Time Raider

Dokumentation einer Spieleentwicklung

Softwarepraktikum SS18 – WS18/19

Dozent: Prof. Dr. Lütticke

Sulfikar Hamka, Armin Maghsoudloo, Hasan Turan, Vincent Brücher

Inhaltsverzeichnis

1. Einleitung

In der folgenden Dokumentation wollen wir zuerst unser Softwareprojekt „Time Raider“ vorstellen. Danach werden wir die verwendeten Programme aufzählen und anschließend genauer auf einzelne Aspekte der Spieleentwicklung wie beispielsweise das Schreiben von Skripten eingehen.

Zu Beginn des Sommersemesters 2018 entschieden wir uns unter der Leitung von Prof. Dr. Lütticke als Softwareprojekt ein eigenes Spiel zu entwickeln. Wir, das sind Sulfikar Hamka, Armin Maghsoudloo, Hasan Turan und Vincent Brücher sind alle Studenten an der Hochschule Bochum. Wir hatten keinerlei Vorkenntnisse und hatten noch nie mit Unity gearbeitet.

Unser Spiel „Time Raider“ ist ein 2D-Plattformer, in dem man durch verschiedene Zeiten reist um sich selbst und seine Familie zu retten. Dabei muss man an verschiedenen Gegner vorbeikommen, Rätsel lösen und gefährlichen Fallen ausweichen.

2. Spielablauf und -mechanik

2.1 Spielidee

Die Hauptfigur ist aufgrund einer ungewollt Zeitreise in die Vergangenheit gereist und muss wieder in seine Gegenwart zurückreisen. Die Zeitmaschine scheint jedoch nicht mehr zu funktionieren. Ab hier übernimmt der Spieler die Kontrolle der Hauptfigur und bewegt sich durch verschiedene Orte aus unterschiedlichen Zeiten. Jeder der vier Akte spielt sich jeweils in einem Zeitalter ab. In jedem Akt beziehungsweise Zeitalter, gibt es jeweils mehrere verschiedene Orte beispielsweise gibt es in der ersten Zeit ein freundlich und ein feindlich gesinntes Dorf, aber auch eine Arena mit einem Boss Gegner, der besonders stark ist, dazu gibt es noch mehrere Level, die diese Orte sinnvoll miteinander verbinden.

Damit die Hauptfigur wieder in die Gegenwart gelangt, muss der Spieler verschiedene Aufgaben bewältigen, um die Zeitmaschine erneut einsetzen zu können. Gegner und Hindernisse aus den jeweiligen Zeiten versuchen den Spieler davon abzuhalten, die Aufgaben zu erledigen. Der Spieler muss Gegner und Hindernisse bewältigen, um Spielfortschritte zu erlangen.

2.2 Spielmechanik

Die Hauptfigur bewegt sich durch verschiedene 2D-Welten. Dieser kann nach links und rechts laufen, springen und kriechen. Der Spieler startet mit 100 Lebenspunkten. Wird der der Spieler von Hindernissen oder einem Gegner getroffen, verliert er Lebenspunkte. Sobald der Spieler keine Lebenspunkte mehr besitzt, hat er das Level verloren und muss von Beginn des Levels anfangen. In jedem Level befinden sich Herzen, die der Spieler aufheben kann, um neue Lebenspunkte zu erhalten.

Um Hindernisse bewältigen zu können, kann die Hauptfigur auch springen. Einige Hindernisse sind zum Beispiel Schluchten. Fällt man in diese hinein, kann man nicht mehr hinauskommen. Der Spieler verliert in diesem Fall Lebenspunkte und wird zum letzten sicheren Checkpoint befördert, den er zuvor erreicht hatte.

Gegner können mit Fern- und Nahkampfwaffen angreifen und vom Spieler angegriffen werden. Mit dem Kaufsystem kann der Spieler sich mit Gegenständen ausrüsten, um sich stärker werdenden Gegnern zu stellen (siehe 2.3.2 Kaufsystem).

2.3 Spielbalancing

Im Laufe des Spiels werden die Hindernisse und Rätsel immer schwieriger zu absolvieren. Der Spieler kann sich zur Bewältigung der Hindernisse keine Vorteile erschaffen. Hier zeigt sich das Können des Spielers. Die Gegner hingegen werden stärker, indem ihre Lebenspunkte erhöht werden, sie mehr Schaden verursachen oder neue Angriffsmuster benutzen. Damit der Spieler die Möglichkeit hat, die Gegner zu besiegen, sollte er sich für den jeweiligen Akt ausrüsten. Dabei muss er sich das Kaufsystem zu Nutzen machen und passende Gegenstände kaufen, die ihm einen Vorteil verschaffen. Eine wichtige Rolle spielen zeitgemäße Fern- und Nahkampfwaffen, damit die Hauptfigur bei stärkeren Gegnern, mehr Schaden anrichten kann.

2.3.1 Gold

„Gold“ ist die Ingame Währung von Time Raider. Mit dieser Währung lässt sich bessere Ausrüstung in verschiedenen Shops in den Städten kaufen. Gold wird von den Gegnern fallengelassen oder liegt zufällig verteilt im Level herum. Dieses sammelt man automatisch auf, indem man über das Goldstück läuft. Die Menge an besessenem Gold wird oben rechts im Spiel angezeigt.

2.3.2 Kaufsystem

Das Kaufsystem ist ein wichtiger Bestandteil des Spiels. Wenn man diesen nicht nutzt, ist es nur schwer möglich, die nächsten Level zu meistern. Der Spieler sollte also in jedem Akt mindestens einmal den Shop aufsuchen und sich für die nächsten Level rüsten. Jeder Akt bietet einen Shop an, der zeitgemäße Nah- und Fernkampfwaffen anbietet. Diese können durch Gold gekauft werden. Der Shop befindet sich meistens in einer sicheren Zone, also in Städte und Dörfer und kann mithilfe des „Shop“-Button geöffnet werden.

2.3.3 Waffen

Wie bereits erwähnt gibt es zahlreiche Waffen. Diese unterteilen sich in Nah- und Fernkampfwaffen, welche der Spieler als auch die Gegner nutzen. Den Großteil der Waffen muss sich der Spieler im Shop kaufen, manche werden ihm aber auch geschenkt. Um den Spielfluss nicht zu stören haben wir uns dagegen entschieden Munition zu begrenzen. Der Spieler kann also beliebig viele Pfeile verschießen.

3. Installationsanleitung

3.1 Voraussetzungen

Für Time Raider wird ein Computer mit Windows Betriebssystem vorausgesetzt, eine Portierung auf Mac und Linux ist jedoch mit ein wenig mehr Aufwand auch möglich.

Die anderen Anforderungen an den Computer sind nicht hoch. Unter den verschiedenen getesteten Systemen kam es zu keinerlei Problemen, selbst ein älterer Laptop konnte das Spiel problemlos abspielen. Um jedoch genaue Systemvoraussetzungen angeben zu können fehlen uns die nötigen Testsysteme. Auf jedem modernen Computer sollte das Spiel jedoch problemlos laufen.

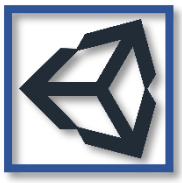
3.2 Installation

Die Installation erfolgt nicht durch einen Wizard, wie es normalerweise üblich ist, sondern durch das Aufspielen der Spieldateien von einem USB-Stick, einer CD oder einem Datenträger auf den jeweiligen Computer. Sobald sich alle Spieldateien auf dem Computer befinden, kann Time Raider gestartet und sofort gespielt werden. Beim Starten des Spiels lassen sich Auflösung, verwendeter Monitor, Grafikqualität und die Steuerung individuell einstellen und an den eigenen Computer anpassen.

4. Verwendete Software

In den folgenden Abschnitten stellen wir euch die von uns verwendeten Programme vor. Um nicht den Rahmen zu sprengen werden nur die wichtigsten genauer erklärt. Zur Kommunikation haben wir WhatsApp, TeamSpeak und Steam genutzt. Um diese Dokumentation und andere Texte zu schreiben nutzen wir Microsoft Word. Die für die Dokumentation zugeschnitten Bilder entstanden in Photoshop. Nun jedoch zu den vier wichtigsten Programmen.

4.1 Unity



Bei der Wahl der Laufzeit- und Entwicklungsumgebung von „Time Raider“ haben wir uns für Unity vor Unreal Engine entschieden. Im Hauptfenster, dem Editor, können Objekte per Drag & Drop erstellt werden und so die verschiedenen Level zusammengebaut werden. Daneben im Fenster „Inspector“ können danach die einzelnen Attribute der Objekte angepasst werden, also wie viel Leben ein Gegner hat, ob ein Objekt Schwerkraft hat und auch selbst erstellte Skripte werden hier hinzugefügt.

Doch das Meiste davon bieten alle Entwicklungsumgebungen, für Unity entschieden wir uns da es sehr einsteigerfreundlich ist, viele Lernmaterialien anbietet und auch Beispiele bereitstellt. Außerdem ist die Community sehr aktiv und stellt Tutorials zu nahezu jedem Thema im Bereich Spielentwicklung zur Verfügung. Leider unterstützt Unity nur C# und JavaScript und wir mussten uns zu Beginn in die Sprachen einarbeiten. Ein Vorteil von Unity hingegen ist, dass es für Projekte die weniger als 100.000€ Umsatz machen kostenlos verfügbar ist.

Leider fehlt Unity ein eingebautes Bildbearbeitungsprogramm und auch die Skripte mussten wir in einem extra Programm erstellen.

4.2 Inkscape

Da wir alle Hintergründe, Charaktere, Gegner und Items selbst zeichnen wollten brauchten wir ein Bildbearbeitungsprogramm. Unsere Wahl fiel dabei auf Inkscape.

Inkscape ist eine Software zur Erstellung und Bearbeitung von Grafiken. Wir entschieden uns gegen das am weitesten verbreitete Bildbearbeitungsprogramm Photoshop und für Inkscape aus drei wichtigen Gründen.

Der erste ist das Inkscape mit sogenannten Vektorgrafiken arbeitet. Diese haben keine feste Pixelgröße und können daher beliebig skaliert werden. Dies wird umgesetzt indem jeder Strich, Punkt, Kreis, etc. als einzelne Objekte gespeichert werden. Dadurch lassen sich auch Fehler leicht im Nachhinein beheben, da man das gewünschte Objekt einfach in der Hierarchie auswählen und bearbeiten kann.

Der zweite Punkt ist, dass Inkscape sehr intuitiv und leicht zu handhaben ist. Die meisten Features sind direkt per Maus ansteuerbar und keine komplizierten Tastenkombinationen müssen auswendig gelernt werden.

Der dritte Grund ist das Inkscape kostenlos verfügbar ist. Da nicht alle Teammitglieder Photoshop besaßen und dies momentan 24€ im Monat kostet war Inkscape eine willkommene Alternative.

4.3 Visual Studio

Um unsere Skripte zu Schreiben entschieden wir uns für Visual Studio. Es unterstützt eine Vielzahl von Programmiersprachen von Basic, über C# bis hin zu Python. Es markiert den Code farbig und schlägt automatisch Verbesserungen vor. Visual Studio wird außerdem von Unity empfohlen, da sich die beiden Programme miteinander verknüpfen lassen. Danach werden von allein einige wichtige Packages eingebunden sobald man ein neues Skript beginnt und man kann direkt mit dem Programmieren beginnen. Auch dabei hilft die Verknüpfung, spricht man ein Objekt an werden direkt die Attribute vorgeschlagen oder andere in dem Objekt verwendete Skript angezeigt. Auch dieses Programm ist kostenlos sofern das Projekt weniger als fünf Entwickler hat oder Open-Source ist.

4.4 GitHub

Zu Beginn unseres Projekts entschieden wir uns für das von Unity gestellte Feature „Unity Teams“ um unseren Fortschritt untereinander auszutauschen. Leider wurde dieses Feature im Laufe unserer Entwicklung umgestaltet und war nur noch für Projekte mit bis zu drei Entwicklern kostenlos. Daher stiegen wir auf den kostenlosen GitHub Desktop Client um.

GitHub ist eine weit verbreitete Versionsverwaltungssoftware. Sie basiert, wie der Name bereits verrät auf Git und ist solange man seine Projekte öffentlich zugänglich macht kostenlos. Mit Ihr können wir von verschiedenen Computern aus arbeiten und trotzdem alle die gleichen Neuerungen erhalten. Hat beispielsweise Entwickler A ein neues Skript erstellt kann Entwickler B dieses einsetzten, ohne dass es über einen USB Stick oder ähnliches übertragen werden muss.

Veränderungen können leicht zurückgenommen werden oder eigene „Branches“, also Kopien des gesamten Projekts zu einem beliebigen Zeitpunkt der Entwicklungsphase erstellt werden. Mit diesen kann man verschiedene Dinge ausprobieren, ohne das gesamte Projekt zu gefährden.

Der Desktop Client bietet dazu noch den Komfort, dass all Dies funktioniert ohne das man Befehle in die Kommandozeile eingeben muss. Mit wenigen Klicks sind wir so in der Lage unsere Neuerungen und Veränderungen untereinander auszutauschen und immer auf dem neusten Stand zu sein.

5. Der Entwicklungsprozess

Im folgenden Kapitel wollen wir euch darstellen wie Time Raider entstanden ist.

5.1 Die Planung

Zu Beginn des Projekts mussten wir mit dem Pflichtenheft die Planung für das gesamte Projekt bewerkstelligen. Das Schwierigste war dabei, dass wir noch komplett unerfahren im Bereich Spieleprogrammierung waren. Wir konnten nur schwer einschätzen welcher Schritt wie lange brauchen würde. Dazu mussten noch viele Entscheidungen mussten getroffen. Zu welchem Genre sollte unser Spiel gehören, welchen Grafikstyle wollten wir benutzen und was sollte die grobe Handlung der Geschichte sein?

Trotz dieser Hürden ist es uns gut gelungen und wird konnten die selbst aufgestellten Meilensteine alle einhalten, wenn man hier und da ein Auge zugedrückt hat. Als gutes Beispiel hierfür dient die Geschichte des Spiels. Diese sollte bereits beim zweiten Meilenstein abgeschlossen sein, änderte sich im Laufe der Entwicklung immer wieder leicht.

Auch das Design der Gegner und deren Animationen hatten wir stark unterschätzt. Diese sollte bereits beim 6. von 16 Meilensteinen fertig sein, wir entdeckten jedoch schnell, dass es viel passender war die Gegner dann zu Designen, wenn das zugehörige Level designt wurde. So wurden die Gegner aus der Zukunft viel später designt als auch das Level der Zukunft erstellt wurde.

5.2 Die Umsetzung

Nach der Planung begannen wir mit der Umsetzung unseres Projekts. Zu Beginn bestand dies hauptsächlich dadrinnen sich in Unity einzuarbeiten und zahllose Tutorials durch zu lesen. Hier zeigte sich schnell wie einsteigerfreundlich Unity ist und wie aktiv die Community ist. Zu nahezu jedem Thema fanden sich Beiträge und Hilfestellungen.

Auch an die anderen Programme musste man sich erst gewöhnen. So hatten unsere frühen Zeichnungen alle einen per Hand gemalten Rand. Später entdeckten wir dann das Inkscape allen Objekten auch selbst Ränder hinzufügen kann.

Nachdem wir mit der Software warm geworden sind begann die eigentliche Arbeit. Langsam arbeiteten wir uns Meilenstein für Meilenstein voran. Wir merkten wie schwer es war selbst so ein kleines Team zu organisieren, so wurde das erste Level geschätzte zehn Mal neu überarbeitet. Aber dabei lernten wir Unity immer besser kennen und so wurde der vierte Akt in einem Bruchteil der Zeit des ersten Aktes fertig gestellt obwohl er die kompliziertesten Mechaniken enthält.

5.3 Das Ziel

Obwohl wir eigentlich im Zeitplan lagen fiel uns Anfang Dezember auf wie viel uns noch fehlte. Viele Kleinigkeiten hatten sich über das Jahr hinweg angesammelt und ein Großteil der Animationen fehlte noch komplett. Im Januar stand jedoch bereits die Installation auf dem Laptop von Prof. Dr. Lütticke bevor und damit die Abgabe des Projekts. Bis dahin musste alles fertig sein. So wurde unsere Freizeit ein wenig eingeschränkt und wir konzentrierten uns auf die Fertigstellung des Projekts.

Jedoch reichte auch das nicht um rechtzeitig fertig zu werden und wir mussten einige wenige Visionen streichen, um unser Ziel dennoch pünktlich zu erreichen. Als Beispiel stehen die Zuschauer im Kolosseum nur regungslos rum, anstatt unseren Helden bei jedem seiner Siege zu bejubeln. Auch das geheime Level, welches man erst bei einem zweiten Spieldurchlauf freischaltet, in dem man im dritten Reich landet musste gestrichen werden. Durch die Streichung dieser Features blieb uns jedoch mehr Zeit das bisherige Spiel zu polieren und hoffentlich alle Fehler auszumerzen.

5.4 Die größten Hindernisse

Da auch andere Sachen nicht immer reibungslos abliefen wollen wir hier noch unsere größten Schwierigkeiten präsentieren.

Eine davon war das Erzählen der Geschichte, da keiner von uns sich gut mit Animationen auskannte und wir auch nicht den Hauptcharakter vertonen wollten fielen diese beiden Möglichkeiten weg. Ein einfacher Text zu Beginn des Spiels schien uns jedoch auch zu langweilig. Daher entschieden wir uns für eine Art Diashow mit jeweils einem kurzen Begleittext. Dadurch konnten wir die Geschichte allerdings nicht in ihrem vollen Ausmaß erzählen und mussten sie auf das nötigste zusammenfassen. Auch im Spiel selbst ist es nur schwer möglich die Welt lebendig wirken zu lassen, ohne den anderen Parteien im Spiel einen tieferen Hintergrund zu verleihen. Die meisten Spieler sind aber zu faul lange erklärende Texte zu lesen wo durch wir es bestmöglich in die kurzen Gespräche verpackten.

Eine weitere Hürde stellte die Handhabung von GitHub und damit die Koordination der Gruppe dar. Arbeiteten zwei Entwickler gleichzeitig an der gleichen Datei kam es immer zu Problemen und mehrfach wurde ein Teil unserer Arbeit dadurch gelöscht. Jedoch lernten wir auch hier mit der Zeit dazu und gegen Ende des Projekts schrieb jeder an welcher Datei er zurzeit arbeitete in einen gemeinsamen Gruppenchat. Aber auch die generelle Koordination stellte ein Hindernis dar, da man oft den Überblick verlor was noch zu erledigen war, hiergegen begannen wir eine große To-do-Liste, die langsam abgearbeitet wurde.

6. Code Erklärungen

Um einen Eindruck zu übermitteln was genau hinter den Kulissen von Time Raider abläuft wollen wir euch auch Teile des Programmcodes vorstellen. Wir haben dabei versucht möglichst verschiedene Teile zu präsentieren, da sich beispielsweise die Angriffsskripte verschiedener Gegner sehr ähneln.

6.1 Parallaxing

```
5   public class Parallaxing : MonoBehaviour {
6
7       public Transform[] backgrounds;
8       private float[] parallaxScales;
9       public float smoothing = 1f;
10
11      private Transform cam;
12      private Vector3 previousCamPos;
13
14      private void Awake()
15      {
16          cam = Camera.main.transform;
17      }
18
19      // Use this for initialization
20      void Start () {
21          previousCamPos = cam.position;
22
23          parallaxScales = new float[backgrounds.Length];
24          for(int i = 0; i < parallaxScales.Length; i++)
25          {
26              parallaxScales[i] = backgrounds[i].position.z *-1;
27          }
28      }
29
30      // Update is called once per frame
31      void LateUpdate () {
32          for(int i = 0; i < backgrounds.Length; i++)
33          {
34              float parallax = (previousCamPos.x - cam.position.x) * parallaxScales[i];
35
36              float backgroundTargetPosX = backgrounds[i].position.x + parallax;
37
38              Vector3 backgroundTargetPos = new Vector3(backgroundTargetPosX,
39                  backgrounds[i].position.y, backgrounds[i].position.z);
40
41              backgrounds[i].position = Vector3.Lerp(backgrounds[i].position,
42                  backgroundTargetPos, smoothing * Time.deltaTime);
43          }
44
45          previousCamPos = cam.position;
46      }
47  }
```

Im Folgenden wird das Skript Parallaxing ausführlich erklärt. Das Skript wird in nahezu allen Szenen eingesetzt und beschäftigt sich mit den Hintergründen der Levels. Diese werden anhand der Bewegung des Spielers verschoben um einen dynamischen Eindruck zu übermitteln.

Zu Beginn werden fünf Variablen festgelegt, welche im weiteren Verlauf noch genauer erläutert werden. Danach folgen drei Funktionen „Awake“, „Start“ und „LateUpdate“. Als Erstes wird die Funktion „Awake“ erörtert.

Beim Betreten einer Szene werden alle benötigten Skripte geladen, auch unser Parallaxing Skript. Dabei werden zunächst die „Awake“ Funktion aufgerufen. Sie wird bei uns dafür benutzt die Variable „cam“ mit der Hauptkamera zu belegen (Zeile 16). Auf diese Weise kann zukünftig leichter auf diese zugegriffen werden.

Nach der „Awake“ Funktion wird die „Start“ Funktion aufgerufen und initialisiert das Parallaxing. Am Anfang wird die aktuelle Kameraposition in „previousCamPos“ abgespeichert (Zeile 21). Dadurch kann man später berechnen in welche Richtung sich die Kamera bewegt hat.

Danach werden die „parallaxScales“ berechnet, sie werden benutzt, um hinterher ausrechnen zu können um wie viel der einzelne Hintergrund verschoben werden soll. Zunächst wird ein float-Array erstellt, welches die gleiche Länge hat wie „backgrounds“ (Zeile 23). Es kann also genau so viele Werte speichern wie es Hintergründe in der jeweiligen Szene gibt. Zum Berechnen der einzelnen Werte wird das Array nun in einer Schleife durchlaufen. Der Wert wird nun auf die Z-Position des Hintergrundes mal minus Eins gesetzt (Zeile 26). Es entscheidet also wie weit der Hintergrund von der Kamera entfernt ist wie schnell er sich bewegt, die weiter entfernt sind bewegen sich langsamer als die Hintergründe die näher an der Kamera dran sind. Dadurch wirkt der Hintergrund lebendiger, da sich weiter entfernte Objekte auch in der realen Welt scheinbar langsamer bewegen als wir selbst.

Sind alle Werte berechnet ist die „Start“ Funktion abgeschlossen. Als letztes widmen wir uns nun der Funktion „LateUpdate“. Diese wird bei jedem Frame des Spiels aufgerufen und führt das eigentliche Parallaxing aus. Um jeden Hintergrund einzeln verschieben zu können werden diese in einer for-Schleife durchlaufen. Innerhalb dieser wird vorerst die Variable „parallax“ berechnet (Zeile 34). Sie ergibt sich aus der alten und neuen Position der Kamera sowie dem zuvor berechneten „parallaxScales“ Wert. Die Variable sagt aus um wie viel der Hintergrund bewegt werden soll.

Mit dieser Variable kann jetzt die „backgroundTargetPosX“, die neue X-Koordinate des Hintergrundes, ermittelt werden (Zeile 36). Dafür wird die alte X-Koordinate mit der „parallax“ Variable addiert.

Aus der neuen X-Koordinate wird nun die neue Position des Hintergrundes, „backgroundTargetPos“ aufgestellt (Zeile 38). Dabei werden Y- und Z-Koordinate aus der alten Position übernommen und mit der neuen X-Koordinate kombiniert.

Daraufhin kann das Parallaxing ausgeführt werden. Anstatt direkt dem Hintergrund die neue Position zuzuweisen wird „Lerp“ benutzt, um ein flüssigeres Bild zu erhalten. Die Methode „Lerp“ berechnet in Abhängigkeit von der Zeit einen Punkt zwischen der alten und der neu bestimmten Position. Wird ein früher Zeitpunkt übergeben gibt Sie einen Punkt näher an der alten Position zurück, bei einem späteren Zeitpunkt wird eine Position näher an der neueren Position zurückgegeben. Das Ergebnis von „Lerp“ wird dann dem Hintergrund als neue Position zugewiesen (Zeile 41).

Danach wird der Vorgang für den nächsten Hintergrund gestartet. Nach dem alle Hintergründe verschoben wurden, wird die aktuelle Kameraposition noch in „previousCamPos“ abgespeichert damit sie im nächsten Durchlauf benutzt werden kann (Zeile 45).

6.2 Drohnen schießen

```
5   public class DrohneSchiesst : MonoBehaviour
6   {
7
8       //spawn finden
9       public Transform bulletspawn;
10      public Rigidbody2D bulletPrefab;
11      public float bulletSpeed = 750;
12      public float schussGeschwindigkeit;
13      public float radius = 20;
14
15      private Transform player;
16      private Rigidbody2D clone;
17      private bool geschossen;
18
19      // Use this for initialization
20      void Start()
21      {
22          player = GameObject.FindGameObjectWithTag("spieler").GetComponent<Transform>();
23      }
24
25      private void Update()
26      {
27          while (geschossen == false)
28          {
29              StartCoroutine(Attack());
30          }
31      }
32
33      IEnumerator Attack()
34      {
35          geschossen = true;
36          yield return new WaitForSeconds(schussGeschwindigkeit);
37          if (Vector3.Distance(player.transform.position, bulletspawn.transform.position) < radius)
38          {
39
40              Vector3 difference = player.position - bulletspawn.position;
41              float rotationZ = Mathf.Atan2(difference.y, difference.x) * Mathf.Rad2Deg;
42              bulletspawn.rotation = Quaternion.Euler(0.0f, 0.0f, rotationZ);
43
44              clone = Instantiate(bulletPrefab, bulletspawn.position, bulletspawn.rotation);
45
46              clone.AddForce(bulletspawn.transform.right * bulletSpeed);
47          }
48          geschossen = false;
49      }
50  }
```

Bei dem folgenden Skript handelt es sich um den Angriff der Drohnen, die hauptsächlich im vierten Akt vorkommen. Allerdings wird ein ähnliches Skript auch für die anderen Gegner mit Fernkampf eingesetzt, welche es in jedem Akt gibt. Das Skript schießt sobald der Spieler in Reichweite der Drohne ist einen Laser auf ihn.

Zunächst werden alle benötigten Variablen definiert. In der einmalig beim Initialisieren der Drohne aufgerufen Methode „Start“ wird anschließend das Objekt des Spielers gesucht. Daraus wird danach die Position des Spielers entnommen und so abgespeichert, dass sie sich automatisch aktualisiert (Zeile 22).

Wie bei dem Parallaxing wird auch die „Update“ Methode hier ein Mal pro Frame aufgerufen. In ihr wird zuerst getestet ob bereits ein Schuss abgefeuert wurde. Trifft dies nicht zu wird die Funktion „Attack“ als Coroutine aufgerufen (Zeile 29). Eine Coroutine wird verwendet da „Attack“ länger braucht als ein Frame lang ist. Würde sie normal gestartet werden müsste das gesamte Spiel warten, bis der Schuss abgefeuert wurde. Die Coroutine pausiert die Methode sobald der Frame fertig ist und startet sie an der Stelle wieder sobald der nächste Frame anfängt. Dies ist besonders wichtig bei Methoden die Wartezeiten eingebaut haben wie unsere „Attack“ Funktion.

In „Attack“ wird zunächst die Variable „geschossen“ auf true gesetzt damit nicht noch ein Schuss abgefeuert werden kann (Zeile 35). Die Variable „geschossen“ war auch die, auf die in der „Update“ Methode getestet wurde. Als Nächstes wird gewartet, dadurch wird die Schussfrequenz der Drohnen reguliert (Zeile 36). Da die Drohne vor jedem Schuss zielen muss haben wir das Warten an den Beginn der Methode gesetzt. Danach wird geprüft ob sich der Spieler in der Reichweite der Drohne befindet. Dazu wird die Distanz zwischen dem Spieler und dem Gewehrlauf berechnet und mit dem Radius, in dem die Drohne Angreifen kann verglichen (Zeile 37).

Ist der Spieler in Reichweite der Drohne brauchen wir den Schusswinkel in dem abgefeuert werden soll. Dafür wird die Differenz zwischen der Spieler- und Drohnenposition im 3D-Raum ermittelt (Zeile 40). Wir benutzen einen 3D-Raum obwohl wir ein 2D-Spiel entwickeln, da beispielsweise der Spieler immerzu im Vordergrund sein soll, ein Hintergrund hingegen soll immer weiter hinten liegen. Aus der Differenz der Positionen können wir nun x und y entnehmen und dann mit der Hilfe des Arkustangens den Schusswinkel berechnen. Dieser wird anschließend noch mit der „Rad2Deg“ Konstanten in Grad umgewandelt (Zeile 41). Der ausgerechnete Winkel wird dann an die Drohne übergeben, dabei verwenden wir „Quaternion.Euler“ um die Winkel in eine Rotation zu verwandeln (Zeile 42).

Nach dem der Schusswinkel fest steht wird als nächstes die Kugel erzeugt. Wir klonen dafür mit der „Instantiate“ Methode die Originalkugel dazu geben wir noch die Position des Laufes sowie den Schusswinkel an (Zeile 44). Wir klonen sie, da alle Drohnen so auf die gleiche Originalkugel zugreifen können. So kann man nicht nur leicht das Aussehen aller Drohnenschüsse auf einmal ändern, sondern eine Drohne kann auch mehrmals hintereinander Feuern, ohne das die Originalkugel jedes Mal abgefeuert wird und deswegen nie ihr Ziel erreicht.

Damit sich nun der Laser in Richtung Spieler bewegt, wird dem „clone“ eine Kraft zugewiesen. Die Stärke der Kraft beziehungsweise die Geschwindigkeit des Lasers ist abhängig von der Variablen „bulletSpeed“ (Zeile 46).

Zu Letzt wird die „geschossen“-Variable wieder auf „false“ gesetzt, damit die Drohne erneut einen Schuss abfeuern kann (Zeile 48).

7. Rückblick

Gegen Ende wollen wir noch einen Rückblick auf das gesamte Projekt werfen. Das Projekt hat uns nicht nur viele Nerven gekostet, sondern auch viel Freude bereitet. Wir haben nicht nur den Umgang mit Unity und Inkscape ausführlich gelernt, sondern auch allgemeinere Dinge die nicht nur in der Spieleprogrammierung hilfreich sind. Zum Beispiel hat sich unsere Teamkoordination stark verbessert, wir übten wie man Lasten- und Pflichtenheft in der Praxis einsetzt und der Umgang mit GitHub verläuft nun fehlerfrei. Nebenbei ist auch noch ein schönes Spiel entstanden, dass hoffentlich vielen Spielern eine Freude bereitet.

8. Danksagungen

Zu Letzt wollen wir noch einen Dank aussprechen. Wir beginnen der Community von Unity welche zahllosen Tutorials und Hilfestellungen im Internet kostenlos zur Verfügung stellt und uns damit den Einstieg in Unity sehr viel leichter gestaltet hat. Ein besonderer Dank gilt Prof. Dr. Lütticke für die Leitung des Softwareprojekts und seine tatkräftige Unterstützung.

Und wir danken auch Ihnen für das Lesen dieser Dokumentation.

Eurer Time Raider Entwicklerteam

Sulfikar Hamka, Armin Maghsoudloo, Hasan Turan und Vincent Brücher