Entwicklung eines Computerspiels mit Unity

Maturaarbeit

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

Inhaltsverzeichnis

[1 Einleitung 1](#_Toc122969072)

[2 Die Spiele-Engine Unity 2](#_Toc122969073)

[2.1 Frameworks 2](#_Toc122969074)

[2.2 Programmiersprachen 2](#_Toc122969075)

[2.3 Die Unity Physik-Engine 3](#_Toc122969076)

[2.3.1 Objects and Shapes 3](#_Toc122969077)

[2.3.1.1 Circle Shapes 3](#_Toc122969078)

[2.3.1.2 Polygone Shapes 3](#_Toc122969079)

[2.3.1.3 Edge Shapes 4](#_Toc122969080)

[2.3.1.4 Chain Shapes 5](#_Toc122969081)

[2.3.2 Das Dynamic Tree Modul 6](#_Toc122969082)

[2.3.2.1 Such-Algorithmus mit Bounding Volume Hierarchy 7](#_Toc122969083)

[2.3.2.2 Erstellung der Bounding Volume Hierarchy 8](#_Toc122969084)

[2.3.3 Algorithmus zum Einfügen weiterer Objekte 9](#_Toc122969085)

[3 Die Entwicklung von «Das Spiel» 11](#_Toc122969086)

[3.1 Einarbeitung in Unity 11](#_Toc122969087)

[3.2 Entwicklung von Spielideen 12](#_Toc122969088)

[3.3 Entwicklung des Spiels 13](#_Toc122969089)

[3.3.1 Freischaltung von Fähigkeiten 13](#_Toc122969090)

[3.3.2 Verwendung einer fallenden Plattform 13](#_Toc122969091)

[3.3.3 Umsetzung von unterschiedlichen Sprüngen 14](#_Toc122969092)

[3.3.4 Erkennung der Bewegungsrichtung 15](#_Toc122969093)

[3.3.5 Zurücksetzen von Spr 15](#_Toc122969094)

[3.3.6 Erkennung der Berührung von Wänden und Boden 16](#_Toc122969095)

[3.3.7 Variable Sprunghöhen 17](#_Toc122969096)

[3.4 Einbindung von speziellen Features 19](#_Toc122969097)

[4 «Das Spiel» 20](#_Toc122969098)

[5 Nachwort 21](#_Toc122969099)

[6 Quellenverzeichnis 22](#_Toc122969100)

[7 Probleme 23](#_Toc122969101)

[Anhang 24](#_Toc122969102)

# Einleitung

Ich wollte schon seit längerer Zeit 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. Als ich angefangen habe nach einem Thema für meine Maturaarbeit zu suchen, überlegte ich mir zuerst Themen in Chemie, hatte auch Informatik im Hinterkopf. Schliesslich erinnerte ich mich wieder an meine vorherigen Ideen und überlegte, welche davon realistische Maturaarbeitsthemen wären. Am Ende entscheid ich mich für die Entwicklung eines Computerspiels, weil die Komplexität je nach Ansatz sehr variabel ist. Da ich vor meiner Maturaarbeit sehr wenig Programmiererfahrung hatte, war es sehr schwer den Umfang der Arbeit einzuschätzen. 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 Wahl des Themas 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 Nachteil hatte. Deshalb entschied ich mich wieder eher nach Bauch Gefühl für Unity. Ein weiterer Punkt, welcher in diese Entscheidung hineinspielte, war das ich bei meinen vorherigen versuchen die Unreal Engine zu installieren, und in dieser etwas zu programmieren, jedes Mal Probleme hatte. Jetzt im Nachhinein kann ich sagen das es eine gute Entscheidung war, weil ich den Überblick über die neue Benutzeroberfläche schnell gefunden habe und mich jetzt gut darin zurechtfinde.

Als erstes fing damit an die Grundlagen von C# lernen, um den Umfang der Maturaarbeit besser einschätzen zu können. Die grösste Schwierigkeit war, dass ich davor noch nie mit Unity, C# und Microsoft Visual Studio gearbeitet habe. Weshalb ich damit anfing die Grundlagen auszuprobieren, bis ich ein besseres Gefühl für die neue Sprache und Programme entwickelt hatte.

# 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] Sie beinhaltet einen Asset Store für Grafiken, Audio und Erweiterungen für Unity welche leicht heruntergeladen werden können. Ausserdem wird im Unity Manual jede Funktion erklärt. Unity wurde in C++ und verwendet Microsofts .NET Framework, um Anwendungen innerhalb von Unity zu programmieren wird C# verwendet. [2] Die Spiele-Engine Unity 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 von Anfang an Grundlegende Funktionen und Rahmenbedingungen, welche bei der Entwicklung der Software hilfreich sind, aber auch einschränken. Je nach Framework sind diese unterschiedlich und deswegen ist es wichtig sich das richtige Framework für das Projekt auszusuchen. Während das Framework den Rahmen und die regelmässig benötigten Funktionen zu Verfügung stellt damit diese nicht jedes von Grund auf an neu programmiert werden müssen, helfen diese nur bei der Entwicklung der Anwendung auf Basis dieser Grundlagen. [3]

## Programmiersprachen

## Die Unity Physik-Engine

Die Physik-Engine Box2D umfasst Module, für die das Verhalten der zweidimensionalen Formen implementieren. Die Physik-Engine ist sehr umfangreich, folgende Module möchte ich detaillierter beschreiben:

* Objects and Shapes
* Pairwise Functions
* Dynamic Tree

### Objects and Shapes

<TODO> Erklärung Objekt

<TODO> Erklärung Shapes, Unterscheidung gefüllte und nicht gefüllte Shapes

Polygone und Kreise sind gefüllt und haben eine Masse, 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 [4].

#### Polygone Shapes

<TODO>schicht edge radius

Polygone bestehen aus Strecken welche zwei Eckpunkte verbindet. Es gibt eine zusätzliche Unterteilung in konvexe und konkave Polygone. Um Polygone herum hat es eine dünne Schicht welche dich Polygone voneinander getrennt hält. Diese Schicht hilft bei «continuous collision» und verhindert, dass die Polygone sich überlappen. Dadurch entstehen aber kleine Abstände zwischen den Formen [4].

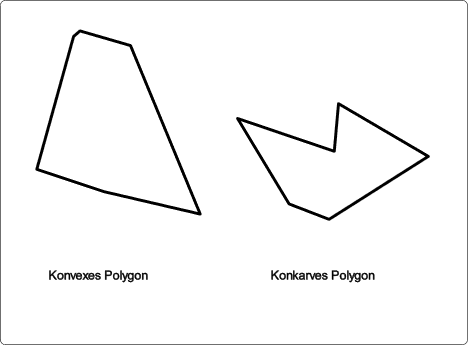


Abbildung 1: Polygone Shapes

#### Edge Shapes

Edge Shapes sind Formen, welche nur 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 [4].

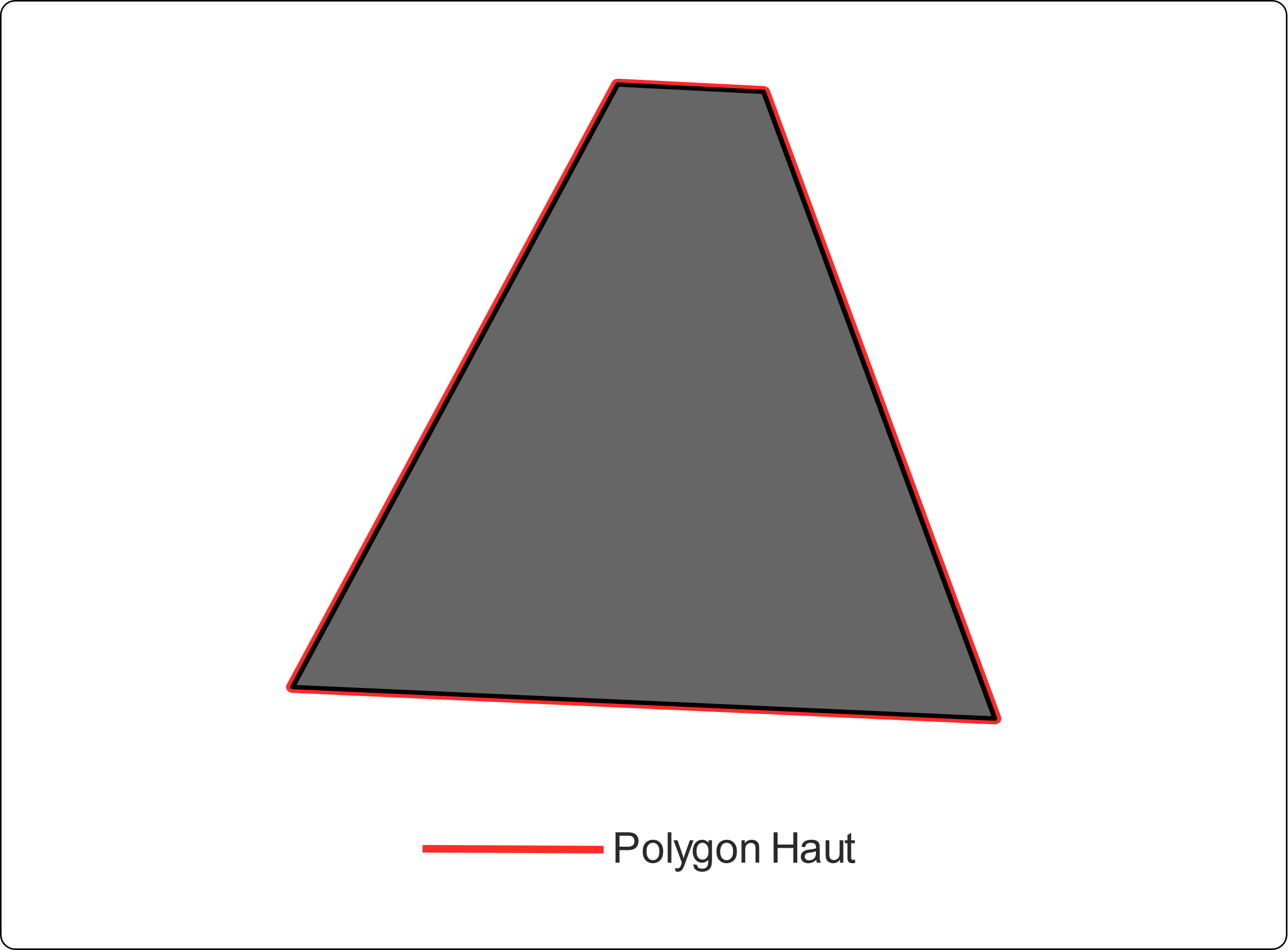


Abbildung 2: Edge Shapes

Die Aneinanderreihung von mehreren Edge Shapes führt zu Zusammenstössen mit den Eckpunkten, sogenannte «Ghost Collisions». Diese können von Box2D behoben werden aber der Algorithmus funktioniert nur auf einer Seite der Linie [4].

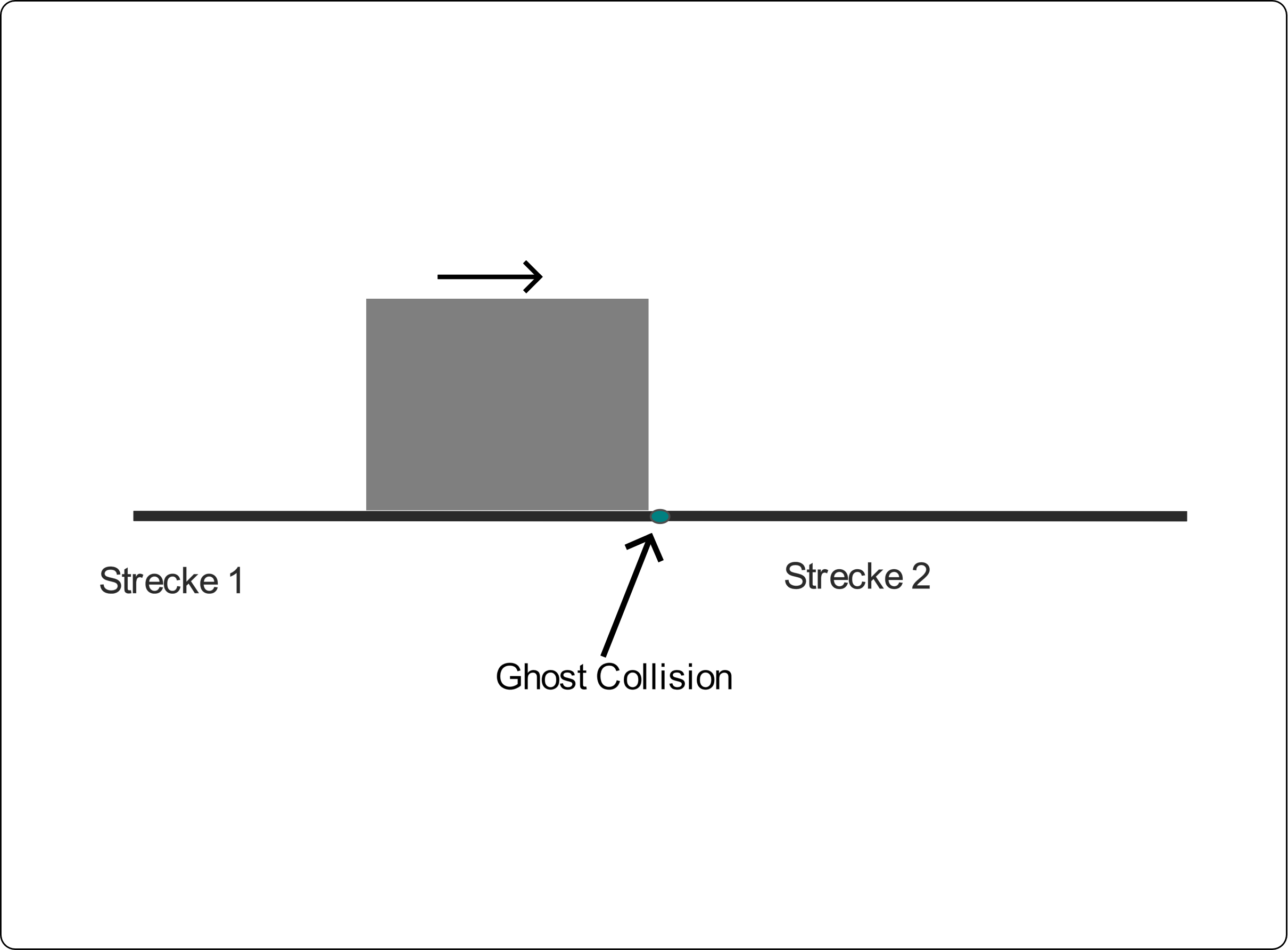


Abbildung 3: Ghost Collission

#### Chain Shapes

Mehrere Strecken werden zu einer Chain Shape aneinandergereiht. Chain Shapes haben nur einseitige Kollisionen. Diese einseitige Kollisionen schützt vor Ghost Collisions. Sie werden vor allem als statische Objekte für die Spielewelt verwendet [4].

<TODO> Beispiel für Verwendung

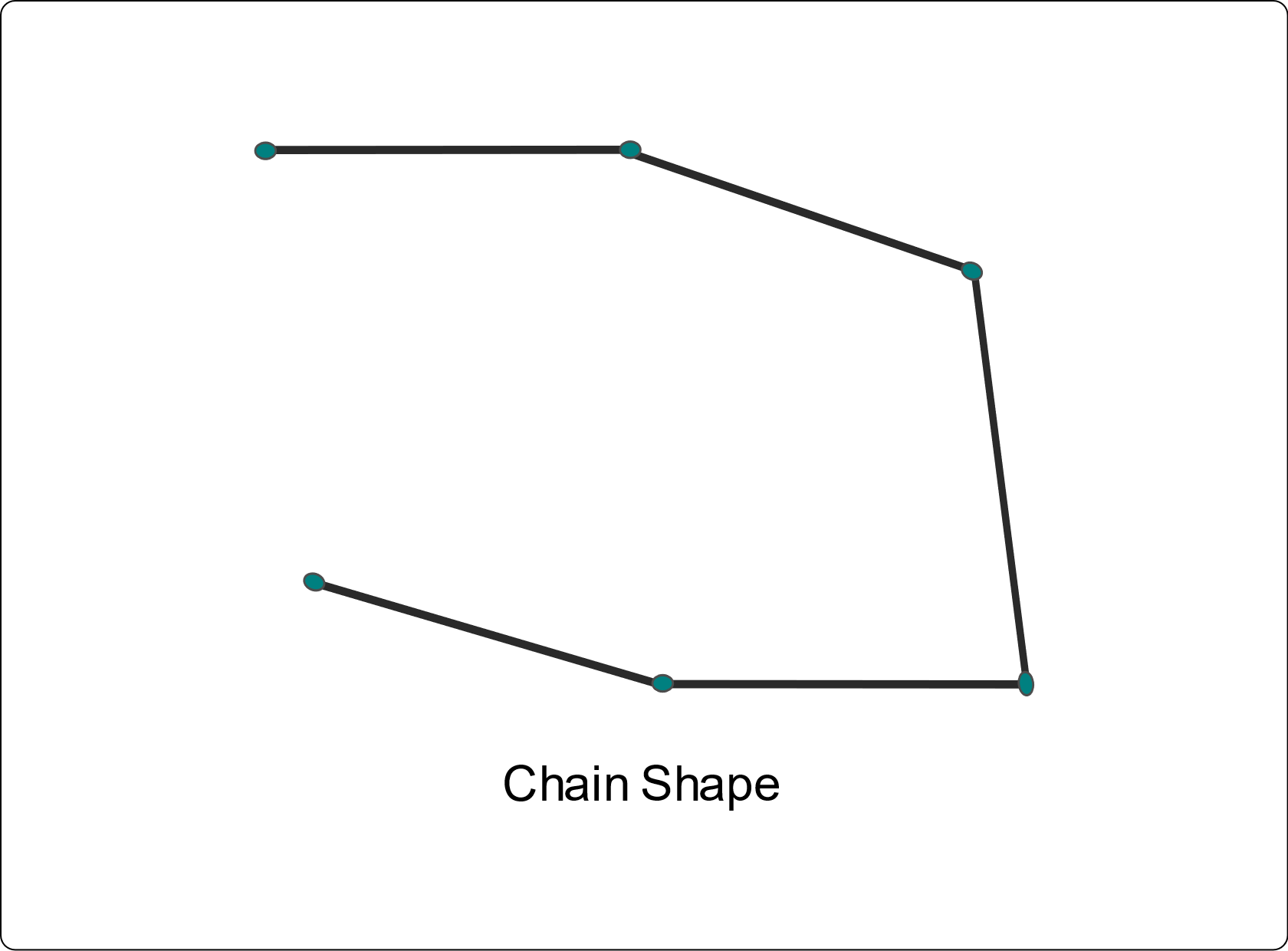


Abbildung 4: Chain Shape

### 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.

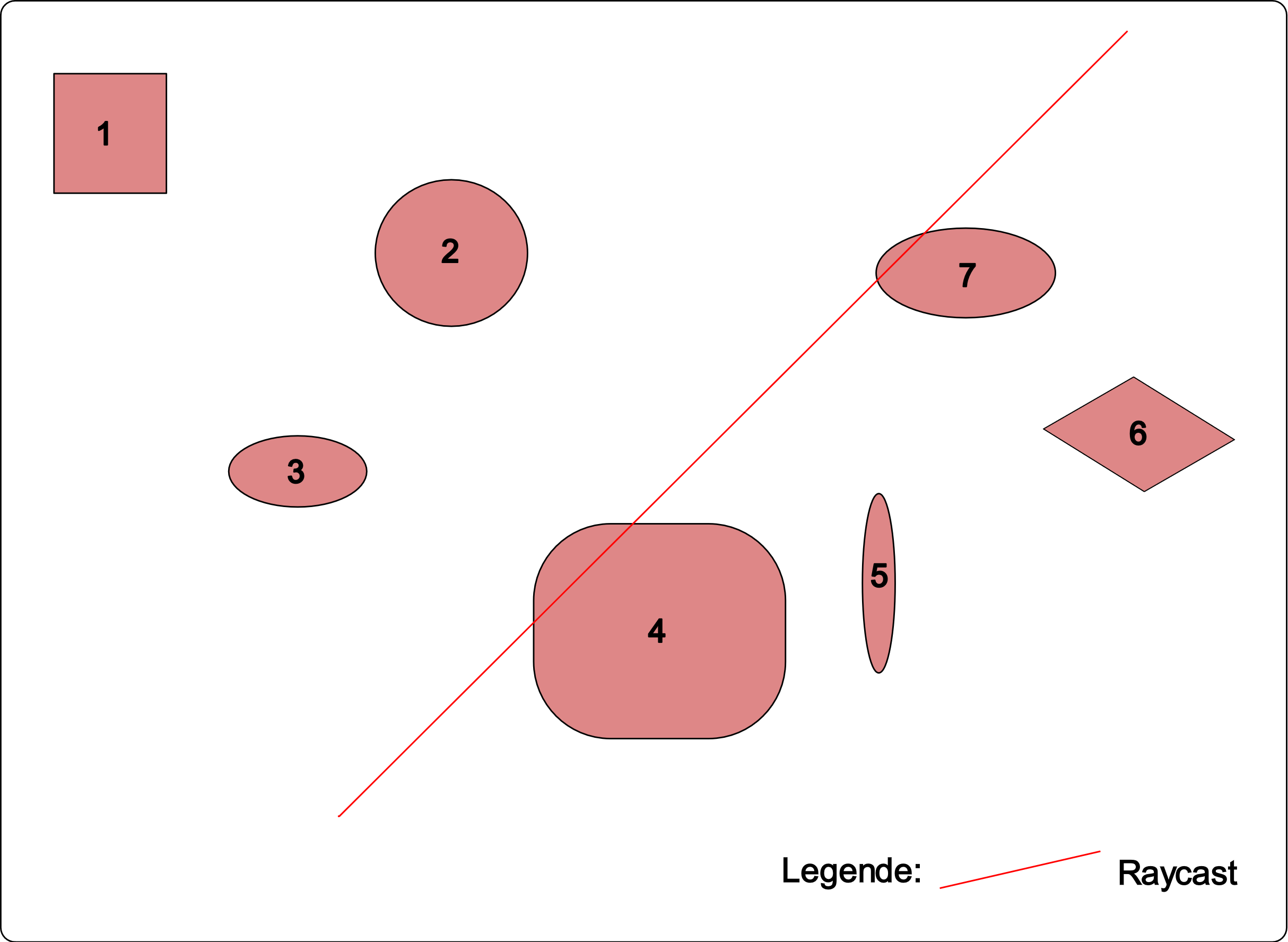


Abbildung 5: Mehrere Objekte und ein Raycast

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].

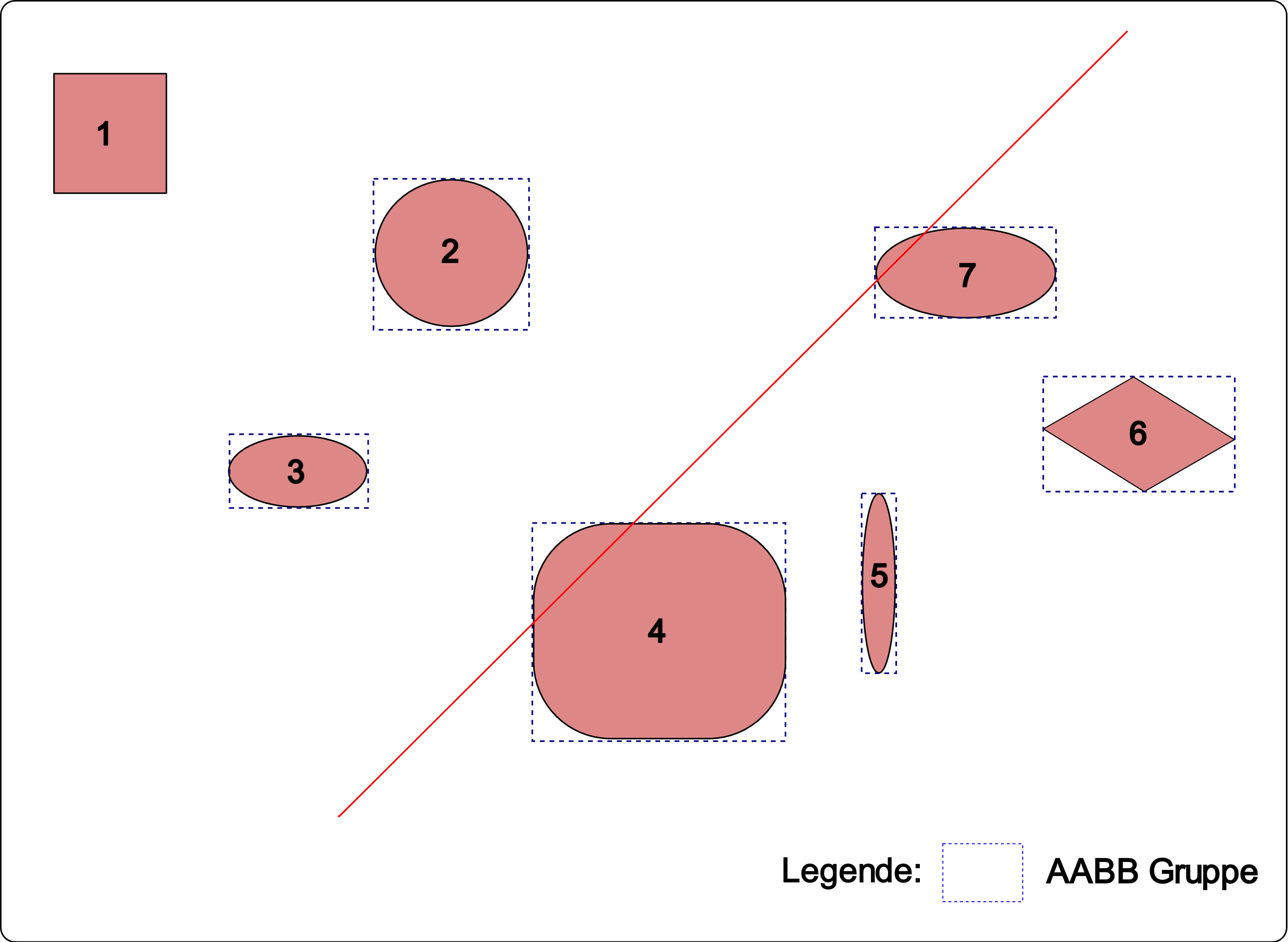


Abbildung 6: Mehrere Objekte, deren AABBs und ein Raycast

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].

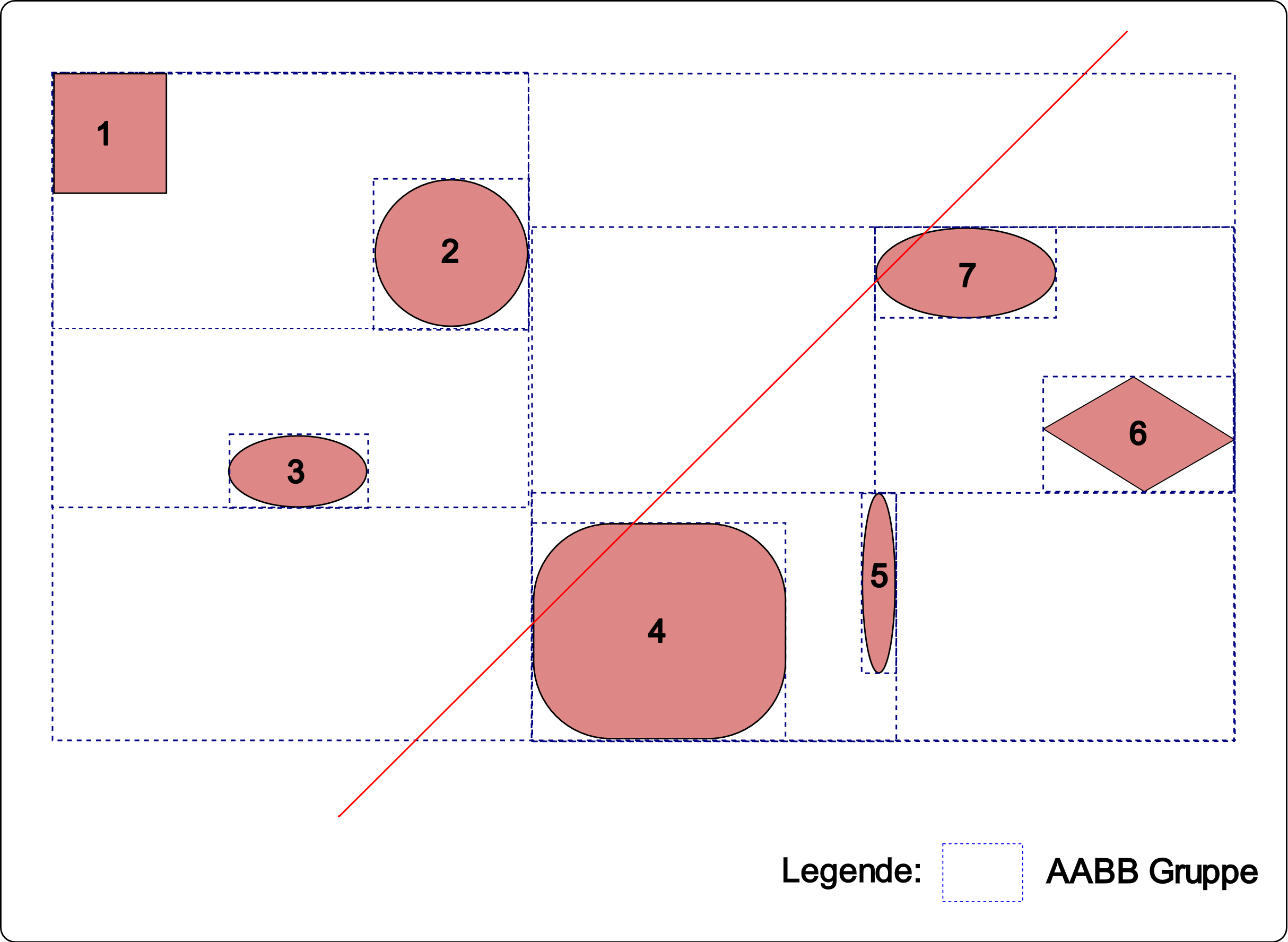


Abbildung 7: AABB-Gruppen der Objekte

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 8: 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 [4].

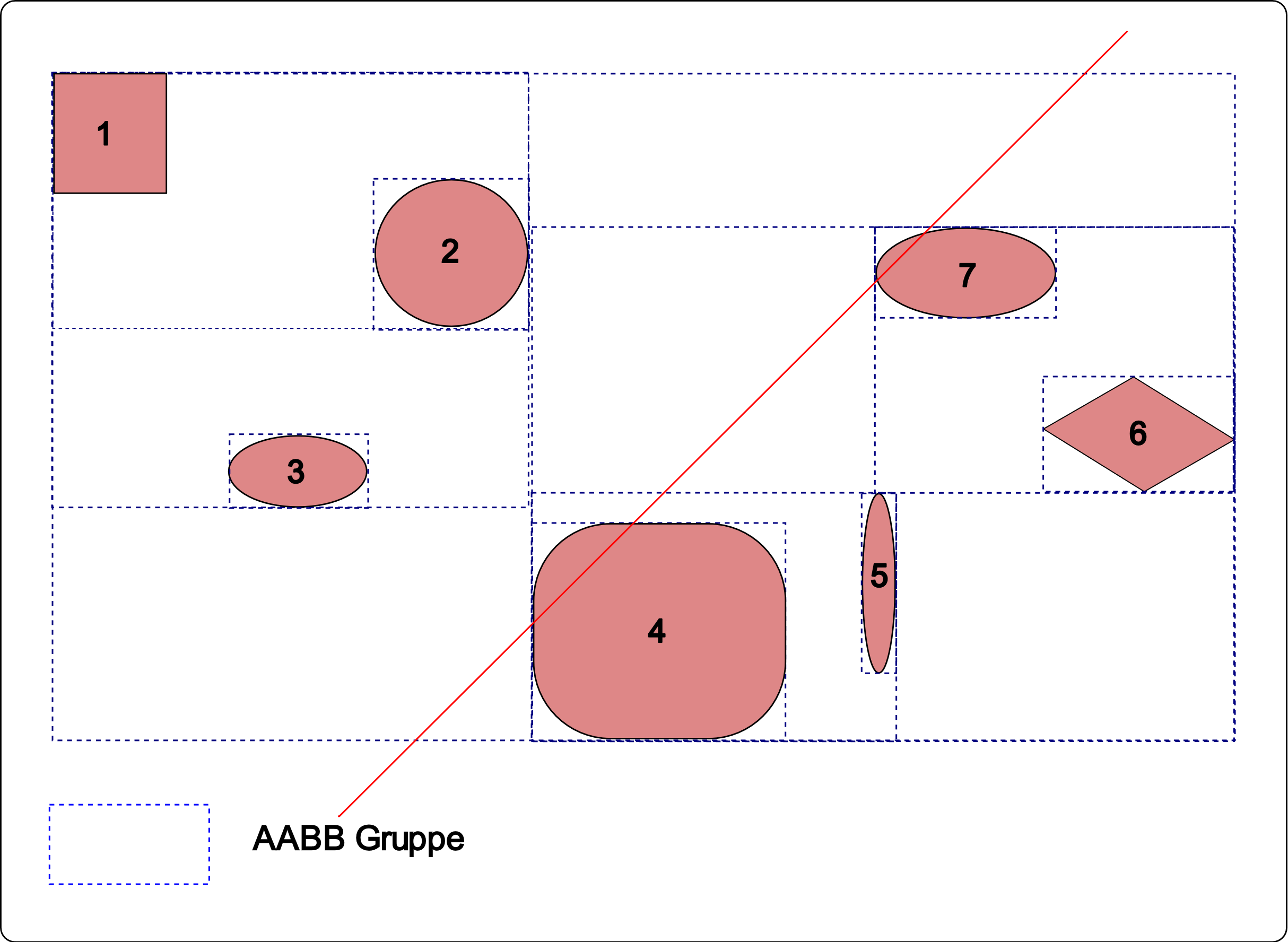
Bei dem Top Down Verfahren wird zuerst die übergreifende AABB erstellt, die alle Objekte beinhaltet. Danach wird die übergreifende Gruppe in zwei Gruppen geteilt. Für die Teilung gibt es zwei Möglichkeiten. Die AABB-Gruppe wird entweder in der Mitte aufgeteilt oder es werden die zwei AABBs mit der geringsten 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 dadurch kleiner werden und weniger Objekte überprüft werden müssen [5].

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 [6].

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 [6].

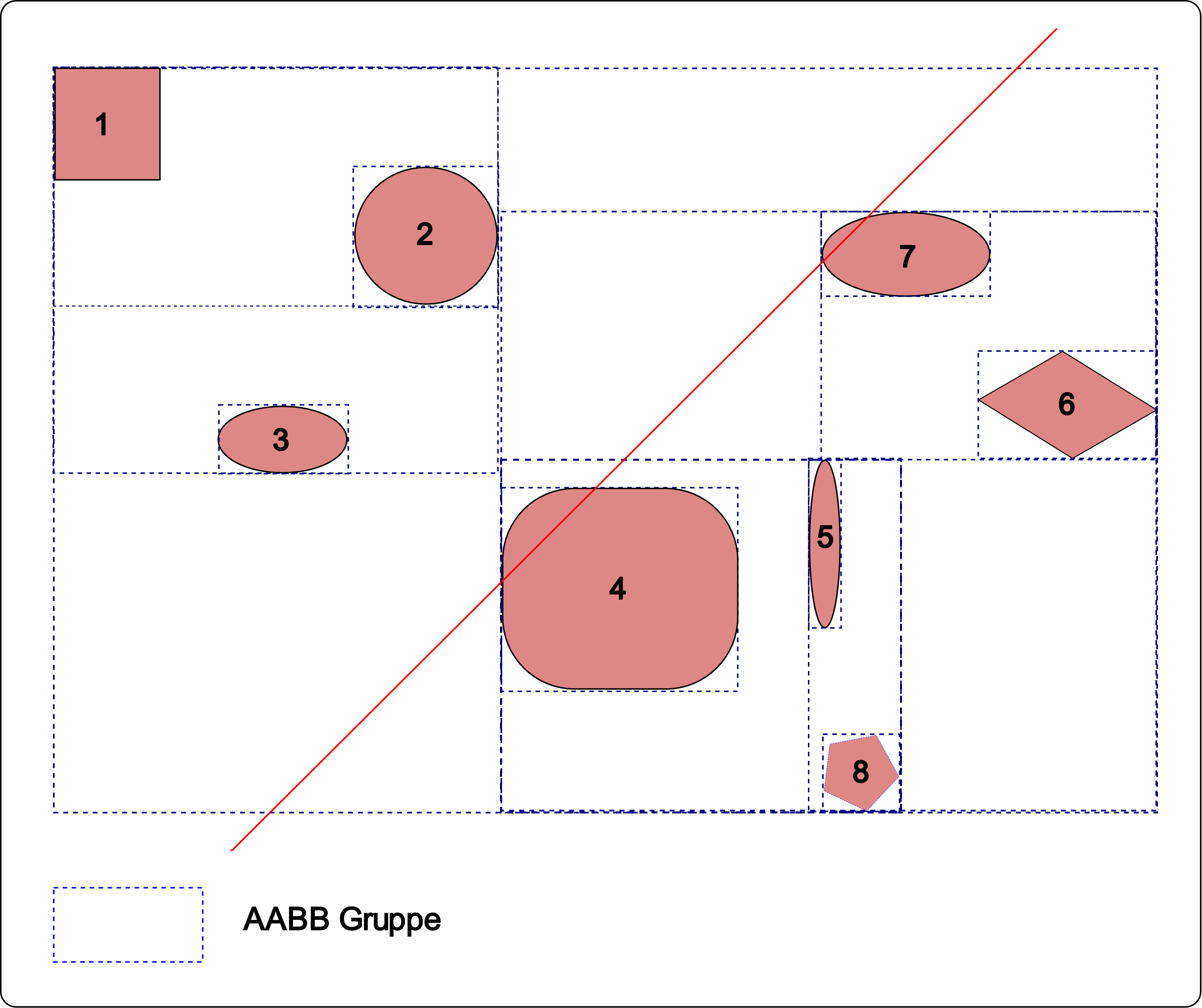
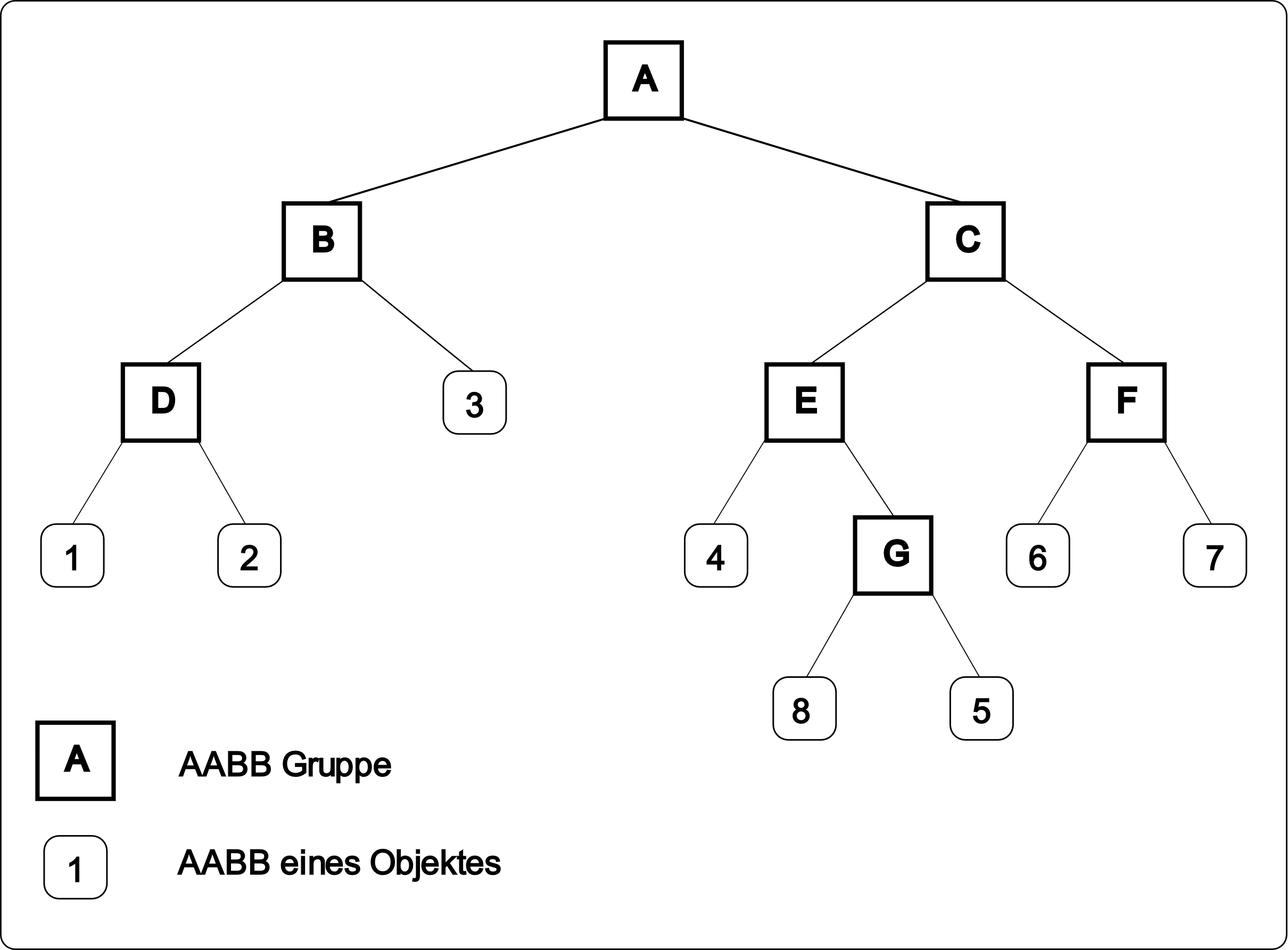
 

Abbildung 9: Grafik vor und nach dem Einfügen eines neuen Objektes

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 [6].

Kostenfunktion einer Bounding Volume Hierarchy:

C(T) =

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Abbildung 10: 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 pro 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 [6].

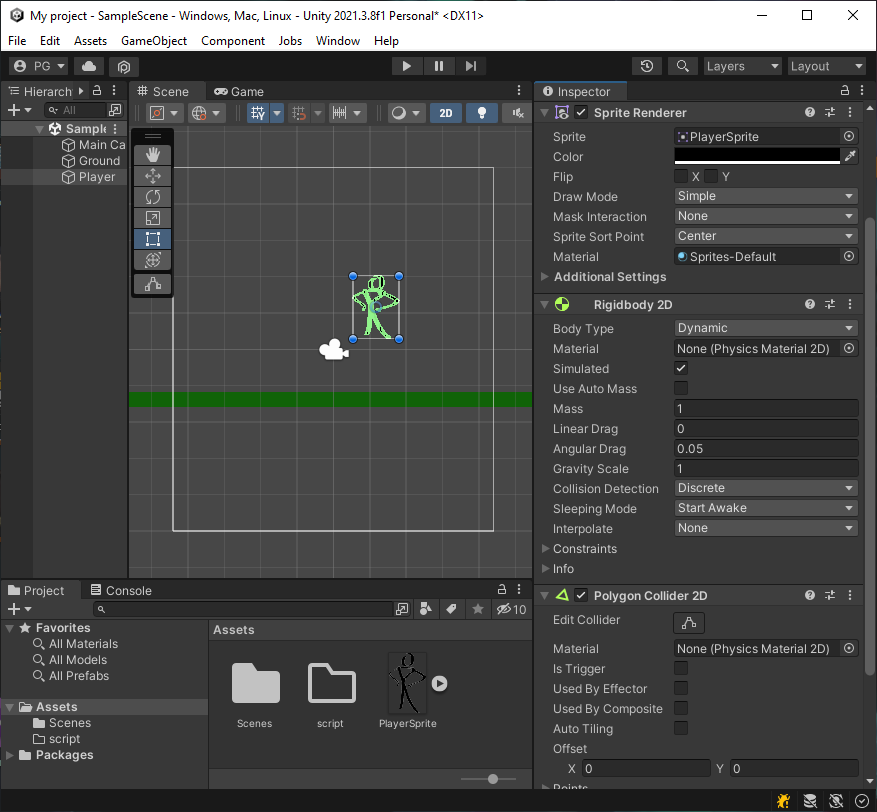
Im oben genannten Beispiel würde sich folgende Formel ergeben:

# Die Entwicklung von «Das Spiel»

## Einarbeitung in Unity

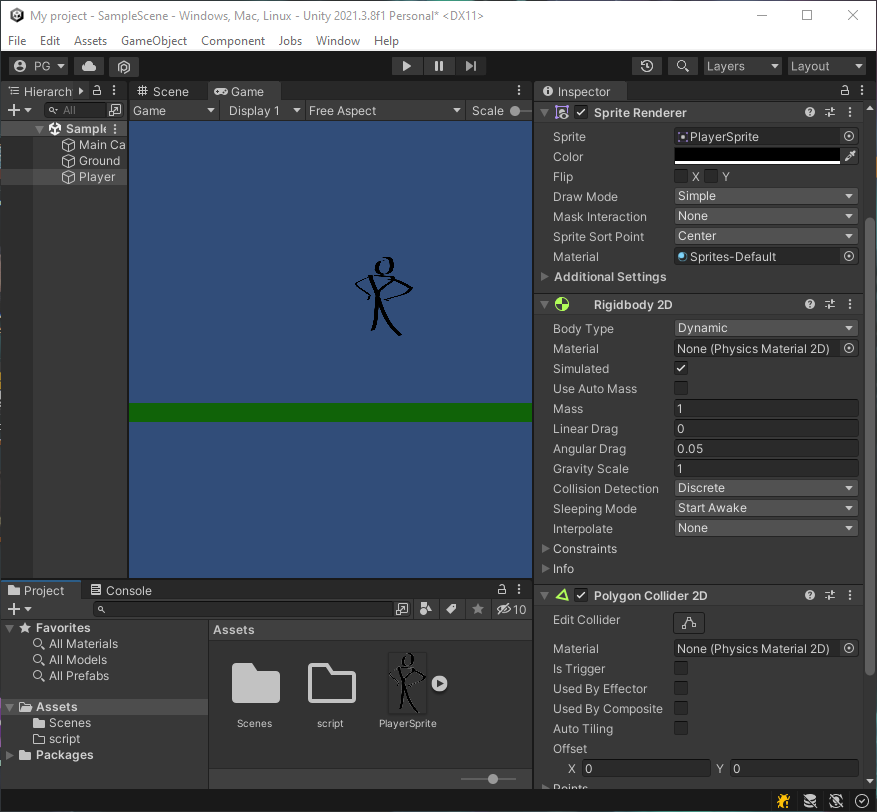
Bevor ich überhaupt anfing Unity auszuprobieren startete ich C# auf [Quelle] zu lernen. Nach der Installation von Unity und Visual Studio programmierte ich die Grundlagen des Spiels mit Hilfe des Tutorial von Pandemonium [Quelle]. Zuerst programmierte ich die Steuerung des Spielers, danach Animationen, Fallen, Sterben und das Erreichen von unterschiedlichen Checkpoints. Meine Wahl fiel auf dieses Tutorial, weil es das einzige war, welches jeden Schritt im Video programmiert und erklärt und es nicht empfohlen wird schon besseren vorgeschriebenen code zu verwenden. Die Geschwindigkeit ist auch sehr langsam was anfangs auch hilfreich war. Obwohl ich die Grundlagen von C# schon angeschaut hatte, konnte sich davon nur sehr wenig wiederverwenden, weil alle benötigten befehle Unity spezifisch waren. Das erste grosse Problem fiel mir während der vierten folge des Tutorials auf. Microsoft Visual Studio hat eine Autokorrektur namens IntelliSense. Aus irgendeinem Grund synchronisierte sich IntelliSense nicht mit Unity weshalb ich vor allem als C# Anfänger mehr Fehler aufgrund der Camel Back (Erklärung?) Schreibweise machte als nötig gewesen wäre, weil die Befehle in C# Case Sensitive sind (Erklärung). Bis zum Ende der Tutorial Reihe fand ich mich immer besser zurecht und konnte auch schon grösstenteils ohne Hilfe meine Ideen programmieren. Ich installierte mir auch GitHub, womit ich meinen vorschritt, speichern kann und im Notfall wenn etwas gar nicht mehr funktioniert auch alle vorherigen speicherstände wiederherstellen. Für neue Teile Unitys wie der Führung der Kamera mit einem Add-on, dem Hauptmenu und Speichern und laden des Spielstandes in einer Binärdatei habe ich trotzdem noch Hilfe von Tutorials benötigt. Ansonsten hat mir das Unity Manual [Quelle], in welchem jede Funktion und Befehl erklärt wird sehr geholfen. Trotzdem war es teilweise sehr schwer den richtigen Befehl mit der passenden Verwendung zu finden. In Meinem selbstgeschriebenen Code entstanden, aber auch mehr Fehler, dadurch dauerte es immer länger die Fehler und ihre Folgen zu beheben. Zum Beispiel löscht Unity beim Laden einer Szene alle Änderungen an einem prefab (einem als Asset gespeicherten Objekt) weshalb ich als ich probiert habe die Datei zu laden zuerst herausfinden musste, warum der Spieler nicht am richtigen Ort steht, schlussendlich habe ich herausgefunden das es eine Funktion gibt, mit der man Objekte beim Szenenwechsel nicht zerstört und in eine getrennte Szene macht, in welcher man sie weiterverwenden kann. Aufgrund dieses Szenenwechsels musste ich aber die Art, in welcher mein restlicher Code auf dieses Objekt zugreift, ändern.

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



Wenn man ein neues Unity Projekt erstellt, sieht man als erstes die Hierarchie (Links), die Szene (in der Mitte) und den Inspector (rechts). Durch einen rechtsklick in der Hierarchie können durch ein Menü die ersten Objekte in die Szene hinzugefügt werden. In der Grafik sind es zwei Objekte einmal ein Rechteck, welches ich Ground genannt habe. Es dient als einfacher Boden. Dem anderen Objekt habe ich eine Strichmännchen-Grafik im Sprite Renderer, welcher sich in dem Inspektor befindet, zugeteilt. Von Anfang an ist auch eine Kamera, durch welche der Spieler das Spiel sieht in der Szene. Damit diese Objekte zusammenstossen können habe ich dem Ground einen Boxcollider2D gegeben und dem Player einen Polygoncollider2D der Polygoncollider2D erkennt die umrisse der Grafik und erstellt die entsprechende Chainshape. Weil sich das Spieler Objekt auch bewegen sollte, habe ich ihm auch noch einen Rigidbody2D gegeben.

Wenn man das Spiel jetzt laufen lässt, fällt der Spieler auf den Boden und weil die beine der Grafik nicht gleich lang sind, kippt er um.

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

Automatisch generierte Beschreibung

### Programmierung der Spielfigur mit C#

Der nächste Schritt ist es dem Spieler die Möglichkeit zu geben das Spielerobjekt zu steuern. Dafür muss zuerst die Rotation auf der z Achse eingefroren werden damit das Spielerobjekt nicht mehr umkippt. Danach kann ein C# Script erstellt und in den Inspektor des Spielerobjekt gezogen werden.

Wenn das Spieler Objekt geladen wird, wird die Start Methode ausgeführt.in dieser wird der variable body der Rigidbody2D des Spieler Objekts zugeteilt. In der Update Methode, welche jeden frame aufgerufen wird, nimmt Unity die Eingabe des Spielers mit Input.GetAxis("Horizontal") diese horizontale Bewegung mit den dazugehörigen Tasten ist in Unity schon standardmässig eingestellt. Danach muss nur noch die Geschwindigkeit des Rigidbody2Ds an die Eingabe, welche in einem Vektor3 als x wert verwendet wird, angepasst werden.

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

Zuerst hatte ich viele unterschiedliche Ideen, von diesen Ideen waren die meisten von Anfang an nicht sehr realistisch also entschied ich mich für einen Plattformer. Wenn ich die Schwierigkeit im Verlauf der Arbeit erhöhen will, hätte ich komplexere Gegner einbauen können, was ich aber nicht machte. 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.

## Entwicklung des Spiels

In den folgenden Kapiteln werden Funktionen des Spiels mit Code Snipplets erklärt.

### Freischaltung von Fähigkeiten

Um zu überprüfen, ob der Spieler eine neue Fähigkeit erworben hat, 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 vorbestimmten Tag verglichen. Wenn das Objekt eine Fähigkeit freischaltet, wird der Boolean welcher die Nutzung der Fähigkeit verbietet in einem anderen Skript auf true gesetzt damit der Spieler diese Fähigkeit verwenden kann. 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

Wenn eine Fallende Plattform erstellt wird , wird ihr in der Start Methode, welche immer nach dem Laden des Objekts Ausgeführt wird, die gleich Gravitation wie dem Spieler gegeben und alle Bewegungen eingefroren. Wenn der Spieler diese Plattform berührt, wird eine Coroutine (TODO: noch erklären) gestartet, in welcher die Plattform gelöst wird und nach unten fällt. Nach 10 Sekunden wird sie wieder an die richtige Stelle, an welcher ein zweites Objekt ist, 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;

}

### Umsetzung von unterschiedlichen Sprüngen

Die Jump() Methode ist in einer Update Methode in welcher bei jedem neuen Frame überprüft wird ob die sprungtaste gedrückt ist. Das erste If Statement ist wahr, wenn der Spieler an einer Wand ist und lässt ihn von dieser abspringen. Damit sich das Spiel für den Spieler besser anfühlt und es nicht bestrafend ist sehr kurz zu spät zu drücken gibt es die Coyote Time in diesem snipplet wird zuerst mit isGrounded() überprüft ob der Spieler auf dem Boden ist wenn er nicht auf dem Boden ist wird geschaut ob der Coyote Counter über null ist. Der Coyote Counter wird in der Update Methode immer, wen der Spieler in Kontakt mit dem Boden ist auf 0.25 Sekunden gesetzt und wen der Spieler den Kontakt verliert, darf er für diese 0.25 Sekunden trotzdem noch Springen nach jedem Sprung wird der Coyote Counter auf 0 gesetzt, um ungewollte Doppelsprünge zu vermeiden. Mit dem letzten if Statement wird überprüft, ob der Spieler 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));

}

### Erkennung der Bewegungsrichtung

Hier nehme ich die horizontale Eingabe des Spielers aus dem inputManager und nehme mit Mathf.Sign nur das Vorzeichen damit sich das Spiel auf Tastatur und Controller genau gleich verhält. Auf der Tastatur kann man nur die werte -1, 0 und 1 eingeben während mit dem stick des Controllers jeder Wert zwischen -1 und 1 möglich ist weshalb das Spieler Objekt sich anders bewegen würde.

if (inputManager.Land.MoveHorizontal.ReadValue<float>() != 0)

{

Horizontalinput = Mathf.Sign(inputManager.Land.MoveHorizontal.ReadValue<float>());

}

else

{

Horizontalinput = 0;

}

Danach wird mit dem nachfolgenden code 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);

### Zurücksetzen von Spr

Dieses Code-Snipplet ist in einer Update(TODO update methode schon erklärt?) Methode. Hier wird in den ersten zwei Zeilen des if-Statements der Spieler bewegt. In diesem Snipplet werden auch der Coyote Counter und der jumpcounter zurückgesetzt, wenn der Spieler den Boden berührt, die überprüfung wird mit der isGrounded() methode vorgenommen.

if (!isDashing)

{

Body.gravityScale = Gravity;

Body.velocity = new Vector2(Horizontalinput \* Speed, Body.velocity.y);

if (isGrounded())

{

coyotecounter = coyoteTime; //reset time

jumpCounter = extraJumps; //reset Jumps

}

else

{

coyotecounter -= Time.deltaTime;

}

}

### Erkennung der Berührung von Wänden und Boden

In der ersten Methode wird überprüft, ob der Spieler auf dem Boden ist, dafür wird ein Boxcast nach unten verwendet. Wenn dieser etwas mit dem Tag GroundLayer überschneidet, ist der Return wert True. <TODO Bild>

In der zweiten Methode wird zuerst überprüft, ob der Spieler schon weitgenug ist, um wallinteraction nutzen zu können, wenn nicht ist der Return Wert immer False. Wenn der Spieler schon weitgenug ist, wird mit einem Boxcast in Blickrichtung des Spielers auf die Seite überprüft, ob der Spieler an einer Wand ist. Wallinteractions ist der Boolean welcher verwendet wird, um das Festhalten an Wänden und Wandsprünge zu kontrollieren. Weil vor diesen Aktionen immer kontrolliert wird, ob der Spieler in Kontakt mit einer Wand ist, ist der einfachste weg diese variable in den code einzubauen an dieser Stelle. <TODO Bild>

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 seine Sprunghöhe steuern kann wird, solange der Spieler Springen gedrückt hält die Gravitation halbiert. Ausserdem kann sich der Spieler im zweiten Teil des Codes an einer Wand festhalten, dabei wird die Gravitation durch 10 geteilt das der Spieler nur noch langsam 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;

}

### Dash-Fähigkeit

Ein Dash ist eine schnelle Bewegung in die von dem Spieler ausgewählte Richtung. Die Distanz welche während einem Dash zurückgelegt wird ist immer gleich.

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 Stop Dashing wird zuerst für die zeit des dashes gewartet. Danach wird der Boolean welcher anzeigt, ob das Spielerobjekt gerade einen Dash ausführt auf False gesetzt. Die Partikel werden deaktiviert und die Steuerung des Spielers wieder aktiviert. Danach wird die Gravitation wieder auf den Standartwert zurückgesetzt.

private IEnumerator StopDashing()

{

yield return new WaitForSeconds(dashingTime);

isDashing = false;

GetComponentInChildren<ParticleSystem>().Stop();

inputManager.Enable();

Body.gravityScale = Gravity;

}

## Einbindung von speziellen Features

* Bei der Einbindung von ton wie auch Grafiken war die Suche nach den passenden Dateien das schwierigste
* Durch Autotiling Erweiterung wurde das Erstellen der Karte angenehmer
* Grafiken waren nachdem man den Kniff dahinter gefunden hat erstaunlich einfach
* Unity übernimmt dabei auch einiges an Arbeit
* 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 weshalb teile des Codes verändern musste danach gab es noch ein Problem das über den Controller Werte von 0-1 und auf der Tastatur nur entweder das eine oder das Andere was obwohl es sehr einfach war Probleme auslöste
* Audio

# «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.

# Nachwort

# Quellenverzeichnis

|  |  |
| --- | --- |
| [1] | Unity Technologies, «Unity,» 2022. [Online]. Available: https://unity.com/. |
| [2] | Unity Technologies, «Unity Documentation,» 2 12 2022. [Online]. Available: https://docs.unity3d.com/Manual/unity-architecture.html. |
| [3] | 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]. |
| [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] | A. Kensler, «Tree rotations for improving bounding volume hierarchies,» in 2008 IEEE Symposium on Interactive Ray Tracing, Los Angeles, IEEE, 2008, pp. 73-76. |
| [9] | Catto Erin, «Box 2D Manual,» Box2D, 2021. [Online]. Available: https://box2d.org/documentation/. [Zugriff am 30 10 2022]. |

# Probleme

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

# 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!