Entwicklung eines Computerspiels mit Unity

Maturaarbeit

|  |  |
| --- | --- |
| Maturaarbeit von:  Philipp Gempp  Müracker 14  8548 Ellikon philipp.gempp@gmx.ch | Eingereicht am 09.01.2023 bei:  Elena Fattorini Kantonsschule Im Lee Rychenbergstrasse 140 8400 Winterthur |

Inhaltsverzeichnis

[1 Einleitung 1](#_Toc123304675)

[2 Die Spiele-Engine Unity 2](#_Toc123304676)

[2.1 Frameworks 2](#_Toc123304677)

[2.2 Programmiersprachen 2](#_Toc123304678)

[2.3 Die Unity Physik-Engine 3](#_Toc123304679)

[2.3.1 Objects and Shapes 3](#_Toc123304680)

[2.3.1.1 Circle Shapes 3](#_Toc123304681)

[2.3.1.2 Polygone Shapes 3](#_Toc123304682)

[2.3.1.3 Edge Shapes und Chain Shapes 5](#_Toc123304683)

[2.3.2 Das Dynamic Tree Modul 7](#_Toc123304684)

[2.3.2.1 Such-Algorithmus mit Bounding Volume Hierarchy 8](#_Toc123304685)

[2.3.2.2 Erstellung der Bounding Volume Hierarchy 9](#_Toc123304686)

[2.3.3 Algorithmus zum Einfügen weiterer Objekte 10](#_Toc123304687)

[3 Die Entwicklung von «Das Spiel» 12](#_Toc123304688)

[3.1 Einarbeitung in Unity 12](#_Toc123304689)

[3.1.1 Erstellung eines Unity-Projektes für ein einfachen Spiel 13](#_Toc123304690)

[3.1.2 Programmierung der Spielfigur mit C# 15](#_Toc123304691)

[3.2 Entwicklung von Spielideen 16](#_Toc123304692)

[3.3 Entwicklung des Spiels 17](#_Toc123304693)

[3.3.1 Entwicklung der Spielewelt in Unity 17](#_Toc123304694)

[3.3.2 Entwicklung des Verhaltens der Objekte in C# 18](#_Toc123304695)

[3.3.2.1 Erkennung der Bewegungsrichtung 19](#_Toc123304696)

[3.3.2.2 Erkennung der Berührung von Wänden und Boden 20](#_Toc123304697)

[3.3.2.3 Variable Sprunghöhen 20](#_Toc123304698)

[3.3.2.4 Umsetzung von unterschiedlichen Sprüngen 21](#_Toc123304699)

[3.3.2.5 Freischaltung von Fähigkeiten 22](#_Toc123304700)

[3.3.2.6 Verwendung einer fallenden Plattform 22](#_Toc123304701)

[3.3.2.7 Air-Dashing 23](#_Toc123304702)

[3.4 Einbindung von speziellen Features 25](#_Toc123304703)

[4 «Das Spiel» 26](#_Toc123304704)

[5 Nachwort 27](#_Toc123304705)

[6 Probleme 28](#_Toc123304706)

[7 Anhang 29](#_Toc123304707)

# Einleitung

Bereits seit längerer Zeit wollte ich ein Computerspiel programmieren, um eine fortgeschrittene Programmiersprache und Entwicklungsumgebung zu erlernen. Aufgrund der häufigen Unterbrechungen in meinem Lernfortschritt wurde daraus nie mehr als eine Idee. 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 überlegte, 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. 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 hatte. 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 hatte. Deshalb entschied ich mich für Unity, da Unity einerseits verbreiteter ist und damit mehr Unterstützung in Communities erhält. Zudem werden Spiele damit einheitlich C# entwickelt und es ist damit einfacher es zu erlernen. 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 schwierig für mich die Maturaarbeit zu planen. Ich fing damit an die Grundlagen von C# zu lernen, damit ich es in Unity besser einsetzen konnte. Anschliessend nutzte ich ein Tutorial für Unity, bevor ich schliesslich mit der Entwicklung des Spieles beginnen konnte.

# Die Spiele-Engine Unity

Unity wurde von Unity Technologies am 8. Juni 2005 veröffentlicht und ist eine Spiele-Engine und ein Framework welche 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, welche 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.

## 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, welche 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].

## Programmiersprachen

## Die Unity Physik-Engine

Die Physik-Engine Box2D umfasst Module, welche 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.

### Objects and Shapes

Die einfachsten Objekte sind leere Objekte, sie haben nur eine Position. Man kann ihnen zusätzlich eine Grafik geben welche man als Sprite hinzufügt und Kollision, welche mit den unterschiedlichen Shapes der Box2D Physik Engine berechnet wird. Wenn ein Objekt sich bewegen sollte und von der Gravitation beeinflusst werden soll, muss ihm ein Box2D Rigidbody gegeben werden.

Polygone und Kreise sind gefüllt, das heisst, dass wenn ein anderes Objekt innerhalb des Colliders ist, wird es an die nächste Position ausserhalb gesetzt, während Edges nur Kanten zur Abgrenzung sind, an denen Objekte kollidieren können.

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

#### Circle Shapes

Kreise haben eine Position und einen Radius, Kreise sind immer gefüllt [1].

#### 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 Schicht welche die Polygone voneinander getrennt hält. Diese Schicht hilft bei «Continuous Collission» und verhindert, dass die Polygone sich überlappen. Durch die Haut entstehen kleine Abstände zwischen den Formen [1].

Bei der «Continuous Collission» wird nicht wie bei den anderen Möglichkeiten überprüft, ob die Collider der Objekte sich berühren sondern es wird der erste Punkt der Kollision berechnet. Dadurch wird verhindert das sich schnell bewegende Objekte durch andere Objekte zwischen den Überprüfungen hindurch fallen [2].

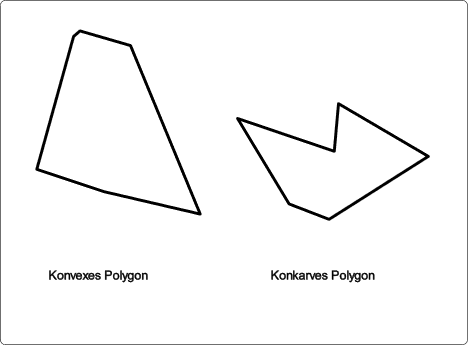
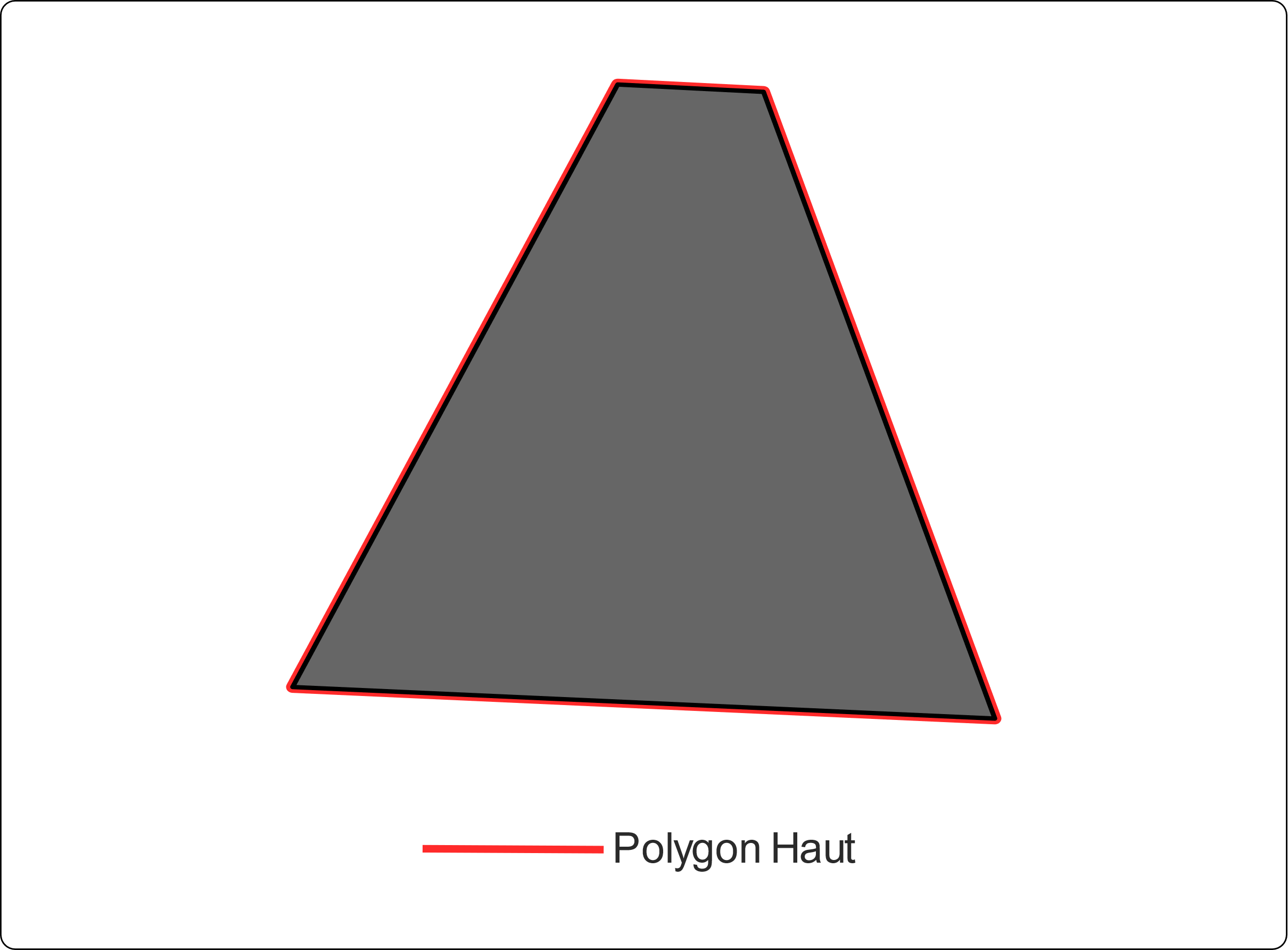
 

Abbildung 1: Polygone Shapes mit Haut

Ein Bild, das Kette, Metallwaren enthält.

Automatisch generierte Beschreibung

Abbildung 2: Polygon Haut in Unity

#### Edge Shapes und Chain Shapes

Edge Shapes sind Formen, welche mit anderen Objekten, welche keine Edge Shapes sind, zusammenstossen können. Edge Shapes werden nur sehr selten einzeln verwendet, sie werden meistes zu Chain Shapes aneinandergekettet [1].

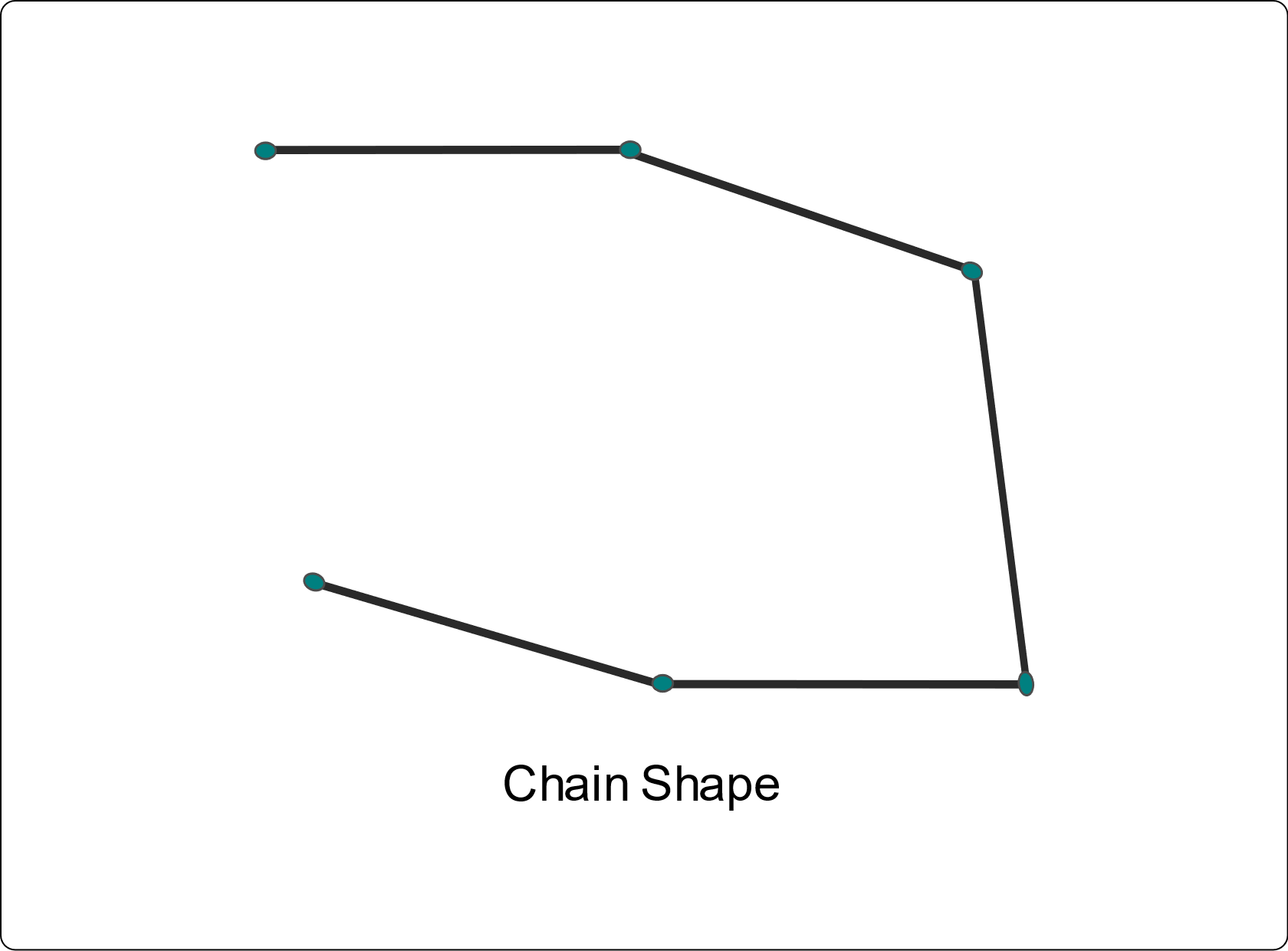


Abbildung 3: Chain Shape bestehend aus mehreren Edge Shapes

Die Aneinanderreihung von mehreren Edge Shapes führt zu Zusammenstössen mit den Verknüpfungspunkten, sogenannte «Ghost Collisions». Diese werden von Box2D behoben, aber der Algorithmus funktioniert nur auf einer Seite der Linie [1].

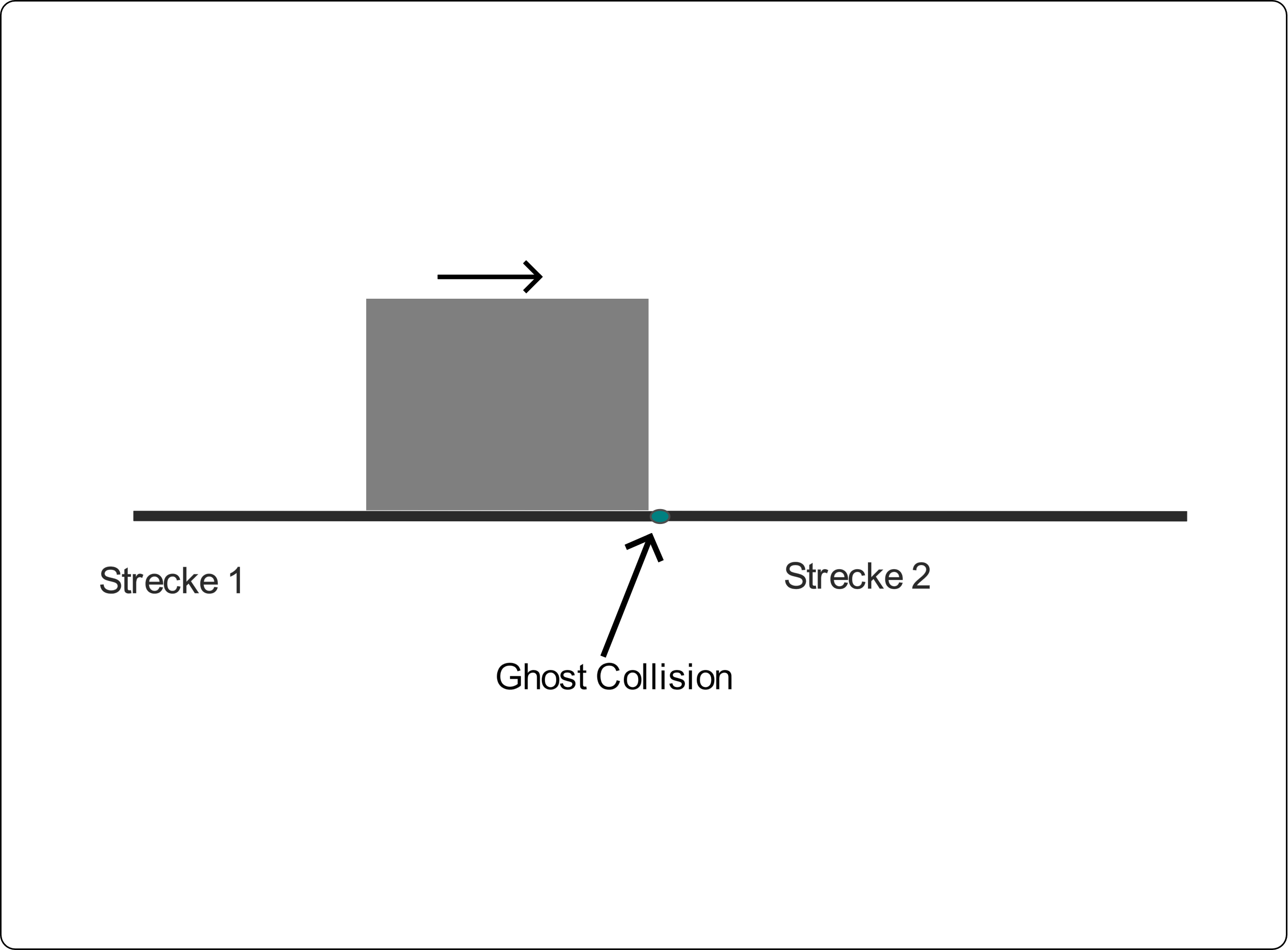


Abbildung 4: Ghost Collission

Chain Shapes haben nur einseitige Kollisionen. Diese einseitige Kollisionen schützen vor Ghost Collisions. Sie werden vor allem für statische Objekte für die Spielewelt verwendet [1].

Chain Shapes werden in Unity verwendet, wenn man für die genauere Bearbeitung des Colliders die Composite Collider Komponente verwendet und in diesem einstellt das die Kollision nur für den Umriss der Form berechnet werden soll [3]. Der Collider wird als hellgrüne Linie dargestellt.

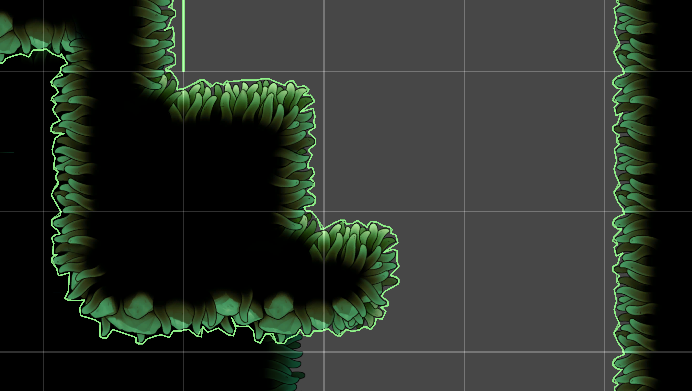


Abbildung 5: Chain Shape in Unity

### 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 Finden 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.

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

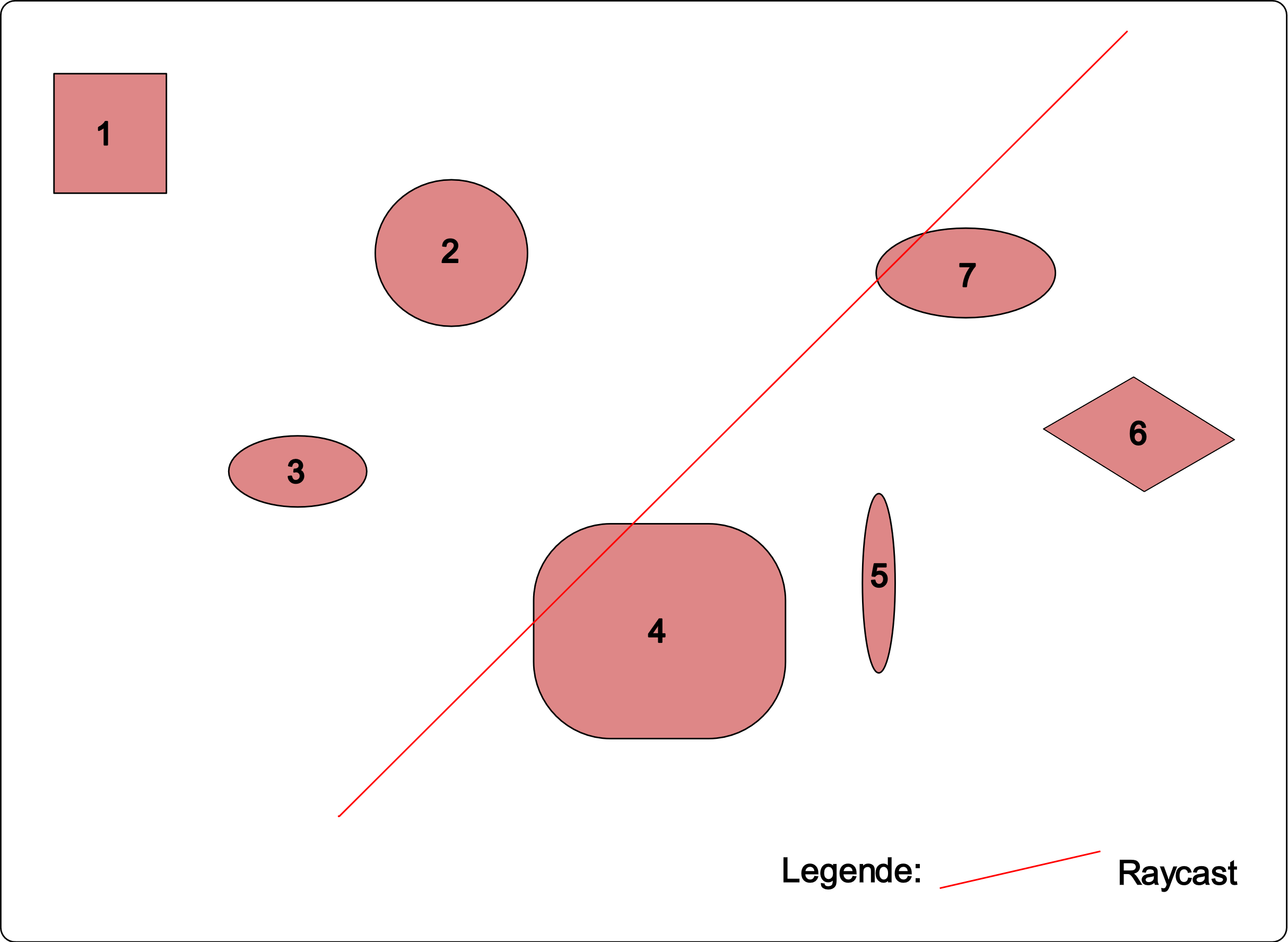


Abbildung 6: Mehrere Objekte und ein Raycast

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 [4].

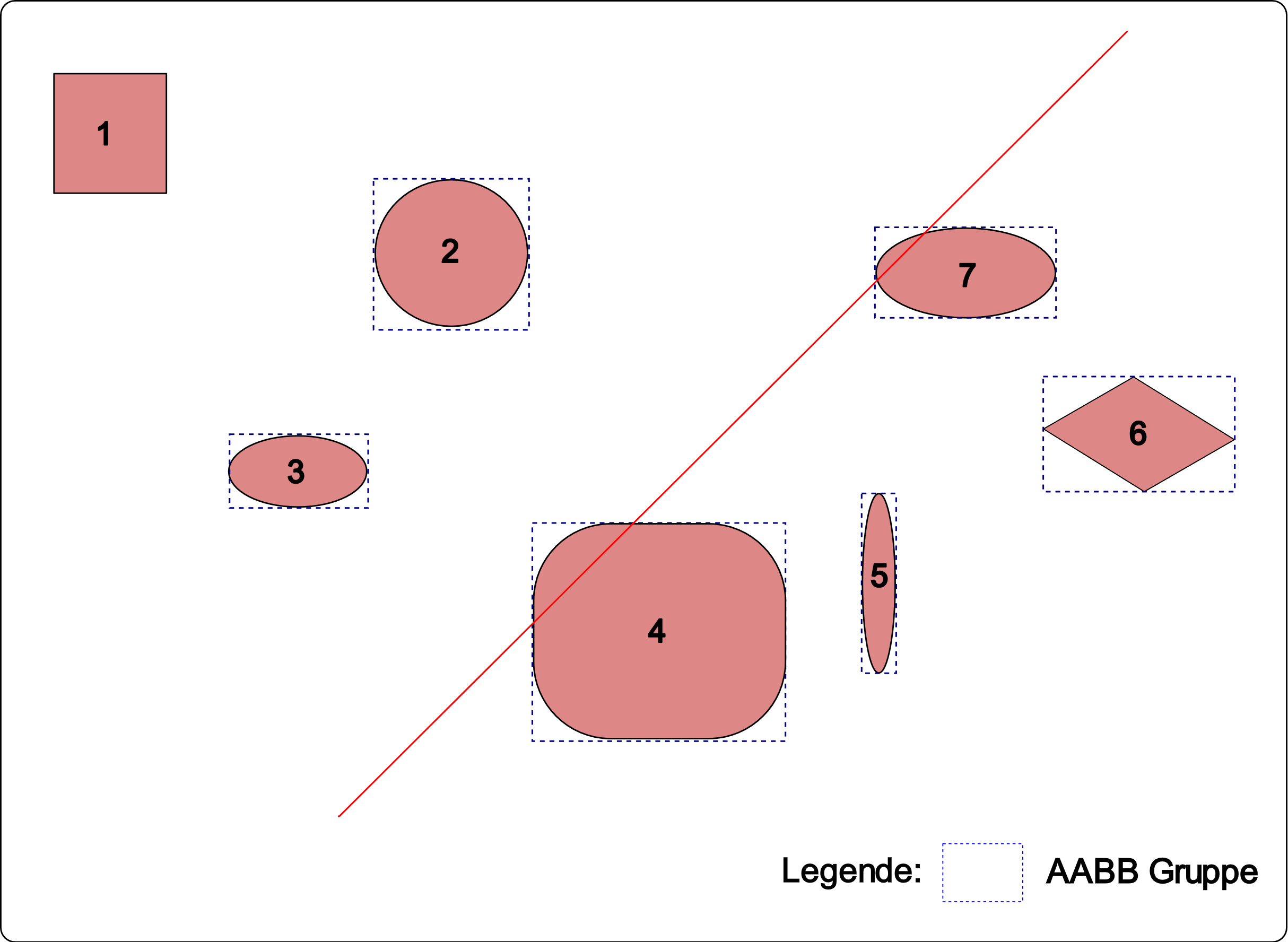


Abbildung 7: Mehrere Objekte, deren AABBs und ein Raycast

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. Wenn ein Raycast eine AABB-Gruppe trifft, wird die Berechnung innerhalb dieser Gruppe fortgeführt, wodurch insgesamt weniger Berechnungen benötigt werden [4].

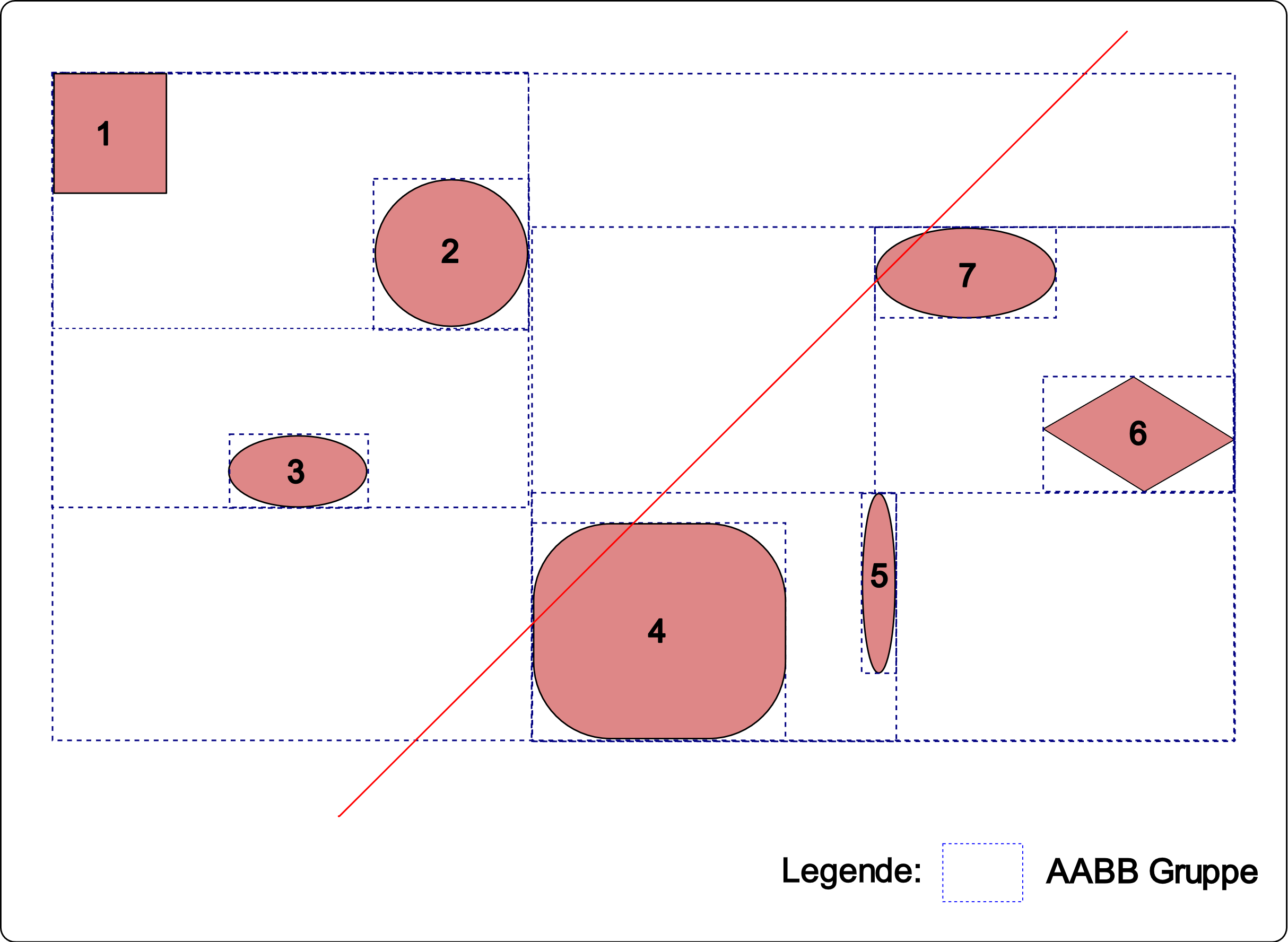


Abbildung 8: AABB-Gruppen der Objekte

Für die Erstellung der AABB-Gruppen werden die AABBs von jeweils zwei Objekten in ein neues AABB zusammengefasst. 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 [5].

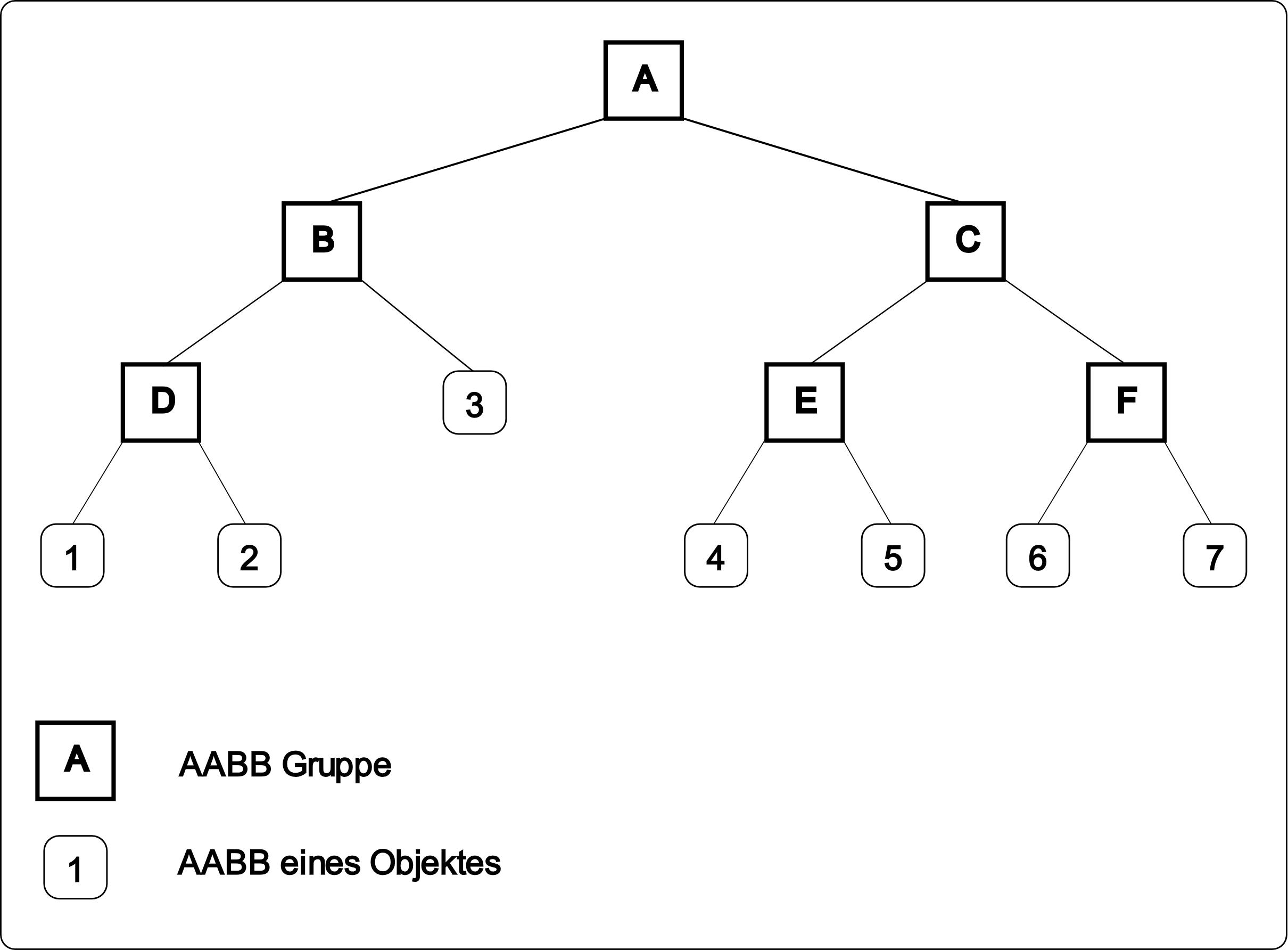


Abbildung 9: Bounding Volumen Hierarchy der AABB-Gruppen

#### 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 durchgeführt [4].

#### Erstellung der Bounding Volume Hierarchy

Es gibt drei Varianten [5] für die Erstellung der Bounding Volume Hierarchy:

* Bottom Up
* Top Down
* Incremental

Bei dem Bottom Up Vorgehen werden die Objekte mit dem jeweiligen Nachbar-Objekt in eine Gruppe zusammengefasst. Danach wird dieser 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].

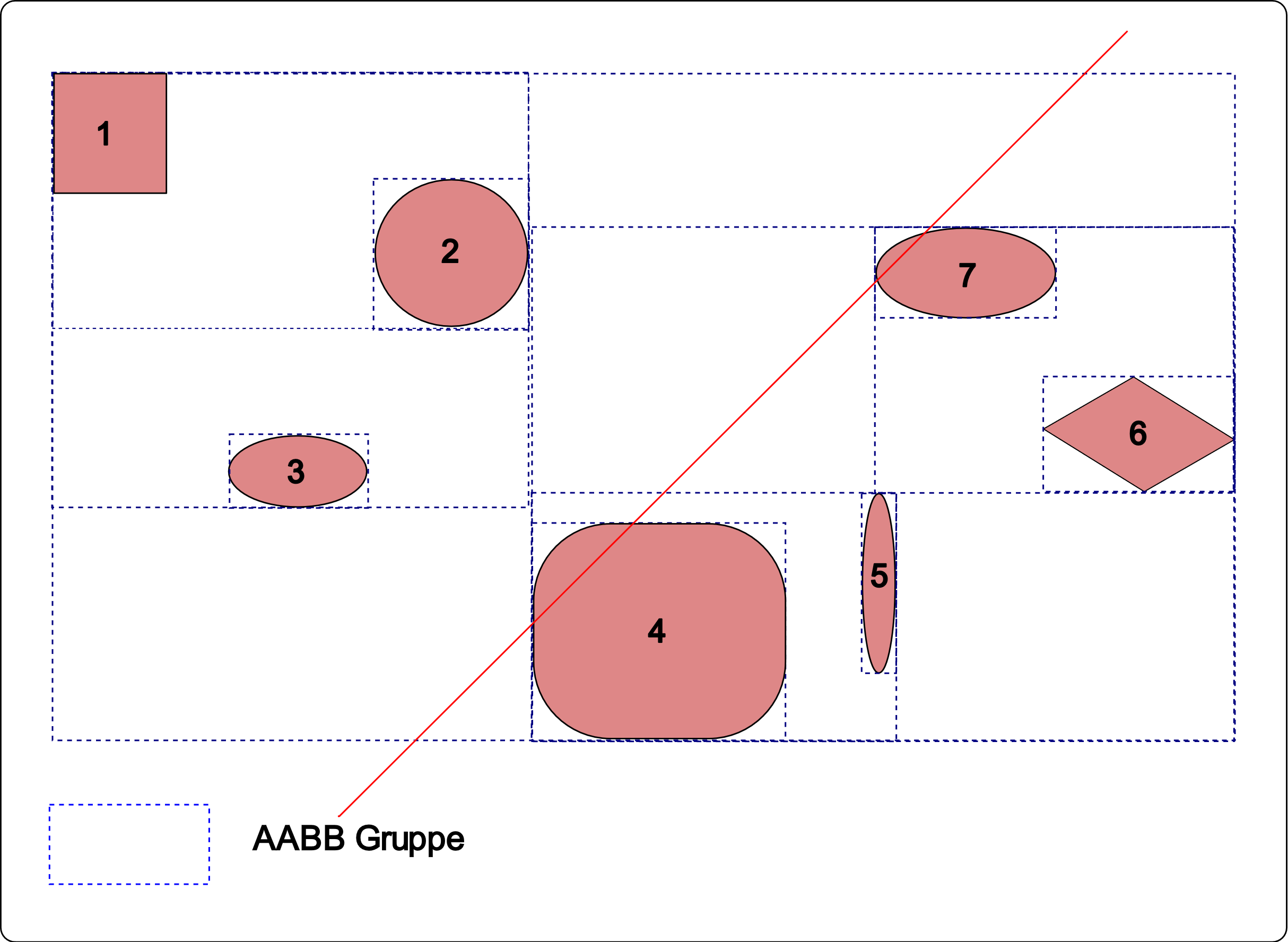
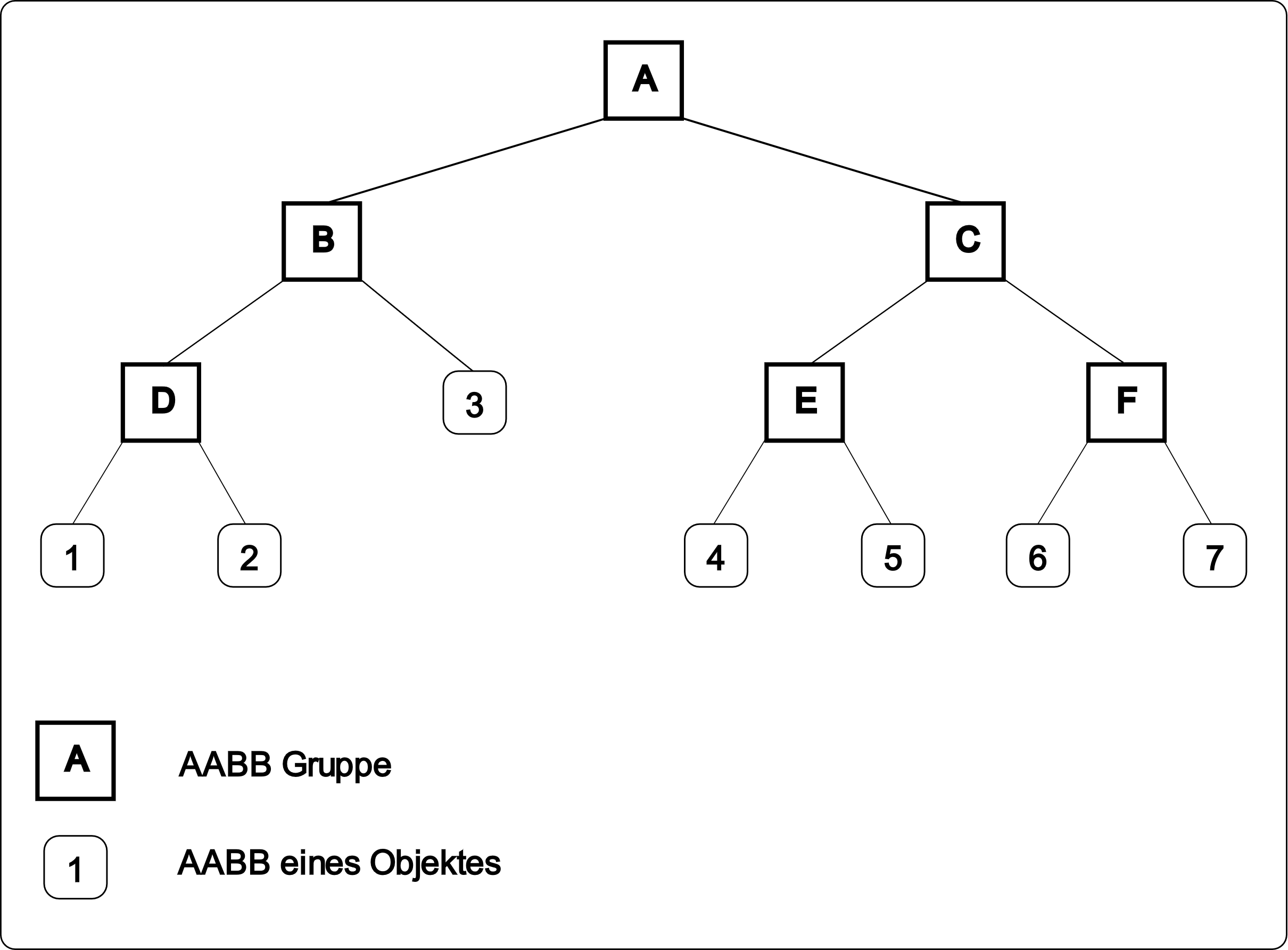
Bei dem Top Down Verfahren wird zuerst die übergreifende AABB erstellt, die alle Objekte beinhaltet. Danach wird die übergreifende Gruppe in zwei Gruppen geteilt. Für die Teilung gibt es zwei Möglichkeiten. Die AABB-Gruppe wird entweder in der Mitte aufgeteilt oder es werden die zwei AABBs mit der geringsten 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 [4].

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 [7].

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.

### 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 [7].

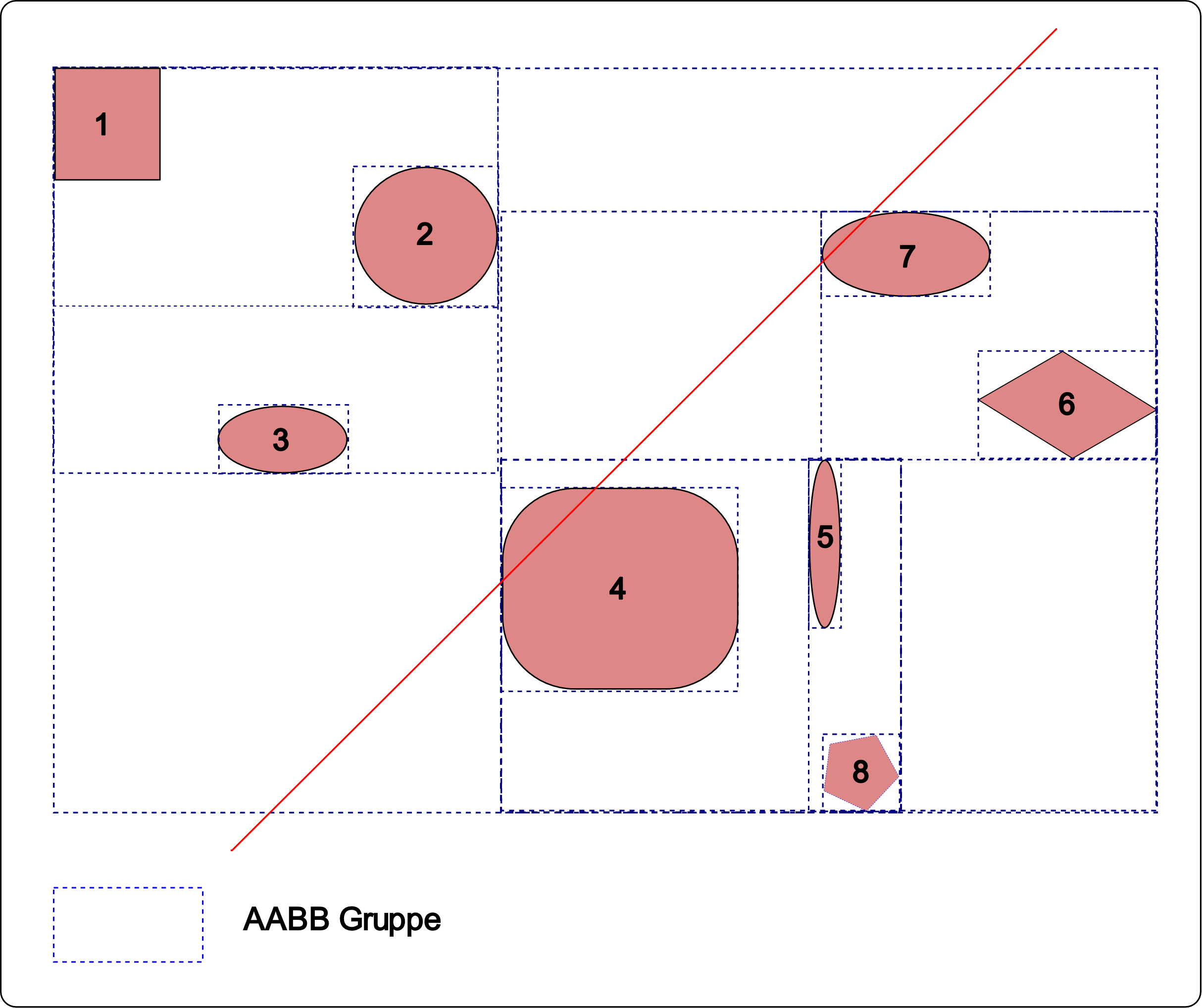
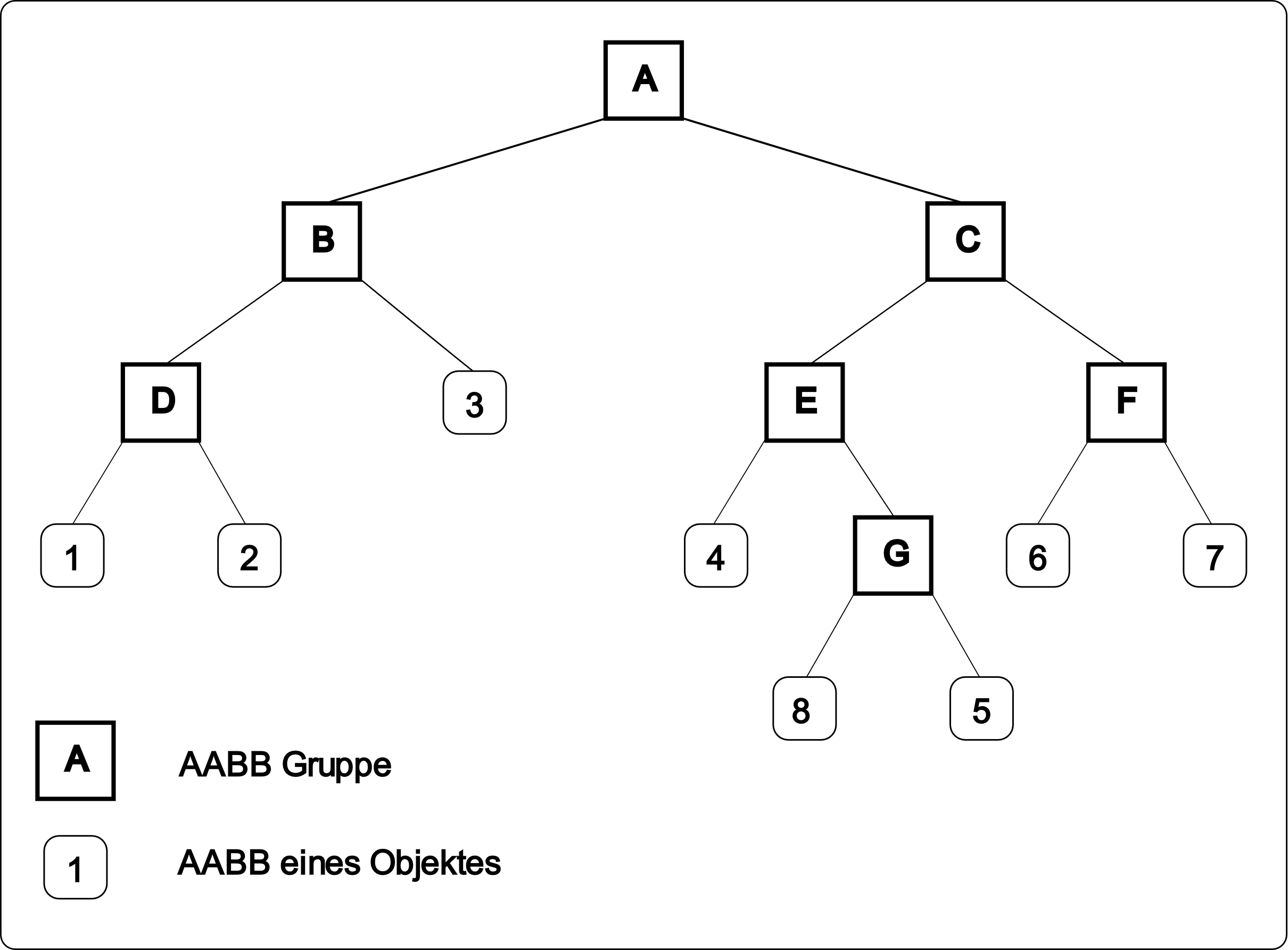
 

Abbildung 10: Grafik vor und nach dem Einfügen eines neuen Objektes

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 [7].

Kostenfunktion einer Bounding Volume Hierarchy:

C(T) =

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 11: 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 [7].

Im oben genannten Beispiel würde sich folgende Formel ergeben:

# Die Entwicklung von «Das Spiel»

## Einarbeitung in Unity

Bevor ich anfing Unity auszuprobieren habe ich die Grundlagen zu C# auf w3schools gelernt [8].Nach der Installation von Unity und Visual Studio programmierte ich eine erste Version des Spiels mit Hilfe des Tutorials von Pandemonium [9]. Zuerst programmierte ich die Steuerung des Spielers, danach Animationen, Hindernisse, die Lebenspunkte des Spielers und Checkpoints. Der Vorteil dieses 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 sich davon nur wenig profitieren, da die benötigten Befehle Unity spezifisch sind. Das ist eine allgemeine Eigenschaft von Frameworks, das heisst auch wenn die Programmiersprache bereits bekannt ist, braucht man eine Einarbeitung in das jeweilige Framework.

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 auch grösstenteils ohne Hilfe meine Ideen programmieren.

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

Für neue 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 [Quelle] 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.

### Erstellung eines Unity-Projektes für ein einfachen 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.

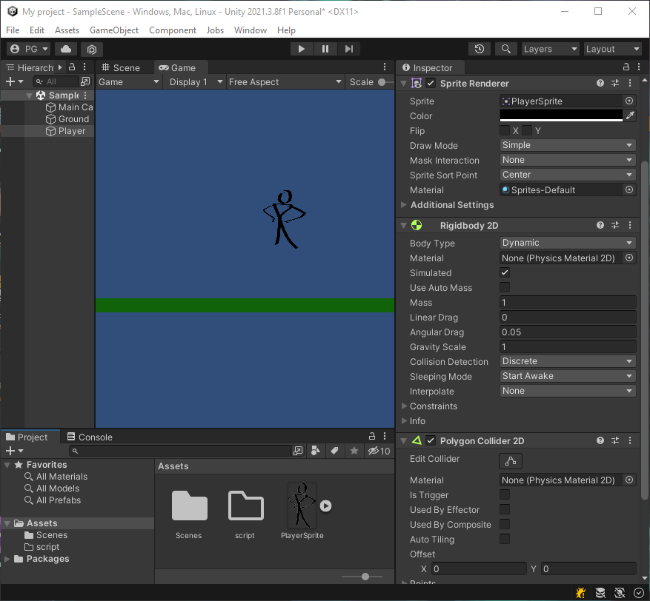
Ein Bild, das Text, Monitor, Screenshot enthält.

Automatisch generierte Beschreibung

Abbildung 12: 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, 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.

 Ein Bild, das Text, Monitor, Screenshot, schwarz enthält.

Automatisch generierte Beschreibung

Abbildung 13: Physik-Engine des einfaches 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.

### Programmierung der Spielfigur mit C#

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

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

Das folgende C#-Skript zeigt die wenigen Anpassungen die notwendig sind, um das Player Objekt 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);

    }

}

## 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.

## Entwicklung des Spiels

In diesem Kapitel wird die Entwicklung des Spiels erklärt, das wie bereits im Kapitel 3.1 Einarbeitung in Unity beschrieben wurde aus der Entwicklung der Spielewelt in Unity und den passenden C#-Skripten besteht.

### Entwicklung der Spielewelt in Unity

#### Statische Objekte der Spielewelt

Bei der Erstellung der Spielewelt war die Suche nach den passenden Grafiken recht aufwändig. 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 [10] sehr hilfreich.

Meine Spielewelt basiert auf zwei Tilemaps, die als Gitter ersichtlich sind in dem die Grafiken platziert werden können. Eine 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, die jedes Objekt besitzt, werden Grösse, Position und Rotation gespeichert. Die Eigenschaften der Tilemap und des Tilemap Renderer steuern die Darstellung des Objektes.

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 RidigBody2D-Komponenten der Objekte in dieser Tilemap sind auf statisch eingestellt, da sie sich nicht bewegen und deshalb die Kollisionen nicht so häufig wie bei einem beweglichen Objekt berechnet werden müssen. 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.

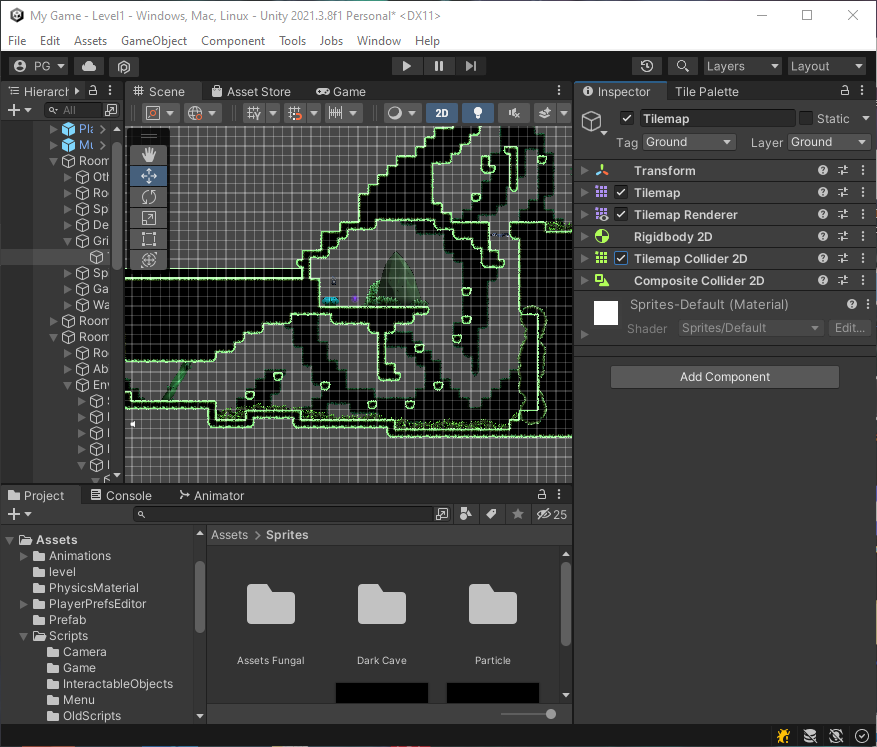


Abbildung 14: Statische Objekte der Spielewelt

#### Bewegliche Objekte der Spielewelt

Die Spielfigur benötigt einige Komponenten, welche 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.

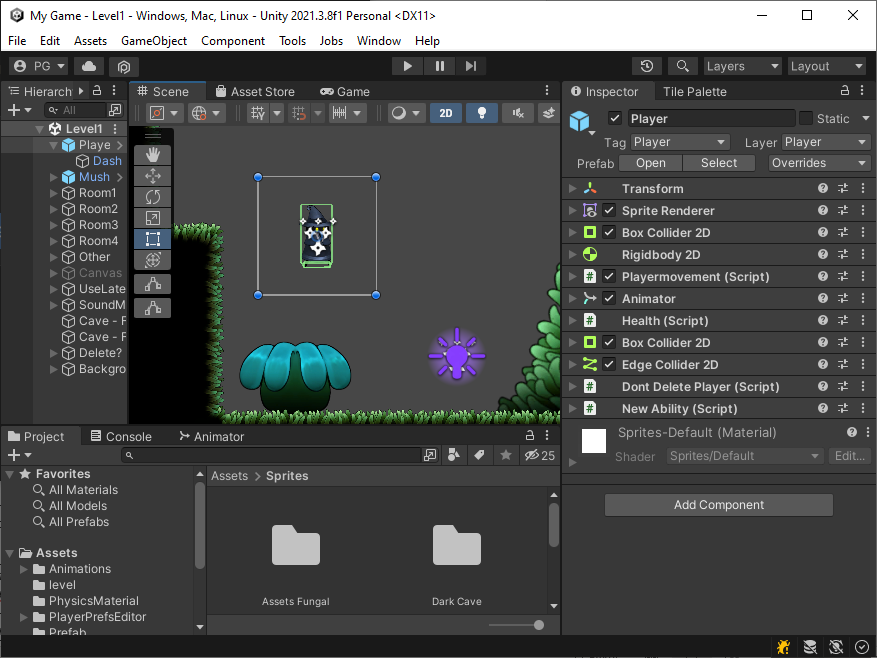


Abbildung 15: Bewegliche Objekte der Spielewelt

Damit die Spielfigur sich bewegen kann, ist ihm das *Playermovent* 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.

### Entwicklung des Verhaltens der Objekte in C#

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

#### Erkennung der Bewegungsrichtung

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

#### 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 Tag 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;

}

}

#### 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 langsamer an der Wand herunter rutscht.

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;

}

#### 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 eingeführt, 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.

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));

}

#### 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 die Fähigkeit freischalten. In den *if* Statements wird der Tag des Objekts, welches die Kollision verursacht hat mit dem Tag 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);

}

}

}

#### Verwendung einer fallenden Plattform

Fallende Plattformen haben die selbe Gravitation wie die Spielfigur, ihre Bewegung wird beim Spielstart jedoch deaktiviert. Sobald die Spielfigur die fallende Plattform berührt wird die Bewegung nach einer Sekunde aktiviert. Nach 10 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;

}

#### 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 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;

}

### Einbindung von weiteren Features

#### Spielemenü

<TODO>

#### 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 ein Audio-Listener in der Szene zuweisen.

#### Game Controler

<TODO bessere Erklärung>

Bei der Controller Einbindung musste ich mich zuerst zwischen zwei Erweiterungen entscheiden wobei die neuere komplizierter, aber auch besser ist, leider hatte diese weniger Anleitungen weshalb ich anfangs ein bisschen herumprobieren musste, aber schlussendlich hat es funktioniert.

### Probleme bei der Entwicklung

Kein Prototyp weshalb ich ohne konkretere Idee der Mechaniken angefangen habe das Spiel zu programmieren

Als ich dann angefangen habe gezielter über die Mechaniken nachzudenken wurde mir schnell klar dass ich das die meisten meiner Ideen sehr stark vor anderen Spielen der gleichen Art inspiriert waren

Angangs setzte ich meinen Fokus auch zu stark auf die Grafik was in zwar hilfreich war weil das Spiel endlich nach einem spiel aussah und nicht mehr nach schwarzen und grauen Rechtecken. Das war der gefühlt erste richtige Fortschritt zu einem spiel aber dadurch wurde ich wiederum von den Mechaniken abgelenkt.

Der schwerste Teil der ganzen Maturaarbeit war nicht wie erwartet der Informatik Anteil sondern das Gamedesign. Während ich immer noch die längste Zeit damit verbrachte die neuen Mechaniken zu programmieren, hatte ich in der zwischen Zeit keine neuen Ideen welche für mein programmierkönnen und meine grafischen Möglichkeiten realistisch waren.

Gleichzeitig habe ich das Gefühl das ich in diesem Jahr zwar sehr viel über Unity und die Funktionen verstanden habe aber trotzdem nicht viel mehr als die Basics gelernt habe. Als kompletter Anfänger in c#, Unity und im Gamedesign hatte die anfängliche Fehleinschätzung der Schwierigkeiten doch stärkere folgen gleichzeitig verbringe ich auch wenn ich schon eine Idee für einen Raum habe und alle Mechaniken schon programmiert sind immer noch eine Stunde bis alles funktioniert wie ich es geplant habe, je nach raum sogar länger.

Lieblingsfehler ; oder 2d vergessen

# «Das Spiel»

Das Ziel des Spieles ist es mit unterschiedlichen Bewegung Mechaniken wie normalem laufen, Sprüngen, wand Sprüngen, Doppelsprüngen und Dashs die Hindernisse des Spielkarte zu überwinden und das Ende des Spieles zu erreichen. Während die einfachsten Hürden, Sprünge von Plattform zu Plattform sind, werden diese im Verlauf des Spieles durch Plattformen, welche bei kontakt herunterfallen oder Stalaktiten welche versuchen den Spieler zu erschlagen erschwert.

<TODO>Grafiken aus Spiel

Die meisten Fähigkeiten des Spielers sind anfangs noch nicht nutzbar sie werden aber im Verlauf des Spiels durch das aufsammeln von Objekten freigeschaltet.

Das Ziel eines Plattformers ist es mit unterschiedlichen Sprüngen und Bewegungsmechaniken durch die unterschiedlichen Hindernisse der Spielewelt zu überwinden und das Ende des Spiels zu erreichen. Als erstes fügte ich Laufen und Springen dem Spiel hinzu. Später gab ich dem Spieler auch die Möglichkeit an Wänden zu klettern und von diesen abzuspringen, mehrfach in der Luft zu springen und eine schnelle Bewegung in eine bestimmte vom Spieler ausgewählte Richtung zu machen. Um dem Spiel schwieriger und gleichzeitig auch einsteigerfreundlicher zu machen, habe ich dem Spieler nicht alle Fähigkeiten von Anfang an gegeben, sondern man kann diese erst ab einem bestimmten Punkt im Verlauf des Spieles freigeschalten. In der Spielewelt platzierte ich auch Objekte, mit welchen der Spieler interagieren kann wie fallende Stalaktiten und Plattformen welche, wenn der Spieler diese Berührt zusammenbrechen, Plattformen die nur in eine Richtung durchquerbar sind, Stacheln die den Spieler Töten und einen Rollenden Stein, welcher das Level hinter dem Spieler zerstört. Später fügte ich auch ein Hauptmenü mit der Möglichkeit Einstellungen an der Grafik und Lautstärke des Spiels anzupassen und zuvor gespeicherte Spielstände zu laden.

# Nachwort

# Quellenverzeichnis

|  |  |
| --- | --- |
| [1] | Catto Erin, «Box 2D Manual,» Box2D, 2021. [Online]. Available: https://box2d.org/documentation/. [Zugriff am 30 10 2022]. |
| [2] | Unity Technologies, «Unity Documentation,» 2 12 2022. [Online]. Available: https://docs.unity3d.com/Manual/unity-architecture.html. |
| [3] | MelvMay, «Unity Technoligies,» 1 7 2020. [Online]. Available: https://forum.unity.com/threads/composite-collider-2d-ghost-collision.923417/. [Zugriff am 29 12 2022]. |
| [4] | 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. |
| [5] | J. Braun, *BVH- und Line-Space-Kombination zur Pathtracing-Beschleunigung,* Koblenz, 2017. |
| [6] | S. M. Omohundro, «Five Balltree Construction Algorithms,» International Computer Science Institute, California, 1989. |
| [7] | J. Bittner, M. Hapala und V. Havran, «Incremental BVH construction for ray tracing,» *Computers & Graphics,* Dezember 2014. |
| [8] | w3schools, «w3schools,» w3schools, [Online]. Available: https://www.w3schools.com/. [Zugriff am 5 5 2022]. |
| [9] | Pandemonium, «YouTube,» [Online]. Available: https://www.youtube.com/watch?v=TcranVQUQ5U&list=PLgOEwFbvGm5o8hayFB6skAfa8Z-mw4dPV&ab\_channel=Pandemonium. [Zugriff am 2022 12 29]. |
| [10] | Maoot, «itch.io,» [Online]. Available: https://maaot.itch.io/. [Zugriff am 25 7 2022]. |
| [11] | S. Luber, «Cloudcomputing Insider,» 5 4 2022. [Online]. Available: https://www.cloudcomputing-insider.de/was-ist-ein-framework-a-1104630/#:~:text=Ein%20Framework%20ist%20ein%20Rahmenwerk,unterstützt%20objekt-%20und%20komponentenorientierte%20Entwicklungsansätze.. [Zugriff am 11 12 2022]. |
| [12] | Unity Technologies, «Unity,» 2022. [Online]. Available: https://unity.com/. |

# Anhang

Abb. 1: Munkel (2003: 17)

Abb. 2: Anschlagbrett Aldi Freiburg i.Br., aufgenommen am [Datum]

(Belege, seien es transkribierte und anonymisierte Interviews, Kindertexte, Zeitungsausschnitte usw.; Bildnachweise o.Ä.; falls nicht nötig, einfach löschen)

Hinweis: Überprüfen Sie bei aus dem Internet runtergeladenen Bildern, ob sie zur Wiederverwendung frei gegeben sind!