**Vorschlag zur Code-Strukturierung**

**1. Hauptkomponenten**

* **MazeGenerator**: Verantwortlich für die Erstellung des Labyrinths.
* **ItemManager**: Verwalten und Platzieren der Items im Labyrinth.
* **PlayerController**: Steuerung des Spielers und Interaktionen mit der Umgebung.
* **UIManager**: Verwalten und Aktualisieren der Benutzeroberfläche.
* **GameManager**: Zentrale Steuerung des Spiels, Verwaltung von Spielzuständen und Kommunikation zwischen den Komponenten.

**2. Klassen und Funktionen**

**MazeGenerator.cs**

csharp

Code kopieren

using UnityEngine;

public class MazeGenerator : MonoBehaviour

{

public GameObject wallPrefab;

public int mazeWidth = 10;

public int mazeHeight = 10;

private void Start()

{

GenerateMaze();

}

private void GenerateMaze()

{

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

{

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

{

Vector3 position = new Vector3(x, 0, z);

if (Random.value > 0.1f) // 90% Chance, dass eine Wand platziert wird

{

Instantiate(wallPrefab, position, Quaternion.identity);

}

}

}

ItemManager.Instance.PlaceItems(mazeWidth, mazeHeight);

}

}

**ItemManager.cs**

csharp

Code kopieren

using UnityEngine;

public class ItemManager : MonoBehaviour

{

public static ItemManager Instance;

public GameObject[] itemPrefabs; // Array von verschiedenen Item-Prefabs

public int numberOfItems = 5; // Anzahl der Items im Labyrinth

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else

{

Destroy(gameObject);

}

}

public void PlaceItems(int mazeWidth, int mazeHeight)

{

for (int i = 0; i < numberOfItems; i++)

{

int x = Random.Range(0, mazeWidth);

int z = Random.Range(0, mazeHeight);

Vector3 position = new Vector3(x, 0, z);

GameObject itemPrefab = itemPrefabs[Random.Range(0, itemPrefabs.Length)];

Instantiate(itemPrefab, position, Quaternion.identity);

}

}

}

**PlayerController.cs**

csharp

Code kopieren

using UnityEngine;

public class PlayerController : MonoBehaviour

{

public float moveSpeed = 5f;

private Rigidbody rb;

private void Start()

{

rb = GetComponent<Rigidbody>();

}

private void Update()

{

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

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

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

rb.AddForce(movement \* moveSpeed);

}

private void OnTriggerEnter(Collider other)

{

if (other.CompareTag("Item"))

{

// Handle item collection

Destroy(other.gameObject);

UIManager.Instance.UpdateScore(1); // Example of score update

}

}

}

**UIManager.cs**

csharp

Code kopieren

using UnityEngine;

using UnityEngine.UI;

public class UIManager : MonoBehaviour

{

public static UIManager Instance;

public Text scoreText;

private int score;

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else

{

Destroy(gameObject);

}

}

public void UpdateScore(int points)

{

score += points;

scoreText.text = "Score: " + score;

}

}

**GameManager.cs**

csharp

Code kopieren

using UnityEngine;

public class GameManager : MonoBehaviour

{

public static GameManager Instance;

private void Awake()

{

if (Instance == null)

{

Instance = this;

}

else

{

Destroy(gameObject);

}

}

// Central game management logic goes here

}

**3. Integration und Zusammenarbeit der Komponenten**

* **MazeGenerator** erstellt das Labyrinth und ruft **ItemManager** auf, um die Items zu platzieren.
* **PlayerController** steuert den Spieler und interagiert mit den Items, dabei aktualisiert er die **UI** durch den **UIManager**.
* **GameManager** koordiniert das Spielgeschehen und stellt sicher, dass alle Komponenten miteinander kommunizieren.

**4. Vorteile der Modularität**

* **Wartbarkeit**: Einzelne Komponenten können unabhängig voneinander entwickelt und getestet werden.
* **Erweiterbarkeit**: Neue Funktionen oder Änderungen können leicht hinzugefügt werden, ohne den gesamten Code zu beeinflussen.
* **Wiederverwendbarkeit**: Module können in anderen Projekten wiederverwendet werden.