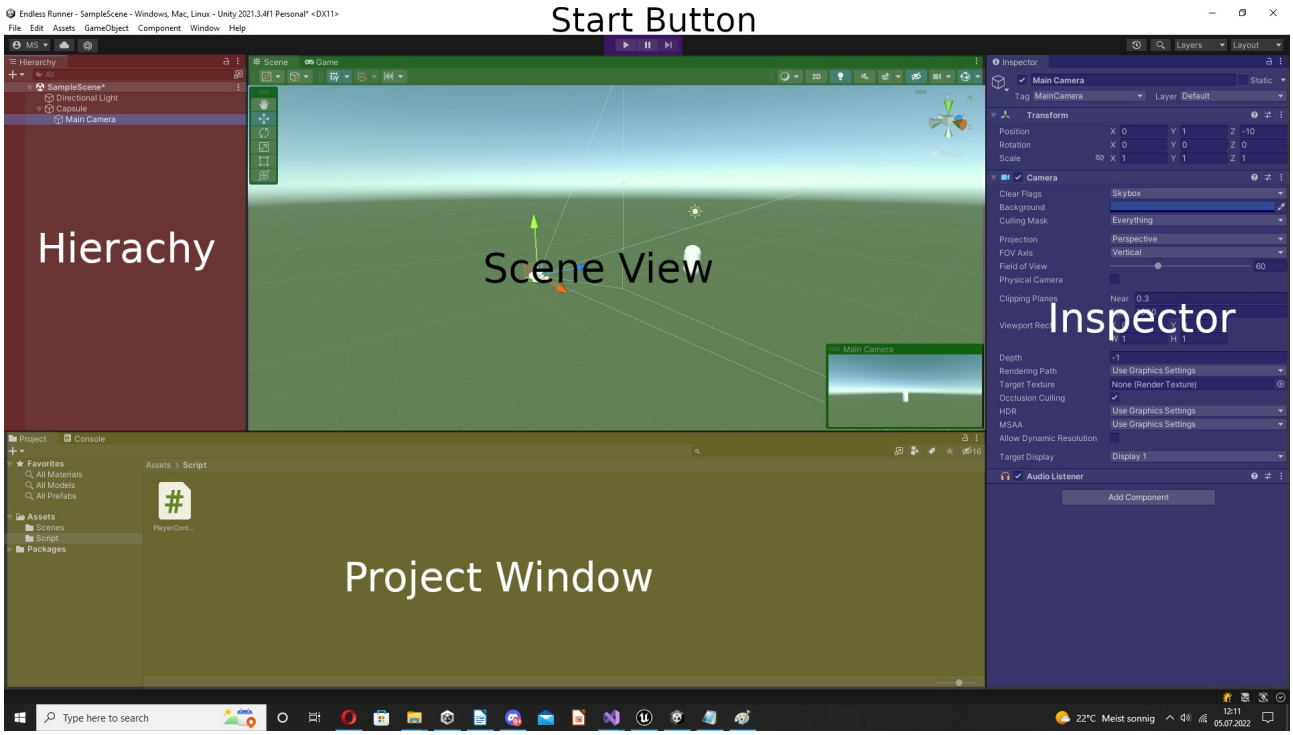
"Endless Runner" - Entwicklung eines Mobilegames

Aufbau der Unity-Oberfläche



Hierachy: Hier werden die Objekte aufgelistet, die in der Scene-View zu sehen sein sollen.

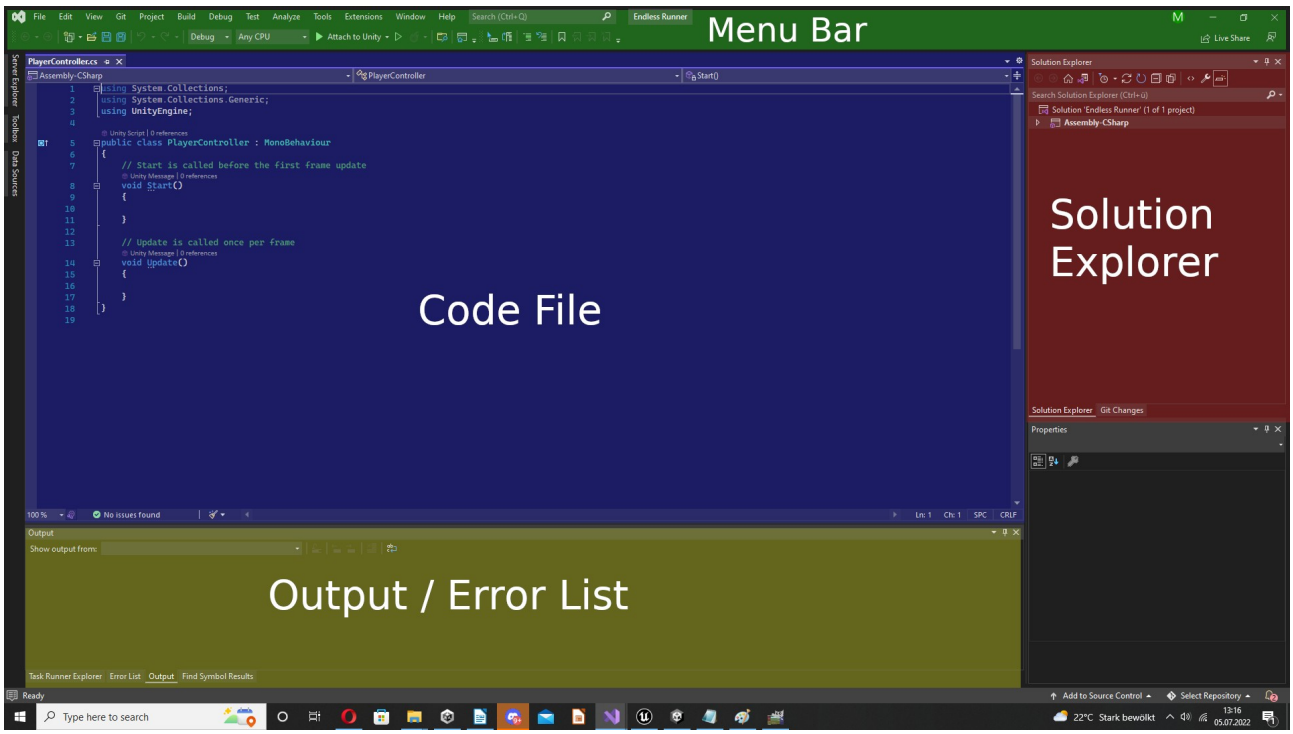
Scene View: Hier wird das Spiel in 3D angezeigt.

Inspector: Hier werden die Funktionalitäten des ausgewählten Game-Objekts dagestellt

Project-Window: Hier befinden sich die Assets (3D-Modells, Bilder, Sound C#-Skripte usw.) des Projektes

Start Button: Drücke hier um das Spiel zu starten

Aufbau der Visual Studio Oberfläche:



Menu-Bar: Hier können alle Einstellungen umgesetzt und das Programm gestartet werden.

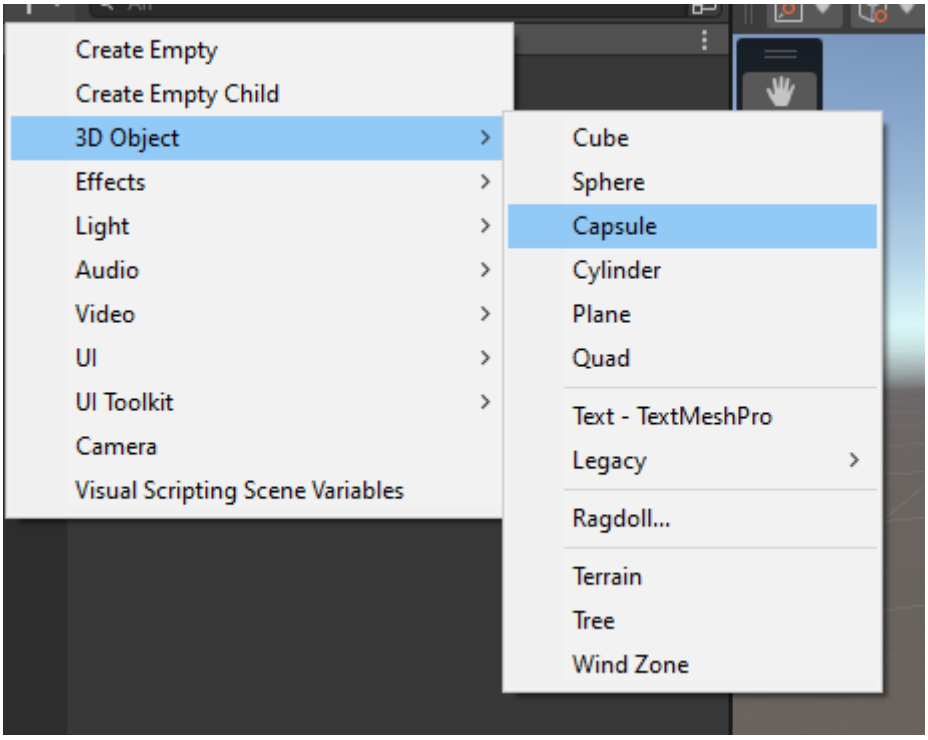
Code-File: Hier wird der Programmablauf geschrieben

Solution-Explorer: Hier werden alle Codedateien wie im Windows-Explorer aufgelistet

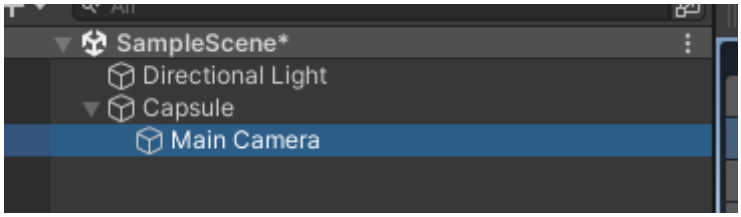
Output/Error-List: Das Feedback von Visual Studio in wie weit der Code funktioniert

1. Erstellen der Spielfigur

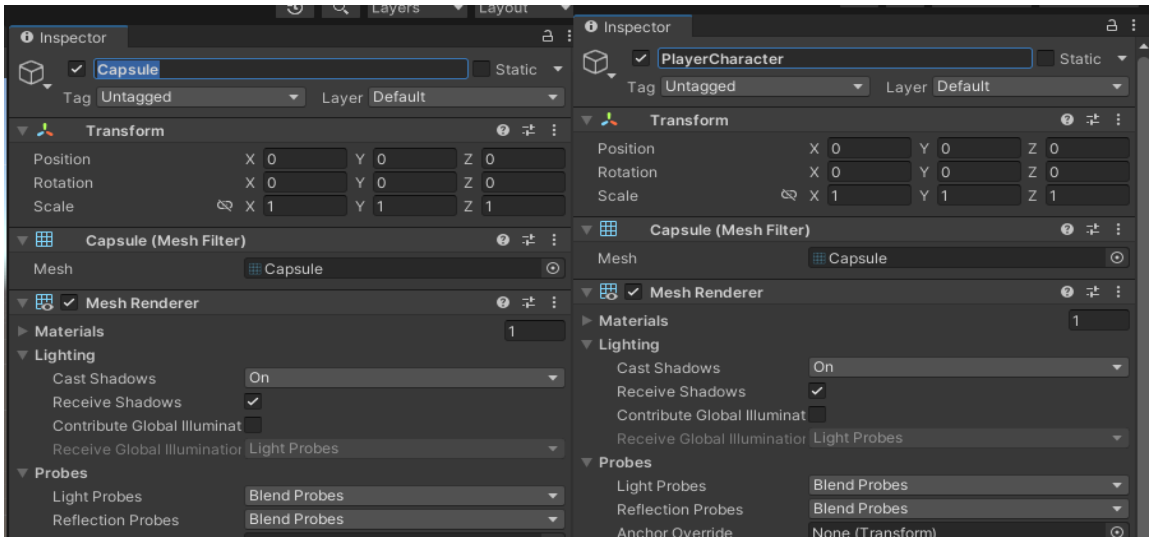
Als erstes muss eine Spielfigur erstellt werden. Dazu drücken wir in der Hierachy auf den +-Knopf und wählen unter dem Reiter 3D-Objekte „Capsule“ aus.



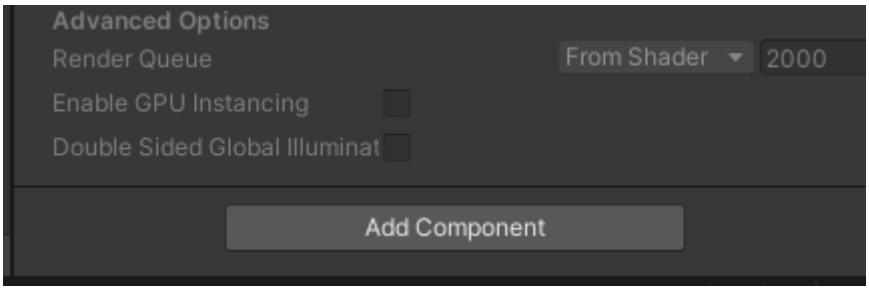
Als nächstes nehmen wir das „Main Camera“-Objekt und schieben es mit der linken Maustaste es über das Kapsel-Objekt um es anzuhängen. Jetzt folgt die Kammera immer unserer Kapsel.



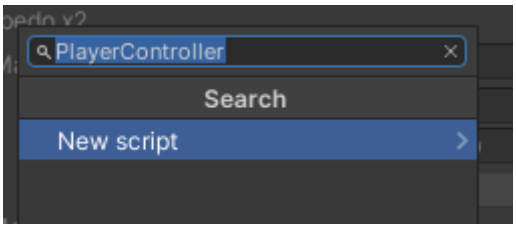
Danach benennnen wir Capsule im Inspector zu „PlayerCharacter“ um, um kenntlich zu machen, dass es sich um die Spielfigur handelt.



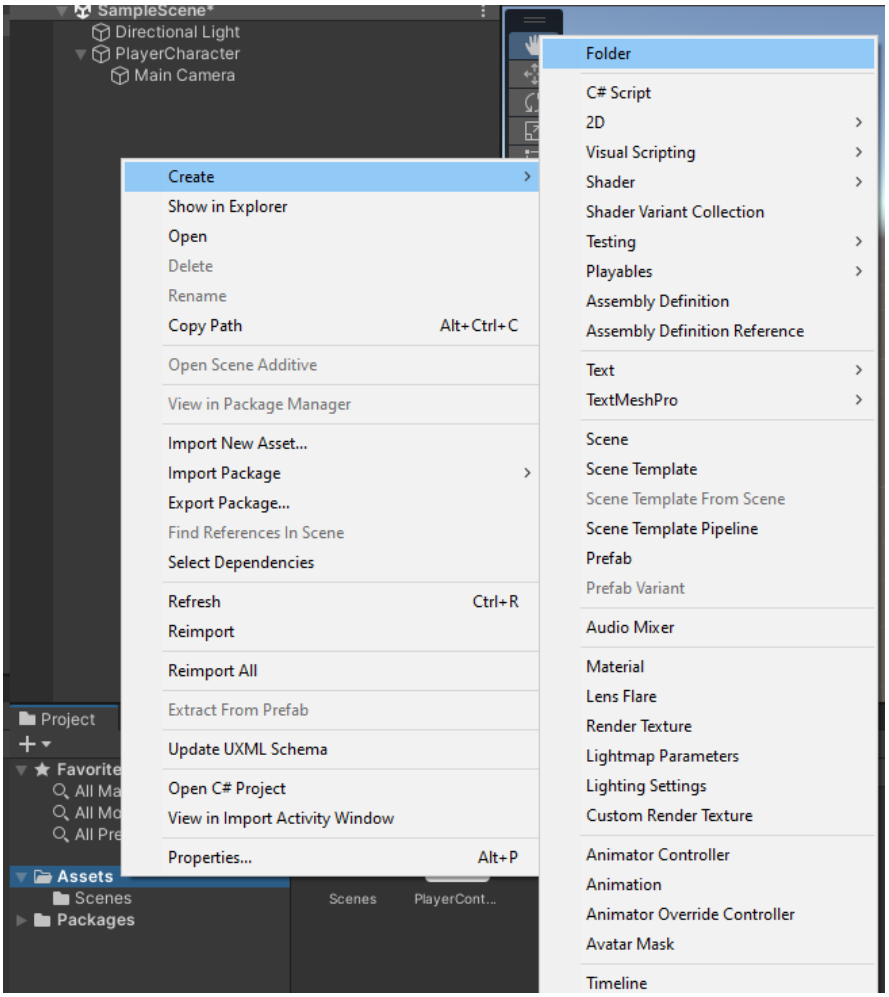
Wir drücken am unteren Ende des Inspectors, wenn PlayerCharacter ausgewählt ist, auf „AddComponent“.



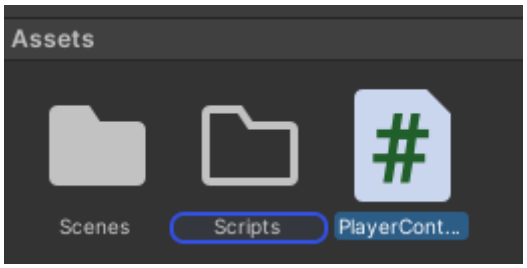
Wir schreiben PlayerController in die Suchleiste und drücken auf „New script“ um ein neues C#-Script zu erstellen. Danach drücken wir auf „Create and Add“



Wir drücken im Project-Window links in der Leiste mit der rechten Maustaste auf Assets, öffnen den Reiter Create und drücken auf Folder um einen neuen Ordner zu erstellen. Diesen nennen wir „Scripts“.



In diesen ziehen wir das neu erstellte C#-Script „PlayerController“ um eine saubere Ordnerstruktur zu haben.



Nun doppelklicken wir auf das PlayerController-Script um es in Visual Studio zu öffnen.

2. Movement

In Visual-Studio sollte das C#-Script wie folgt aussehen:

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

public class PlayerController : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {

    }

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

    }
}
```

Als erstes fügen wir Update (in die geschweiften Klammern {}) folgende Zeile hinzu:

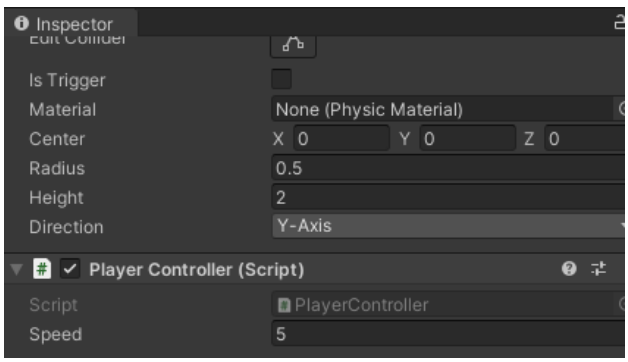
```
transform.position += Vector3.forward * Time.deltaTime;
```

Nachdem gespeichert wurde, sollte unsere Spielfigur mit Kamera, immer wenn wir Start drücken sich grade nach vorne bewegen.

Da sie noch etwas langsam ist, fügen wir die Float-Variable Speed, eine Gleitkommazahl, der PlayerController-Klasse hinzu und multiplizieren sie mit dem Wert der transform.position in der Update übergeben wird. Auch hier muss gespeichert werden. (Strg + UmschaltLinks + S)

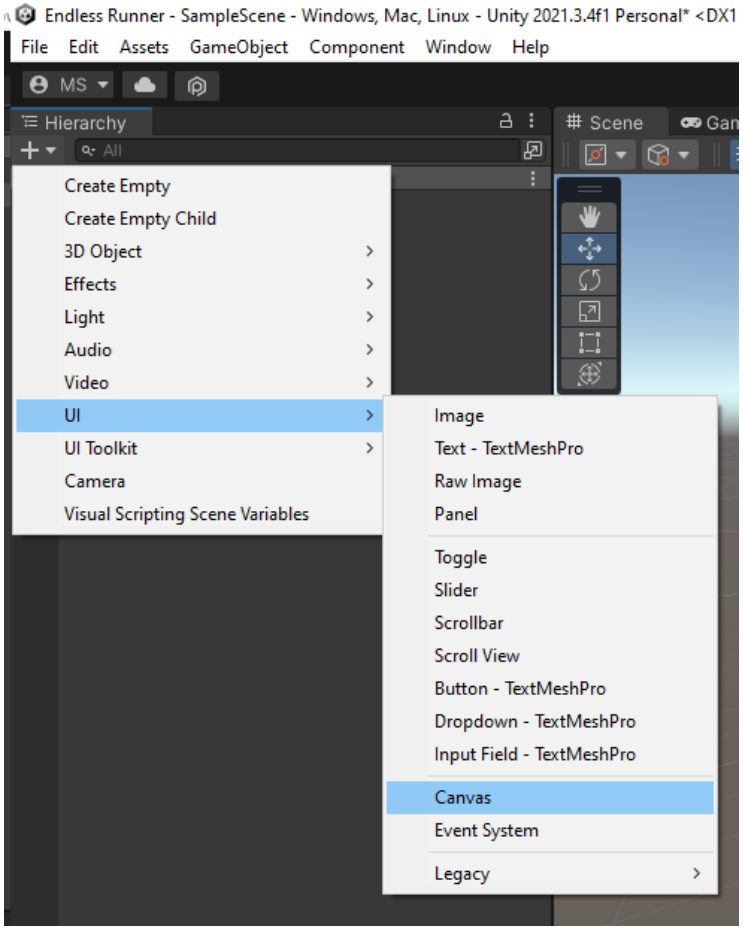
```
public class PlayerController : MonoBehaviour
{
    [SerializeField] private float _speed = 5.0f;
    // Start is called before the first frame update
    void Start()
    {

    }

    // Update is called once per frame
    void Update()
    {
        transform.position += Vector3.forward * _speed * Time.deltaTime;
    }
}
```

Die Geschwindigkeit kann im Inspektor angepasst werden.



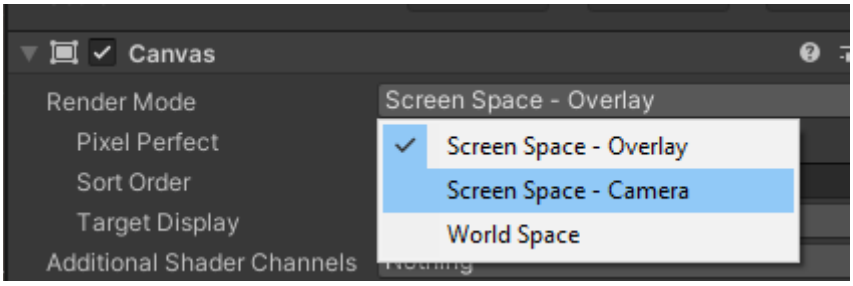
3. Interaktion

Als nächstes erstellen wir in der Hirachy ein Canvas-Objekt. Auf diesem kann UI erstellt werden.

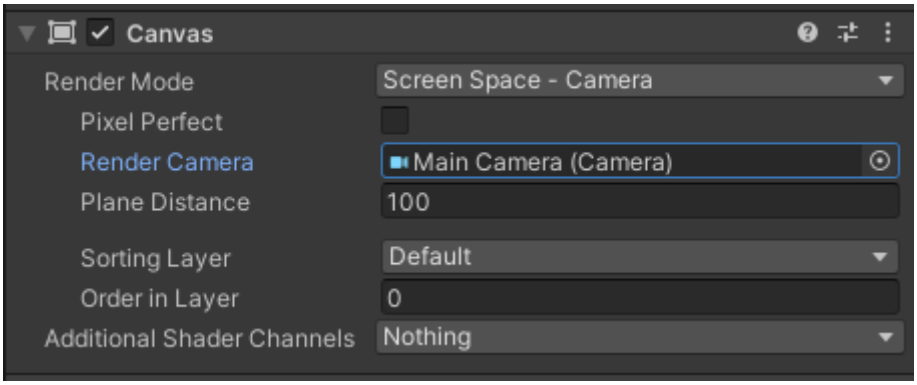


Als erstes müssen einige Einstellungen getroffen werden.

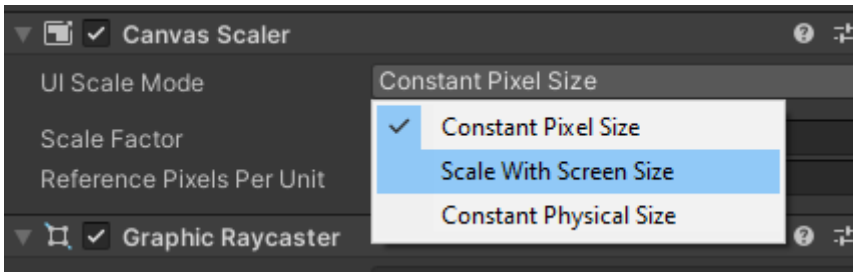
Der Render-Mode des Canvas muss im Inspector zu „Screen Space – Camera“ geändert werden.



Die Main Camera aus der Hirachy muss in die Rander Camera im Canvas des Inspektors gezogen werden.

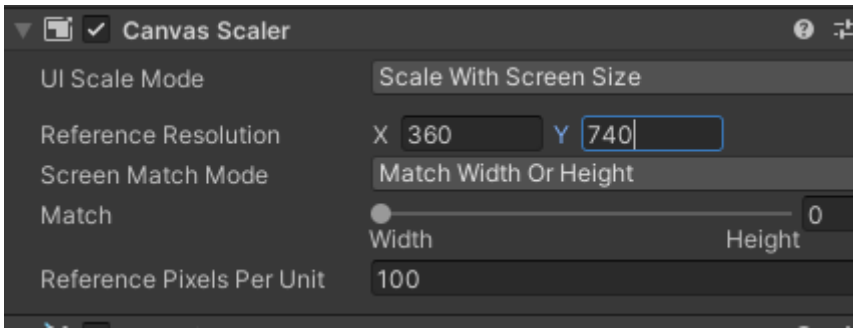


Im Canvas-Scaler im Inspektor wird der „UI Scale Mode“ zu „Scale With Screen Size“ geändert.

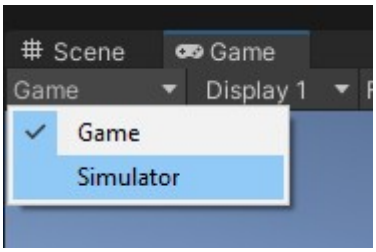


Die Reference-Resolution wird einem Mobile-Bildschirm angepasst und zu 360x740.

Die Größe muss gegebenensfalls dem Endgerät angepasst werden
https://www.icwebdesign.co.uk/common-viewport-sizes



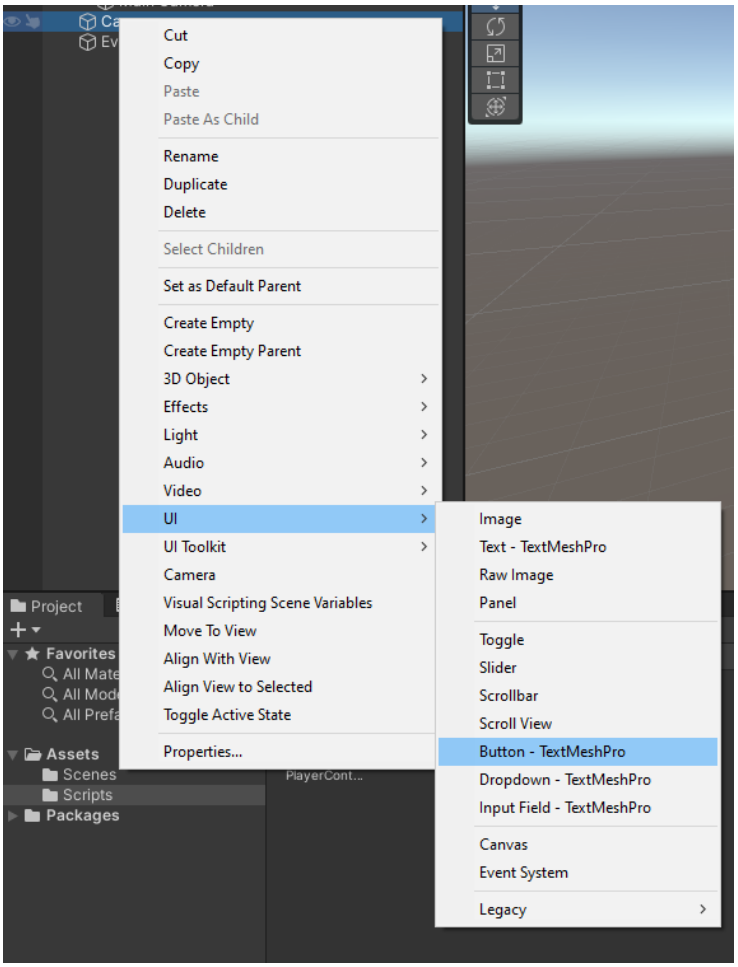
Um eine Mobile Ansicht zu bekommen, drücken wir in der Scene-View auf den Game-Tab und wechelsen von Game zu Simulator.



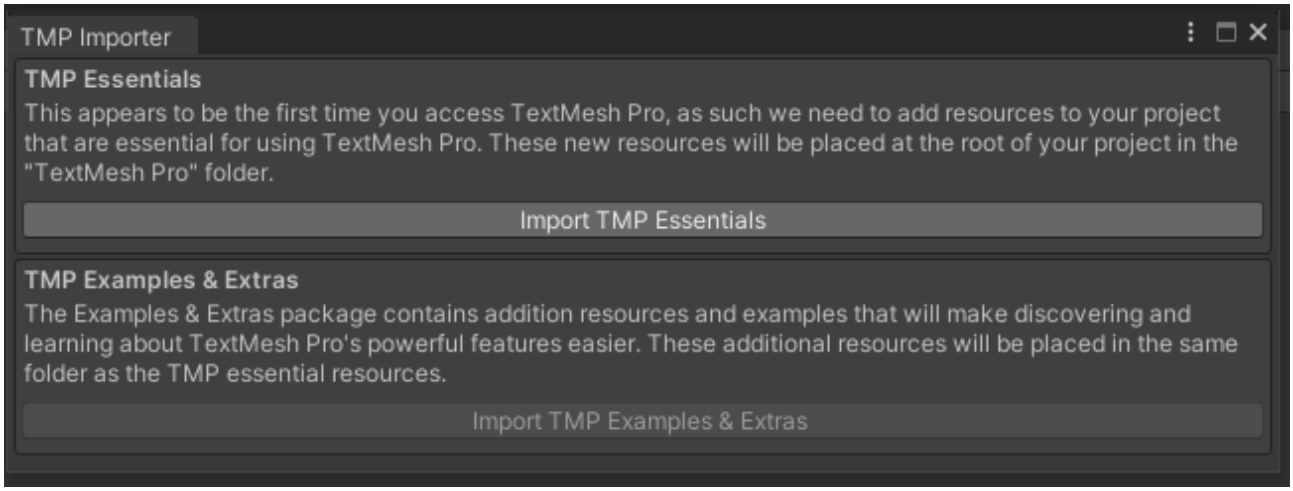
Nun können wir das gewünschte Mobile-Gerät auswählen.

Da wir nun eine UI-Oberfläche haben, können wir sie mit Objekten befüllen.

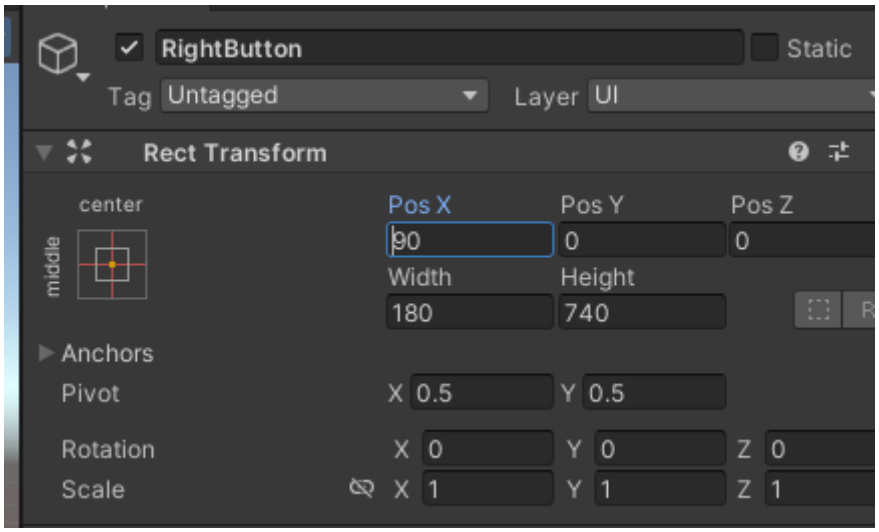
Wir klicken mit der rechten Maustaste auf das Canvas in der Hirachy und wählen im Reiter UI „Button – TextMeshPro“ aus.



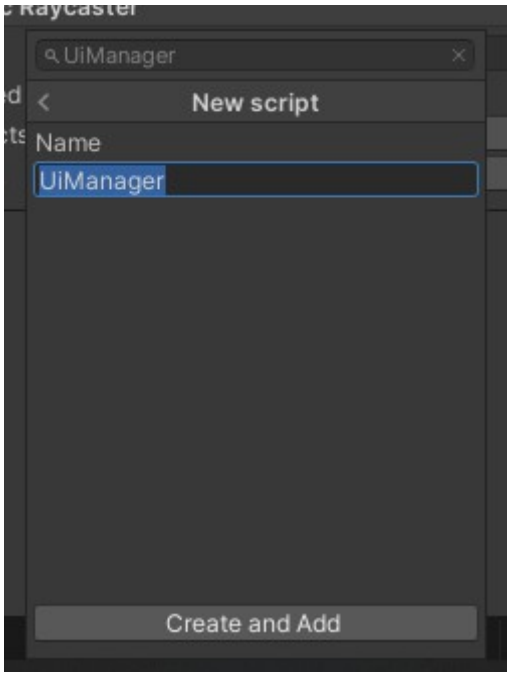
Nun müssen wir die TMP Essentials importieren



Der Button wird als Right-Button benannt, seine X-Position wird auf -90 gesetzt, seine Breite auf 180 und seine Höhe auf 740 (Größe muss dem Endgerät angepasst werden, er sollte halb so breit und so hoch wie der Bildschirm des Endgerätes sein und Pos X ist immer 0,5 * Breite)



Als nächstes wollen wir dem Button Funktionalität geben und erstellen im Inspector des Canvas, wie schon im PlayerCharacter ein C#-Script und nennen es „UiManager“.

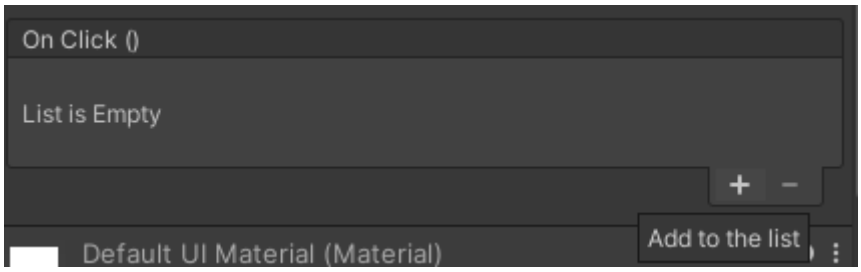


Auch dieses Script kommt in den Scripts-Ordner im Project-Window.

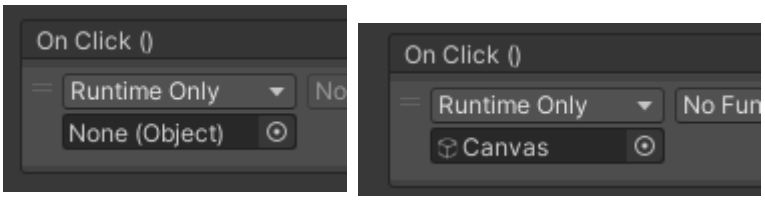
Wir fügen UiManager die Methode OnRightButtonTouched hinzu und Speichern (Strg + UmschaltLinks + S)

```
public class UiManager : MonoBehaviour
{

    public void OnRightButtonTouch()
    {
        Debug.Log("Right Button Touched!");
    }
```

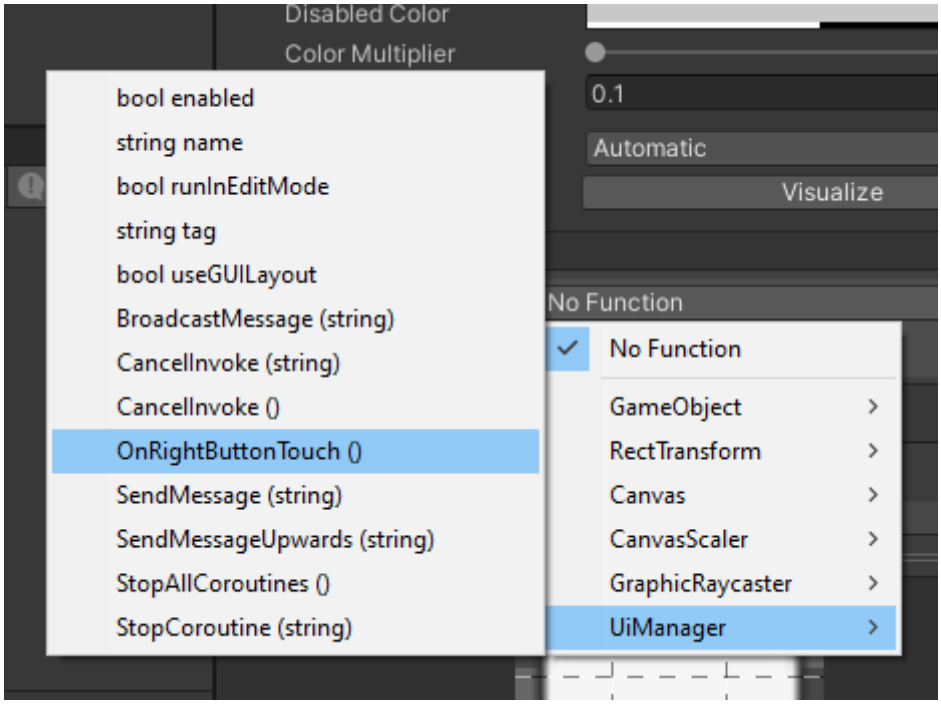
Nun müssen wir die Methode noch dem Button zuweisen. Dazu rufen wir RightButton im Inspektor in Unity auf und drücken bei On Click () auf +.



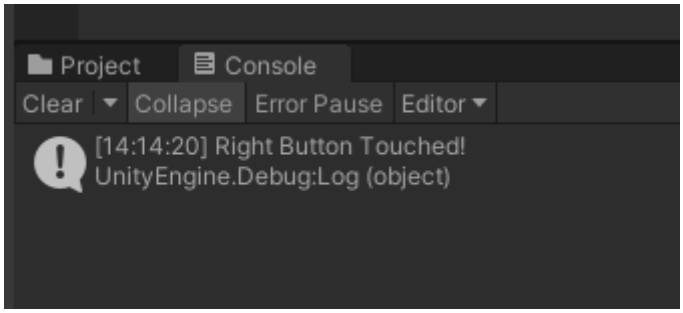
Nun ziehen wir das Canvas aus der Hirachy in „None (Object)“.



Nun können wir bei „No Function“ unter dem Reiter UIManager die eben erstellte Methode „OnRightButtonTouch ()“ auswählen.



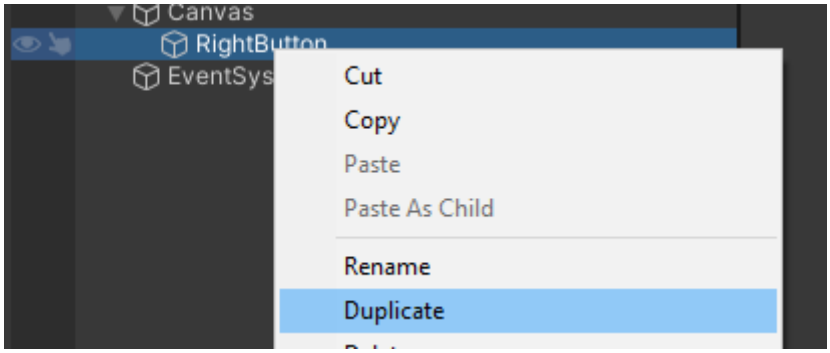
Wenn wir nun das Spiel starten und im Project-Window den Tab „Console“ auswählen, können wir wann immer wir auf den Button drücken die Textausgabe „Right Button Touched!“ lesen.



Da wir unser so erstellten rechten Rand des Touchpads natürlich im Spiel nicht sehen wollen, gehen wir im Inspector des RightButton auf die Image-Komponente und setzen bei Color den Wert A (für Alpha bzw. Transparenz) auf 0.

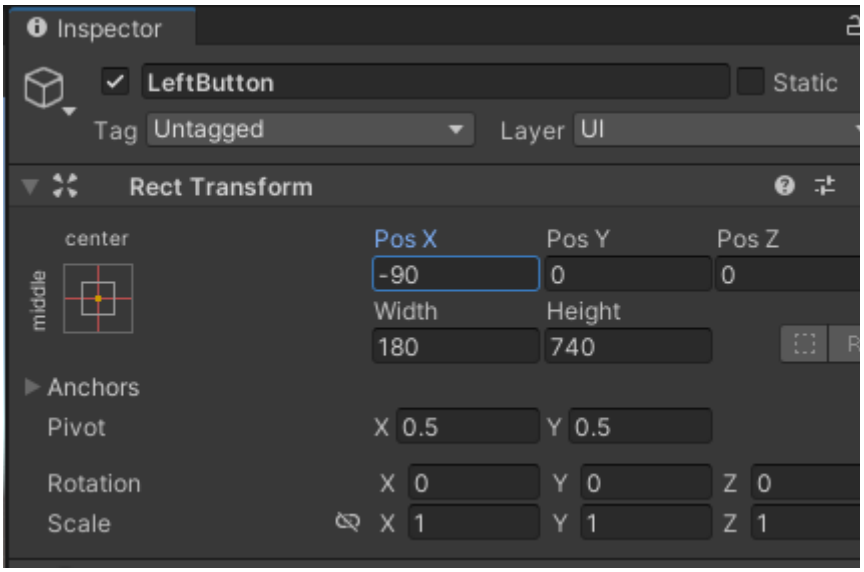
Wir löschen das an RightButton angehängte Objekt Text (TMP), damit der Text auf dem Button verschwindet.

Da wir nun die rechte Hälfte des Bildschirms ansprechen können wollen wir auch die linke ansprechen. Wir klicken auf den RightButton in der Hirachy und dann auf Dublicate.



Nun sollte ein Klon des RightButton mit dem Namen RightButton (1) entstanden sein.

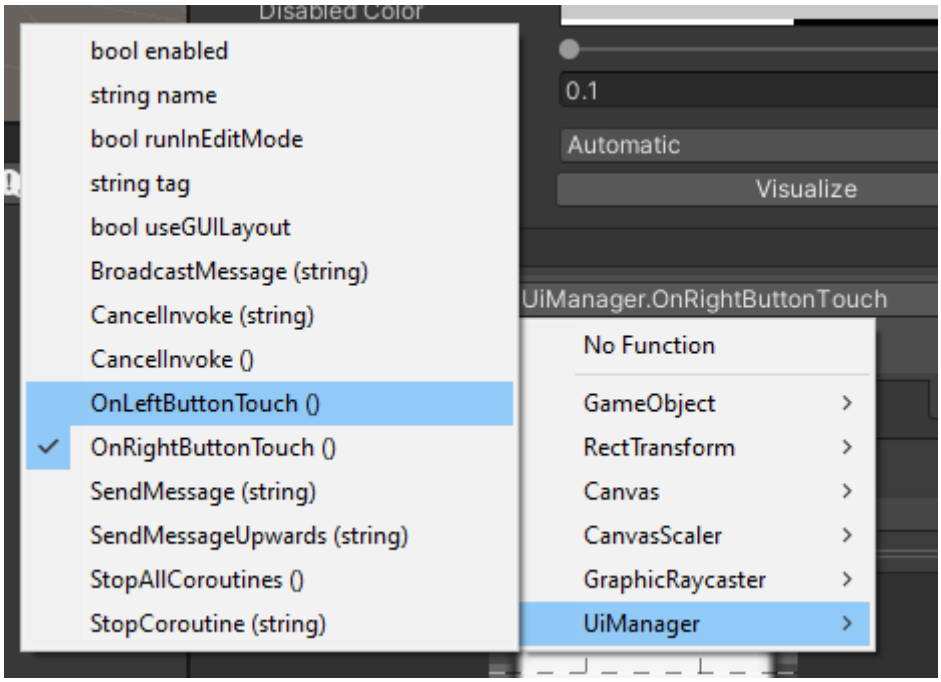
Dieser wird im Inspektor zu LeftButton umgenannt und vor seine X-Position wird ein – geschrieben.



Auch er bekommt im UiManager in VisualStudio eine eigene Methode namens OnLeftButtonTouched:

```
public class UiManager : MonoBehaviour
{

    public void OnRightButtonTouch()
    {
        Debug.Log("Right Button Touched!");
    }

    public void OnLeftButtonTouch()
    {
        Debug.Log("Left Button Touched!");
    }
```

Im Inspector des LeftButton müssen wir nun noch unter OnClick() die Methode auf OnLeftButtonTouch() anpassen.



Nun haben wir einen vollständig ansprechbaren Smartphone-Bildschirm.

4. Rotation

Nun da wir Interaktion für den Spieler haben, müssen wir Spielmechaniken bauen, um diese nutzbar zu machen.

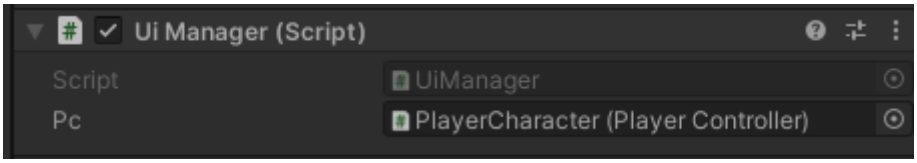
Wir öffnen nun den PlayerController.cs-Script im Script-Ordner im Project-Window und fügen die Methoden RotatePlayerRight und RotatePlayerLeft hinzu, um der Spielfigur eine 90°-Drehung zu ermöglichen.

```
public class PlayerController : MonoBehaviour
{
    [SerializeField] private float _speed = 5.0f;

    public void RotatePlayerRight()
    {
        transform.rotation =
            Quaternion.Euler( transform.localEulerAngles.x,
                              transform.localEulerAngles.y + 90,
                              transform.localEulerAngles.z );
    }


    public void RotatePlayerLeft()
    {
        transform.rotation =
            Quaternion.Euler( transform.localEulerAngles.x,
                              transform.localEulerAngles.y - 90,
                              transform.localEulerAngles.z );
    }
```

Nun erstellen wir im UiManager ein Objekt für den PlayerController um auf ihn zugreifen zu können.

```
public class UiManager : MonoBehaviour
{
    [SerializeField] private PlayerController _pc = null;
```

Nun ziehen wir PlayerCharacter aus der Hirachy in das Pc-Feld im Ui-Manager des Canvas im Inspector um die Spielfigur dem Ui zugreifbar zu machen.



Nun erweitern wir die Methoden im Ui-Manager wie folgt.

```
public class UiManager : MonoBehaviour
{
    [SerializeField] private PlayerController _pc = null;

    public void OnRightButtonTouch()
    {
        Debug.Log("Right Button Touched!");
        _pc?.RotatePlayerRight();
    }

    public void OnLeftButtonTouch()
    {
        Debug.Log("Left Button Touched!");
        _pc?.RotatePlayerLeft();
    }
```

Nun müssen wir nur noch dafür sorgen, dass sich die Spielfigur in Blickrichtung bewegt. Dazu erstellen wir im PlayerController die Variable _moveDir für die Blickrichtung und entwickeln die Methode InitMoveDirection, die wir in RotatePlayerRight(), RotatePlayerLeft und Start() implementieren. Das Vector3.forward in der Update ersetzen wir durch _moveDir.

```
public class PlayerController : MonoBehaviour
{
    [SerializeField] private float _speed = 5.0f;
    [SerializeField] private Vector3 _moveDir = Vector3.zero;

    public void InitMoveDirection()
    {
        switch(transform.localEulerAngles.y)
        {
            case 0: _moveDir = Vector3.forward; break;
            case -270: //nobreak
            case 90: _moveDir = Vector3.right; break;
            case -180: //nobreak
            case 180: _moveDir = Vector3.back;  break;
            case -90: //nobreak
            case 270: _moveDir = Vector3.left; break;
        }
    }
    public void RotatePlayerRight()
    {
        transform.rotation =
            Quaternion.Euler( transform.localEulerAngles.x,
                              transform.localEulerAngles.y + 90,
                              transform.localEulerAngles.z );

        InitMoveDirection();
    }


    public void RotatePlayerLeft()
    {
        transform.rotation =
            Quaternion.Euler( transform.localEulerAngles.x,
                              transform.localEulerAngles.y - 90,
                              transform.localEulerAngles.z );

        InitMoveDirection();
    }
```

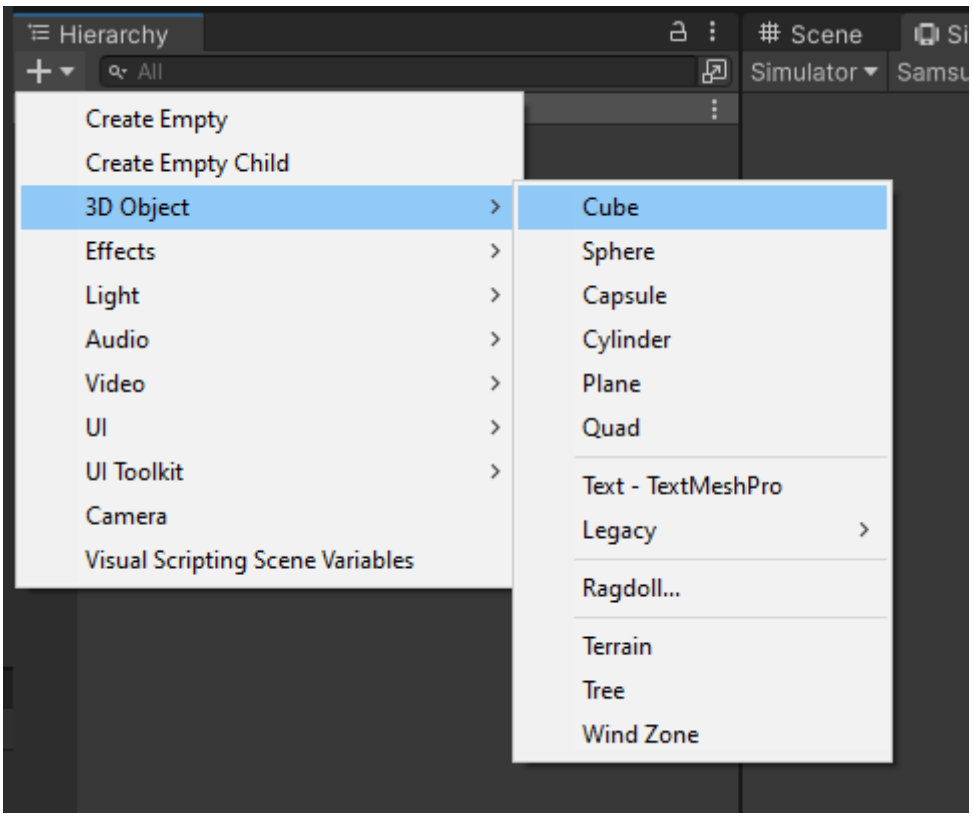
```
    // Start is called before the first frame update
    void Start()
    {
        InitMoveDirection();
    }

    // Update is called once per frame
    void Update()
    {
        transform.position += _moveDir * _speed * Time.deltaTime;

    }
}
```

Nun sollte sich die Spielfigur in Blickrichtung bewegen.

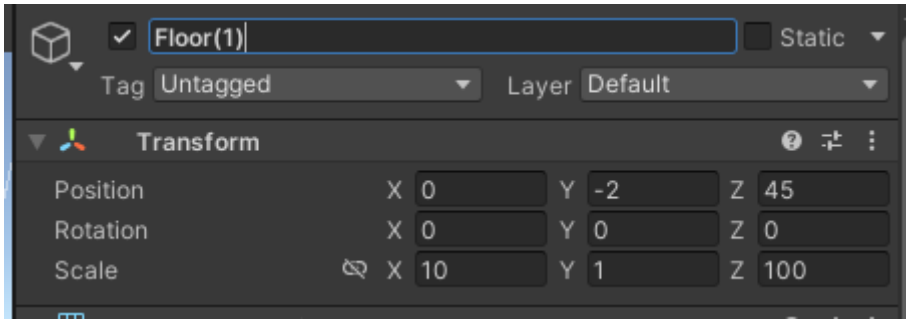
5. Der Level

Nun da wir eine Spielfigur haben, wollen noch einen Level bauen in dem sie sich bewegen kann.

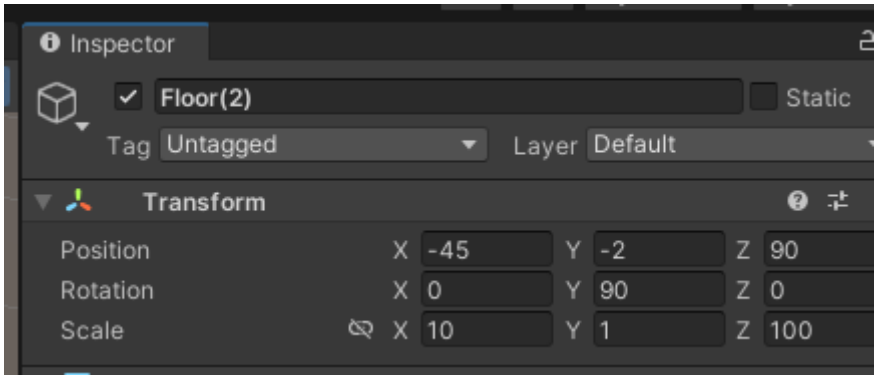
Dazu erstellen wir in der Hirachy unter „3D-Objects“ einen Cube.

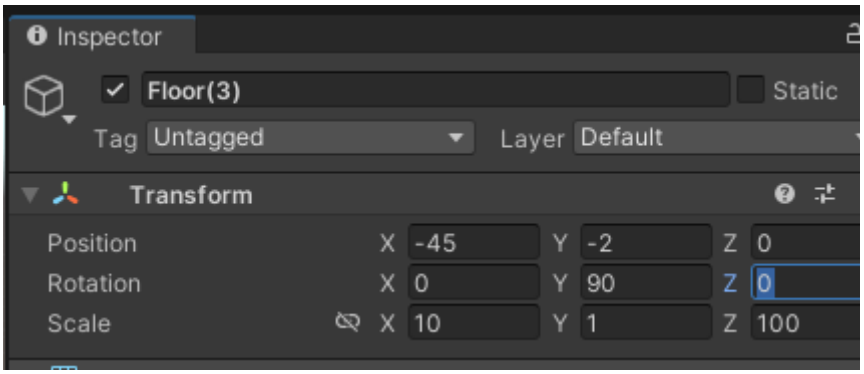


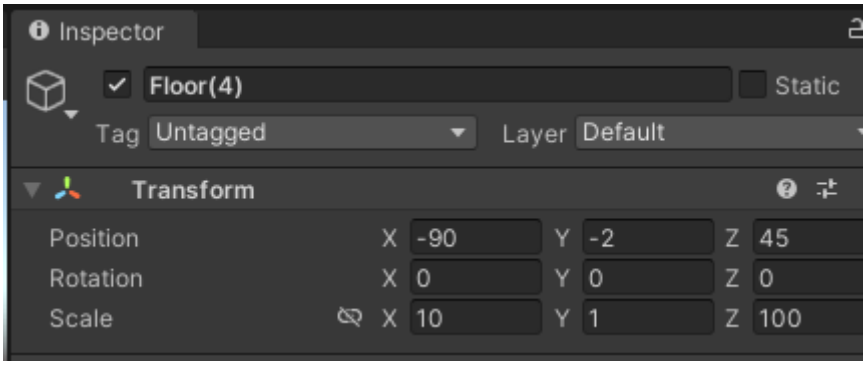
Diesem geben wir im Inspektor die unten angegebenen Werte und nennen ihn Floor(1).



Wir erstellen 3 weitere mit den unten angegebenen Werten und Namen.







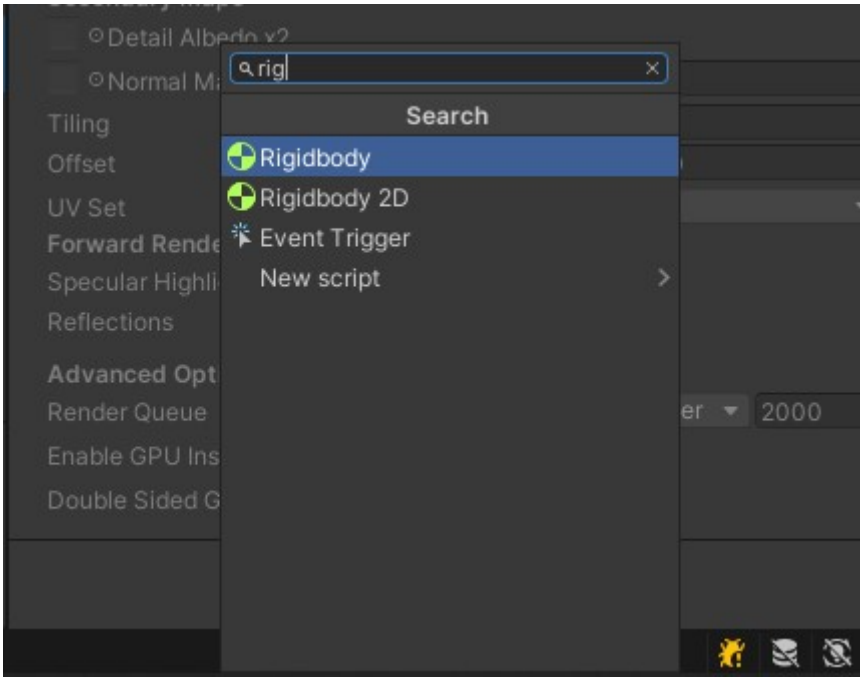
Nun haben wir einen spielbaren Level.

6. Game Over

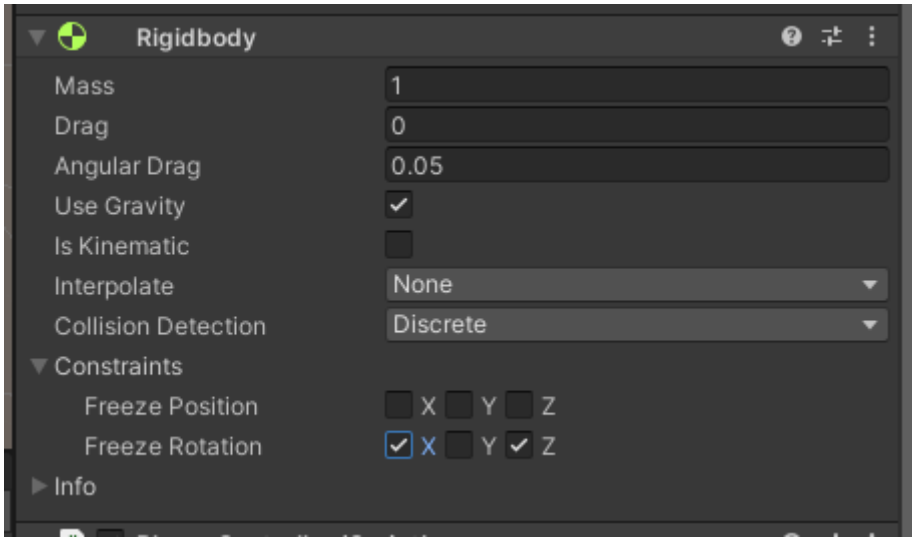
Da wir jetzt einen begehbaren Level haben, wollen wir unserer Spielfigur die Möglichkeit zu verlieren geben.
Dazu fügen wir dem PlayerController einen sogenannten Rigidbody hinzu. Damit ermöglichen wir ihm Schwerkraft und andere Physikberechnung.

```
[RequireComponent(typeof(Rigidbody))]
public class PlayerController : MonoBehaviour
{
    [SerializeField] private float _speed = 5.0f;
    [SerializeField] private Vector3 _moveDir = Vector3.zero;
    [SerializeField] private Rigidbody _rb = null;
```

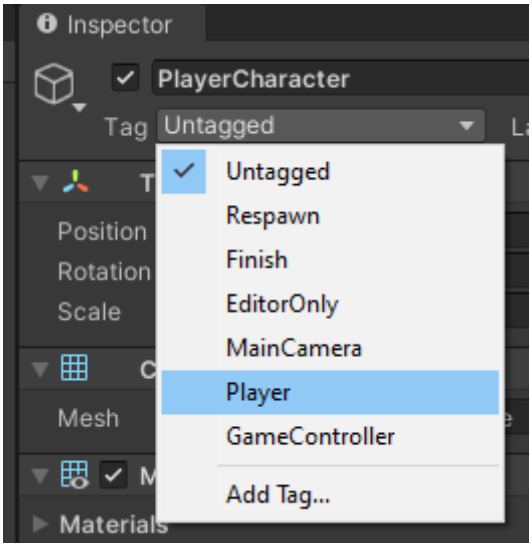
Den Rigidbody fügen wir dem PlayerCharacter mit AddComponent im Inspector hinzu.



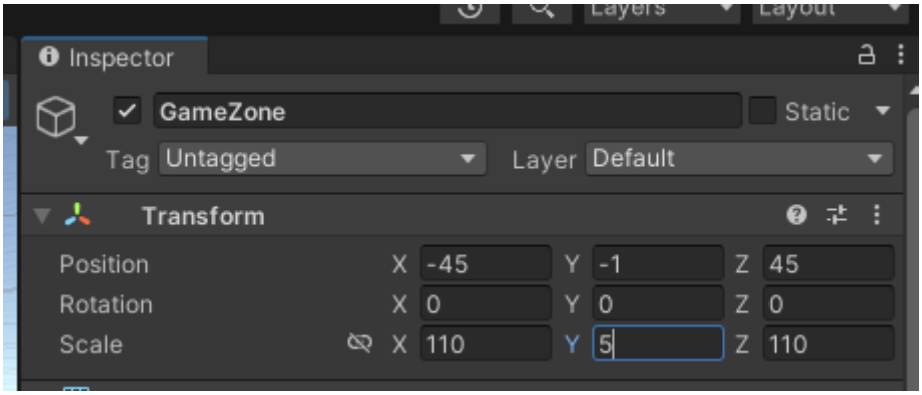
Bei der nun erstellten Rigidbody-Komponente müssen wir die Häckchen bei Freeze Rotation X und Z anklicken damit unsere Spielfigur zwar vom Rand des Levels fallen kann, aber nicht umkippen kann.



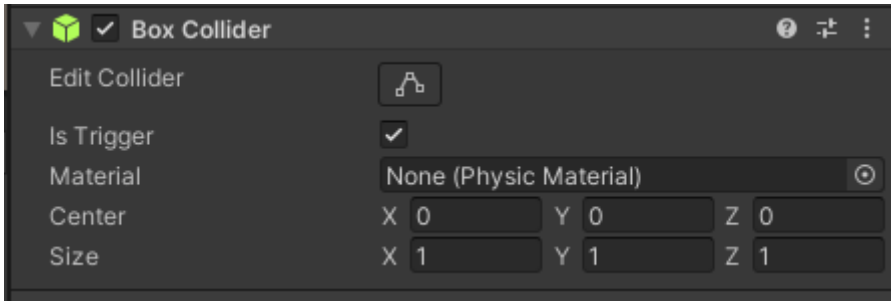
Nun müssen wir im Inspector des PlayerCharcters den Tag Player auswählen.



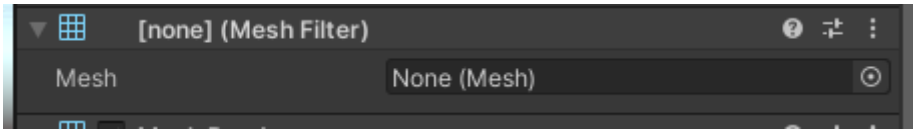
Nun erstellen wir einen weiteren Cube mit den folgenden Maßen und nennen ihn „GameZone“



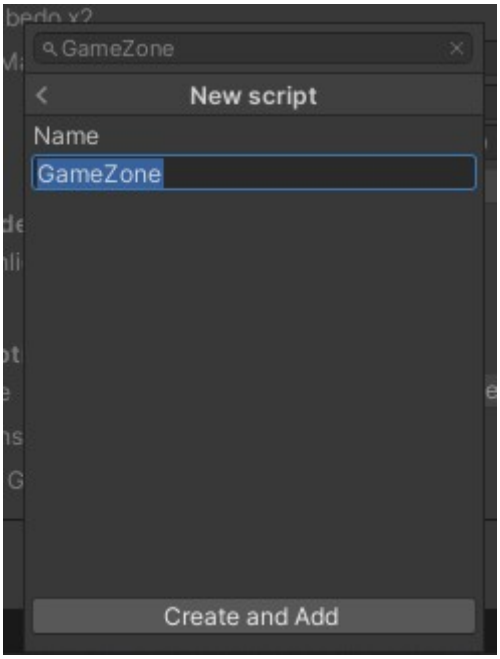
In seiner Box-Collider-Komponente muss der Button „Is Trigger“ mit einem Häckchen ausgewählt sein.



Sein Mesh-Filter muss auf None gestellt sein.



Zu guter Letzt bekommt unsere GameZone einen Script namens GameZone. Auch dieser kommt in den Scripts-Ordner.



Der C#-Script wird in VisualStudio um eine Methode erweitert.

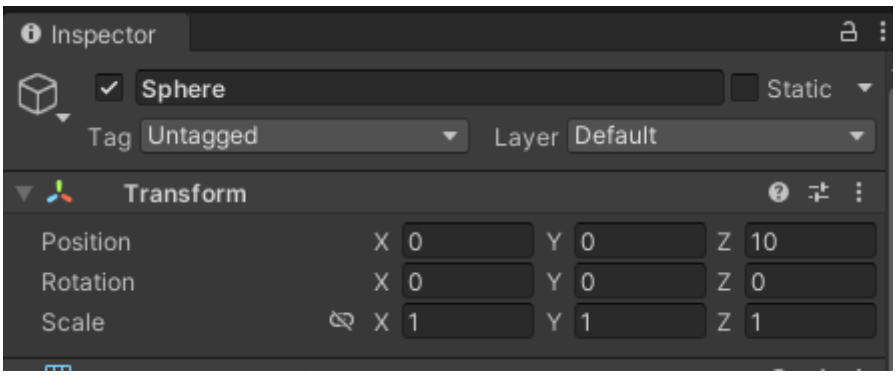
```
public class GameZone : MonoBehaviour
{

    private void OnTriggerExit(Collider other)
    {
        if(other.tag == "Player")
        {
            Destroy(other.gameObject);
            Application.Quit();
        }
    }
```

Nun wird die Spielfigur gelöscht und das Spiel beendet wenn die Spielfigur vom Spielfeld fällt.

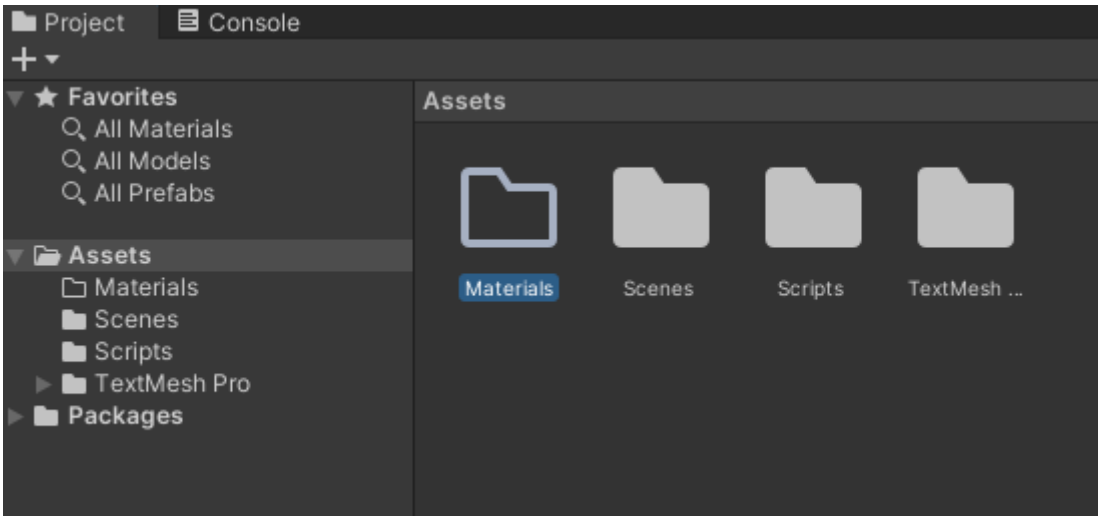
7.Items

Zuletzt bauen wir nun aufsammelbare Items.

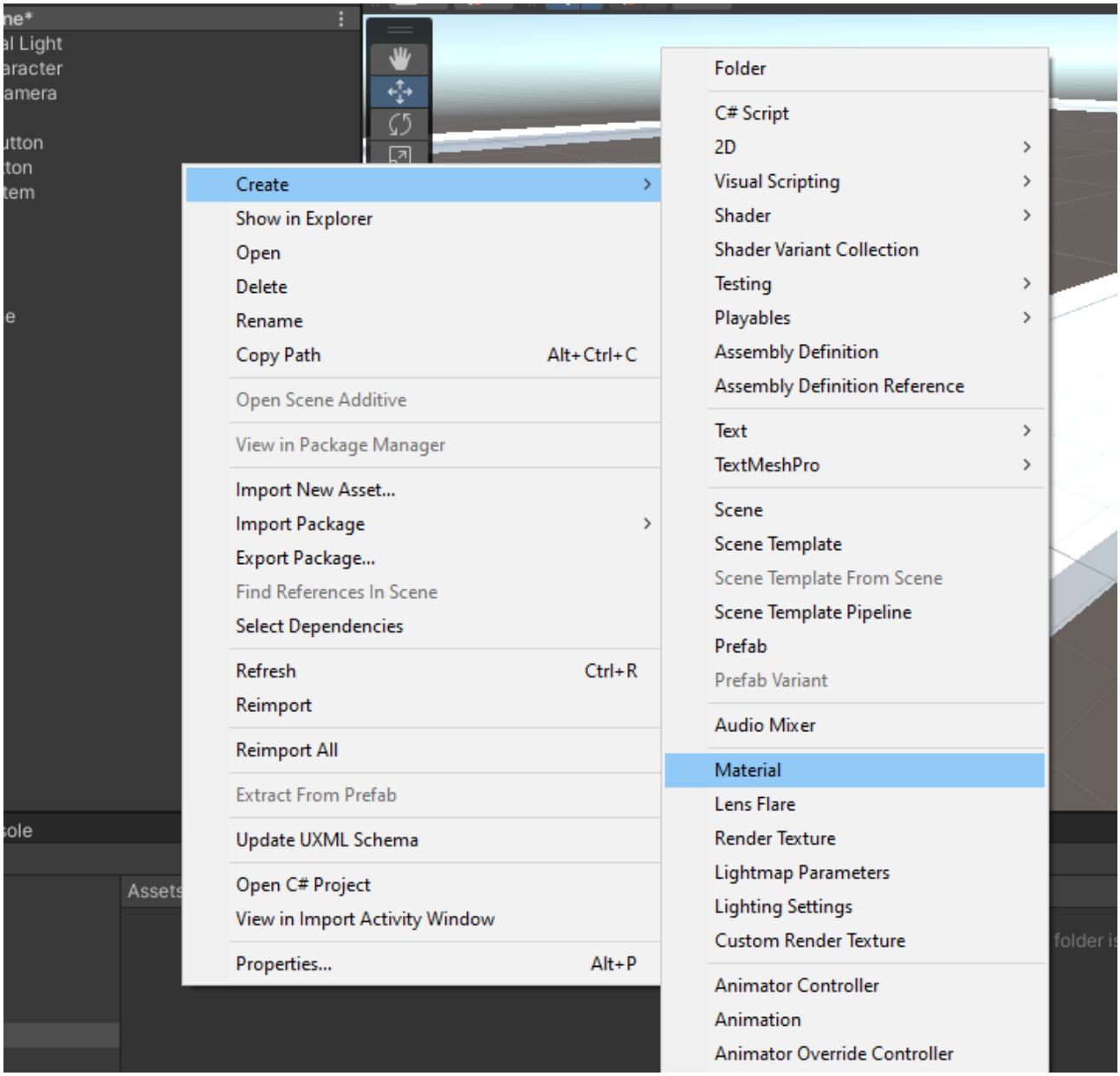
Dazu erstellen wir in der Hirachy eine Sphere (unter 3D-Objekten) mit den folgenden Werten:



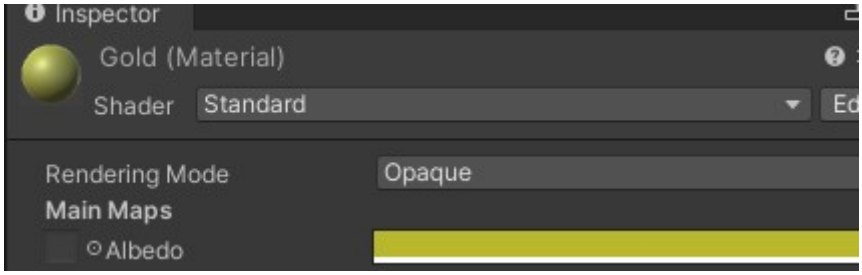
Wir erstellen im Project-Window einen neuen Ordner namens Materials.



In diesem erstellen wir mit rechtem Mausklick unter dem Reiter Create ein neues Material und nennen es Gold.

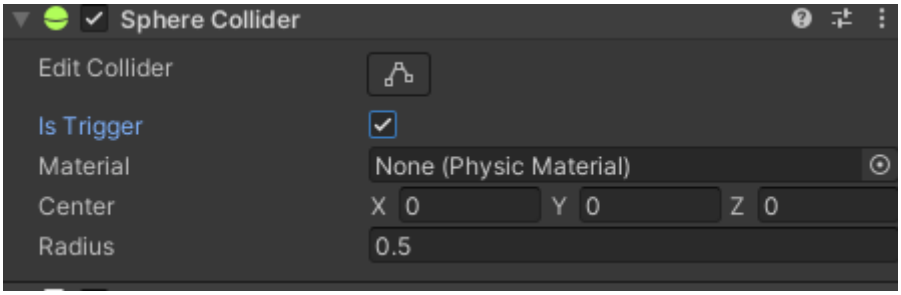


Im Inspector des Materials geben wir dem Albedo wert eine goldgelbe Farbe.



Danach ziehen wir das Material auf unsere Sphere in der Scene-View und benennen sie im Inspector zu „Item“ um.

Auch bei ihrem Sphere-Collider ist es wichtig das „IsTrigger“ auswählbar ist.



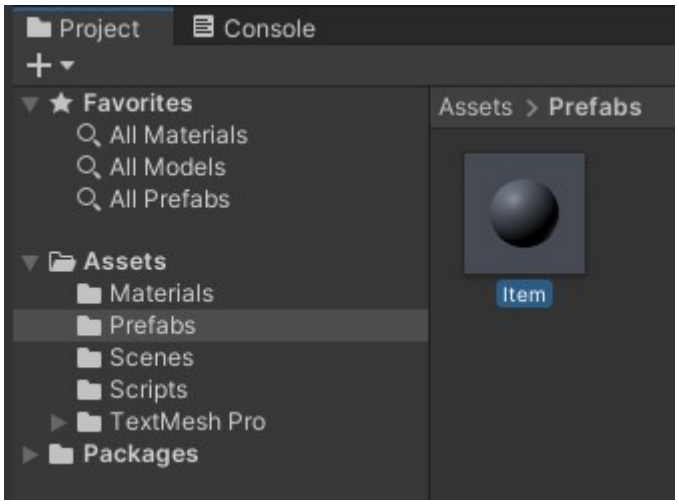
Sie bekommt mit AddComponent ein Script namens Item mit folgendem Inhalt:

```
public class Item : MonoBehaviour
{

    private void OnTriggerEnter(Collider other)
    {
        if(other.tag == "Player")
        {
            Destroy(gameObject);
        }
    }
```

Nun haben wir ein aufsammelbares Item.

Wollen wir mehr davon erstellen wir im Project-Window einen weiteren Ordner namens „Prefabs“.

Wir ziehen Item aus der Hirachy in den Prefab-Ordner.

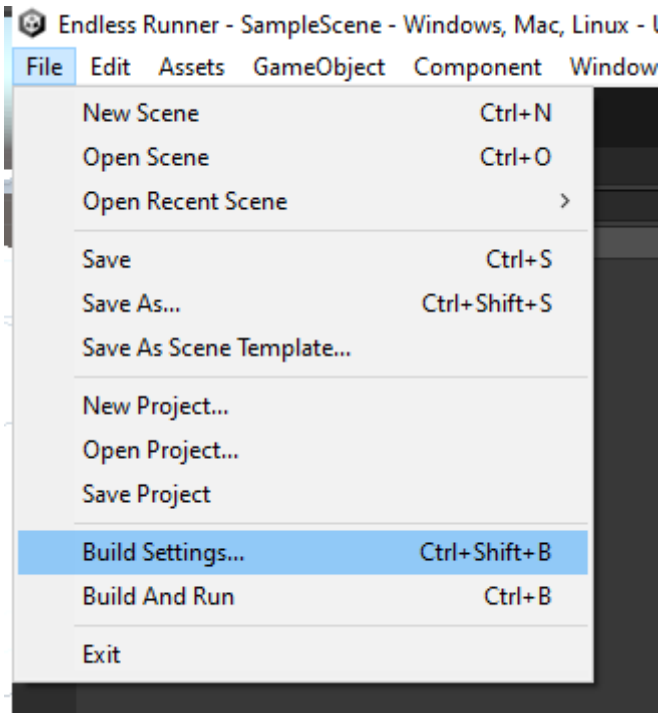


Nun haben wir ein vorgefertigtes Muster für Items, die wir nur noch in die Scene-View ziehen müssen, damit sie existieren.

8. Das Android-Game erstellen

Nun da wir ein fertiges Spiel haben, wollen wir es für Android builden.

Dazu wählen wir in der Menüleiste von Unity den Reiter File aus und dann Build-Settings.



Im nun geöffneten Fenster fügen wir die erstellte Szene mit Add Open Scenes hinzu. Dann entscheiden wir uns bei Platform für Android, drücken unten rechts auf den „Switch Platform“-Knopf. Nachdem Unity geladen und neu kompiliert hat, können wir mit Build eine .apk-Datei für Smartphone erstellen.