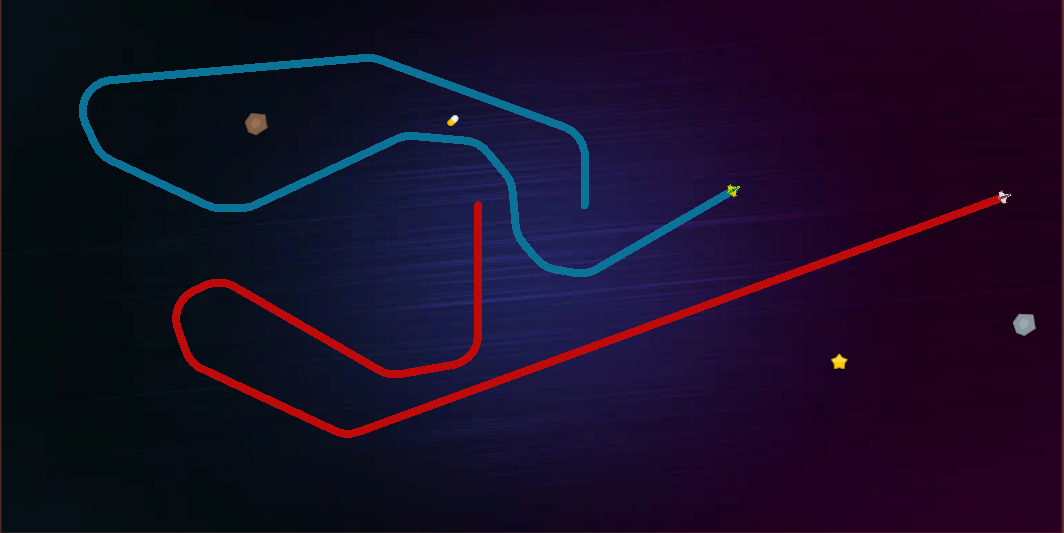
Dokumentation zur Projektarbeit

Unity Spiel: Insnakity



Von Benjamin Derksen

**Inhaltsverzeichnis**

Abbildungsverzeichnis 5

(Abkürzungsverzeichnis) 8

1 Einleitung 9

2 Planungsphase 9

2.1 Terminplanung 9

2.2 Sachmittelplanung 10

3 Durchführungsphase 11

3.1 Player Script 11

3.1.1 Move Funktionen 11

3.1.2 Neubeginn 12

3.1.3 Name und Richtung des Spiels 12

3.1.4 Aufbau der StartsSzene 12

3.1.5 Aufsetzen des ButtonManager 15

3.1.6 Die GameScene 16

3.1.7 Aufsetzen des Players 17

3.1.8 Aufsetzen der Line 17

3.1.9 Aufsetzen des Heads 25

3.1.10 FixedUpdate vs Update 35

3.1.11 Die Collisions für das Tag Tod 36

3.1.12 Der BonusSpawner 37

3.1.13 Die PowerUp Prefabs 40

41

3.1.14 Der Countdown 46

3.1.15 Der SceneSwitcher 49

3.1.16 Die PlayAgain Szene 51

3.1.17 Programmierung des gameDecider.cs 51

3.1.18 Hintergrundmusik für jede Szene 53

3.1.19 Nicht benutzt: PlayerSpawner.cs 54

3.1.20 Nicht benutzt: CameraShaker.cs 54

3.2 Aufgetretene Fehler 54

3.2.1 Fehler im Player Script 54

3.2.2 Fehler im Food Script 55

3.2.3 Fehler mit Collider 55

3.2.4 Fehler wenn ich Schriftart hinzufüge 55

3.2.5 Rendering von Texten 55

3.2.6 Schwerwiegender Fehler: Namensgebung ist wichtig! 57

3.2.7 Edge Collider zeichnet nicht richtig 57

3.2.8 Missing Reference Exception 58

3.3 Test 58

3.3.1 Testdurchführung 58

4 Projektabschluss 59

4.1 Projektübergabe 59

4.2 Abnahmeprotokoll 59

4.3 Projektbewertung 59

Quellenverzeichnis 60

Persönliche Erklärung 61

# Abbildungsverzeichnis

Abbildung 1: Projektplanung 9

Abbildung 2: Move Funktion 10

Abbildung 3: Schriftarten Generator 12

Abbildung 4: Anordnung StartSzene 13

Abbildung 5: Startszenen Überschrift 13

Abbildung 6: Fertige StartsSzene 14

Abbildung 7: Programm ButtonManager 14

Abbildung 8: Build Settings Unity 14

Abbildung 9: Buttonzuweisung Unity 15

Abbildung 10: GameSzene 15

Abbildung 11: Player 16

Abbildung 12: Line Einstellungen Unity 16

Abbildung 13: LineRenderer Einstellungen Unity 17

Abbildung 14: EdgeCollider2D Einstellungen Unity 18

Abbildung 15: Collider in Szene 18

Abbildung 16: Line.cs Variablen 19

Abbildung 17: Line.cs Start 20

Abbildung 18: Line.cs Update 21

Abbildung 19: Line.cs Ohne padding 21

Abbildung 20: Line.cs mit Padding 22

Abbildung 21: Line.cs SpawnPoint 23

Abbildung 22: Line.cs Ergebnis 23

Abbildung 23: Head Einstellungen in Unity 24

Abbildung 24: Head Sprite Renderer Einstellungen in Unity 24

Abbildung 25: Head Rigidbody2D Einstellungen in Unity 25

Abbildung 26: Head CircleCollider2D Einstellungen in Unity 25

Abbildung 27: Head CicleCollider2D in Szene 25

Abbildung 28: Head Audio Source Einstellungen in Unity 26

Abbildung 29: Head.cs Variablen 27

Abbildung 30: Head Einstellungen in Unity 27

Abbildung 31: Head.cs Start 28

Abbildung 32: Head.cs Update 28

Abbildung 33: Head.cs FixedUpdate 29

Abbildung 34: Head Z-Achsen Rotierung 29

Abbildung 35: Head Z-Achsen Beweis 29

Abbildung 36: Head.cs OnTriggerEnter2D 31

Abbildung 37: Tags in Unity 31

Abbildung 38: Player1 Tag 31

Abbildung 39: Head.cs Pill 32

Abbildung 40: Head.cs Shield 32

Abbildung 41: Head.cs Star 33

Abbildung 42: FixedUpdate vs Update, geschnitten aus https://unity3d.com/de/learn/tutorials/topics/scripting/update-and-fixedupdate 34

Abbildung 43: Tod Collisions in Szene 35

Abbildung 44: playerEnd\_Bottom in Unity 35

Abbildung 45: BonusSpawner Einstellungen in Unity 36

Abbildung 46: BonusSpawner.cs Variablen 37

Abbildung 47: BonusSpawner.cs Start 1 37

Abbildung 48: BonusSpawner.cs Start 2 38

Abbildung 49: Ergebnis in der Szene 39

Abbildung 50: Star Prefab Einstellungen in Unity 40

Abbildung 51: Star Sprite 41

Abbildung 52: StarScript.cs 41

Abbildung 53: Shield Prefab Einstellungen in Unity 42

Abbildung 54: Shield Sprite 42

Abbildung 55: ShieldScript.cs 42

Abbildung 56: Pill Einstellungen in Unity 43

Abbildung 57: Pill Sprite 43

Abbildung 58: PillScript.cs 43

Abbildung 59: Meteor Einstellungen in Unity 44

Abbildung 60: Meteor Sprite 45

Abbildung 61: Countdown in Szene 45

Abbildung 62: Countdown Einstellungen in Unity 46

Abbildung 63: Countdown im Animator 46

Abbildung 64: Countdown im AnimationController 47

Abbildung 65: delayedStart.cs Variablen 47

Abbildung 66: delayedStart.cs Start 47

Abbildung 67: delayedStart.cs StartDelay 48

Abbildung 68: SceneSwitcher Einstellungen in Unity 48

Abbildung 69: SceneSwitcher.cs Variablen 49

Abbildung 70: SceneSwitcher.cs Start 49

Abbildung 71: SceneSwitcher.cs Update 49

Abbildung 72: PlayAgain Szene 50

Abbildung 73: gameDecider.cs Variablen 50

Abbildung 74: gameDecider.cs Start 50

Abbildung 75: gameDecider.cs Update 51

Abbildung 76: Ergebnis gameDecider.cs 51

Abbildung 77: StartsSzene AudioSource Settings in Unity 52

Abbildung 78: Fehler mit Collider 54

Abbildung 79: Fehler Schriftart 54

Abbildung 80: Rendering von Text 55

Abbildung 81: SceneManager Namensgebungfehler 56

Abbildung 82: Edge Collider zeichnet falsch 56

Abbildung 83: Missing Reference Exception 57

# (Abkürzungsverzeichnis)

# Einleitung

Mein Projekt ist ein Multiplayer Spiel welches zwei Schlangen-artige vom Spieler gesteuerte Player gegeneinander kämpfen lässt. Im Spiel braucht man eine gute Reaktionszeit, um rechtzeitig auf die hinterlassenen Wände von den Schlangen zu reagieren. Das Spiel wird mit Hilfe der Game Engine Unity erstellt und auf GitHub gespeichert um zwischen Arbeitslaptop und Computer zu synchronisieren, außerdem bietet mir GitHub eine Sicherheit, dass das Projekt immer verfügbar ist. Das Spiel wird dann auf einen Webserver geladen betrieben von einem Raspberry Pi 2 und von da aus innerhalb des Browsers gespielt.

# Planungsphase

## Terminplanung

Das Projekt fand jeden Mittwoch von 12:30 Uhr bis 15:45 Uhr vom 12.12.2018 bis zum 12.04.2019 statt.

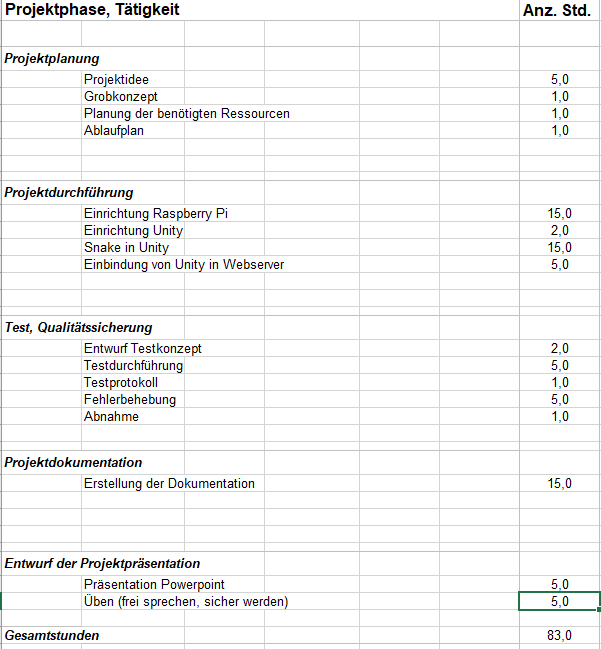


Abbildung 1: Projektplanung

## Sachmittelplanung

* Mein eigener Computer und Laptop
* Raspberry Pi 2 aus der Schule
* Unity Game Engine
* Visual Studio Community
* Github

# Durchführungsphase

## Player Script

Als erstes musste ich den Player erstellen der ist momentan nur ein Schwarzer Kreis. Der Player sollte der vom Spieler gesteuerten Maus folgen und selbst wenn die Maus stoppt sollte die Schlange sich weiter bewegen das der Spieler dazu gezwungen ist, die Richtung vor zu geben was das Ganze zu einem einzigartigen Movement macht.

### Move Funktionen

Im Folgenden zeige ich ein Ausschnitt meines Player Scriptes, diese Methode sorgt dafür das der Spieler sich mit der Maus bewegen kann:

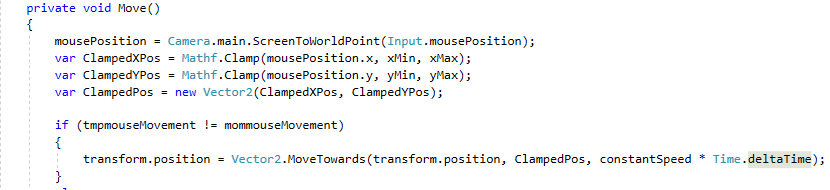


Abbildung 2: Move Funktion

Als erstes wird die mousePosition mit Hilfe einer Unity Funktion in 2D Vector umgewandelt, weil die Maus Position ja eigentlich nur Pixel sind.

Danach benutze ich die Clamp Funktion um den Spieler innerhalb seines Movements zu beschränken er soll sich ja innerhalb der Szene befinden und nicht rausfliegen.

Danach erstelle ich einen 2D Vektor aus der x und y Position, weil die Funktion MoveTowards nur einen 2D Vektor annimmt.

Wenn die Maus sich bewegt soll die neue Position mit Hilfe der Methode MoveTowards ermittelt werden.

### Neubeginn

Nachdem mir aufgefallen ist, dass ich nach mehreren Wochenenden das Problem mit der Maus ohne aufwendige Mathematische Funktionen, die ich nicht erklären kann nicht lösen konnte, z.b Lerp, musste eine Alternative her. Also bin ich zu dem Entschluss gekommen die Skripte alle so zu überarbeiten das sie mit einer Tastatur funktionieren anstatt mit einer Maus. Das bedeutet das ich ab dem Teil nochmal alle Skripte neu schreiben und erklären werde.

### Name und Richtung des Spiels

Zuerst muss man überlegen wie das Spiel heißen soll und wie es aussehen soll. Welche Gefühle sollen das Spiel übermitteln?

Diese Fragen, muss sich jeder Game Designer fragen bevor man überhaupt programmiert, sonst programmiert man tausend Sachen die man am Ende gar nicht braucht wie bei mir.

Damit mir das nicht nochmal passiert habe ich mich für ein Retro Game Design entschieden, welches Nostalgie wecken soll aber durch neue Spiel-Mechaniken trotzdem aktuell ist.

Das Spiel heißt Insnakity: Ein Namenspiel aus Snake und Insane.

### Aufbau der StartsSzene

#### Erstellen eines Fonts unter Unity

Als allererstes will ich meine Startsszene aufbauen, das mache ich mit der Hilfe von TextMeshPro. Dieses Addon erlaubt mir Text viel schöner auf zu bereiten. Danach habe ich mir eine Schriftart ausgesucht.

|  |  |
| --- | --- |
| Abbildung 3: Schriftarten Generator | Hier generiere ich aus dem Font ein Unity brauchbare Schriftart.  Die Einstellung habe ich nicht verändert. |

#### Canvas-Erstellung unter Unity

Um in Unity Text und Buttons dar zu stellen braucht man ein Canvas. Unter diesem Canvas werden alle Buttons und Texte zu Kindern.

|  |  |
| --- | --- |
| Abbildung 4: Anordnung StartSzene | Das ist die Anordnung der GameStart Szene |

|  |  |
| --- | --- |
| Abbildung 5: Startszenen Überschrift | Zu beachten sind die Rect Transforms, das sind die Pfeile an den Ecken, um jedes Bild ob Handy, Pc oder Notebook dar zu stellen muss Unity wissen wo die Ecken des Textes sind um die richtige Position zu kalkulieren. |

#### Ergebnis StartSzene

|  |  |
| --- | --- |
| Abbildung 6: Fertige StartsSzene | Das ist mein fertiger Start Bildschirm. Der Retro Aspekt kommt sofort zum Vorschein. |

### Aufsetzen des ButtonManager

#### Programmierung des ButtonManager.cs

|  |  |
| --- | --- |
| Abbildung 7: Programm ButtonManager | Die Programmierung der Buttons ist ganz einfach:  Zuerst darf man nicht vergessen UnityEngine.SceneManagement hinzuzufügen. Nachdem man das getan hat kann man den Scene Manager nach den Szenen fragen, wenn man sie denn in der Build Liste hinzugefügt hatte. |

#### ButtonManager in Unity



Abbildung 8: Build Settings Unity

|  |  |
| --- | --- |
| Abbildung 9: Buttonzuweisung Unity | Damit Unity weiß bei welchem Button, welche Funktion ausgeführt werden soll, muss man das im Inspector angeben. |

### Die GameScene

|  |  |
| --- | --- |
| Abbildung 10: GameSzene | Die GameScene sollte relativ schlicht gehalten werden, weil zu viel Ablenkung im Hintergrund beim Spielen eher als störend empfunden wird. Zu beachten sind die Raumschiffe welche Linien hinter sich ziehen und der Countdown. Beides werde ich im Folgenden erklären. |

### Aufsetzen des Players

|  |  |
| --- | --- |
| Abbildung 11: Player | Der Player besteht aus zwei Skripte, der steuernde Kopf und die hinter sich gezogene Linie aus EdgeCollider. Hier wird es etwas kompliziert. |

### Aufsetzen der Line

|  |  |
| --- | --- |
| Abbildung 12: Line Einstellungen Unity | Als erstes erkläre ich die Line, die braucht ein Transform damit Unity weiß wo sich das Object befindet. Ein Line Renderer zeichnet die Linie in die Szene ausgehend von der Spieler Position. Der Edge Collider 2D, welcher bestimmt wann etwas mit etwas kollidieren, so bestimmt man wann ein Spieler zum Beispiel stirbt. Das BlueMat zeichnet die Linie in Blau, aber wichtig ist das der Shader auf Sprites/Default steht sonst kommt 3D Rendering mit Lichteffekten ins Spiel. |

#### Der Line Renderer in Unity

|  |  |
| --- | --- |
| Abbildung 13: LineRenderer Einstellungen Unity | Der Line Renderer ist eigentlich das wichtigste in meinem Spiel, ohne den könnte ich keine Linien hinter mir ziehen, ohne es wirklich aufwendig zu programmieren.  Die Einstellungen die ich verändert habe, sind einmal das BlueMat, das lässt mich die Linie in Blau zeichnen.  Dann die Size um die Linie in die Szene zu zeichnen, um zu erkennen ob alles funktioniert und die Collider nicht ineinander Kollidieren,aber dazu später mehr.  Die Color habe ich angepasst und die Vertices, die besagen wie weich die Kanten sind, desto höher ich das setze desto unperformanter wird das Spiel.  Dann die Width die ausmacht wie dick die gezeichnete Linie sein soll und die Order in Layer, die die Linie überhalb des Hintergrundes zeichnet sonst verschwindet sie dahinter. |

#### Der EdgeCollider2D in Unity

|  |  |
| --- | --- |
| Abbildung 14: EdgeCollider2D Einstellungen Unity    Abbildung 15: Collider in Szene | Hier zu sehen, die Einstellungen des EdgeCollider2D, dort ist eigentlich nur eins wichtig und zwar den Collider als Trigger einzustellen sonst passiert nichts, wenn der Player die Linie überschreitet.  Nebenan sieht man die grüne Linie das ist der Collider. |

#### Programmierung des Skriptes: Line.cs

|  |  |
| --- | --- |
| Abbildung 16: Line.cs Variablen | Als erstes musste ich System.Linq hin zu fügen damit kann ich Listen Funktionen benutzen die einem das Leben später einfacher machen.  Das [SerializeField] bedeutet das ich diese Variable im Unity Inspector sehen kann, das zeige ich später genau.  Um die Linie zu zeichnen brauch ich die Position des Heads, deswegen Transform.  Das padding wird unten nochmal genau erklärt, vereinfach ist es der Abstand zwischen zwei Punkten.  Dann habe ich ein Objekt vom Linerenderer und dem Edge Collider erstellt, das new braucht man um nach dem Spiel wieder mit neuen Linien anfangen zu können.  Die pointsList brauche ich um den zu steuernden Kopf zu folgen. |

|  |  |
| --- | --- |
| Abbildung 17: Line.cs Start | Die Start Methode wird in Unity hervorgerufen, wenn das Spiel startet.  Hier hole ich mir nochmal die Komponenten des GameObjectes, weil die Information nicht im Object steht, sondern in den einzelnen Komponenten, wenn man das nicht tut, kriegt man eine ObjectReferenceException.  Dann spawne ich den ersten Punkt. |

|  |  |
| --- | --- |
| Abbildung 18: Line.cs Update    Abbildung 19: Line.cs Ohne padding    Abbildung 20: Line.cs mit Padding | Die Update Methode wird nach jedem Frame (Bildsequenz) hervorgerufen.  Als erstes Checke ich ob die Liste aus headList überhaupt gefüllt ist, wenn ich das nicht tue bombardiert mich Unity mit MissingReferenceExceptions. Wie ich in den aufgetretenen Fehlern bei 3.2.8 erklärt habe.  Die headList ist eine Liste aus Koordinaten, des zu steuernden Kopfes, so weiß die Linie wohin sie zeichnen soll.  Jetzt kommt das nächste if Statement, hier checke ich ob die Distanz zwischen dem zuletzt hinterlegten Punkt und dem Spieler größer als der Abstand ist, den man verändern kann. Ein bisschen kompliziert, aber ich veranschauliche mal was ich damit meine:  Links sieht man was passiert, wenn ich das nicht tue, nach kürzester Zeit sammeln sich viele Punkte im Array, weil ich nach jedem Frame einen Punkt zeichne, wenn man das 2-mal macht, weil ich ja 2 Player habe, merkt man das es sehr stark ruckelt.  Im Vergleich zu rechts sind sehr viel weniger Punkte, weil ich jedes Mal nach dem Abstand frage ob ich einen neuen Punkt zeichnen darf. Das Resultat ist, dass das Spiel auch nach langen Sessions flüssig läuft. |

|  |  |
| --- | --- |
| Abbildung 21: Line.cs SpawnPoint    Abbildung 22: Line.cs Ergebnis | Nun zur Funktion die, die eigentliche Arbeit erledigt.  Die Liste die ich gesammelt habe, die praktisch die Fußstapfen des Spielers sind lade ich nun in den EdgeCollider2D rein, jetzt folgt der EdgeCollider2D unserem Player, die tödliche Linie wird hinter dem Player gezogen.  Dann füge ich der Liste die Position des Spielers hinzu.  Jetzt füge ich dem LineRenderer dieselbe Anzahl an Punkten in der Liste zu, wie der Spieler sich bewegt hat.  Danach kann ich die eigentliche Position zeichnen an der letzten Stelle an dem der Spieler war und wo der Spieler jetzt ist.  Das Ergebnis ist links zu sehen. |

### Aufsetzen des Heads

#### Head in Unity

|  |  |
| --- | --- |
| Abbildung 23: Head Einstellungen in Unity | Das ist die Übersicht des Heads. Der Head ist der Player, der die Linie steuert.  Der Sprite Renderer zeichnet den Sprite (mein Raumschiff) auf die Szene, das Skript erkläre ich wieder unten.  Das Rigidbody2D ist die Physik die ich brauch um zu kollidieren, und der eigentliche CircleCollider2D der wieder bestimmt wann man kollidiert.  Und die Audio Source für einen Sound den man wiedergibt, wenn etwas Bestimmtes passiert. |

#### Die SpriteRenderer Einstellungen des Heads

|  |  |
| --- | --- |
| Abbildung 24: Head Sprite Renderer Einstellungen in Unity | Hier habe ich nur zwei Sachen verändert, einmal das Sprite, welches mein Raumschiff beinhaltet und die Order in Layer damit der Player über der Linie gezeichnet wird. |

#### Die Rigidbody2D Einstellungen des Heads

|  |  |
| --- | --- |
| Abbildung 25: Head Rigidbody2D Einstellungen in Unity | Hier ist es wichtig das man Kinematic einstellt sonst hat man eine Physik und die Linie fliegt herum. |

#### Die CircleCollider2D Einstellungen des Heads

|  |  |
| --- | --- |
| Abbildung 26: Head CircleCollider2D Einstellungen in Unity    Abbildung 27: Head CicleCollider2D in Szene | Hier ist es wieder wichtig den Trigger einzustellen, außerdem soll der Radius des Colliders die Sprite umranden. |

#### Die Audio Source Einstellungen des Heads

|  |  |
| --- | --- |
| Abbildung 28: Head Audio Source Einstellungen in Unity | Hier muss nur der Clip der abgespielt werden soll, reingezogen werden.  Dann habe ich das Play on Awake ausgemacht, weil der Sound nicht direkt abgespielt werden soll, sondern nur wenn was passiert.  Das Loop habe ich angeklickt falls der Sound zuende geht soll er weiter abgespielt werden. |

#### Programmierung des Skriptes: Head.cs

|  |  |
| --- | --- |
| Abbildung 29: Head.cs Variablen    Abbildung 30: Head Einstellungen in Unity | Weiter geht’s mit dem Head Skript, hier wird der Spieler gesteuert.  Als erstes sieht man das ich [Header()] benutze, das mache ich damit der Unity Inspector aufgeräumter ist, wie man unten sehen kann. Mit [SerializeField] zeige ich diese Variablen im Inspector. So kann ich Variablen ändern ohne in das Programm zu müssen.  Nun zu den zwei Variablen speed und rotationSpeed, wie der Name schon sagt ist es die Geschwindigkeit mit welcher, der Player sich bewegt. Die Power-Up Variablen sind alle für die Zeit der Variablen da (in Sekunden).  Die inputAxis ist ein Preset von Unity damit man nicht jede Taste programmieren muss, dazu später mehr.  Danach hole ich mir den spriteRenderer um eine Farbe bei Aufnahme eines PowerUp zu ändern.  Und die AudioSource um einen bestimmten Sound abzuspielen.  Ich habe [HideInInspector] benutzt, weil ich diese Variable nicht im Inspektor angezeigt haben will, aber public gemacht werden muss, damit ich später von außen zu greifen kann.  isDead ist eine Variable die auf true gesetzt wird wenn der Spieler stirbt. invulnerable wird auf true gesetzt sobald der Spieler unverwundbar ist. |

|  |  |
| --- | --- |
| Abbildung 31: Head.cs Start | Die Start Methode holt sich wie vorher die benötigten Komponenten um diese später zu verändern. |

|  |  |
| --- | --- |
| Abbildung 32: Head.cs Update | Die Update Methode aktualisiert jeden Frame den Input aus der Tastatur. |

|  |  |
| --- | --- |
| Abbildung 33: Head.cs FixedUpdate    Abbildung 34: Head Z-Achsen Rotierung    Abbildung 35: Head Z-Achsen Beweis | Hier sieht man das ich FixedUpdate benutze anstatt Update. Erklärung dazu kommt noch.  Als erstes benutze ich die Funktion Translate, diese Funktion bewegt mein Raumschiff in eine Richtung mit Hilfe von Vector2.up, welche das Raumschiff nach oben bewegt. DeltaTime ist wichtig, weil die Geschwindigkeit ansonsten ungleichmäßig schnell auf verschiedenen Computern ist wegen der unterschiedlichen Framerate. Space.Self erlaubt uns den Spieler um sich selbst zu drehen.  Als nächstes will ich den Spieler auch nach links oder rechts bewegen. Hier übergebe ich Vector3.forward welche die Z Achse bewegt, das bedeutet das man rotieren kann, bewiesen mit den beiden Bildern zu der linken. Das dann mal minus horizontal um den Spieler mit der Tastatur zu bewegen, minus weil die Tastatur sonst invertiert wird. Horizontal gibt uns ein Wert zwischen -1 und 1 deswegen kann man das rein multiplizieren. Dann mal die rotationsGeschwindigkeit die man im Inspector verändern kann und wieder DeltaTime.  . |

|  |  |
| --- | --- |
| Abbildung 36: Head.cs OnTriggerEnter2D    Abbildung 37: Tags in Unity    Abbildung 38: Player1 Tag | Nun weiter mit den Kollisionen, Unity gibt mir auch hier wieder eine Funktion die OnTriggerEnter2D heißt mit der ich Kollisionen mit Hilfe von Tags identifiziere.  Der erste Tag heißt Tod, ziemlich selbsterklärend, wenn der Player hiermit kollidiert soll er sterben und aus der Szene verschwinden, welches ich mit Destroy(gameObject) und isDead = true erreiche.  Dann folgen Power-Ups, welche alle unterschiedliche Eigenschaften besitzen, die folgen noch.  Abschließend habe ich noch Ignoreable dieses Tag ist dafür da das man unsterblich werden kann aber trotzdem nicht aus der Szene fliegen darf, deswegen auch Tod und Ignoreable. |

|  |  |
| --- | --- |
| Abbildung 39: Head.cs Pill | Das erste Power-Up mit dem Namen Pill verlangsamt den Spieler. Das habe ich erreicht indem ich den Speed verringere, dann als Indikator für den Spieler, die Farbe verändere und jetzt solange warte, wie die Variable die ich vorgegeben habe. Natürlich muss ich danach alles wieder zurück einstellen sonst bleibt man ja für immer verlangsamt. |

|  |  |
| --- | --- |
| Abbildung 40: Head.cs Shield | Das zweite Power Up: Shield macht den Spieler für eine kurze Zeit unsterblich.  Hierfür habe ich ein boolean gesetzt mit invulnerable. Das gameObject kann erst zerstört werden, wenn invulnerable == false ist, kann man oben nochmal sehen.  Auch hier wieder eine andere Farbe, warten und die Einstellungen zurückstellen. |

|  |  |
| --- | --- |
| Abbildung 41: Head.cs Star | Am Ende des Skriptes angelangt wartet der Star, von Mario inspiriert, spielt er denselben Sound ab wie bei Mario, macht einen unsterblich und verschnellert ein. Sehr gut um Spieler die Wand vor die Nase zuhalten um zu gewinnen. |

#### Die inputAxis

|  |  |
| --- | --- |
|  | Ich habe vorhin die inputAxis erwähnt mit dessen Hilfe ich nicht jede einzelne Taste programmieren muss.  Nebenan zeige ich wie sie aufgebaut wurde und das ich noch eine brauche um den zweiten Player steuern zu können.  Hier wird einmal a und d und left und right als Positiv und Negativ eingegeben, schon erledigt Unity den Rest für mich.  Das Spiel würde sogar mit einem JoyStick funktionieren. |

### FixedUpdate vs Update

|  |  |
| --- | --- |
| Abbildung 42: FixedUpdate vs Update, geschnitten aus https://unity3d.com/de/learn/tutorials/topics/scripting/update-and-fixedupdate | FixedUpdate ist sehr kompliziert aber eigentlich bedeutet es nur das Unity gleichmäßig Updatet.  Wenn ich das Ganze in Update schiebe werden die Raumschiffe extrem schnell. Ich habe mir gemerkt das alles was bewegt wird, also mit Physik zutun hat in FixedUpdate soll, um einen gleichmäßigen Ablauf zu erlangen.  Links sieht man wann genau Update und wann FixedUpdate gecallt wird. |

### Die Collisions für das Tag Tod

|  |  |
| --- | --- |
| Abbildung 43: Tod Collisions in Szene    Abbildung 44: playerEnd\_Bottom in Unity | Da ich vorhin den Tod genannt habe, veranschauliche ich das Ganze nochmal.  Wenn ich invulnerable (unverwundbar) bin darf ich nicht aus den Szenen fliegen, deswegen brauchte ich hier ein Tag. Die grünen Linien sind der Collider = Tod.  Auch hier wieder Tag setzen und is Trigger anticken. |

### Der BonusSpawner

#### Der Bonusspawner in Unity

|  |  |
| --- | --- |
| Abbildung 45: BonusSpawner Einstellungen in Unity | Das Verhalten der Power Ups habe ich ja schon programmiert, aber irgendwie müssen sie ja auch auf der Szene landen, und ich wollte das sie immer woanders spawnen um immer ein anderes Spiel zu generieren, was den Spaß erhöht.  Nebenan sieht man die einzelnen Prefabs die ich reingezogen habe.  Ein Prefab ist eine Ansammlung von Komponenten, welche man immer wieder benutzen möchte. Geht man jetzt davon aus das ich ein zweites Level mache kann ich die Prefabs einfach reinziehen und muss sie nicht neu gestalten. |

#### Programmieren des BonusSpawner.cs

|  |  |
| --- | --- |
| Abbildung 46: BonusSpawner.cs Variablen | Als allererstes braucht der BonusSpawner den Abstand zur Wand also die Variable padding.  Dann muss ich eine neue Liste initialisieren und im Unity Inspector anzeigen damit man die Prefabs reinziehen kann.  Die Variablen xMin, xMax, yMin, yMax sind die Grenzen der Kamera, dazu komme ich gleich, der Index ist selbsterklärend. |

|  |  |
| --- | --- |
| Abbildung 47: BonusSpawner.cs Start 1 | In der Start Methode initialisiere ich die Camera und dann hole ich mir für jede Variable das Ende und addiere oder subtrahiere den Abstand. Sonst kann es sein das ein Objekt zur Hälfte außen spawnt.  Die übergebenen Einsen und Nullen sind die Ecken. 1,1 ist zum Beispiel oben rechts.  Das Skript bewirkt das ich die Prefabs nur innerhalb der Kamera spawne, was ja das einzige ist was man sehen kann. |

|  |  |
| --- | --- |
| Abbildung 48: BonusSpawner.cs Start 2 | Hier gehe ich die Liste durch und für jedes Prefab in der Liste errechne ich mit Hilfe der Random Range eine neue Position innerhalb der x und y-Achse, die 0 am Ende stellt die z-Achse dar. Die muss mit angegeben werden, weil das ja ein Vector3 ist.  Danach Instantiate ich, dass bedeutet das ich das Prefab in die Szene lade.  Instantiate braucht folgende Parameter:  Das Prefab was geladen werden soll: bonusList[Index]  Zweitens die transform.position, also die Position des Prefabs in der Szene: randomPosition  Und die Rotation des Objektes, welche mit sehr komplizierten Quaternions angegeben wird.  Dann inkrementiere ich den Index um zum nächsten Prefab zu kommen. |

#### BonusSpawner: Das Ergebnis in der Szene

|  |  |
| --- | --- |
| Abbildung 49: Ergebnis in der Szene | Das Ergebnis:  Alles spawnt zufällig bis auf die Player. |

### Die PowerUp Prefabs

#### Das Star Prefab

|  |  |
| --- | --- |
| Abbildung 50: Star Prefab Einstellungen in Unity    Abbildung 51: Star Sprite    Abbildung 52: StarScript.cs | Damit der BonusSpawner ganz genau weiß was er spawnen soll, muss man die Prefabs erstellen, dies ist auch schnell erledigt.  Das ist das Prefab für den Stern, hier muss das Sprite wieder in den Sprite Renderer gezogen werden. Außerdem wie immer nicht vergessen isTrigger anzuklicken. Zusätzlich muss der Stern bei Kollision aus der Szene verschwinden das ist in dem Mini Skript unten schnell erledigt, ab hier ist es nur noch Replikation deswegen werde ich die anderen Prefabs zwar zeigen, aber nicht erklären, weil es immer das gleiche ist.  Nicht zu vergessen sind die Tags, sonst kann ich kein Verhalten programmieren, man sollte wissen, mit was man genau kollidiert. |

#### Der Shield Prefab

|  |  |
| --- | --- |
| Abbildung 53: Shield Prefab Einstellungen in Unity | Abbildung 54: Shield Sprite    Abbildung 55: ShieldScript.cs |

#### Das Pill Prefab

|  |  |
| --- | --- |
| Abbildung 56: Pill Einstellungen in Unity | Abbildung 57: Pill Sprite    Abbildung 58: PillScript.cs |

#### Die beiden Meteoren

|  |  |
| --- | --- |
| Abbildung 59: Meteor Einstellungen in Unity    Abbildung 60: Meteor Sprite | Die beiden Meteoren brauchen kein Skript wenn man sie anstößt ist das Spiel vorbei, das bringt ein bisschen Zufälligkeit ins Spiel. |

### Der Countdown

|  |  |
| --- | --- |
| Abbildung 61: Countdown in Szene | Das Spiel startet nun, die Spieler bewegen sich, aber wenn man jetzt in die Szene reinlädt, fehlt irgendetwas. Man wird direkt in die Szene geworfen. Es fehlte ein Countdown. |

#### Der Countdown in Unity

|  |  |
| --- | --- |
| Abbildung 62: Countdown Einstellungen in Unity | Ein Countdown der von 3 bis 1 zählt ist nichts anderes als eine Animation, also brauche ich hier neben den Sprite Renderer für die Zahl auch ein Animator.  Wichtig ist hier, den Controller rein zu ziehen und den Update Mode auf unscaled Time zu stellen. |

#### Der Countdown im Animator

|  |  |
| --- | --- |
| Abbildung 63: Countdown im Animator | Die Animation gespeichert, nun kann ich die einzelnen Sprites in die Zeiten ziehen und abspielen. |

#### Der Countdown im Animationscontroller

|  |  |
| --- | --- |
| Abbildung 64: Countdown im AnimationController | Damit Unity weiß wann eine Animation abgespielt werden soll muss man dies dem Controller mitteilen. Bei mir ist das ganz simpel ich will die Animation direkt laden also im Entry. |

#### Programmieren des Countdowns: delayedStart.cs

|  |  |
| --- | --- |
| Abbildung 65: delayedStart.cs Variablen | Als erstes holt man die Animation die abgespielt werden soll vom Inspector. |

|  |  |
| --- | --- |
| Abbildung 66: delayedStart.cs Start | Dann starte ich die Coroutine, eine Coroutine lässt uns ein Timer erstellen. |

|  |  |
| --- | --- |
| Abbildung 67: delayedStart.cs StartDelay | Erstmal stoppt man die Zeit mit Time.timescale = 0, dann kalkuliert man die pauseTime mit der Zeit die vergangen ist, also 0 mit 3f also 3 Sekunden.  Dann wartet man solange bis die Zeit verstrichen ist und gibt dem Parameter null zurück.  Jetzt muss man noch die Animation aus der Szene bekommen mit SetActive, und die Zeit wieder einstellen damit das Spiel losgehen kann. |

### Der SceneSwitcher

#### Der SceneSwitcher in Unity

|  |  |
| --- | --- |
| Abbildung 68: SceneSwitcher Einstellungen in Unity | Jetzt hat man also die Spieler, den Countdown und die Power-Ups. Aber wer bestimmt wann das Spiel vorbei ist beziehungsweise wer gewonnen hat? |

#### Programmieren des SceneSwitcher.cs

|  |  |
| --- | --- |
| Abbildung 69: SceneSwitcher.cs Variablen | Um zu bestimmen wer gewonnen hat brauche ich beide Players, weil ich beide isDeads brauche um zu bestimmen wer zuerst gestorben ist. Dann brauche ich noch den ButtonManager um in die Szene PlayAgain() zu kommen, damit ich dem Spieler gratulieren kann der gewonnen hat. |

|  |  |
| --- | --- |
| Abbildung 70: SceneSwitcher.cs Start | Auch hier muss man wieder die Komponenten der Player holen um mit der Instanz zu kommunizieren. |

|  |  |
| --- | --- |
| Abbildung 71: SceneSwitcher.cs Update | Jetzt checke ich jeden Frame ob ein Spieler gestorben ist und dann speichere ich es, um das später anzuzeigen, in ein String ein. |

### Die PlayAgain Szene

|  |  |
| --- | --- |
| Abbildung 72: PlayAgain Szene | Ähnlich zur StartsSzene habe ich eine Szene erstellt indem man, wenn man gewonnen hat wieder spielen kann. Die Überschrift ist leer, weil ich hier dem Spieler gratuliere mit dem Code gameDecider.cs. |

### Programmierung des gameDecider.cs

|  |  |
| --- | --- |
| Abbildung 73: gameDecider.cs Variablen | Hier musste ich das using TMPro hinzufügen um TextMeshPro Funktionen nutzen zu können.  Hier braucht man nicht SerializeField machen, weil das Skript direkt auf dem GameObject sitzt. |

|  |  |
| --- | --- |
| Abbildung 74: gameDecider.cs Start | Nun habe ich mir die Komponente geholt um sie manipulieren zu können. |

|  |  |
| --- | --- |
| Abbildung 75: gameDecider.cs Update | Jetzt setze ich in der Update Funktion den String mit dem gewonnen Spieler in die Szene.  Dann frage ich welcher Spieler gewonnen hat um die Farbe des Textes zu bestimmen, das macht das Ganze intuitiver. |

#### Das Resultat des gameDecider.cs

|  |  |
| --- | --- |
| Abbildung 76: Ergebnis gameDecider.cs | Dem Spieler wird in seiner Farbe gratuliert. |

### Hintergrundmusik für jede Szene

|  |  |
| --- | --- |
| Abbildung 77: StartsSzene AudioSource Settings in Unity | Das Hinzufügen einer Hintergrund Musik ist ganz einfach. Man muss einfach nur eine AudioSource hinzufügen und seinen gewünschten Titel hineinziehen.  Play on Awake sollte angeklickt sein damit der Clip direkt abgespielt wird, loopen sollte man auch falls jemand lange im Bildschirm bleibt.  Da jede Szene hier gleichbleibt und nur der Titel sich ändert gehe ich nicht weiter auf die Hintergrund Musik ein. |

### Nicht benutzt: PlayerSpawner.cs

|  |  |
| --- | --- |
|  | Hier habe ich versucht die Spieler random spawnen zu lassen, hat auch funktioniert aber der EdgeCollider hat nicht richtig gezeichnet, wie ich in 3.2.7 in den Fehlern beschrieben habe. Also habe ich das Skript entfernt und die Spieler so in der Szene gelassen. Interessant ist noch die SpawnUpside Methode in der ich ein Spieler mit einer 50% Chance falsch rum gespawnt habe. |

### Nicht benutzt: CameraShaker.cs

|  |  |
| --- | --- |
|  | Im Camera Shaker wollte ich die Kamera bei einem Tod des Spielers leicht zittern lassen aber aus einem mir nicht ersichtlichen Grund ist die Kamera dann nicht zurück in ihre originale Position, also habe ich das Skript rausgenommen. |

## Aufgetretene Fehler

### Fehler im Player Script

Ein großes Problem ist heraus zu finden ob die Maus sich irgendwann bewegt oder nicht, das geschieht teilweise sehr ungenau und es gibt auch wenig im Internet zu finden. Das zweite Problem beim Movement war, dass ich versucht habe mit der Funktion Velocity und AddForce zu arbeiten, diese sind aber für meine Vorhaben ungeeignet. Was mich zwei Tage gekostet hat war später mit der Funktion MoveTowards innerhalb zwei Minuten zu erledigen.

### Fehler im Food Script

Das Problem besteht hierbei die Punkte zu beschränken das nicht zu viele auf dem Spielfeld sind, vor allem in Verbindung auch immer andere Prefabs zu laden die andere Eigenschaften und Farben aus einer Liste holen.

### Fehler mit Collider

Meine Collisions triggern nicht, weil es diesen Error gibt:

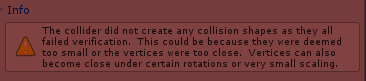


Abbildung 78: Fehler mit Collider

### Fehler wenn ich Schriftart hinzufüge



Abbildung 79: Fehler Schriftart

### Rendering von Texten

Ein Fehler hatte ich mit dem Rendering von Texten hier hilft ein Addon aus dem Assetstore aus Unity. Man sieht im Bild sehr gut den Unterschied, wenn man ran zoomt:

Das Obere: TextMeshPro

Das Untere: Normaler Unity-Text

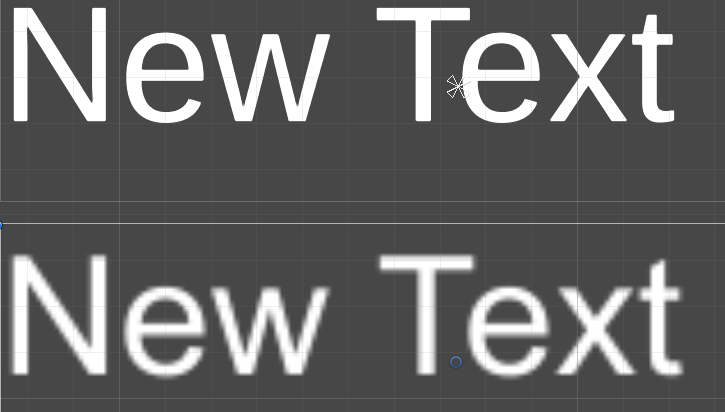


Abbildung 80: Rendering von Text

### Schwerwiegender Fehler: Namensgebung ist wichtig!

|  |  |
| --- | --- |
| Abbildung 81: SceneManager Namensgebungfehler | Das hat mich viel Zeit gekostet.  Wenn man genau hinschaut sieht man das ich ausversehen das Script so genannt habe wie die Methode die ich callen wollte. Das heißt SceneManager konnte nicht aus der Bibliothek geholt werden, weil es schon existierte.  So ein fast schon lächerlicher Fehler kann einen das ganze Projekt zerstören, weil man nicht weiterkommt.  Ein simples umbenennen des Skriptes behob den Fehler. |

### Edge Collider zeichnet nicht richtig

|  |  |
| --- | --- |
| Abbildung 82: Edge Collider zeichnet falsch | Der Edge Collider zeichnet dieselbe Strecke wie die Schlange, aber hier macht er es mit einem leichten Offset. |

### Missing Reference Exception

|  |  |
| --- | --- |
| Abbildung 83: Missing Reference Exception | Der Fehler passiert, weil nach dem zerstören des GameObjectes, ich immer noch versuche auf die Liste zu zugreifen. |

## Test

### Testdurchführung

Ich habe das Spiel mit meiner Freundin zusammen getestet. Uns ist aufgefallen das man das Spiel nicht schließen kann, dass die Musik zu laut war und dass die Tastatur beim Player2 invertiert war.

# Projektabschluss

## Projektübergabe

Das Projekt ist bis auf den Online-Multiplayer auf dem Rasperry Pi vollständig.

## Abnahmeprotokoll

Das einzige was nicht realisiert worden ist, ist der Server auf dem Rasperry Pi und die Multiplayer Funktion. Das habe ich gelöst indem man zu zweit an einer Tastatur spielen kann.

## Projektbewertung

Soll-Ist-Vergleich:

Soll: Funktionsfähiges Unity Spiel mit Multiplayer-Funktion auf dem Raspberry Pi

Ist: Funktionsfähiges Unity Spiel mit Multiplayer-Funktion an Tastatur

# Quellenverzeichnis

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

StartsSzene Hintergrund: <https://flipwallpapers.com/retro-wallpaper-high-resolution/>

Hintergrund Musik von Race to Mars.mp3 und DOS-88 – Far away.mp3: <https://dos88.itch.io/dos-88-music-library>

Hintergrund Musik StartSzene M.O.O.N. – Dust: <https://www.youtube.com/watch?v=DpRUKLKCY-w>

Verwendete Schriftart: <https://www.dafont.com/de/furiosa.font>

Invincibility Star.mp3 aus Mario: <https://downloads.khinsider.com/game-soundtracks/album/super-mario-bros>

Kenney Assets Space Shooter Redux für Raumschiff, Countdown, Pille, Stern und Meteoren: <https://kenney.nl/assets/space-shooter-redux>

# Persönliche Erklärung

Ich, Benjamin Derksen, bestätige das, dass Projekt von mir erstellt worden ist.