* **Ejemplo en Unity**

Crear una aplicación en Unity que simule el funcionamiento de un laberinto.

### **Paso 1: Instalación de Unity y Creación del Proyecto**

1. **Instalar Unity:**
   * Descarga e instala **Unity Hub** desde aquí.
   * Una vez instalado, abre Unity Hub y crea una nueva cuenta si no tienes una.
   * En Unity Hub, selecciona la pestaña **Installs** y elige la versión de Unity que deseas instalar (te recomiendo la última versión estable).
2. **Crear un Nuevo Proyecto:**
   * Abre Unity Hub y selecciona **New Project**.
   * Elige la plantilla **3D** (ya que trabajaremos con objetos 3D en el laberinto).
   * Asigna un nombre a tu proyecto (por ejemplo, Laberinto) y selecciona la ubicación donde deseas guardarlo.
   * Haz clic en **Create** para crear el proyecto.

### **Paso 2: Configuración del Escenario en Unity**

1. **Configurar la Escena:**
   * Una vez que el proyecto esté abierto, verás la ventana principal de Unity.
   * En la parte superior de la ventana, verás la pestaña **Hierarchy**, que es donde aparecerán todos los objetos de tu escena.
   * Verás que Unity ha creado automáticamente una **Main Camera** y una **Directional Light**. Estos son necesarios para visualizar la escena.
2. **Crear el Suelo:**
   * Haz clic derecho en la ventana **Hierarchy** y selecciona **3D Object > Plane**.
   * Esto creará un objeto de plano que actuará como el suelo del laberinto.
   * Asegúrate de que el objeto está centrado en el origen (0, 0, 0). Puedes comprobarlo seleccionando el objeto y verificando la posición en la pestaña **Inspector**.
3. **Añadir Material al Suelo (Opcional):**
   * Si deseas personalizar el aspecto del suelo, puedes crear un material personalizado.
   * Haz clic derecho en la carpeta **Assets** en la ventana **Project** y selecciona **Create > Material**.
   * Asigna un nombre al material (por ejemplo, SueloMaterial) y luego puedes cambiar el color en la sección **Albedo**.
   * Arrastra y suelta el material en el objeto plano en la escena.

### **Paso 3: Crear el Laberinto**

1. **Crear Paredes:**
   * Haz clic derecho en la ventana **Hierarchy** y selecciona **3D Object > Cube**.
   * Esto creará un cubo que representará una pared en el laberinto.
   * Ajusta la escala del cubo para que se asemeje a una pared. Por ejemplo, puedes cambiar la escala en el **Inspector** a (1, 2, 1) para que sea más alto que el jugador.
   * Duplica este cubo (Ctrl+D o Cmd+D en Mac) y colócalo para formar un laberinto simple.
2. **Organizar el Laberinto:**
   * Duplica las paredes y colócalas en la escena para formar un laberinto. Puedes hacerlo de manera manual o utilizar un script para generar las paredes automáticamente (más adelante veremos cómo hacerlo programáticamente).

### **Paso 4: Crear el Jugador**

1. **Añadir el Jugador:**
   * Haz clic derecho en la ventana **Hierarchy** y selecciona **3D Object > Sphere**.
   * Esto representará al personaje del jugador.
   * Ajusta la posición de la esfera para que esté por encima del plano (por ejemplo, posición (0, 0.5, 0)).
2. **Añadir un Rigidbody:**
   * Con la esfera seleccionada, ve al panel **Inspector** y haz clic en **Add Component**.
   * Busca **Rigidbody** y añádelo al jugador. Esto permitirá que el jugador se vea afectado por la física (gravedad, colisiones, etc.).
3. **Crear el Script del Jugador:**
   * Ve a la carpeta **Assets** en la ventana **Project** y haz clic derecho para crear una nueva carpeta llamada Scripts.
   * Dentro de la carpeta Scripts, haz clic derecho y selecciona **Create > C# Script**. Llámalo PlayerController.
   * Haz doble clic en el script para abrirlo en tu editor de código preferido (Unity usa Visual Studio por defecto).
4. **Escribir el Código del Jugador:**
   * A continuación, escribe el siguiente código en el script PlayerController:

using UnityEngine;

public class PlayerController : MonoBehaviour

{

public float speed = 5f;

private Rigidbody rb;

void Start()

{

rb = GetComponent<Rigidbody>();

}

void Update()

{

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

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

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

rb.AddForce(movement \* speed);

}

private void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Exit"))

{

Debug.Log("¡Has ganado!");

}

else if (other.CompareTag("Enemy"))

{

Debug.Log("¡Has perdido! Un enemigo te atrapó.");

}

else if (other.CompareTag("Collectible"))

{

Debug.Log("¡Recolectado!");

Destroy(other.gameObject);

}

}

}

* + Guarda el script y regresa a Unity. Asigna este script a la esfera (arrastrándolo al objeto en la jerarquía o seleccionando la esfera y usando **Add Component**).

### **Paso 5: Crear la Salida y Objetos Recolectables**

1. **Añadir la Salida:**
   * Crea otro cubo en la escena que represente la salida. Puedes cambiar su color a algo diferente (como verde).
   * Etiqueta este objeto como Exit (selecciona el cubo y en la parte superior del **Inspector**, verás la opción **Tag**. Crea una nueva etiqueta llamada Exit y asígnala al cubo).
2. **Añadir Objetos Recolectables:**
   * Crea varios objetos recolectables en el laberinto (por ejemplo, esferas más pequeñas).
   * Etiqueta estos objetos como Collectible.

### **Paso 6: Crear los Enemigos**

1. **Añadir Enemigos:**
   * Crea objetos que representen a los enemigos (puedes usar cubos o esferas de un color diferente).
   * Asigna un Rigidbody a los enemigos si deseas que también sigan la física.
2. **Crear el Script de Enemigos:**
   * Crea un nuevo script llamado EnemyController en la carpeta Scripts.
   * Añade el siguiente código:

using UnityEngine;

public class EnemyController : MonoBehaviour

{

public Transform[] waypoints;

public float speed = 2f;

private int currentWaypointIndex = 0;

void Update()

{

if (waypoints.Length == 0) return;

Transform targetWaypoint = waypoints[currentWaypointIndex];

Vector3 direction = (targetWaypoint.position - transform.position).normalized;

transform.position = Vector3.MoveTowards(transform.position, targetWaypoint.position, speed \* Time.deltaTime);

if (Vector3.Distance(transform.position, targetWaypoint.position) < 0.1f)

{

currentWaypointIndex = (currentWaypointIndex + 1) % waypoints.Length;

}

}

}

* + Asigna este script a los enemigos.
  + En la ventana **Inspector**, verás un campo para Waypoints. Esto te permitirá definir los puntos por donde el enemigo se moverá. Crea objetos vacíos en la escena para usar como waypoints y asígnalos al enemigo.

### **Paso 7: Prueba y Mejora el Juego**

1. **Prueba el Juego:**
   * Haz clic en el botón **Play** en la parte superior de Unity para iniciar el juego.
   * Asegúrate de que el jugador puede moverse, recolectar objetos, evitar enemigos y llegar a la salida.
2. **Añadir Mejoras:**
   * Añade más niveles.
   * Implementa un sistema de puntuación.
   * Añade efectos visuales y de sonido para mejorar la experiencia.

**AJUSTES**

**¿Cómo genero las paredes automáticamente?**

Para generar las paredes del laberinto de manera automática en Unity, puedes utilizar un **script de generación procedural**. Este enfoque te permitirá crear un laberinto con paredes dispuestas aleatoriamente o siguiendo algún algoritmo específico.

Mediante un script podrás crear las paredes de forma automática, en lugar de colocarlas manualmente.

### **Paso 1: Crear el Script de Generación del Laberinto**

1. **Crear el Script:**
   * Ve a la carpeta **Assets** en la ventana **Project** de Unity.
   * Haz clic derecho y selecciona **Create > C# Script**.
   * Nombra el script como MazeGenerator.
   * Haz doble clic en el script para abrirlo en el editor de código (como Visual Studio).

**Escribir el Código del Script:** A continuación, te proporciono un script básico para generar paredes en un patrón de laberinto aleatorio:  
  
using UnityEngine;

public class MazeGenerator : MonoBehaviour

{

public GameObject wallPrefab; // Prefab de pared

public int width = 10; // Ancho del laberinto

public int height = 10; // Altura del laberinto

public float cellSize = 1f; // Tamaño de cada celda del laberinto

void Start()

{

GenerateMaze();

}

void GenerateMaze()

{

// Generar las paredes del borde del laberinto

for (int x = 0; x <= width; x++)

{

for (int z = 0; z <= height; z++)

{

// Generar paredes en los bordes

if (x == 0 || z == 0 || x == width || z == height)

{

InstantiateWall(x, z);

}

else

{

// Generar paredes internas de manera aleatoria

if (Random.value > 0.7f) // Probabilidad del 30% de generar una pared

{

InstantiateWall(x, z);

}

}

}

}

}

void InstantiateWall(int x, int z)

{

Vector3 position = new Vector3(x \* cellSize, 1f, z \* cellSize);

Instantiate(wallPrefab, position, Quaternion.identity);

}

}

### **Explicación del Código:**

* **wallPrefab:** Este es el objeto que se utilizará como pared. Puedes asignar cualquier Prefab de un cubo o cualquier otro objeto 3D que desees utilizar como pared.
* **width y height:** Estos valores determinan el tamaño del laberinto. Puedes ajustarlos para hacer el laberinto más grande o más pequeño.
* **cellSize:** El tamaño de cada celda del laberinto. Esto controla la distancia entre las paredes.
* **GenerateMaze():** Este método genera el laberinto. Primero, genera paredes en los bordes del laberinto y luego crea paredes internas de manera aleatoria con una probabilidad del 30%.
* **InstantiateWall():** Este método instancia las paredes en la posición especificada.

### **Paso 2: Configurar el Script en Unity**

1. **Crear un Prefab para la Pared:**
   * En la ventana **Hierarchy**, haz clic derecho y selecciona **3D Object > Cube**. Este cubo representará una pared en el laberinto.
   * Ajusta la escala del cubo para que parezca una pared (por ejemplo, puedes establecer la escala en X, Y y Z como (1, 2, 1)).
   * Arrastra el cubo desde la **Hierarchy** a la carpeta **Assets** en la ventana **Project**. Esto convertirá el cubo en un Prefab.
   * Elimina el cubo original de la **Hierarchy**.
2. **Asignar el Prefab al Script:**
   * Crea un **Empty GameObject** en la **Hierarchy** y nómbralo MazeGenerator.
   * Asigna el script MazeGenerator al objeto MazeGenerator.
   * En el **Inspector**, verás un campo llamado Wall Prefab. Arrastra y suelta el Prefab del cubo en este campo.
3. **Ajustar Parámetros:**
   * Puedes ajustar el ancho (width), altura (height) y el tamaño de celda (cellSize) desde el **Inspector** para personalizar el laberinto.

### **Paso 3: Probar la Generación del Laberinto**

1. **Iniciar el Juego:**
   * Haz clic en el botón **Play** en la parte superior de Unity para ejecutar la escena.
   * Verás que el script genera un laberinto con paredes en los bordes y algunas paredes internas de manera aleatoria.
2. **Ajustar la Probabilidad de las Paredes Internas:**

Si deseas más o menos paredes internas, puedes ajustar la línea:  
csharp  
Copiar código  
if (Random.value > 0.7f)

* + Cambiando el valor 0.7f por algo más bajo (por ejemplo, 0.5f) generará más paredes, mientras que un valor más alto generará menos paredes.

### **Paso 4: Expandir la Generación del Laberinto (Opcional)**

Si deseas un laberinto más estructurado, puedes implementar algoritmos de generación de laberintos, como el **Algoritmo de Backtracking Recursivo**, **Algoritmo de Prim**, o **Algoritmo de Kruskal**. Estos algoritmos son más complejos, pero producen laberintos con caminos claros y garantizados desde el inicio hasta la salida.

Un ejemplo simple utilizando el Algoritmo de Backtracking Recursivo requeriría crear una cuadrícula de celdas y luego "cavar" pasillos entre ellas, asegurándote de que siempre haya un camino disponible.

**¿Cómo crear Waypoints?**

Para crear **Waypoints** en Unity, de manera que puedas definir puntos de referencia por los cuales el enemigo se moverá. Utilizaremos **objetos vacíos** en la escena para representar estos Waypoints y los asignaremos al enemigo.

Paso 1: Crear los Waypoints en la Escena

1. **Abrir Unity y tu Escena:**
   * Asegúrate de que tu proyecto esté abierto en Unity y que estés trabajando en la escena donde quieres implementar los Waypoints.
2. **Crear Objetos Vacíos para los Waypoints:**
   * En la ventana **Hierarchy**, haz clic derecho y selecciona **Create Empty**. Esto creará un **GameObject vacío** en la escena. Este objeto vacío servirá como un punto de referencia (Waypoint).
   * Nombra este objeto como Waypoint1.
   * Coloca este objeto vacío en una posición en la escena donde desees que el enemigo se mueva. Puedes moverlo utilizando las herramientas de **Mover** (el ícono con flechas) o ajustando las coordenadas en el **Inspector**.
   * Repite este proceso para crear más Waypoints. Por ejemplo, crea Waypoint2, Waypoint3, etc., y colócalos en diferentes posiciones en la escena.

Paso 2: Crear un Grupo de Waypoints

1. **Crear un Objeto Vacío para Agrupar los Waypoints:**
   * En la ventana **Hierarchy**, haz clic derecho y selecciona **Create Empty** para crear otro objeto vacío. Este objeto actuará como un contenedor para todos tus Waypoints.
   * Nombra este objeto como WaypointsGroup.
   * En la ventana **Hierarchy**, selecciona todos los Waypoints (Waypoint1, Waypoint2, etc.) y arrástralos como hijos del objeto WaypointsGroup. Esto te permitirá organizar mejor los Waypoints.

Paso 3: Asignar los Waypoints al Enemigo

1. **Crear el Script para el Movimiento del Enemigo:**
   * Ya deberías tener el script EnemyController.cs que maneja el movimiento del enemigo. Vamos a actualizarlo para que funcione con los Waypoints.

**Modificar el Script EnemyController.cs:** Abre el script EnemyController.cs y ajusta el código para que el enemigo se mueva entre los Waypoints que acabas de crear.  
  
using UnityEngine;

public class EnemyController : MonoBehaviour

{

public Transform[] waypoints; // Array de Waypoints

public float speed = 2f; // Velocidad de movimiento del enemigo

private int currentWaypointIndex = 0; // Índice del Waypoint actual

void Update()

{

// Si no hay waypoints, no hacer nada

if (waypoints.Length == 0) return;

// Moverse hacia el Waypoint actual

Transform targetWaypoint = waypoints[currentWaypointIndex];

Vector3 direction = (targetWaypoint.position - transform.position).normalized;

transform.position = Vector3.MoveTowards(transform.position, targetWaypoint.position, speed \* Time.deltaTime);

// Si el enemigo ha alcanzado el Waypoint, pasar al siguiente

if (Vector3.Distance(transform.position, targetWaypoint.position) < 0.1f)

{

currentWaypointIndex = (currentWaypointIndex + 1) % waypoints.Length; // Pasar al siguiente Waypoint

}

}

}

1. **Asignar los Waypoints al Script en Unity:**
   * Selecciona al enemigo en la **Hierarchy**.
   * En el **Inspector**, verás el componente EnemyController que has asignado anteriormente.
   * Verás un campo llamado Waypoints que es un array de Transform.
   * Haz clic en el ícono de **flecha pequeña** al lado de Waypoints, y verás una lista con tamaño 0.
   * Ajusta el tamaño del array para que coincida con la cantidad de Waypoints que has creado (por ejemplo, si creaste 3 Waypoints, ajusta el tamaño a 3).
   * Arrastra cada Waypoint desde la **Hierarchy** a los elementos del array Waypoints en el **Inspector** (por ejemplo, arrastra Waypoint1 al primer slot, Waypoint2 al segundo, y así sucesivamente).

Paso 4: Probar la Configuración

1. **Iniciar el Juego:**
   * Haz clic en el botón **Play** para iniciar el juego.
   * Verifica que el enemigo se mueve entre los Waypoints en el orden que has definido.

Paso 5: Ajustar el Movimiento y Waypoints (Opcional)

1. **Ajustar la Velocidad del Enemigo:**
   * Puedes ajustar la velocidad del enemigo cambiando el valor de la variable speed en el **Inspector** o directamente en el script.
2. **Modificar la Posición de los Waypoints:**
   * Si el enemigo no sigue la ruta que deseas, simplemente selecciona los Waypoints en la **Hierarchy** y muévelos a nuevas posiciones en la escena.

**¿Cómo cambiar la vista de la cámara?**

Para cambiar la vista de la cámara y ver la escena completa del juego desde arriba en Unity, puedes ajustar la posición y rotación de la cámara principal. Aquí te muestro cómo hacerlo paso a paso:

Paso 1: Ajustar la Cámara Principal en la Escena

1. **Selecciona la Cámara Principal:**
   * En la ventana **Hierarchy**, busca el objeto llamado **Main Camera** y selecciónalo.
   * Si no puedes encontrarla fácilmente, puedes hacer clic en el icono de búsqueda en la ventana **Hierarchy** y escribir "Main Camera".
2. **Cambiar la Posición de la Cámara:**
   * Con la cámara seleccionada, ve a la ventana **Inspector**.
   * Cambia las coordenadas de la posición de la cámara (el componente Transform) para que esté ubicada **arriba** del laberinto, mirando hacia abajo.
   * Por ejemplo, si el laberinto está centrado en (0, 0, 0), podrías ajustar la posición de la cámara a algo como:
     + **Position X:** 0
     + **Position Y:** 20 (esto coloca la cámara a una altura de 20 unidades sobre el laberinto)
     + **Position Z:** 0
3. **Rotar la Cámara para Mirar hacia Abajo:**
   * Para que la cámara mire hacia abajo, debes rotarla en el eje X.
   * En el componente Transform de la cámara en el **Inspector**, ajusta la rotación a algo como:
     + **Rotation X:** 90
     + **Rotation Y:** 0
     + **Rotation Z:** 0
4. **Ajustar el Campo de Visión (Opcional):**
   * Si no puedes ver todo el laberinto en la vista, puedes ajustar el campo de visión (FOV) de la cámara.
   * En el **Inspector**, busca el componente **Camera** (que está justo debajo del componente Transform).
   * Cambia el valor de **Field of View (FOV)**. Reducir el FOV (por ejemplo, a 30) hará que la cámara tenga un ángulo más estrecho y "vea" menos espacio, mientras que aumentarlo (por ejemplo, a 90 o más) ampliará el campo de visión.

Paso 2: Probar la Nueva Vista de la Cámara

1. **Iniciar el Juego:**
   * Haz clic en el botón **Play** en la barra superior para iniciar el juego.
   * Ahora deberías poder ver toda la escena desde arriba, permitiéndote seguir los movimientos del personaje, los enemigos y otros objetos en el laberinto sin que las paredes interfieran con la vista.

Paso 3: Ajustar según Necesidad

1. **Mover la Cámara o Cambiar la Rotación:**
   * Si no ves todo el laberinto, ajusta la posición Y (altura) de la cámara para que esté más alta o cambia la rotación para tener un ángulo diferente.
   * También puedes mover la cámara en los ejes X y Z si el laberinto no está centrado.
2. **Alternar entre Cámaras (Opcional):**
   * Si quieres cambiar entre diferentes vistas, puedes crear varias cámaras y habilitarlas/deshabilitarlas en diferentes momentos usando scripts.

**Pero….y si Exit no funciona, y si los colisionadores tampoco…**

Para asegurarte de que el personaje pueda detectar cuando colisiona con el cubo de salida (Exit) en Unity, debes asegurarte de que todo esté configurado correctamente, desde el componente Collider hasta el código de detección de colisión en el script. Vamos a revisar paso a paso cómo hacerlo correctamente.

Paso 1: Configurar el Objeto de Salida (Exit)

1. **Selecciona el Objeto de Salida:**
   * En la ventana **Hierarchy**, selecciona el objeto que representa la salida, que llamaremos "Exit".
2. **Agregar un Collider:**
   * Asegúrate de que el objeto de salida tenga un componente Collider (como un BoxCollider o SphereCollider). Si no lo tiene, puedes añadirlo haciendo clic en el botón **Add Component** en la ventana **Inspector** y buscando "Box Collider" o "Sphere Collider".
   * **Importante:** Marca la opción **Is Trigger** en el Collider. Esto permite que el objeto detecte las colisiones sin impedir el movimiento del personaje (es decir, se tratará como una "zona de activación" en lugar de una barrera física).
3. **Agregar la Etiqueta (Tag):**
   * Asegúrate de que el objeto de salida tenga la etiqueta "Exit".
   * Para añadir la etiqueta, selecciona el objeto, ve a la parte superior de la ventana **Inspector** donde dice **Tag**, haz clic en el desplegable y selecciona **Add Tag** si no está ya creada.
   * Añade una nueva etiqueta llamada "Exit" y asígnala al objeto.

Paso 2: Revisar el Script del Jugador para Detectar Colisiones

Ahora revisa el script PlayerController.cs para asegurarte de que está configurado correctamente para detectar colisiones con la salida.

Aquí está el código que deberías tener para detectar la colisión con el objeto etiquetado como "Exit":

using UnityEngine;

public class PlayerController : MonoBehaviour

{

public float speed = 5f;

private Rigidbody rb;

void Start()

{

rb = GetComponent<Rigidbody>();

}

void Update()

{

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

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

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

rb.AddForce(movement \* speed);

}

private void OnTriggerEnter(Collider other)

{

// Detecta si el jugador colisiona con la salida

if (other.CompareTag("Exit"))

{

Debug.Log("¡Has ganado!");

// Aquí puedes detener el juego, mostrar una pantalla de victoria o cambiar de escena

}

else if (other.CompareTag("Enemy"))

{

Debug.Log("¡Has perdido! Un enemigo te atrapó.");

// Aquí puedes detener el juego o reiniciarlo

}

else if (other.CompareTag("Collectible"))

{

Debug.Log("¡Recolectado!");

// Incrementar la puntuación

Destroy(other.gameObject); // Eliminar el objeto recolectable

}

}

}

Paso 3: Verifica la Configuración del Personaje

1. **Verifica el Rigidbody del Personaje:**
   * El personaje debe tener un componente Rigidbody para que Unity pueda gestionar las físicas del movimiento.
   * El Rigidbody debe tener la opción **Is Kinematic** **desactivada** para que las colisiones puedan detectarse correctamente.
2. **Verifica el Collider del Personaje:**
   * Asegúrate de que el personaje tenga un Collider (como un SphereCollider o BoxCollider). Este Collider se necesita para detectar colisiones con otros objetos.

Paso 4: Probar el Juego

1. **Ejecuta el Juego:**
   * Haz clic en el botón **Play** en la barra superior para ejecutar el juego y mover el personaje hacia la salida.
   * Observa en la consola de Unity para asegurarte de que el mensaje "¡Has ganado!" se imprima cuando el personaje cruce el objeto de salida.
2. **Verifica las Colisiones:**
   * Si no ocurre nada, asegúrate de que tanto el personaje como el objeto de salida tienen los Colliders configurados correctamente y de que el objeto de salida está etiquetado como "Exit".

**¿Cómo crear las etiquetas?**

El error "Exit is not defined" generalmente ocurre cuando intentas comparar la etiqueta de un objeto en Unity, pero esa etiqueta no existe o no ha sido asignada correctamente. Vamos a revisar paso a paso cómo asegurarte de que la etiqueta "Exit" está definida y correctamente asignada al objeto de salida.

Paso 1: Crear la Etiqueta "Exit"

1. **Crear la Etiqueta:**
   * Selecciona el objeto que deseas usar como salida en la ventana **Hierarchy**.
   * Ve a la ventana **Inspector**.
   * En la parte superior del **Inspector**, verás un campo llamado **Tag**. Haz clic en el desplegable junto a **Tag** y selecciona **Add Tag**.
2. **Añadir la Etiqueta:**
   * En la nueva ventana que se abre, verás una lista de etiquetas disponibles y un botón que dice **+**. Haz clic en el **+** para añadir una nueva etiqueta.
   * En el campo que aparece, escribe **Exit** (con mayúscula inicial) y guarda el cambio.
3. **Asignar la Etiqueta al Objeto:**
   * Vuelve a seleccionar el objeto de salida en la ventana **Hierarchy**.
   * En la ventana **Inspector**, en el campo **Tag**, selecciona ahora la etiqueta **Exit** que acabas de crear.

Paso 2: Verificar el Código de Comparación de la Etiqueta

En tu script, asegúrate de que estás utilizando el método CompareTag() correctamente, y que el nombre de la etiqueta coincide exactamente con la que has creado ("Exit"). Aquí está el código para revisar:

private void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Exit"))

{

Debug.Log("¡Has ganado!");

// Aquí puedes detener el juego o cambiar de escena

}

else if (other.CompareTag("Enemy"))

{

Debug.Log("¡Has perdido! Un enemigo te atrapó.");

// Aquí puedes detener el juego o reiniciarlo

}

else if (other.CompareTag("Collectible"))

{

Debug.Log("¡Recolectado!");

// Incrementar la puntuación

Destroy(other.gameObject); // Eliminar el objeto recolectable

}

}

Asegúrate de que:

* El texto "Exit" en el script es **exactamente** igual a la etiqueta que has creado (respetando mayúsculas y minúsculas).

Paso 3: Comprobar los Colliders

Además de las etiquetas, asegúrate de que ambos objetos (el personaje y la salida) tienen Colliders:

* El Collider del objeto de salida debe estar marcado como **Is Trigger**.
* El personaje debe tener un Collider y un Rigidbody con **Is Kinematic** desactivado.

Paso 4: Probar Nuevamente

1. Guarda todos los cambios y ejecuta el juego.
2. Verifica si el mensaje "¡Has ganado!" aparece en la consola de Unity cuando el personaje colisiona con el objeto de salida.

Si el Problema Persiste

Si el problema persiste y el error aún aparece, prueba los siguientes pasos:

* Asegúrate de que el script está asignado correctamente al objeto del jugador.
* Verifica que la etiqueta esté correctamente asignada y no esté escrita de forma incorrecta en el código.

Si realizas todos estos pasos, Unity debería reconocer la etiqueta "Exit" y permitir que el evento de colisión funcione correctamente.

**¿Es necesaria una etiqueta en el jugador principal?**

No es obligatorio que el personaje tenga una etiqueta específica para detectar la colisión con la salida ("Exit"). Sin embargo, si quieres que el personaje interactúe de alguna manera especial con otros objetos o scripts, podrías asignarle una etiqueta como "Player". Esto es útil en caso de que quieras referirte al personaje de forma específica en el código, o si necesitas manejar diferentes tipos de colisiones.

Cuándo usar etiquetas en el personaje:

1. **Identificación del personaje:**
   * Si tienes varios personajes en la escena o necesitas distinguir el personaje del jugador de otros objetos en el juego, podrías usar la etiqueta "Player".
2. **Uso en scripts:**
   * Si más adelante planeas agregar funcionalidad adicional, como detectores de enemigos, zonas seguras, o si tienes varios jugadores en la escena, la etiqueta "Player" puede ser útil.

Cómo añadir una etiqueta "Player" al personaje:

1. Selecciona el objeto que representa al personaje en la ventana **Hierarchy**.
2. Ve al **Inspector**.
3. En la parte superior del **Inspector**, en el campo **Tag**, selecciona **Add Tag**.
4. Haz clic en el botón **+** para crear una nueva etiqueta.
5. Escribe "Player" (con la "P" mayúscula) en el campo y guarda el cambio.
6. Luego, selecciona nuevamente el objeto del personaje y asígnale la etiqueta **Player**.

Ejemplo de uso de la etiqueta "Player" en un script:

Si en el futuro necesitas referirte al personaje por su etiqueta, puedes hacerlo de la siguiente manera:

private void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Player"))

{

// Código específico para cuando el objeto etiquetado como "Player" entra en la colisión

Debug.Log("El jugador ha entrado en la zona.");

}

}

**¿Sigue sin funcionar correctamente?**

Si el personaje no está activando el contacto con el objeto etiquetado como "Exit" y todo está configurado correctamente, es posible que haya algunos detalles que revisar en la configuración de los colisionadores o en el script.

Aquí tienes algunos pasos para solucionar el problema:

1. Verificar los colisionadores:

* Asegúrate de que tanto el personaje como el objeto de la salida ("Exit") tengan componentes de colisión adecuados:
  + **Personaje:** Debe tener un Collider (por ejemplo, SphereCollider si es una esfera) y un Rigidbody. El Rigidbody no debería estar marcado como Kinematic, ya que eso podría evitar colisiones.
  + **Exit:** Debe tener un Collider (por ejemplo, BoxCollider) y debe estar marcado como Is Trigger en el inspector. Esto permite que el objeto actúe como un desencadenador de eventos en lugar de ser una barrera física.

2. Revisar el script:

Asegúrate de que el script esté correctamente configurado para detectar la colisión con el objeto "Exit". Aquí tienes un ejemplo básico para revisar:

using UnityEngine;

public class PlayerController : MonoBehaviour

{

public float speed = 5f;

private Rigidbody rb;

void Start()

{

rb = GetComponent<Rigidbody>();

}

void Update()

{

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

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

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

rb.AddForce(movement \* speed);

}

private void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Exit"))

{

Debug.Log("¡Has ganado!");

// Aquí puedes detener el juego o mostrar un mensaje de victoria

}

}

}

3. Etiqueta del objeto "Exit":

* Verifica que el objeto de la salida tenga la etiqueta **exactamente** como "Exit". Recuerda que las etiquetas en Unity son sensibles a mayúsculas y minúsculas. Asegúrate de que está escrito correctamente y que el objeto de salida tiene asignada esta etiqueta.

4. Comprobación visual de colisiones:

Para verificar que las colisiones realmente están ocurriendo, puedes habilitar la opción de **Gizmos** en la ventana de juego para visualizar los colisionadores durante el juego. Esto te permitirá ver si el personaje y la salida están realmente interactuando.

5. Otros posibles problemas:

* **Capas de colisión:** Revisa las capas en las que están colocados el personaje y la salida. Asegúrate de que las capas no estén configuradas para ignorar colisiones entre sí en el **Collision Matrix** (Edit > Project Settings > Physics > Layer Collision Matrix).
* **Rigidbody:** Asegúrate de que el personaje tenga un Rigidbody asignado, ya que es necesario para detectar colisiones con Trigger en Unity.

6. Probar con otros objetos:

* Para comprobar que el problema no está relacionado con el objeto "Exit", intenta crear otro objeto simple con un BoxCollider y la etiqueta "Exit". Si este nuevo objeto funciona correctamente, es posible que haya algún problema específico con el objeto de salida original.

Si después de realizar estos pasos sigue sin funcionar, prueba con este código más básico que imprime un mensaje cuando cualquier colisionador entra en contacto con el personaje. Esto ayudará a depurar el problema:

private void OnTriggerEnter(Collider other)

{

Debug.Log("Colisionaste con: " + other.name);

}

Esto imprimirá el nombre de cualquier objeto que toque al personaje en la consola de Unity. Así podrás ver si el personaje está detectando la colisión con cualquier objeto.

**¿Cómo puedo reducir el grosor de las paredes del laberinto?**

Para hacer más delgadas las paredes del laberinto en Unity, debes ajustar las escalas de los cubos (que estás utilizando para crear las paredes). Aquí te explico cómo puedes hacerlo:

Paso 1: Selecciona los cubos que forman las paredes

1. En la **Jerarquía** de Unity, selecciona los cubos que forman las paredes del laberinto. Si tienes muchos cubos y deseas cambiar todos al mismo tiempo, selecciona uno y mantén presionada la tecla Shift o Ctrl (en Windows) o Cmd (en macOS) para seleccionar múltiples objetos.

Paso 2: Ajusta la escala de los cubos

1. En el **Inspector** de Unity (normalmente en la parte derecha de la pantalla), verás las propiedades del objeto seleccionado, entre ellas las propiedades de **Transform**.
2. Ajusta la **escala** en el eje que deseas hacer más delgado. Generalmente, querrás ajustar el valor en el eje **X** o **Z** (dependiendo de la orientación de tus paredes). Si actualmente las paredes tienen una escala de (1, 1, 1), puedes reducir la escala en el eje correspondiente para hacerlas más delgadas. Por ejemplo, si reduces la escala del eje **X** a 0.1, la pared se volverá más delgada en ese eje.  
   Ejemplo:
   * Si tus paredes están orientadas en el eje X:
     + Cambia la **Escala X** a un valor más bajo, como 0.1.
   * Si están orientadas en el eje Z:
     + Cambia la **Escala Z** a un valor más bajo, como 0.1.

Paso 3: Asegúrate de que el colisionador también se ajuste

Al cambiar la escala del cubo, el **collider** del objeto debería ajustarse automáticamente a la nueva forma, ya que los colliders predeterminados (como el BoxCollider) escalan junto con el objeto.

Si por alguna razón el collider no se ajusta correctamente, puedes editarlo manualmente:

1. En el **Inspector**, encuentra el componente BoxCollider o el tipo de collider que estés usando.
2. Haz clic en el botón **Edit Collider** para ajustar el tamaño y la forma del collider manualmente si es necesario.

Paso 4: Verifica en la vista de juego

1. Vuelve a la **Vista de Juego** y asegúrate de que las paredes se vean como esperas. Puedes hacer varios ajustes en la escala para obtener el grosor de las paredes que prefieras.

Paso 5: Automatización (opcional)

Si estás generando las paredes del laberinto automáticamente a través de un script, puedes ajustar la escala directamente en el script. Por ejemplo:

// Si estás generando un cubo para la pared

GameObject wall = Instantiate(wallPrefab, position, Quaternion.identity);

wall.transform.localScale = new Vector3(0.1f, 1f, 1f); // Esto hace la pared más delgada en el eje X

Este código ajusta la escala de cada cubo generado para que sea más delgado en el eje X.

**Pero….¿Por qué se caen los objetos?**

Pues por la fuerza de la gravedad. Revisa en el inspector de objetos sus características en la opción Rigidbody y allí deshabilita la opción “Use gravity”. Evitarás que los objetos se caigan.

**¿Y si quiero un marcador, puntuación y demás?**

Para implementar un marcador en Unity que se incremente cuando el personaje recoja un objeto "collectible" y se reduzca cuando el personaje sea tocado por un enemigo, y para terminar el juego al llegar a la salida, sigue estos pasos:

Paso 1: Crear el Marcador

1. **Añadir un objeto UI para mostrar el marcador:**
   * En el menú de Unity, ve a GameObject -> UI -> Text para crear un nuevo objeto de texto en la interfaz de usuario.
   * Cambia el texto inicial a "Score: 0" en el Inspector del objeto Text.
   * Ajusta la posición y el tamaño del texto según sea necesario para que sea visible en la pantalla.

**Añadir una variable para el marcador en el script del jugador:**Abre el script PlayerController.cs y añade una variable para el marcador y un método para actualizarlo. Aquí tienes un ejemplo de cómo hacerlo:

using UnityEngine;

using UnityEngine.UI; // Necesario para trabajar con UI

public class PlayerController : MonoBehaviour

{

public float speed = 5f;

public Text scoreText; // Referencia al texto del marcador

private Rigidbody rb;

private int score = 0; // Variable para el marcador

void Start()

{

rb = GetComponent<Rigidbody>();

UpdateScoreText(); // Inicializar el texto del marcador

}

void Update()

{

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

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

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

rb.AddForce(movement \* speed);

}

private void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Exit"))

{

Debug.Log("¡Has ganado!");

// Mostrar mensaje de victoria o finalizar el juego

Application.Quit();

}

else if (other.CompareTag("Enemy"))

{

Debug.Log("¡Has perdido! Un enemigo te atrapó.");

// Reducir el marcador en 5 puntos y mostrar mensaje de derrota

score -= 5;

UpdateScoreText();

}

else if (other.CompareTag("Collectible"))

{

Debug.Log("¡Recolectado!");

// Incrementar el marcador en 10 puntos y destruir el objeto recolectable

score += 10;

UpdateScoreText();

Destroy(other.gameObject);

}

}

// Método para actualizar el texto del marcador

void UpdateScoreText()

{

scoreText.text = "Score: " + score;

}

}

**Notas:**

* + Asegúrate de arrastrar el objeto de texto de la UI desde la jerarquía al campo scoreText en el Inspector del script PlayerController. **¿Cómo?** Esta es una guía detallada paso a paso para asegurarte de que el objeto de texto de la UI se arrastre correctamente al campo scoreText en el Inspector del script PlayerController:

### **Paso a Paso: Asignar el Objeto de Texto a la Variable scoreText**

1. **Crea el Objeto de Texto:**
   * En Unity, asegúrate de tener un objeto de texto en tu escena. Si aún no lo has creado, ve al menú superior y selecciona GameObject -> UI -> Text para añadir un nuevo objeto de texto a tu escena.
   * Se creará un objeto de texto en la jerarquía de tu escena bajo un objeto llamado Canvas. Por defecto, se le puede asignar un nombre como ScoreText.
2. **Selecciona el Objeto de Texto:**
   * En la jerarquía de la escena (en la parte izquierda del editor de Unity), busca el objeto de texto que has creado (debería estar bajo Canvas).
   * Haz clic en el objeto de texto para seleccionarlo. Deberías ver el componente Text en el panel Inspector (en la parte derecha del editor).
3. **Abre el Inspector del Script PlayerController:**
   * En la jerarquía de la escena, selecciona el objeto que tiene el script PlayerController adjunto (generalmente el objeto que representa al jugador).
   * En el panel Inspector, deberías ver el componente PlayerController con sus variables públicas expuestas, incluido el campo scoreText.
4. **Arrastra el Objeto de Texto al Campo scoreText:**
   * Con el Inspector del objeto que tiene el script PlayerController abierto, localiza el campo scoreText en el componente PlayerController.
   * Haz clic y mantén presionado el objeto de texto en la jerarquía (el objeto ScoreText).
   * Arrastra el objeto de texto desde la jerarquía hasta el campo scoreText en el Inspector del script PlayerController.
   * Suelta el objeto de texto en el campo scoreText. Ahora debería estar asignado correctamente.
5. **Verifica la Asignación:**
   * Asegúrate de que el campo scoreText en el Inspector ahora muestra el nombre del objeto de texto que arrastraste.
   * Esto indica que la referencia al objeto de texto ha sido correctamente asignada al campo scoreText en el script.

### **Resumen Visual:**

1. **En la jerarquía de la escena:**
   * Selecciona el objeto de texto (por ejemplo, ScoreText).
2. **En el Inspector del script PlayerController:**
   * Encuentra el campo scoreText.
3. **Arrastra y Suelta:**
   * Arrastra el objeto de texto desde la jerarquía y suéltalo en el campo scoreText en el Inspector.

### **Prueba el Marcador:**

* **Inicia el Juego:** Después de realizar esta asignación, ejecuta el juego en Unity.
* **Verifica el Texto:** Interactúa con los objetos en el juego para asegurarte de que el marcador se actualice correctamente en la interfaz de usuario.

Paso 2: Configurar el Enemigo

Si aún no lo has hecho, asegúrate de que los enemigos tienen colisionadores y que están configurados con la etiqueta "Enemy". Esto es necesario para que el método OnTriggerEnter del script del jugador detecte la colisión correctamente.

Paso 3: Configurar la Salida

Asegúrate de que el objeto que representa la salida tiene un Collider y la etiqueta "Exit". Este objeto debe estar configurado como Trigger para que OnTriggerEnter funcione correctamente.

Paso 4: Verificar la Interacción

1. **Verifica las etiquetas y colisionadores:**
   * Asegúrate de que los objetos recolectables tengan la etiqueta "Collectible".
   * Los enemigos deben tener la etiqueta "Enemy".
   * La salida debe tener la etiqueta "Exit".
2. **Prueba el juego:**
   * Inicia el juego y mueve al personaje para recoger objetos, tocar enemigos y alcanzar la salida. Verifica que el marcador se actualice correctamente y que el juego termine al llegar a la salida.
3. **Ajusta según sea necesario:**
   * Si encuentras algún problema, revisa los valores en el Inspector y asegúrate de que todos los objetos tengan los componentes y etiquetas necesarios.

**Añadamos nuevas opciones:**

Partimos de este código:

using UnityEngine;

using UnityEngine.UI; // Necesario para trabajar con UI

public class PlayerController : MonoBehaviour

{

public float speed = 5f;

public Text scoreText; // Referencia al texto del marcador

private Rigidbody rb;

private int score = 0; // Variable para el marcador

void Start()

{

rb = GetComponent<Rigidbody>();

UpdateScoreText(); // Inicializar el texto del marcador

}

void Update()

{

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

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

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

rb.AddForce(movement \* speed);

}

private void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Exit"))

{

Debug.Log("¡Has ganado!");

// Mostrar mensaje de victoria o finalizar el juego

Application.Quit();

}

else if (other.CompareTag("Enemy"))

{

Debug.Log("¡Has perdido! Un enemigo te atrapó.");

// Reducir el marcador en 5 puntos y mostrar mensaje de derrota

score -= 5;

UpdateScoreText();

}

else if (other.CompareTag("Collectible"))

{

Debug.Log("¡Recolectado!");

// Incrementar el marcador en 10 puntos y destruir el objeto recolectable

score += 10;

UpdateScoreText();

Destroy(other.gameObject);

}

}

// Método para actualizar el texto del marcador

void UpdateScoreText()

{

scoreText.text = "Score: " + score;

}

}

**Provoquemos el fin del juego al llegar a “Exit” y al ser atacados por un “enemigo”**

using UnityEngine;

using UnityEngine.UI; // Necesario para trabajar con UI

using UnityEngine.SceneManagement; // Necesario para recargar la escena

public class PlayerController : MonoBehaviour

{

public float speed = 5f;

public Text scoreText; // Referencia al texto del marcador

public GameObject gameOverPanel; // Panel para mostrar mensaje de fin de juego

public Text gameOverText; // Texto del mensaje de fin de juego

private Rigidbody rb;

private int score = 0; // Variable para el marcador

void Start()

{

rb = GetComponent<Rigidbody>();

UpdateScoreText(); // Inicializar el texto del marcador

if (gameOverPanel != null)

{

gameOverPanel.SetActive(false); // Asegúrate de que el panel esté oculto al inicio

}

}

void Update()

{

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

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

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

rb.AddForce(movement \* speed);

}

private void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Exit"))

{

Debug.Log("¡Has ganado!");

EndGame("¡Has ganado!");

}

else if (other.CompareTag("Enemy"))

{

Debug.Log("¡Has perdido! Un enemigo te atrapó.");

// Reducir el marcador en 5 puntos y mostrar mensaje de derrota

score -= 5;

UpdateScoreText();

EndGame("¡Has perdido!");

}

else if (other.CompareTag("Collectible"))

{

Debug.Log("¡Recolectado!");

// Incrementar el marcador en 10 puntos y destruir el objeto recolectable

score += 10;

UpdateScoreText();

Destroy(other.gameObject);

}

}

// Método para actualizar el texto del marcador

void UpdateScoreText()

{

scoreText.text = "Score: " + score;

}

// Método para finalizar el juego

void EndGame(string message)

{

if (gameOverPanel != null && gameOverText != null)

{

gameOverPanel.SetActive(true); // Mostrar el panel de fin de juego

gameOverText.text = message; // Configurar el mensaje de fin de juego

}

Time.timeScale = 0; // Pausar el juego

}

}

**¿Por qué en el código mazegenerator.cs las paredes que construye automaticamente están por encima del plano?**

El problema de que las paredes se construyan por encima del plano en el script MazeGenerator.cs puede deberse a cómo se establecen las posiciones de las paredes. Asegúrate de que las paredes se colocan a la altura correcta en el mundo 3D. En Unity, los objetos se posicionan en un espacio tridimensional con coordenadas X, Y y Z. Si el plano de tu laberinto está en la coordenada Y = 0, y las paredes se colocan en Y = 1, por ejemplo, es probable que las paredes se posicionen por encima del plano.

Vamos a ajustar el script para asegurarnos de que las paredes se coloquen correctamente en la misma altura que el plano del laberinto. Aquí hay una versión ajustada del script que configura la altura de las paredes para que estén alineadas con el plano:

Script Ajustado MazeGenerator.cs

using UnityEngine;

public class MazeGenerator : MonoBehaviour

{

public GameObject wallPrefab; // Prefab de pared

public int width = 10; // Ancho del laberinto

public int height = 10; // Altura del laberinto

public float cellSize = 1f; // Tamaño de cada celda del laberinto

public float wallHeight = 1f; // Altura de las paredes

void Start()

{

GenerateMaze();

}

void GenerateMaze()

{

// Generar las paredes del borde del laberinto

// Asegúrate de que las paredes estén alineadas con el plano en Y = 0

for (int x = 0; x <= width; x++)

{

for (int z = 0; z <= height; z++)

{

// Generar paredes en los bordes

if (x == 0 || z == 0 || x == width || z == height)

{

InstantiateWall(x, z);

}

else

{

// Generar paredes internas de manera aleatoria

if (Random.value > 0.7f) // Probabilidad del 30% de generar una pared

{

InstantiateWall(x, z);

}

}

}

}

}

void InstantiateWall(int x, int z)

{

Vector3 position = new Vector3(x \* cellSize, wallHeight / 2, z \* cellSize);

Instantiate(wallPrefab, position, Quaternion.identity);

}

}

Explicación de los Cambios:

1. **Altura de la Pared (wallHeight):**
   * Se ha añadido una variable wallHeight que permite ajustar la altura de las paredes.
2. **Posición de la Pared:**
   * La posición de la pared se ajusta para que esté en la mitad de su altura, position = new Vector3(x, wallHeight / 2, z), asegurando que la base de la pared esté en el nivel Y = 0 del plano.

Pasos para Configurar en Unity:

1. **Asignar el Prefab de la Pared:**
   * Asegúrate de que el prefab de la pared (wallPrefab) tiene una escala de 1 en X y Z, y una altura adecuada en Y. La altura del prefab debe ser igual a wallHeight.
2. **Ajustar el Plano del Laberinto:**
   * Verifica que el plano del laberinto está en Y = 0. Si es necesario, ajusta su posición en el Inspector para asegurarte de que se alinea con las paredes.
3. **Probar el Script:**
   * Ejecuta el juego en Unity y verifica que las paredes se colocan correctamente en el nivel del plano. Si ves que las paredes están por encima o por debajo del plano, ajusta la variable wallHeight hasta que se alineen correctamente.

**Ahora desde aquí seguid vosotros…..**