Dokumentation-

Spielentwicklung in Unity

Tower Defense Ultimate

Jaron Liskow und Tim Fischer

Seminarkurs 11 Bindestrichinformatik

Inhalt

[Erste Anfänge 4](#_Toc187008388)

[Erster grober Entwurf des Projekts. 4](#_Toc187008389)

[Tim Vorkenntnisse: 5](#_Toc187008390)

[Jaron Vorkenntnisse: 5](#_Toc187008391)

[Ziele: 5](#_Toc187008392)

[Schritte (Planung & Arbeit): 6](#_Toc187008393)

[Entwicklungsumgebung: 6](#_Toc187008394)

[Versionsverwaltung/Zusammenarbeit: 6](#_Toc187008395)

[Erstes Ziel (Tim): 6](#_Toc187008396)

[Einarbeitung 6](#_Toc187008397)

[Vorgehensweise 6](#_Toc187008398)

[Was macht der Code? 7](#_Toc187008399)

[Erstes Ziel (Jaron): 7](#_Toc187008400)

[Vorbereitung: 8](#_Toc187008401)

[Teil 1 - mein Script: 8](#_Toc187008402)

[Teil 2 - alles neu 9](#_Toc187008403)

[Zweites Ziel (Tim) 9](#_Toc187008404)

[Teil 1 - Leben von Gegnern 10](#_Toc187008405)

[Teil 2 - Schaden an der Base 12](#_Toc187008406)

[Zweites Ziel (Jaron) 12](#_Toc187008407)

[Teil 1: Was muss ich überhaupt machen? -- Position ermitteln 13](#_Toc187008408)

[Teil 2: Türme auswählen können 14](#_Toc187008409)

[Teil 3: Nicht nur Platzieren sondern auch Bewegen (Uiii spannend) 15](#_Toc187008410)

[Teil 4: Reichweitenvisualisierung 17](#_Toc187008411)

[Drittes Ziel (Tim) 18](#_Toc187008412)

[Vorgehensweise 18](#_Toc187008413)

[Was macht der Code? 20](#_Toc187008414)

[Viertes Ziel (Tim) 20](#_Toc187008415)

[Vorgehensweise 20](#_Toc187008416)

[Drittes Ziel (Jaron) 21](#_Toc187008417)

[Reflektion Q1 21](#_Toc187008418)

[Tim 21](#_Toc187008419)

[Probleme 21](#_Toc187008420)

[Tim: 22](#_Toc187008421)

[Jaron: 22](#_Toc187008422)

[Vorarbeit Q2 (Nicht Q1) 22](#_Toc187008423)

[Fünftes Ziel (Tim) 22](#_Toc187008424)

# Erste Anfänge

**Struktur**

* **Projektbeschreibung und Einleitung**
  + Vorstellung des Projekts und eigener Person.
  + Eigene Vorkenntnisse und Projektideen.

# Erster grober Entwurf des Projekts.

* **Ziele des Projekts** *//siehe Oben – Noch nicht strukturiert*
  + Aufteilung der Ziele in:
    - **Obligatorische Ziele** (unbedingt erforderlich)
    - **Optionale Ziele** (zusätzliche Erweiterungen)
  + Unterteilung der Ziele in:
    - **Grobziele** (allgemein formulierte Teilziele)
    - **Feinziele** (detaillierte Arbeitsschritte zum Erreichen der Grobziele)
* **Planung und Arbeitsschritte**
  + Aufgliederung in einzelne Schritte und Aufgabenbereiche. *//siehe oben*
  + Mögliche Checkliste zum Abhaken abgeschlossener Schritte.
* **Entwicklungsumgebung** 
  + Vorstellung der genutzten Tools, Programmiersprachen, und anderer Ressourcen.
  + Begründung der Wahl der Entwicklungsumgebung.
* **Fortschrittsdokumentation**
  + Beschreibung der regelmäßigen Fortschritte im Projekt.
  + Darstellung und Erklärung des verwendeten Codes.
  + Lösung aufkommender Probleme und Herausforderungen.
* **Reflexion der Projektarbeit**
  + Jedes Halbjahr Reflexion des eigenen Arbeitsprozesses.
    - Eigene Motivation und mögliche Schwankungen.
    - Herausforderungen und Hindernisse im Projektverlauf.
    - Vorgehensweise bei der Lösung von Problemen.
  + Zielerreichung und Zeitmanagement:
    - Welche Ziele wurden schneller oder langsamer als erwartet erreicht?
    - Gründe für Abweichungen im Zeitplan und den Erfolg der Erreichung

## Tim Vorkenntnisse:

Informatik Leistungskurs, Vorkenntnisse aus der 10. Klasse Informatik Schwerpunkt, etwas Erfahrung mit Java durch den Java – Hamster (Grobe Struktur der Sprache), Erfahrung mit Spielprogrammierung durch Snap.

## Jaron Vorkenntnisse:

Informatik Leistungskurs, Vorkenntnisse aus der 10. Klasse Informatik Schwerpunkt, etwas Erfahrung mit Java durch den Java – Hamster (Grobe Struktur der Sprache), Erfahrung mit Spielprogrammierung durch Snap, leichte Vorkenntnisse in Unity, Vorwissen in C++.

# Ziele:

* Obligatorisch:
  + 3rd Person (Jaron)
    - Spielfigur
    - Steuerung der Figur mit Maus und WASD
    - Steuerung der Kamera durch Maus und Spielfigur
      * Mausempfindlichkeit
  + Das erste Level (Grundstruktur für In Game)
    - Funktionierendes Gesundheitssystem (Anzeige)
      * Base mit Lebenspunkten
      * Wenn die Base zerstört ist, ist das Spiel verloren
      * Gegner mit Lebenspunkten
      * Wenn ein Gegner stirbt, erhält der Spieler Münzen
    - Gegner Bewegen sich auf einem Pfad zur Base und fügen Schaden zu, wenn sie an der Base ankommen
    - Wellen von Gegnern und gruppiertes Spawning
    - Türme schießen auf Gegner und fügen denen Schaden zu
      * Reichweite der Türme
      * Projektile bewegen sich auf den nächsten Gegner zu
    - Platzieren und Kaufen der Türme durch Spieler
    - Türme können geupgradet werden
  + UI
    - Startbildschirm & „Endbildschirm“ (Game Over)
    - Anzeige der Leben, verfügbarer Münzen, Türme etc.
  + Weitere Level
    - Erweiterte UI
      * Levelauswahl
    - Verschiedene Gegner und Türme mit Unterschiedlicher Animation und unterschiedlichen Eigenschaften
    - Weitere Pfade in einem Level
  + Namen für Spiel
* Optional:
  + 3dimensionale Bewegung des Spielers
    - Hoch – runter Bewegung mit Maus oder Tastatur
  + Level mit 3dimensionaler Bewegung der Gegner und Platzierung der Türme

# Schritte (Planung & Arbeit):

Entwicklungsumgebung:

Wir haben uns dafür entschieden, eine Game Engine zu benutzen, die viele Tools bereitstellt, weil es somit einfacher ist, komplexe Spiele zu programmieren. Dabei haben wir uns zunächst für Unreal Engine entschieden, haben jedoch relativ schnell zu Unity gewechselt, weil Unity bekannter ist, es dadurch mehr Tutorials gibt und dadurch anfängerfreundlicher ist. Mit Unity programmieren wir in C# mithilfe von Visual Studio (2022). - Tim

## Versionsverwaltung/Zusammenarbeit:

Um zu ermöglichen, dass Tim und ich beide an unseren PCs am Spiel arbeiten können und nicht immer einen USB Stick oder so tauschen müssen, habe ich mich entschieden, Github für die Versionsverwaltung zu nutzen. Also habe ich ein Public Repository erstellt und Tim als Mitbearbeiter eingeladen. Im Folgenden habe ich ein Projekt in Unity erstellt und die kompletten Projektdateien in das Repository hinzugefügt. Danach musste Tim nur noch die Dateien herunterladen und das Projekt in Unity öffnen. Ich habe einfach ein Main Branch erstellt, um alles so simpel wie möglich zu halten. Jetzt können wir immer problemlos am der neusten Version des Spiels arbeiten. - Jaron

## Erstes Ziel (Tim):

### Einarbeitung

Meine erste Aufgabe bestand darin, die erste Funktion für unsere Gegner zu programmieren, sie sollten also zunächst in einem bestimmten Intervall an einer bestimmten Position geklont werden und von dort aus ein Ziel verfolgen. Um anzufangen musste ich mich vorerst in Unity einarbeiten. Dafür habe ich erste Dinge in einem anderen Projekt getestet und mich mit der Umgebung vertraut gemacht. Geholfen haben mir dabei YouTube – Tutorials und ChatGPT. Gestellte Prompts waren zum Beispiel *„Wie Programmiere ich in Unity?“* oder *„Wo finde ich den Unity Editor?“* auch erste Skriptideen, wie: *„ich will, dass eine Kugel sich im Spiel hin und her bewegt“.* Ich habe recht schnell festgestellt, dass das Erstellen von einfachen Skripts recht schnell mit ChatGPT geht und wusste, dass ich weiterhin viel mit ChatGPT arbeiten werde. Da jedoch die Skripts von ChatGPT nicht immer das tun, was man erreichen will, war ich nach kurzer Zeit doch dazu gezwungen, zu verstehen, was der Code von ChatGPT macht und musste mich mit den Befehlen in den Skripten auseinandersetzen. Dazu habe ich ChatGPT Fragen gestellt wie: *„Was bedeutet 3f“* (Das f stellte sich als ein Verweis auf den Datentyp float heraus) oder *„was macht public und was macht private?“.* Ich habe zuerst Stunden damit verbracht, einfache Befehle zu verstehen und anwenden zu können. Schließlich habe ich mich daran getan, zum Einstieg das Sonnensystem zu simulieren. Wichtige Befehltypen dafür waren: „transform“ und „Mathf“

### Vorgehensweise

Um schließlich mit meiner Aufgabe zu beginnen habe ich in einer neuen Szene („PathEditor“) einen Zylinder namens Spawner erstellt, an dem die Gegner starten sollen, einen Quader namens Base2 an dem die Gegner schlussendlich ankommen sollten, und zwei weitere Objekte (einen Zylinder, Bot1 und eine Kugel, Bot2) erstellt. Nun sollten die bots, die in bestimmten Abständen geklont werden, von Spawner zur Base gelangen. Umgesetzt habe ich dies mit einem Skript, das ich mithilfe von ChatGPT und folgendem Video: [6 Ways to move object in Unity [Tutorial] - YouTube](https://www.youtube.com/watch?v=-thhMXmTM7Q&ab_channel=FeriFm) erstellt hab. Dieses Skript heißt BotsOnPath und wird auf den Spawner angewandt. Zuerst wollte ich das Spawnen der Bots hinbekommen und habe ChatGPT folgenden Prompt gegeben: *ich will einen Zylinder erstellen, der an einem bestimmten Punkt spawned, 8 Schritte nach vorne geht, verschwindet und am Ausgangspunkt wieder ein neuer Zylinder spawned, der wieder 8 Schritte nach vorn geht usw.* Er hat mir erklärt, dass ich die Bots zu Prefabs machen muss und mir den Code für den Prompt gegeben, bei dem wieder mehrere Befehle, von denen ich noch nie etwas gehört habe verwendet wurden. Zum Beispiel „Quanternion“ und „StartCoroutine“. Nachdem ich den Code verstanden habe wollt ich wissen, *„wie ich […] ein Objekt zu einem anderen Objekt laufen lassen?“* kann, um mein Ziel zu erreichen (von Spawner zu Base). Das hab ich jedoch nicht weiter verwendet, weil ich stattdessen auf ein Tool in dem Assetstore von Unity gestoßen bin, mit dem man kurvige Pfade in der Unity Scene erstellen kann, den Bézier Path Creator (siehe [Bézier Path Creator | Utilities Tools | Unity Asset Store](https://assetstore.unity.com/packages/tools/utilities/b-zier-path-creator-136082)). *„Wie kann ich einen weg [mithilfe des Path Creators ] erstellen? Ich kann ihn bis jetzt nur im Spielmodus verwenden, aber nicht wirklich einen Pfad erstellen“*, ChatGPT kannte das Tool und hat mir erklärt, wie ich einen Pfad erstelle und wie ich ein Objekt entlang des Pfades bewegen kann:

void Update() {

distanceTravelled += speed \* Time.deltaTime;

// Setze die Position des Objekts entlang des Pfades

transform.position = pathCreator.path.GetPointAtDistance(distanceTravelled);

}

Diesen Code baute ich dann in den vorherigen Code ein, sodass die Bots nun entlang des Pfades in Richtung Base laufen, in einem bestimmten zeitlichen Abstand zueinander spawnen und wenn sie an der Base angekommen sind, zerstört werden-

### Was macht der Code?

Zu diesem Zeitpunkt machte das Skript BotsOnPath folgendes: es bekommt 3 Public – GameObjects vorgegeben, den Pfad, eine Variable „moveSpeed“ und eine weitere Variable „wait“. In dem Scirpt gibt es bisher keine „void Update()“ – Funktion, sondern nur eine „void Start“ – Funktion. Diese führt eine Coroutine (SpawnBots) aus, die mithilfe einer weiteren Funktion (GetBotToSpawn()), zwischen den public GameObjects durchwechselt und diese mithilfe einer weiteren Funtkion (SpawnNewBot) das zu spawnende GameObject am Anfang des Pfades instanziiert, für „wait“ – Sekunden wartet, den nächsten Gegner instanziiert usw. und mithilfe einer neuen Coroutine (MoveBotsAlongPath) jeden Gegner, sobald er instanziiert wurde, abhängig von der Variable „moveSpeed“, in einer bestimmten Geschwindigkeit den Pfad entlang laufen lässt, bis der Pfad zu Ende ist. Am Pfadende wird der Gegner zunächst zerstört.

## Erstes Ziel (Jaron):

Meine erste Aufgabe war es, einen gut funktionierenden und anpassbaren third-person controller zu erstellen.

### Vorbereitung:

Auf Youtube habe ich ein sehr informatives Video zu meinem Thema gefunden: <https://www.youtube.com/watch?v=4HpC--2iowE>

In diesem Video wurde ein einfacher Controller mithilfe von Cinemachine erstellt. Dieses Konzept habe ich teilweise übernommen, aber noch für unsere Bedürfnisse abgewandelt.

### Teil 1 - mein Script:

Ganz zu Beginn des Scripts habe ich den Animator für die Charakteranimation, 3 Floats für die Kamerasteuerung und die Kamera initialisiert. In der void start Methode wird nochmal der Animationcontroller des Charakters mit folgender Zeile im Code definiert: anim = GetComponent<Animator>();

So wird vom GameObject das Element „Animator“ gesucht und als anim im Script definiert. In der void update Methode, die jeden Frame einmal abläuft, wird zuerst Unitys-Inputsystem mit zwei Floats geupdatet:

float horizontal = Input.GetAxisRaw("Horizontal");

float vertical = Input.GetAxisRaw("Vertical");

Danach wird ein Vector erstellt, der die Beiden Bewegungsrichtungen kombiniert.

Vector3 direction = new Vector3(horizontal, 0f, vertical).normalized;

In der folgenden Schleife wird abgefragt, ob der Vector direction eine Änderung im Wert erhält, wenn ja wird der Kamerawinkel angepasst.

if (direction.magnitude >= 0.1f)

{

float targetAngle = Mathf.Atan2(direction.x, direction.z) \* Mathf.Rad2Deg + cam.eulerAngles.y;

float angle = Mathf.SmoothDampAngle(transform.eulerAngles.y, targetAngle, ref turnSmoothVelocity, turnSmoothTime);

transform.rotation = Quaternion.Euler(0f, angle + mouseX, 0f);

Vector3 moveDir = Quaternion.Euler(0f, targetAngle, 0f) \* Vector3.forward;

controller.Move(moveDir.normalized \* speed \* Time.deltaTime);

}

Um ehrlich zu sein, habe ich diesen Code nur aus dem Video abgeschrieben, da die Syntax sehr komplex ist. Der nächste Teil ist komplett von mir geschrieben und kontrolliert die Animationen während Tastendrucks:

if (Input.GetKey(KeyCode.A) || Input.GetKey(KeyCode.W) || Input.GetKey(KeyCode.S) || Input.GetKey(KeyCode.D))

{

anim.SetFloat("Blend", 1.0f, 0.1f, Time.deltaTime);

}

else

{

anim.SetFloat("Blend", 0.0f, 0.1f, Time.deltaTime);

}

Die Blend Variable bestimmt im Animationscontroller den Übergang zwischen der Idle- und der Laufanimation.

Danach kommt noch eine Abfrage, ob Shift gedrückt wird, dann wird die Speedvariable hochgesetzt und der Charakter bewegt sich schneller. Auch habe ich noch Methoden erstellt, um den Cursor auf dem Screen zu locken oder nicht, diese werden aber nicht mehr verwendet. Nachdem das Script fertig war, konnte ich in Unity alles richtig konfigurieren. Als erstes brauchte ich einen simplen Charakter zu testen, diesen habe ich aus dem Asset Store heruntergeladen. Die Kamerabewegung musste über ein neues leeres GameObject gesteuert werden, welches als FreeLook Kamera von Cimemachiene instanziiert wird. Damit wird auch automatisch die Main Kamera an die Kamerabewegung gebunden, damit man die Bewegung auch sieht. Danach musste ich nur noch den Bewegungsraum für die Kamera einstellen, ein Smoothing hinzufügen und die Kamera war theoretisch bereit. Jedoch hatte ich immer noch Probleme mit der Mausgeschwindigkeit und Kamerklipping. Mit ChatGPT habe ich versuch das Geschwindigkeitsproblem zu lösen, er hat aber die Kamerabewegung von der Bewegung des Charakters getrennt, wonach die Bewegungsrichtung des Charakters nicht mehr mit der Kamerabewegung synchronisiert war.

### Teil 2 - alles neu

Aufgrund von eben genannten Problemen, habe ich mich entschieden den Third Person Controller aus dem Asset Store herunterzuladen.

<https://assetstore.unity.com/packages/tools/game-toolkits/third-person-controller-basic-locomotion-free-82048>

Dieses Tool hat es mir erlaubt, einfach einen fertigen Charakter und ein fast fertiges Script ins Spiel eizuarbeiten. Fast fertig war es nur, da ich umsetzen wollte, dass die Kamera sich nur bewegt solange man die rechte Maustaste gedrückt hält. Also musste ich mich in den Aufbau des Scripts einlesen und eine if-Abfrage an entsprechender Stelle ergänzen.

Danach habe ich noch den Charakter im Spiel hochskaliert. Nun musste ich die besten Werte für Laufgeschwindigkeit, Sprunghöhe und Länge und mehr testen und finden.

## Zweites Ziel (Tim)

Meine zweite Aufgabe war ein Schadenssystem zu erstellen. Das heißt, jeder Gegner benötigt eine Lebensanzeige, die Base benötigt eine Lebensanzeige und die Lebensanzeige muss „weniger“ werden, wenn das Objekt Schaden nimmt.

### Teil 1 - Leben von Gegnern

#### Vorgehensweise

Zuerst habe ich ChatGPT gefragt, *„wie kann ich in Unity eine Healthbar erstellen?*“, da seine Erklärung jedoch für mich noch sehr unverständlich und kompliziert war, habe ich mir ein Tutorial auf YouTube rausgesucht: [How to make a HEALTH BAR in Unity!](https://www.youtube.com/watch?v=BLfNP4Sc_iA&ab_channel=Brackeys).

Nachdem ich dem Tutorial Schritt für Schritt gefolgt bin und ein paar Anpassungen vorgenommen hatte, hatte ich eine Szene, mit einer Healthbar, die Schaden bekommt, sobald ich „E“ drücke. Das Tutorial gab mir auch Hinweise darauf, wie ich die Healthbar an ein Gameobject anfügen kann, sodass es sich in der Scene mit dem Gameobject mitbewegt. In meiner Testszene (Healthbar), hatte ich aber nur dir Healthbar, also eine Canvas, mit einem Gamobject, welches einen Slider und ein „HealthSlider“ – Script besitzt und drei weiter Gamobjects (Fill Area, Fill und Background), die für das Design der Healthbar verantwortlich sind. Außerdem, hatte die Scene ein „DamageTest“ – Script. Dieses war zunächst dafür gedacht, um zu testen, ob die Healthbar funktioniert.

Um die Healthbar nun mit den Gegnern zu kombinieren, kopierte ich die Canvas mit ihren Children (Objekte, die einem anderen Objekt zugeordnet sind und sich verändern, wenn man das übergeordnete Objekt (Parent) verändert) und fügte sie in die „PathEditor“ – Scene ein. Danach erstellte ich einen „TestEnemy“, der zunächst kein Prefab war und hauptsächlich zur Formatierung der Healthbar über dem Gegner diente. Aus dem eben erwähnten Video, entnahm ich auch den Code, der die Healthbar immer zur Kamera gucken lässt und übertrug ihn auf ein Skript namens „Billboard“, das auf der Canvas der Healthbar liegt. Mithilfe eines „CameraRotate“ – Skripts, dass auch ChatGPT für mich erstellt hat *(wie kann ich die Main Kamera um die Scene drehen lassen?)* konnte ich die Funktion des Billboards testen. Um den „TestEnemy“ schließlich zu einem Prefab zu machen, musste ich im Billboard – Skript direkt auf die Main Kamera verweisen, da, sobald durch das BotsOnPath – Skript ein Klon von dem GameObject erstellt wird, der Public Verweis nicht erhalten blieb und eine „MissingReference“ – Exception entstanden ist (Die Kamera war als „public Gameobject“ in dem Billboard definiert, sodass man sie für jedes Gamobject im Inspector definieren muss).

Nun musste ich nur noch den Prefab für BotsOnPath als Public GameObject definieren und hatte damit Gegner mit einer Lebensanzeige, die einen Pfad entlanglaufen und Schaden nehmen, wenn der Spieler „E“ drückt.

#### Was macht der Code?

Die Skripts, die für das Spiel relevant sind, sind das DamageTest – Skript, das Billboard – Skript, welches nur sagt, dass nach jedem Frame (LateUpdate), die Healthbar sich so drehen soll, dass sie zur Hauptkamera guckt, und das HealthSlider -Skript. Auf das CameraRotate – Skript werde ich nicht eingehen, da dies nur zum Testen der Billboard Funktion diente.

DamageTest: Dieses Skript besitzt 3 Public Variablen, public int currentHealth; public int maxHealth und public HealthSlider healthbar; (Referenz zur Lebensanzeige), eine void Start() Funktion, eine public void TakeDamage(int damage) Funktion und zu diesem Zeitpunkt noch eine Update Funktion.

Die Start Funktion setzt die currentHealth auf den Wert von maxHealth und setzt den maximalen Wert der Healthbar des Objekts, auf dem das DamageTest Skript liegt, auf den Wert maxHealth.

Die TakeDamage Funktion fügt dem Objekt, auf dem das DamageTest Skript liegt schaden zu. Es setzt also den Wert von currentHealth auf currentHealth – damage, aktualisiert die HealthBar und zerstört das Objekt, wenn currentHealth kleiner gleich Null ist.

In der Update Funktion steht, dass wenn „E“ gedrückt wird, TakeDamage(20) ausgeführt werden soll.

HealthSlider: Das HealthSlider-Skript zu schreiben, war nicht allzu schwer, da vieles ChatGPT für mich getan hat. Die größte Schwierigkeit bestand darin, das Skript im Inspector richtig anzuwenden, sodass die Funktionen des Skripts wie gewünscht funktionieren. Den Code, den mir ChatGPT gegeben hat, sah folgendermaßen aus:

public class HealthSlider : MonoBehaviour

public class HealthSlider : MonoBehaviour

{

public Slider slider; // Referenz auf den Slider

public Gradient gradient; //Farbe

public Image fill;

// Setze die maximale Gesundheit im Slider

public void SetMaxHealth(int health) // Wie viel soll der Slider MAXIMAL anzeigen?

{

slider.maxValue = health;

slider.value = health;

fill.color = gradient.Evaluate(1f);

}

// Aktualisiere die aktuelle Gesundheit im Slider

public void Sethealth(int health) // Wie viel soll der Slider ANZEIGEN?

{

slider.value = health;

fill.color = gradient.Evaluate(slider.normalizedValue);

}

}

An diesem Code habe ich lediglich die Kommentare geändert (diese Kommentare sind bereits meine Kommentare). Der Code sagt kurzgesagt, dass wenn SetMaxHealth(int health) aufgerufen wird, der maximale Wert der Healthbar auf health gesetzt wird und, wenn Sethealth(int health) aufgerufen wird, der aktuelle Wert von der Healthbar auf health gesetzt wird.

Um das Skript zum Funktionieren zu bringen, musste ich einen Gradient erstellen, diesen als Public Gradient für das Skript definieren, auf das richtige Image hinweisen, auf den richtigen Slider im Inspector hinweisen und im DamageTest-Skript richtig auf die Funktionen hinweisen. In diesem Skript musste ich zuerst verstehen, welche Begriffe vorgegebene Begriffe für etwas sind, also ob Gradient auch anders heißen könnte und ob gradient auch anders heißen könnte, oder ob das jeweils Datentyp von C# sind, wie zum Beispiel „int“. „*kann ich Gradient gradient auch zu Hose hose umbenennen, wenn ich im Inspector auf den richtigen Gradient hinweise*“, es stellte sich heraus, dass ich zwar Gradient nicht zu Hose ändern kann, weil es ein Datentyp ist, aber dafür gradient zu hose ändern könnte, weil dies nur der Name, der für den Datentyp im Skript verwendet wird.

### Teil 2 - Schaden an der Base

Bislang gibt es Gegner, die einen Pfad entlanglaufen und Schaden bekommen können, der im Spiel über den Gegnern angezeigt wird. Es fehlen jetzt die Leben der Base.

#### Vorgehensweise

Die Base benötigt ein Skript, wie das DamageTest-Skript und ein HealthSlider-Skript, um eine Lebensanzeige wie die Gegner zu besitzen. Da wir das DamageTest-Skript für die Base später wahrscheinlich anpassen müssen, ohne das DamageTest-Skript von den Gegnern zu verändern, erstellte ich ein neues Skript, welches genau die gleiche Funktion wie das DamageTest-Skript hatte und nannte es „BaseHealth“.

Zur einfacheren Formatierung bzw. Bearbeitung des Pfades, habe ich die Base als ein Children zu dem Path- Gameobjekt zugefügt, sodass nun alles, das auf dem Pfad stattfindet unter einem Parent (Path) steht. Dazu gehört der Pathspawner, auf dem vor allem das BotsOnPath Skript liegt, der PathCreator mit dem Pfad und die Base mit deren HealthBar und den dazugehörigen Skripts.

Zudem sollte die Base Schaden bekommen, wenn sie von einem Gegner getroffen/ berührt wird. Um in Unity zu erfassen, ob sich zwei Objekte berühren oder miteinander Kollidieren, müssen beide Objekte einen Collider besitzen. Ich habe also der Base einen Box-Collider gegeben, da die Base die Form eines Quaders hat, und dem Gegnerprefab einen Capsule-Collider, da dieser die Form einer Kugel hatte. --[Unity 3D Collisions, Colliders, & Hitboxes (In 3 Minutes!!) - YouTube](https://www.youtube.com/watch?v=mkErt53EEFY&ab_channel=RoyalSkies)—Außerdem benötigt mindestens eines der Beiden Objekte einen Rigidbody, welchen ich auf das Gegnerprefab gelegt habe.

Mit Hilfe des Videos und ChatGPT *(„ich will, dass die clone von den bot prefabs der Base Schaden zufügen“)* habe ich mir ein Skript namens MainDamage zusammengebastelt, das auf jedem Gegnerprefab liegt, damit unterschiedliche Gegner unterschiedlich viel Schaden zufügen können. Ich hatte die Schadenfunktion zuerst auf der Base liegen, habe jedoch festgestellt, dass dadurch alle Gegner gleich viel Schaden zufügen.

#### Was macht der Code?

Wie bereits beschrieben, ist die Schadensanzeige der Base und die BaseHealth nicht viel anders, als die Skripts, die ich bereits beschrieben habe, mit der Ausnahme, dass der Gradient als erstes ein anderer war, der besitzt jedoch jetzt wieder die Farben, wie der der Gegner. Die einzige Änderung am Code ist also das MainDamage Skript.

In diesem wird zuerst eine Variable Damage definiert, die die Anzahl des Schadens angibt und in der Start Methode, wird auf das GameObject „Base2“ hingewiesen. Das Skript besitzt außerdem eine OnCollisionEnter Methode. Eine vorgegebene Methode, die jedes Mal, wenn das Objekt mit einem anderen Objekt mit Collider (Deshalb benötigen sowohl die Base, als auch die Gegner einen Collider) kollidiert, ausgeführt wird. In dieser wird, wenn das Objekt, mit dem der Gegner kollidiert, „Base2“ heißt, der Base damage Schaden zugefügt und danach das Gameobjekt (also der Gegner) zerstört, sodass der Base nicht jeden weiteren Frame damage Schaden zugefügt wird.

## Zweites Ziel (Jaron)

Meine nächste Aufgabe, und auch die, an der mit Abstand am längsten gesessen habe, war die Entwicklung eines Systems zur Platzierung von Gegenständen im Spielbereich. Dies soll ermöglichen, dass der Spieler sein letztendliches Ziel erfüllen kann, die Gegner zu töten. Und ohne irgendwelche Türme hat er nichts, was schießen kann. Diese Türme müssen also irgendwie platziert werden.

Zu Beginn dieses Teils muss ich sagen, dass ich viele Iterationen meines Codes leider nicht gespeichert habe und nur noch die finalen Versionen zur Verfügung habe. Ideen, die ich zwischendurch hatte und ich letztendlich nicht umgesetzt habe, kann ich leider nicht mehr detailliert anbringen. Ich werde jedoch immer an gegebener Stelle skizzieren, welche Ideen und Gedanken ich hatte.

### Teil 1: Was muss ich überhaupt machen? -- Position ermitteln

Das große Ziel war mir natürlich die gesamte Zeit bewusst. Zu Beginn habe ich mich hauptsächlich informiert, wie ich Positionsdaten abrufen kann. Dadurch, dass wir uns für eine Third Person Umgebung entschieden haben, hat sich im Verlauf sehr viel als sehr viel schwieriger als erwartet entpuppt. (Ja komischer Satz 😉) Recht schnell war mir klar, dass ich nicht an einen Raycast als Methode der Positionsbestimmung vorbeikommen würde. Youtube sei gepriesen, so kam ich an ein Tutorial, welches mir ein sehr gutes Verständnis für dieses Thema gegeben hat. So habe ich mich dazu entschieden, einen externen „SpawnHandler“ als ein neues GameObject zu definieren, der all die Logik und Positionsberechnungen zentral verwaltet. Das habe ich auch getan, damit nicht jeder Turm, der im Spiel gespawnt wird, im Hintergrund das Skript zu laufen hat und nur ein GameObject die Positionsdaten an die entsprechenden Prefabs weitergibt und bestimmt, wo diese Initialisiert werden.

void Update()

{

// Wenn das Spawning aktiviert ist und die linke Maustaste gedrückt wird

if (spawnEnabled && Input.GetMouseButtonDown(0))

{

if (gameManager.SpendCredits(cost)) // Tims Änderung -- wenn der Preis bezahlbar ist.

{

// Raycast von der Kamera zum Mauszeiger

Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

RaycastHit hit;

// Wenn der Raycast etwas trifft

if (Physics.Raycast(ray, out hit))

{

// Spawnen des ausgewählten GameObjects an der Hit-Position

GameObject spawnedObject = Instantiate(prefabsToSpawn[selectedPrefabIndex], hit.point + new Vector3(0, 3, 0), Quaternion.identity);

Dies ist ein Auszug aus dem Skript, welches jeden Frame des Spiels läuft (void Update Methode). Ein von mir gesetztes Ziel war es, den Prozess des Spawnens des Turms erst zu starten, wen die linke Maustaste gedrückt wurde. Dies geschieht in der ersten Zeile, gemeinsam mit der Abfrage ob ein gesetzter Boolean „SpawnEnabled“ true ist. Dieser Wert wird während des Drückens auf den Knopf auf der UI true gesetzt und die If-Abfrage wird ausgeführt. Danach folgt eine Abfrage, ob der Turm auch bezahlt werden kann, dies gehört aber zu Tims Bereich. Jetzt wird’s spannend. Mit „Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);“ wird ein simulierter Strahl von der Perspektive des Spielers in das Spiel, auf den Boden, geschossen. Die Referenz zu der Camera.main ist unabdingbar, da der Strahl ansonsten nicht wüsste, wo er starten soll. So jetzt wo der Strahl abgeschossen wurde, muss nur noch geschaut werden, wo er auftrifft. Das gestaltet sich aber nicht so einfach. So soll der Turm ja bspw. nicht im Spieler oder anderen Türmen gespawnt werden. Demzufolge müssen alle Objekte, die nicht der Boden sind, so im Inspektor konfiguriert werden, dass sie vom Raycast Hit ignoriert werden. Jetzt wo der Raycast/Strahl den Boden und nix anderes getroffen hat, kann ein Hitpoint erstellt werden. Dies erfolgt aber nicht so wie man es denkt. Unzwar wird erst mit „if (Physics.Raycast(ray, out hit))“ geprüft, ob der Strahl etwas trifft und dann wird mit diesen Parametern direkt ein GameObject instanziiert.

GameObject spawnedObject = Instantiate(prefabsToSpawn[selectedPrefabIndex], hit.point + new Vector3(0, 3, 0), Quaternion.identity);

Das prefabsToSpawn[selectedPrefabIndex] ist hierbei das aus einem Array ausgewählte GameObject, welches einem Knopf von der Auswahl der Türme zugeordnet ist. Wie das funktioniert erkläre ich später. Der Hitpoint wird nun mit einem Offset versetzt, damit der Turm auch auf dem Boden, und nicht im Boden gespawnt wird.

Danach werden wird erstens der Name des gespawnten Objekts verändert, ein Tag vergeben und ein Boolean spawned auf true gesetzt. Dies geschieht nur, um das Objekt für andere Skripte klar erkennbar zu machen.

Zusammenfassen können wir für diesen Teil also, dass irgendetwas weiß, wohin es muss. Aber nur wenn es darf und bezahlt werden kann. An der Stelle muss ich nochmals betonen, dass allein dieser Teil sehr lange gedauert hat, da ich sehr lange gebraucht habe, um den RaycastHit unabhängig von der direkten Bewegung des Charakters zu halten und die Mausposition entsprechend der Kameraposition zu bestimmen.

### Teil 2: Türme auswählen können

Ein etwas kürzerer Abschnitt bestand darin, es zu ermöglichen verschiedene Arten von Türmen spawnen lassen zu können. Dies gliedert sich in 2 Konstrukte:

* Das Skript, welches ein Array erstellt und bestimmte auf dem UI erstellte Buttons mit einer Liste von Prefabs verbindet
* Eine UI welche funktionierende Buttons hat und die Möglichkeit, diese mit dem Skript zu verknüpfen.

Angefangen habe ich, ein einfaches UI Element zu erstellen, darauf gehe ich aber noch im Thema erste UI Versuche ein. Fakt ist nur, wir haben zwei Buttons, die sobald gedrückt das Spawnen des entsprechenden Turms aktivieren sollen.

void Start()

{

// Initialisierung der Buttons:

for (int i = 0; i < spawnButtons.Length; i++)

{

int buttonIndex = i; // Speichern des Button-Index für den Event-Handler

spawnButtons[i].onClick.AddListener(() => {

selectedPrefabIndex = buttonIndex; // Setze den ausgewählten Index, wenn der Button gedrückt wird

spawnEnabled = true; // Aktiviere das Spawning

//Tims Änderung

UpdateCost();

});

}

}

Diese Start Methode läuft einmal ab, sobald das Skript aufgerufen wird. Voraussetzung für diese Methode sind, ein Array von Prefabs, ein Array von Buttons, ein Integer der das ausgewählte Element und dessen Position im Array angibt. In der for-Schleife wird zunächst ein temporärer int erstellt, der das Lesen der Position ermöglicht. Danach wird bei durchlauf der Schleife geprüft, ob ein Knopf gedrückt wird. Jeder Durchlauf der Schleife entspricht einem einzelnen Knopf. Sobald ein Knopf gedrückt wird, wird das Äquivalent als Element im Array der Prefabs als Zahl im int Wert selectedPrefabIndex gespeichert, damit dieser später beim Instanziieren abgerufen werden kann.

Im Verlauf dieses Abschnitts war mir ChatGPT tatsächlich eine sehr große Hilfe, da eines meiner Hauptprobleme der korrekte Syntax in C# ist. Hierbei konnte es mir natürlich sehr gut helfen.

### Teil 3: Nicht nur Platzieren sondern auch Bewegen (Uiii spannend)

An dieser Stelle kann der Spieler also nach Drücken des Buttons, wo der Name des entsprechenden Turms steht, mit einem Mausklick den Turm an einer Stelle fest platzieren. Da dachte ich mir das ist doch irgendwie blöd. Bspw wenn man sich mal verklickt oder einfach unzufrieden ist, hat man keine Möglichkeit mehr, die Position des Turmes zu ändern? Nein nicht mit mir!!

Um ein späteres Verschieben des Turms zu ermöglichen musste ein zweites Skript her, welches wieder auf dem guten SpawnHandler liegt.

Jetzt greife ich die Identifizierung auf, die ich schon in Teil 1 angebracht habe. Beim Instanziieren des Turms werden im zwei Dinge „gegeben“: ersten ein Tag (LastSpawned“) und ein Name. Für das Preview Skript (sehr kreativer Name) ist nur der Tag wichtig. So wird am Beginn der Update Methode folgendes ausgeführt:

previewObject = GameObject.FindWithTag("lastSpawned");

Das previewObject ist das GameObject, welches eben vom anderen Skript gespawned wurde. Über den Tag kann auch dieses Skript des selbe GameObject problemlos finden und mit ihm interagieren.

if (spawnScript.spawned && previewObject != null && Input.GetMouseButton(0))

{

Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);

RaycastHit hit;

// Wenn der Raycast etwas trifft

if (Physics.Raycast(ray, out hit))

{

// Setze die Position des Vorschau-Objekts auf den Trefferpunkt

previewObject.transform.position = new Vector3(hit.point.x, 3, hit.point.z);

}

}

Jetzt wird wieder ähnlich wie beim ersten Teil ein Raycast erstellt, der von der Position der Kamera auf den Boden geht. Für ihn gelten dieselben Bedingungen wie für den anderen auch. Der große Unterschied ist nun nur, dass das GameObject nicht instanziiert wird, sondern nur dessen Position verändert wird. Dies geschieht in der Hit if Abfrage mit dem eben genannten Code. Auch wichtig dazu zu erwähnen ist, dass dieser Code auch jeden Frame solange durchläuft bis folgendes passiert: (was folgt, ist der zweite Teil der Update Methode)

if (Input.GetMouseButtonUp(0))

{

spawnScript.spawned = false; // Setze spawned auf false, wenn die linke Maustaste losgelassen wird

if (previewObject != null)

{

previewObject.tag = "Untagged";

}

}

Sobald also die linke Maustaste losgelassen wird, wird die Bedingung spawned aus dem anderen Spawnscript auf false gesetzt und der Tag vom GameObject entfernt, damit das Skript nicht mehr darauf zugreifen kann und dessen Position verändern kann.

### Teil 4: Reichweitenvisualisierung

Für diesen Teil ist mein Ziel eigentlich recht simpel. Nachdem der Spieler auf den Button zum spawnen gedrückt hat und der Turm an einer Stelle instanziiert wird, soll ein Ring oder eine kreisförmige Fläche um den Turm erscheinen, welche die Reichweite des Turms visualisiert. So soll der Spieler direkt und effektiv einschätzen können, wo er seine Türme platzieren muss.

Generell musste ich bevor ich irgendwelchen Code schreiben konnte, ein Teil namens „Range“ in jedem Turmprefab erstellen. Bei mir ist das jetzt einfach nur ein blauer, flacher Zylinder, der auf dem Boden aufliegt. Dieses Teil ist standartmäßig deaktiviert, im Spiel also nicht zu sehen.

void selectPart()

{

previewObject = GameObject.FindWithTag("lastSpawned");

// Find "Range" within the prefab

partToActivate = previewObject.transform.Find("Range").gameObject;

// If "Range" is not found, issue a warning

if (partToActivate == null)

{

Debug.LogWarning("Range Objekt im Prefab nicht gefunden!");

}

}

Mit dieser Methode wird das Objekt „Range“, von ich eben sprach, als ein GameObject im Skript gespeichert bzw. definiert. In der ersten Zeile wird also nach einem GameObject im Spiel gesucht, welches den Tag „lastSpawned“ trägt und dieses wird als previewObject hinterlegt. Der Tag wird von Skript in Teil 1 vergeben. Danach wird im Prefab das Range Object definiert, was dazu führt, dass es in der Update Methode des Skripts aufgerufen werden kann. Der Rest ist nur noch für Troubleshooting.

void Update()

{

// Check if spawnScript is assigned and spawned is true

if (spawnScript != null && spawnScript.spawned)

{

selectPart();

if (partToActivate != null)

{

partToActivate.SetActive(true); // Activate the "Range" part

}

else

{

Debug.LogWarning("Kein Teilobjekt zugewiesen!");

}

}

Hier wird zunächst überprüft, ob ein spawnskript hinterlegt ist(was es immer ist, nur für Debug) und ob der Wert spawned in diesem Skript true ist. Der Wert spawned sagt aus, ob der Turm instanziiert wurde. Passiert dies, wird durch die eben erklärte Methode das Teil Range gefunden und danach aktiviert. So erscheint das Teil im Spiel und es wird erkenntlich, wie weit der Turm schießen kann.

// Check if spawnScript is assigned and spawned is false

else if (spawnScript != null && !spawnScript.spawned)

{

if (partToActivate != null)

{

partToActivate.SetActive(false); // Deactivate the "Range" part

}

}

}

Sobald daraufhin der Platzierungsvorgang durch Beenden des Herumschiebens beendet wird, wird der Wert spawned wieder auf false gesetzt und das eben definierte Teil Range deaktiviert. Es ist im Spiel nicht mehr sichtbar.

Dieser ganze Teil bringt also keine fundamentalen Funktionen ins Spiel, sondern trägt hauptsächlich zur besseren Orientierung im Spiel und zum Spielspaß bei. Wichtig zu erwähnen ist auch noch, dass am Start des Skripts das Teil auf null gesetzt wird. Im Verlauf des Spiels wird der Wert des Teils einfach nur nacheinander überschrieben.

## Drittes Ziel (Tim)

Um die Gegner töten zu können, soll man Türme platzieren können, die die Gegner beschießen. Mein Teil bestand darin, die Funktion der Türme zu programmieren. Ich brauchte also einen Turm, der sich in die Richtung des nächstgelegenen Gegners dreht und diesen abschießt, bis er entweder tot ist oder ein anderer Gegner näher an dem Turm dran ist.

### Vorgehensweise

Zunächst habe ich mich auf das Finden, des nächstgelegenen Gegners in einem bestimmten Radius konzentriert. Man soll also eine range angeben können, in der dann das Target gesucht wird. ChatGPT hat mir dann relativ schnell eine UpdateTarget Funktion gegeben, die eiwandfrei funktioniert hat, weshalb ich diese ein zu eins übernommen habe:

void UpdateTarget()

{

// Sucht nach allen Gegnern mit dem Tag "Enemy"

GameObject[] enemies = GameObject.FindGameObjectsWithTag(enemyTag);

float shortestDistance = Mathf.Infinity;

GameObject nearestEnemy = null;

// Finde den nächsten Gegner innerhalb der Reichweite

foreach (GameObject enemy in enemies)

{

float distanceToEnemy = Vector3.Distance(transform.position, enemy.transform.position);

if (distanceToEnemy < shortestDistance)

{

shortestDistance = distanceToEnemy;

nearestEnemy = enemy;

}

}

// Wenn ein Gegner gefunden wurde, setze ihn als Ziel

if (nearestEnemy != null && shortestDistance <= range)

{

target = nearestEnemy.transform;

}

else

{

target = null; // Kein Gegner in Reichweite

}

}

Diese Funktion prüft zwar erst nachdem der nächstgelegene Gegner gefunden wurde, ob er in der Reichweite liegt, erfüllt aber trotzdem die Voraussetzungen. Sie wird in der Start Funktion immer wieder ausgeführt. Außerdem gab mir ChatGPT eine relativ einfache Funktion, die die Reichweite des Turms im Editor anzeigt, auch, wenn das Spiel momentan nicht läuft.:

void OnDrawGizmosSelected()

{

Gizmos.color = Color.red;

Gizmos.DrawWireSphere(transform.position, range);

}

Der Rest des Tower Skripts, den mir ChatGPT gegeben hat, hingegen war weniger überzeugend. Ich musste viele Variablen hinzufügen, die Update Funktion überarbeiten, in welcher der Turm konstant in die Richtung des Targets gucken soll und in einem bestimmten Abstand Projektile instanziieren soll, und die Art, wie die Projektile geschossen werden. Ich entschied mich dagegen, so wie ChatGPT es haben wollte, die Funktion der Projektile in das Tower Skript zu legen und erstellte deshalb ein Projectile Skript für die Projektile, das kontrollieren sollte, wie sich die Projektile bewegen.

Auf dieses Skript bin ich besonders stolz, da ich, bis auf ein paar Syntax – Fehler, das komplette Skript ohne Hilfe geschrieben habe, so dass es anfangs ziemlich gut funktionierte. Ich habe keine weitere Möglichkeit gefunden, mit der ich von dem Projectile Skript aus auf das richtige Tower Skript zugreifen kann, sodass das Projektil die richtige Reichweite und die richtige Anzahl an Schaden usw. hat, als wieder den nächstgelegenen Turm zu suchen, also kopierte ich die Logik der UpdateTarget Funktion und baute sie so um, dass es am Anfang den nächstgelegenen Turm findet. Von diesem Turm sollte das Skript nun die nötigen Variablen nehmen (Entfernung, Speed, Schaden usw.), die dann in der Update Funktion dazu dienen, das Projektil zum Target zu bewegen. Um dem Gegner Schaden zuzufügen, ließ ich das Skript auf das DamageTest Skript von dem Target zugreifen und die TakeDamage Funktion ausführen, sobald das Projektil am Ziel ist.

### Was macht der Code?

Das Tower Skript ist dafür verantwortlich den Turm in die richtige Richtung zu drehen, so wie bereits beschrieben und diverse Variablen abzufragen, die im Inspector definiert werden: range (die Reichweite), projectilePrefab (die Art der Projektils), fireRate (die Schussfrequenz), bulletSpeed (die Geschwindigkeit des Projektils), damageAmount (den Schaden).

Das Projectile Skript ist hauptsächlich für die Bewegung des Projektils verantwortlich, wie bereits beschrieben. Dafür sucht es anfangs den nächstgelegen Tower und nimmt sich von ihm die Variablen range, bulletSpeed und damageAmount. Dann sucht es sich den nächstgelegenen Gegner und „verfolgt“ ihn bis es ankommt und ihm Schaden zufügt oder bis das Ziel selbst zerstört wird.

## Viertes Ziel (Tim)

Während unserer Testphase, haben wir einige Bugs festgestellt. Für zwei davon war ich verantwortlich, einmal, dass das BotsOnPath Skript auf die Bots (Gegner Objekte) zugreifen will, auch wenn sie bereits zerstört sind und dass die Projektile in der Luft hängen bleiben, wenn das ursprüngliche Ziel zerstört ist. Es war zwar witzig, den Bug auszutesten, da man mit dem Spieler auf den Kugeln laufen und springen konnte, es war aber nicht Sinn des Spiels. Außerdem stellten wir fest, dass es sinnvoll wäre, eine Markierung für den Weg der Gegner im Spiel zu haben, da man ihn bisher nur im Editor und nicht im Spielmodus sieht.

### Vorgehensweise

Als erstes wollte ich die Markierung einfügen, die ich komplett ohne Hilfe eingerichtet habe. Ich habe ein Prefab erstellt, und diesen mithilfe der Art, wie die Bots den Path langlaufen (Sie errechnen an einer Weite a, die Position auf dem Pfad, teleportieren sich dorthin und ändern a um einen Wert e) den Pfad entlang initialisieren lassen.

Die Bugs zu fixen war etwas aufwendiger bzw. schwieriger für mich. Ich stellte jedoch nach ein paar Stunden fest, dass ich beide mit einer Lösung fixen kann, und zwar mithilfe eines IsAlive public boolean, der für einen Gegner false wird, sobald dieser stirbt. Diesen habe ich dann in das BotsOnPath Skript eingefügt, dass es nur solange versuchen soll, den Bots zu bewegen, bis er Tod ist, und in das Projectile Skript eingefügt, sodass es das Projektil zerstören soll, wenn das Ziel Tod ist.

## Drittes Ziel (Jaron)

### Teil 1: Main Menü

Zwischendurch wollte ich mich von der Frustration abschlagen, dass alle möglichen Versuche, das Spawnsystem zu laufen zu kriegen, gescheitert sind, ablenken. Letztendlich habe ich begonnen, ein einfaches erstes UI für unser Spiel zu erstellen. Zum jetzigen Zeitpunkt hat noch alles die Standardtexturen von Unity, aber alles, was ich gemacht hat funktioniert. Zu Beginn habe ich mich darauf fokussiert, ein simples Hauptmenü für unser Spiel zu erstellen. So habe ich auf Youtube ein schönes Video gefunden, wie man ein UI in Unity erstellt. Als Hauptmenü habe erstmal einfach einen farbigen Hintergrund erstellt. Darauf kamen dann drei Buttons. Der erste, und oberste, Button der drei heißt ganz einfach „PLAY“ und öffnet nach dem Drücken das erste Level des Spiels. Wenn in Zukunft eine Möglichkeit zu speichern erstellt wird, wird dieser Knopf den aktuellen Speicherstand laden. Zum jetzigen Zeitpunkt ist dies aber noch nicht der Fall. Um den Wechsel in Level eins zu ermöglichen, wurde ein Skript erstellt, in der eine Methode namens „PlayGame“ definiert ist.

public void PlayGame()

{

SceneManager.LoadScene(SceneManager.GetActiveScene().buildIndex + 1);

}

Hierbei wird ganz einfach die nächste Szene im SceneManager geladen. Die Reihenfolge muss dann extern im Buildinspector eingestellt werden.

Der nächste Button öffnet ein einfaches Optionsfenster. Darauf sollen in Zukunft verschiedene Einstellungen wie Lautstärke und mehr einstellbar sein. Bis jetzt wird beim Klick einfach ein GameObject namens „OptionsPanel“ aktiviert. Auf diesem neuen Fenster befindet sich dann wieder ein Cancelbutton, mit dem das Fenster geschlossen werden kann. Dabei wird einfach nur das Fenster als GameObject deaktiviert.

Der unterste Button trägt den Titel „Quit“ und viel mehr macht er auch nicht – er beendet das Spiel. Hierzu wurde wieder im selben Skript wie beim Playbutton eine Methode erstellt.

public void QuitGame()

{

Debug.Log("Game is quitting...");

Application.Quit();

}

Mit dem ersten Befehl wird in der Konsole von Unity eine Meldung ausgegeben, dass das Spiel geschlossen wird. Mit dem zweiten Befehl wird das Spiel dann auch geschlossen. Dabei handelt es sich um eine Unity interne Funktion.

### Teil 2: Spawnpanel

# Reflektion Q1

## Tim

Im Großen und Ganzen bin ich sehr zufrieden mit meiner Arbeit an dem Projekt und mit dem, was ich bis jetzt erreicht habe. Ich kannte vor vier Monaten keinen einzigen Befehl von C#, ich kannte Unity nur als das, was beim Starten mancher Spiele auf dem Ladebildschirm sieht. Jetzt kann ich bereits vollfunktionsfähige Skripte schreiben, die ein bisschen mehr sind als „Bewege dich 3 Einheiten nach vorn“. Ich benötige dennoch bei Vielem Hilfe und da ChatGPT relativ schnell sehr passende Antworten liefert, habe ich das Projekt bis jetzt hauptsächlich mit Hilfe von ChatGPT bearbeitet. Man könnte zwar meinen, dass ChatGPT „cheaten“ wäre und ich muss zugeben, dass es das Lernen der Programmiersprache auf alle Fälle sehr viel leichter gestaltet. Wenn es zum Beispiel um das Verständnis spezieller Code Teile geht, kann man ChatGPT präzise Fragen stellen, ohne dass man sich Stunden lang durch irgendwelche Webseiten schlagen muss. Ich bin mir sicher, dass man ohne ChatGPT auch gut lernen kann, mir gefällt aber die Arbeit mit ChatGPT besser, weil es für mich leichter ist. Cheaten ist ChatGPT meiner Meinung nach aber nicht, da er nicht immer genau das macht, was man will, man kann also selten komplette Codes von ChatGPT schreiben lassen.

Zudem bin ich sehr zufrieden damit, dass ich durchgängig mehr oder weniger motiviert an dem Projekt gearbeitet habe. Ich war vor allem am Anfang sehr motiviert mich mit dem Thema auseinanderzusetzten, da es etwas völlig Neues für mich war. Es ist viel komplexer als zum Beispiel der Java Hamster, den ich aus dem Unterricht kenne, weshalb man viel mehr Sachen machen kann.

Trotz meiner durchgängigen Motivation und meiner Arbeit mit ChatGPT gab es Probleme bzw. bugs, Dinge die nicht so sein sollten, wie sie waren, die mir zeitweise auch Motivation geraubt haben. Während des Programmierens gab es Stolpersteine, wie die richtige Verwendung des PathCreators am Anfang zu der ich mir mehrere Videos angucken musste und ChatGPT viele Fragen stelle musste. Ich bin relativ froh darüber, dass es für mich nicht allzu viele Probleme gab. Das Spawnen und Bewegen der Gegner war relativ schnell erledigt, das Bauen der Lebensanzeige war etwas aufwendig, weil ich vieles im Inspector einstellen musste und dadurch nicht auf die Hilfe von ChatGPT zählen konnte, das weitere Schadensystem, und die Funktion der Türme, verliefen jedoch ziemlich reibungslos. Es gab hier und da immer wieder neue Befehle, die ich unter die Lupe nehmen musste.

Die größten Problem kamen als Bugs am Ende auf, als wir alle Funktionen in eine Szene gepackt haben. Jaron ist aufgefallen, dass die Projektile in der Luft hängen bleiben, sobald das Ziel stirbt, und dass das BotsOnPath Skript versucht auf die Gegner zuzugreifen, die bereits zerstört sind. An diesen Bugs habe ich mehrere Stunden gesessen, bis mir endlich die Idee gekommen ist, einen public Boolean auf dem DamagTest Skript zu erstellen, der auf false gesetzt wird, sobald das Objekt stirbt. Dieser Boolean löste sowohl das Problem des BotsOnPath Skript, als auch, nach ein paar Anpassungen, das Problem der Projektile.

Zusammenfassend bin ich sehr stolz auf das, was ich erreicht habe. Ich habe es geschafft bei keinem Problem zu verzweifeln und zu programmieren, während ich die Sprache lerne.

# Probleme

Stichpunktartig erstmal alle Probleme bzw. Dinge, die ziemlich kompliziert waren und sich nicht so einfach lösen ließen (z.B Bugs)

## Tim:

* Verwendung des PathCreators
  + Es war zuerst nur möglich, den PathCreator bei laufendem Spiel zu verwenden
  + Ich musste mich auch zuerst an das Programm gewöhnen und wusste nicht, was es bedeutet, wenn man zum Beispiel auf ein Objekt klickt.
* Formatieren der Gegner und der dazugehörigen Healthbar
  + Ich hatte das Konzept von Parent und Children in Unity noch nicht verstanden und habe als erstes versucht, die Healthbar am Anfang des Spieles sich dem Gegner anzufügen, sobald der Gegner gespawned wird.
* Projectile Bug
  + Die Projektil sind in der Luft hängen geblieben, wenn das Target gestorben ist, bevor das Projektil angekommen war.
* BotsOnPath Bug
  + Das BotsOnPath Skript hat weiterhin versucht, auf die Gegner zuzugreifen, nachdem sie gestorben sind und es entstanden dadurch für jeden Frame und jeden getöteten Gegner eine MissingReference - Exception.
* Base Damage Bug
  + Der Base wurde doppelt zu viel Schaden zugefügt, weil sie zwei Collider besaß
* Generelle Nutzung von ChatGPT
  + Zum Einstieg war ChatGPT für einfache Skript sehr Hilfreich und konnte mir konkrete Fragen zu bestimmten Funktionen in Unity gut erklären. Je fortgeschrittener jedoch die Skripte waren, die ich schreiben wollte, desto mehr Fehler machte ChatGPT (nicht im Syntax, aber in der Logik) und ich musste mehr und mehr Fragen stellen, um das Skript, was ChatGPT mir gab zu verstehen und dementsprechend zu verbessern. Dennoch nehme ich ChatGPT bis jetzt als beste Möglichkeit, sich mit Unity auseinander zu setzen, wahr.

## Jaron:

# Vorarbeit Q2 (Nicht Q1)

## Fünftes Ziel (Tim)

Mein nächstes Ziel bestand darin, das Gruppierte Spawning zu programmieren. Dazu gehört:

Dafür war es nötig, mein komplettes BotsOnPath Skript zu überarbeiten (Siehe ChatGPT). Sehr schnell mit ChatGPT

Bug mit Projektilen, die aufgrund der vielen Gegner hinunter wackeln.

Weg mit dem Vorwärtslaufen der Bots bei „W“

Neue Enemies + Componente Float speed in DamageTest und dementsprechend BotsonPath angepasst

Türme schießen nur, wenn „spawned“ nicht aktiv ist. 🡪 Von Jarons Skript

Credit einstellen: Auf der UI- Canvas neuen Text „Credits“ mit dem UpdateCredit – Script von ChatGPT (Siehe unity3)

GameManager Instance ausprobiert, hat nicht wirklich funktioniert, also erstmal so, wie gewohnt mit public in Inspector. In Gamemanager – Skript jetzt addCoins und spendCoins

Geschafft: Das TurretSpawn Skript nimmt den Credit und den Richtigen Price wert und verwendet diese. Gekauft wird mit der SpendCredit – Funktion auf dem GameManager Skript.

Erstes level: Ich habe das Erste Level Gestaltet, nachdem ich erfolgreich den Preis und die Belohnung eingerichtet habe. + Einen Neuen Bot (Boos). Fehler: Die Bots geben reward wenn von der Base zerstört