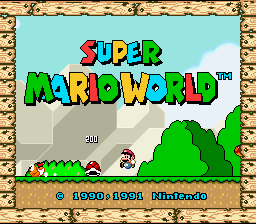
a

**SuperMarioVT – Projekt**

**Spielentwickler: Jeganivas Ulaganathan & Vassilios Tsiotras**

**Klasse: BK-TA14**



**Inhaltsverzeichnis**

Abbildungsverzeichnis 3

1 Einleitung 4

2 Planungsphase 5

2.1 Terminplanung 5

2.4 Ablaufplanung 6

3 Durchführungsphase 7

3.1 Aufgetretene Fehler 16

4 Projektabschluss 17

4.1 Projektübergabe 17

4.2 Abnahmeprotokoll 17

4.3 Projektbewertung 17

Quellenverzeichnis 18

# Abbildungsverzeichnis

Abbildung 1: Hintergrund-Büsche 7

Abbildung 2:Hintergrund Untergrundlevel 7

Abbildung 3: Hintergrund Levels 7

Abbildung 4: Sprite für Enemy-Pilz 8

Abbildung 5: Sprite für Coins 8

Abbildung 6: Sprite für Fragezeichen 8

Abbildung 7: Sprite für Pipes 8

Abbildung 8: Sprite für Mario 8

Abbildung 9: Sprite für Boden 9

Abbildung 10: Sprite für Ziel 9

Abbildung 11: Sprite für Rocket 9

Abbildung 12: Level-GUI Sprite Editor 9

Abbildung 13: Components Player 10

Abbildung 14: Animation-Controller Player 11

Abbildung 15: Components Enenemy Pilz 12

Abbildung 16: Components Coins 13

Abbildung 17: Layout Startmenü 14

Abbildung 18: Levels bilden/einbeziehen in Unity 14

Abbildung 19: Skript Movement.cs 16

Abbildung 20: Skript Movement.cs 16

Abbildung 21: Skript Movement.cs 17

Abbildung 22: Skript BetterJump.cs 20

Abbildung 23: Skript MovemenENbrcs 21

Abbildung 24: Skript MovementRocket.cs 22

Abbildung 25: Skript DestroyEnemys.cs 22

Abbildung 26: Skript Movement.cs 16

Abbildung 27: Skript Movement.cs 17

Abbildung 28: Skript Movement.cs 16

Abbildung 29: Skript Movement.cs 17

Abbildung 30: Skript Movement.cs 16

Abbildung 31: Skript Movement.cs 17

# Einleitung

Aufgabe des Projekts ist es ein Super Mario Spiel für einen Spieler bereitzustellen, der folgende Features besitzt:

• Spieler bekommt eine Startseite zum Spielstarten

• Spieler bekommt ein Super Mario Figur als sein Avatar

• Spieler bewegt sich mit den Pfeiltasten:

1. Leertaste=Zum springen

2. Linkstaste=Schritt nach links

3. Rechtstaste=Schritt nach rechts

• Spieler muss Coins sammeln, dies ist auch gleichzeitig sein Highscore ist

• Spieler wird mit „Gegenständen“ konfrontiert, die er ausweichen muss.

Ein Zusammenstoß führt zum Game Over.

• Es gibt ein Rohr indem der Spieler reingehen kann um Coins zu sammeln.

• Erreicht der Spieler das Ziel gewinnt der Spieler.

Um dieses Spiel zu programmieren verwenden wir die Umgebung Unity, weil im Unity die Spielumgebung herzustellen wesentlich einfach ist. Die Programmiersprache ist C#.

Da wir ein großes Interesse am Programmieren besitzen, entschieden wir uns direkt für ein Programmier-Projekt in der Richtung Game-Development. Da fiel uns das Klassikerspiel „SuperMario“ ein und dachten dies wäre ein passendes Projekt, das sehr umfangreich ist.

# Planungsphase

## Terminplanung

Abgabe Themenvorschläge: 17.10.2018

Themen besprechen: 24.04.2018

Abgabe Projektanträge: 07.11.2018

Rückgabe und Besprechung Projektanträge: 14.11.2018

Abgabetermin Endfassung Projektanträge: 21.11.2018

Projektgenehmigung: 28.112018

Ende Planungsphase: 05.12.2018

Ende der Planungsphase: 05.12.2018

Beginn der Projekt Durchführungsphase: 12.12.2018

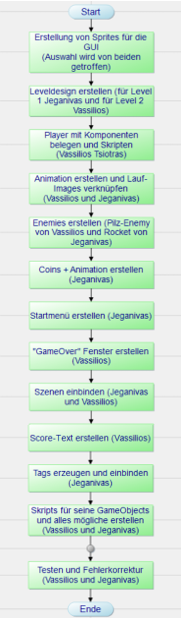
Ende Projekt Durchführungsphase: 27.03.2018

Abgabetermin Projektdokumentation: 10.04.2018

Bekanntgabe Dokumentationsergebnisses 17.05.2018

Projektpräsentation 27.05.2018

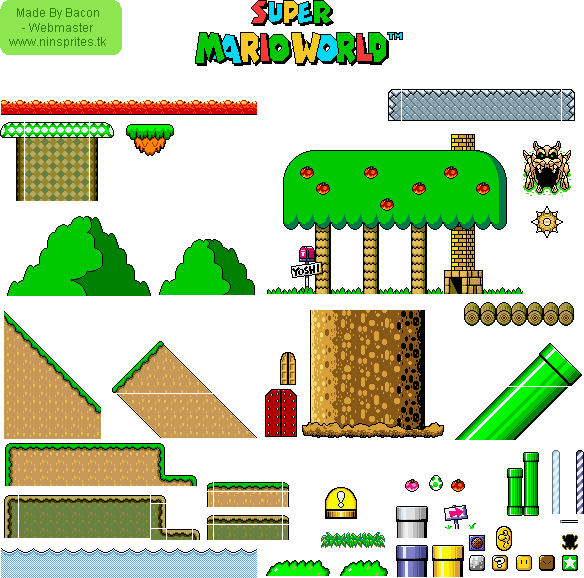
## Ablaufplanung



# Durchführungsphase

Schritt 1: Sprites

Für das Projekt SuperMario, das in einer Game-Engine, Unity, erstellt wird braucht man natürlich zuerst Objekte, Hintergründe etc. Die aus Images bestehen. Auf der Seite <https://www.spriters-resource.com/snes> haben wir die einzelnen Images gefunden:



Hintergrund-Büsche

Abbildung 1

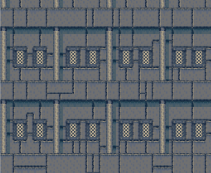


Abbildung 3

Abbildung 2

Hintergrund für Untergrundlevel Hintergrund für die Levels



Abbildung 4

Coins um die Animation zu erstellen

Abbildung 5

Enemy-Pilze um die

Animation zu erstellen



Abbildung 7



Abbildung 6

Pipes

Fragezeichen um die Animation zu erstellen

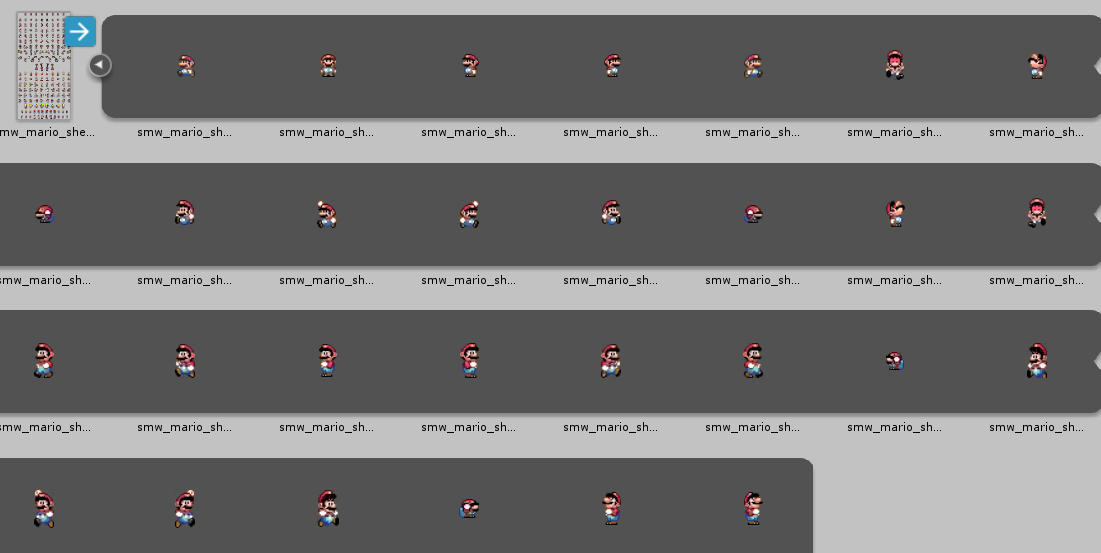


Abbildung 8

SuperMario um die Animationen zu erstellen.



Abbildung 10

Abbildung 9

Ziel

Boden



Abbildung 11

Rackete+Erstellung von Animation

Schritt 2: Level-GUI

Als nächstes haben wir unsere Grafische Oberfläche für unsere Levels erstellt. Müssten dafür die Images als Sprites „erzeugen“. Dazu erstellt man dank Unity ein leeres GameObject und fügt das Bild hinein und öffnet den Sprite Editor und schneidet sein gewünschtes Bild zurecht und fügt es in der Spielumgebung ein. Und fügt natürlich bei Boden, Pipes etc. Colliders hinzu damit der Spieler nicht runterfällt.

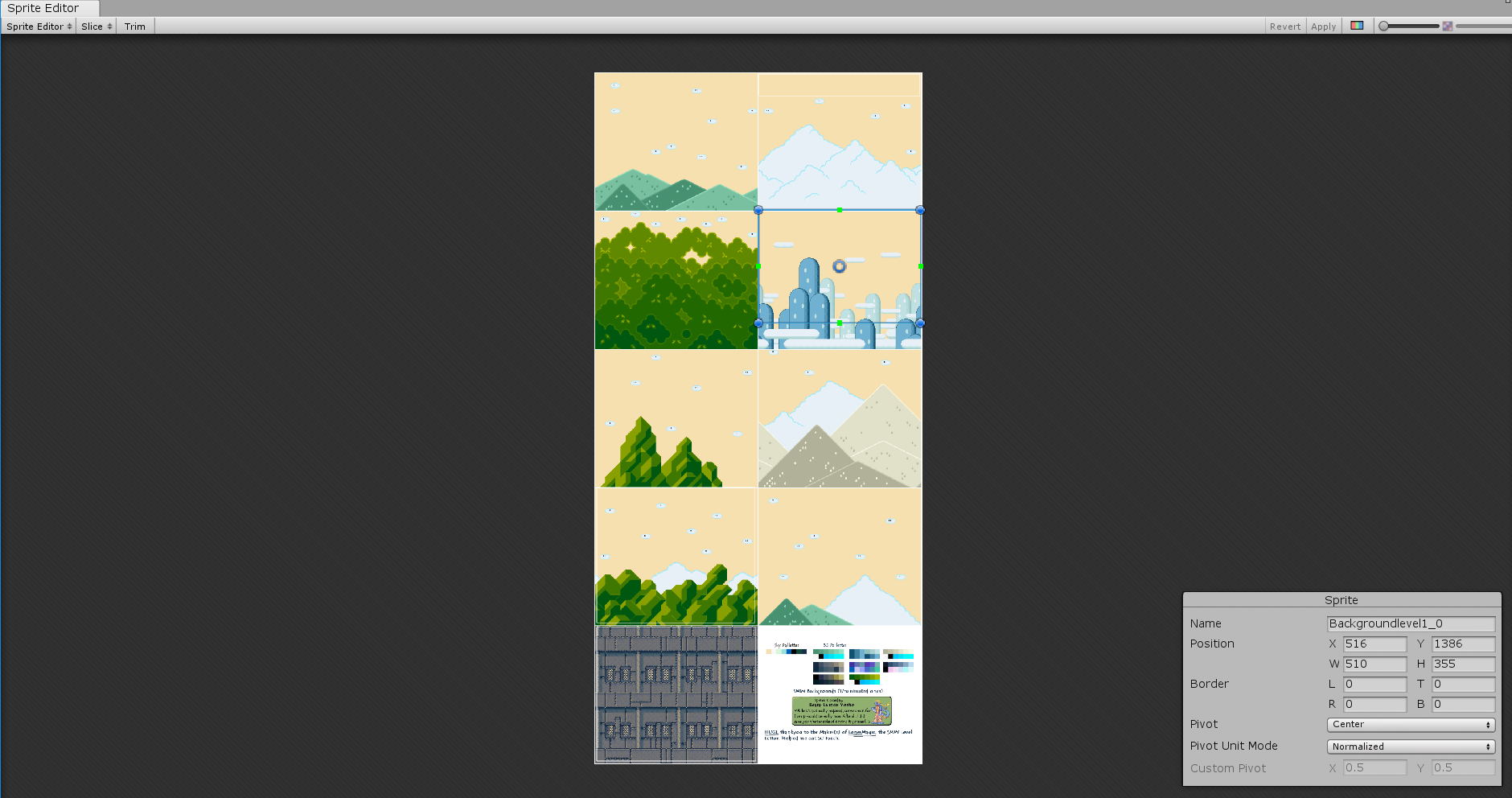
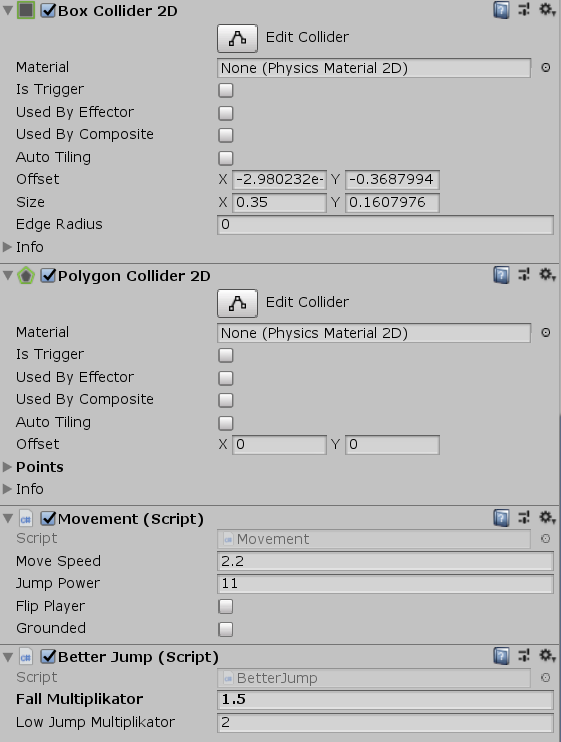


Abbildung 12

Schritt 3: Player (Mario)

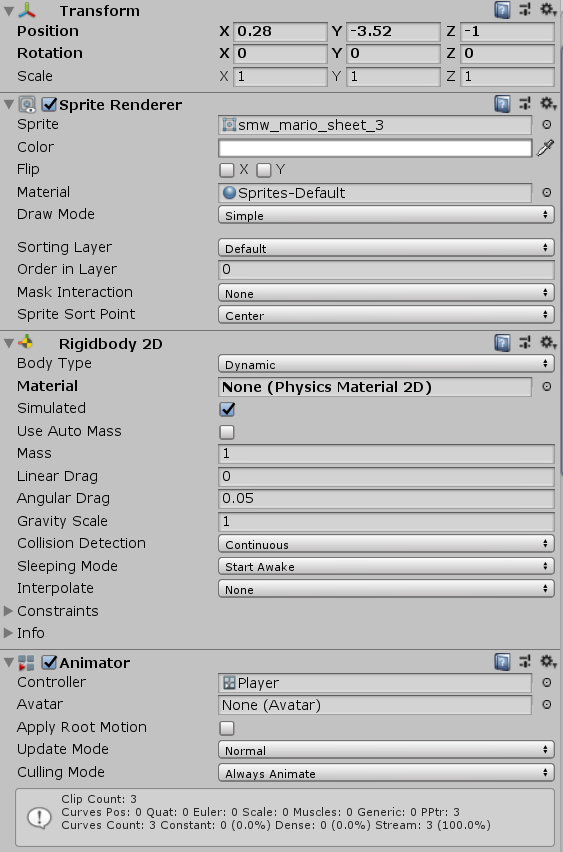
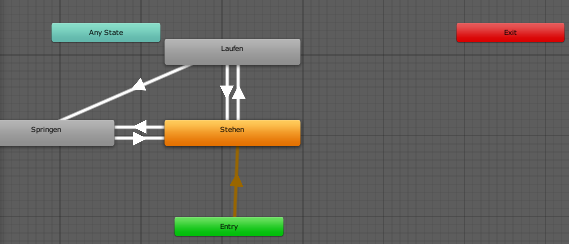
Schritt 4: Animation-Controller (für Player)

Abbildung 13

**Box Collider2D:** Player kriegt einen Box Collider damit er nicht durch die Gegenstände durchlaufen kann (Wichtig!: Das Gegenstand muss auch einen Collider besitzen.)

**Polygon Collider2D:** Hat die selbe Funktion wie Box Collider, allerding kann man diesen genauer anpassen an das Objekt.

**Movemement (Script):** Script für Bewegung und zerstören von Enemies.

**Better Jump (Script):** Proportionale Sprung Funktion.

**Sprite Renderer**: Rendert ein Sprite für 2D-Grafiken.

**Rigidbody 2D:** Unser Spieler bekommt physikalische Eigenschaften (wie zb. Schwerkraft).

**Animator:** Animator dient für das Animation-Controller. Der Conroller dient zur Bewegung, bzw. Image wechsel, für den Player.

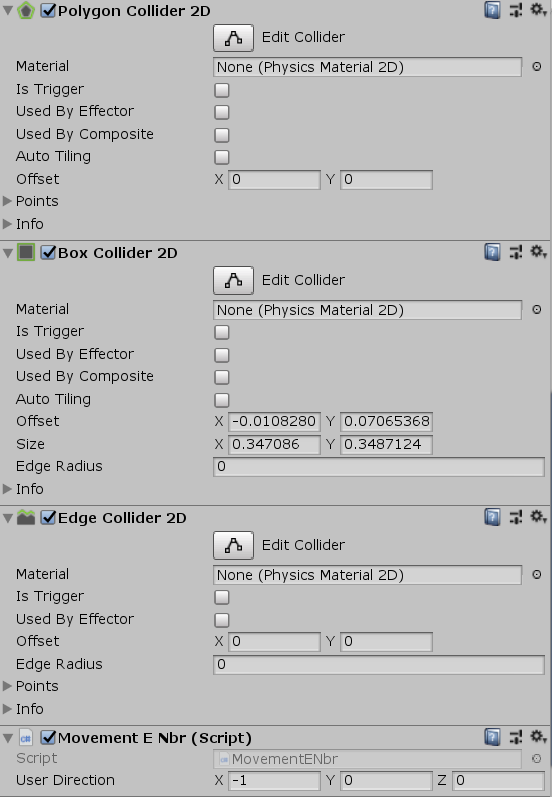
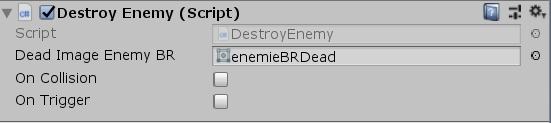
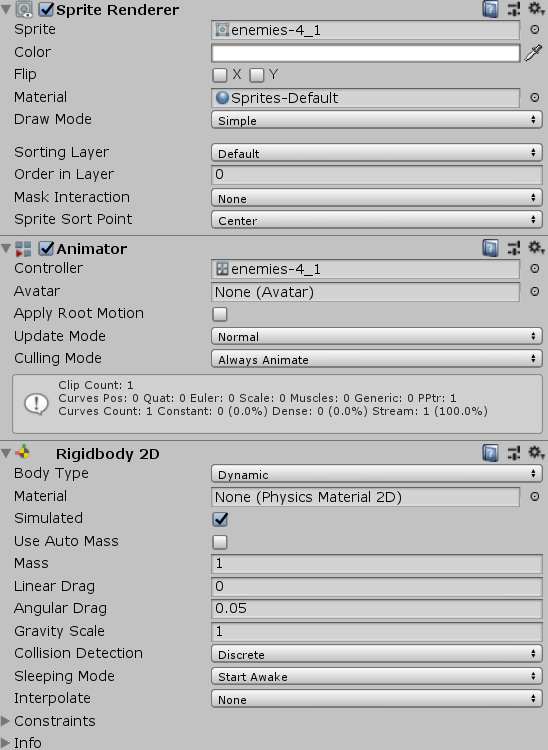
Abbildung 14

Animationen, die zur Verfügung stehen: Stehen, Laufen (Rechts, Links), Springen. Während er steht wird die Steh-Animation durchgeführt, wenn er während stehen springt wird die Spring-Animation durchgeführt. Wenn die Rechts oder Links-Taste betätigt wird die Animation von Laufen durchgeführt. Während laufen kann der Spieler auch Springen. Wenn der Spieler keine Tasten betätigt wird natürlich die Steh-Animation durchgeführt.

Laufen besitzt die Variable „Speed“, dies dient dazu um zu entscheiden ob der Spieler nach rechts oder links läuft. Und Springen besitzt die Variable „Springen“. Weiteres wird im Skript-Bereich beschrieben.

Schritt 5: Enemy-Pilz

Als nächstes werden Enemies erstellt. Indem fall unser Pilz-Enemy.



**Sprite Renderer:** Rendert ein Sprite für 2D-Grafiken.

**Animator:** Animator dient für das Animation-Controller. Der Conroller dient zur Bewegung, bzw. Image wechsel, für den Enemy.

**Rigidbody 2D:** Unser Pilz bekommt physikalische Eigenschaften (wie zb. Schwerkraft).

**Polygon Collider2D:** Hat dieselbe Funktion wie Box Collider, allerding kann man diesen genauer anpassen an das Objekt.

**Box Collider2D:** Player kriegt einen Box Collider damit er nicht durch die Gegenstände durchlaufen kann (Wichtig! Das Gegenstand muss auch einen Collider besitzen.)

**Edge Collider:** Eine unsichtbare Form, mit der physische Kollisionen für ein Objekt verarbeitet werden.

**Movement E Nbr (Script):** Skript für das Bewegen von Pilz. Die X-Achse ist auf – gesetzt, weil unser Pilz sich nur links bewegt.

**Destroy Enemy:** Image Wechsel bei zerstören von Pilz durch Mario.

Abbildung 15

Schritt 6: Enemy-Rocket

Bei Enemy-Rocket ist der Aufbau genau gleich wie Pilz, nur es besitzt kein Destroy Enemy (Skript) und kein Edge Collider. Dafür hat unser Rocket einen kinematischen Körper, damit er in der Luft sich bewegen kann.

Schritt 7: Coins

**Sprite Renderer:** Hat immer noch die gleiche Funktion, wie zuvor auch.

**Animator:** Da ist die Animation für das Drehen von Coins.

**Capsule Collder2D:** Ist ebenfalls ein Collider aber in einer Kreisform.

**Coins(Skript):**Funktion wann Coin verschwindet und somit der Score hochzählt.

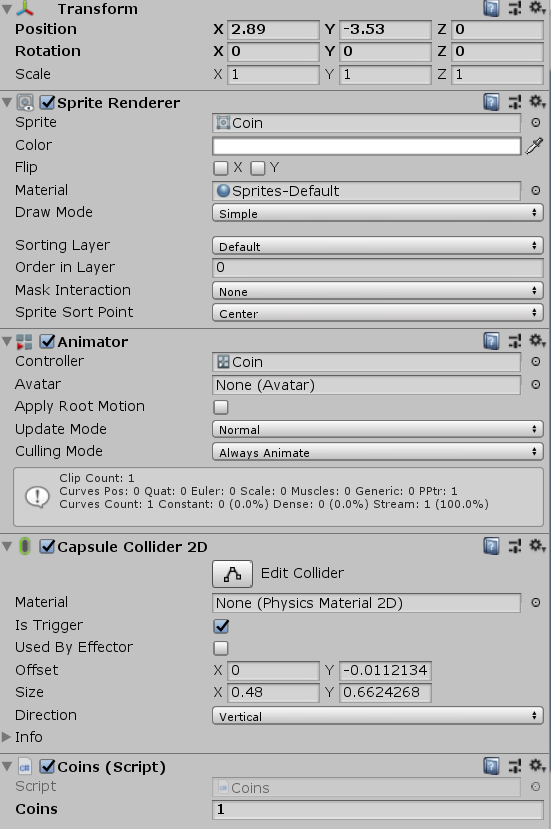


Abbildung 16

Schritt 8: Startmenü/GameOver



Abbildung 17

Unser Startmenü besteht aus einem Hintergrundbild, Textfeld (in Unity: TextMeshPro)

und aus 2 Buttons:

* START: Da wird die erste Szene abgespielt indem fall Level1.
* QUIT: Passiert nichts.

Beim GameOver hat der Spieler eine Wahl, das Spiel zu neustarten oder zu beenden.

Schritt 9: Szenen einbinden

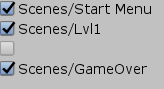
Damit man Szenen switchen kann, muss man die zuerst in Unity Bilden bzw. einbinden:

Abbildung 18

Levelswitches sind vorhanden in: Startmenü, Enemies (GameOver), Pipes (für Untergundlevel), Ziel (1. Von Level 1 zu Level 2; 2. Endziel „EndGame“).

Alles andere wird natürlich geskriptet, was und wann die Szene geswitcht werden soll, beim Drücken eines Buttons oder durch eine Kollision mit GameObject. Dazu erfahren Sie mehr im Skript-Teil.

Schritt 10: Score

Unser Score besteht (Oberflächlich) aus einem Textfeld der durch einen Canvas oben links positioniert ist. Score ist am Anfang gleich 0 gesetzt. Das Textfeld ändert sich bei jedem einsammeln von Coins.

Schritt 11: Tags

Bestimmte GameObjects besitzen Tags, die wir selbst benennen können. Dies vereinfacht das Skripten von unseren Player. Damit man nicht jedes Mal alles neu schreiben muss, wie zb. Destroy bei Coins und Enemies, kann man ganz de Spieler taggen und somit sagen was passiert, wenn ein Objekt mit diesem Objekt (mit der tag) zusammenprallt.

Player-Tag: EnemyBRColl (die Funktion wird im Skript beschrieben)

Boden und Pipes: GroundColl (die Funktion wird im Skript beschrieben)

Schritt 12: Skripts

Das wichtigste was für das Spiel nötig sind natürlich die Skripts. Dank Unity konnten wir die Charaktere und Levelumgebung erstellen, aber durch das Skripten wird es natürlich erklärt wie und was, wann passieren sollte. Die Skripts erklären auch wie und wann die Komponenten benutzt und modifiziert werden:

**Movement.cs:**

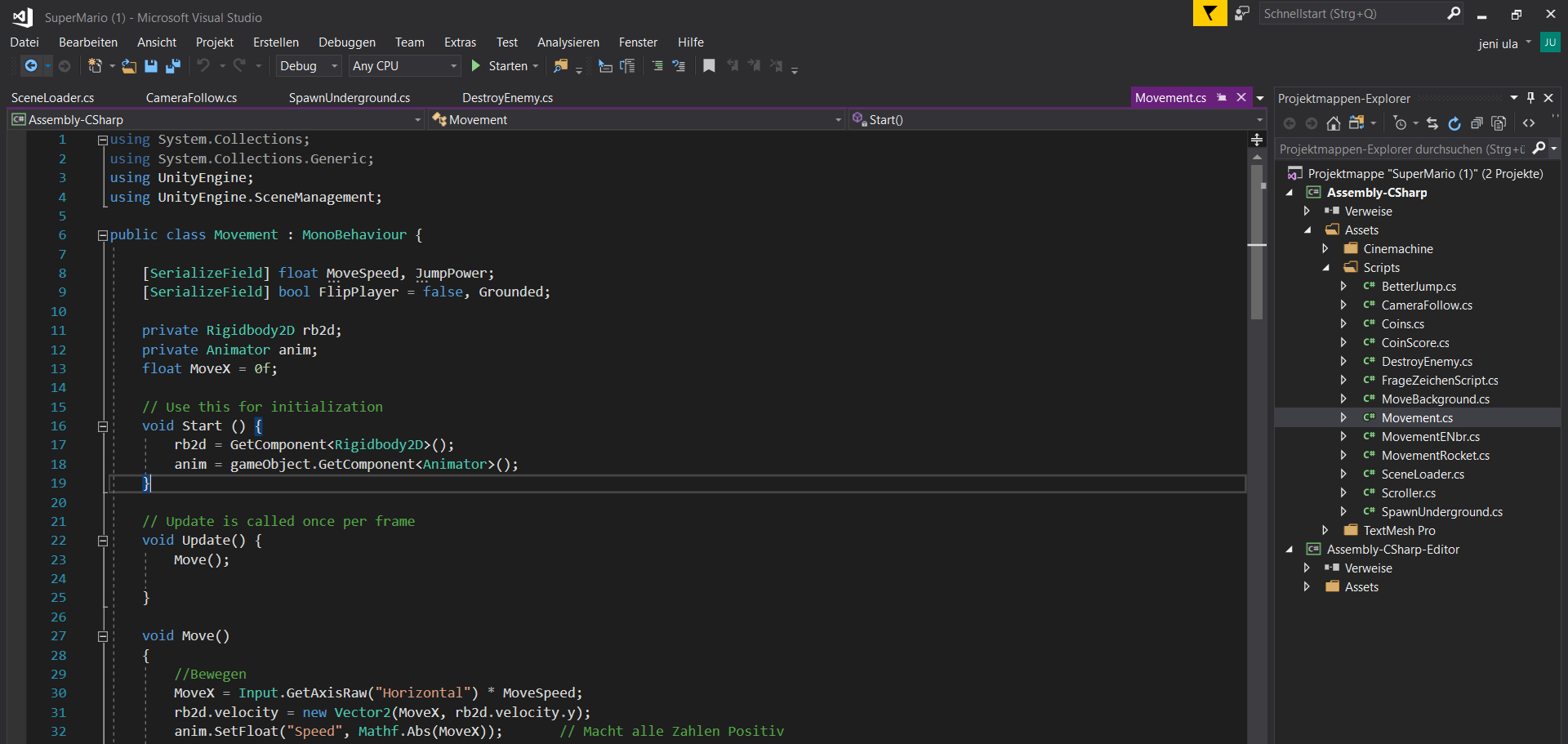


Abbildung 19

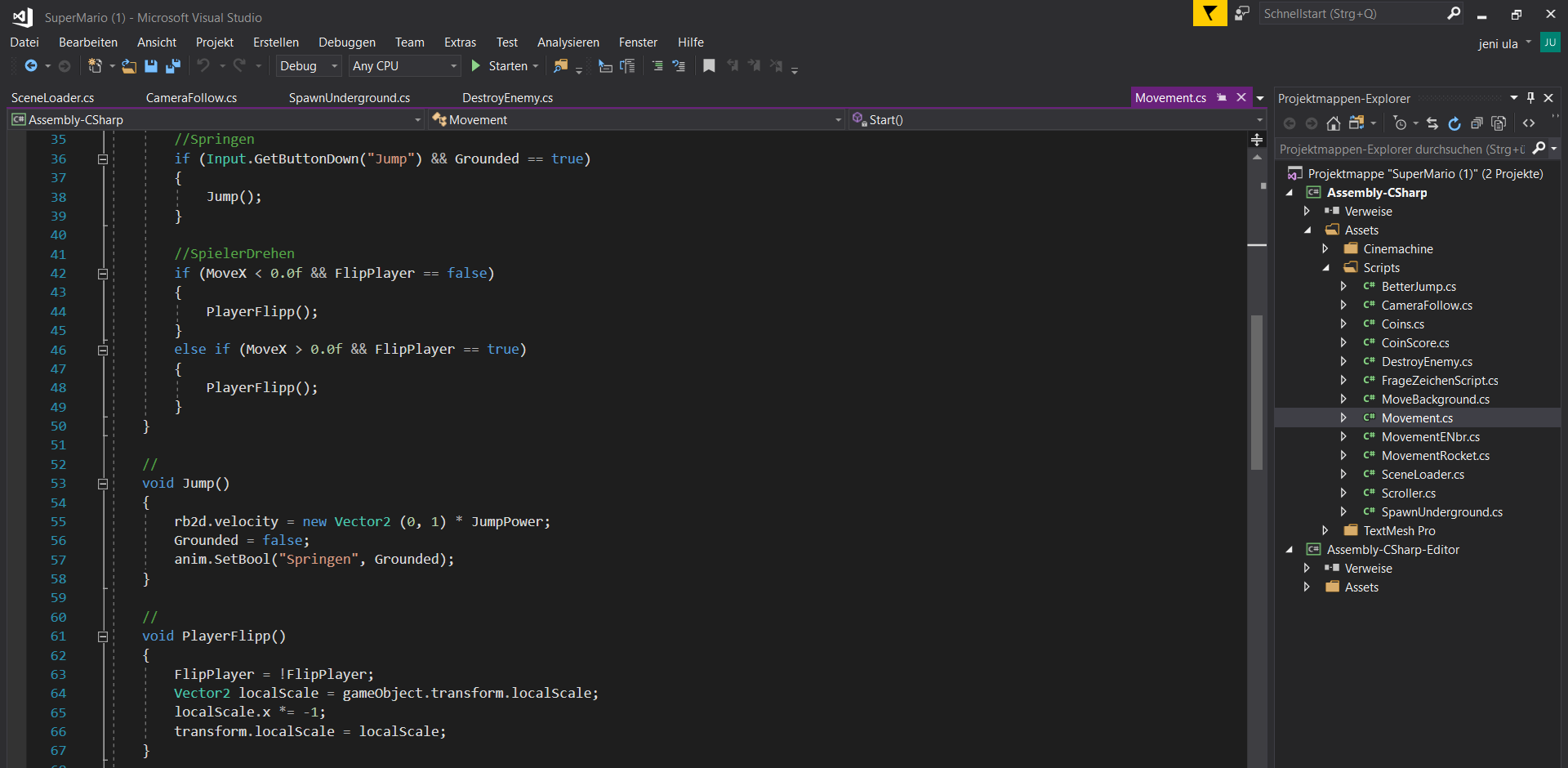


Abbildung 20

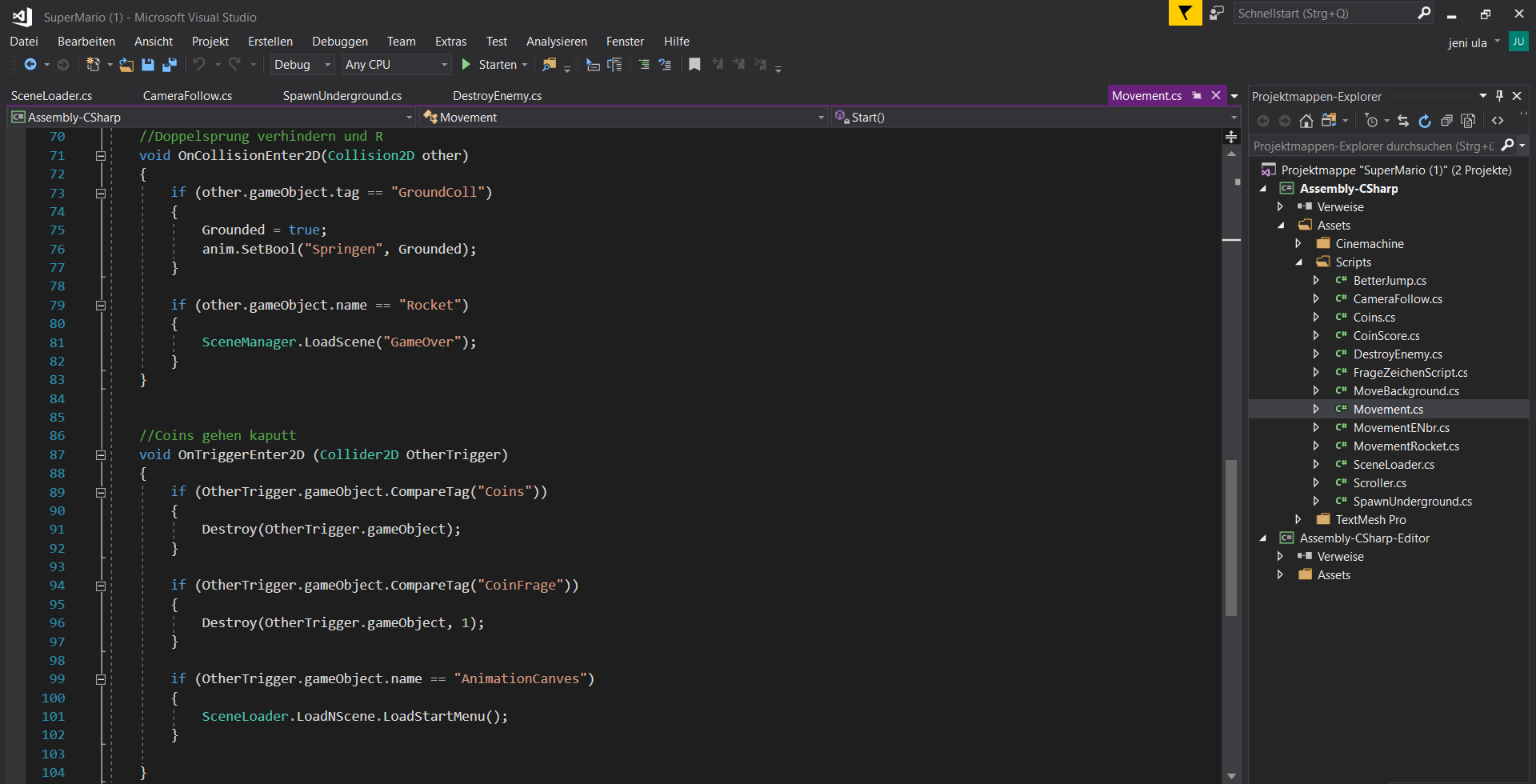


Abbildung 21

**SerializedField:** Wird verwendet um die Variablen in der Game-Engine zu sehen und sie dort anzupassen. (Bspw. Den MoveSpeed ohne jedes mal in den Script zu ändern, einfach über die Engine variieren kann).

Variable Grounded als Boolien um Später dann zu checken ob der Spieler gelandet ist oder nicht (True/false).

**Rigidbody2D-Klasse** (Komponente eines Obejktes, falls benötigt) dient für die Physik des Objektes.

**Animator-Klasse** (Komponente) dient für den Animator-Controller um diesen im Script später dann richtig zu Codieren, damit die Animationen richtig funktionieren.

**Funktionen:**

**Start():** Das Programm geht zu aller erst in diese Funktion um die Initialisierungen zu ergreifen.

-rb2d: Die Komponente Rigidbody2D wird in rb2d gespeichert. Im späteren Vorgang ist so nicht mehr nötig jedes Mal „gameObjekt.GetComponent<Rigitbody2D>()..“ zu schreiben.

-Anim: nimmt die Klasse / Component wie das Rigidbody nur hier wird der Animator genommen.

**Move():**

**// Bewegen**

-Getaxis: Von Unity vorgegeben "Horizontal" / "Vertical" x und y Achse. Tastenbelegung kann man selber anpassen. Der Wert liegt für die Eingabe von Tastatur im Bereich -1 / 1, dieser Wert wird dann mit dem MoveSpeed ​​Multipliziert um die Schnelligkeit Dies wird in MoveX gespeichert.

-rb2d.velocity: Lineare Geschwindigkeit des Objekts.

-Vector2 wird verwendet die Positionen darzustellen (x, y).

-anim.SetFloat: SetFloat wird verwendet, um Float-Werte an den Animator zu senden,um Übergänge zu aktivieren. So wird als beispiel die Werte übergeben, damit der Animator weiss ob er jetzt läuft oder rennt. es gibt auch weiter wie SetBool, SetInteger usw. Im Animator muss man dann die Werte definieren, welche Animationen beinflussen dürfen.

**// Springen:**

GetButtonDown ("Jump") Jump ist von Unity als Name vorgegeben und den Input kann man ebenfalls anpassen.

**Jump ():** rb2d.velocity: Hier wird diesmal für die y-Achse verwendet. Eingedrückt: wird auf false gesetzt. SetBool: selbes Prinzip wie bei SetFloat nur als Bool-Wert.

**FlipPlayer ():** Diese Funktion wechselt nur das Image / sprite des Players. Ansonsten hätte man vom Player die Bilder doppelt bzw. für links und rechts tragen müssen und auch doppelte Animationen erstellen müssen.

-Vector2 Localscale: Dort wird die aktuelle Position des Players verwendet und wird dann auf -1 / + 1 gesetzt um ihn zu drehen.

-Transform: Bekommt jedes Objekt in Unity, diese beinhaltet Position, Drehung und Skalierung.

**OnCollisionEnter:** -Wird gesendet, wenn ein ankommender Collider mit dem Collider dieses Objekt ist in Kontakt kommt. Dort kann man codieren was passieren soll, wenn die zusammenstoßen. Informationen werden über den übergebenen Parameter angegeben. hier wird das andere Objekt geprüft Collider vom Objekt, welches ein Tag hat (GroundColl), zusammenstößt. ist das der Fall wird auf den richtigen Ort gesetzt und der Spieler kann wieder springen.

Im nächsten Schritt wird nach dem Namen des anderen Objektes gesucht, dies fällt zusammen verliert dieser Spieler.

-SceneManager: hilft von Szene zu Szene zu wechseln.

**OnTriggerEnter:** macht dasselbe wie CollisionEnter, nur das die anderen Collider auf Trigger gestellt wurden. Trigger bedeutet, dass man durch laufen oder drinne stehen kann im anderen Collider, da gibt es keine "Wand".

-Hier werden die Tags von den Coins benutzt um den Coin zu zerstören in Berührung kommt. So stellt es da diese diese aufgesammelt wurden. Dasselbe Prinzip hat auch die Coins die von Fragezeichen kommen, diese werden dann erst nach einer Sekunde zerstört.

**BetterJump.cs:**

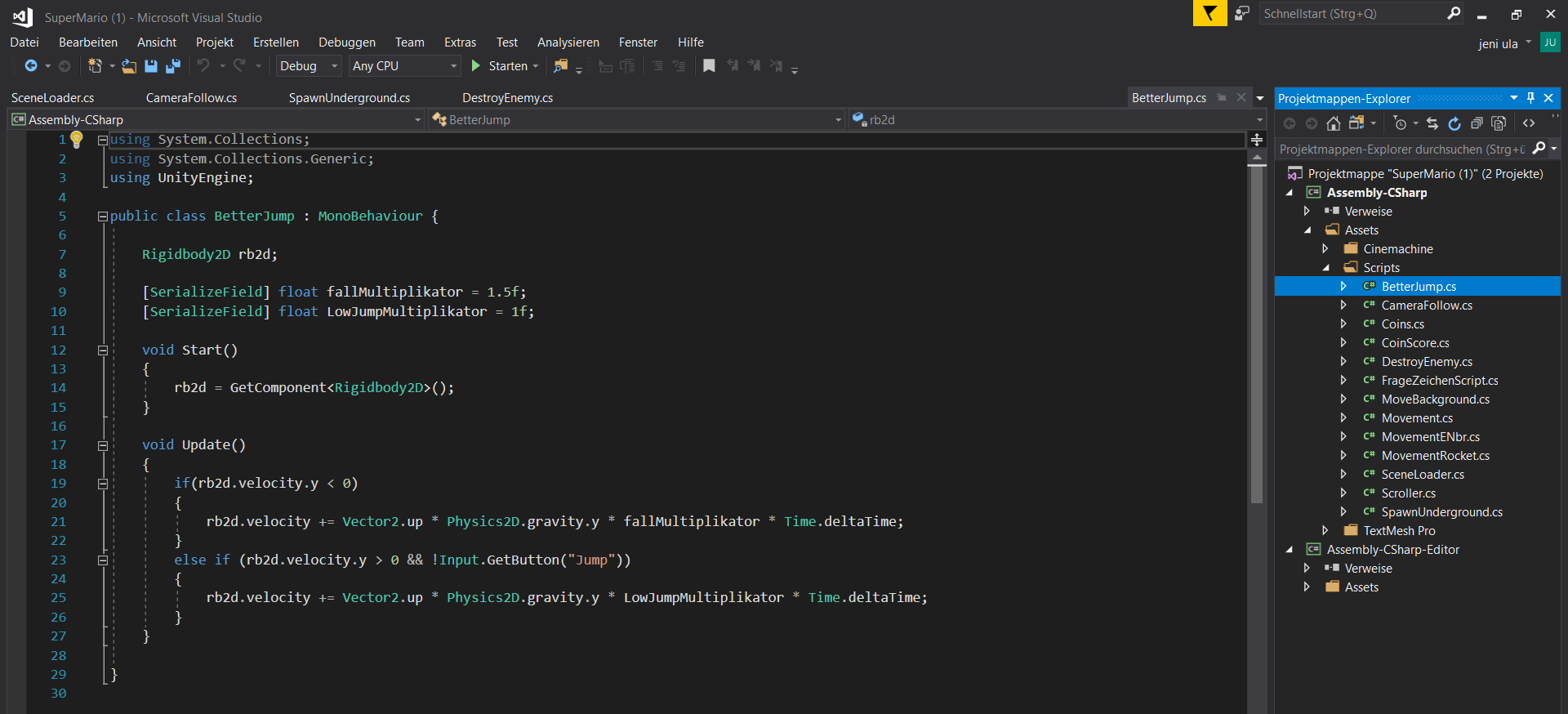


Abbildung 22

-Time.deltaTime: Ist die Fertigstellungszeit in Sekunden seit dem letzten Frame. Mann multipliziert das mit dem fallmultiplikator und den vector2.up macht es eine Bewegung in yRichtung. Auch nEinheiten (fallmultiplikator) \* Vector2.up (y-Richtung) \* Time.DeltaTime.

-die Physik2D.gravity.y ist für die Gravitation in y Richtung, diese wird hinzugefügt um das gefallener zu machen. Geprüft wird hier die Richtung höher oder kleiner null wird dementsprechend das spring und Fall Verhältnis zu bestimmen.

**MovementENbr.cs:**

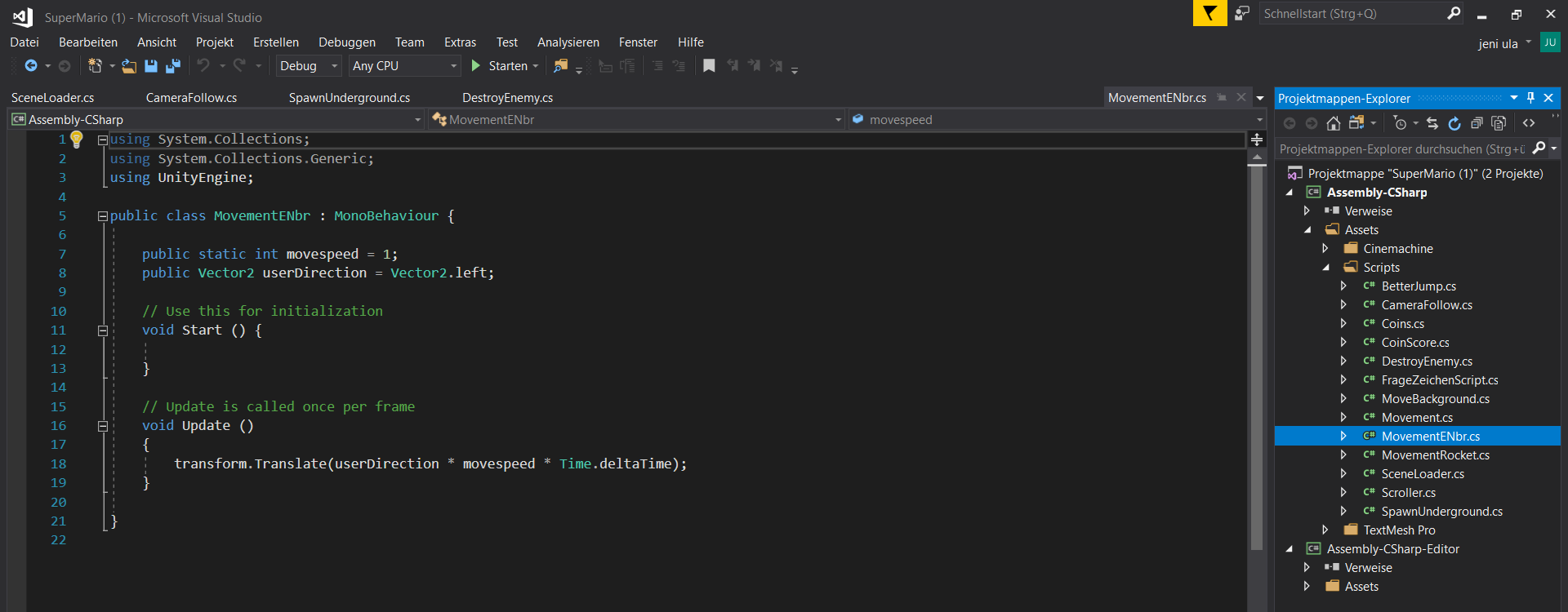


Abbildung 23

**MovementENbr ():** vector2.left ist eine kurze schreibweise von "new vector2 (-1,0)" welche in userDirection eingepspeicher wird. In der Update-Funktion wird durch transform.translate der Braune Pilz nach links bewegt.

**MovementRocket.cs:**

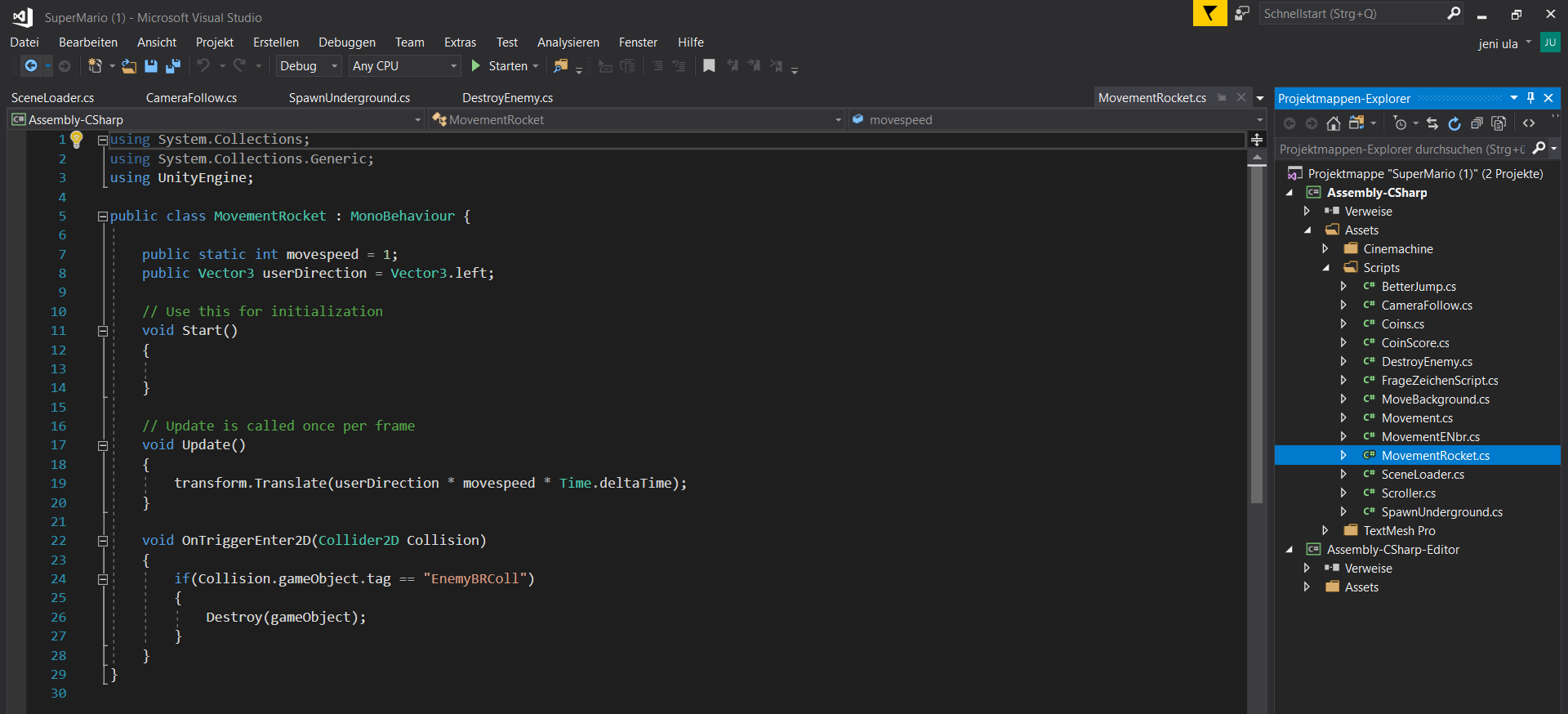


Abbildung 24

**Update():** Die Updatefunktion ist die Selbe wie bei Braunen Pilz (MovementENBr).

Hier kommt noch dazu, wenn der Spieler auf die Rakete drauf springt.

**DestroyEnemy.cs:**

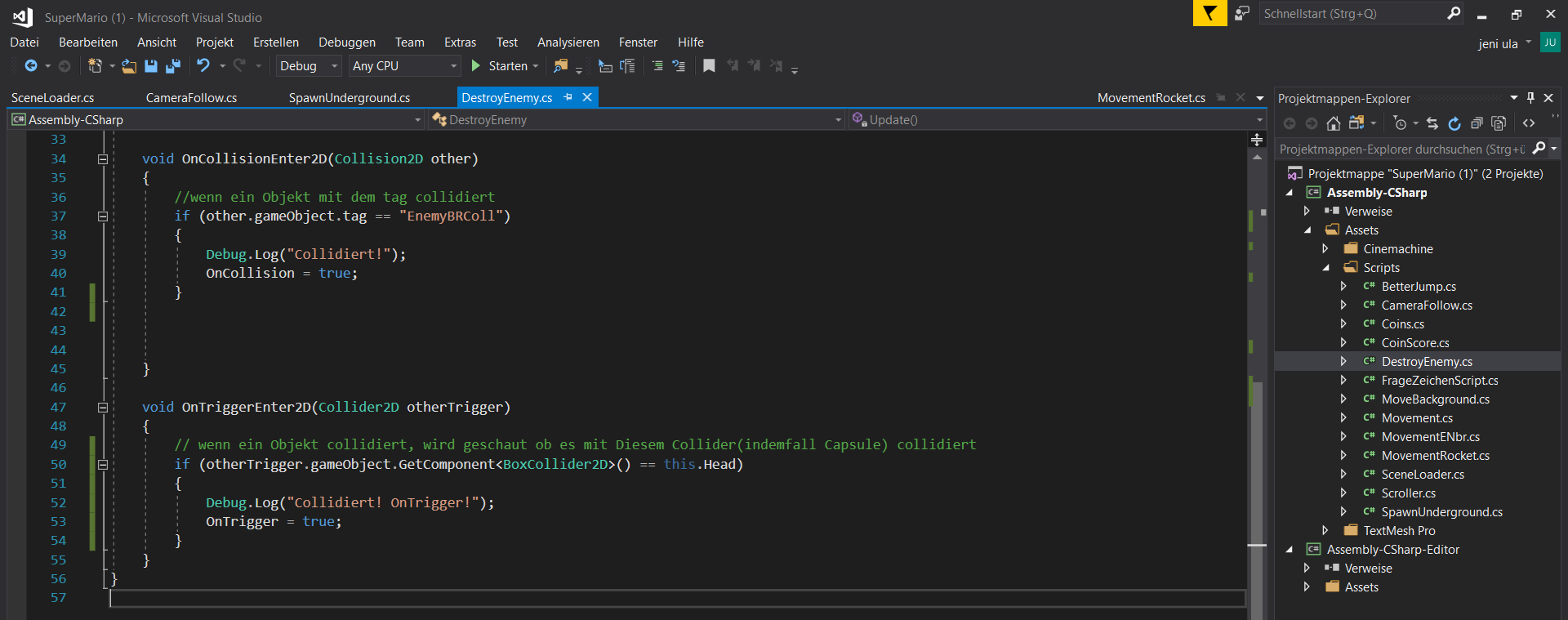


Abbildung 25

-Hier wird Das Bild von dem gegner (Pilz / EnemyBR) gewechselt, wenn er stirbt in das SterbeImage.

Die Klasse / komponente Sprite ist für das Image, das sich ändern soll und ist auf SerializedField, damit wir das Bild über die Oberfläche rein machen können.

-Ein Edge Collider ist ein Strich Collider, damit der Player auch den Pilz zerstören kann.

**OnCollisionEnter:** wird der Spieler sich mit dem gegner berührt. Ist der Fall wird OnCollision auf true gesetzt.

-In OnCollisionEnter wird dann der Player mit diesem "Strich-Collider" treffen. Ist der Fall wird OnTrigger auf true gesetzt.

**Update ():** hier wird jetzt geprüft, ob beide Collider treffen, wenn der Fall ist

dem Pilz das Bild übergeben, um zu verdeutlichen das dieser zerstört wurde.

-Ist das nicht der Fall und nur der normale Collider so bekommt man den Screen, dass man verloren hat. (wechselt zur Szene GameOver).

**FrageZeichenScript.cs:**

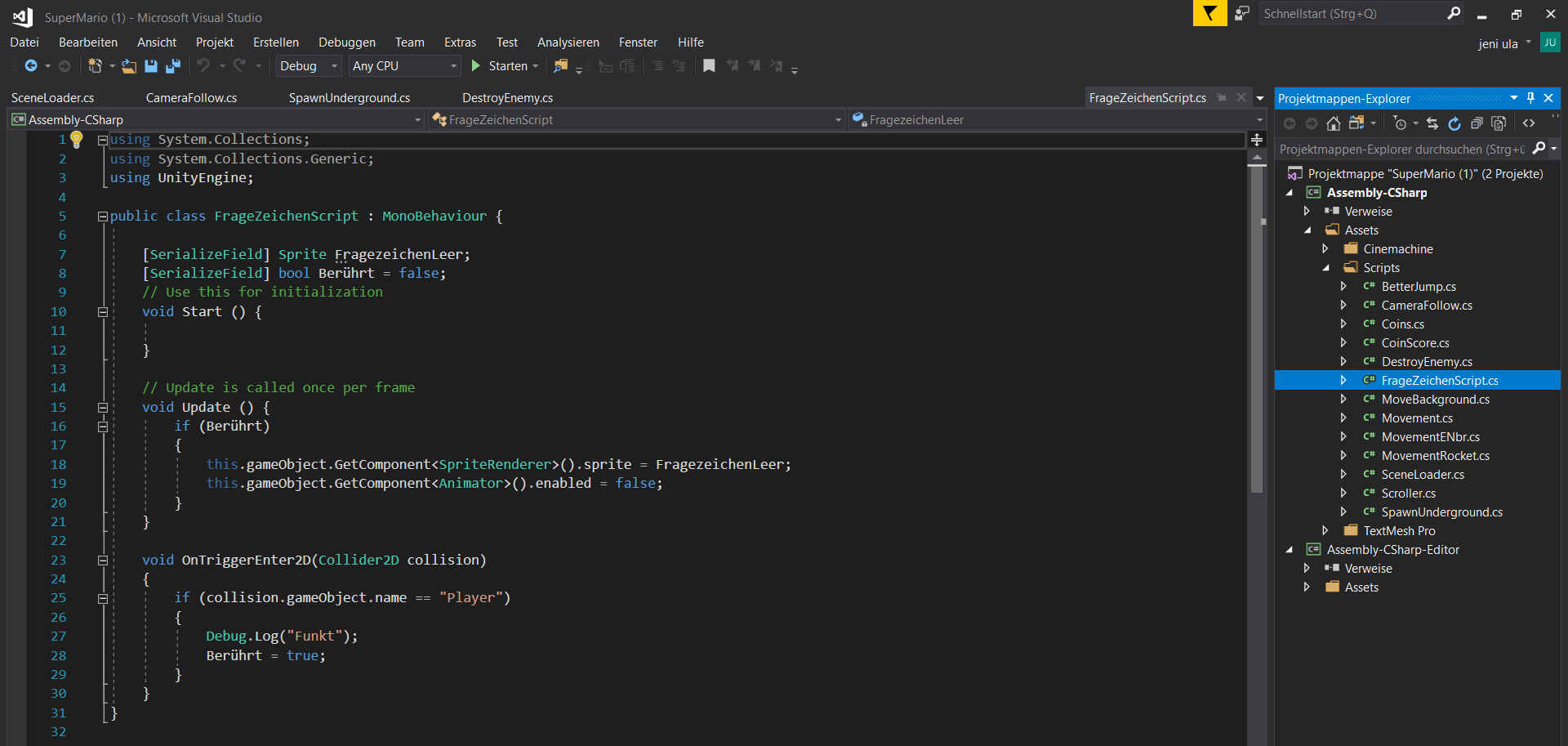


Abbildung 26

-Hier wird ebenfalls die Klasse / Komponente von Sprite verwendet um leere Fragezeichen zu übergeben.

**OnTriggerEnter:** Der Player wird mit dem Fragezeichen geändert.

**Update():** Wird abgefragt, ob es wahr oder falsch ist, ist es wahr, dann

wird das Fragezeichen geändert und die Animation wird gestoppt.

**CameraFollow.cs:**

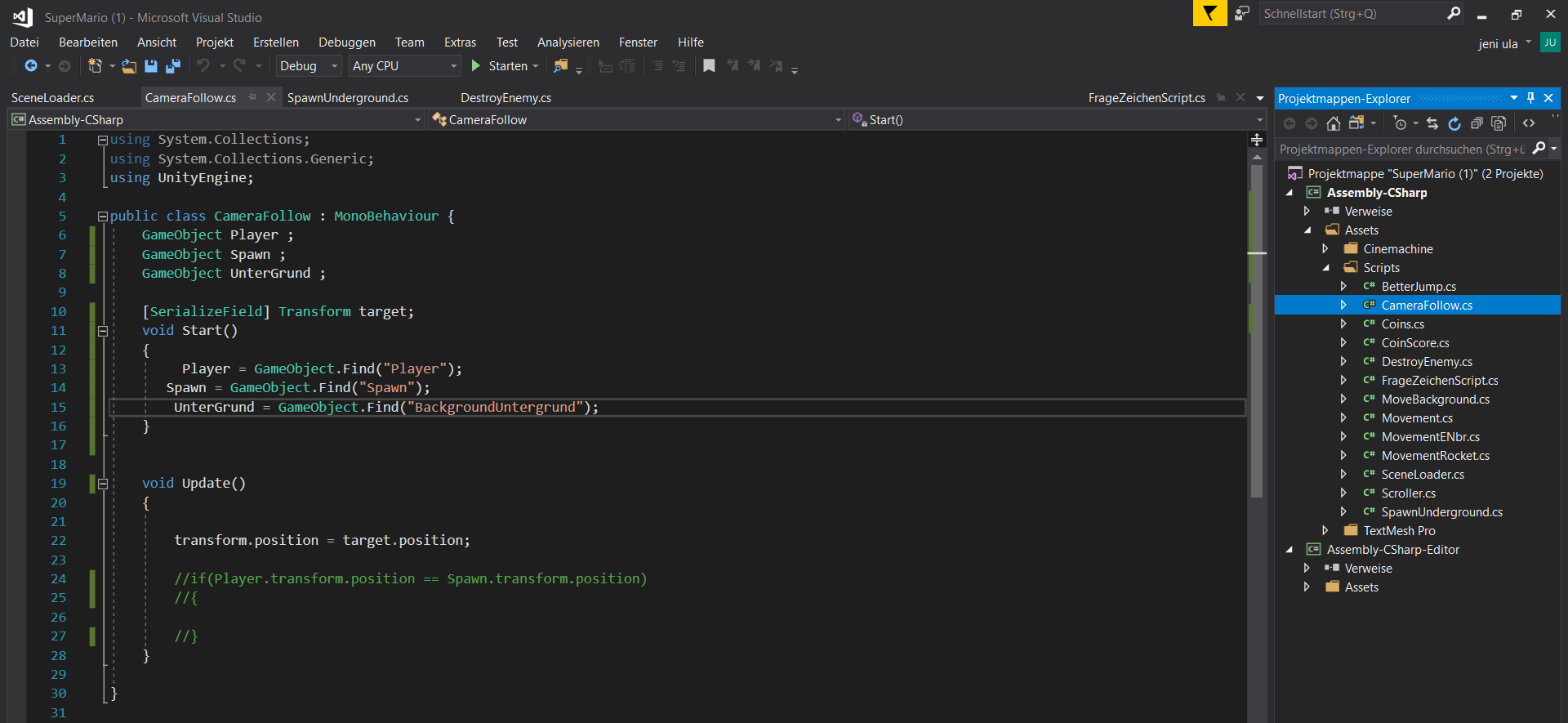


Abbildung 27

Transform soll dein GameObjekt über Unity übergeben werden, damit die Kamera auf den Hintergrund fokussiert.

**Update():** wird in der Kameraposition die Position des Background gespeichert.

**MoveBackground.cs:**

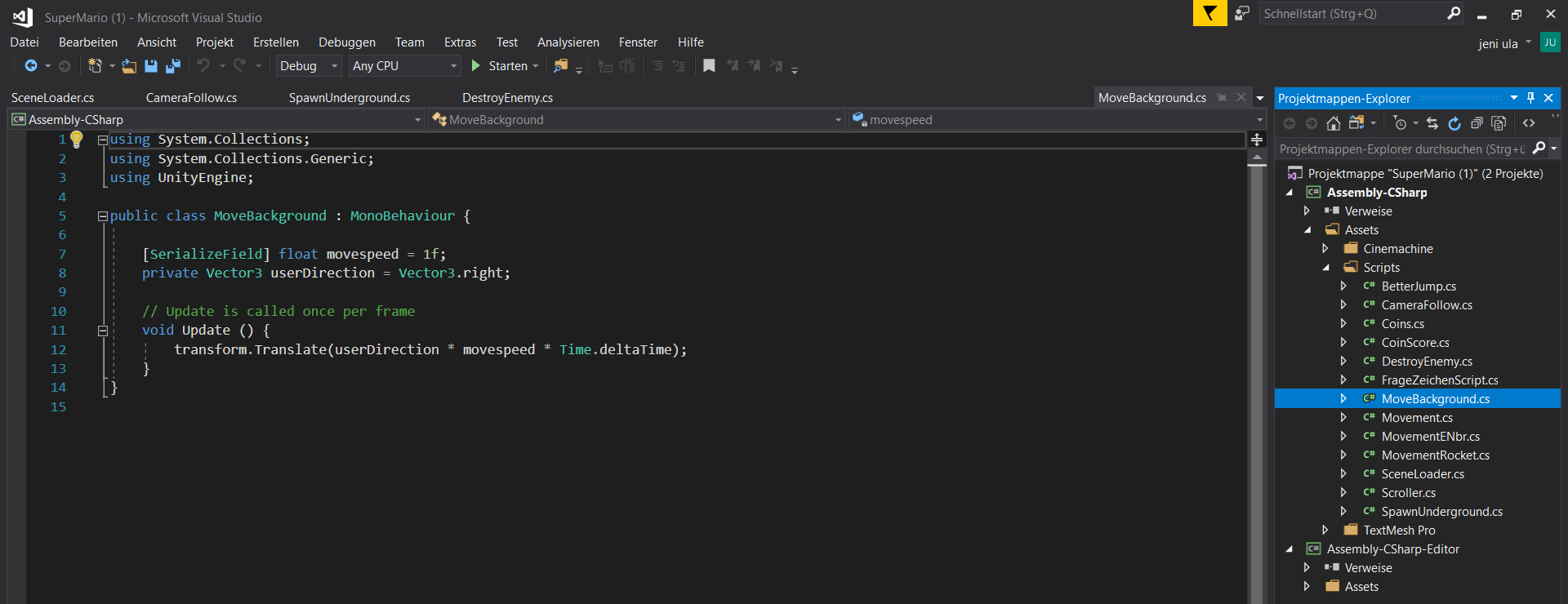


Abbildung 28

Hier wird der Hintergrund auf der X-Achse nach rechts verschoben. Selbe Funktion wie im Script MovementENbr, nur das der hintergrund statt links nach rechts geht.

**SceneLoader.cs:**

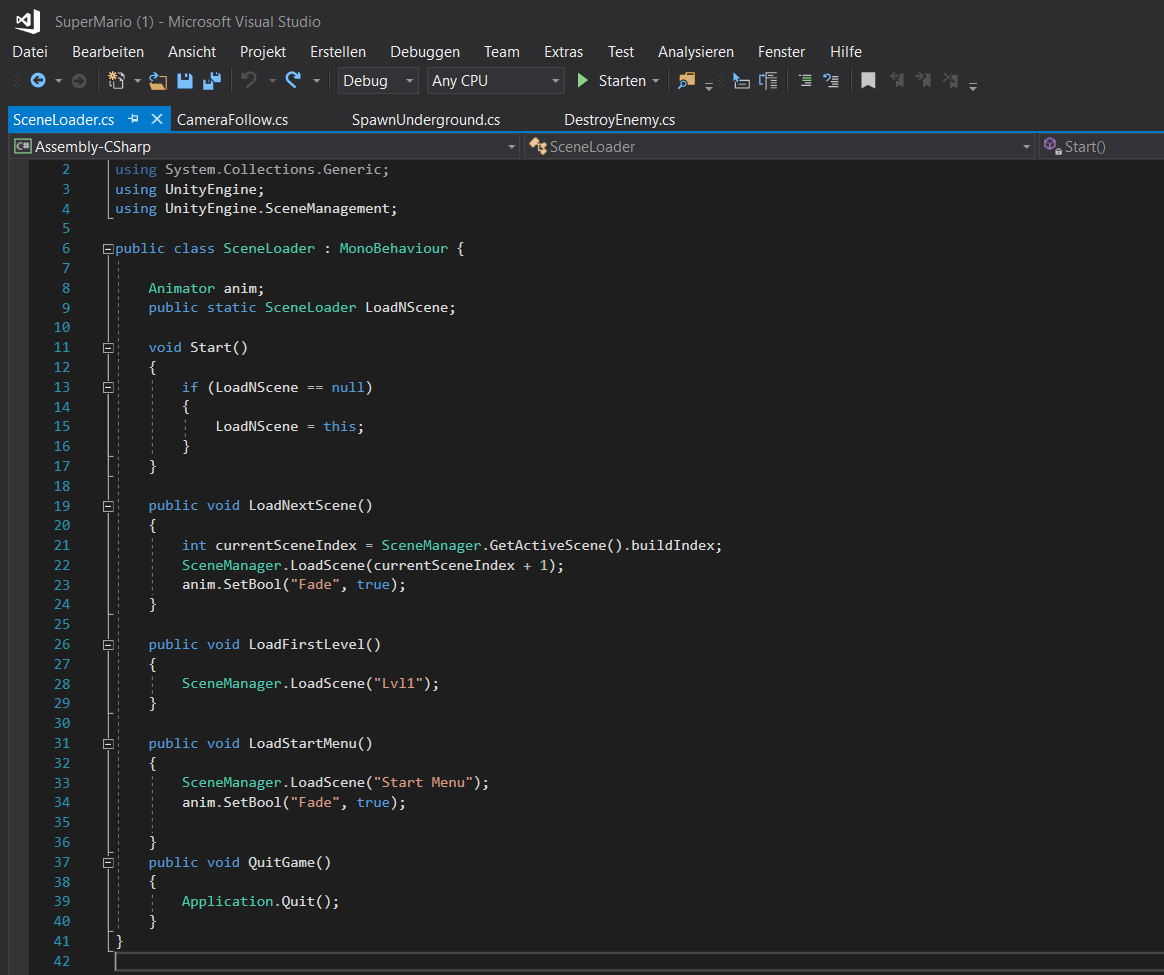


Abbildung 29

-Der Sceneloader wird für die Scenen in Unity verwendet, um diese zu wechseln (Bsp. Von Level 1 zu Level 2)

**LoadNextScene():**

-currentSceneIndex wird die aktuelle Szene übertragen mit buildIndex, welche über Unity erstmal erstellt werden müssen. Dort kann man dann auswählen welche Szene nummer 0, 1, 2 ... bekommen soll.

LoadScene: Soll die aktuele + 1 auch die nächste Szene öffnen.

**LoadFirstLevel():** Das erste Level wird geladen.

**LoadStartMenu():** StartMenü wird geladen.

**QuitGame():** Wenn nach Beendigung des Programms die Anwendung erstellt wird, setzt sie Szenen.

-Dieses Skript wird in den Buttons eingefügt. Später dann auch am besten, die nächste Szene wird geladen.

**Coin.cs:**

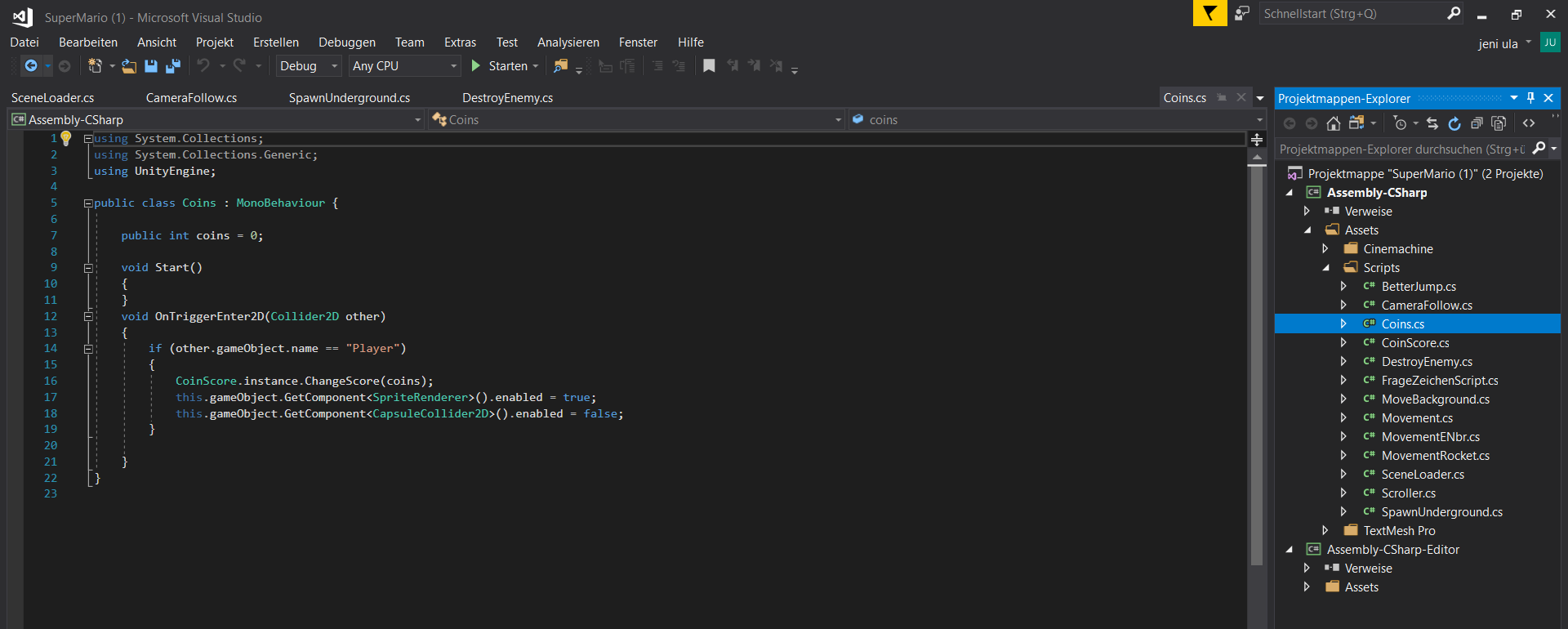


Abbildung 0

Sobald der Player mit dem Coin Collidiert wird, wird der Score hoch gezählt, danach der Spriterenderer (bekommt den Sprite für den Coin) Fragezeichen, da diese von anfang CapsuleCollider auf false damit wird der Score nicht mehr angezeigt.

**CoinScore.cs:**

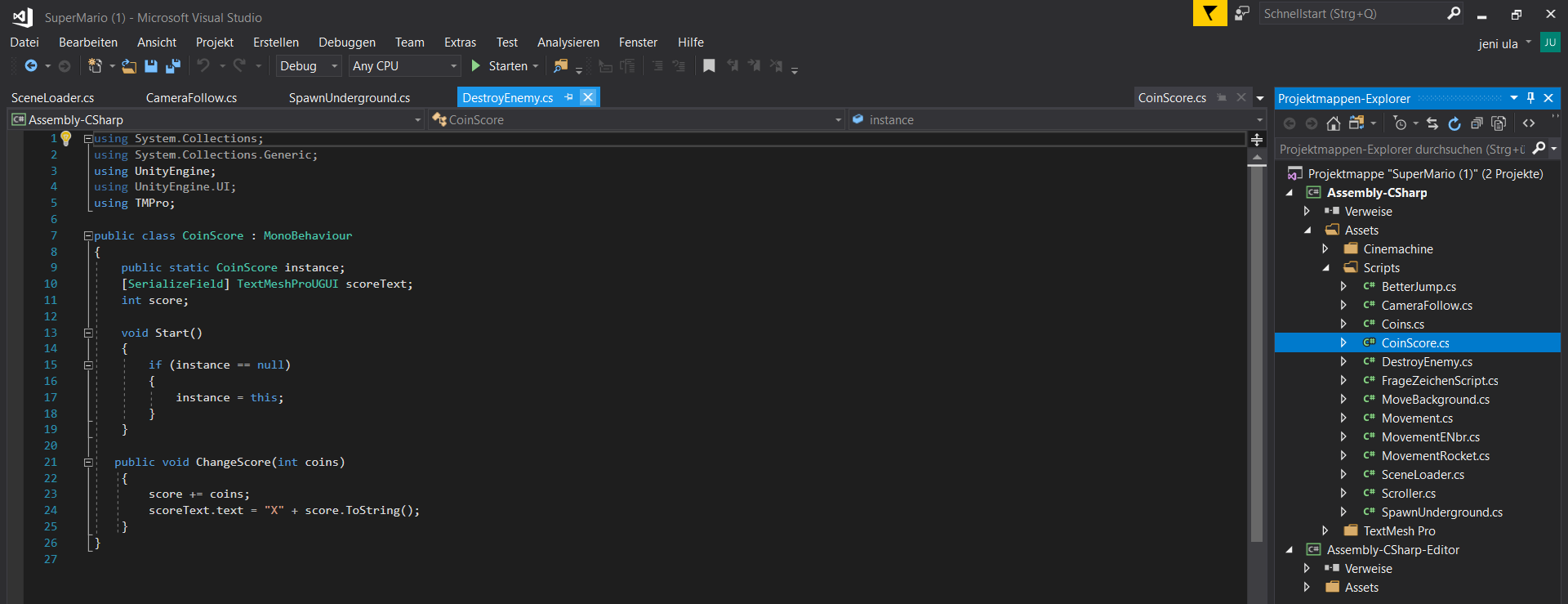


Abbildung 31

Zählt die Coins hoch und übergibt es dem TextMeshProGUI, welches wir über die Engine reingezogen habe.

TextMeshPro ist eine UI, die ständig sichtbar ist und / oder wie in diesem Fall.

## Aufgetretene Fehler

Hinweis: Fehler sind beim Testen aufgetreten! Ansonsten wurde bei jeder Änderung/Hinzufügung von Skripts, das Spiel getestet und lief fehlerfrei!

1. Background

Beim Testen unseres Spieles haben wir festgestellt, dass sich die Kamera mit dem Spieler sich bewegt, weil wir die Kamera auf unseren Spieler gesetzt haben, aber das Hintergrund stehen bleibt. Um dieses Problem zu beheben haben wir gemeinsam versucht die Kamera auf das Hintergrund zu fixieren. Da trat aber das Problem auf, dass sich die Kamera unproportional bewegte. Uswwwwwwwww

1. Score

Unser Score-Zähler zählt aus unergründlicher Weise falsch hoch. Wir haben alles Mögliche versucht, indem wir unseren Score-Text anders gestalten haben und oder das Skript geändert haben. Leider konnten wir das nicht fixieren.

1. Enemies zerstören

Da die Pilz-Enemies erst durch das draufspringen von Mario zerstört werden und die Raketen durch einen „Kopfnuss“ oder von drauflanden, hat dies große Probleme zubereitet. Zuerst codierten wir so, dass unser Spieler mit seine Enemies kollidiert, egal von welcher Richtung, und die sich dann auflösen. Dies lösten wir indem man mehrere, unterschiedliche Colliders hinzugefügt haben. Collider für den Kopfbereich von Mario und für den Fußbereich. Da wir ein Collider nur einmal im Skript verwenden dürfen hatten wir keine andere Wahl als unterschiedliche Colliders anzuwenden wie CapsuleCollider, EdgeCollider usw. Dies hatte aber auch den Vorteil das wir unsere Objekte auf Millimeter genau mit einem Collider verweisen könnten. Und konnten somit pro Collider eine Funktion erstellen und die Probleme lösen.

# Projektabschluss

## Projektübergabe

Das Projekt wird als ein funktionsfähiges Spiel weitergeben mit ein, zwei kleine Fehler. Die aber das Spielablauf nicht stark beeinflussen.

## Abnahmeprotokoll

Unser Score-Zähler funktioniert nicht einwandfrei, ist zwar mit eingebunden aber zählt nicht richtig hoch, wie wir es schon bei dem letzten Kapitel „Aufgetretene Fehler“ beschrieben haben.

## Projektbewertung

Soll-Ist-Vergleich:

|  |  |
| --- | --- |
| Soll: | Ist: |
| Startmenü | Ist vorhanden |
| GameOver | Ist vorhanden |
| Zwischenmenü | Nicht mit eingebunden |
| Avatar: Mario | Ist vorhanden |
| Avatar: Daisy | Nicht mit abgebunden, stattdessen ein „Ziel“ |
| Hindernisse bzw. Enemies | Sind vorhanden |
| Level 1 & Level 2 mit Untergrundlevel | Sind vorhanden |

**Abweichungen zur Planung:**

Plangemäß wurde alles eingehalten, auch die Arbeitsaufteilungen wurden eingehalten.

**Abweichungen vom ursprünglichen Ziel:**

Wie man es schon von Soll-Ist erkennen kann, haben wir unseren Projekt mit Zufriedenheit abgeschlossen, bis auf das Avatar „Daisy“, das durch das „Ziel“ ersetzt wurde, und der Score-Zähler nicht hochzählt, ist alles andere vorhanden, genau wie wir es geplant haben. Zeitmäßig haben wir alles eingehalten:

# Quellenverzeichnis

<https://www.spriters-resource.com/snes>

<https://docs.unity3d.com/Manual/index.html>

(Persönliche Erklärung)