitan el desarrollo rápido de aplicaciones.

**Conclusión**

C++ y C# son lenguajes potentes pero orientados a diferentes tipos de proyectos y necesidades. C++ es ideal para aplicaciones donde el rendimiento y el control de hardware son cruciales, mientras que C# es excelente para el desarrollo rápido y seguro de aplicaciones en el ecosistema de Microsoft. Conociendo estas diferencias, puedes elegir el lenguaje que mejor se adapte a tus objetivos de desarrollo.

**C# y Unity**

El motor de juegos Unity utiliza principalmente el lenguaje de programación \*\*C#\*\* para el desarrollo de scripts y lógica del juego. C# es un lenguaje de programación potente y versátil que se integra perfectamente con el entorno de desarrollo de Unity, proporcionando una gran cantidad de funcionalidades y una sintaxis moderna que facilita el desarrollo de juegos.

**Razones para Utilizar C# en Unity**

1. Integración Nativa: Unity tiene soporte nativo para C#, lo que significa que todas las funcionalidades y herramientas de Unity están diseñadas para trabajar con C# de manera eficiente.

2. Facilidad de Uso: C# es conocido por su sintaxis clara y moderna, lo que facilita el aprendizaje y el desarrollo rápido de aplicaciones.

3. Documentación y Comunidad: Hay una amplia documentación y una gran comunidad de desarrolladores que utilizan C#, lo que facilita encontrar recursos, tutoriales y soporte.

4. Funcionalidades Avanzadas: C# en Unity permite el uso de características avanzadas como la programación asíncrona, genéricos, y eventos, lo que proporciona gran flexibilidad y poder en el desarrollo de juegos.

**Ejemplo de Script en C# en Unity**

Aquí tienes un ejemplo simple de un script en C# para Unity que mueve un objeto en respuesta a las entradas del usuario:

using UnityEngine;

public class PlayerController : MonoBehaviour

{

public float speed = 10.0f;

void Update()

{

float moveHorizontal = Input.GetAxis("Horizontal");

float moveVertical = Input.GetAxis("Vertical");

Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);

transform.Translate(movement \* speed \* Time.deltaTime);

}

}

**Explicación del Ejemplo**

1. \*\*Importar Librería\*\*: `using UnityEngine;` - Importa la librería principal de Unity.

2. \*\*Clase `PlayerController`\*\*: Define una clase que hereda de `MonoBehaviour`, lo que permite que el script se adjunte a un GameObject en Unity.

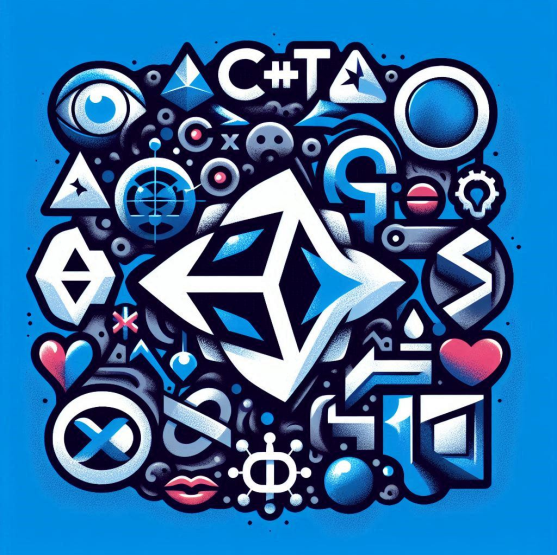
3. \*\*Variable `speed`\*\*: Define la velocidad del movimiento del jugador.

4. \*\*Método `Update()`\*\*: Se llama una vez por frame y se utiliza para manejar la entrada del usuario.

5. \*\*Lectura de Entrada\*\*: `Input.GetAxis("Horizontal")` y `Input.GetAxis("Vertical")` leen la entrada del teclado.

6. \*\*Movimiento\*\*: Se crea un vector de movimiento y se aplica al transform del objeto para moverlo en el espacio.

En resumen, Unity utiliza C# como su principal lenguaje de programación debido a su integración perfecta con el motor, su facilidad de uso y sus potentes características que permiten el desarrollo de juegos complejos y de alta calidad.

****

**Ejemplo de juego con Unity**

Vamos a crear un juego simple en Unity donde el jugador controla una esfera que se mueve por un plano y recoge objetos coleccionables. Al recoger todos los objetos, el juego muestra un mensaje de victoria.

**Paso 1: Preparar el Escenario**

1. \*\*Crear un Nuevo Proyecto en Unity\*\*.

2. \*\*Agregar un Plano\*\*:

- GameObject -> 3D Object -> Plane.

3. \*\*Agregar una Esfera\*\*:

- GameObject -> 3D Object -> Sphere.

- Posicionar la esfera sobre el plano. Poner un 0.5 en eje Y.

4. \*\*Agregar Objetos Coleccionables\*\*:

- GameObject -> 3D Object -> Cube.Poner en X 4 y X -4

- Duplicar y posicionar varios cubos sobre el plano.

5. \*\*Agregar una Cámara\*\*:

- La cámara principal debe posicionarse de manera que tenga una buena vista del plano y la esfera.

**Paso 2: Crear Scripts en C#**

En Assests-> Boton derecho-> Create-> C# Script

1. Script para Mover la Esfera

\*\*PlayerController.cs\*\*:

using UnityEngine;

public class PlayerController : MonoBehaviour

{

public float speed = 10.0f;

void Update()

{

float moveHorizontal = Input.GetAxis("Horizontal");

float moveVertical = Input.GetAxis("Vertical");

Vector3 movement = new Vector3(moveHorizontal, 0.0f, moveVertical);

transform.Translate(movement \* speed \* Time.deltaTime);

}

}

**Explicación:**

- `speed`: Variable pública para ajustar la velocidad de movimiento.

- `Update()`: Método que se llama una vez por frame.

- `Input.GetAxis("Horizontal")` y `Input.GetAxis("Vertical")`: Capturan la entrada del teclado.

- `Vector3 movement`: Crea un vector de movimiento basado en la entrada del usuario.

- `transform.Translate()`: Mueve la esfera.

**2. Script para Recoger Objetos**

\*\*Collectible.cs\*\*:

using UnityEngine;

public class Collectible : MonoBehaviour

{

void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Player"))

{

gameObject.SetActive(false);

GameManager.instance.CollectItem();

}

}

}

**Explicación:**

- `OnTriggerEnter(Collider other)`: Método que se llama cuando otro objeto entra en el collider del coleccionable.

- `other.CompareTag("Player")`: Comprueba si el objeto que ha colisionado tiene el tag "Player".

- `gameObject.SetActive(false)`: Desactiva el objeto coleccionable.

- `GameManager.instance.CollectItem()`: Llama al método `CollectItem()` del `GameManager`.

**3. Script para Gestionar el Juego**

\*\*GameManager.cs\*\*:

using UnityEngine;

using UnityEngine.UI;

public class GameManager : MonoBehaviour

{

public static GameManager instance;

private int totalCollectibles;

private int collectedCount;

public Text winText;

void Awake()

{

if (instance == null)

{

instance = this;

}

else

{

Destroy(gameObject);

}

}

void Start()

{

totalCollectibles = GameObject.FindGameObjectsWithTag("Collectible").Length;

collectedCount = 0;

winText.gameObject.SetActive(false);

}

public void CollectItem()

{

collectedCount++;

if (collectedCount >= totalCollectibles)

{

winText.gameObject.SetActive(true);

winText.text = "¡Has ganado!";

}

}

}

**Explicación:**

- `instance`: Instancia estática para implementar el patrón Singleton.

- `totalCollectibles`: Total de objetos coleccionables en el nivel.

- `collectedCount`: Contador de objetos coleccionados.

- `winText`: Referencia al texto de victoria en la UI.

- `Awake()`: Inicializa la instancia del GameManager.

- `Start()`: Inicializa los contadores y desactiva el mensaje de victoria.

- `CollectItem()`: Incrementa el contador de objetos coleccionados y muestra el mensaje de victoria si todos los objetos han sido recogidos.

**Paso 3: Configuración en el Editor de Unity**

***1. Asignar Scripts a los Objetos***

*Asignar `PlayerController` a la esfera:*

1. Selecciona la esfera en la jerarquía (el panel a la izquierda que muestra todos los objetos de la escena).

2. En el Inspector (el panel a la derecha que muestra las propiedades del objeto seleccionado), haz clic en el botón "Add Component".

3. Escribe `PlayerController` en la barra de búsqueda y selecciona el script `PlayerController` que creaste. Esto asignará el script a la esfera.

*Asignar `Collectible` a cada cubo coleccionable:*

1. Selecciona un cubo coleccionable en la jerarquía.

2. En el Inspector, haz clic en "Add Component".

3. Escribe `Collectible` en la barra de búsqueda y selecciona el script `Collectible`.

4. Repite este proceso para cada cubo coleccionable en la escena.

*Crear un objeto vacío y asignarle `GameManager`:*

1. En la jerarquía, haz clic derecho y selecciona `Create Empty`.

2. Nombra este objeto "GameManager".

3. Con el objeto "GameManager" seleccionado, en el Inspector, haz clic en "Add Component".

4. Escribe `GameManager` en la barra de búsqueda y selecciona el script `GameManager`.

***2. Configurar los Colliders***

*Asegurarse de que la esfera y los cubos coleccionables tengan colisionadores:*

1. Selecciona la esfera en la jerarquía.

2. En el Inspector, verifica que tenga un componente `Sphere Collider`. Si no, haz clic en "Add Component" y selecciona `Sphere Collider`.

3. Selecciona cada cubo coleccionable en la jerarquía.

4. En el Inspector, verifica que cada uno tenga un componente `Box Collider`. Si no, haz clic en "Add Component" y selecciona `Box Collider`.

*Habilitar la opción "Is Trigger" en los colisionadores de los cubos coleccionables:*

1. Selecciona un cubo coleccionable en la jerarquía.

2. En el Inspector, en el componente `Box Collider`, marca la casilla "Is Trigger".

3. Repite este proceso para cada cubo coleccionable en la escena.

**3. Configurar Tags**

*Asignar la tag "Player" a la esfera:*

1. Selecciona la esfera en la jerarquía.

2. En el Inspector, en la parte superior, verás un desplegable llamado "Tag".

3. Haz clic en el desplegable y selecciona "Player". Si no ves la opción "Player", selecciona "Add Tag...", haz clic en el símbolo `+` para añadir una nueva etiqueta, escribe "Player" y luego asígnala a la esfera.

*Asignar la tag "Collectible" a cada cubo coleccionable:*

1. Selecciona un cubo coleccionable en la jerarquía.

2. En el Inspector, en la parte superior, verás un desplegable llamado "Tag".

3. Haz clic en el desplegable y selecciona "Collectible". Si no ves la opción "Collectible", selecciona "Add Tag...", haz clic en el símbolo `+` para añadir una nueva etiqueta, escribe "Collectible" y luego asígnala a cada cubo coleccionable.

Asegúrate de que ambos objetos tienen Collider y Rigidbody (créalo con Add component si no está) correctamente configurados.

La esfera debe tener un Rigidbody con las opciones predeterminadas.

El cubo debe tener un Rigidbody configurado como Kinematic para detectar colisiones sin ser afectado por la física.

Usa en Assets -> botón derecho-> Create->Material para crear texturas con el color que quieras y luego arrástralos a la esfera o a los ladrillos o al plano.

***4. Configurar la UI***

*Crear un objeto de texto en la UI para el mensaje de victoria:*

1. En la jerarquía, haz clic derecho y selecciona `UI ->Legacy-> Text`.

2. Esto creará un Canvas (lienzo) y un objeto de texto dentro de él. El Canvas es un área de dibujo para todos los elementos UI.

3. Selecciona el objeto `Text` en la jerarquía.

4. En el Inspector, puedes modificar las propiedades del texto, como su contenido, fuente, tamaño, color, etc. Para este ejemplo, cambia el texto a "¡Has ganado!" y ajusta su tamaño y posición para que sea visible en la pantalla.

*Asignar el objeto de texto a la variable `winText` en el script `GameManager`:*

1. Selecciona el objeto "GameManager" en la jerarquía.

2. En el Inspector, verás el componente `GameManager` con una variable pública `winText`.

3. Arrastra el objeto `Text` desde la jerarquía hasta el campo `winText` en el Inspector. Esto vinculará el objeto de texto al script `GameManager`.

***Resumen***

- \*\*Asignar scripts\*\*: Añadir scripts a objetos a través del Inspector usando "Add Component".

- \*\*Configurar colliders\*\*: Asegurarse de que los objetos tienen colisionadores adecuados y marcar "Is Trigger" cuando sea necesario.

- \*\*Configurar tags\*\*: Añadir etiquetas personalizadas a objetos para identificarlos fácilmente.

- \*\*Configurar UI\*\*: Crear y configurar elementos UI (como texto) y vincularlos a scripts para mostrar mensajes y otra información.

Estos pasos permiten configurar y vincular todos los componentes necesarios en Unity para hacer funcionar el juego simple.

**Fichero Exe**

Generar fichero para un exe en Windows:

Ir a File->Build Settings-> Seleccionar el Sistema Operativo y hacer click en Build o Build and Run.

**Resultado Final**

Al ejecutar el juego, el jugador puede mover la esfera usando las teclas de flecha o WASD. Cuando la esfera colisiona con un cubo coleccionable, el cubo desaparece y el contador de objetos recogidos se incrementa. Una vez que se han recogido todos los cubos, aparece un mensaje de victoria en la pantalla.

Este ejemplo muestra cómo crear un juego simple en Unity utilizando C#, abordando conceptos clave como movimiento del jugador, detección de colisiones, y gestión del juego.