Kantonsschule Im Lee

Maturitätsarbeit

Herbstsemester 2022/23

Entwicklung eines Computerspiels mit Unity

Philipp Gempp
Klasse 4f

Betreut von Elena Fattorini
Kantonsschule Im Lee
Rychenbergstrasse 140

8400 Winterthur, 09.01.2023

Inhaltsverzeichnis

1 Einleitung

Bereits seit längerer Zeit wollte ich ein Computerspiel programmieren, um eine fortgeschrittene Programmiersprache und Entwicklungsumgebung zu erlernen. Während ich nach einem Thema für meine Maturaarbeit suchte, überlegte ich mir zuerst Themen in Chemie, Informatik hatte ich ebenso im Hinterkopf. Ich erinnerte mich an meine vorherigen Ideen und ich machte mir Gedanken, welches konkrete Informatik-Thema gut als Maturaarbeit wäre. Schliesslich entschied ich mich für die Entwicklung eines Computerspiels, da die Komplexität je nach Ansatz sehr variabel ist. Zudem faszinieren mich Computerspiele nicht nur als Spieler, sondern auch als Betrachter der verschiedenen, sehr aufwendig gestalteten Spielewelten. Vor meiner Maturaarbeit hatte ich wenig Programmiererfahrung, weshalb es schwer war, den Umfang der Arbeit abzuschätzen. Zusätzlich musste ich mich zwischen den unterschiedlichen Sprachen und Spiele-Engines entscheiden.

Bei der Auswahl der Spiele Engine nahm ich Unity und die Unreal Engine in die engere Auswahl, da ich von diesen Engines auch schon vor der Maturaarbeit viel Positives gehört habe. In den unterschiedlichen Vergleichen wurde auch klar, dass auf dem Niveau, auf welchem ich die Funktionen benötigen werde, keine der beiden einen richtigen Vor- oder Nachteil hat. Deshalb entschied ich mich für Unity, da Unity einerseits verbreiteter ist und damit mehr Unterstützung in Communities erhält, was sich später noch als hilfreich herausstellte. Zudem werden Spiele damit einheitlich in C# entwickelt. Im Nachhinein kann ich sagen, dass es eine gute Entscheidung war, weil ich den Überblick über die Unity Benutzeroberfläche schnell gefunden habe und mich gut darin zurechtfinde.

Die grösste Schwierigkeit war, dass ich bisher nicht mit Unity, C# und Microsoft Visual Studio gearbeitet habe, daher war es etwas knifflig für mich die Maturaarbeit zu planen. Zuerst lernte ich die Grundlagen von C#, um es in Unity besser einsetzen zu können. Anschliessend nutzte ich ein Tutorial für Unity, sodass ich schliesslich mit der Entwicklung des Spieles beginnen konnte.

2 Die Spiele-Engine Unity

Unity wurde von Unity Technologies am 8. Juni 2005 veröffentlicht und ist eine Spiele-Engine, die viele unterschiedliche Spieleplattformen unterstützt, wie zum Beispiel PCs, Spielkonsolen, mobile Geräte und Webbrowser [1]. Die Engine beinhaltet einen Asset Store für Grafiken, Audio und Erweiterungen, die leicht heruntergeladen und installiert werden können. Unity wurde in C++ entwickelt und verwendet Microsofts .NET Framework. Um Anwendungen innerhalb von Unity zu programmieren, wird C# verwendet [2].

Die Spiele-Engine besteht aus mehreren Komponenten: den drei Engines für Grafik, Physik und Audio, sowie Komponenten für Ein- und Ausgabe, Netzwerk und Ressourcenverwaltung. Auf die Physik-Engine werde ich später noch genauer eingehen.

2.1 Frameworks

«Ein Framework ist ein Rahmenwerk für die Softwareentwicklung und Programmierung, dass die Grundstruktur und das Programmiergerüst für die zu erstellende Software vorgibt. Es erleichtert den Programmierern die Arbeit und unterstützt objekt- und komponentenorientierte Entwicklungsansätze» [3].

Im einfachsten Sinn bietet ein Framework dem Entwickler der Anwendung grundlegende Funktionen und einen Rahmen, die bei der Entwicklung der Software hilfreich sind. Je nach Framework sind die Funktionen unterschiedlich und deswegen ist es wichtig, sich das richtige Framework für das Projekt auszusuchen. Das Framework stellt die benötigten Funktionen zur Verfügung damit diese nicht für jedes Projekt von Grund auf an neu programmiert werden müssen [3].

2.2 Die Unity Physik-Engine

Die in Unity verwendete Physik-Engine Box2D umfasst Module, die das Verhalten der zweidimensionalen Objekte implementieren. Die Physik-Engine ist sehr umfangreich, die folgenden Module möchte ich detaillierter beschreiben:

- Objects and Shapes
- Dynamic Tree

Diese Module stellen sehr wichtige Funktionen für mein Spiel zur Verfügung: die Berechnung, ob die Spielfigur andere Objekte berührt, und die Gravitation der Spielfigur bei Sprüngen.

2.2.1 Objects and Shapes

Die Unity Spielewelt wird aus Objekten gebildet, denen Grafiken und weitere Komponenten zugewiesen werden, um das Verhalten zu steuern. Die Graphiken werden mit unterschiedlichen Shapes umrandet, und die Shapes wiederum werden genutzt um Kollisionen ermitteln zu können.

Objekte können leer oder gefüllt sein, ein anderes Objekt kann sich nicht innerhalb des Shapes eines gefüllten Objekts befinden.

Die Formen in Box2D werden in Circle Shapes und Polygon Shapes eingeteilt.

2.2.1.1 Circle Shapes

Circle Shapes haben eine Position und einen Radius, sie sind immer gefüllt [4].

2.2.1.2 Polygone Shapes

Polygone sind geschlossene Flächen und bestehen aus Strecken, die jeweils zwei Eckpunkte verbindet. Es gibt eine Unterteilung in konvexe und konkave Polygone, das ist wichtig bei der späteren Berechnung von Kollisionen. Um Polygone herum hat es eine dünne Haut, die die Polygone voneinander trennt. In Box2D wird die Haut nur bei Polygonen erwähnt, während es in Unity auch möglich ist sie bei Chain Shapes zu verwenden. Durch die Haut entstehen kleine Abstände zwischen den Formen [4].

Die linke Abbildung beschreibt Polygone und der umliegenden Haut. Die rechte Abbildung zeigt die Implementierung in Unity, wie eine Grafik mit einem Polygone und der Haut eingefasst wird.

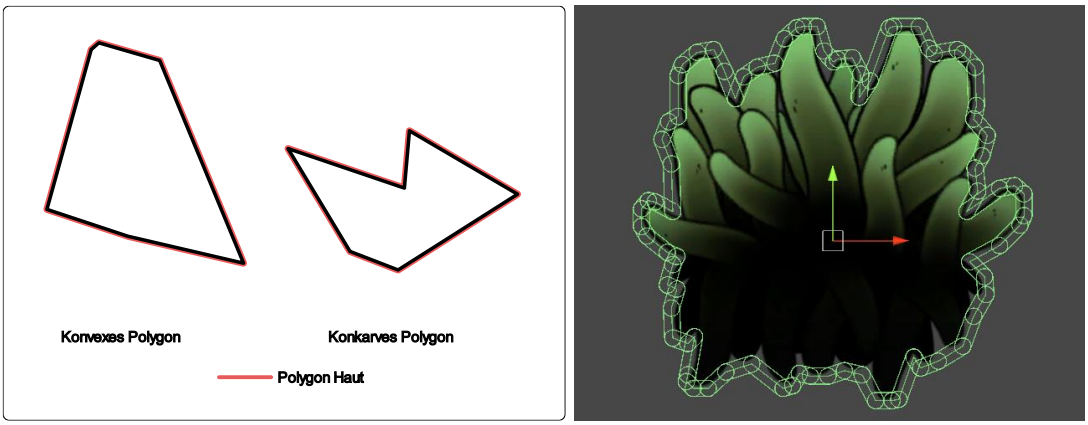

Abbildung 1: Polygone Shapes in Unity

2.2.1.3 Edge Shapes und Chain Shapes

Edge Shapes sind Formen, welche mit anderen Objekten, zusammenstossen können und damit für die Kollisionsberechnung benötigt werden. Edge Shapes werden nur sehr selten einzeln verwendet, sie werden meistes zu Chain Shapes aneinandergekettet [4].

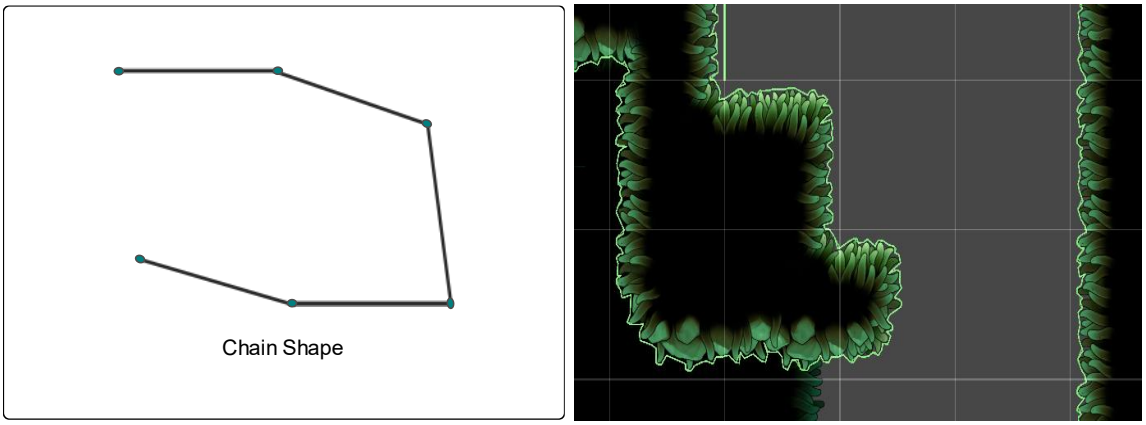

Abbildung 2: Chain Shape in Unity

Chain Shapes werden in Unity verwendet, wenn man für die genauere Bearbeitung des Colliders die Composite Collider Komponente verwendet [5].

2.2.1.4 Kollisionen zwischen Shapes

Es gibt verschiedene Arten der Berechnung von Kollisionen:

- die einfache Berechnung, bzw «Discrete Collission» prüft, ob zwei Objekte sich berühren
- die «Continuous Collission» berechnet die Zeit bis sich zwei Objekte berühren, wenn sie ihre Bewegung fortsetzen

Bei der «Continuous Collission» wird das sogenannte «Tunneling» verhindert, bei dem sich schnell bewegende Objekte durch andere Objekte zwischen den Überprüfungen hindurch bewegen [2].

2.2.2 Das Dynamic Tree Modul

Ein Spiel besteht in der Regel aus sehr vielen Objekten. Für den Spielverlauf muss häufig die Beziehung zwischen Objekten geprüft werden, zum Beispiel zur Prüfung:

- ob zwei Objekte miteinander kollidieren,
- ob ein Objekt aus Perspektive der Spielfigur sichtbar ist,
- oder zum Ermitteln des Bodens.

Für die Berechnung werden sogenannte Casts verwendet, die man sich als Linien (Raycast) oder Flächen (Boxcast) vorstellen kann. Mit den Casts wird geprüft, ob sich diese mit den Objekten schneiden. Die Längen der Casts sind konfigurierbar, um das Verhalten der Objekte zu bestimmen, zum Beispiel wie frühzeitig ein gegnerisches Objekt die Spielfigur erkennt und auf sie zusteuert.

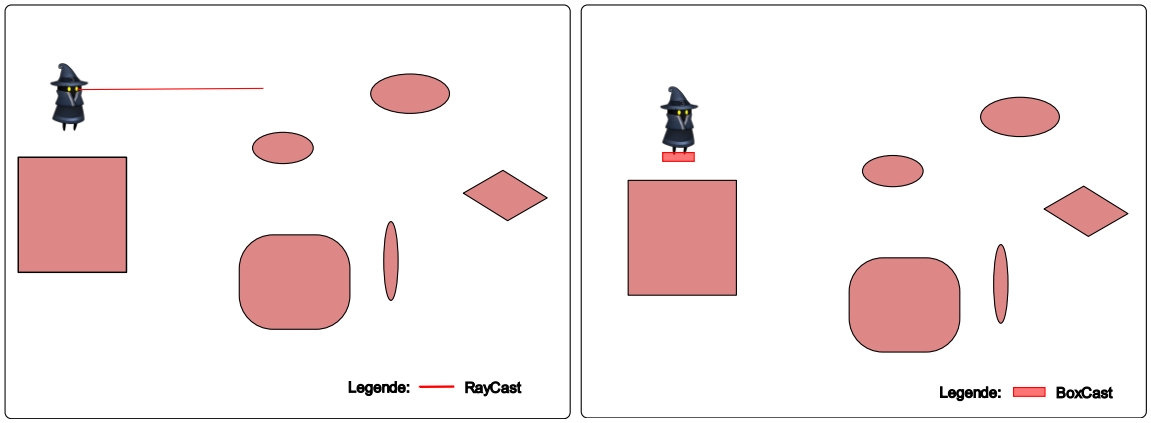

Abbildung 3: Spielfigur mit Raycast und Boxcast

Die folgende Grafik illustriert einen Cast und sieben Objekte, wobei zwei der Objekte vom Cast geschnitten werden.

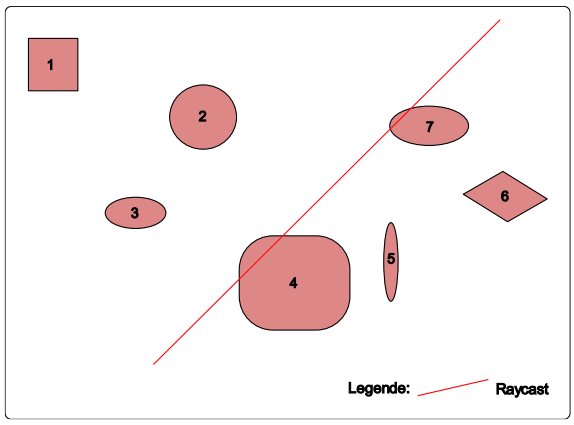

Abbildung 4: Mehrere Objekte und ein Raycast

Die Berechnung von Schnittpunkten komplizierter Objekt-Formen wäre sehr langsam, daher werden die Objekte durch rechteckige Rahmen eingegrenzt. Für jedes Objekt wird mit den oberen und unteren Ecken sogenannte Axis Aligned Bounding Boxes (AABB) berechnet, wie am Beispiel der folgenden Grafik zu sehen ist [6].

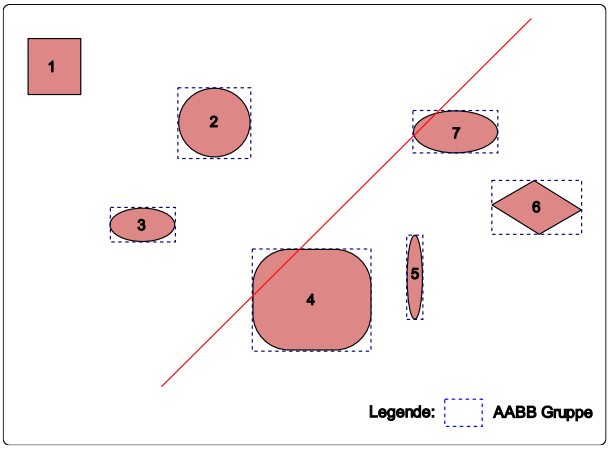

Abbildung 5: Mehrere Objekte, deren AABBs und ein Raycast

2.2.2.1 Optimierung der Berechnungen mit Bounding Volume Hierarchy

Je mehr Objekte vorhanden sind, desto länger würde die Prüfung dauern, wenn jedes Objekt einzeln berechnet wird, was man Brute Force Vorgehen nennt. Um die Effizienz der Berechnung zu verbessern, werden die AABBs der Objekte in Gruppen zusammengefasst, die wiederum in einer Hierarchy abgespeichert werden. Die Hierarchy muss einmalig berechnet werden und kann anschliessend für die effiziente Suche genutzt werden.

Wenn ein Raycast eine AABB-Gruppe trifft, wird die Berechnung innerhalb dieser Gruppe fortgeführt, wodurch insgesamt weniger Berechnungen benötigt werden [6].

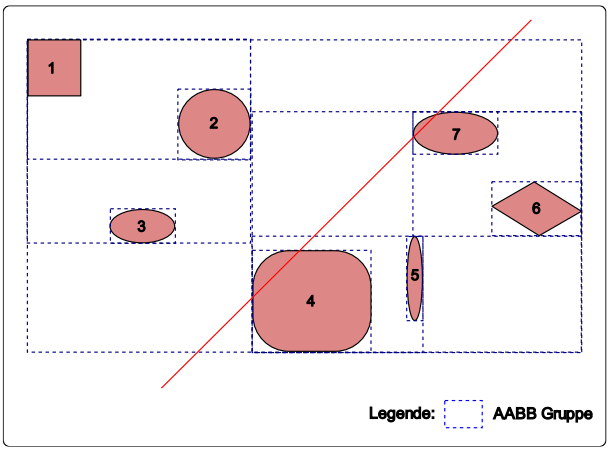

Abbildung 6: AABB-Gruppen der Objekte

Für die Erstellung der AABB-Gruppen werden die AABBs von jeweils zwei Objekten in ein neues AABB zusammengefasst. Die verschiedenen Varianten für den Algorithmus werden im Kapitel «2.2.2.3 Erstellung der Bounding Volume Hierarchy» vorgestellt.

Dieser Vorgang wird wiederholt, bis eine AABB-Gruppe übrig ist, die alle Objekte beinhaltet. Die AABBs der Objekte und der Objektgruppen werden in einem Binärbaum, einer sogenannten Bounding Volume Hierarchy (BVH) gespeichert [7].

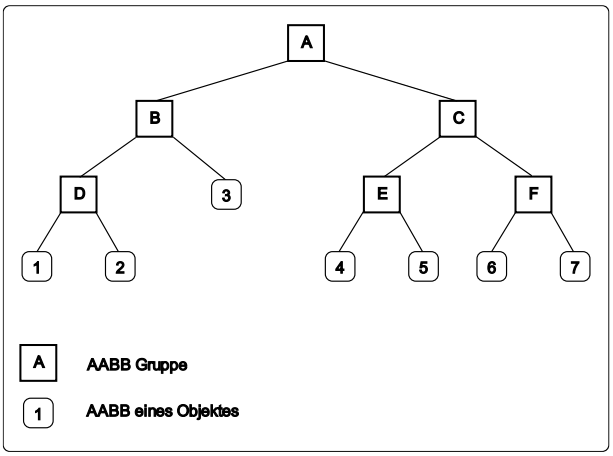

Abbildung 7: Bounding Volume Hierarchy der AABB-Gruppen

2.2.2.2 Such-Algorithmus mit Bounding Volume Hierarchy

Durch die Bounding Volume Hierarchy können zuerst die grössten AABBs kontrolliert werden, wodurch bereits viele Objekte ausgeschlossen werden können, falls eines der AABBs an der Spitze der Hierarchie nicht vom Raycast getroffen wird.

Dieser Vorgang wird wiederholt bis die AABBs nur noch einzelne Objekte enthalten. Anschliessend wird die Berechnung für die tatsächliche Objektform, die im AABB enthalten ist, durchgeführt [6].

2.2.2.3 Erstellung der Bounding Volume Hierarchy

Es gibt drei Varianten für die Erstellung der Bounding Volume Hierarchy [7]:

- Bottom Up
- Top Down
- Incremental

Das Vorgehen für die drei Varianten wird nur prinzipiell beschrieben, da es in den Physik-Engines unterschiedliche Implementierungsvarianten gibt. Die exakte Implementierung in Box2D ist in der Dokumentation nicht beschrieben.

Beim Bottom Up Vorgehen werden die Objekte mit dem jeweiligen Nachbarobjekt in eine Gruppe zusammengefasst. Danach wird der Vorgang mit den AABBs der neu erstellten Gruppen wiederholt, bis nur noch eine AABB übrig ist. Mit dem Bottom Up Vorgehen werden alle Objekte gleichmässig in der Bounding Volume Hierarchy verteilt [6].

Bei dem Top Down Verfahren wird zuerst die übergreifende AABB erstellt, die alle Objekte beinhaltet. Danach wird die übergreifende Gruppe in zwei Gruppen geteilt. Für die Teilung gibt es zwei Möglichkeiten. Die AABB-Gruppe wird entweder in der Mitte aufgeteilt oder es werden die zwei AABBs mit der geringsten Oberfläche gesucht. Bei der ersten Variante kann die Unterteilung durch die einfachere Berechnung effizienter durchgeführt werden, die dadurch entstehenden Gruppen sind jedoch sehr unterschiedlich. Bei der zweiten Variante dauert das Erstellen der Bounding Volume Hierarchy länger, die entstehenden Gruppen sind jedoch bei der späteren Berechnung effizienter, weil die AABBs kleiner werden und dadurch weniger Objekte überprüft werden müssen [7].

Das inkrementale Verfahren fängt mit einem einzelnen Objekt im Bounding Volume Hierarchy an und es werden die weiteren Objekte eingefügt und mit einem anderen Objekt in eine AABB zusammengefasst. Der dadurch entstehende Baum ist meistens ineffizient und kann durch einen Algorithmus verbessert werden [8].

Je kleiner die Flächen der AABBs sind, desto geringer ist die Wahrscheinlichkeit, dass ein Raycast das AABB trifft. Daher ist das Verfahren für die Erstellung der Bounding Volume Hierarchy sehr wichtig für die Effizienz der Berechnungen und damit für die Reaktionsgeschwindigkeit des Spieles. Bei mehreren tausend Objekten in einer Szene macht es einen erheblichen Rechenunterschied, ob alle, nur die Hälfte der Objekte oder sogar weniger überprüft werden müssen.

In der Dokumentation wird nicht erwähnt, welcher Algorithmus in Unity verwendet wird.

2.2.2.4 Algorithmus zum Einfügen weiterer Objekte

Wenn ein neues Objekt in die Bounding Volume Hierarchy eingefügt werden soll, wird mit einem der anderen Objekte eine neue AABB-Gruppe gebildet, die beide Objekte enthält. Die darüberliegenden AABB-Gruppen werden so angepasst, dass sie die neue AABB-Gruppe einschliessen [9].

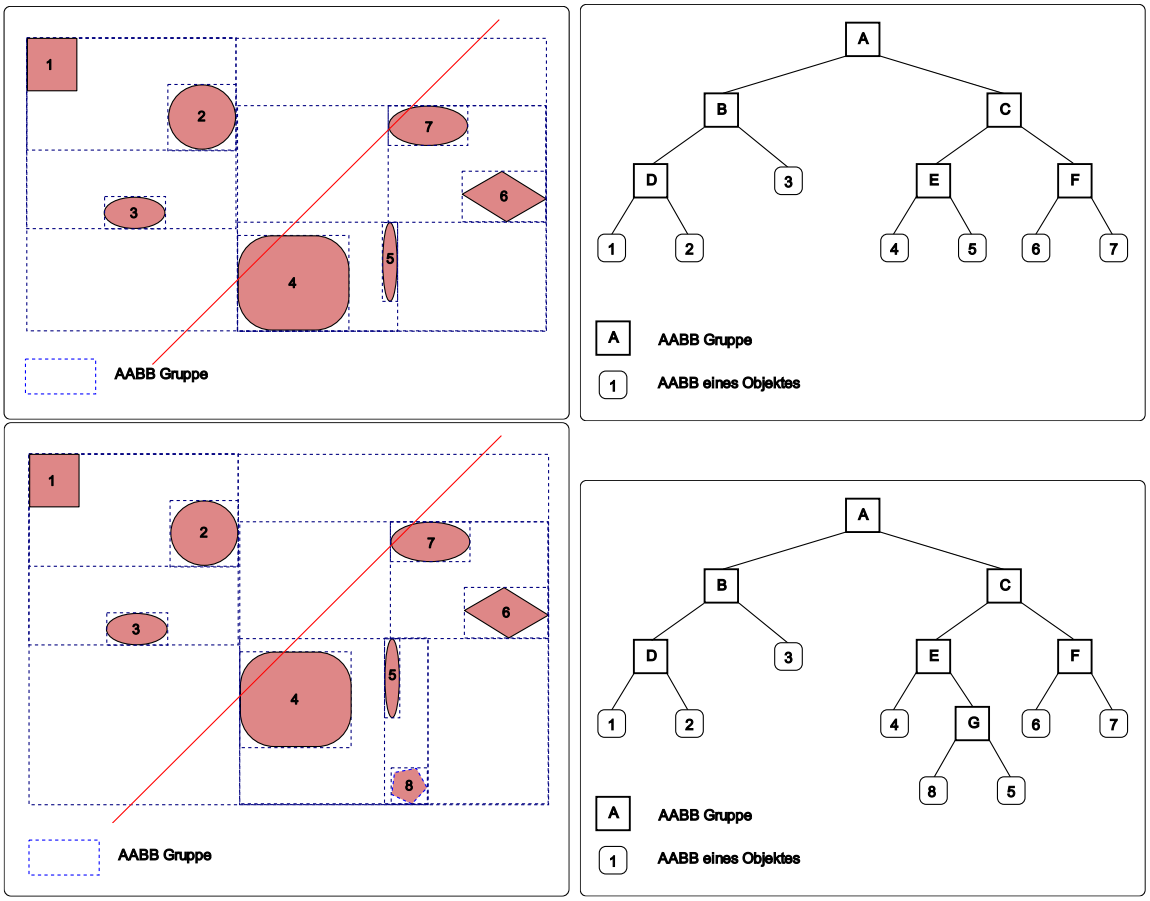

Abbildung 8: Grafik vor und nach dem Einfügen eines neuen Objektes

Für die Berechnung der besten Option für die neue AABB-Gruppe werden die Einfügekosten des neuen Objektes in den bereits vorhandenen Gruppen verglichen. Unter Einfügekosten versteht man die Summe der Zunahmen aller Flächen der betroffenen AABB-Gruppen [9].

Kostenfunktion einer Bounding Volume Hierarchy:

$C(T) = \sum_{i \in Nodes} SA(i)$

Hierbei steht C(T) für die Kosten der Bounding Volume Hierarchy und SA für die Surface Area eines Objektes.

Der folgende C++-Code illustriert die Kostenberechnung für eine Bounding Volume Hierarchy.

```
float ComputeCost(Tree tree)
{
  Float cost = 0.0f;
  For (int i=0; i<tree.nodeCount; ++i)
  {
    Cost += Area(tree.nodes[i].box);
  }
  return cost;
}
```

Abbildung 9: Kostenfunktion als C++-Code

Auch hier gilt es, einen effizienten Weg für die Berechnung zu finden, ohne die Kosten aller Möglichkeiten berechnen zu müssen. Mit Hilfe der Surface Area Heuristic (SAH) wird der Vergleich für jede AABB-Gruppe durchgeführt und die beste Option wird weiterverfolgt.

Die Einfügekosten setzen sich aus der Fläche der neuen AABB-Gruppe und der Summe der Flächenvergrösserung der darüber liegenden AABB-Gruppen zusammen [9].

Im oben genannten Beispiel würde sich folgende Formel für das achte Objekt ergeben:

$$C_8 = SA(8) + \Delta SA(G) + \Delta SA(E) + SA(E) + SA(C) + SA(A)$$

3 Die Entwicklung von «Cut Capers»

Ich habe das Spiel «Cut Capers» genannt, da «to cut a caper» im Englischen «Luftsprünge machen» bedeutet.

3.1 Einarbeitung in Unity

Bevor ich anfing Unity auszuprobieren, habe ich die Grundlagen zu C# auf w3schools gelernt [10]. Nach der Installation von Unity und Visual Studio programmierte ich eine erste Version des Spiels mit Hilfe des Tutorials von Pandemonium [11]. Zuerst programmierte ich die Steuerung des Spielers, danach Animationen, Hindernisse, die Lebenspunkte des Spielers und Checkpoints. Der Vorteil des Tutorials ist das schrittweise Vorgehen, womit man Unity sehr gut verstehen lernt. Andere Tutorials bieten fertige Code-Fragmente an, die viele Funktionen abdecken aber schwerer zu verstehen sind. Obwohl ich die Grundlagen von C# schon angeschaut hatte, konnte ich davon nur wenig profitieren, da die benötigten Befehle spezifisch für Unity sind. Das ist eine allgemeine Eigenschaft von Frameworks, das heisst, dass man eine Einarbeitung in das jeweilige Framework benötigt, auch wenn die Programmiersprache bereits bekannt ist.

Während des Tutorials fiel mir auf, dass Microsoft Visual Studio eine automatische Vervollständigung namens IntelliSense hat, dass die Entwicklung erheblich vereinfacht. Damit konnte ich viele Fehler vermeiden, zum Beispiel bei Case-sensitiven Methodennamen bei der CamelCase-Schreibweise. Bis zum Ende des Tutorials fand ich mich immer besser zurecht und konnte grösstenteils auch ohne Hilfe meine Ideen programmieren.

Um meinen Fortschritt zu speichern, installierte ich GitHub. Damit konnte ich die vorherigen Programmstände wiederherstellen im Fall, dass etwas gar nicht mehr funktionieren sollte.

Für weitere Teile Unitys, wie der Führung der Kamera mit einem Add-on, dem Hauptmenu und dem Speichern und Laden des Spielstandes in einer Binärdatei, habe ich noch weitere Hilfe von Tutorials und der Unity Community benötigt.

Im Laufe der Einarbeitung konnte ich mit Hilfe des Unity Manuals [2] weitere Funktionen und Befehle verwenden. Trotzdem war es teilweise sehr schwierig, den richtigen Befehl mit der passenden Verwendung zu finden.

Das folgende Unity-Projekt zeigt an einem kleinen Beispiel, wie man mit Unity und C# entwickelt, und wie die Physik-Engine bereits bei einem sehr kleinen Beispiel den Entwicklungsaufwand erheblich reduziert.

3.1.1 Erstellung eines Unity-Projektes für ein einfaches Spiel

Wenn man ein neues Unity Projekt erstellt, sieht man als erstes die Hierarchie aller Objekte im linken Panel, die Szene im mittleren Panel und den Inspector im rechten Panel. In der Hierarchie können Objekte in die Szene hinzugefügt werden. Im Screenshot des Beispiels sind es zwei Objekte: ein Rechteck, das ich *Ground* genannt habe und eine Grafik für die Spielfigur. Der Spielfigur habe ich eine Strichmännchen-Grafik im Sprite Renderer zugewiesen, der sich im Inspektor Panel befindet. Die Kamera, durch die der Spieler das Spiel in der Szene sieht, wird automatisch hinzugefügt.

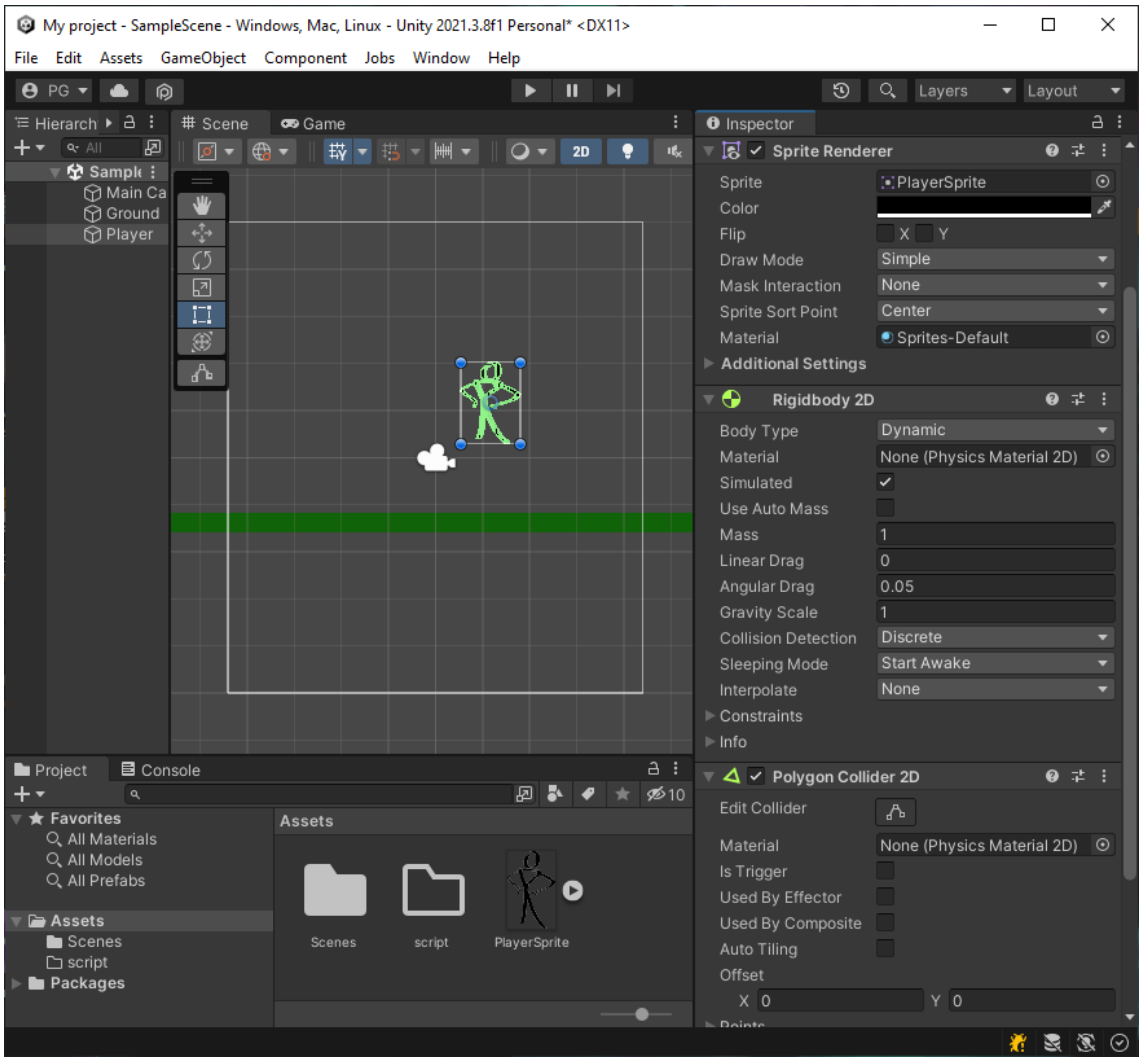

Abbildung 10: Einfaches Beispiel in Unity

Das Beispiel soll zeigen, wie die Spielfigur aufgrund der Gravitation auf den Boden fällt. Damit die Objekte zusammenstossen können, habe ich dem Boden einen BoxCollider2D und der Spielfigur einen PolygonCollider2D zugewiesen. Der PolygonCollider2D erkennt die Umrisse der Grafik und erstellt automatisch den entsprechenden Collider, der im Scene Panel grün dargestellt ist. Damit sich die Spielfigur bewegen kann und weitere physikalische Eigenschaften, wie die Gravitation, bekommt, habe ich ihm einen Rigidbody2D zugewiesen.

Wenn man das Spiel startet, fällt die Spielfigur auf den Boden und kippt um, da die Beine der Grafik nicht gleich lang sind.

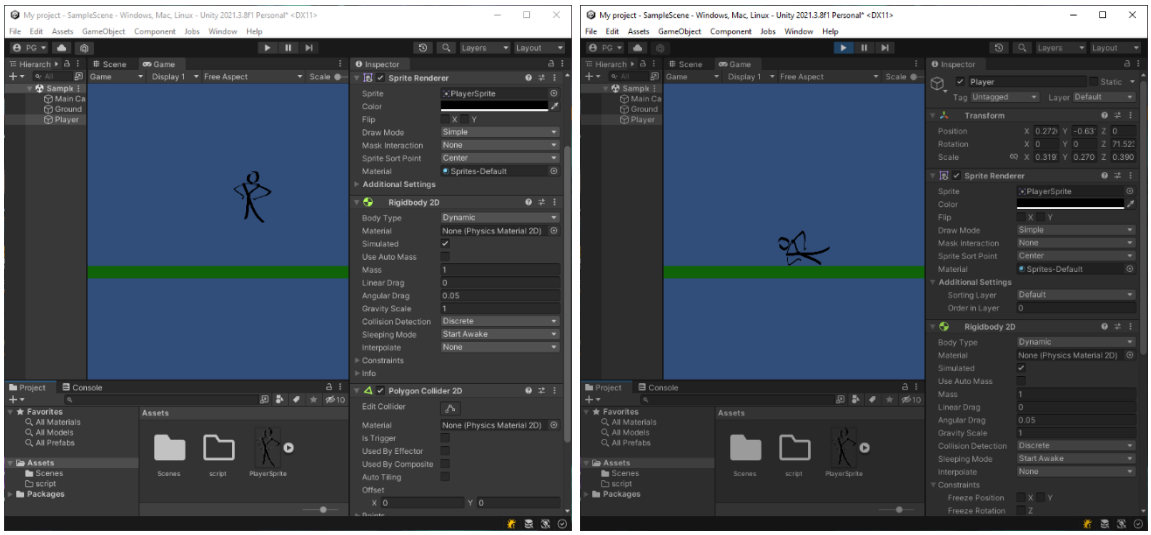

Abbildung 11: Physik-Engine in einem einfachen Beispiel

Das kleine Beispiel zeigt bereits, wie mit wenigen Schritten ein minimales Spiel erstellt werden kann, dessen Objekte aufgrund der Physik-Engine von Unity ein komplexes Verhalten erhalten.

3.1.2 Programmierung der Spielfigur mit C#

Im nächsten Schritt soll dem Spieler die Möglichkeit gegeben werden, die Spielfigur nach links und rechts zu bewegen. Damit das Objekt nicht mehr umkippt, muss in den Eigenschaften des Objektes die Rotation der Z-Achse fixiert werden. Danach wird ein C#-Skript erstellt und im Inspektor der Spielfigur zugewiesen.

Wenn die Spielfigur geladen wird, wird die Start Methode des C#-Skriptes ausgeführt. Darin wird der Variable Body die Rigidbody2D-Komponente der Spielfigur zugewiesen. In der Update Methode, welche bei der Berechnung jedes Frames aufgerufen wird, wird die Eingabe, der von Unity standardmässig belegten Tasten in die horizontale Bewegung der Spielfigur übersetzt.

Das folgende C#-Skript zeigt die wenigen Anpassungen die notwendig sind, um die Spielfigur bewegen zu können.

```
public class PlayerMovement : MonoBehaviour
{
   private Rigidbody2D body;
   [SerializeField] private int speed;
   void Start()
   {
      body = GetComponent<Rigidbody2D>();
   }
  void Update()
   {
      body.velocity = new Vector3(Input.GetAxis("Horizontal")* speed, 0, 0);
   }
}
```

3.2 Entwicklung von Spielideen

Für den Einstieg in die Spieleprogrammierung entschied ich mich dazu, ein *Platform Game* zu entwickeln, in deutschsprachigen Ländern wird häufig auch von Jump ‘n’ Run Spielen gesprochen.

Die Grundfähigkeiten der Spielfigur ist, wie es der Name der Spielgattung sagt, «rennen» und «springen». Im Spielverlauf kommen noch «Mehrfachsprünge», «klettern» und «Air Dashing» hinzu. Letzteres wird immer Englisch ausgedrückt, im Deutschen könnte man es mit «flitzen» übersetzen.

Die 2D-Landschaft soll mehrere Levels umfassen mit unterschiedlichen Grafiken. Im Laufe des Spiels werden die Fähigkeiten der Spielfigur erweitert, so wie es bei gängigen Plattformspielen üblich ist. Die Spielfigur muss die Fähigkeiten im Spiel einsammeln.

Im ersten Level sind die Grafikelemente noch statisch, die Spielfigur muss die Hindernisse überwinden und den Weg durch die Spielewelt finden. In den höheren Levels bewegen sich die Objekte und können die Spielfigur töten.

Wenn die Spielfigur stirbt, kann das Spiel vom letzten Checkpoint aus weiter gespielt werden, bis die Spielfigur alle Levels durchlaufen hat.

In den Spieleinstellungen können Grafik und Audio des Spiels angepasst werden und der Spielstand gespeichert werden, bzw. ein gespeicherter Spielstand geladen werden.

3.3 Entwicklung des Spiels

In diesem Kapitel wird die Entwicklung des Spiels erklärt. Wie bereits im Kapitel «3.1 Einarbeitung in Unity» beschrieben wurde, gehören die Entwicklung der Spielewelt in Unity und die Entwicklung der passenden C#-Skripte, in denen das Verhalten der Spielfigur und anderer beweglicher Objekte implementiert wird, dazu.

3.3.1 Entwicklung der Spielewelt in Unity

3.3.1.1 Statische Objekte der Spielewelt

Bei der Erstellung der Spielewelt war die Suche nach den passenden Grafiken recht aufwendig. Der Unity Asset Store und die Webseite itch.io waren sehr hilfreich. Die Unity Erweiterung Auto-Tiling war bei der Verwendung der Grafiken von Maoot [12] sehr hilfreich, damit kann man die Grafiken per Drag&Drop einfügen und Auto-Tiling verwendet die jeweils passende Grafik.

Meine Spielewelt basiert auf zwei Tilemaps, die als Gitter ersichtlich sind, in dem die Grafiken platziert werden können. Die erste Tilemap enthält die reinen Hintergrundobjekte, die keine Kollisionen auslösen, das heisst die Grafiken sind nur für die Optik des Spieles eingefügt. Die zweite Tilemap enthält die Böden und Wände der Spielewelt. In der Transform Eigenschaft, werden Grösse, Position und Rotation definiert.

Der Tilemap für die Böden und Wände wurde zudem eine Tilemap Collider Komponente zugewiesen, damit die enthaltenen Objekte unter der Kontrolle der Physik-Engine stehen. Die RigidBody2D-Komponenten der Objekte in dieser Tilemap sind auf statisch eingestellt, das heisst, die Objekte bewegen sich nicht. Der Tilemap Collider bildet Polygone um die eingefügten Grafiken, die für die Berechnung der Kollisionen notwendig sind. Durch die Composite Collider Komponente werden die Polygone durch eine ChainShape ersetzt, deren Einstellungen man verändern kann, um die Oberfläche zu glätten. Damit wird verhindert, dass die Spielfigur an kleinen Unebenheiten hängen bleibt.

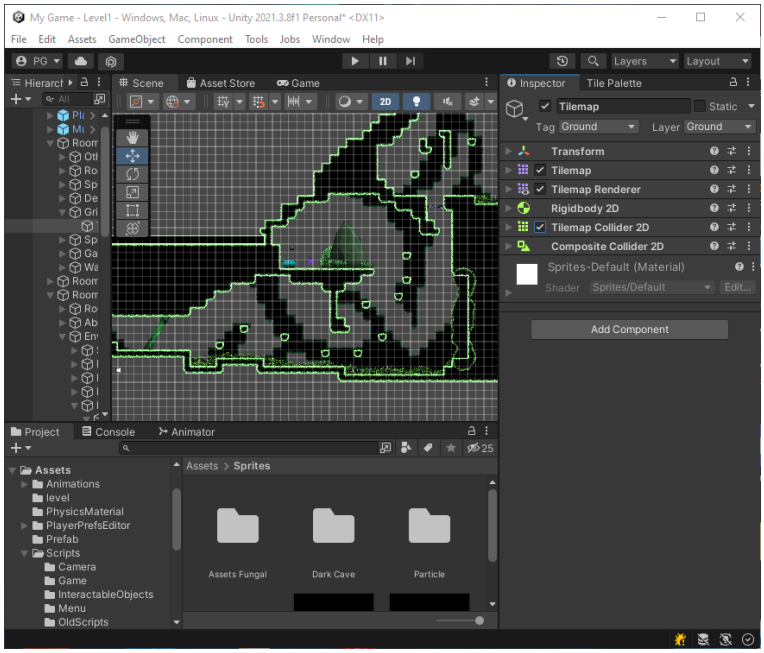

Abbildung 12: Statische Objekte der Spielewelt

3.3.1.2 Bewegliche Objekte der Spielewelt

Die Spielfigur benötigt einige Komponenten, die im Inspektor aufgelistet werden. Um die Kollisionen der Spielfigur berechnen zu können sind ihr mehrere Collider zugewiesen. Ein Box Collider beinhaltet die Spielfigur ohne ihre Füsse, er hat keine Reibung damit die Spielfigur nicht an Wänden hängenbleibt. Der Edge Collider bildet zwei Schrägen über den Füssen mit dem gleichen Material ohne Reibung damit der Spieler nicht an Unebenheiten im Boden hängenbleibt. Der zweite Box Collider beinhaltet nur die Füsse der Spielfigur, er hat ein Material mit Reibung, da er den Boden berührt.

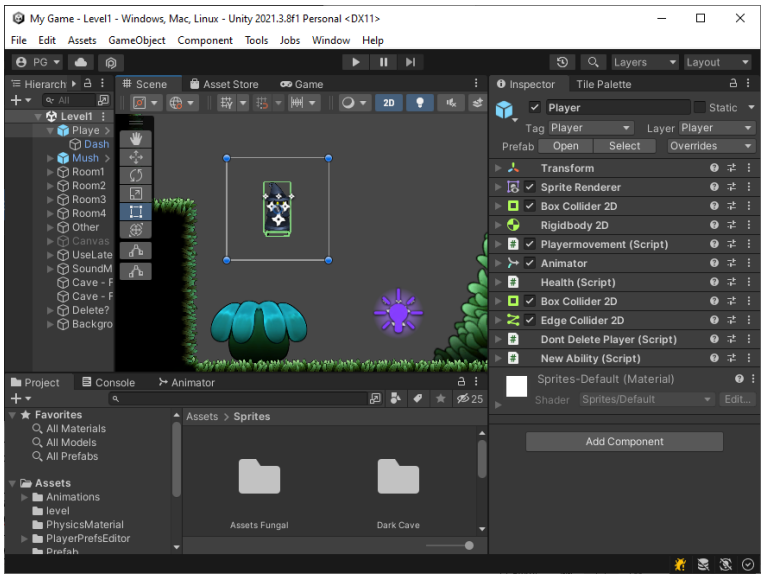

Abbildung 13: Bewegliche Objekte der Spielewelt

Damit sich die Spielfigur bewegen kann, ist ihm das *Playermovement* Skript zugewiesen, das alle Bewegungen steuert. Das *Health* Skript steuert die Lebenspunkte der Spielfigur, das *New Ability* Skript kontrolliert das Aufsammeln neuer Fähigkeiten. Damit neue Spielstände in einer Szene geladen werden können und diese Informationen gespeichert bleiben, besitzt die Spielfigur das *Dont Delete Player* Skript. Die *Animator* Komponente steuert die Animation der Spielfigur.

3.3.2 Entwicklung des Verhaltens der Objekte in C#

Im vorhergehenden Kapitel wurden die Skripte der Spielfigur definiert. In den folgenden Kapiteln werden Snipplets des *Playermovement* Skripts und des *New Ability* Skripts erklärt.

3.3.2.1 Erkennung der Bewegungsrichtung

Das Spiel soll sowohl mit der Tastatur als auch mit dem Game Controller gespielt werden können. Die zwei Eingabevarianten unterscheiden sich jedoch im Wertebereich. Während die Tastatur nur die Ganzzahlen [-1, 0, 1] liefert, wird vom Game Controller ein Dezimalwert zwischen -1 und 1 geliefert. Im folgenden Snipplet wird der Dezimalwert des Game Controllers in die entsprechenden Ganzzahlen umgerechnet.

```
if (inputManager.Land.MoveHorizontal.ReadValue<float>() != 0)
    {
      Horizontalinput = Mathf.Sign(inputManager.Land.MoveHorizontal.ReadValue<float>());
    }
    else
    {
      Horizontalinput = 0;
    }
```

Danach wird mit dem nachfolgenden Snipplet die Blickrichtung des Spielers der Bewegungsrichtung angepasst.

```
 if (Horizontalinput > 0.01f)
   transform.localScale = playerSize;
else if (Horizontalinput < -0.01f)
   transform.localScale = new Vector3(-playerSize.x, playerSize.y, playerSize.z);
```

3.3.2.2 Erkennung der Berührung von Wänden und Boden

In der *isGrounded()* Methode wird überprüft, ob sich die Spielfigur auf dem Boden befindet, dafür wird ein Boxcast nach unten verwendet. Wenn dieser sich mit einem Objekt mit dem Marker *GroundLayer* überschneidet, ist der Return Wert *true*.

In der *onWall()* Methode wird überprüft, ob die Spielfigur schon weit genug im Spielverlauf ist, um an Wänden klettern zu können. Sofern *wallInteraction* den Wert *true* hat, wird wiederum ein Boxcast verwendet, um zu prüfen, ob sich die Spielfigur in ihrer Blickrichtung an einer Wand befindet.

```
private bool isGrounded()
  {
      RaycastHit2D raycastHit = Physics2D.BoxCast(boxCollider.bounds.center,
      boxCollider.bounds.size, 0, Vector2.down, 0.1f, groundLayer);
      return raycastHit.collider != null;
  }
  private bool onWall()
  {
     if (wallInteractions == true)
     {
        RaycastHit2D raycastHit = Physics2D.BoxCast(boxCollider.bounds.center,
        boxCollider.bounds.size, 0, new Vector2(transform.localScale.x, 0), 0.1f, wallLayer);
        return raycastHit.collider != null;
     }
     else
     {
        return false;
     }
  }
```

3.3.2.3 Variable Sprunghöhen

Damit der Spieler die Sprunghöhe der Spielfigur steuern kann, wird im ersten *if* Statement die Gravitation halbiert, solange er die Sprungtaste gedrückt hält.

Wenn die Spielfigur sich an der Wand befindet, muss eine weitere Taste gedrückt werden, damit sich die Spielfigur an der Wand festhält. Im zweiten *if* Statement wird die Gravitation durch 10 geteilt, damit die Spielfigur langsam an der Wand herunterrutscht.

```
if (inputManager.Land.jump.ReadValue<float>() == 0 && Body.velocity.y > 0 && !isDashing)
     {
        Body.velocity = new Vector2(Body.velocity.x, Body.velocity.y / 2);
     }
if (onWall() && inputManager.Land.RT.ReadValue<float>() != 0)
     {
        Body.gravityScale = Gravity / 10;
        Body.velocity = Vector2.zero;
     }
```

3.3.2.4 Umsetzung von unterschiedlichen Sprüngen

Bei der Berechnung eines neuen Frames wird in der *Update()* Methode geprüft, ob der Spieler gerade die Sprungtaste drückt, daraufhin wird die *Jump()* Methode aufgerufen.

Beim ersten *if* Statement wird geprüft, ob die Spielfigur gerade springen darf.

Im zweiten *if* Statement wird geprüft, ob die Spielfigur an einer Wand ist und lässt sie von dieser Position mit der *WallJump()* Methode abspringen.

Im dritten *if* Statement wird geprüft, ob die Spielfigur sich am Boden befindet. Das Spiel würde sich sehr schwer spielen lassen, wenn der Absprung vom Boden nur möglich wäre, wenn die Spielfigur im Kontakt mit dem Boden ist. Um das Springen angenehmer für den Spieler zu machen wird ein sogenannter Coyote Timer verwendet, der es dem Spieler erlaubt, auch wenn der Bodenkontakt bereits abgebrochen ist, einen Sprung durchzuführen. Der Coyote Timer wird häufig auf etwa 0.25 Sekunden eingestellt, innerhalb dieser Zeitspanne kann die Spielfigur springen, auch wenn sie den Kontakt zum Boden bereits verloren hat.

In einem weiteren *if* Statement wird geprüft, ob die Spielfigur einen weiteren Sprung in der Luft machen kann.

```
private void Jump()
  {
    if (coyotecounter < 0 && !onWall() && jumpCounter <= 0) return;
    SoundManager.Instance.PlaySound(JumpSound);
    if (onWall() && inputManager.Land.RT.ReadValue<float>() != 0) WallJump();
    else
    {
      if (isGrounded())
      {
        Body.velocity = new Vector2(Body.velocity.x, JumpPower);
      }
      else
      {
        if (coyotecounter > 0)
          Body.velocity = new Vector2(Body.velocity.x, JumpPower);
        else
        {
          if (jumpCounter > 0)
          {
            Body.velocity = new Vector2(Body.velocity.x, JumpPower);
            jumpCounter--;
          }
        }
      }
      coyotecounter = 0;
    }
  }
  private void WallJump()
  {
    Body.AddForce(new Vector2(-Mathf.Sign(transform.localScale.x) * wallJumpX, wallJumpY));
  }
```

3.3.2.5 Freischaltung von Fähigkeiten

Um zu überprüfen, ob die Spielfigur eine neue Fähigkeit erwirbt, wird mit dem Befehl *OnCollisionEnter2D* nach neuen Kollisionen mit Objekten gesucht, die eine Fähigkeit freischalten. In den *if* Statements wird der Marker des Objekts, welches die Kollision verursacht hat mit dem Marker der freischaltbaren Fähigkeiten verglichen. Für die Freischaltung von Fähigkeiten, wird die jeweilige Variable im *Playermovement* Skript angepasst. Danach wird das entsprechende Objekt deaktiviert.

```
public class NewAbility : MonoBehaviour
{
  private void OnCollisionEnter2D(Collision2D collision)
  {
    if (collision.transform.CompareTag("WallAbilities"))
    {
      transform.GetComponent<Playermovement>().wallInteractions = true;
      collision.gameObject.SetActive(false);
    }
    if (collision.transform.CompareTag("DoubleJump"))
    {
      transform.GetComponent<Playermovement>().extraJumps = 1;
      collision.gameObject.SetActive(false);
    }
    if (collision.transform.CompareTag("DashAbility"))
    {
      transform.GetComponent<Playermovement>().dashAbility = true;
      collision.gameObject.SetActive(false);
    }
  }
}
```

3.3.2.6 Verwendung einer fallenden Plattform

Fallende Plattformen haben dieselbe Gravitation wie die Spielfigur, ihre Bewegung wird beim Spielstart jedoch deaktiviert. Sobald die Spielfigur die fallende Plattform berührt, wird die Bewegung nach 0.25 Sekunden aktiviert. Nach 3 Sekunden wird die fallende Plattform wieder an die ursprüngliche Stelle zurückgesetzt.

```
private Rigidbody2D Body;
  [SerializeField] private Transform StartingLocation;
  private GameObject PlayerObject;
  private void Start()
  {
    PlayerObject = GameObject.FindGameObjectsWithTag("Player")[0];
    Body = GetComponent<Rigidbody2D>();
    Body.gravityScale = PlayerObject.GetComponent<Playermovement>().Gravity;
    Body.constraints = RigidbodyConstraints2D.FreezeAll;
  }
  private void OnCollisionEnter2D(Collision2D collision)
  {
    if (collision.transform.tag == "Player")
    {
      StartCoroutine(FallPlatform());
    }
  }
  private IEnumerator FallPlatform()
  {
    yield return new WaitForSeconds(1);
    Body.constraints = RigidbodyConstraints2D.None;
    Body.gravityScale = PlayerObject.GetComponent<Playermovement>().Gravity;
    yield return new WaitForSeconds(10);
    Body.gravityScale = 0;
    transform.position = StartingLocation.position;
    transform.rotation = StartingLocation.rotation;
    Body.constraints = RigidbodyConstraints2D.FreezeAll;
  }
```

3.3.2.7 Air-Dashing

Air-Dashing ist eine schnelle Bewegung, in die vom Spieler ausgewählte Richtung. Die Distanz, welche während einem Dash zurückgelegt wird, ist immer gleich.

Beim Drücken der Dash-Taste wird die Richtung über die normalen Bewegungstasten geprüft. Die Gravitation der Spielfigur wird auf Null gesetzt und die Eingaben werden deaktiviert. Zudem werden Partikel aktiviert, die beim Dash der Spielfigur sichtbar sind. Durch das Setzen von *isDashing* auf true wird die Geschwindigkeit der Spielfigur erhöht.

```
if (inputManager.Land.Dashbutton.triggered == true && canDash)
    {
      isDashing = true;
      canDash = false;
      SoundManager.Instance.PlaySound(DashSound);
      GetComponentInChildren<ParticleSystem>().Play(true);
      dashingDirection = new Vector2(inputManager.Land.MoveHorizontal.ReadValue<float>(),
      inputManager.Land.DashDirection.ReadValue<float>());
      if (dashingDirection == Vector2.zero)
      {
        dashingDirection = new Vector2(transform.localScale.x, 0);
      }
      inputManager.Disable();
      Body.gravityScale = 0;
      Body.velocity = Vector2.zero;
      StartCoroutine(StopDashing());
    }
if (isDashing)
    {
      Body.velocity = dashingDirection.normalized * dashingVelocity;
    }
if (isGrounded() && !isDashing && dashAbility)
    {
        canDash = true;
    }
```

In der Coroutine *StopDashing()* wird für die Dauer des Dashes abgewartet. Danach wird *isDashing* wieder auf false gesetzt, die Gravitation auf den Standardwert gesetzt, die Partikel deaktiviert und die Steuerung des Spielers wieder aktiviert.

```
private IEnumerator StopDashing()
  {
    yield return new WaitForSeconds(dashingTime);
    isDashing = false;
    GetComponentInChildren<ParticleSystem>().Stop();
    inputManager.Enable();
    Body.gravityScale = Gravity;
  }
```

3.3.3 Einbindung von weiteren Features

3.3.3.1 Spielemenü

Um ein Menü in Unity einzufügen, kann ein Canvas verwendet werden, der immer den ganzen Bildschirm einnimmt. Im Canvas können dann *UserInterface* Objekte platziert werden, wie zum Beispiel Textfelder, Knöpfe, Slider oder Dropdowns. Den Objekten können anschliessend Skripte zugewiesen werden. Im folgenden Bild sieht man die Menü-Einträge, um ein neues Spiel zu starten, ein gespeichertes Spiel zu laden, die Spieloptionen anzupassen und das Spiel zu verlassen.

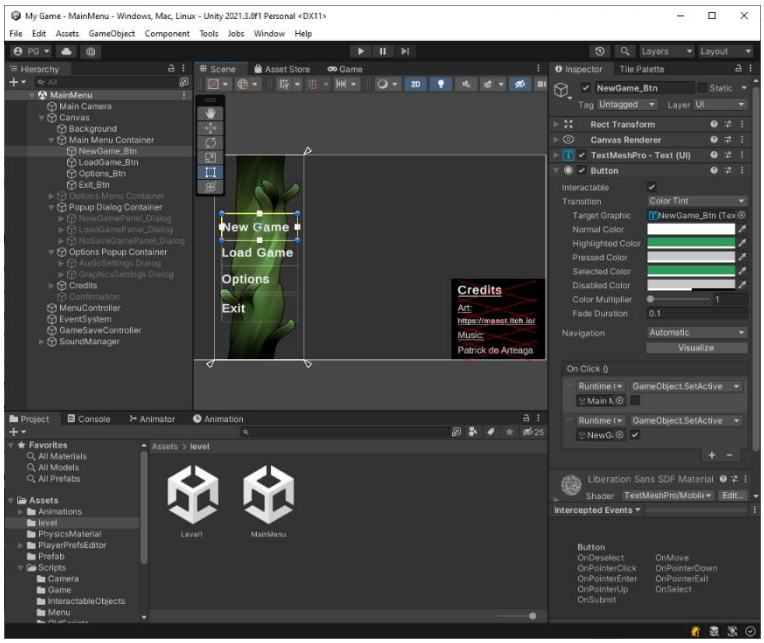

Abbildung 14: Einbindung des Spielemenüs

3.3.3.2 Audio

Die Musik in meinem Spiel ist von Patrick de Arteaga, der eine grosse Sammlung von Musikstücken für die nichtkommerzielle Nutzung gratis im Internet anbietet. Um Audiodateien abzuspielen, muss man diese in eine Audioquelle einfügen und einem *Audio Listener* in der Szene zuweisen.

3.3.3.3 Game Controller

Bei der Controller Einbindung musste ich mich zuerst zwischen zwei Erweiterungen entscheiden, wobei die neuere komplizierter, aber auch besser ist. Leider standen bei dieser weniger Anleitungen zur Verfügung, weshalb ich anfangs ein bisschen herumprobieren musste, aber schlussendlich hat es funktioniert. Bei der Neuen Version des *inputManagers* kann man unterschiedliche *ActionMaps* erstellen, wenn unterschiedliche Tastenbelegungen verwendet werden sollen, wie zum Beispiel zur Unterscheidung ob das Spiel an Land oder unter Wasser läuft. Den unterschiedlichen Aktionen können Tasten zugewiesen werden. Der Vorteil des neuen Input Managers ist, dass mehrere Tasten für jede Aktion möglich sind, das heisst, dass man die Eingaben der Tastatur und des Controllers in einem Befehl abfragen kann.

3.3.4 Rückblickende Betrachtung der Entwicklung

Nach der Einarbeitung in Unity startete ich mit einem einfachen Spiel, das ich Schritt für Schritt weiterentwickelt habe. Bei diesem iterativen Ansatz verbesserte ich abwechselnd die Grafik, die Spielewelt, die Funktionen der Spielfigur und die anderen Komponenten. Mit dem Umfang des Spieles wurden Anpassungen immer aufwendiger, da ich die bestehenden Teile nochmals überarbeiten musste. Dennoch war der «Learning by Doing» Ansatz sehr gut geeignet, um Unity zu erlernen und Spass daran zu haben, wie sich das Spiel entwickelt.

Im Nachhinein würde ich anfangs weniger Zeit in die Grafiken investieren und zuerst einen Prototyp der Mechaniken machen. Dann hätte ich beispielsweise früher gemerkt, welche Mechaniken sich noch nicht so gut anfühlen. Wenn ich das Spiel im aktuellen Zustand spiele, finde ich die Bewegung zwar gut, aber die Beschleunigung und das Abbremsen des Spielers würde ich gerne auf eine andere Art lösen. Die Anpassung der Grundlagen würde allerdings sehr viele Auswirkungen haben, die neue Fehler verursachen könnten und ich jede einzelne Fähigkeit wieder von neuen in unterschiedlichen Situationen testen müsste.

Bei einem umfangreichen Projekt würde es sicherlich sinnvoll sein vorab zu planen, welche Mechaniken und Funktionalitäten man einbauen möchte, und diese in Prototypen zu entwickeln und zu testen. Die getesteten Mechaniken und Funktionalitäten kann man anschliessend bei der Entwicklung des Spiels übernehmen.

4 Beschreibung des Spieles «Cut Capers»

Das Ziel eines Plattformspieles ist es, mit unterschiedlichen Sprüngen und Bewegungsmechaniken die unterschiedlichen Hindernisse der Spielewelt zu überwinden und das Ende des Spiels zu erreichen. Während die einfachsten Hindernisse die Sprünge von Plattform zu Plattform sind, werden diese im Verlauf des Spieles durch Plattformen, welche bei Kontakt herunterfallen oder Stalaktiten, die die Spielfigur erschlagen können, erschwert.

Die meisten Fähigkeiten der Spielfigur sind anfangs noch nicht nutzbar, sie werden im Verlauf des Spiels durch das Aufsammeln von Objekten freigeschaltet.

4.1 Die «Cut Capers» Spielewelt

Die Spielewelt gliedert sich in vier Gebiete, in denen unterschiedliche Fähigkeiten vom Spieler genutzt werden müssen, um die Hindernisse zu überqueren. Ich habe noch viele Ideen für die Erweiterungen, die es leider nicht mehr in diese Version von «Cut Capers» geschafft haben. Ich freue mich darauf das Spiel zu erweitern, Mechaniken zu überarbeiten und damit mehr über Unity und C#-Entwicklung zu lernen.

4.1.1 Plattformen und Wände

Am Anfang des Spieles muss die Spielfigur über gezielte Sprünge von Plattform zu Plattform durch die Spielewelt bewegt werden. Die Schwierigkeit der Sprünge wird stetig erhöht, daher muss der Spieler die Tastatur oder den Game Controller immer genauer verwenden und zusätzliche Tasten nutzen.

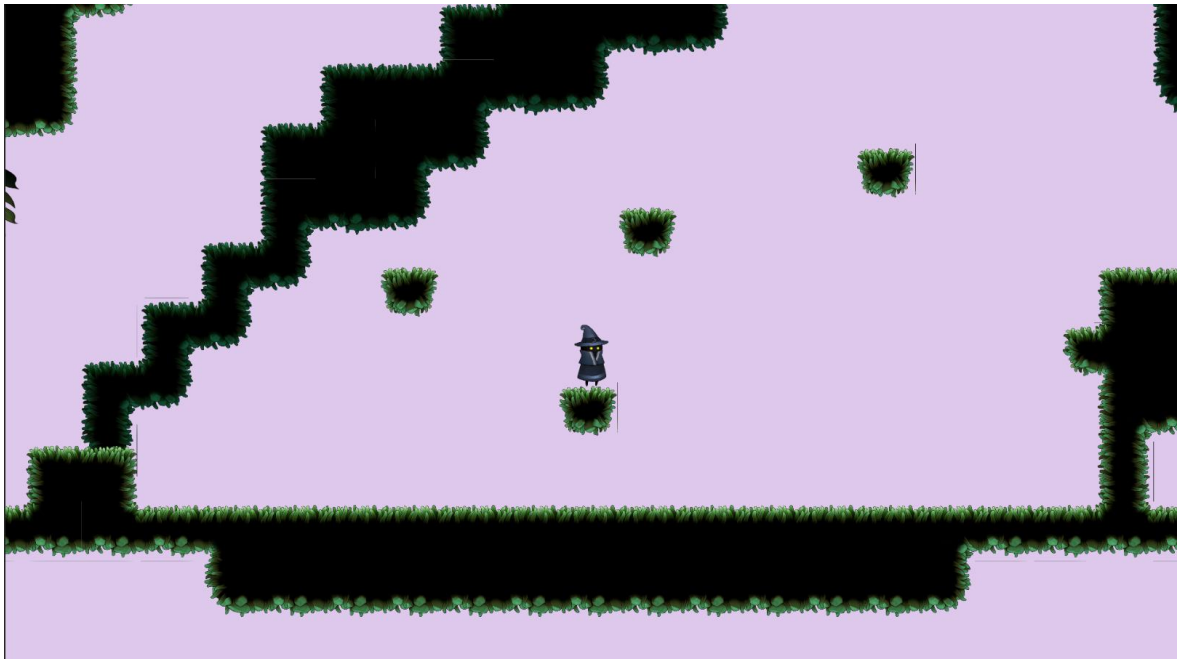

Abbildung 15: "Cut Capers" Sprünge über Plattformen

Der Boden wird zudem verändert, während die Spielfigur am Anfang auf den Boden fällt und die Sprünge wiederholen kann, fällt die Spielfigur im Verlauf des Spieles in ein Dornengestrüpp und wird an den letzten Checkpoint zurückgesetzt.

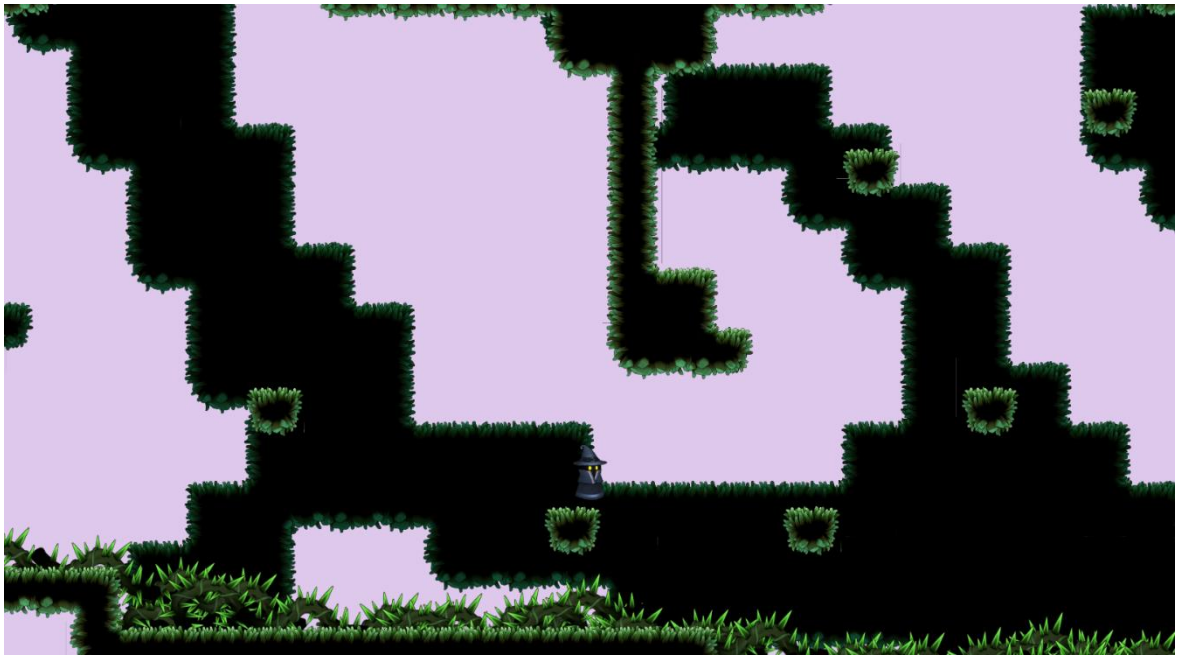

Abbildung 16: "Cut Capers" Dornengestrüpp

Bei den Wandsprüngen muss der Spieler weitere Tasten verwenden, damit die Spielfigur an den Wänden hochspringen kann.

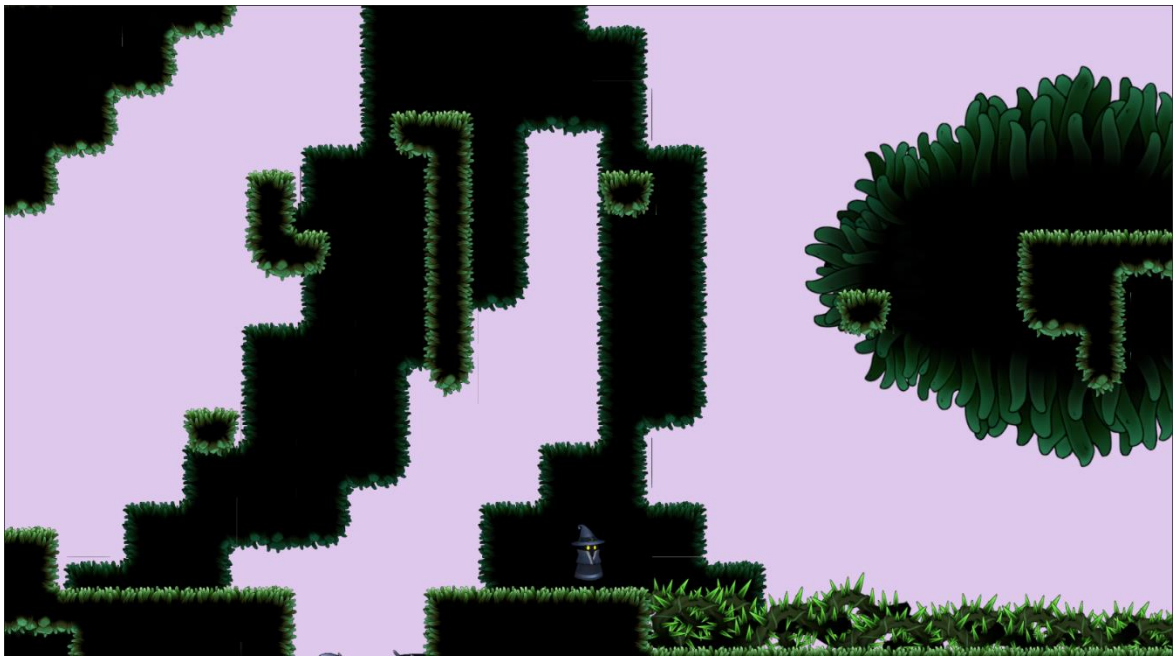

Abbildung 17: "Cut Capers" Wandsprünge

4.1.2 Bewegte Hindernisse

In diesem Gebiet muss sich die Spielfigur vor bewegten Hindernissen schützen. Ein grosser Stein rollt hinter dem Spieler her und zerstört die Plattformen.

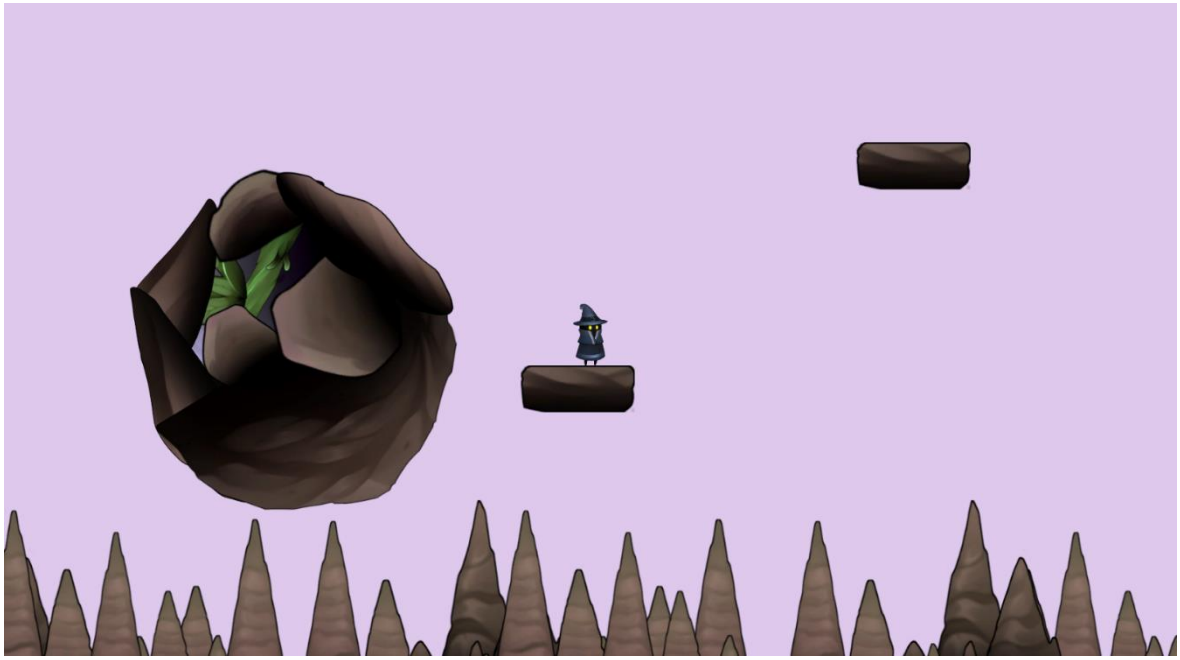

Abbildung 18: "Cut Capers" Der rollende Stein

Dünne Plattformen fallen nach Kontakt mit der Spielfigur nach unten, was dem Spieler keine Zeit lässt, über den nächsten Sprung nachzudenken.

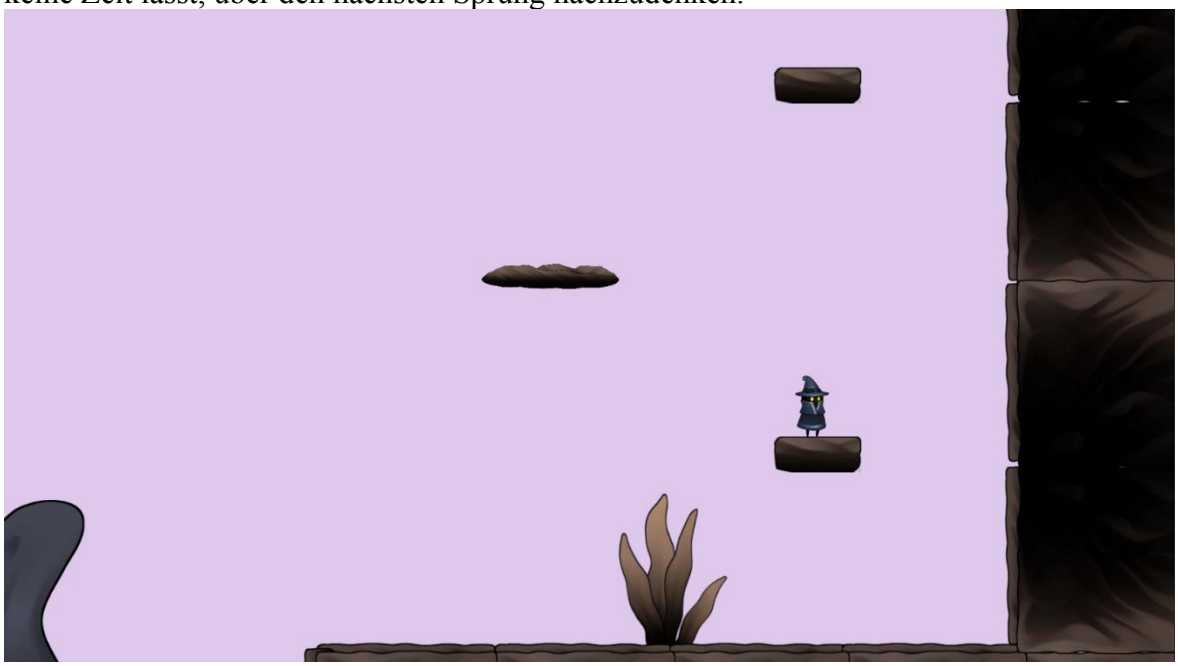

Abbildung 19: "Cut Capers" Fallende Plattformen

4.1.3 Sprungpilze und Air-Dashes

Das nächste Gebiet bringt weitere Mechaniken. Der Sprungpilz wirft die Spielfigur nach oben.

Abbildung 20: "Cut Capers" Sprungpilze

Zudem kann die Spielfigur nun mit Air-Dashes über grössere Distanzen bewegt werden. In der folgenden Abbildung benutzt die Spielfigur den Dash, um eine Schlucht zu überqueren.

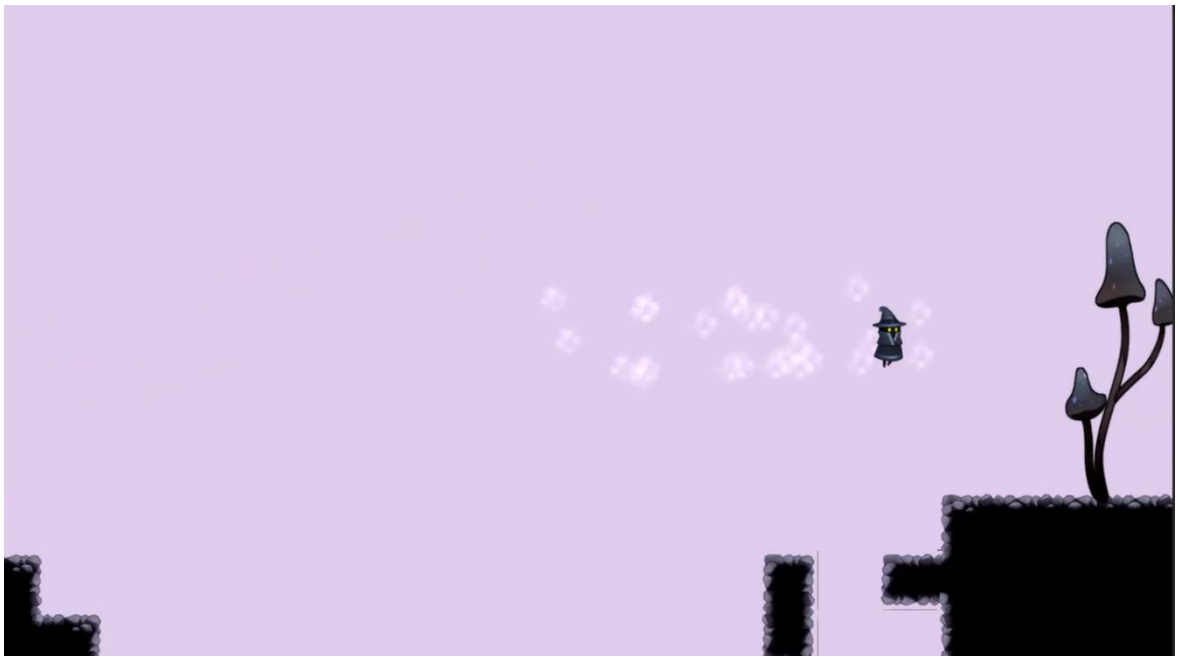

Abbildung 21: "Cut Capers" Air-Dashing

4.1.4 Angreifende Hindernisse

In diesem Gebiet werden die Hindernisse nochmals schwieriger. Der grüne Schleim bewegt sich auf einer geraden zwischen zwei Punkten hin und her und ist schwierig zu überspringen.

Abbildung 22: "Cut Capers" Der grüne Schleim

Der orange Schleim bewegt sich direkt auf die Spielfigur zu, die Spielfigur muss es durch die Höhle schaffen, bevor der Schleim sie erreicht. Dabei bleibt kaum Zeit zum Überlegen oder zur Wiederholung von Sprüngen.

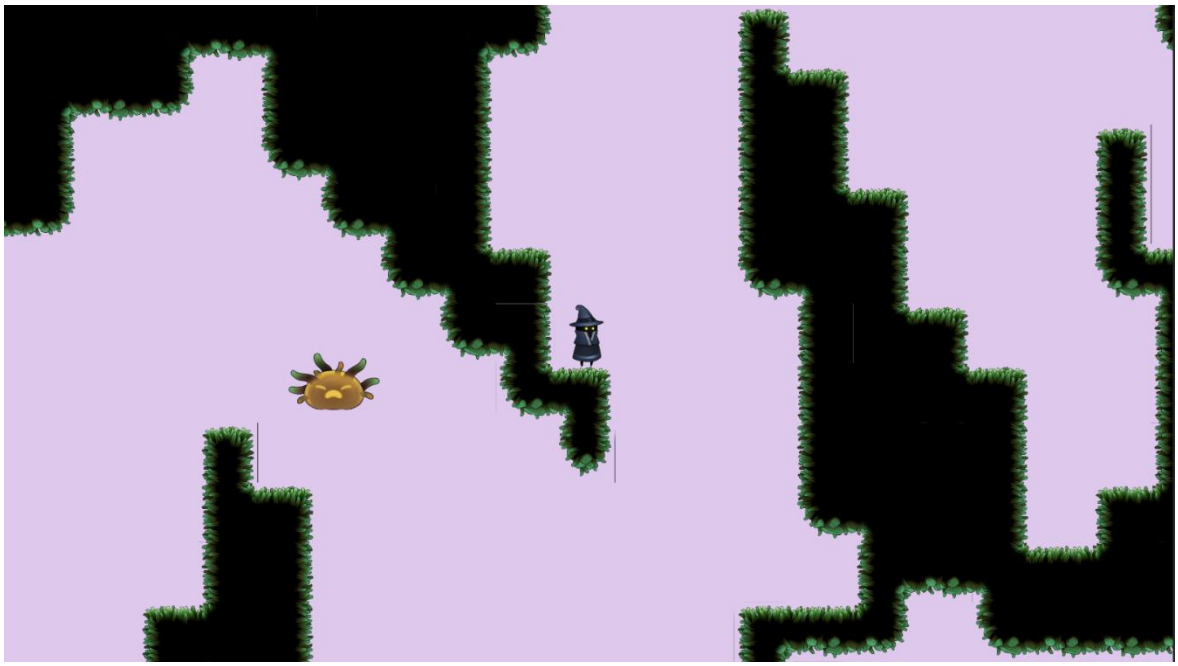

Abbildung 23: "Cut Capers" Der orange Schleim

4.2 Dokumentation, Source Code und Spiele

Die folgenden Teile dieser Arbeit stehen als Dateien zur Verfügung:

- Die Spiele-Anleitung als PDF-Dokument
- Das Unity Projekt mit allen Sourcen als ZIP-Datei
- Das Spiel «Cut Capers» als ausführbare Datei auf Windows Computern
- Das Spiel «Cut Capers» als ausführbare Datei auf Apple Computern
- Videoaufnahme des kompletten Spielablaufs

5 Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich während der Erstellung dieser Maturaarbeit unterstützt und motiviert haben.

Ich bedanke mich bei Frau Elena Fattorini, die meine Maturaarbeit betreut und bewertet hat. Sie hat sich viel Zeit genommen, um mit mir die Arbeit und das Vorgehen zu besprechen. Ihre Kritik, Ihre Anmerkungen und Ihre Ideen waren sehr wertvoll und hilfreich.

Ein weiteres Dankeschön gilt der aktiven Unity Community, mit deren Hilfe ich diverse Probleme lösen konnte und die mir Tipps bei der Programmierung geben konnte.

Abschliessend gilt ein grosses Dankeschön meinen Eltern und meiner Schwester Christina für das Korrekturlesen und die moralische Unterstützung der Arbeit.

Philipp Gempp

6 Abbildungsverzeichnis

7 Quellenverzeichnis

[1] Unity Technologies, «Unity,» 2022.
[Online]. Available: https://unity.com/
[Zugriff am 29.12.2022]

[2] Unity Technologies, «Unity Documentation,» 2.12.2022.
[Online]. Available: https://docs.unity3d.com/Manual/unity-architecture.html
[Zugriff am 29.12.2022]

[3] S. Luber, «Cloudcomputing Insider,» 5.4.2022.
[Online]. Available: https://www.cloudcomputing-insider.de/was-ist-ein-framework-a-1104630
[Zugriff am 11.12.2022].

[4] Catto Erin, «Box 2D Manual,» Box2D, 2021.
[Online]. Available: https://box2d.org/documentation/
[Zugriff am 30.10.2022].

[5] MelvMay, «Unity Technoligies,» 1.7.2020.
[Online]. Available: https://forum.unity.com/threads/composite-collider-2d-ghost-collision.923417/
[Zugriff am 29.12.2022].

[6] J. Goldsmith und J. Salmon, «Automatic Creation of Object Hierarchies for Ray Tracing,» in *IEEE Computer Graphics and Applications, vol. 7, no. 5*, 1987, pp. 14-20.

[7] J. Braun, *BVH- und Line-Space-Kombination zur Pathtracing-Beschleunigung,* Koblenz, 2017

[8] S. M. Omohundro, «Five Balltree Construction Algorithms,» International Computer Science Institute, California, 1989

[9] J. Bittner, M. Hapala und V. Havran, «Incremental BVH construction for ray tracing,» *Computers & Graphics,* Dezember 2014

[10] w3schools, «w3schools,» w3schools,
[Online]. Available: https://www.w3schools.com/
[Zugriff am 5.5.2022]

[11] Pandemonium, «YouTube,»
[Online]. Available: https://www.youtube.com/watch?v=TcranVQUQ5U&list=PLgOEwFbvGm5o8hayFB6skAfa8Z-mw4dPV&ab_channel=Pandemonium
[Zugriff am 29.12.2022]

[12] Maoot, «itch.io,»
[Online]. Available: https://maaot.itch.io/
[Zugriff am 25.7.2022]