


Einführung in Unity, Teil 2

In der Unity-Einführung Teil 2 werden wir das kleine 3D-Spiel „Roll-a-ball“ des ersten Unity-Tutorials  programmieren.

Ziel ist es, *GameObjects* mit Hilfe von C#-Skripten Leben einzuhauchen.

Vor C# muss man keine Angst haben. Wer Java kann, kann auch C#.

Sie können – eine Unity-Installation vorausgesetzt – die Übungen auch zuhause mit Hilfe des Video-Tutorials in Ruhe nachvollziehen. Das Tutorial enthält mehr Information als die Darstellung hier, insbesondere über den Umgang mit der Online-Hilfe. Am Ende wird auch gezeigt, wie sich ein Score einblenden lässt. Wegen der Kürze der Zeit unterbleibt dies hier.

Aufgabe 1 (Einstieg)

- a) Starten Sie die Aufgabe mit der Anlage eines neuen Projekts „Roll a Ball“.

Das Projekt enthält im *Project View* unter *Assets* einen Unterordner „*Scenes*“. Benennen Sie die „*SampleScene*“ darin in *rollaball* um.

- b) Legen Sie eine *Plane* an und benennen Sie sie im Inspektor in *Ground* um.

Sorgen Sie für folgende Werte der *Transform*-Komponente von *Ground*:

Position (0, 0, 0)

Rotation (0, 0, 0)

Skalierung (2, 1, 2)

- c) Als nächstes legen wir ein Spieler-Objekt an. Dazu erzeugen wir eine *Sphere* (Kugel) und benennen Sie im Inspektor in *Player* um.

Sorgen Sie für folgende Werte der *Transform*-Komponente von *Player*:

Position (0, 0.5, 0)

Rotation (0, 0, 0)

Skalierung (1, 1, 1)

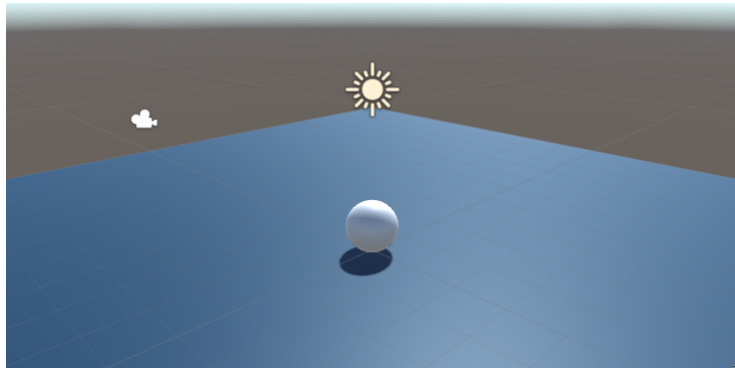
Mit dieser Wahl liegt die Kugel mit dem Durchmesser 1 genau auf dem Grund.

- d) Die weiße Kugel ist auf dem weißen Grund nur schwer zu erkennen.

Erzeugen Sie im *Project View* einen neuen Ordner *Materials* und darin ein Material *Background*. Für dieses Material setzen wir Albedo auf einen satten Blauwert (im Color-Picker die RGB-Werte (0, 32, 64)).

Über *Drag & Drop* ordnen wir das Material *Background* der Ebene *Ground* zu.

- e) Für das *Directional Light* setzen wir in der *Transform*-Komponente die Y-Rotation auf 60, um eine bessere Beleuchtung der Szene zu erzielen.



Aufgabe 2 (Bewegung des Spielers)

a) Nun sorgen wir für eine Bewegung des *Player*-Objekts und nutzen dafür die Physik-Engine. Wir fügen *Player* als Voraussetzung hierfür eine *Rigidbody*-Komponente hinzu.

b) Als nächstes wollen wir, dass wir *Player* mit der Tastatur bewegen können. Wir fügen dazu zu *Player* ein Skript hinzu. Zunächst erzeugen wir allerdings im *Project View* einen Ordner *Scripts*.

Danach selektieren wir *Player* und wählen im Inspektor „Add Component“ → „New Script“. Als Namen für das Skript geben wir „PlayerController“ ein. Die Voreinstellung „C Sharp“ für die Skriptsprache übernehmen wir. Mit „Create and Add“ wird das Skript angelegt und automatisch mit dem *Player*-Objekt verknüpft.

Wir können das Skript im *Project View* auswählen, es wird dann im Inspektor angezeigt (allerdings nicht editierbar). Verschieben Sie das Skript im *Project View* in den Unterordner *Scripts*, damit das Projekt eine saubere Struktur hat.

c) Wie editiert man das Skript? Mit *MonoDevelop*, *Visual Studio Code* oder *Visual Studio*. Standardmäßig ist *Visual Studio* eingestellt. Sie können dies unter „Edit“ → „Preferences“ → „External Tools“ einstellen.

Es gibt mehrere Wege, die Entwicklungsumgebung mit dem Skript zu starten:

- Doppelklick auf den Skript-Namen im *Project View*.
- Skript im *Project View* selektieren und Button *Open* im Inspektor drücken.
- *Player*-Objekt im *Scene View* oder der Hierarchie selektieren und Skriptnamen in der zugehörigen Komponente im Inspektor doppelklicken.

d) In der Entwicklungsumgebung sieht man den automatisch angelegten Rumpf des Skripts:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class PlayerController : MonoBehaviour {

    // Use this for initialization
    void Start () {
```

```

    }

    // Update is called once per frame
    void Update () {

    }

}

```

Die Kommentare geben bereits an, wo im Lebenszyklus des Spiels die einzelnen Methoden ausgeführt werden:

Start bei der Initialisierung, enthält also Code, der ganz zu Beginn ausgeführt wird.

Update enthält Code, der vor dem Zeichnen eines jeden Bilds (*Frames*) ausgeführt wird.

Es gibt aber noch weitere Methoden. Wir machen aus **Update** die Methode

FixedUpdate, die gegenüber **Update** den Vorteil hat, in festen Intervallen aufgerufen zu werden. Sie ist *frame rate*-unabhängig und eignet sich daher für Physik-Simulationen besser.

e) Wir brauchen jetzt die Eingabe des Benutzers. Dazu nutzen wir die Klasse **Input**. Mit

```

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

```

fragen wir in **FixedUpdate** Eingaben über die Pfeiltasten ab. Mehr Information zu **Input** und weiteren Klassen findet man in der *Unity Scripting Reference* .

Wir packen nun die beiden Werte als *x*- und *z*-Komponente in ein **Vector3**-Objekt namens **movement** (mit **new** erzeugen). Die *y*-Komponente setzen wir auf **0f**. Dieser Vektor ist eine gerichtete Kraft, die wir auf das **Player**-Objekt anwenden wollen.

f) Um die Kraft anwenden zu können, brauchen wir Zugriff auf die **Rigidbody**-Komponente von **Player**. Wir definieren dazu ein **privates** Attribut **rb** vom Typ **Rigidbody**. Dieses belegen wir in **Start** wie folgt:

```

rb = GetComponent<Rigidbody>();

```

Mit der **GetComponent**-Methode kann ein *GameObject* auf seine Komponenten zugreifen.

In **FixedUpdate** wenden wir nun den Kräftevektor **movement** auf **rb** an. Finden Sie in der *Unity Scripting Reference* zu **Rigidbody.AddForce** heraus, wie man das macht.

Testen Sie das Spiel im *Play*-Modus. Durch Drücken der Pfeiltasten kann man den Ball tatsächlich bewegen, allerdings sehr langsam.

g) Wir wollen in der Lage sein, die Geschwindigkeit anzupassen. Dazu nutzen wir einen sehr pfiffigen Mechanismus von Unity. Öffentliche Attribute im Skript werden in der zugehörigen Komponente im *Unity Editor* als Eingabefelder sichtbar.

Wir fügen im **PlayerController**-Skript ein öffentliches **float**-Attribut **speed** ein. In **FixedUpdate** multiplizieren wir den Kräftevektor mit **speed**, bevor wir ihn auf den **Rigidbody** **rb** anwenden.

Wie durch Zauberhand taucht im *Unity Editor* das zugehörige Eingabefeld bei der Skript-Komponente von *Player* auf. Experimentieren Sie mit den Eingabewerten, bis sich die Steuerung im *Play-Mode* so verhält, wie Sie es erwarten.

Aufgabe 3 (Kamerabewegung)

- a) *Player* bewegt sich nun. Wir möchten, dass sich die Kamera mitbewegt. Wir setzen die Kamera auf die folgenden Koordinaten:

Position (0, 10, -10)

Rotation (45, 0, 0)

Eine einfache Möglichkeit, die Kamera mit der Kugel zu bewegen, besteht darin, die Kamera in Hierarchie per *Drag & Drop* zu einem Kind von *Player* zu machen. Damit bewegt sich die Kamera mit der Kugel, rotiert allerdings auch mit ihr. Nicht so gut.

Wir lösen die Eltern-Kind-Beziehung zwischen *Player* und Kamera wieder und stellen eine Kopplung per Skript her.

- b) Dazu selektieren wir die Kamera und fügen über den Inspektor eine Skript-Komponente hinzu. Das Skript nennen wir *CameraController* und verschieben es im *Project View* auch gleich unter *Scripts*.

Öffnen Sie das Skript in der Entwicklungsumgebung und fügen Sie ein öffentliches Attribut *player* von Typ *GameObject* hinzu sowie ein privates Attribut *offset* vom Typ *Vector3*.

Nach Speichern taucht bei der Kamera in der Skript-Komponente das Eingabefeld *Player* im Inspektor auf. Wir ziehen das *GameObject Player* aus der Hierarchie in dieses Eingabefeld. Auf diese Weise schaffen wir die Verknüpfung zwischen *Player* im *Unity Editor* und Attribut *player* im Skript *CameraController*.

In *Start* berechnen wir den Abstandsvektor *offset* als die im *Unity Editor* eingestellte Differenz zwischen *Player* und der Kamera:

```
offset = transform.position - player.transform.position;
```

Jedes *GameObject* hat ja eine Transform-Komponente. *transform* ist die Transform der Kamera (das Skript ist ja an die Kamera angehängt), *player.transform* ist hingegen die Transform von *Player*.

- c) Aus der *Update*-Funktion machen wir eine *LateUpdate*-Methode. Diese wird nach Anwendung der Spielelogik ausgeführt. Wir setzen darin die Transformation (genauer die Position) der Kamera auf den richtigen Wert:

```
transform.position = player.transform.position + offset;
```

Testen im *Play-Mode*: Kamera folgt *Player*.

Aufgabe 4 (Anlage des Spielfelds)

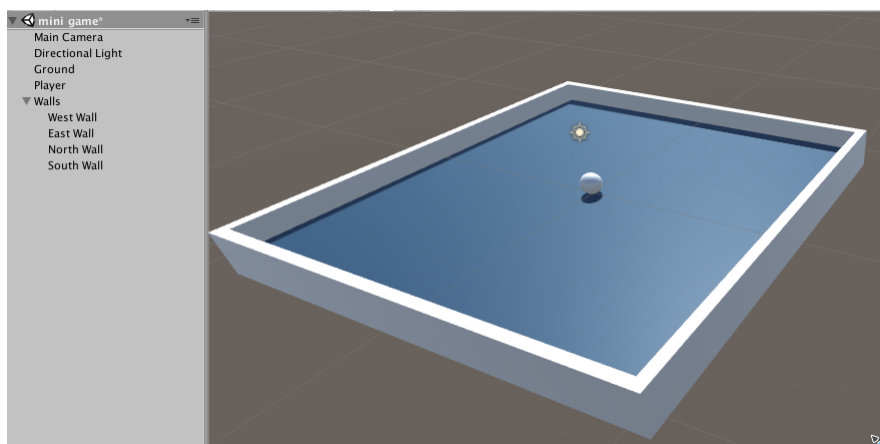
- a) Zunächst legen wir ein leeres, nur der Organisation dienendes *GameObject* als Kind von *Board* an und nennen es *Walls*. Achten Sie darauf, dass das Objekt bei (0, 0, 0) positioniert ist. Nun fügen wir ein *GameObject* vom Typ *Cube* hinzu, nennen es *West Wall* und machen es zum Kind von *Walls*. Im Inspektor setzen wir die Dimensionen wie folgt:

Position (-10, 0, 0)
 Rotation (0, 0, 0)
 Skalierung (0.5, 2, 20.5)

Damit haben wir eine Begrenzung. Wir selektieren *West Wall* und duplizieren das Objekt mit *Edit* → *Duplicate*. Das Duplikat benennen wir um in *East Wall*. Indem wir die *x*-Koordinate der Position auf 10 (statt -10) setzen, verschieben wir *East Wall* an die richtige Stelle.

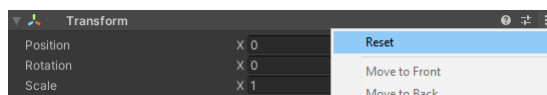
- b) Legen Sie nun auch eine nördliche und südliche Begrenzungswand als Kinder von *Walls* an. Finden Sie selber heraus, wie.

Testen im *Play*-Mode: Kugel wird von Wänden aufgehalten.



Aufgabe 5 (Sammelobjekte anlegen)

- a) Wir fügen einen *Cube* als Kind von *Board* hinzu und nennen ihn *Pick Up*. Mit *Reset* können wir die Koordinaten der Transform des neuen Objekts auf den Ursprung legen.



Das Objekt ist dann teilweise durch *Player* verdeckt. Wir deaktivieren *Player*,



um das Objekt ungestört sehen zu können. Das Objekt ist halb in der Ebene. Wir setzen die Transform von *Pick Up* wie folgt:

Position (0, 0.5, 0)
 Rotation (45, 45, 45)
 Skalierung (0.5, 0.5, 0.5)

- b) Wir wollen, dass das Sammelobjekt rotiert, damit es auffälliger ist. Dazu fügen wir *Pick Up* ein Skript Rotator hinzu. In dessen Update-Methode modifizieren wir die Transform-Komponente von *Pick Up* wie folgt:

```
transform.Rotate (new Vector3 (15, 30, 45) * Time.deltaTime);
```

Die Methode Rotate rotiert den Würfel. Im Vector3 sind die drei Euler-Winkel der Rotation angegeben. Damit die Rotation nicht von der *frame rate* abhängig ist, multiplizieren wir den Rotationsvektor mit *Time.deltaTime*. Dies ist die Zeit, die seit dem letzten Update verstrichen ist.

Testen im *Play-Mode*: *Pick Up* rotiert.

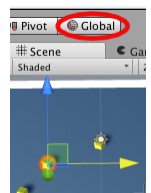
- c) Wir machen nun *Pick Up* zu einem *Prefab*. Dazu erzeugen wir im *Project View* einen Ordner *Prefabs* und ziehen das *Pick Up*-Objekt in diesen.

Änderungen am *Prefab* wirken sich auf alle davon abgeleiteten Objekte aus.

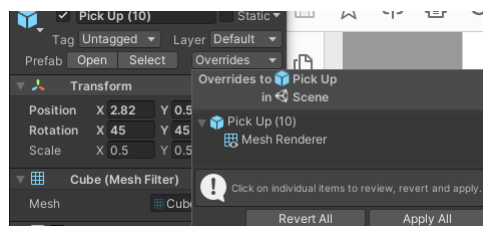
- d) Wir erzeugen nun ein leeres, nur der Organisation dienendes *GameObject* namens *Pick Ups* (rufen Sie ggf. das Reset-Menü für dieses Objekt auf, falls es nicht im Ursprung liegt) und machen *Pick Up* zu dessen Kindobjekt.

Wir duplizieren *Pick Up* 10-12 mal. Schneller als mit dem „Duplicate“-Menüeintrag geht dies mit der Tastenkombination **Strg** + **C** (Kopieren), **Strg** + **V**.

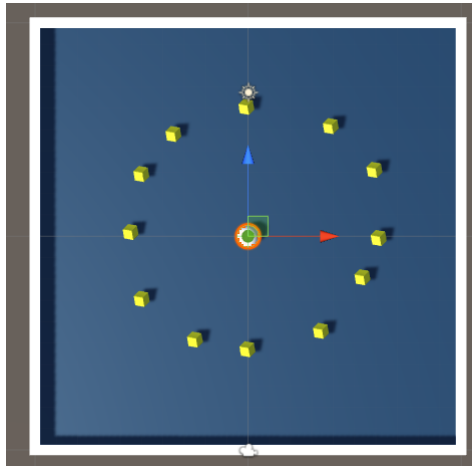
Mit dem roten (x) und dem blauen (z) Pfeil können wir die Duplikate parallel zu *Ground* verschieben. Dazu sollte das Koordinatensystem der *Pick Ups* auf „Global“ eingestellt sein (sonst verschiebt man entlang der rotierten Achsen des *Pick Ups*).



- e) Um die *Pick Ups* auffälliger zu machen, generieren wir ein neues Material mit gelbem Albedo (unter *Materials* als *Pick Up* speichern) und färben das *Prefab* damit. Die Einstellung lässt sich mit *Overrides* → *Apply All* auf alle *Prefabs* übertragen.



- f) Wir machen nun *Player* wieder sichtbar, indem wir die *Active*-Checkbox setzen.



Aufgabe 6 (Kollisionserkennung)

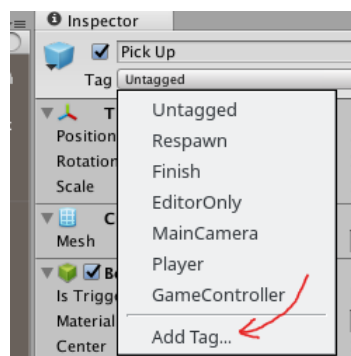
- a) Zum Einsammeln der Sammelobjekte müssen Kollisionen zwischen diesen und *Player* erkannt werden. Dafür sind *Collider*-Komponenten zuständig. Diese lösen *Events* aus.

Fügen Sie zu *PlayerController* die folgende Methode hinzu:

```
void OnTriggerEnter(Collider other)
{
    if (other.gameObject.CompareTag ("PickUp"))
    {
        other.gameObject.SetActive (false);
    }
}
```

Diese *Event*-Behandlung wird ausgelöst, sobald *Player* mit einem Objekt mit *Collider*-Komponente kollidiert. Es soll aber nur etwas passieren, wenn das andere Objekt *other* ein *Pick Up* ist – wir setzen *other* dann auf inaktiv.

- b) Wie erkennen wir *Pick Ups*? Dazu versehen wir das *Prefab* mit einem *Tag* mit Wert *Pick Up*. Im Code oben überprüft die *CompareTag*-Methode, ob das Objekt *other* diesen Tag besitzt.



Konkret wählen wir das *Pick Up*-Prefab und klappen die mit *Tag* benannte *Select Box* auf. Dort gibt es das gewünschte Tag noch nicht. Mit „Add Tag“ (danach ein kleines Plus-Symbol suchen)

können wir es erzeugen. Wichtig: Danach müssen wir es mit der *Select Box* noch einmal explizit zuweisen. (Erzeugen und Zuweisen sind separate Schritte).

c) Damit alles erwartungsgemäß funktioniert, müssen wir noch drei Kleinigkeiten im *Pick Up-Prefab* modifizieren:

- Setzen der Checkbox „*Is Trigger*“ in der *Box Collider*-Komponente,
- Hinzufügen einer *Rigidbody*-Komponente und
- Setzen der Checkbox „*Is Kinematic*“ in der *Rigidbody*-Komponente.

Testen im *Play*-Mode: Sobald die Kugel mit einem Sammelobjekt kollidiert, verschwindet dieses.

Fragen zur Übung (prüfungsrelevant)

- 1.) Ein C#-Skript sei eine Komponente eines *GameObjects* in der *Unity-Game Engine*. Was passiert, wenn man Attribute der zugehörigen Klasse als öffentlich deklariert? Was vereinfacht sich dadurch?
- 2.) Wann erfolgt die Ausführung der Methoden *Update*, *LateUpdate* und *FixedUpdate* im Lebenszyklus eines Unity-Skripts? Wozu eignen sich die Methoden?
- 3.) Auf welche Arten kann man in Unity-Anwendungen dafür sorgen, dass sich die Kamera mit einem Objekt mitbewegt?
- 4.) Wozu nutzt man *Tags* in Unity?