Einstieg in Unity Teil 3 – Die Macht der Programmierung

Description

Da wir in Unity nun per Code einen Würfel zum drehen gebracht haben und damit auch in der Lage sind, es beliebig zu bewegen, gehen wir im dritten und letzten Teil einen Schritt weiter: Wir duplizieren und manipulieren es.

Wozu soll das gut sein?

Besonders in der Spieleentwicklung ist Dynamik wichtig. Das heißt, dass das Spiel auf das Geschehen individuell reagieren muss. Stell dir eine Belohnung vor, die davon abhängig ist, wie gut der Spieler eine Aufgabe bewältigt hat. Und je nachdem tauchen dann ein, drei oder neun Objekte auf. Das sind Ereignisse, die man nur mühsam direkt im Level einbauen kann, aber dafür gibt es ja Code.

Was uns erwartet

Ich habe die Aufgabe zweigeteilt. Im ersten Schritt werden wir lernen, wie wir den Würfel kopieren. Wir erstellen dabei zwei Kopien: Ein Würfel erscheint links, der andere rechts vom Original. Beide erben dabei die Eigenschaften des Originals.

Im zweiten Schritt machen wir aus einem kleinen Würfel einen sehr großen, der aus insgesamt 27 kleinen Würfeln besteht. Dieser große Würfel wird sich dann drehen wie ein einzelner. Aber schau selbst.

https://www.bytegame.de/wp-content/uploads/2023/12/BigCube.mp4

Das große Problem

Denken wir die Aufgabe zu Ende. Wir kopieren das Original, die Eigenschaften werden an die Kopien weitergegeben. Was bedeutet das?

Genau, sie erzeugen weitere Kopien. Würden wir ein entsprechendes Skript an den Originalwürfel

binden, hätten wir eine unendliche Schleife, in der jeder neue Würfel zwei weitere hervorbringt. Unity bzw. das Programm würde sich aufhängen.

Das ist so, weil wir die Kopien beim Objektstart, also `void Start()` anfertigen.

Die Lösung

Um es zu umgehen, müssen wir ein unsichtbares Objekt in der Szene erstellen. Ich habe es mal *Scene-Controller* genannt. Das geht ganz einfach. In der *Hierarchy* links klicken und dann im Menü auf *Create Empty*.

Das bedeutet wiederum, dass wir das Skript so erstellen müssen, dass wir ein Originalobjekt vorgeben müssen. Hier wiederum gibt es einen kleinen Stolperstein: Es kann sein, dass es dieses Objekt nicht gibt. Wir müssen somit auch eine kleine Fehlermeldung für die Konsole generieren, falls dieser Fall eintritt.

Skript Create_duplicate_objects

Wenn du dein *Empty* hast, kannst du das Skript bei deinen Ressourcen erstellen und schon einmal dem *Empty* zuweisen. Ich zeige dir zuerst den Code, dann gehen wir es schrittweise durch.

```
using UnityEngine;

public class Create_duplicate_objects : MonoBehaviour
{
    public GameObject objPrefab; // Das Objekt-Prefab, das dupliziert werden s

    void Start()
    {
        // Versuche, das ursprüngliche Objekt in der Szene zu finden
        GameObject originalObj = GameObject.Find(objPrefab.name);

        if (originalObj != null)
        {
            // Erstelle ein Objekt rechts vom Original
            Vector3 rightPosition = originalObj.transform.position + new Vecto
            Instantiate(objPrefab, rightPosition, originalObj.transform.rotati

            // Erstelle ein Objekt links vom Original
            Vector3 leftPosition = originalObj.transform.position + new Vector
            Instantiate(objPrefab, leftPosition, originalObj.transform.rotatio

            // Deaktiviere das Skript nach der Erstellung
            enabled = false;
        }
        else
        {
            Debug.LogError("Original object not found in the scene. Make sure
        }
    }
```

```
}
```

Mit `public GameObject objPrefab;` holen wir uns das Objekt, welches wir kopieren möchten. Wenn du das Skript dem *Empty* zugewiesen hast, musst du also noch den Würfel auf *Obj Prefab* ziehen.

Mit `GameObject originalObj = GameObject.Find(objPrefab.name);` suchen wir dieses Objekt. Anschließend prüfen wir mit `if (originalObj != null)`, ob es da ist. Wenn nicht, geben wir unten mit `Debug.LogError` eine entsprechende Fehlermeldung an die Konsole.

Jetzt kommen die entscheidenden Zeilen:

```
Vector3 rightPosition = originalObj.transform.position + new Vector3(2.0f, 0, 0);
```

Hier wird eine neue `Vector3` mit dem Namen `rightPosition` erstellt. Die Position dieses Vektors wird durch die Addition der aktuellen Position des `originalObj` und einem weiteren Vektor `(2.0f, 0, 0)` bestimmt. Das bedeutet, dass die neue Position (`rightPosition`) um 2 Einheiten in der X-Richtung verschoben wird, während die Y- und Z-Koordinaten unverändert bleiben.

```
Instantiate(objPrefab, rightPosition, originalObj.transform.rotation);
```

Hier wird ein neues Spielobjekt instantiiert. Die Methode `Instantiate` wird verwendet, um eine Kopie des objPrefab-Spielobjekts zu erstellen. Dieses neue Spielobjekt wird an der Position `rightPosition` platziert, die wir zuvor berechnet haben. Die Rotation des neuen Spielobjekts wird auf die Rotation des `originalObj` gesetzt. Das bedeutet, dass die kopierte Instanz an der berechneten Position erscheint und die gleiche Ausrichtung wie das ursprüngliche Objekt hat.

Das Gleiche machen wir dann auch mit dem Objekt, welches links platziert werden soll.

Wichtig: Wir berechnen immer erst die neue Position, dann kopieren wir das Objekt und setzen es auf diese Position!

`enabled = false;` deaktiviert das Skript. Eigentlich ist es an dieser Stelle unwichtig. Es garantiert nur, dass nachfolgend nichts mehr aufgerufen wird. Damit sollte man allerdings vorsichtig sein. Wenn man noch mit `void Update()` arbeiten will, wird das nicht mehr funktionieren.

Der große Würfel

Bevor wir loslegen, müssen wir uns noch eine Sache überlegen. Es erscheint jetzt einfach, das Original 26 Mal zu kopieren, um einen großen Würfel zu erhalten. Okay, wir müssten die beiden oberen Zeilen entsprechend oft kopieren und die Werte anpassen, aber das ist zu viel Arbeit. Dafür hat – wie wir sehen werden – der Programmiergott Schleifen erfunden. Das Problem wird ein anderes sein: Wie drehen wir alle 27 Würfel, damit es aussieht, als wäre es ein einziger großer Würfel?

Mit dem aktuellen Wissen konnte man auf die Idee kommen, dass man bei jedem Frame alle Würfel durchgehen muss um die aktuelle Position und Rotation zu berechnen. Das klingt nach sehr viel Code und sehr viel Mathematik. Aber es gibt eine einfachere Lösung!

Dieses Problem lösen wir, indem wir einen Container erstellen. In diesen kopieren wir alle 27 rein. Am Ende drehen wir nicht jeden einzelnen Würfel, sondern lediglich den Container. Den Rest macht Unity für uns. Wie du sehen wirst, brauchen wir für die Drehung der 27 Würfel nur drei Zeilen, nämlich für jede Achse eine.

Bevor wir mir dem Skript loslegen, gehen wir in der *Hierarchy* auf den Würfel und deaktivieren das Skript für die Drehung (*Object_Routation*).

Skript Create_grid_of_objects

Ja, jetzt kommt gleich sehr viel Code. Aber keine Sorge: Nahezu alles, was jetzt kommt, hatten wir bereits in der kleinen Serie. Falls du Schwierigkeiten mit der dreifachen for-Schleife hast, dann zerbrich dir nicht den Kopf. In einer späteren C#-Serie wirst du alles erfahren, was du darüber wissen musst.

```
using UnityEngine;

public class Create_grid_of_objects : MonoBehaviour
{
    public GameObject objPrefab; // Das Objekt-Prefab, das dupliziert werden s
    public int gridWidth = 3; // Anzahl der Objekte in der Breite
    public int gridHeight = 3; // Anzahl der Objekte in der Höhe
    public int gridDepth = 3; // Anzahl der Objekte in der Tiefe
    public float spacing = 1.75f; // Abstand zwischen den Objekten
    public float rotationSpeedX = 60.0f; // Geschwindigkeit der Drehung um die
    public float rotationSpeedY = 30.0f; // Geschwindigkeit der Drehung um die
    public float rotationSpeedZ = 100.0f; // Geschwindigkeit der Drehung um di

    private GameObject objContainer;

    void Start()
    {
        // Finde das ursprüngliche Objekt in der Szene
        GameObject originalCube = GameObject.Find(objPrefab.name);

        if (originalCube != null)
        {
            objContainer = new GameObject("objContainer"); // Erstelle den Con

            Vector3 originalPosition = originalCube.transform.position;

            for (int x = 0; x < gridWidth; x++)
            {
                for (int y = 0; y < gridHeight; y++)
                {
                    for (int z = 0; z < gridDepth; z++)
                    {
                        objContainer.transform.position = originalPosition;
                        Vector3 offset = new Vector3(x * spacing, y * spacing,
                        Vector3 cubePosition = originalPosition + offset;
                        GameObject cube = Instantiate(objPrefab, cubePosition,
                        cube.transform.parent = objContainer.transform; // Mac
```

```
                    }
                }
            }

            Destroy(originalCube); // Lösche das Original
        }
        else
        {
            Debug.LogError("Original object not found in the scene. Make sure
        }
    }

    void Update()
    {
        // Drehe den Container um die X-, Y- und Z-Achsen
        objContainer.transform.Rotate(Vector3.right, rotationSpeedX * Time.del
        objContainer.transform.Rotate(Vector3.up, rotationSpeedY * Time.deltaT
        objContainer.transform.Rotate(Vector3.forward, rotationSpeedZ * Time.d
    }
}
```

Das Skript kannst du dem *Scene-Controller* zuweisen und *Create_duplicate_objects* deaktivieren. Vergiss nicht, auch dem neuen Skript den Würfel als Prefab zuzuweisen! Wenn alles geklappt hat, sollte sich die Szene bei dir so verhalten, wie im oberen Video.

Ich habe das Skript so geschrieben, dass du die Anzahl der Kopien in alle Richtungen selbst definieren kannst. Ebenso die Drehgeschwindigkeit und natürlich den Abstand. Du hast also einige Parameter, mit denen du herumexperimentieren kannst. Und das funktioniert natürlich nicht nur mit Würfeln. Du kannst auch Bälle, Kaffeetassen oder Zombies auf diese Weise kopieren und drehen.

Erklärungen

Vor dem Start bestimmen wir zunächst den Container mit `private GameObject objContainer;`. Erstellt wird er dann in Zeile 23 so:

```
objContainer = new GameObject("objContainer");
```

In der dritten Schleife passiert das Wesentliche:

```
for (int z = 0; z < gridDepth; z++)
{
    objContainer.transform.position = originalPosition;
    Vector3 offset = new Vector3(x * spacing, y * spacing, z * spacing) - new
    Vector3 cubePosition = originalPosition + offset;
    GameObject cube = Instantiate(objPrefab, cubePosition, Quaternion.identity
    cube.transform.parent = objContainer.transform; // Mache das Objekt zum Ki
}
```

`objContainer.transform.position = originalPosition;` Damit setzen wir die Position des Containers auf die Position des Original-Objekts. Wenn wir das nicht tun, ist der große Würfel später *irgendwo*, aber nicht dort, wo das Original war.

```
Vector3 offset = new Vector3(x * spacing, y * spacing, z * spacing) - new
Vector3(spacing, spacing, spacing);
```
und
```
Vector3 cubePosition = originalPosition + offset;
```

Damit bestimmen wir die Position des neuen Würfels. In der nächsten Zeile instanziieren wir es und daraufhin weisen wir es dem Container zu.

Mit Destroy(originalCube); zerstören wir das Original. Wir brauchen es nicht mehr, schließlich haben wir 3×3×3, also 27 Würfel erschaffen.

Und ja, in `void Update()` machen wir letztlich das selbe wie im zweiten Tutorial. Wir drehen in diesem Fall aber kein einzelnes Objekt, sondern den Container.

Ausblick

In dieser kurzen Einführungsserie hast du mit Sicherheit viel gelernt. Am besten ist es, das Gelernte anzuwenden. Du kannst nun selbst herumexperimentieren. In Zukunft wird es hier noch weitere Tutorials geben, in denen du noch sehr viel lernen wirst.

Danke für deine Zeit und weiterhin viel Spaß in der aufregenden Welt der Spieleentwicklung.

Weiterführende Links

Unity – Einstieg in die Spieleentwicklung
Einstieg in Unity Teil 1 – 3D und erste Szene
Einstieg in Unity Teil 2 – Der drehende Würfel
Warum soll ich programmieren lernen?

Externe Links

Unity Dokumentation

Date Created
15. Dezember 2023
Author
sven