

MASTERARBEIT

zur Erlangung des akademischen Grades
„Master of Science in Engineering“
im Studiengang Game Engineering und Simulation

Evaluierung von Algorithmen zur Fragmen- tierung von 3D-Objekten in Realtime

Ausgeführt von: Michael Prommer, BSc.

Personenkennzeichen: 1910585010

BegutachterInnen: FH-Prof. Dipl.-Ing. Alexander Nimmervoll
Dipl.-Ing. Dr. Gerd Hesina

Wien, den 2. Mai 2022

Eidesstattliche Erklärung

„Ich, als Autor / als Autorin und Urheber / Urheberin der vorliegenden Arbeit, bestätige mit meiner Unterschrift die Kenntnisnahme der einschlägigen urheber- und hochschulrechtlichen Bestimmungen (vgl. Urheberrechtsgesetz idG sowie Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idG).

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig angefertigt und Gedankengut jeglicher Art aus fremden sowie selbst verfassten Quellen zur Gänze zitiert habe. Ich bin mir bei Nachweis fehlender Eigen- und Selbstständigkeit sowie dem Nachweis eines Vorsatzes zur Erschleichung einer positiven Beurteilung dieser Arbeit der Konsequenzen bewusst, die von der Studiengangsleitung ausgesprochen werden können (vgl. Satzungsteil Studienrechtliche Bestimmungen / Prüfungsordnung der FH Technikum Wien idG).

Weiters bestätige ich, dass ich die vorliegende Arbeit bis dato nicht veröffentlicht und weder in gleicher noch in ähnlicher Form einer anderen Prüfungsbehörde vorgelegt habe. Ich versichere, dass die abgegebene Version jener im Uploadtool entspricht.“

Wien, 2. Mai 2022

Unterschrift

Kurzfassung

Die vorliegende Arbeit befasst sich mit Algorithmen zum Zerstören von Objekten, insbesondere für Videospiele. Das Ziel ist es eine Übersicht über aktuell verwendete Methoden und einen Einblick in die Implementierung solcher zu geben. Weiters werden in dieser Arbeit zwei dieser Methoden gegenübergestellt und in Kriterien wie etwa Leistung, Qualität und Komplexität verglichen.

Um diese Ziele zu erreichen wurde zuerst eine ausführliche Literaturrecherche durchgeführt und im theoretischen Teil der Arbeit diese vermittelt. Anschließend wurden Details zur Implementierung von zwei Methoden nähergebracht. Weiters wurden die Leistung dieser Methoden mithilfe einer quantitativen Analyse gemessen und deren Qualität der Zerstörung und Komplexität der Implementierung verglichen.

Der Vergleich zeigte, dass eine Methode um einiges effizienter und einfacher in der Implementierung ist, jedoch die Qualität der Fragmente nicht weiter verbessert werden kann. Im Rahmen dieser Arbeit konnte nur eine grundlegende Version der anderen Methode entwickelt werden und ist dadurch nur für eine geringe Anzahl von Fragmenten in Echtzeit einsetzbar. Jedoch kann diese in einigen Aspekten optimiert und erweitert werden.

Schlagworte: Fragmentierung, Zerstörung, Voronoi, 3D

Abstract

This thesis deals with algorithms for destroying objects, especially for the use in video games. The goal is to give an overview of currently used methods and an insight into the implementation of such methods. Furthermore, two of these methods are contrasted in this thesis and compared in criteria such as performance, quality and complexity.

In order to achieve these goals, first an extensive literature review was conducted and this was conveyed in the theoretical part of the thesis. Afterwards, details on the implementation of two methods were presented. Additionally, the performance of these methods was measured using a quantitative analysis and their quality of destruction and complexity of implementation were compared.

The comparison showed that one method is much more efficient and easier to implement, but the quality of the fragments cannot be further improved. Within the scope of this work, only a basic version of the other method could be developed and is thus only applicable for a few fragments in real time. However, this can be optimized and extended in some aspects.

Keywords: Fragmentation, Destruction, Voronoi, 3D

Danksagung

Zuerst möchte ich mich bei meinen Eltern bedanken, die mir mein Studium an der Fachhochschule Technikum Wien ermöglicht haben und mich stets unterstützt haben. Weiters bedanke ich mich bei meinen Studienkollegen Jonas, Maki, Marius, Robert und Theo für die gemeinsame Zeit während des Studiums. Ebenso möchte ich mich bei meinem Freund Marcel bedanken für die Unterstützung beim Projekt, wenn ich nicht mehr wusste.

Mein Dank gilt auch meinem Betreuer FH-Prof. Dipl.-Ing. Alexander Nimmervoll, einerseits für die interessanten Lehrveranstaltungen während des Studiums, aber auch für das hilfreiche Feedback während dem Schreiben der Arbeit.

Und ein besonderer Dank geht an meine Freundin und Partnerin Eli, die während des gesamten Studiums jeden Tag für mich da war und mich beruhigen konnte, wenn ich mich überfordert fühlte.

Inhaltsverzeichnis

1 Einleitung	1
2 Voronoi Diagramm	2
2.1 Definition	2
2.2 Einsatzgebiete	5
2.3 Delaunay Triangulation	8
2.4 Generieren der Punktmenge	10
2.5 Datenstrukturen	12
2.5.1 Delaunay Struktur	13
2.5.2 Winged-Edge	13
2.5.3 Doubly-Connected Edge List (DCEL)	15
2.6 Algorithmen zur Berechnung des Voronoi Diagrams	17
2.6.1 Brute-Force	18
2.6.2 Fortune Algorithmus	19
2.6.3 Divide & Conquer rekursiv	21
2.6.4 Inkrementeller Flipping Algorithmus	22
2.6.5 Bowyer-Watson Algorithmus	24
2.7 Resümee	25
3 State of the Art	26
3.1 Fragmentierung in Videospielen	26
3.2 Geometriebasierte Methoden	29
3.2.1 Austausch des Objektes	30
3.2.2 Dynamische Methoden	33
3.3 Physikalisch basierte Methoden	35
3.3.1 Mass-Spring Model	36
3.3.2 Finite-Elemente-Methode	38
3.4 Clipping	40
3.4.1 Constructive Solid Geometry	41
3.4.2 Schneiden eines Meshes mit einer Ebene	41
3.5 Resümee	44

4 Implementierung	45
4.1 Entwicklungsumgebung	45
4.1.1 Hardware	45
4.1.2 Applikationen	45
4.2 Features	46
4.2.1 2D Visualisierungen	46
4.2.2 Austausch mit vorgebrochenen Model	48
4.2.3 Voronoibasierte Zerstörung	52
5 Ergebnisse und Diskussion	59
5.1 Leistung	59
5.2 Qualität	61
5.3 Komplexität	63
6 Zusammenfassung und Ausblick	65
Literaturverzeichnis	66
Abbildungsverzeichnis	73
Tabellenverzeichnis	76
Quellcodeverzeichnis	77
Abkürzungsverzeichnis	78

1 Einleitung

In modernen Videospielen und Filmen werden verschiedenste Effekte verwendet um dem Spieler den die Welt glaubhafter und realer erscheinen zu lassen. Einer davon ist die Zerstörung von unterschiedlichen Objekten im Spiel wie einfache Gegenstände oder in manchen Spielen sogar zerstörbare Landschaften. Hierfür existieren verschiedene Methoden, die im Rahmen dieser Arbeit begutachtet werden. Das Ziel dieser Arbeit ist die Implementierung und Evaluierung von zwei Methoden zum Fragmentieren von Objekten in Echtzeit. Dabei werden folgende Fragen untersucht und beantwortet:

- Welche Schritte werden für die Entwicklung einer Methode zum Zerstören von Objekten benötigt?
- Wie werden Methoden zum Zerstören von Objekten anhand der Kriterien Leistung, Qualität und Komplexität der Implementierung analysiert, gemessen und in weiterer Folge verglichen?

Um die genannten Punkte zu erreichen, wird eine ausführliche Literaturrecherche durchgeführt und im theoretischen Teil der Arbeit der aktuelle Stand der Forschung nähergebracht. Mithilfe der Implementierung von zwei Methoden zum Zerstören von Objekten wird im praktischen Teil der Arbeit ein Einblick in die Implementierung gegeben. Die erste der beiden Methoden ist das Austauschen des Objekts mit einem bereits vorgebrochenen Objekt zum Zeitpunkt der Kollision. Bei der zweiten wird zuerst das sogenannte Voronoi Diagramm erstellt und abhängig von diesem werden die Fragmente zur Laufzeit erstellt. Anschließend wird eine Gegenüberstellung dieser zwei implementierten Algorithmen durchgeführt.

Zuerst werden die Grundlagen des Voronoi Diagramms nähergebracht, wie etwa die Definition, die Einsatzgebiete und wie dieses sowohl in 2D als auch in 3D berechnet werden kann. Danach wird der aktuelle Stand der Forschung gezeigt, wo unter anderem die verschiedenen Möglichkeiten zum Zerstören von Objekten gezeigt werden und welche Herausforderungen bei Echtzeitanwendungen existieren. Zusätzlich wird in diesem Kapitel das Schneiden des Objektes erläutert, da dies unabhängig von der gewählten Methode zum Einsatz kommt und ein komplexer Schritt der Zerstörung ist. Durch das nächste Kapitel der Arbeit, der Implementierung, wird gezeigt, wie diese beiden Methoden entwickelt und getestet wurden. Des Weiteren werden die Ergebnisse evaluiert und diskutiert um die entwickelten Methoden zu vergleichen. Zuletzt wird die Arbeit durch eine Zusammenfassung und einem Ausblick abgeschlossen.

2 Voronoi Diagramm

In den 1970er Jahren entstand das Gebiet der algorithmischen Geometrie (engl. Computational Geometry), welches sich mit geometrischen Problemen befasst [15, S. 2]. Sie kann definiert werden, als die systematische Untersuchung von Algorithmen und Datenstrukturen für geometrische Objekte, wobei der Schwerpunkt auf exakten Algorithmen liegt, die möglichst schnell sind. Heute gibt es eine reiche Sammlung geometrischer Modelle und Algorithmen, darunter auch das Voronoi Diagramm. Dadurch, dass ein Großteil der geometrisch basierten Methoden, welche in Kapitel 3.2 erklärt werden, auf dem Voronoi-Diagramm basieren, folgt eine genauere Definition und Beschreibung von diesem. Zuerst werden in diesem Kapitel die Definition des Voronoi Diagramms und dessen Einsatzgebiete erläutert. Anschließend folgt eine Übersicht der Algorithmen, mit denen ein Voronoi Diagramm berechnet werden kann und wie dieses unter anderem auch für die Zerstörung von Objekten verwendet werden kann.

2.1 Definition

Voronoi Diagramme sind grundlegende Datenstrukturen, die in der algorithmischen Geometrie eingehend untersucht wurden. Ein Voronoi-Diagramm kann definiert werden, als die Aufteilung einer Ebene mit n Punkten in konvexe Polygone. Jedes dieser Polygone enthält genau einen von n Punkten und im Falle des euklidischen Abstands ist jede Position in einem bestimmten Polygon näher an n als an jedem anderen [10]. Der euklidische Abstand wird folgendermaßen definiert: In der Ebene, wenn Punkt p die Koordinaten (p_1, p_2) und Punkt q die Koordinaten (q_1, q_2) hat, wird die euklidische Distanz zwischen p und q mit folgender Formel berechnet:

$$d_{Euklid}(p, q) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2}. \quad (1)$$

Das Ergebnis ist eine Unterteilung der Ebene in eine Menge von Regionen, die sogenannten Voronoi Zellen (engl. cells), die zusammen das Voronoi Diagramm bilden, wie in Abbildung 1 zu sehen ist [62, S. 1].

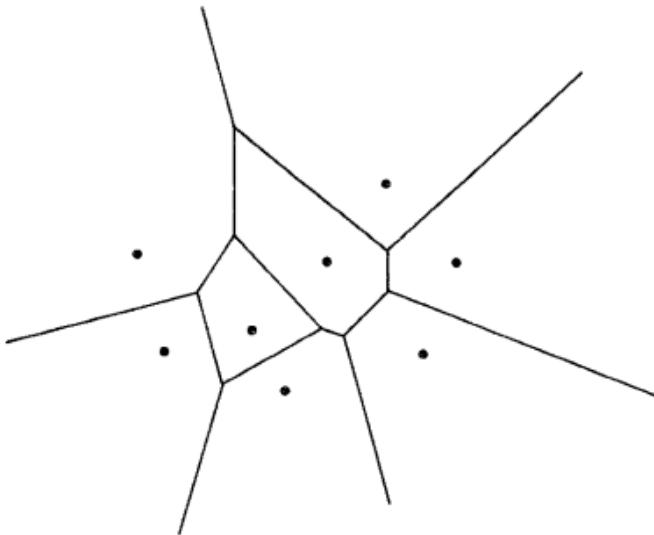


Abbildung 1: Voronoi Diagramm mit acht Punkten in der Ebene und mit dem euklidischen Abstand berechnet [6]

Benannt ist das Voronoi Diagramm nach dem russischen Mathematiker Georgy Fedoseevich Voronoy (1868-1908), welcher das Diagramm zur Untersuchung quadratischer Formen nutzte [83]. Shamos [68] und Shamos und Hoey [69] führten das zweidimensionale Voronoi Diagramm in die algorithmische Geometrie ein. Sie präsentierten einen effizienten Algorithmus für dessen Konstruktion, um eine Lösung des *Closest-point problems* oder auch *k-nearest neighbor problem* genannt vorzustellen, welches in Kapitel 2.2 beschrieben wird. Auch andere Formen des Voronoi Diagramms wurden von mehreren Autoren erwogen. Shamos und Hoey [69] führten Voronoi Diagramme mit einer höheren Ordnung ein. Dies bedeutet bei einem Voronoi Diagramm mit der Ordnung k , können die k Punkte, die zu einem neuen Punkt x am nächsten liegen bestimmt werden, indem das Polygon ermittelt wird, in dem x liegt. Ein Beispiel, wie ein Voronoi Diagramm mit Ordnung zwei aussieht, wird in Abbildung 2 gezeigt.

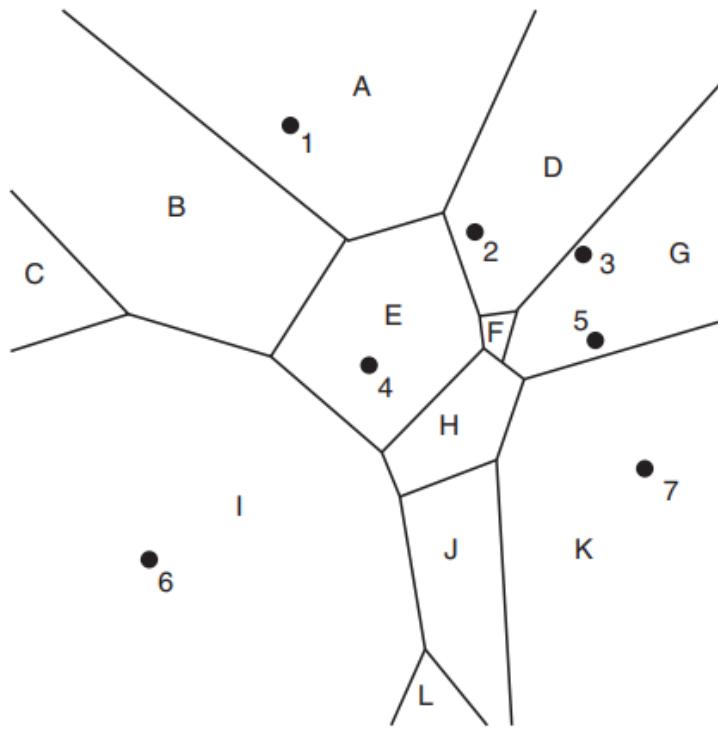


Abbildung 2: Ein Voronoi Diagramm mit Ordnung 2. Im Vergleich zum regulären Voronoi Diagramm enthält nicht jede Zelle einem Punkt der Punktmenge. [43]

Eine weitere Art des Voronoi Diagramms haben Lee und Drysdale [41] sowie Kirkpatrick [37] untersucht, indem sie die Punkte, welche zum Berechnen verwendet werden, beispielsweise mit Linien ersetzen. In der Arbeit von Aurenhammer und Edelsbrunner [7] wird noch eine weitere interessante Form des Voronoi Diagramms untersucht, indem jedem Punkt p der gegebenen Menge ein positives Gewicht $w(p)$ zugewiesen wird, welches die Stärke von p ausdrückt. Die originalen Voronoi Diagramme sind der Spezialfall, bei dem alle Punkte das gleiche Gewicht besitzen. Diese neue Struktur wird als gewichtetes Voronoi Diagramm einer Menge von gewichteten Punkten bezeichnet.

In den meisten Fällen werden Voronoi Diagramme mit dem euklidischen Abstand zwischen den Punkten berechnet, aber auch andere Abstandsfunktionen wie beispielsweise die Manhattan-Distanz

$$d_{Manhattan}(p, q) = |(p_1 - q_1)| + |(p_2 - q_2)| \quad (2)$$

oder die Maximumsnorm (Tschebyschew-Norm)

$$d_{Tschebyschew}(p, q) = \max(|(p_2 - q_1)|, |(q_2 - p_1)|) \quad (3)$$

können verwendet werden und ergeben infolgedessen eine unterschiedliche Aufteilung des Raumes wie in Abbildung 3 zu sehen ist. Bei dem euklidischen Abstand sind die Kanten der konvexen Voronoi Zellen die Mittelsenkrechten der benachbarten Punkte [73].

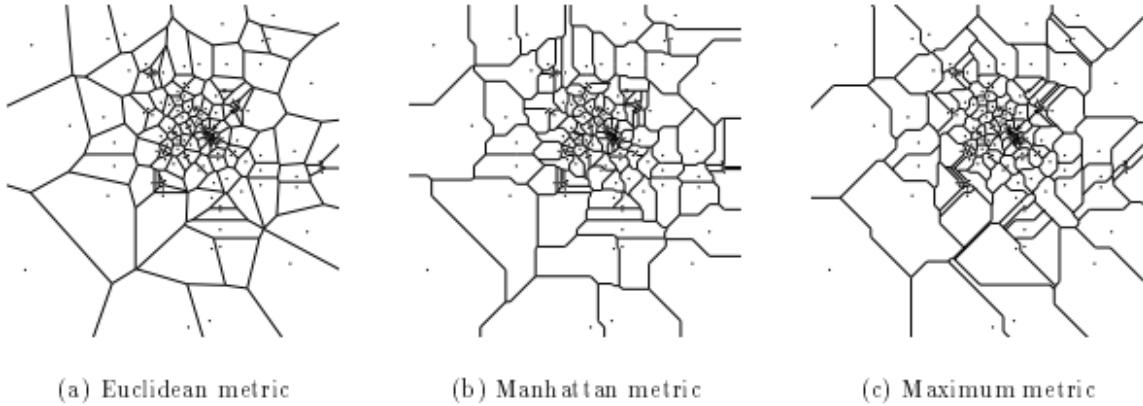


Abbildung 3: Berechnung des Voronoi Diagramms mit dem euklidischen Abstand, der Manhattan-Distanz und der Maximumsnorm. [34]

Unabhängig davon, welche Metrik für die Berechnung verwendet wird, besteht ein Voronoi Diagramm immer aus einer Sammlung von Regionen, wobei jede Region durch eine Reihe von Flächen, Kanten und Eckpunkten definiert ist. Die Anzahl solcher Flächen, Kanten und Eckpunkte beschreibt die Komplexität des Voronoi Diagramms und ist auch eine untere Schranke dafür, wie lange es dauert, ein Voronoi Diagramm zu berechnen [67].

2.2 Einsatzgebiete

Interessanterweise ist das Konzept des Voronoi Diagramms, beziehungsweise dessen Aufteilung, viel älter als die Person, nach der es benannt ist und wird in verschiedenen Bereichen eingesetzt, wie beispielsweise in der Astronomie, Anthropologie, Biologie, Chemie, Geografie, Physik, Physiologie, Statistik und Robotik [62, S. 2] [67]. Außerdem kommen Voronoi Diagramme auch in der Natur vor, wie zum Beispiel in dem Muster eines Schildkrötenpanzers (Abbildung 4), im Zellsystem von Blättern (Abbildung 5) und in der Art und Weise, wie getrocknete Erde aufgerissen wird (Abbildung 6). Aufgrund der Unregelmäßigkeiten an den Rändern handelt es sich bei den genannten Beispielen nicht um makellose Voronoi Diagramme, dennoch ist das Muster des Voronoi Diagramms klar erkennbar [73].



Abbildung 4: Voronoi Muster eines Schildkrötenpanzers [85]



Abbildung 5: Zellsystem eines Blattes ähnelt dem Voronoi Diagramm [73]



Abbildung 6: Getrocknete rissige Erde. Da dies in der Realität einem Voronoi Diagramm ähnelt, kann es für Videospiele verwendet werden, um realistische Risse zu modellieren [73]

Ein Grund für das Auftreten von Voronoi Diagrammen in der Natur ist, dass sie typischerwei-

se Wachstum modellieren. Euklidische Voronoi Diagramme entsprechen einem Wachstum mit konstanter Geschwindigkeit, während andere Arten von Voronoi Diagrammen komplexere Systeme modellieren. Bei zweiterem ist die Geschwindigkeit nicht einheitlich, entweder konstant oder variabel für jede Zelle, wie etwa abhängig von der aktuellen Größe [85].

Die Konstruktion des Voronoi Diagramms eignet sich gut, um eine Art Einflussbereich um jeden Standort herum zu definieren [85]. Besonders das gewichtete Voronoi Diagramm kann bei folgenden geografischen Problemen eingesetzt werden. Angenommen, in einem bestimmten geografischen Gebiet wird eine Reihe von gleichzeitig intakten Betrieben betrachtet. Dabei ist jedem Betrieb eine Zahl zugeordnet, die dessen Reichweite auf die Kundschaft ausdrückt. Daraus kann abgeleitet werden, dass große Betriebe normalerweise eine größere Kundschaft anziehen als kleine Betriebe, jedoch wird ein kleiner Betrieb dominierender für die nähere Umgebung sein als ein größerer, welcher weiter entfernt ist. Das gewichtete Voronoi Diagramm kann nun diese Information erfassen und den Bereich der Reichweite für jeden Betrieb deutlich machen [7].

Weitere Problemstellungen, wo das gewichtete Voronoi Diagramm zum Einsatz kommen könnte, sind bei der Planung und Errichtung von Sendern mit unterschiedlicher Stärke oder beim Bauen von Aufladestationen für Elektroautos, abhängig von der Kapazität der Ladestationen, den Verkehrsbedingungen und der Dichte der Elektrofahrzeuge in der Region [74]. Ein Vergleich zwischen einem regulären Voronoi Diagramm und einem gewichteten Voronoi Diagramm wird in Abbildung 7 gezeigt.

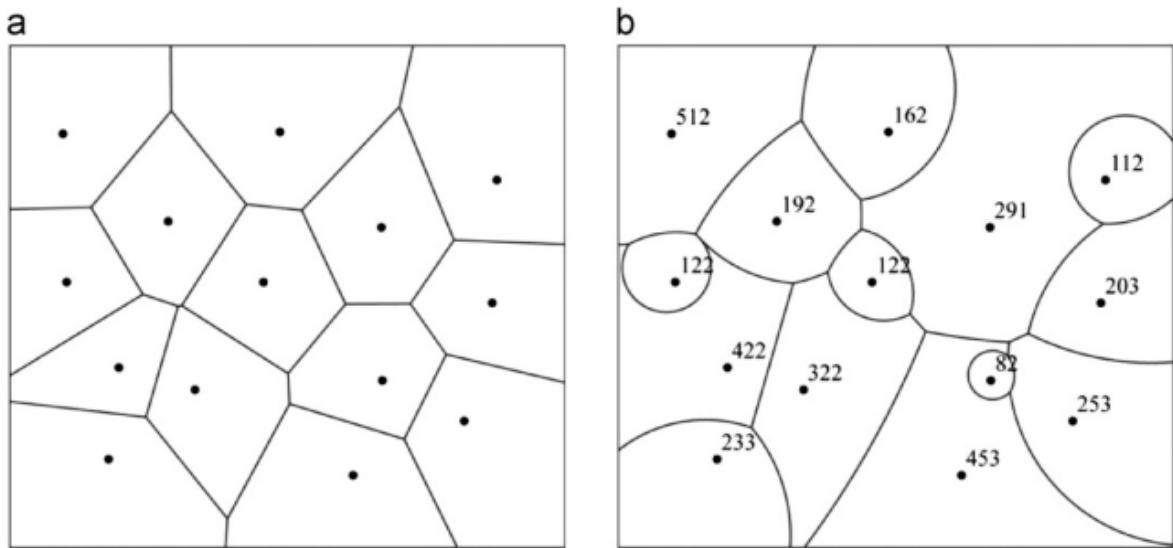


Abbildung 7: a) Reguläres Voronoi Diagramm.
b) Gewichtetes Voronoi Diagramm mit gleicher Punktmenge. Die Zahlen geben das Gewicht des jeweiligen Punktes an. [27]

2.3 Delaunay Triangulation

Die meisten der Algorithmen, die zur Berechnung des Voronoi Diagramms verwendet werden, berechnen nicht direkt das Voronoi Diagramm sondern eine eng verwandte Struktur, die Delaunay Triangulation. Diese wurde von dem Mathematiker Boris Delone (russische Schreibweise des ursprünglichen und üblichen französischen Delaunay) entwickelt, nachdem er als Teenager Georgy Voronoy kennengelernt hat und von dessen Arbeit inspiriert wurde [19].

Die Delaunay Triangulation ist eine Triangulation einer Ebene mit einer Menge von diskreten Punkten, sodass kein Punkt innerhalb des Umkreises eines Dreiecks liegt, wie in Abbildung 8 ersichtlich [73]. Diese Triangulation hat die einzigartige Eigenschaft, dass der minimale Winkel aller Dreiecke maximiert ist und dadurch so nah wie möglich an gleichseitige Dreiecke herankommt [85]. Dies kann auch in höheren Dimensionen verwendet werden, wie beispielsweise in 3D, wo Dreiecke zu Tetraedern und deren Umkreise zu Kugeln werden.

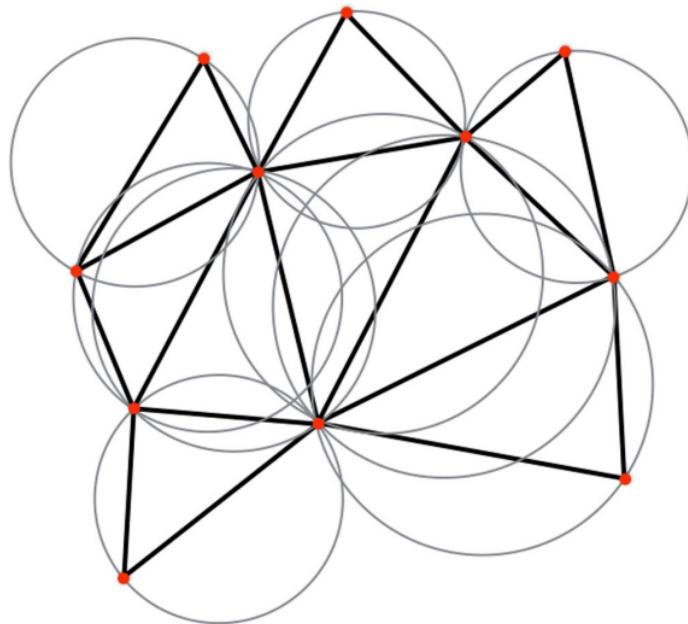


Abbildung 8: Eine Delaunay Triangulation in der Ebene, zusammen mit den Umkreisen der gebildeten Dreiecke. Per Definition enthält kein Umkreis einen beliebigen Eckpunkt der Triangulation. [38]

Triangulationen einer Punktmenge sind in der Computergrafik von großer Bedeutung. Die Triangulation ist für das polygonbasierte Rendering, welches von APIs wie OpenGL und Direct3D verwendet wird, unerlässlich. Das liegt daran, um Objekte darstellen zu können, müssen diese in Polygone, meist Dreiecke, geteilt werden. Die Triangulation wird in einer Vielzahl von Anwendungen eingesetzt, wie etwa bei der prozeduralen Generierung von Welten oder der Verminderung von Dreiecken in Meshes (engl. mesh decimation) aufgrund von beispielsweise Performance Gründen [20].

Eine weitere besondere Eigenschaft, die die Delaunay Triangulation besitzt, ist im Zusammenhang mit dem Voronoi Diagramm ersichtlich. Die Delaunay Triangulation ist im Falle des euklidischen Abstandes, unabhängig von der Dimension, der duale Graph des Voronoi Diagramms und vice versa [40]. Aus dieser Dualität ergibt sich, dass aus der Kenntnis der einen Struktur die andere extrahiert werden kann. Dieser Zusammenhang der beiden Modelle wird in Abbildung 9 gezeigt.

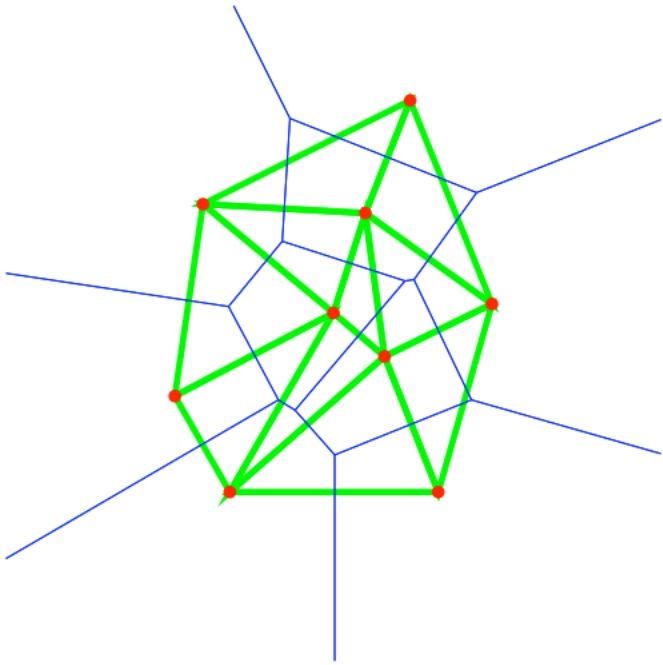


Abbildung 9: Dualität des Voronoi Diagramms (blaue Linien) und der Delaunay Triangulation (grüne Linien).[85]

Um nun bei einer zweidimensionalen Delaunay Triangulation das zugehörige Voronoi Diagramm zu extrahieren, werden alle Umkreismittelpunkte der Dreiecke verbunden, die eine gemeinsame Kante besitzen. Jeder Umkreismittelpunkt der Triangulation ist ein Punkt eines Polygons des Voronoi Diagramms [20].

Die Dualität zwischen dem Voronoi Diagramm und der Delaunay Triangulation funktioniert in \mathbb{R}^3 in derselben Art und Weise. Jedes Element der einen Struktur entspricht genau einem Element der anderen, wie in folgender Abbildung 10 veranschaulicht wird. Ein Delaunay-Punkt wird zu einer Voronoi-Zelle, eine Delaunay-Kante wird zu einer Fläche des Voronoi Diagramms, eine dreieckige Fläche eines Delaunay-Tetraeders wird zu einer Voronoi-Kante und ein Delaunay-Tetraeder wird zu einem Voronoi-Punkt. Ein Voronoi-Eckpunkt befindet sich im Zentrum der Kugel, die von seinem dualen Tetraeder umschlossen wird. Weiters haben zwei Eckpunkte der Punktmenge nur dann eine Delaunay-Kante, die sie verbindet, wenn ihre beiden jeweiligen dualen Voronoi-Zellen benachbart sind [39].

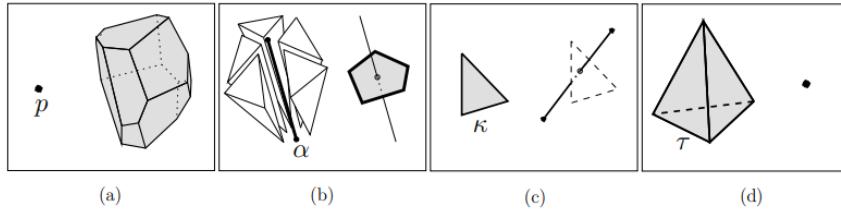


Abbildung 10: Dualität zwischen den Elementen des Voronoi Diagramms und der Delauany Triangulation in 3D [39].

2.4 Generieren der Punktmenge

Bei den oben genannten Einsatzgebieten des Voronoi Diagramms wird die Delaunay Triangulation, beziehungsweise das Voronoi Diagramm üblicherweise von einer bereits gegebenen Punktmenge gebildet. Diese wird abhängig von der Problemstellung generiert. Im oben genannten Beispiel der Einflussbereiche von Betrieben in Kapitel 2.2 entspräche ein Standort eines Betriebes einem Punkt und demnach bilden alle Standorte die benötigte Punktmenge. Diese Darstellungen werden auch als empirische Punktmengen bezeichnet. Obwohl die einzelnen Objekte selbst keine Punkte sind, sind solche Darstellungen möglich, weil die physischen Größen der Objekte im Verhältnis zu den Entfernungen zwischen ihnen als auch zur Größe der Region, in der sie vorkommen, sehr klein sind [62, S. 495].

Ist keine empirische Punktmenge vorhanden, muss zuerst eine Punktmenge generiert werden, mit der das Voronoi Diagramm gebildet werden kann. Diese Erstellung der Punktmenge ist nicht unbedeutend, da die Größe und Position der Voronoi Regionen abhängig von der Punktmenge sind. Dies ist vor allem im Hinblick auf das dynamische Zerstören von Objekten von Bedeutung, da die Fragmente mithilfe der Voronoi Zellen berechnet werden, welche durch die Position der Punkte bestimmt werden. Infolgedessen spielt die Aufteilung der Punkte eine erhebliche Rolle, wie die Fragmente eines Objektes letztendlich aussehen. Unter Berücksichtigung der Zufälligkeit eines Bruches sollten die Punkte zufällig verteilt sein. In Wirklichkeit hängen die Ergebnisse der Rissbildung und des Bruchs jedoch von der Position und dem Impuls der zuerst auftreffenden Stelle ab. Daher sollte die Erstellung der Startpunkte von diesem Faktor abhängig gemacht werden [58]. Mithilfe folgender Abbildung werden die Auswirkungen hinsichtlich der Fragmente eines Objektes verdeutlicht.

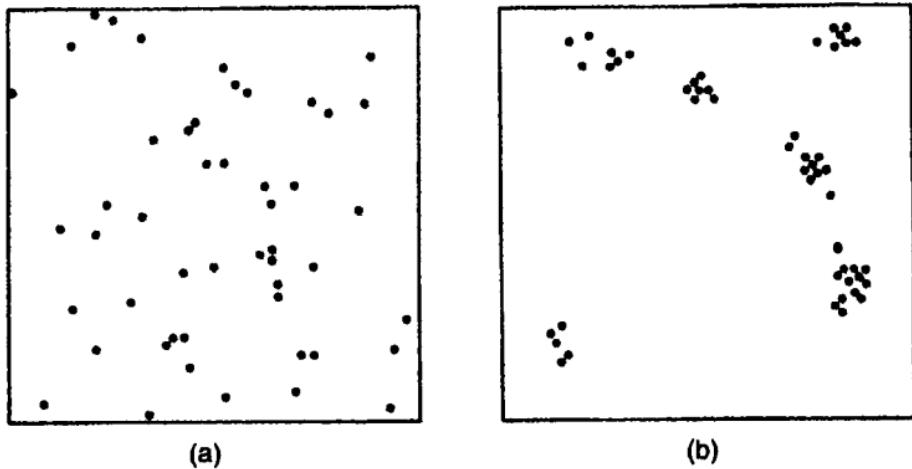


Abbildung 11: Unterschiedliche Punktmengen: a) zufällig generierte Punkte; b) Punktmenge mit Clustern. [62, S. 496]

In Abbildung 11 werden zwei unterschiedliche Punktmengen verglichen. Einerseits eine zufällig generierte Punktmenge, andererseits eine Punktmenge mit mehreren Clustern. Das Voronoi Diagramm der ersten Punktmenge würde eine ausgeglichene Verteilung der Fragmente ergeben. Dies könnte nützlich sein für die Erstellung von Fragmenten die keine genaue Aufprallstelle besitzen. Bei der zweiten Punktmenge ergeben sich aus den Clustern mehrere kleine Fragmente und zwischen den einzelnen Clustern größere. Infolgedessen wäre dies geeignet, um eine Einschlagstelle eines Projektils zu simulieren.

Um Fragmente mit gleicher Größe zu erstellen, müssen die Punkte mit gleichmäßigem Abstand generiert werden. Dies ist mithilfe des Lloyd Algorithmus möglich, bei dem zuerst eine Punktmenge zufällig generiert und anschließend optimiert wird, um gleichmäßige Zellen zu erschaffen. Diese Optimierung der Punktmenge ist ein iterativer Prozess und verschiebt die Punkte in Richtung des Schwerpunktes ihrer zugehörigen Voronoi Zellen und erzeugt dadurch immer gleichmäßigere Punktverteilungen [17]. Den Ablauf des Lloyd Algorithmus wird in Abbildung 12 veranschaulicht.

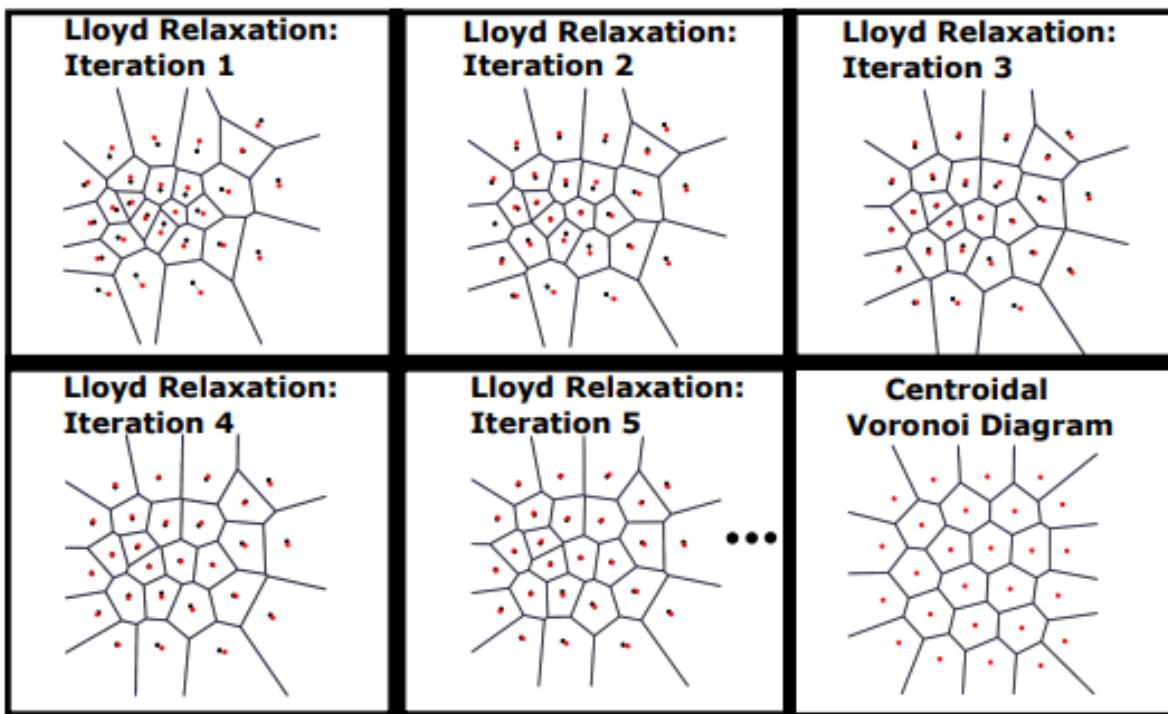


Abbildung 12: Iterationen des Lloyd Algorithmus führen zu einer gleichmäßigeren Punkteverteilung. [42]

2.5 Datenstrukturen

Unabhängig davon, wie ausgereift und leistungsfähig ein Rendering-System ist, muss eine Szene geometrische Objekte enthalten, um Bilder erzeugen zu können. Daher ist die Art und Weise, wie geometrische Objekte für effizientes Rendering und Manipulation dargestellt werden, ein äußerst wichtiges Thema in der Computergrafik. Es gibt zwar viele verschiedene Darstellungsmöglichkeiten von Polyedern, aber die Datenstrukturen Winged-Edge und Half-Edge, auch DCEL genannt, sind die beliebtesten und am häufigsten verwendeten [56].

Eine Datenstruktur organisiert die zu verarbeitenden Daten so, dass die Beziehungen zwischen den Elementen dargestellt werden und die mit den Daten durchzuführenden Operationen unterstützt werden. Beim Aufbau einer Datenstruktur ist vor allem die Effizienz ein wichtiger Aspekt, insbesondere in Bezug auf Zeit und Speicher [57]. Zu den wichtigsten Zielen, die eine räumliche Datenstruktur erfüllen muss, gehören:

- Effizienter Umgang mit großen, dynamisch variierenden Datenmengen in interaktiven Anwendungen.
- Schneller Zugriff auf Objekte, die mithilfe einer genauen Anfrage identifiziert werden.
- Effiziente Ermittlung von benachbarten Elementen.

- Eine gleichmäßige Speichernutzung ohne Fragmentierungen [57].

Eine extreme Form der Darstellung des Voronoi Diagramms wäre, alle erforderlichen Daten explizit anzugeben. Dies würde bedeuten, für jede Voronoi Zelle eine Liste aller angrenzenden Voronoi Punkte und Kanten und eine Liste aller zusammenhängenden Voronoi Zellen. Diese Art der Darstellung würde jedoch zu viel Platz benötigen. Daher sollte eine Datenstruktur ausgewählt werden, die kompakt ist, aber immer noch umfangreich genug ist, um die benötigten Elemente des Diagramms effizient abrufen zu können.

In den folgenden Unterkapiteln werden drei Möglichkeiten vorgestellt, wie das Voronoi Diagramm gespeichert werden kann. Dazu gehören die Delaunay Struktur, welche das Ergebnis der Delaunay Triangulation speichert und durch die Dualität zum Voronoi Diagramm, können die benötigten Informationen abgerufen werden. Bei der Winged-Edge Datenstruktur und der Doubly-Connected Edge List wird das Voronoi Diagramm direkt gespeichert, jedoch in unterschiedlicher Art und Weise.

2.5.1 Delaunay Struktur

Wie bereits erwähnt wird bei der Delaunay Struktur die Delaunay Triangulation gespeichert und nicht das Voronoi Diagramm direkt. Da jedoch die Voronoi Regionen und Kanten aus der Triangulation rekonstruiert werden können, besteht kein Informationsverlust. Einer der Vorteile der Speicherung der Triangulation anstelle der Voronoi Polygone besteht darin, dass die Anzahl der Eckpunkte und Kanten eines Dreiecks immer konstant sind. Dies führt zu einer Struktur von Datensätzen mit einer festen Länge, die die Implementierung vereinfacht und die Effizienz der Speicherung und des Abrufens verbessert. Der Nachteil besteht darin, dass Berechnungen durchgeführt werden müssen, um das Voronoi Diagramm bei Bedarf zu rekonstruieren. Der Aufwand für diese Ableitung ist im schlimmsten Fall $O(n^2)$, da für jedes Dreieck die Nachbarn gefunden werden müssen. Werden jedoch beim Konstruieren des Delaunay Diagramms die angrenzenden Dreiecke gespeichert, kann mit erhöhten Speicherverbrauch die Effizienz gesteigert werden. Weiters kann dies sogar von Vorteil sein, da aus der Punktmenge der Triangulation jede beliebige Form des Voronoi Diagramms rekonstruiert werden kann, je nachdem, welche Form der Benutzer gerade benötigt. Die Datenstruktur für die Delaunay Triangulation kann auf drei Arten implementiert werden, je nachdem, ob die Eckpunkte, die Kanten oder die Dreiecke gespeichert werden. [25]

2.5.2 Winged-Edge

Die Winged-Edge Datenstruktur wurde von Baumgart bereits 1975 vorgestellt [9], ursprünglich für den Bereich der Computer Vision. Sie wurde ausgiebig untersucht und analysiert und ist zu einem beliebten Konzept in der Computergrafik geworden [56].

Die Winged-Edge Datenstruktur kann entweder für die Speicherung der Delaunay Triangulation, oder explizit für das Voronoi Diagramm verwendet werden. Diese Datenstruktur enthält drei Listen: eine für Eckpunkte, eine für Kanten und eine für Dreiecke. Dieser wird in folgendem Quellcode 1 dargestellt.

```
struct Vertex
{
    Vector3 position;
    Edge edge;
}

struct Face
{
    Edge edge;
}

struct Edge
{
    Vertex start, end;
    Face left, right;
    Edge prev_left, prev_right, next_left, next_right;
}
```

Listing 1: Aufbau der Winged-Edge Datenstruktur

Jeder Eckpunkt speichert seine Koordinaten und eine Referenz zu einer beliebigen angrenzenden Kante. Eine Fläche speichert den Index einer beliebigen der Kanten, die die Fläche bilden. Der größte Teil der topologischen Informationen wird innerhalb einer Kante gespeichert. Ein Eintrag einer Kante enthält die Indizes der Endpunkte, Referenzen zu den Flächen auf der linken und rechten Seite, sowie die Vorgänger und Nachfolger der Kante im und gegen den Uhrzeigersinn. Eine typische Kante wird mithilfe von Abbildung 13 veranschaulicht.

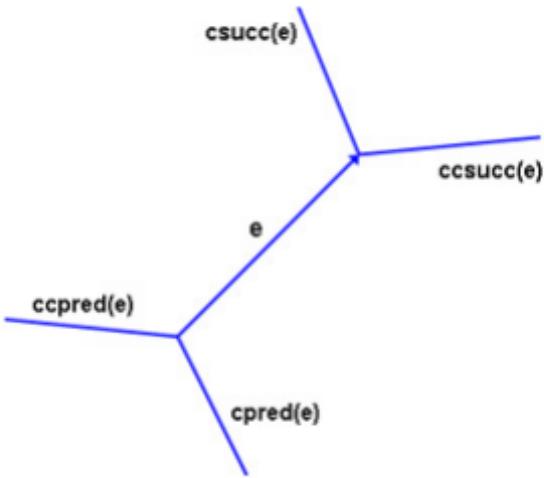


Abbildung 13: Eine Kante der Winged-Edge Datenstruktur [10]

Diese Datenstruktur liefert Information über lokale Beziehungen zwischen den Voronoi Punkten, Voronoi Kanten und Voronoi Polygonen, sodass eine Vielzahl von Informationen relativ einfach abgerufen werden können [62, S. 235].

Die Winged-Edge Datenstruktur ist eine Möglichkeit zur Darstellung von geometrischen Graphen in der Ebene. Um diese Datenstruktur jedoch für das Voronoi Diagramm verwenden zu können, muss das Voronoi Diagramm ein wenig angepasst werden. Das liegt daran, dass sich das Voronoi Diagramm insofern von einem geometrischen Graphen unterscheidet, dass die nach außen führenden Voronoi Kanten unendlich lang sind. Um das Voronoi Diagramm als Graphen betrachten zu können, muss eine geschlossene Form, wie etwa ein Kreis oder ein Rechteck, eingeführt werden, die groß genug ist, um alle Voronoi Punkte zu umgeben. Die Schnittpunkte der unendlichen Kanten und der geschlossenen Form bilden infolgedessen die neuen Endpunkte der Kanten [62, S. 236f].

Die Vorteile der Winged-Edge Datenstruktur sind möglicherweise aufgrund der komplexen Struktur der Kanten nicht sofort offensichtlich. Diese Komplexität der kantenbasierten Struktur trägt jedoch dazu bei, viele topologische Anfragen einfach und effizient durchzuführen, wie beispielsweise das Finden aller angrenzenden Kanten oder Flächen eines Punktes [56].

Sowohl die Winged-Edge als auch die im folgenden Kapitel beschriebene Datenstruktur DCEL sind häufig verwendete Darstellungen für Meshes.

2.5.3 Doubly-Connected Edge List (DCEL)

Eine weitere Möglichkeit das Voronoi Diagramm zu speichern ist mithilfe der Half-Edge Datenstruktur oder auch Doubly-Connected Edge List genannt, die von Muller und Preparata im

Rahmen ihrer Arbeit nach der Suche der Überschneidung zweier konvexer Polyeder präsentierte wurde [53]. Die DCEL ist der Winged-Edge Datenstruktur hinsichtlich des Aufbaus ähnlich, da ebenfalls die Kanten die meisten Informationen über die Struktur des Voronoi Diagramms besitzen. Diese Datenstruktur umfasst Eckpunkte, Flächen und Halbkanten. Wie der Name schon sagt, wird eine Kante in ein paar von Halbkanten mit entgegengesetzter Richtung aufgeteilt. In Abbildung 14 wird ein Ausschnitt eines Halbkanten Meshes gezeigt.

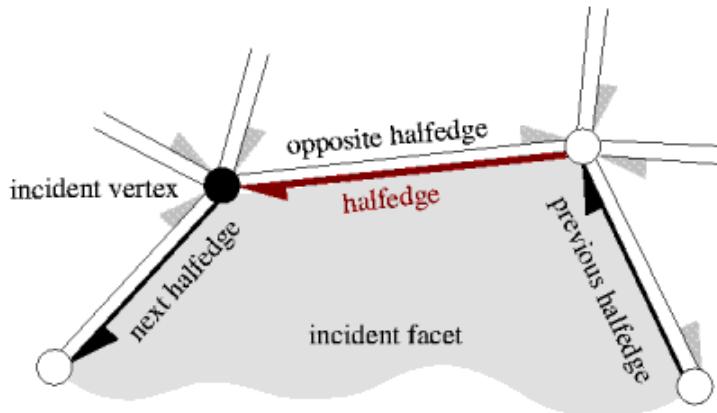


Abbildung 14: Aufbau der Doubly-Connected Edge List [76]

In jeder Halbkante wird eine angrenzende Fläche, entweder der Ausgangspunkt oder der Zielpunkt der Kante und die gegenüberliegende Halbkante gespeichert. Ebenso wird auch für jede Fläche und jeden Eckpunkt eine angrenzende Halbkante gespeichert [76]. Die Halbkanten, die eine Fläche begrenzen, bilden dadurch eine kreisförmig verkettete Liste. Diese Listen können entweder im oder gegen den Uhrzeigersinn um die Fläche verlaufen [64]. Da die Doubly-Connected Edge List eine Erweiterung der Winged-Edge Datenstruktur ist, muss der Graph eine geschlossene Form bilden, um eine gültige DCEL-Struktur zu besitzen. Das bedeutet, dass wieder alle nach außen führenden, unendlichen Kanten geschlossen werden müssen [15, S. 29-33 & S. 157].

Die Ergebnisse für die meisten Adjazenzabfragen, also das Abfragen von benachbarten Elementen, wird direkt in der Datenstruktur für die Kanten, Eckpunkten und Flächen gespeichert. Zum Beispiel können die Flächen oder Eckpunkte, die an eine Halbkante grenzen, leicht gefunden werden, wie in folgendem Ausschnitt veranschaulicht wird.

```

Vertex v1 = halfEdge.vertex;
Vertex v2 = halfEdge.pair.vertex;

Face f1 = halfEdge.face;
Face f2 = halfEdge.pair.face;

```

Listing 2: Abfrage der angrenzenden Eckpunkte und Flächen einer Halbkante

Das in Quellcode 2 verwendete `halfEdge.pair` beschreibt die entgegengesetzte, angrenzende Halbkante. Ein etwas komplexeres Beispiel ist die Iteration über die an eine Fläche angrenzenden Halbkanten. Da die Halbkanten um eine Fläche eine kreisförmig verkette Liste bilden und die Fläche eine Referenz auf eine dieser Halbkanten speichert, kann wie folgt vorgegangen werden [30]:

```

Halfedge edge = face.edge;

do
{
    //use edge for something
    edge = edge.next;
} while (edge != face.edge)

```

Listing 3: Iteration der an einer Fläche angrenzen Halbkanten.

Zusammenfassend lässt sich sagen, dass es mehrere Möglichkeiten gibt das Voronoi Diagramm zu speichern. Das Implementieren der Winged-Edge Datenstruktur oder der DCEL ist komplexer und erfordert mehr Zeit, jedoch erleichtert es die anschließenden benötigten Abfragen für das Voronoi Diagramm. Des Weiteren bietet das Verwenden einer der beiden Datenstrukturen eine effizientere Speichernutzung und möglicherweise auch eine verbesserte Performance der verwendeten Algorithmen.

2.6 Algorithmen zur Berechnung des Voronoi Diagrams

In diesem Kapitel werden verschiedene Möglichkeiten vorgestellt, dass Voronoi Diagramm zu berechnen. Algorithmen, die zum Berechnen des Voronoi Diagramms benötigt werden, haben grundsätzlich zwei Herangehensweisen. Eine Möglichkeit ist, das Voronoi Diagramm direkt zu berechnen, wie es etwa bei der Brute-Force Methode, dem Fortune Algorithmus oder dem Divide & Conquer Algorithmus der Fall ist. Bei der zweiten Herangehensweise wird zuerst die

Delaunay Triangulation von der Punktmenge erstellt und aufgrund der Dualität zum Voronoi Diagramm, kann dies dementsprechend berechnet werden. Beispiele für Algorithmen, die diesen Ansatz verwenden, sind der Bowyer-Watson Algorithmus oder der Flip Algorithmus. Unabhängig davon, welche dieser beiden Ansätze ein Algorithmus verwendet, unterscheiden sich diese zusätzlich darin, ob sie nur im zweidimensionalen Fall oder in beliebige Dimensionen verwendet werden können.

2.6.1 Brute-Force

Der Brute-Force Ansatz ist am einfachsten zu implementieren und eignet sich gut für die frühen Phasen der Entwicklung. Vorteil dieses Ansatzes ist, dass rasch ein erster Prototyp implementiert werden kann, um einen Einstieg in das Thema zu bekommen. Ein Nachteil dieses Algorithmus ist jedoch, dass die Performanz nicht berücksichtigt wird und bei größeren Punktemengen die Zeit zur Berechnung des Voronoi Diagramms rapide ansteigt.

Der Aufbau des Brute-Force Ansatzes setzt sich aus folgenden Schritten zusammen [73]:

1. Wähle einen Startpunkt P aus der Punktmenge.
2. Berechne die Distanz von P zu jedem anderen Punkt.
3. Wähle den Punkt Q , der P am nächsten liegt.
4. Berechne die Mittelsenkrechte zwischen den Punkten P und Q .
5. Entferne alle Punkte auf der anderen Seite der Mittelsenkrechten, die weiter entfernt sind von dieser als Q .
6. Entferne Q
7. Wiederhole Schritte 3-6, bis keine Punkte mehr übrig sind.
8. Berechne die Schnittpunkte der Mittelsenkrechten.
9. Verbinde die Schnittpunkte im oder gegen den Uhrzeigersinn, um ein Polygon zu erstellen.

Diese Schritte werden in Abbildung 15 verdeutlicht.

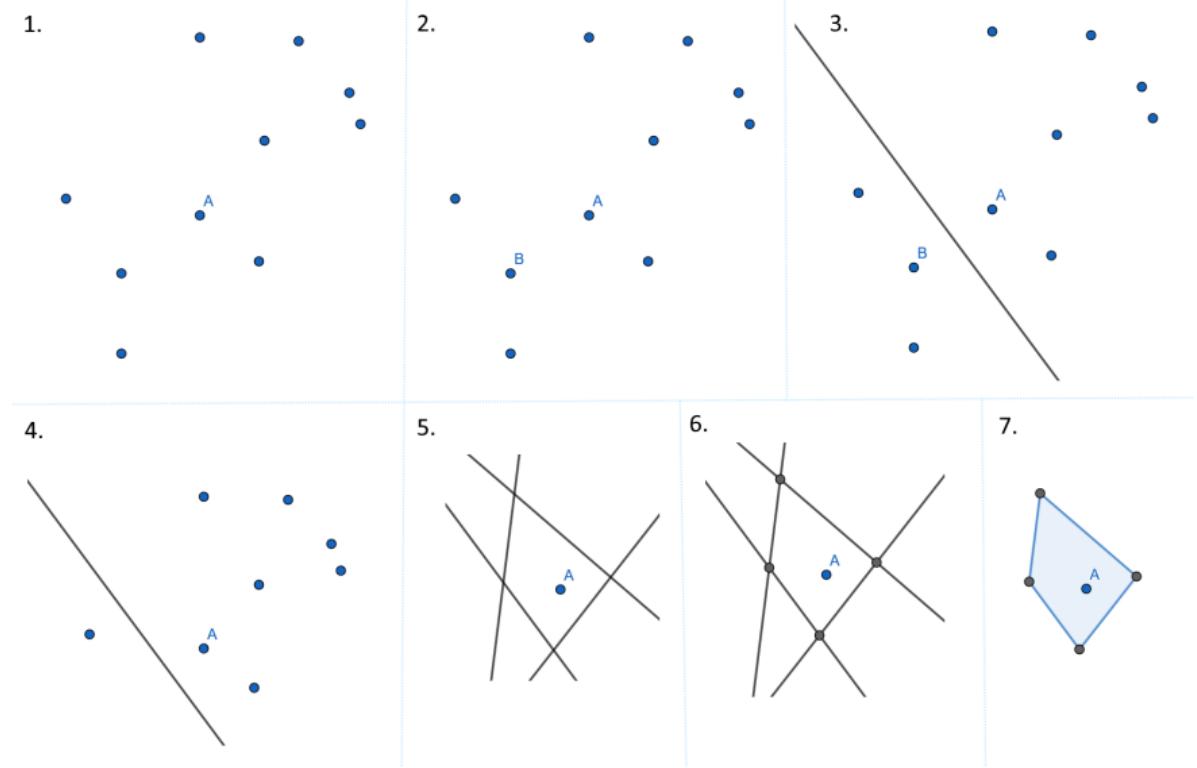


Abbildung 15: Schritte für die Berechnung einer einzelnen Zelle des Voronoi Diagramms mithilfe eines Brute Force Ansatzes [73]

Das Ergebnis der eben beschriebenen Schritte ist eine einzelne Voronoi Zelle, das bedeutet dieses Verfahren muss für alle Punkte wiederholt werden, um das gesamte Voronoi Diagramm zu erhalten. Eine Voronoi Zelle kann potenziell $n - 1$ Kanten besitzen, wobei n die Anzahl der Punkte darstellt. Das bedeutet, dass die Komplexität für die Berechnung jeder einzelnen Zelle nach O-Notation $O(n^2)$ beträgt. Infolgedessen ergibt sich für die Berechnung des gesamten Diagramms eine Komplexität von $O(n^3)$ [73]. Daraus ergibt sich, dass dieser Ansatz nicht für Echtzeitanwendungen verwendet werden kann.

2.6.2 Fortune Algorithmus

Um nun das Voronoi Diagramm bei größeren Punktmengen zu berechnen oder um es für Echtzeitanwendungen verwenden zu können, wird ein effizienterer Algorithmus benötigt. Eine Möglichkeit ist der Fortune Algorithmus. Dieser besitzt eine Zeitkomplexität von $O(n \log n)$ und einen Speicheraufwand von $O(n)$ [15, S. 151f]. Es stellt sich die Frage, ob diese Zeitkomplexität noch verbessert werden kann, jedoch ist dies nicht der Fall, wie Shamos und Hoey [69] bereits bewiesen haben, dass jeder Algorithmus zur Berechnung des Voronoi Diagramms im schlimmsten Fall $O(n \log n)$ Zeit benötigt. Daraus folgt, dass der Fortune Algorithmus optimal ist.

Der Fortune Algorithmus wurde von Steven Fortune in Jahr 1986 im Rahmen seiner Arbeit *A Sweep-line Algorithm for Voronoi Diagramms* vorgestellt und basiert auf der Sweep-line Technik [22]. Bei dieser Technik wird eine horizontale Linie entlang der Ebene verschoben und hält an, wenn ein relevanter Punkt erreicht wird. Die direkte Berechnung des Voronoi Diagramms nur mithilfe der Sweep-line ist schwierig, da die Voronoi Zelle eines Punktes möglicherweise von der Sweep-line durchschnitten wird, lange bevor der zugehörige Punkt selbst erreicht worden ist. Deshalb wird eine weitere Linie hinzugefügt, die sogenannte Beachline. Diese besteht aus Parabeln, die definiert sind durch die Punkte, die die Sweep-line bereits geschnitten hat und durch die Sweep-line selbst. Dies wird in Abbildung 16 veranschaulicht.

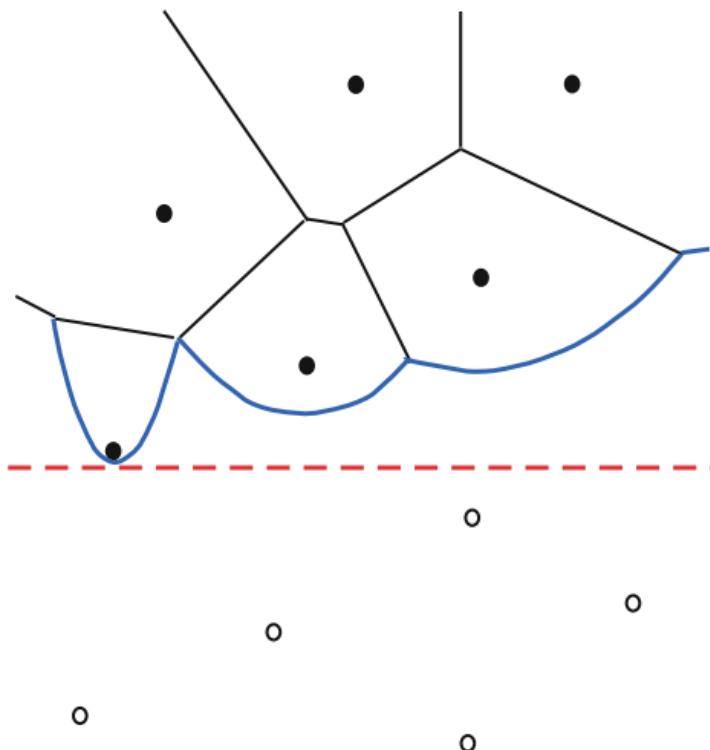


Abbildung 16: Aufbau des Fortune Algorithmus. Die rote strichlierte Line ist die Sweep-line, die sich entlang der Ebene bewegt und die Beachline setzt sich aus den blauen Parabeln zusammen. Gefüllte Kreise sind Punkte, die bereits von der Sweep-line erreicht worden sind, während nicht gefüllte Kreise noch hinzugefügt werden müssen. Die schwarzen Linien sind bereits Kanten des Voronoi Diagramms. [35]

Im Laufe des Algorithmus treten zwei Arten von Ereignissen auf: Punkttereignisse und Kreisereignisse. Ersteres tritt auf, wenn die Sweep-line einen Punkt der Punktmenge erreicht. In diesem Fall wird der Punkt dem Voronoi Diagramm hinzugefügt und eine neue Parabel ausgehend von diesem Punkt wird zu der Beachline hinzugefügt. Alle Punkttereignisse sind bereits im Voraus bekannt, da jeder Punkt der Punktmenge einem Ereignis entspricht [73].

Ein Kreisereignis tritt ein, wenn sich zwei Parabeln, die durch eine andere Parabel getrennt

waren, schneiden. Wenn dieses Ereignis eintritt, schneiden sich zwei Kanten des Voronoi Diagramms und daraus folgt, dass der Schnittpunkt zum Voronoi Diagramm hinzugefügt wird und eine neue Kante an diesem Schnittpunkt beginnt (siehe Abbildung 17).

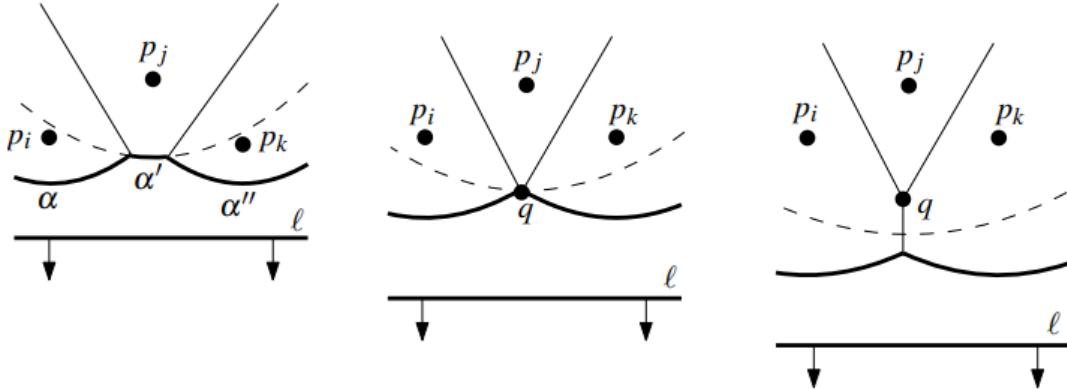


Abbildung 17: Ablauf eines Kreisereignisses. Eine Parabel verschwindet von der Beachline und bildet eine neue Kante des Voronoi Diagramms. [15, S. 154]

Obwohl der Fortune Algorithmus optimal in Hinsicht auf Zeit und Speicherverbrauch, kann dieser nur im zweidimensionalen Fall verwendet werden. Um ein Voronoi Diagramm in 3D erstellen zu können, muss ein anderer Algorithmus verwendet werden.

2.6.3 Divide & Conquer rekursiv

Eine weitere Methode, mit der die Delaunay Triangulation oder das Voronoi Diagramm direkt mit $O(n \log n)$ Aufwand berechnet werden kann ist mithilfe einer Divide & Conquer Strategie, die von Shamos und Hoey [68] präsentiert wurde. Das Grundprinzip der Divide & Conquer Strategie basiert darauf, dass die Punktmenge rekursiv in kleinere Teilmengen unterteilt wird und das Voronoi Diagramm von diesen kleineren Teilmengen gebildet wird. Der wesentliche Teil dieses Algorithmus ist jedoch der darauffolgende Schritt: das Zusammenführen der Voronoi Diagramme der kleineren Teilmengen [62, S. 252].

Um beide Voronoi Diagramme zusammenzuführen, müssen neue Kanten und Eckpunkte erzeugt werden und überflüssige Elemente aus den Diagrammen entfernt werden. Die neuen Kanten und Eckpunkte werden berechnet, indem eine polygonale Trennlinie zwischen den beiden Teilen erzeugt wird und jene Kanten der Teildiagramme abgeschnitten werden, die über diese Trennlinie hinausgehen [62, S. 253f]. Diese polygonale Linie wird von unten nach oben konstruiert. Um die unterste Kante zu finden, wird von beiden Diagrammen die Konvexe Hülle gebildet und die Mittelsenkrechte der untersten Punkte der Diagramme gebildet. Anschließend wird ein Punkt betrachtet, welcher sich entlang der Mittelsenkrechten von unten nach oben bewegt, bis die beiden obersten Punkte erreicht worden sind. Das Ergebnis des Divide & Conquer Algorithmus wird in Abbildung 18 gezeigt.

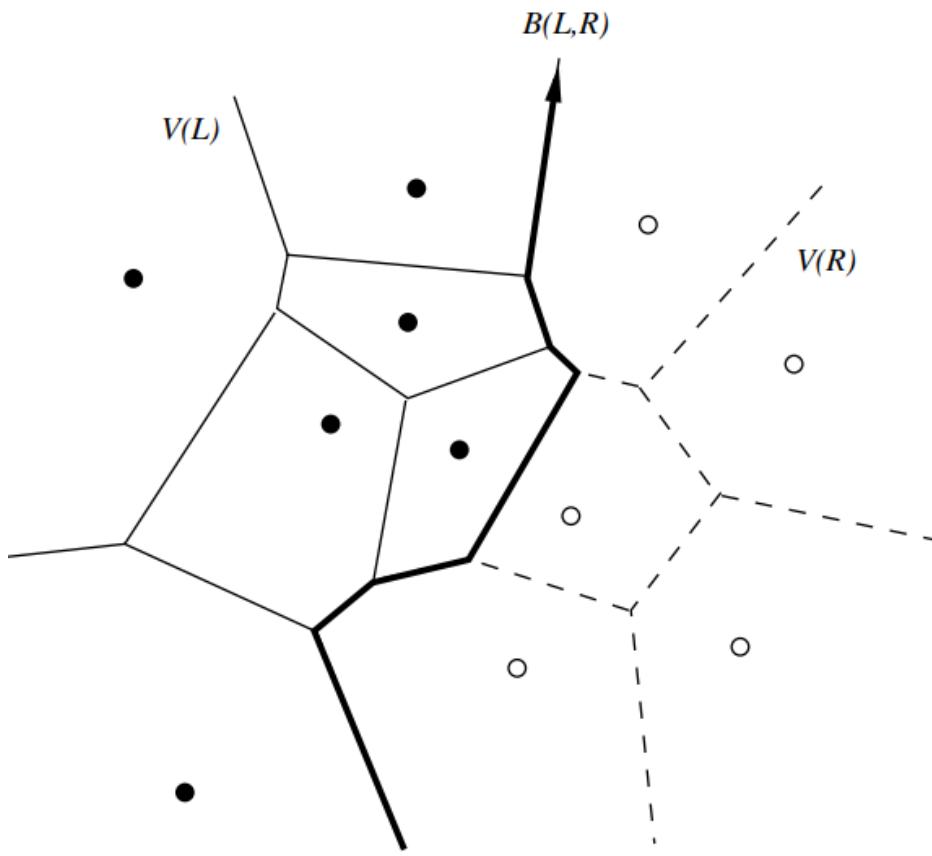


Abbildung 18: Endprodukt des Divide & Conquer Algorithmus. Die polygonale Trennlinie bildet die neuen Kanten des gesamten Voronoi Diagramms. [8]

Es wurde bereits mehrfach gezeigt, dass eine derartige Divide & Conquer Methode im schlimmsten Fall $O(n \log n)$ Zeit benötigt [8] [62, S. 251ff] [68], doch Ohya et al [61] haben gezeigt, dass dies nicht nur im schlimmsten Fall ist, sondern auch im Durchschnitt diese Zeit benötigt. Daher ist diese Methode im Durchschnitt nicht optimal.

2.6.4 Inkrementeller Flipping Algorithmus

Bei einem flipbasierten Algorithmus wird im Gegensatz zu den bereits erwähnten Methoden das Voronoi Diagramm nicht direkt berechnet, sondern zuerst die Delaunay Triangulation. Diese kann bei einer flipbasierten Methode auf zwei Arten erstellt werden. Eine Möglichkeit ist zuerst eine beliebige Triangulation der Punktmenge zu konstruieren und anschließend einzelne Subtriangulationen durch andere Triangulationen zu ersetzen, bis die Eigenschaften der Delaunay Triangulation erfüllt sind [29]. Dieses Ersetzen wird auch als Flip bezeichnet und ist ein wesentlicher Bestandteil dieser Methode. Die Gesamtlaufzeit dieses Algorithmus ist $O(n^2)$, da es $O(n)$ Dreiecke gibt, die gegeneinander überprüft werden müssen [39].

Eine andere Möglichkeit ist mit einem Hilfstetraeder zu beginnen, das alle Punkte der Punktmenge enthält. Anschließend werden die Punkte einer nach dem anderen in die Triangulation eingefügt und in jedem Schritt wird ein Tetraeder gefunden, das den einzufügenden Punkt enthält. Dann wird der Punkt mit dem Tetraeder verbunden und durch eine Folge von Flips wird eine ordnungsgemäße Delaunay Triangulation konstruiert [87].

Wie bereits erwähnt wird der Vorgang, bei dem eine Triangulation durch eine andere ersetzt wird als Flip bezeichnet. Flips werden durch die Anzahl der Elemente vor und nach dem Flip benannt. In 3D existieren grundsätzlich vier Typen von Flips:

- Flip14 - ersetzt ein Tetraeder durch vier und fügt dadurch einen neuen Punkt in die Triangulation ein.
- Flip41 - ersetzt vier Tetraeder durch eins und entfernt dabei einen Punkt aus der Triangulation.
- Flip23 - ersetzt zwei Tetraeder durch drei. Entfernt eine innere Fläche, die von den Tetraedern geteilt wird und ersetzt sie durch drei neue Innenflächen.
- Flip32 - ersetzt drei Tetraeder durch zwei. Entfernt drei innere Flächen, die von den Tetraedern geteilt wird und ersetzt sie durch eine neue Innenfläche. [87]

Diese Arten von Flips werden in Abbildung 19 dargestellt.

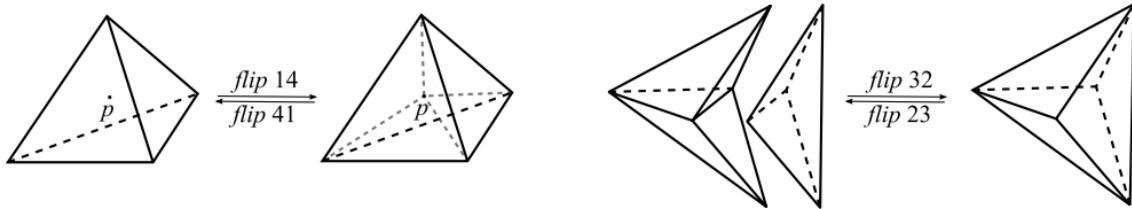


Abbildung 19: Grundlegende Typen von Flips. [87]

Offensichtlich ist der Flip14 die Umkehrung des Flips41 und der Flip23 die Umkehrung des Flips32. Bei bestimmten Fällen können die eben erwähnten Flips flache Tetraeder erzeugen, was in den meisten Fällen unerwünscht ist. Um dieses Problem zu lösen, gibt es noch weitere Arten von Flips, die zum Einsatz kommen, wenn beispielsweise der einzufügende Punkt auf einer Fläche oder Kante liegt. Die Zeitkomplexität dieses Algorithmus ist im schlimmsten Fall $O(n^2)$, jedoch beträgt sie im Durchschnitt $O(n \log n)$ [87].

2.6.5 Bowyer-Watson Algorithmus

Ein weiterer inkrementeller Algorithmus ist der Bowyer-Watson Algorithmus. Dieser wurde von Adrian Bowyer und David F. Watson unabhängig voneinander entwickelt und in derselben Ausgabe von *The Computer Journal* veröffentlicht [12] [84]. Der Algorithmus funktioniert in d -Dimensionen und die erwartete Zeitkomplexität beträgt $O(n^{(2d-1)/d})$ [87]. Der Einfachheit halber wird der Algorithmus zuerst in 2D beschrieben und anschließend werden die benötigten Änderungen erläutert, um den Algorithmus auf 3D zu erweitern.

Zu Beginn wird auf der gleichen Weise wie beim inkrementellen Flipping Algorithmus ein Hilfsdreieck konstruiert, welches alle Punkte der Punktmenge enthält. Dieses Dreieck wird auch als Superdreieck (engl. *super triangle*) bezeichnet. Danach werden die Punkte, einer nach dem anderen, in die Triangulation eingefügt. Jedes Mal, wenn ein Punkt eingefügt wird, werden alle Dreiecke gesucht, deren Umkreis den einzufügenden Punkt enthalten, auch schlechte Dreiecke (engl. *bad triangle*) genannt (siehe Abbildung 20 (a)) und werden aus der Triangulation entfernt. Dadurch entsteht ein Loch innerhalb der Triangulation, wie in Abbildung 20 (b) zu sehen ist. Dieses Loch wird unter Verwendung des neuen Punktes neu trianguliert, indem jede Kante des Lochs mit dem neuen Punkt verbunden wird und dadurch ein neues Dreieck der Triangulation bildet (Abbildung 20 (c)).

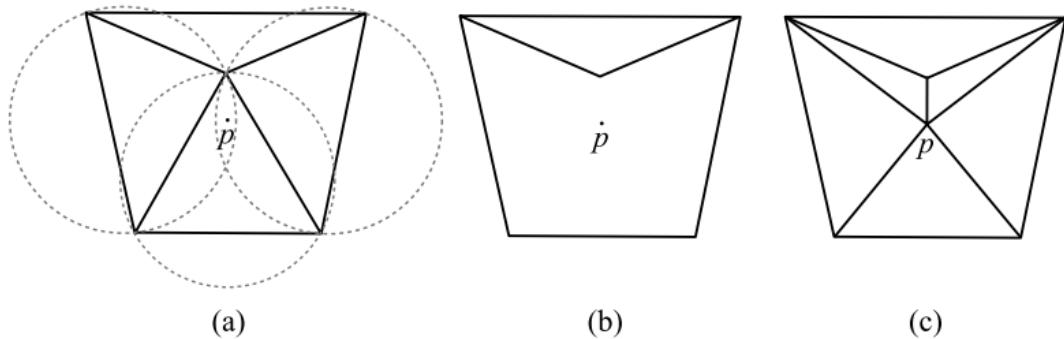


Abbildung 20: Ablauf beim Einfügen eines Punktes beim Bowyer-Watson Algorithmus. [87]

Nachdem alle Punkte eingefügt und trianguliert wurden, werden zum Schluss noch alle Dreiecke entfernt, die einen gemeinsamen Eckpunkt mit dem Superdreieck haben. Dieser Schritt kann ignoriert werden, wenn das Superdreieck zukünftige Berechnungen nicht beeinträchtigt oder wenn bekannt ist, dass neue Punkte hinzugefügt werden. Wenn diese neuen Punkte zur bereits bestehenden Triangulation hinzugefügt werden, wird das Superdreieck nicht benötigt, jedoch ist es erforderlich, wenn Punkte außerhalb der triangulierten Bereichs hinzugefügt werden [73].

Um diesen Algorithmus nun auf 3D zu erweitern, werden Dreiecke durch Tetraeder, Umkreise durch Umkugeln und Kanten durch Flächen ersetzt.

Ein wesentlicher Bestandteil des Algorithmus ist das Lokalisieren der Dreiecke, deren Umkreis den einzufügenden Punkt enthalten. Um eine schnellere Laufzeit zu erhalten, kann die Lokalisierung der schlechten Dreiecke beziehungsweise Tetraeder verbessert werden, sobald das erste gefunden ist. Der oben erwähnte Ablauf iteriert durch alle Dreiecke, um die schlechten Dreiecke zu finden. Dies kann erweitert werden, indem jedes Dreieck eine Referenz auf seine benachbarten Dreiecke besitzt. Nachdem das erste schlechte Dreieck gefunden wurde, wird die Iteration gestoppt und die benachbarten Dreiecke können durchsucht und überprüft werden, um alle schlechten Dreiecke zu finden [73]. Dies schränkt den Suchbereich auf einen kleineren Teil des Diagramms ein.

Dieser Algorithmus wird für den praktischen Teil für die Berechnung der Delaunay Triangulation verwendet. Falls erforderlich, wird das Voronoi Diagramm daraus abgeleitet.

2.7 Resümee

Das Voronoi Diagramm ist ein wichtiges Modell in der Geometrie und Computergrafik. Zusätzlich kommt es auch in anderen Bereichen zum Einsatz, wie etwa der Architektur, der Gebäudeplanung oder in der Biologie zum Modellieren von Zellwachstum. In dieser Arbeit wird es für die Erstellung von Fragmenten bei einer Zerstörung eines Objektes verwendet. Daher wurden in diesem Kapitel die Definition des Voronoi Diagramms und dessen Einsatzgebiete dargestellt.

Eine besondere Eigenschaft, die das Voronoi Diagramm besitzt, ist die Dualität zur Delaunay Triangulation. Aufgrund dieser Dualität ist es möglich aus der Kenntnis der einen Struktur die andere zu bekommen. Sowohl die Delaunay Triangulation als auch das Voronoi Diagramm werden für den praktischen Teil dieser Arbeit verwendet. Deshalb wird für die Berechnung dieser Strukturen der Bowyer-Watson Algorithmus verwendet, da dieser zuerst die Delaunay Triangulation berechnet und das Voronoi Diagramm wird, falls erforderlich, abgeleitet. Mehr Details zu der Implementierung werden in Kapitel 4 gezeigt.

3 State of the Art

Dieses Kapitel gibt einen Überblick über verschiedene Ursachen, wie eine Fragmentierung von Objekten in einem Videospiel entstehen kann. Dahingehend wird erläutert, welche Eigenschaften ein Algorithmus zum Zerstören von Objekten berücksichtigen sollte, um der spielenden Person ein zufriedenstellendes und realistisches Ergebnis zu liefern. Anschließend wird gezeigt, welche Methoden bisher in Videospielen eingesetzt wurden, wie diese aufgebaut sind und welche Vor- und Nachteile sie bringen.

3.1 Fragmentierung in Videospielen

Im Zuge der Veröffentlichung von Computerspielen wurde eine Industrie mit zunehmend interaktiven Inhalten und neuen Spielerlebnissen hervorgebracht. Neue Computerspiele werden in regelmäßigen Abständen veröffentlicht und mit ihren jeweiligen besonderen Merkmalen beworben [86]. Diese Merkmale können rein visuelle Effekte sein, wie etwa realistische Wasser- oder Rauchsimulationen. Des Weiteren können Effekte durch Interaktionen der spielenden Person ausgelöst werden, wie zum Beispiel die Zerstörung von Objekten. Diese Zerstörungseffekte wie unter anderem explodierende Gebäude, zerspringendes Glas oder die Zerstörung von Objekten in einer Szene, sind in heutigen Computerspielen immer häufiger zu sehen und tragen erheblich zur immersiven Erfahrung bei [50]. Mittlerweile existiert bereits ein eigener Steam-Tag auf der Vertriebsplattform Steam [82], welcher Spiele markiert, die Elemente von Fragmentierungen und Zerstörungen besitzen, unabhängig davon, ob das Spiel dieses Feature als Grundmechanik verwendet oder um beeindruckende grafische Effekte in einem Spiel zu erzielen. [36]. Jedoch stellt das Zerstören von Objekten in Videospielen keine Neuheit dar. Bereits bei früheren Arcade Spielen, wie Space Invaders, gab es Objekte, die dem der Spielenden Deckung gaben, welche durch die Gegner zerstört wurden. Eine der ersten Spielserien, welche nicht nur zerstörbare Objekte, sondern sogar eine in Echtzeit zerstörbare Umgebung zu einem bedeutenden Teil der Spielmechanik machten, war *Red Faction* von dem Entwicklerstudio *Deep Silver Volition* [16]. Der erste Teil der Serie, *Red Faction*, implementierte eine primitive Form der Deformation der Umgebung mithilfe der Geometry Modification Technology (Geo-mod), welche Teil der gleichnamigen Engine war [3]. Die *Geo-mod* Technologie bewirkte zwar eine verformbare Welt, jedoch nicht ohne gewisse Einschränkungen. Alle Löcher, die durch Zerstörung entstanden, hatten ungefähr die gleiche Form und Größe, unabhängig davon, wie die Umgebung getroffen wurde. Darüber hinaus war die Anzahl der Löcher stark begrenzt und nach kurzer Zeit

würden keine neuen Löcher mehr entstehen [45]. Nichtsdestotrotz war bereits der erste Teil der *Red Faction* Reihe ein bedeutender Schritt in der Entwicklung von zerstörbaren Landschaften und Gebäuden.

Der Vergleich zwischen den ersten beiden Spielen der Serie und dem dritten, *Red Faction: Guerilla*, zeigt den umfassenden Einfluss auf die Spieleentwicklung, den ein Generationenwechsel in der Hardware mit sich bringen kann. *Deep Silver Volition* war in der Lage, für *Guerilla* das *Geo-mod 2.0*-System zu entwickeln, welches die verbesserte Hardware der neuen Konsolen-generation nutzt. Durch diese neue und verbesserte Engine wird der spielenden Person ein weitaus größeres Spektrum an Zerstörungen ermöglicht. [70]

Die Entwicklungszeit von *Red Faction: Guerilla* betrug fünf Jahre, wobei die Entwickler*innen vier Jahre davon für die Implementierung des Systems für die Zerstörung benötigten. Dadurch, dass dieses System vollkommen physikalisch basiert ist, mussten die Entwickler*innen darauf achten, dass die Gebäude und Strukturen architektonisch korrekt gebaut waren, da diese in einer der ersten Iterationen der Entwicklung in sich selbst zusammengefallen sind. Infolgedessen ist es jedoch in *Guerilla* möglich, alle von Menschenhand geschaffenen Objekte, jedes Gebäude, jede Struktur und jede Mauer mit einem immens hohen Grad an Realismus zu zerstören. [51]

Ein weiteres Merkmal, welches dieses realistische System untermauert, ist, dass Objekte nicht immer in der gleichen Art und Weise zerbrechen, sondern zersplittern, verbiegen oder kleine Stücke von diesen abfallen. Die Zerstörung eines Objekts bedingt nicht immer gezwungenermaßen, dass dieses in kleine Stücke zerbricht. Dies ist in der nachstehenden Abbildung 21 ersichtlich, wo ein Turm beinahe vollkommen unversehrt stecken bleibt. [3]



Abbildung 21: Turm bleibt nach einer Zerstörung hängen und zerfällt nicht in kleinere Stücke [3].

Eine zusätzliche Besonderheit, die dieses Spiel aufgrund des physikalisch basierten Systems

besitzt, ist, dass Bauwerke, wie etwa Brücken, einstürzen, wenn genügend Stützen weggeschlagen wurden. Dadurch, dass dieses Zerstörungssystem auch vollkommen im Multiplayer-Modus vorhanden ist, kann diese Eigenschaft taktisch vorteilhaft genutzt werden.

Gründe für das Zerstören von Objekten in Videospielen sind zahlreich. Dazu gehören beispielsweise das Zerstören einer Kiste, um dessen Inhalt zu erhalten, das Zerspringen einer Glasscheibe durch den Einschlag einer Kugel oder das Zerstören der gegnerischen Deckung. Diese Effekte sollten jedoch einen ausreichenden Grad an Realismus besitzen, der durch Partikeleffekte verstärkt werden kann. Bevor allerdings ein Zerstörungssystem erfolgreich in ein Videospiel integriert werden kann, gibt es eine Reihe an strengen Kriterien, die erfüllt werden müssen, um einen reibungslosen Ablauf zu gewährleisten: Die Performance des Systems muss effizient sein, um Spielinhalte zuverlässig in ein Level integrieren zu können, ohne dass es zu gelegentlichen Einbrüchen der Bildrate oder Verzögerungen in der Tastatur- bzw. Controlereingabe kommt. Weiters darf das Spiel nicht ausfallen, denn selbst einmalige oder seltene Ausfälle können die Spielerfahrung schnell negativ beeinflussen. Dazu kommt noch, dass diese Kriterien unabhängig davon gelten sollen, wie der Anwender mit dem System interagiert. Es sollte nicht davon ausgegangen werden, dass der Benutzer in einer vorhersehbaren Weise handelt. Zusätzlich kann die Speichernutzung ein weiteres Problem ergeben, insbesondere bei Konsolenspielen. Zuletzt sollten im Zerstörungssystem Funktionen und Parameter bereitgestellt werden, die die Eigenschaften einer Zerstörung kontrollieren. So können vielseitige Ergebnisse erschaffen und die Entwicklung des Spiels vereinfacht und beschleunigt werden. [63]

Einige der Herausforderungen bei der Fragmentierung von Objekten haben mit der Art und Weise zu tun, wie 3D-Modelle aufgebaut sind. Die Form eines Objektes wird durch ein sogenanntes Mesh dargestellt, welches eine Sammlung von Punkten, Kanten und Flächen ist. Ebendieses wird in Abbildung 22 gezeigt.

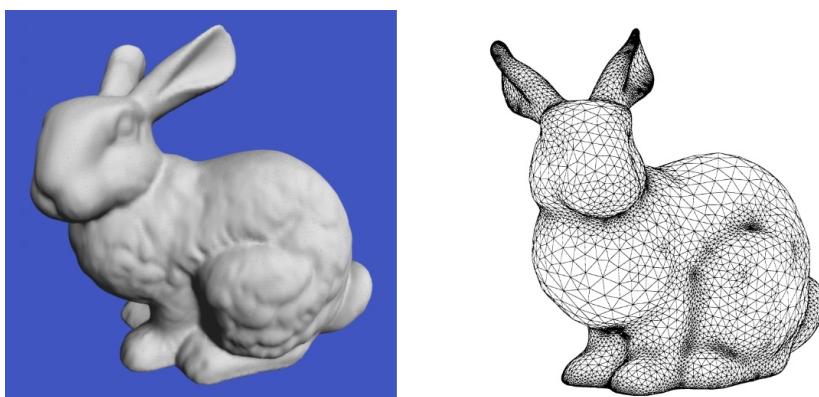


Abbildung 22: Links: Das Stanford-Bunny, eines der am häufigsten verwendeten Testmodelle in der Computergrafik [71].
Rechts: Das darunterliegende Mesh [24].

Ein Problem, welches 3D-Meshes mit sich bringen, ist jenes, dass diese keine Volumina, son-

dern hohle Schalen sind. Eine Herausforderung, die daraus entsteht, ist das Entstehen von Löchern beim Entfernen eines Teils des Meshes. Um diese Löcher zu füllen, müssen neue Dreiecke erstellt und zum Mesh hinzugefügt werden [28].

Es existieren bereits verschiedene Methoden, um Objekte zu zerstören. Eine sehr bekannte Technik ist, dass Objekte bereits im Vorfeld von einem 3D-Artist oder mithilfe einer bestimmten Software gebrochen werden. Wenn während des Spielens die Zerstörung eintritt, werden die ursprünglichen Modelle durch ihre vorgebrochenen Gegenstücke ersetzt [50]. Die eben genannte Methode ist nur eine von mehreren Techniken, welche in Kapitel 3.2 und 3.3 genauer beschrieben werden. Eine Gegenüberstellung der Vor- und Nachteile dieser Techniken erfolgt ebenso in diesen Kapiteln.

3.2 Geometriebasierte Methoden

Die Zerstörung von Objekten ist ein aktives Forschungsgebiet der Computergrafik, das sowohl in Videospielen, als auch bei Simulationen Anwendung findet. In der Literatur wird oft zwischen *brittle*(spröde, zerbrechlich) und *ductile*(dehnbar, formbar) Fragmentierung unterschieden. Das Adjektiv spröde beschreibt in dem Fall steife und feste Materialien, die sich in der Regel nur minimal Verformen bevor sie brechen, wie beispielsweise Glas, Keramik, Beton oder Stein. Ein Merkmal von dieser Art der Fragmentierung ist, dass sich die Risse sehr schnell ausbreiten und vom menschlichen Auge nicht wahrgenommen werden können. Im Gegensatz dazu beinhaltet eine Fragmentierung von dehbaren Materialien elastische Verformungen des Objekts und Risse breiten sich auf unterschiedliche Weise aus und viel langsamer als bei spröden Materialien. Die unterschiedlichen Ergebnisse einer Fragmentierung von spröden und dehbaren Objekten wird in Abbildung 23 gezeigt. Aufgrund der unterschiedlichen Eigenschaften der beiden Arten werden unterschiedliche Methoden für die jeweilige Anwendungen verwendet. [26]

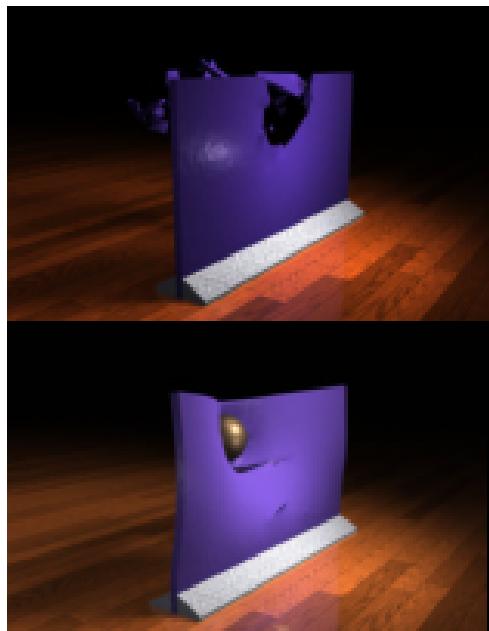


Abbildung 23: Oben: Ergebnis eines Bruchs eines spröden Materials

Unten: Ergebnis eines dehnbaren Materials, erkennbar an der Verformung des Objektes [60]

Diese Arbeit bezieht sich vorwiegend mit der Zerstörung von spröden Materialien in Echtzeit. In den folgenden Kapiteln werden bereits vorhandene Arbeiten und deren Methoden vorgestellt, welche sich beispielsweise in der grundlegenden Funktion, der Performance und den Gebieten, in denen diese angewandt werden, unterscheiden.

Geometriebasierte Methoden, oder auch Prozedurale Methoden genannt, können Muster von Rissen und Brüchen erzeugen, die visuell glaubwürdig sind, ohne sich auf den tatsächlichen physikalischen Vorgang oder dessen genaue Simulation zu stützen. Physikalisch basierte Simulationen gelten oft als rechenaufwändig und bieten nicht ausreichend Kontrolle über die Ausbreitung der Risse. Im Vergleich dazu beruhen die prozeduralen Methoden auf Tools, welche eine umfangreiche Kontrolle über die erhaltenen Rissmuster, sowie über die Größe Form der Fragmente bieten. Diese Methoden können noch weiter unterteilt werden in zwei Unterkategorien: das Austauschen des Objektes durch ein vorgebrochenes Gegenstück und das dynamische Zerstören. [52]

3.2.1 Austausch des Objektes

Wie bereits erwähnt ist in Videospielen eine oft genutzte Technik das Austauschen des 3D-Objektes zur Laufzeit durch das vorgebrochene Gegenstück. Diese statische Methode wurde aufgrund ihrer Einfachheit bereits in vielen beliebten Spielen erfolgreich eingesetzt [50].

Diese Technik überzeugt vor allem durch ihre Performance, da das darunterliegende Mesh

nicht geändert wird, sondern das Objekt einfach nur ausgetauscht wird. Dafür muss zwar wie bereits erwähnt ein vorgebrochenes Gegenstück zur Entwicklungszeit erstellt werden, welches von einem 3D-Artist manuell oder mithilfe eines von mehreren Tools erstellt werden kann, aber das Ersetzen des Models zur Laufzeit ist keine aufwändige Operation. Obwohl der Austausch des Objektes für den herkömmlichen User nicht erkennbar ist, kann es zusätzlich mithilfe von Partikeleffekten wie beispielsweise Staub oder Rauch versteckt werden.

Beispiele für Tools zum Erstellen von vorgebrochenen Modellen sind zum Beispiel das *Cell Fracture Add-On* in *Blender* [11] oder das *Chaos Destruction System* [81] welches in der *Unreal Engine* bereits integriert ist. Unabhängig davon, ob das vorgebrochene Objekt manuell oder mit einem Tool erstellt wird, besitzen beide Möglichkeiten viel Kontrolle über die Zerstörung des Models, wie beispielsweise die Größe und die Anzahl der Fragmente oder der anzuwendende Algorithmus, wie in Abbildung 24 dargestellt wird [14].

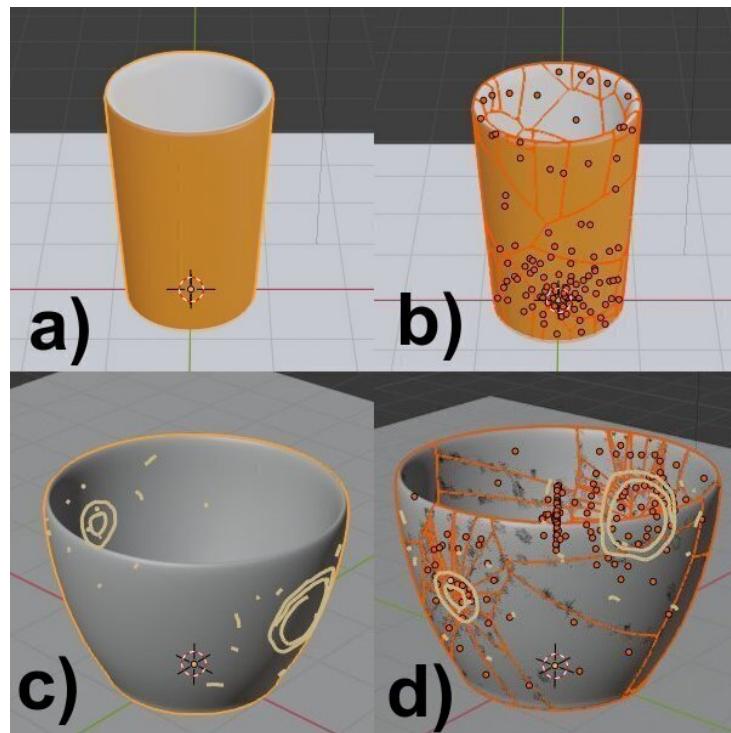


Abbildung 24: a) Model eines Bechers
 b) Das gebrochene Model des Bechers mit Standardeinstellungen des *Cell Fracture Add-Ons*.
 c) Model einer Trinkschale
 d) Das gebrochene Gegenstück, welches mithilfe des *Annotation Pencil* erstellt wurde. Dabei werden durch das Zeichnen von Punkten und Kreisen die Fragmente erstellt. [66]

Die meisten dieser Muster der gebrochenen Objekte beruhen jedoch üblicherweise auf einer Partitionierung mithilfe des Voronoi Diagramms, welches in Kapitel 2 erklärt wurde [52]. Diese weit verbreitete Technik ist zwar vom rechnerischen Aufwand gering, hat aber den Nachteil,

dass das Bruchmuster nicht mit der Einschlagstelle übereinstimmt und dass die entstandenen Fragmente immer gleich sind und nicht weiter fragmentiert werden können [50]. Dies kann aber durch mehrere Methoden verbessert werden: Zunächst ist es generell sinnvoll die Geschwindigkeit des eintreffenden Projektils auf die Fragmente anzuwenden. Zusätzlich ist es möglich mehrere vorgebrochene Gegenstücke eines Objektes zur Entwicklungszeit zu erstellen und zum Zeitpunkt des Austausches nach dem Zufallsprinzip auszuwählen [14]. Außerdem gibt es die Möglichkeit mehrere Stufen einer Zerstörung eines Objektes zu modellieren, um eine mehrfache Zerstörung zu ermöglichen, wie beispielsweise in Abbildung 25 gezeigt wird.



Abbildung 25: Das Tauschen eines Türmodells in der *Source Engine*. Die aufeinanderfolgende Beschädigung der Tür im Spiel erzeugt immer diese Abfolge, unabhängig davon, an welcher Stelle genau der Schaden angewandt wurde [18].

Bei beiden der eben genannten Verbesserungen entsteht jedoch auch der Nachteil, dass die Entwicklungszeit erhöht wird, da zusätzliche Objekte oder Stufen modelliert werden müssen. Vor allem wenn das grundlegende Objekt in der Entwicklung geändert wird, müssen infolgedessen alle davon abhängigen Modelle angepasst werden. Zusätzlich entsteht natürlich auch ein erhöhter Speicherverbrauch, durch die zusätzlich vorhandenen vorgebrochenen Gegenstücke.

Ning [58] versucht den Realismus dieser Technik zu erhöhen, indem Informationen über den Aufprall, insbesondere die Position des Einschlagpunktes und die Geschwindigkeit des Projektils verwendet werden, um die Verteilung der Voronoi Punkte zu bestimmen, mit einer höheren Dichte von Punkten um das Zentrum des Einschlags. Dies führt zu einem gebrochenen Objekt mit kleineren Fragmenten in der Nähe des Einschlagpunktes wobei die Bruchstücke mit zunehmender Entfernung vom Aufprall immer größer werden. Dies ist jedoch in einem Videospielkontext nicht direkt anzuwenden, da der Einschlagpunkt vorher bekannt sein muss. Die einzige Ausnahme davon sind Zwischensequenzen, da diese zwar in Echtzeit gerendert werden aber immer einem festen Skript folgen [14].

3.2.2 Dynamische Methoden

Dynamische Methoden versuchen nun die Nachteile von vorgebrochenen Objekten zu beheben. Ziel ist es eine Technik zu entwickeln welche abhängig vom Einschlagpunkt dynamisch Fragmente erstellt, mehrere Stufen der Zerstörung eines Modells zulässt und trotz alldem soll der rechnerische Aufwand gering sein um diese Methode in Echtzeitanwendungen verwenden zu können. Eine der Herausforderungen, die durch das dynamische Zerstören entsteht, ist das Schneiden, auch *cutting* oder *clipping* genannt, des Meshes zur Laufzeit. Welche Methoden dafür vorhanden sind, werden in Kapitel 3.4 aufgezählt und genauer beschrieben.

Najim et al [54] präsentieren in ihrer Arbeit einen Algorithmus für das dynamische Zerstören von 3D Objekten mithilfe einer Voronoi-Tesselierung und graphics processing unit (GPU) Shadern. In dem Artikel wird eine Methode beschrieben, bei der die Berechnungen für die Voronoi-Tesselierung von der GPU durchgeführt werden und nur die Ausgangspositionen von der central processing unit (CPU) bearbeitet werden. Infolgedessen ermöglicht diese Methode eine schnelle und effiziente Frakturierung von Objekten, jedoch ist es wichtig zu beachten, dass das System nur mit einseitigen 3D-Objekten funktioniert, das heißt bei Objekten bei denen nur die der Kamera zugewandten Normalen gerendert werden. Zusätzlich werden bei dieser Methode die entstandenen Löcher der Frakturierung nicht mit Dreiecken gefüllt und dadurch entstehen hohle Schalen wie in Abbildung 26 zu sehen ist [54].

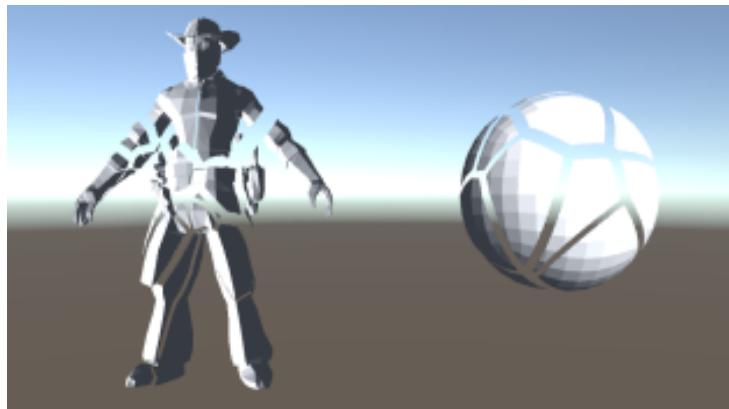


Abbildung 26: Das Ergebnis von zwei verschiedenen 3D-Objekten die während der Laufzeit gebrochen wurden, jedoch wurden die entstandenen Löcher nicht gefüllt [54].

Eine der ersten dynamischen Methoden, welche nach wie vor effizient genug für die Anwendung in Videospielen ist, veranschaulichen Müller et al. [50] in dem Werk „*Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions*“. In ihrer Arbeit verwenden sie bereits eine von Su et al. [72] vorgestellte Technik, nämlich das Verwenden eines generischen sogenannten fracture patterns, welches zur Laufzeit mit dem Einschlagpunkt abgestimmt wird und mit dem zu brechenden Objekt mithilfe von booleschen Operationen verglichen wird, um dynamische Bruchstücke zu erzeugen. Diese Bruchmuster sind eine vorher berechnete und erstellte Zerlegung des Raums, typischerweise eines einfachen Würfels, welche für verschiedene

Typen von Objekten verwendet werden können, zum Beispiel kann ein Spinnennetzmuster für alle Glasobjekte verwendet werden, wie in Abbildung 27 gezeigt wird [50]. Das heißt, richtig eingesetzt kann diese Technik einige der Probleme mit vorgebrochenen Modellen lösen, wie beispielsweise die Ausrichtung der Fragmentierung auf die Aufprallstelle oder den Zeitaufwand für die vorherige Erstellung von fragmentierten Objekten, da der Artist nur Bruchmuster erzeugen muss, die mehrfach für verschiedene Objekttypen verwendet werden können [14].

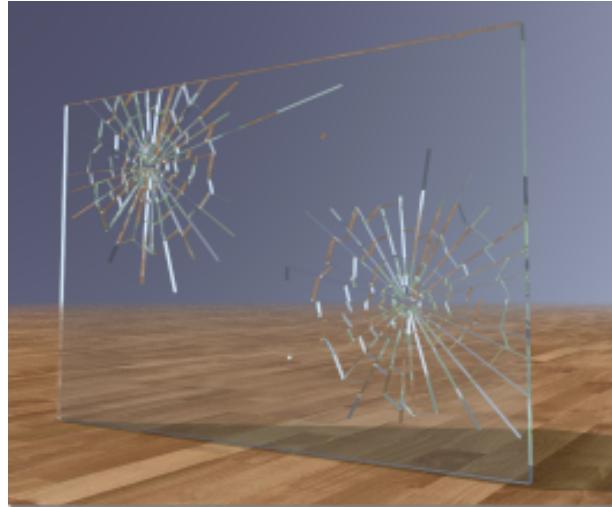


Abbildung 27: Ein Glasfenster welches durch ein Spinnennetzmuster an mehreren Stellen realistisch zerbrochen wurde [50].

Der eigentliche Grundgedanke des Ansatzes von Müller et al. [50] besteht darin, die visuelle Geometrie als eine Zusammensetzung von konvexen Formen darzustellen, wobei jede konvexe Form einen einzigartigen Teil des visuellen Meshes enthält. Diese konvexen Formen werden mithilfe ihres neuartigen Algorithmus, Volumetric Approximate Convex Decomposition (VACD), erstellt. Infolgedessen ermöglicht die Zerlegung des Objekts eine lokale Anwendung der eben genannten Bruchmustern zu geringer Rechenleistung, da in der Regel nur ein kleiner Teil der konvexen Formen von der Operation betroffen ist. Su et al. [72] haben festgestellt, dass die booleschen Operationen, die für die Anwendung der fracture patterns erforderlich sind, direkt auf dem Mesh durchgeführt werden könnten, wiesen aber darauf hin, dass es schwierig wäre, solche Operationen effizient und robust zu implementieren. Doch aufgrund der Zerlegung des Objektes in mehrere konvexe Formen durch den VACD-Algorithmus, ist es möglich die booleschen Operationen direkt anzuwenden und dadurch Modelle effizient in Echtzeit zu brechen [50].

3.3 Physikalisch basierte Methoden

Die Geometrischen Methoden sind aufgrund der Vereinfachungen die vorgenommen worden, um eine Fragmentierung in Echtzeit durchzuführen, begrenzt. In der Regel wird davon ausgegangen, dass die Brüche an der Aufprallstelle auftreten, es sei denn, sie sind bei einem vorgebrochenen Modell ausdrücklich für einen anderen Ort bestimmt. Wenn ein Bruch auftritt, erfolgt er sofort im gesamten Material. Bei Kollisionen kann dies einen realistischen Effekt erzeugen, aber in Situationen, in denen das Material unter Spannung gehalten wird, simuliert es das typische Verhalten nicht genau [14]. In der Realität würden Objekte, auf die Kräfte einwirken, im Material Spannung erzeugen, die wiederum zu einer Verformung des Materials vor dem Bruch führen würde. Das heißt, dass geometrische Methoden Fragmentierungen solcher Objekte nicht genau wiedergeben, da sie keine elastische Verformung vor oder nach dem Brechen abbilden. Infolgedessen wird ihre Verwendung auf Materialien wie Keramik und Glas, also eher starre Materialien, beschränkt [14].

Physikalisch basierte Methoden versuchen nun realistische Verformungen, Brüche und Rissausbreitung zu reproduzieren, indem die grundlegenden physikalischen Prozesse berücksichtigt werden. Dabei gibt es zwei kritische Werte für die Spannung, die ein Material erfahren kann: die Streckgrenze und die Zugfestigkeit. Wenn die aufgebrachte Spannung bis zur Streckgrenze ansteigt, wird das Material als elastisch verformt bezeichnet, was bedeutet jede Verformung, die bis zu dieser Grenze auftritt, kann rückgängig gemacht werden. Sobald die Spannung abgebaut wird, kehrt das Objekt im Wesentlichen in seinen ursprünglichen Zustand zurück [5]. Übersteigt die Spannung die Streckgrenze des Materials, handelt es sich um eine plastische Verformung, das heißt jede weitere Verformung über diesen Punkt hinaus ist dauerhaft. Wird die Spannung noch weiter erhöht, so erreicht das Material schließlich seine Zugfestigkeit und fängt an, ab diesem Punkt zu brechen [14].

Dadurch, dass die physikalisch basierten Methoden die grundlegenden Prozesse der Bruchmechanik berücksichtigen, werden zwar Ergebnisse erzielt, die einen hohen Grad an Realismus besitzen, jedoch sind diese Methoden ohne Vereinfachungen meist zu rechenaufwändig, um sie in einem Echtzeitkontext verwenden zu können [60]. Aufgrund der realistischen Fragmentierung und hohen Berechnungszeit der physikalischen Methoden, werden diese häufig bei Simulationen eingesetzt, um akkurate Resultate zu erzielen. Jedoch haben die Techniken für physikalisch basierte Fragmentierung in den letzten zwanzig Jahren erhebliche Fortschritte gemacht. Echtzeitsimulation auf kostengünstigen Rechnern können heute Ergebnisse liefern, die früher mehrere Stunden an Berechnungen auf High-End Computern benötigt hätten [63]. Dadurch finden sich auch physikalisch basierte Methoden in Videospielen wieder.

Die zwei wichtigsten Techniken der physikalisch basierten Methoden, welche in den folgenden Kapiteln näher betrachtet werden, sind das Mass-Spring Model und die Finite-Elemente-Methode. Bei beiden Methoden wird das zu simulierende Objekt in kleinere, überschaubare Teile zerlegt, bevor anhand von physikalischen Gesetzen und der Bruchmechanik ermittelt wird,

ob ein Bruch auftreten wird [14].

3.3.1 Mass-Spring Model

Das Mass-Spring Model ist eine der einfachsten Methoden zur Modellierung eines verformbaren Körpers. Eine der ersten Arbeiten, die das Mass-Spring Model nutzten um Verformungen und Brüche bei Animation darzustellen, wurde von Terzopoulos und Fleischer [75] präsentiert. Grundlegend basiert die Technik des Mass-Spring Models darauf, dass ein Objekt in diskrete Elemente unterteilt wird, wobei jedes Element eine eigene Masse und Position besitzt. Weiters werden diese paarweise mit Sprungfedern verbunden, die jeweils eine eigene Steifigkeit, Dämpfungsfaktor und Länge haben [52]. Wenn Kräfte auf das Objekt einwirken, werden die Elemente bewegt und die daran befestigten Federn gedehnt oder gestaucht, was wiederum Kräfte erzeugt, die das Gleichgewicht des Modells wiederherstellen. Diese Kräfte werden mithilfe des Hookeschen Gesetzes

$$F = D \cdot \Delta l \quad \text{beziehungsweise} \quad \Delta l = \frac{F}{D} \quad (4)$$

berechnet, wobei F die einwirkende Kraft, D die Steifigkeit der Feder und Δl die Dehnung beschreibt. Durch die Verwendung von Federn zur Verbindung der Elemente im Objekt ist es mit dieser Methode möglich, das elastische Verhalten von Materialien zu simulieren, welches mit Geometrie basierten Methoden nicht möglich ist. Sobald die Kraft von dem 3D-Objekt genommen wird, bringen die Federn das Objekt mit Hilfe der Dämpfung wieder in seine ursprüngliche Form zurück. Um nun eine Fragmentierung durchzuführen, wird die Feder, sobald ihre Länge einen kritischen Punkt überschritten hat entfernt und von weiteren Berechnungen ausgeschlossen [14]. Die Korrektheit und Genauigkeit des resultierenden Bruchs, hängt von den verwendeten Elementen innerhalb des Objektes ab. Norton et al. [59] verwendeten Elemente aus Knoten in Achtergruppen, um Würfel zu bilden, wobei die Knoten mit ihren Nachbarn durch Federn verbunden waren und die Würfel mit den nächsten durch ihre gemeinsamen Knoten. Eine Konstruktionstechnik besteht darin, jede beliebige Form mit Würfeln zu approximieren, jedoch führt dies zu treppenartigen Objekten. Um Objekte abzurunden, lockerten Norton et al. [59] die Anforderungen, dass die Elemente keine präzisen Würfel sein müssen, das heißt, dass die Kanten nicht gleich lang sein müssen. Wie die Elemente aufgebaut sein können und wie daraus ein Objekt erstellt wird, ist in Abbildung 28 zu sehen.

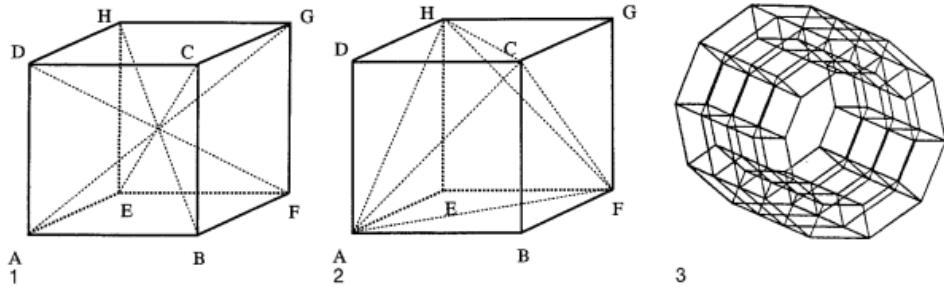


Abbildung 28: 1) Ein Würfel mit den zugehörigen Verbindungen. Jeder der 12 Kanten des Würfels, sowie jeder der vier inneren Diagonalen ist eine Feder zugeordnet.
 2) Anderer Aufbau des Würfels mit Flächendiagonalen. Wieder ist jeder der 12 Kanten des Würfels eine Feder zugeordnet, jedoch diesmal nicht den inneren Diagonalen, sondern den sechs Flächendiagonalen.
 3) Ein Zylinder wird mit einem Netz aus deformierten Würfeln modelliert. [59]

Mazarak et al. [47] verwendeten einen voxelbasierten Ansatz zur Modellierung fester Objekte. Die Voxel sind untereinander verbunden und diese Verbindungen zwischen ihnen sind unendlich starr und lassen keine Flexibilität im Objekt zu, sodass benachbarte Voxel fest miteinander verbunden bleiben. Das heißt, die Voxel sind im Verhältnis zueinander unbeweglich, können aber zu komplexeren Strukturen, auch Körper genannt, gruppiert werden. Diese Körper können so angeordnet werden, dass sie die gewünschte Form erhalten. Um die stufenförmigen Kanten zu verringern, können die Voxel so klein wie nötig gemacht werden [77]. Eine Darstellung der Anordnung der Voxel und Körper sind in Abbildung 29 zu sehen.

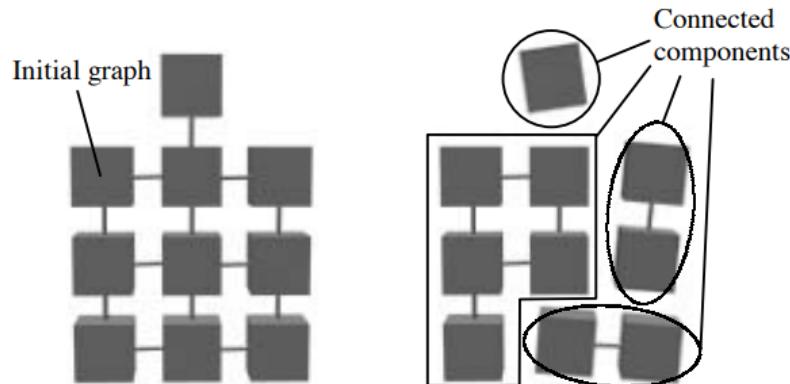


Abbildung 29: Verbundene Voxel des Mass-Spring Models [47]

Der Bruch eines Objekts wird nun simuliert, indem wieder die Verbindungen zwischen den verbundenen Körpern unterbrochen werden, wodurch auch eine Rissbildung in dem Objekt nachgeahmt wird. Mazarak et al. [47] verwenden diese Methode, um die Auswirkungen einer Druckwelle auf umliegende Objekte zu simulieren, um infolgedessen Explosionen modellieren zu können, die zu realistischen Trümmern führen. Bei einfachen Modellen ist es mit dieser Me-

thode sogar möglich die notwendigen Berechnungen in Echtzeit durchzuführen, wird jedoch eine Kollisionserkennung benötigt, ist diese Technik für einen Echtzeitkontext zu rechenaufwändig.

Die Qualität und Performance der Zerstörung kann stark von der Auflösung der Elemente abhängen. Ist die Auflösung zu niedrig, sind die Objekte und die erzeugten Fragmente groß und blockartig, werden jedoch schnell und effizient berechnet, ist diese allerdings zu hoch, steigt der für das Objekt benötigte Speicherplatz und die dafür erforderliche Rechenleistung [14]. Zusätzlich haben bisher genannte Methoden nicht die Entstehung vom Staub, der bei einer Zerstörung auftritt, beachtet. Imagire et al. [33] schlagen daher eine Zerstörung auf drei Ebenen vor: grobe Zerstörung, feine Trümmer und Staub. Die grobe Zerstörung wird mithilfe der Extended Distinct Element Method (EDEM) vorgenommen, welche eine Mass-Spring Methode ist, die ursprünglich von Meguro und Hakuno [49] entwickelt wurde. Bei dieser werden kugelförmige Elemente im Objekt platziert und anstatt die Elemente in Würfel zu gruppieren, wird jedes Element als Grundlage für ein 3D-Voronoi Diagramm verwendet. Die Verteilung und die Menge der feinen Trümmer und des Staubs werden auf der Grundlage der Bruchenergie, also jene Energie die zum Zerbrechen des Objekts führt, berechnet. Zuerst wird die Energie dazu verwendet, die maximale Größe der Trümmer zu ermitteln, wenn diese kleiner als das EDEM-Element ist, wird das Element weiter zerlegt. Schließlich wird die erzeugte Staubmenge auf der Grundlage Bruchenergie berechnet und mithilfe der Flüssigkeitsdynamik simuliert [14]. Diese Methode ermöglicht zwar eine hochwertigere Zerstörung, ohne dass die Auflösung der gruppierten Elemente stark erhöht werden muss, jedoch erwies sich die Simulation auf allen drei Ebenen als zu rechenaufwändig und langsam, um diese Technik für Echtzeitanwendungen nutzen zu können.

Der große Nachteil des Mass-Spring-Models ist, dass es viel manuelle Arbeit beim Aufbau des Netzwerkes und beim Erstellen von Objekten benötigt [55]. Ein weiteres generelles und wesentliches Problem bei der Mass-Spring-Methode ist, dass beim Auftreten der Fragmentierung die genaue Lage und Ausrichtung des Bruchs nicht bekannt sind [77]. Nichtsdestotrotz wurde das Modell außergewöhnlich gut in *BeamNG.Drive* implementiert, einem Echtzeit-Fahrzeugsimulator, der auf einer Mass-Spring-Methode zur Simulation von Fahrzeugverformungen basiert [46]. Jedoch wird die Methode nur zur Verformung der 3D-Objekte verwendet. Für die Zerstörung werden entweder vorgebrochene Objekte verwendet und diese ausgetauscht oder grobe Teile des Autos wie beispielsweise eine Autotür werden als ganzes vom Objekt getrennt und nicht in kleinere Fragmente unterteilt.

3.3.2 Finite-Elemente-Methode

Die Finite-Elemente-Methode (FEM) wird in der Mechanik, genauer gesagt in der Festigkeits- und Verformungslehre verwendet, um komplexe mechanische Komponenten zu simulieren, die mit einfachen Methoden nicht analysiert werden können [14]. Sie ist eine der meistverwen-

deten Methoden für Simulationen. Bei der FEM wird das Objekt in eine Menge von diskreten Elementen unterteilt, in der Regel Dreiecke in 2D und Tetraeder in 3D. Diese Elemente verbinden sich mit ihren Nachbarn durch ihre gemeinsamen Knotenpunkte. Dadurch wird bei der FEM das Problem nicht auf dem ursprünglichen Mesh, sondern auf der endlichen Menge von Knotenpunkten definiert, was zu einer Reihe von algebraischen Gleichungen führt, die numerisch gelöst werden [52]. Anstatt ein kontinuierliches Problem zu lösen, versucht diese Methode, die diskreten Position der Knotenpunkte zu bestimmen. Das heißt, die Spannung des Objekts kann nun über diese einfacheren Elemente berechnet werden, wobei Kräfte, Verschiebungen und Randbedingungen auf die Knoten der Elemente angewendet werden.

Die Finite-Elemente-Methode ermöglicht, im Gegensatz zur Mass-Spring-Methode, wesentlich genauere Berechnungen der Spannungen und Verformungen, denen ein Körper unter einer einwirkenden Last ausgesetzt ist, aufgrund der maßgeblichen mechanischen Gleichungen. Zusätzlich können auch tatsächliche Materialeigenschaften wie die Streckgrenze und Zugfestigkeit definiert werden können, um genau zu bestimmen, wie sich ein Körper verformt oder wie er bricht [14].

O'Brien und Hodgins [60] implementierten eine Version der Finite-Elemente-Methode, der auf der linearen elastischen Bruchmechanik basiert. In ihrem Ansatz werden 3D-Objekte mit einem Mesh aus tetraedrischen Elementen modelliert, wie in Abbildung 30 zu sehen ist.

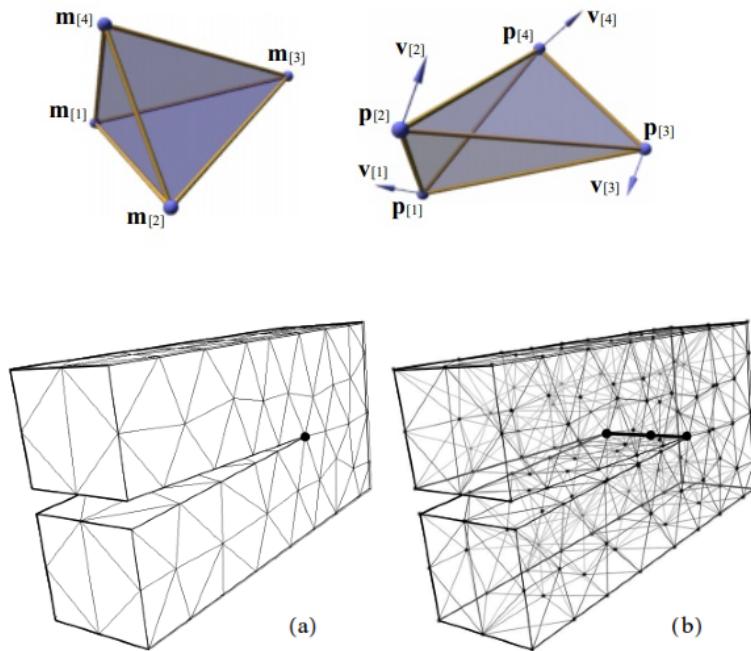


Abbildung 30: Oben: Aufbau eines tetraedrischen Elements.

Unten: Tetraedrisches Mesh eines einfachen Objekts.

In (a) sind nur die äußeren Flächen gezeichnet, in (b) wird die innere Struktur gezeigt [60].

Innerhalb jedes Elements wird das Material lokal durch eine Funktion mit einer endlichen Anzahl von Parametern beschrieben. In der Simulation wird bestimmt, wo Risse beginnen und in welche Richtungen sie sich ausbreiten sollen, indem die Spannungen analysiert werden, die bei der Verformung des Meshes entstehen. Obwohl diese Arbeit realistische Ergebnisse liefert, kann diese Methode aufgrund des hohen Rechenaufwands nicht in Echtzeitanwendungen verwendet werden.

Parker und O'Brien [63] bauten jedoch auf den Methoden von O'Brien [60] auf und optimierten in ihrer Arbeit „*Real-Time Deformation and Fracture in a Game Environment*“ eine Finite-Elemente-Methode bis zu den Punkt, indem das System effizient genug für ein kommerzielles Videospiel war. Ihr Ziel war es ein Zerstörungssystem zu entwickeln, welches robust gegenüber unvorhersehbaren Benutzerinteraktionen ist, schnell genug, um sinnvolle Szenarien in Echtzeit zu modellieren, geeignet für die Verwendung bei der Entwicklung eines Spiellevels und mit angemessenen Steuerelementen, die es den Erstellern erlaubt, die gewünschte Fragmentierung zu erzielen. Diese Optimierung erzielten sie vor allem durch Multithreading und durch das Aufteilen des Körpers in mehrere Teile, um Verformungen parallel zu berechnen. Das von Parker und O'Brien [63] beschriebene System wurde als separate Physik-Engine implementiert, die mit anderen Spielkomponenten integriert werden kann. Es wird derzeit von *Pixelux Entertainment* unter dem Markennamen *DMM* [1] kommerziell vertrieben und wurde beispielsweise bereits erfolgreich in der Xbox 360- und der PS3-Version des von *LucasArts* veröffentlichten Videospiels *Star Wars: The Force Unleashed* und in dem für Xbox One und PC erschienen Third-Person Shooter *Quantum Break* des finnischen Entwicklerstudios *Remedy Entertainment* [2] eingesetzt.

3.4 Clipping

Unabhängig davon, ob eine Geometriebasierte oder physikalisch basierte für das Berechnen der Fragmente verwendet wird, muss das Mesh zum Zeitpunkt der Fragmentierung geschnitten werden. Wie bereits erwähnt werden Meshes in Videospielen nicht als Volumina modelliert, sondern nur deren Oberflächen und sind dadurch hohl. Daraus ergibt sich, dass beim Brechen eines Meshes Löcher entstehen, die wiederum gefüllt werden müssen, um abgeschlossene Fragmente zu erhalten. Müller et al. [50] haben gezeigt, dass das Schneiden des Meshes sogar der komplexeste Schritt des gesamten Algorithmus ist und das Schreiben von numerisch stabilem Code alles andere als trivial ist. In diesem Kapitel werden zwei Methoden für das Schneiden von Meshes dargestellt.

3.4.1 Constructive Solid Geometry

Eine Methode zum Generieren der Fragmente ist die sogenannte Constructive Solid Geometry (CSG). Die CSG ist ein Algorithmus zur Erzeugung komplexer 3D-Objekte durch die Zusammensetzung von soliden simpleren Körpern mithilfe von boolescher Operationen der Mengenlehre wie etwa Vereinigung, Differenz und Schnittmenge [21]. Diese werden in Abbildung 31 anhand eines Würfels und einer Kugel veranschaulicht.

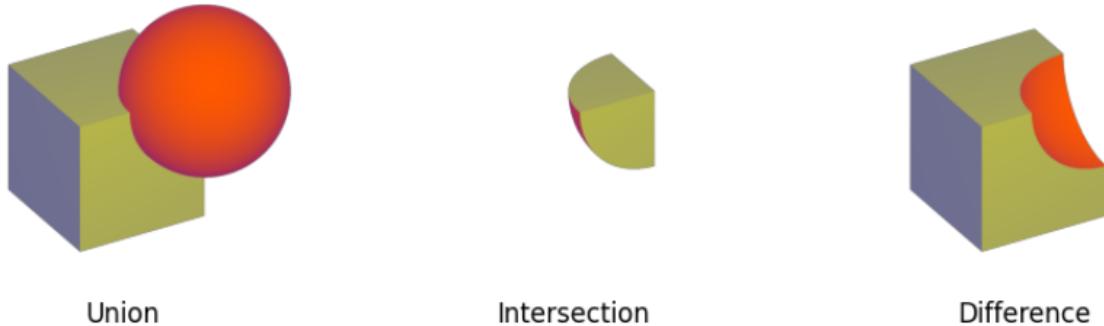


Abbildung 31: CSG-Operationen eines Würfels und einer Kugel und die daraus resultierenden Objekte. [23]

Meist wird für die Darstellung von CSG ein Binärbaum verwendet, wobei die Primitive die Blätter sind, die CSG-Operationen die Knoten und das finale Objekt die Wurzel ist [32]. Dabei ist jedoch zu beachten, dass die Operationen nicht kommutativ sind, deshalb sind die Kanten des Baumes geordnet. Üblicherweise wird die Menge der einfachen Objekte der Blätter des Baumes auf Würfel, Zylinder, Prismen, Pyramiden, Kugeln und Kegel beschränkt, die genaue Anzahl hängt jedoch von der Implementierung ab.

CSG wäre eine ideale Lösung, um die Schnittmenge der Voronoi Zellen und des Meshes zu finden und dadurch die Fragmente zu berechnen, jedoch wird für eine CSG-Operation kein Mesh benötigt, sondern vollständige Volumen [4]. Da die Zellen des Voronoi Diagramms in diesem Projekt durch eine Menge von Flächen dargestellt werden wäre eine Umwandlung für die Operation nötig. Nach der Berechnung des Fragmentes mithilfe von CSG müsste das Objekt wieder in eine Mesh Darstellung umgewandelt werden. Weiters haben Su et al. [72] festgestellt, dass die booleschen Operationen zwar auch direkt auf den Netzen durchgeführt werden könnten, wiesen aber darauf hin, dass eine robuste Implementierung solcher Operation kompliziert wäre und möglicherweise nicht in Echtzeit durchgeführt werden kann.

3.4.2 Schneiden eines Meshes mit einer Ebene

Das Schneiden eines Meshes erfordert mehrere Schritte. Im ersten Schritt wird, das Mesh und eine Ebene, entlang welcher das Mesh geschnitten werden soll, definiert. Der zweite Schritt

besteht darin, die Flächen, die die Ebene schneiden aufzutrennen und dadurch das Mesh in zwei separate Teile aufzuteilen. Anschließend müssen die Löcher, die durch das Schneiden entstanden sind, trianguliert werden um das Mesh zu schließen. [65]

Dadurch das jede Fläche eines Meshes aus Dreiecken besteht, kann das Resultat des Schneidens berechnet werden, indem jedes Dreieck des Meshes mit der Ebene geschnitten wird. Das Schneiden eines einzelnen Dreiecks mit einer Ebene beginnt damit, dass die Kanten des Dreiecks mithilfe der gespeicherten Eckpunkte definiert werden. Nachdem die Kanten konstruiert wurden, werden diese eine nach der anderen mit der Ebene mithilfe folgender Formel geschnitten:

$$d = \frac{(p - l_0) \cdot \vec{n}}{(l_1 - l_0) \cdot \vec{n}} \quad (5)$$

Die Ebene ist definiert durch einen Punkt auf der Ebene p und einem Normalvektor \vec{n} und die Punkte l_0 und l_1 beschreiben die Kante des Dreiecks. Mithilfe dieser Formel ist nicht nur bekannt, ob die Kante die Ebene schneidet, sondern es kann auch mit d der Schnittpunkt einfach berechnet werden.

Da die Ebene den Raum in zwei Teile teilt, nämlich einen unterhalb und einen oberhalb der Ebene, gibt es für jedes der Dreiecke drei mögliche Ergebnisse. Entweder befindet sich das gesamte Dreieck unterhalb oder oberhalb der Ebene oder es wird von der Ebene geschnitten. Die ersten zwei Situationen sind einfach zu lösen, indem das gesamte Dreieck der entsprechenden Seite zugewiesen wird. Wird das Dreieck jedoch von der Ebene geschnitten, muss dieses in zwei Teile geteilt werden. Bei jedem Dreieck das geschnitten wird, liegt einer der Eckpunkte auf einer Seite der Ebene, während sich die beiden anderen Eckpunkte auf der anderen Seite befinden [44]. Eine weitere Eigenschaft ist, dass immer nur zwei Kanten des Dreiecks geschnitten werden können, niemals nur eine oder alle drei [65].

Nachdem die neuen Eckpunkte mithilfe der Lösung aus Formel 5 berechnet wurden, müssen die neuen Flächen konstruiert werden. Das Schneiden eines Dreiecks ergibt immer ein kleineres Dreieck und ein Viereck, welches wiederum in zwei Dreiecke aufgeteilt wird, um es zum Mesh hinzufügen zu können. Dies wird in Abbildung 32 veranschaulicht.

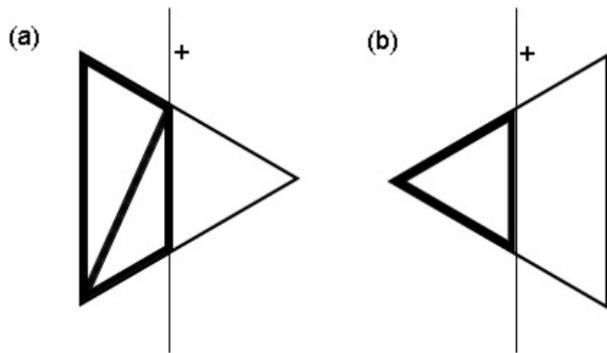


Abbildung 32: Schneiden eines Dreiecks mit einer Ebene. [31]

Das Schneiden wird für jedes Dreieck im ursprünglichen Mesh durchgeführt und die Ergebnisse werden dem entsprechenden neuen Meshes zugeordnet.

Nachdem dies erfolgt ist, ist zwar das Mesh in zwei Teile geteilt worden, jedoch fehlt noch ein wesentlicher Schritt. Da Meshes keine Volumina sind, sondern nur Oberflächen entstehen nach dem Schneiden Löcher, die mit neuen Dreiecken gefüllt werden müssen. In Abbildung 33 wird gezeigt, dass das Mesh ohne dem Füllen der Löcher hohl ist und um die Illusion von festen Körpern aufrechtzuerhalten, müssen diese gefüllt werden.

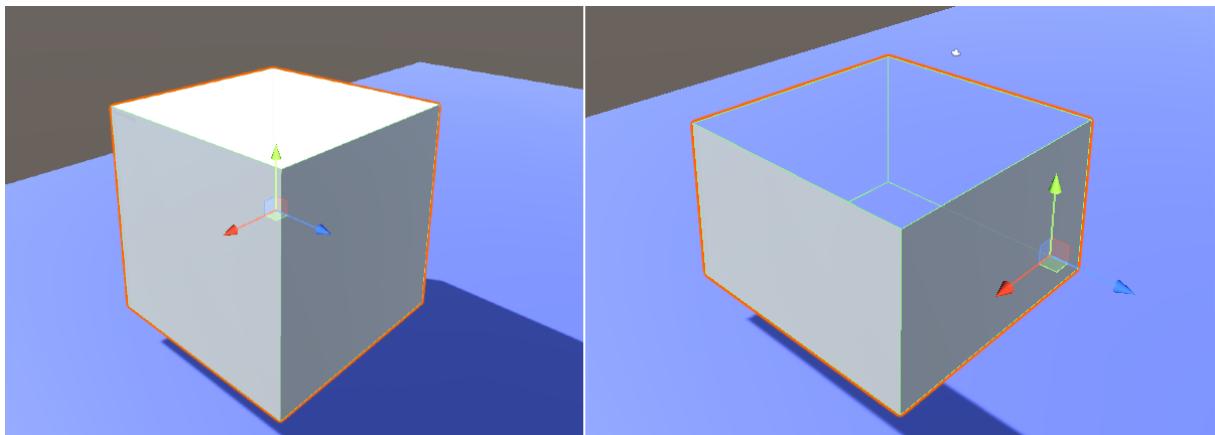


Abbildung 33: Entstandenes Loch nach dem Schneiden des Würfels mit einer horizontalen Ebene. Die Unterseite und die beiden hinteren Flächen werden aufgrund von Backface Culling nicht gerendert. Quelle: Eigene Darstellung

Die Schnittpunkte, die mithilfe der Formel 5 berechnet wurden, liegen auf einer Ebene, die die neue Fläche ergibt, um das Loch zu füllen. Diese Punktmenge muss nun trianguliert werden, um die Fläche korrekt rendern zu können. Dies kann grundsätzlich auf zwei Arten erreicht werden. Eine Möglichkeit ist das direkte Triangulieren der Punktmenge mithilfe der Berechnung der konvexen Hülle oder das Verwenden eines Algorithmus zum Berechnen der Delaunay Triangulation. Eine weitere Option ist das Bilden eines Polygons mithilfe der Schnittpunktpaare,

die beim Schneiden eines Dreiecks entstehen und eine Kante bilden. Ausgehend von diesem Polygon kann anschließend eine Triangulation berechnet werden, indem beispielsweise ein Eckpunkt mit jedem anderen Eckpunkt verbunden wird, mit dem er nicht bereits verbunden ist. Diese Option ist möglicherweise effizienter und schneller als das Berechnen der Delaunay Triangulation, jedoch entstehen bei der Delaunay Triangulation jene Dreiecke mit dem größten minimalen Winkel und dadurch verringert sich die Anzahl an splitterigen Dreiecken, welche bei weiteren Berechnungen zu Problemen führen könnten.

3.5 Resümee

Die Zerstörung von Objekten oder Landschaften ist in Filmen oder in Computerspielen immer häufiger zu sehen, unabhängig davon, ob ein Spiel zerstörende Objekte als Grundmechanik verwendet, oder um beeindruckende grafische Effekte erzielt werden wollen. Daher ist dies ein aktives Forschungsgebiet in der Computergrafik, um diese Effekte für Simulationen oder Echtzeitanwendungen wie Videospiele verwenden zu können. In diesem Kapitel wurden einerseits Gründe für das Implementieren von zerstörbaren Objekten genannt, beispielsweise um einen taktischen Vorteil zu erlangen, andererseits wurde ein Einblick in den Aufbau von Objekten gegeben und die grundlegenden Herausforderungen die dadurch beim Entwickeln eines Algorithmus zum Fragmentieren entstehen. Dabei wird grob zwischen zwei Kategorien unterschieden, den geometriebasierten Methoden und den physikalisch basierten Methoden. Um dehbare Materialien und dessen Bruchverhalten zu simulieren, wird auf physikalisch basierte Methoden zurückgegriffen. Diese werden aufgrund ihrer realistischen Ergebnisse aber hohen Berechnungszeit vorwiegend in Simulationen eingesetzt, jedoch finden sich diese Techniken mit Vereinfachungen und Optimierungen heutzutage auch oft in Echtzeitanwendungen wieder. Geometriebasierte Methoden versuchen nun eine Fragmentierung zu verursachen, ohne die grundlegenden physikalischen Prozesse zu berücksichtigen und sind meist in Videospiele aufzufinden, vor allem der Austausch des zu zerstörenden Objekts mit einem vorgebrochenen Gegenstück. Die geometrischen Methoden werden hauptsächlich für spröde Materialien, die sich vor dem Bruch nur in geringem Maße plastisch verformen, wie beispielsweise Keramik, Glas oder auch Stein, verwendet.

Ungeachtet dessen, welche Methode verwendet wird, muss für das Konstruieren der Fragmente zum Zeitpunkt der Zerstörung das Mesh geschnitten werden. Dies ist keineswegs ein triviales Problem und führt häufig zu numerischen Instabilitäten. Dadurch bleibt das effiziente Schneiden von Meshes eine Herausforderung.

4 Implementierung

Nachdem die notwendigen theoretischen Grundlagen ausführlich erläutert wurden, folgt der praktische Teil der Arbeit, die Implementierung. Das Ziel dieser Arbeit ist eine Evaluierung von Algorithmen zur Fragmentierung von Objekten in Echtzeitanwendungen wie etwa Videospielen. Um dieses Ziel zu erreichen wurden zwei Methoden implementiert um sie in unterschiedlichen Eigenschaften wie beispielsweise Performance, Qualität der Zerstörung und Komplexität der Implementierung zu vergleichen. In diesem Kapitel werden die Entwicklungsumgebung und die für die Durchführung des Projekts verwendeten Applikationen beschrieben. Weiters werden Details zur Implementierung der zwei verwendeten Methoden gezeigt, nämlich der Austausch des Objektes mit einem vorgebrochenen Model und ein dynamischer Voronoibasierter Algorithmus.

4.1 Entwicklungsumgebung

In diesem Kapitel wird die Hardware und Software erläutert, die zur Realisierung des Projektes und zur Messung der Performance der Algorithmen verwendet wurden.

4.1.1 Hardware

Die gesamte Implementierung wurde auf einem zwei Jahre alten Mid/High-End Gaming-PC mit Windows 10/11 durchgeführt. Die genauen Hardwarespezifikationen sind wie folgt:

- Prozessor: AMD Ryzen 7 3700X
- Arbeitsspeicher: 32GB
- Grafikkarte: NVIDIA GeForce GTX 1070

4.1.2 Applikationen

Die Algorithmen wurden in der Game-Engine Unity, Version 2020.2.4f1, implementiert. Sämtliche Skripts für Unity wurden in C# in der integrierten Entwicklungsumgebung Rider von JetBrains geschrieben. Diese Anwendungen wurden ausgewählt, da diese Teil des Industrie-standards sind und eine große Anzahl an Dokumentation und hilfreichen Features besitzen.

Für die Erstellung der vorgebrochenen Objekte wurde das *Cell Fracture Add-On* von der 3D-Modellierungssoftware Blender verwendet.

4.2 Features

einleitung
schreiben

4.2.1 2D Visualisierungen

Wie bereits erwähnt wurden im Rahmen dieser Arbeit zwei Methoden implementiert, die zur Fragmentierung eines Objektes verwendet werden können. Da eine dieser Methoden eine Voronoibasierte ist, wurde zuerst mit einer Implementierung des Voronoi Diagramms in 2D begonnen. Dafür wird mithilfe des Bowyer Watson Algorithmus, welcher in Kapitel 2.6.5 erläutert wurde, zuerst die Delaunay Triangulation berechnet. Anschließend wird durch die Dualität der beiden Graphen das Voronoi Diagramm abgeleitet. In 2D wird dies mit folgendem Pseudocode erreicht.

Algorithmus 1 : Ableiten des Voronoi Diagramms mithilfe der Delaunay Triangulation

Data :

DT = Delaunay Triangulation

Cells = [] // Voronoi Zellen werden hier gespeichert

foreach Eckpunkt P in DT **do**

 T = finde alle benachbarten Dreiecke von P

 U = finde alle Umkreismittelpunkte der Dreiecke in T

 sortiere U im Uhrzeigersinn

 erstelle neue Zelle durch Verbinden der sortierten Punkte

 füge neue Zelle zu Cells hinzu

end

Sowohl die Konstruktion der Delaunay Triangulation, als auch die Umwandlung zum Voronoi Diagramm besitzen hier einen Aufwand von $O(n^2)$. Als Datenstruktur wurde die in Kapitel 2.5.1 erläuterte Delaunay Struktur verwendet.

Das Konstruieren der Voronoi-Zellen ist in 3D ähnlich aufgebaut und wird im folgenden Unterkapitel des voronoibasierten Algorithmus genauer erläutert. Die Visualisierungen wurden mithilfe der in Unity integrierten Handles [78] durchgeführt und werden in folgenden Abbildungen 34 und 35 gezeigt.

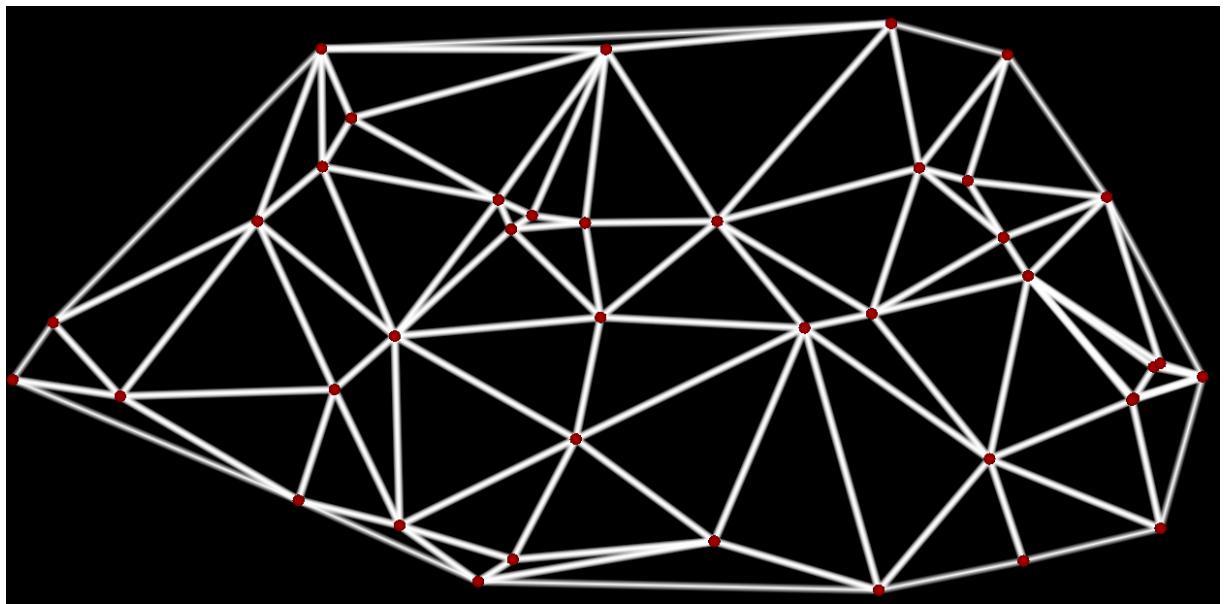


Abbildung 34: Delaunay Triangulation einer zufälligen Punktmenge. Quelle: Eigene Darstellung

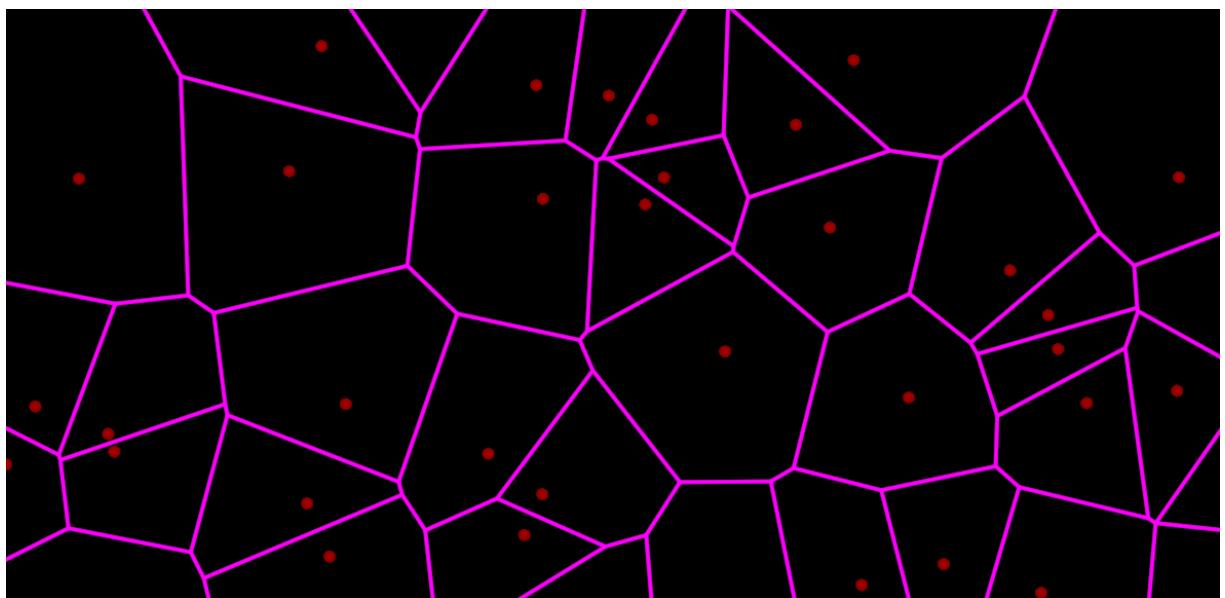


Abbildung 35: Voronoi Diagramm einer zufälligen Punktmenge. Quelle: Eigene Darstellung

Bevor das Programm gestartet wird, kann die gewünschte Menge an Punkten eingestellt werden. Ebenso können die Grenzen festgelegt werden, in denen die Punkte generiert werden. Die Punkte werden mithilfe des in der Unity Engine integrierten Pseudozufallszahlengenerator erstellt. Dieser Generator ist ein Xorshift 128 Algorithmus, der auf der Arbeit „*Xorshift RNGs*“ von George Marsaglia basiert. Er wird zu Beginn des Programms mit einem Seed des Betriebssystems initialisiert und gespeichert. Dadurch ergibt sich bei jedem wiederholtem Starten des Programms eine neue Verteilung der generierten Punkte.

Nachdem das Programm im Editor gestartet und die Punkte generiert wurden, kann zur Laufzeit eingestellt werden, ob die Delaunay Triangulation, das Voronoi Diagramm oder beides gleichzeitig visualisiert wird. Das erstellte Skript mit den eben genannten Parametern zur Punktgenerierung und Visualisierung der Graphen wird in Abbildung 36 dargestellt.

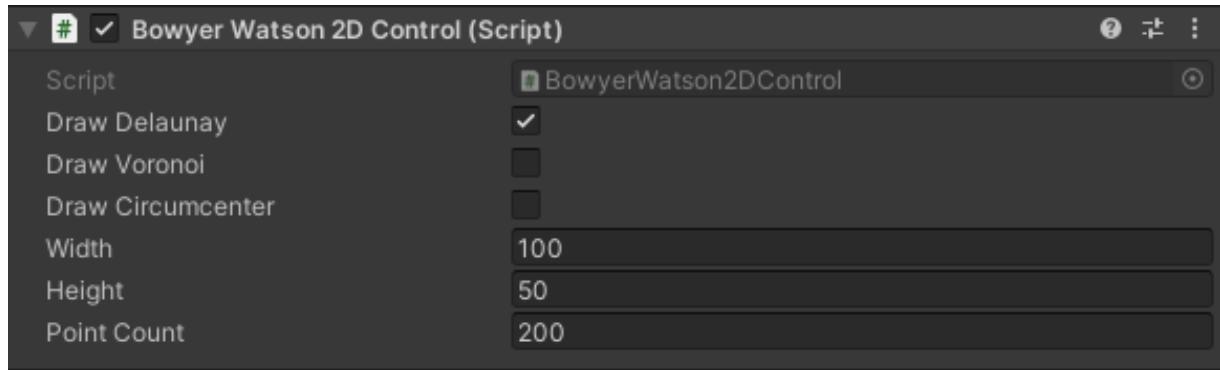


Abbildung 36: Übersicht der Komponente zur Visualisierung der Delaunay Triangulation und des Voronoi Diagramms. Quelle: Eigene Darstellung

Durch das Implementieren der Delaunay Triangulation und des Voronoi Diagramms in 2D wurde ein erster Einblick und ein Verständnis für die Algorithmen und die benötigten Datenstrukturen erlangt. Diese Implementierung war sowohl hilfreich, um das Visualisieren der Algorithmen in Unity kennenzulernen, als auch für die Umsetzung der voronoibasierten Zerstörung in 3D.

4.2.2 Austausch mit vorgebrochenen Model

Die erste der implementierten Methoden ist das Austauschen des 3D-Objektes mit einem bereits vorgebrochenen Gegenstück. Hierbei wird ein Mesh zur Darstellung des Objektes verwendet, bis das Objekt mit einer Kraft belastet wird, die größer als ein vorher festgelegter Wert ist. Zu diesem Zeitpunkt wird das Objekt einfach durch ein passendes Gegenstück, bestehend aus Fragmenten, ersetzt. Diese Methode ist demnach eine statische, geometriebasierte Methode, da die Fragmente nicht dynamisch zur Laufzeit erstellt werden und die grundlegenden physikalischen Prozesse nicht berücksichtigt werden.

Für die Implementierung dieser Methode, wurde als Erstes das vorgebrochene Gegenstück des gewünschten Models erstellt. Dafür kann auf mehrere zur Verfügung stehende Software zurückgegriffen werden wie etwa das *Cell Fracture Add-Ons* von Blender oder die Fragmente werden von einem 3D-Artist manuell erstellt. Hier wurde erstere Variante ausgewählt.

Um nun die Fragmente mithilfe dieses Add-Ons zu erstellen wird zuerst das 3D-Objekt in Blender geöffnet. Dies kann entweder ein selber modelliertes Objekt sein oder ein frei zur Verfügung stehendes Model. Hier wurde das Stanford-Bunny ausgewählt, da es ein beliebtes Testmodel in

der Computergrafik ist, das Add-On kann jedoch für jedes beliebige Objekt verwendet werden. Anschließend wird das Add-On ausgewählt, welches ein Fenster mit einer Reihe an Parametern besitzt, wie in Abbildung 37 gezeigt wird.

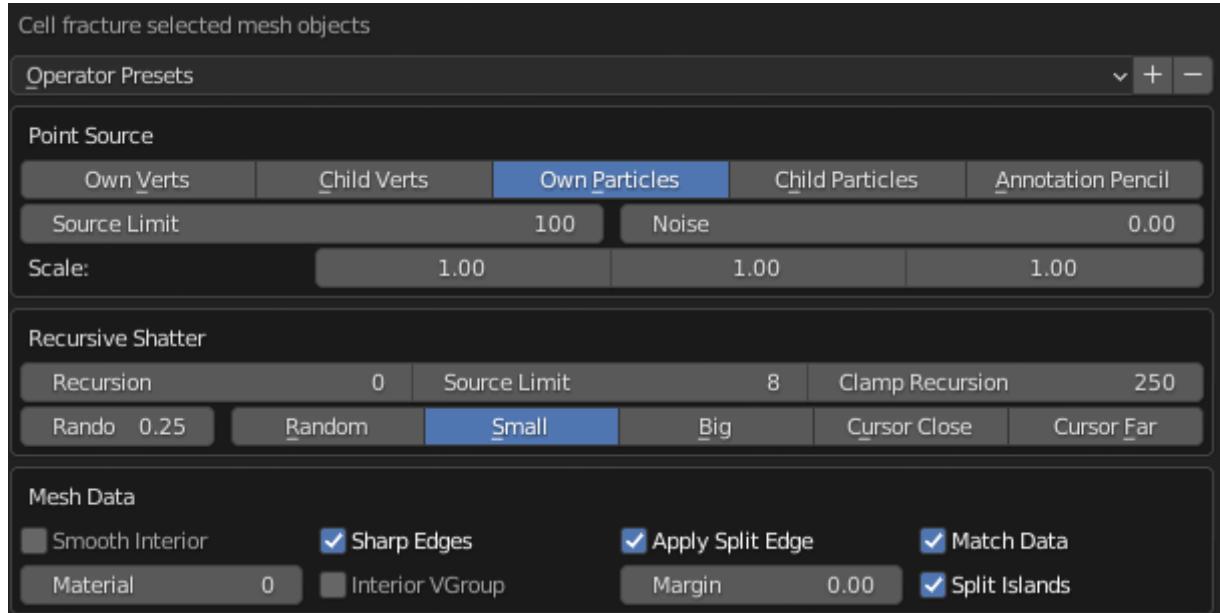


Abbildung 37: Einstellungen des *Cell Fracture Add-Ons* in Blender, die zum Erzeugen von unterschiedlichen Fragmenten gesetzt werden können. [11]

Mithilfe dieser Optionen können die gewünschten Fragmente erzielt werden. Die wichtigsten Einstellungen, um das Ergebnis der Fragmentierung zu verändern sind folgende:

- **Point Source:** bestimmt die zu verwendete Punktmenge zum Generieren der Fragmente:
 - *Own Verts*: generiert für jeden Eckpunkt des Objektes eine gebrochene Zelle
 - *Child Verts*: generiert für jeden Eckpunkt der Kind-Objekte eine gebrochene Zelle
 - *Own Particles*: generiert für jedes Partikel im Partikelsystem des Objektes eine gebrochene Zelle
 - *Annotation Pencil*: Fragmente werden abhängig von den gezeichneten Elementen des *Annotation Pencil* erstellt, wie bereits in Abbildung 24 gezeigt

Anmerkung: Die eben beschriebenen Einstellungen zum Auswählen der verwendeten Punktmenge können auch miteinander kombiniert werden

- *Source Limit*: beschränkt die Anzahl der Fragmente auf ein festgelegtes Limit
- *Noise*: fügt der Punktmenge Noise hinzu um eine zufälligere Frakturierung erhalten
- *Scale*: ermöglicht das Verändern der Größe der Fragmente mithilfe von drei Werten

die den Achsen entsprechen

- *Recursive Shatter*: ermöglicht das weitere Zerstören von bereits erzeugten Fragmenten, um detailliertere Fragmente zu erhalten
 - *Recursion*: bestimmt die Tiefe der Rekursion
 - *Source Limit*: beschränkt die Anzahl der Fragmente die bei einer rekursiven Fragmentierung entstehen können
 - *Clamp Recursion*: stoppt die rekursive Zerstörung, wenn die festgelegte Anzahl erreicht wurde
 - *Rando*: erhöht oder verringert die Wahrscheinlichkeit mit der eine Rekursion auftritt. Die Einstellungen nebenbei bestimmen, welche Zellen von der Rekursion betroffen werden sollten
- *Mesh Data*: verändert die Fragmente der Zerstörung an sich nicht, kann jedoch für die Schattierung oder die Anwendung von unterschiedlichen Materialen für die Innen- und Außenflächen verwendet werden

Nachdem von dem 3D-Model das gewünschte fragmentierte Gegenstück erstellt wurde, wird dieses als beispielsweise *FBX*-Datei exportiert und in Unity importiert. Um nun eine realistische Zerstörung mithilfe dieser Methode zu erstellen, müssen mehrere Schritte durchgeführt werden. Die Fragmente besitzen nach dem Importieren bereits eine *Mesh Filter* und eine *Mesh Renderer* Komponente und werden für das Anzeigen des Objektes benötigt. Damit die Fragmente mit anderen Objekten kollidieren und die Bewegungen von der Unity-Physik-Engine gesteuert werden, werden zuerst allen Fragmenten des gebrochenen Objekts die notwendigen Komponenten hinzugefügt, die bereits in Unity vorhanden sind: ein *Mesh Collider* und ein *Rigidbody*. Anschließend wird dies als Unity-Prefab gespeichert, um das Objekt später instanziieren zu können.

Als nächsten Schritt wurde ein Skript erstellt, welches zu dem zu zerstörenden Objekt hinzugefügt wird, um das Austauschen zur Laufzeit zu verarbeiten. Hierfür wird das eben erstellte Prefab zu dem Skript hinzugefügt. Anschließend wird im Skript zum Zeitpunkt der Kollision das Objekt ausgetauscht. Dies wird erreicht, indem die in Unity eingebaute Methode *OnCollisionEnter* genutzt wird. Diese enthält ein *Collision*-Objekt als Parameter, welches Informationen über das kollidierte Objekt, den Gesamtimpuls der Kollision und die relative Geschwindigkeit enthält. Nachdem diese Methode durch eine Kollision ausgelöst wird, wird das Objekt ausgetauscht, indem das Prefab des gebrochenen Objekts instanziert wird und das Originale Model gelöscht beziehungsweise auf inaktiv gesetzt wird. Der gesamte Ablauf dieses Algorithmus wird in Abbildung 38 veranschaulicht.

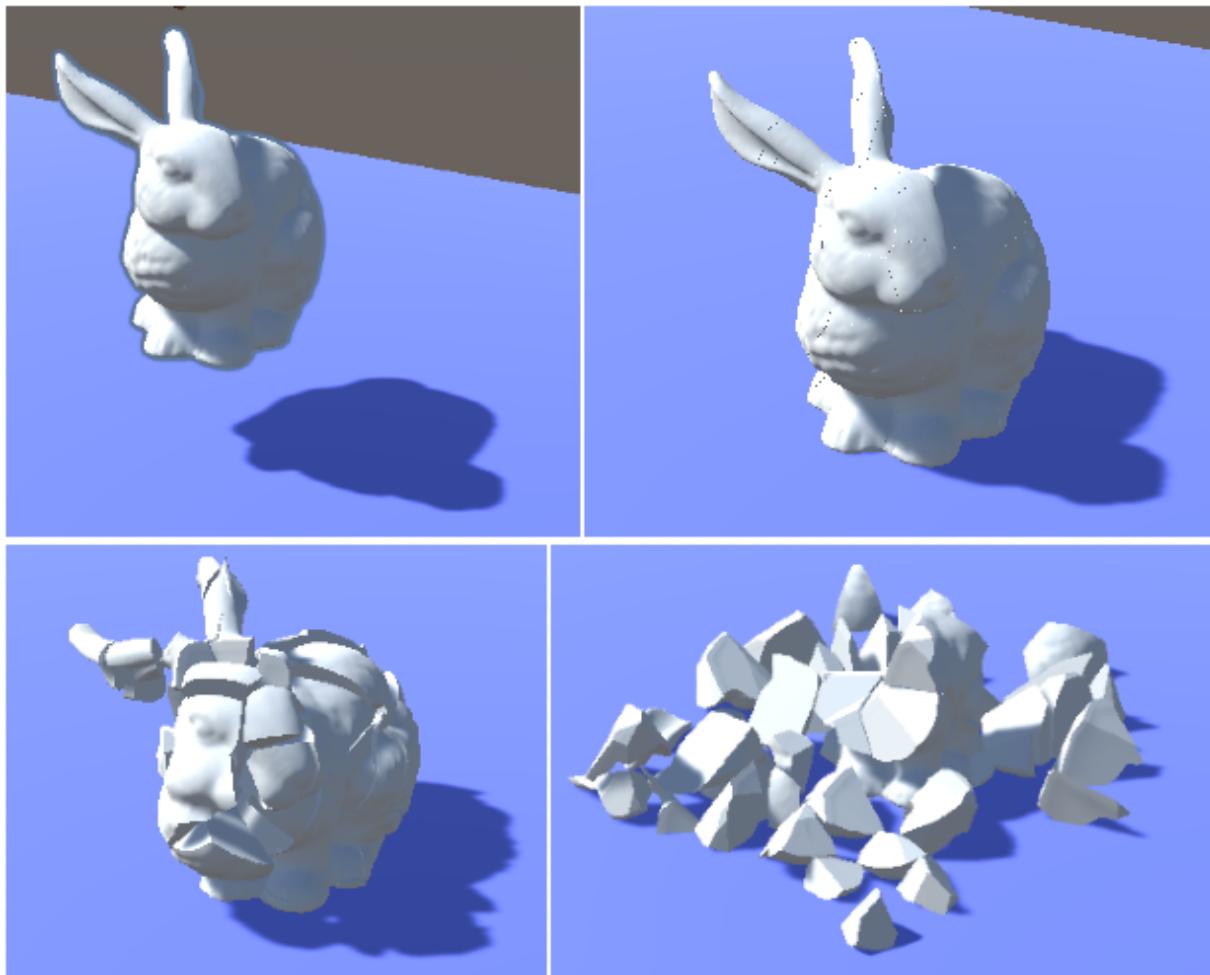


Abbildung 38: Oben links: Das 3D-Model wird aus einer Höhe fallengelassen.
 Oben rechts: Beim Aufprall mit dem Boden wird dieses mit dem gebrochenen Gegenstück ausgetauscht.
 Unten links: Fragmente zersplittern aufgrund der Geschwindigkeit des Aufpralls.
 Unten rechts: Ergebnis einer Fragmentierung eines Objektes mit ungefähr 50 Fragmenten. Quelle: Eigene Darstellung

Hier wurde das Model fallengelassen und bei der Kollision mit dem Boden mit dem gebrochenen Gegenstück ausgetauscht. Mit anderen Worten wurde die Kollision durch die eigene Bewegung des Objektes ausgelöst und liefert bei dieser Art von Kollision bereits beachtliche Ergebnisse. Wird jedoch das Objekt selber von einem Projektil getroffen, sollte bei dem Austauschen des Models eine Kraft auf die Fragmente angewendet werden.

Das Austauschen des Objektes ist zwar keine dynamische Methode, bei der die Fragmente zur Laufzeit und abhängig von der Einschlagstelle berechnet und erstellt werden, jedoch können wesentlich realistischere Ergebnisse erzielt werden, wenn eine Kraft abhängig des Kollisionspunkts auf die Fragmente angewendet wird. Dies kann beispielsweise mit der in Unity integrierten Methode *Rigidbody.AddExplosionForce* durchgeführt werden [80]. Damit wird eine Kraft auf

den Rigidbody ausgeübt, die einen Explosionseffekt simuliert. Die Explosion wird als Kugel mit einer bestimmten Position, einem Radius und einer Stärke definiert. Alles außerhalb der Kugel wird nicht von der Explosion betroffen und die Kraft nimmt proportional zum Abstand vom Mittelpunkt ab. Als Ausgangspunkt für die Kugel kann die Einschlagstelle gewählt werden, da diese Teil des *Collision*-Objektes ist. Der Radius kann abhängig von der Größe und Geschwindigkeit des eingehenden Projektil gewählt werden oder als ein konstanter Wert. Dadurch werden Fragmente die näher an dem Einschlagpunkt sind einer größeren Kraft ausgesetzt als Fragmente die weiter entfernt sind.

Um zu verhindern, dass die Szene nach dem Zerstören aufgrund der hohen Anzahl an Fragmenten unübersichtlich wird, kann nach dem Erstellen der Fragmente in Unity hinzugefügt werden, dass diese nach einer bestimmten Anzahl von Sekunden zerstört werden.

4.2.3 Voronoibasierte Zerstörung

Für die zweite Implementierung eines Algorithmus zum Zerstören von Objekten wurde ein dynamischer und voronoibasierter Ansatz ausgewählt. Das bedeutet, die Fragmente des 3D-Models werden zur Laufzeit berechnet und erstellt. Falls eine Einschlagstelle eines Projektils vorhanden ist, kann diese demnach in die Konstruktion der Fragmente mit einberechnet werden. Weiters erlaubt diese Methode im Gegensatz zum einfachen Austauschen des Objektes mit einem vorgebrochenen Gegenstück das mehrmalige Zerstören eines Models. Alle notwendigen Berechnungen dieser Methode müssen jedoch äußerst optimiert sein, um einen reibungslosen Ablauf ohne Einbrüche der Framerate zur Laufzeit zu gewähren.

Der grobe Ablauf des gesamten Algorithmus wird in folgendem Pseudocode gezeigt.

Algorithmus 2 : Abfolge der voronoibasierten Zerstörung

```
if Kollision then
    Initialisiere Punktmenge
    Berechne Delaunay Triangulation DT mit Bowyer-Watson-3D
    Extrahiere Voronoi Diagramm VD von DT
    foreach Zelle Z in VD do
        Initialisiere neues Fragment
        foreach Fläche von Z do
            Schneide das Mesh mit der Fläche
            Verwerfe alle Dreiecke auf der positiven Seite der Fläche (außerhalb)
            Füge Dreiecke auf negativen Seite der Fläche zu Fragment hinzu (innerhalb)
            Fülle das entstandene Loch mit neuen Dreiecken
        end
        Erstelle Unity-Mesh mit Dreiecken von Fragment
    end
    Lösche das originale Mesh
end
```

Zu Beginn der Entwicklung wurde der Einfachheit halber keine kollisionsabhängige Generierung der Punktmenge durchgeführt. Zuerst wird die Punktmenge mit zufälligen Punkten innerhalb der Grenzen des Meshes initialisiert. Die Grenzen eines Meshes können mithilfe der Unity-Komponente *MeshRenderer* abgefragt werden, da diese ein *bounds*-Attribut besitzt [79]. Dieses Attribut ist die *Axis-Aligned Bounding Box*, die das Model vollständig umschließt und kann verwendet werden, um grobe Annäherungen von der Position des Objekts und dessen Größe zu erhalten.

Nachdem die gewünschte Anzahl an Punkten generiert wurde, wird diese Punktmenge mithilfe des Bowyer-Watson Algorithmus trianguliert beziehungsweise zu Tetraedern geformt. Dafür wird zuerst ein Supertetraeder generiert, welches alle Punkte der Punktmenge enthält. Anschließend werden die Punkte einer nach dem anderen hinzugefügt und neu trianguliert, wie in Kapitel 2.6.5 erläutert. Zuletzt werden alle Tetraeder entfernt, die einen Punkt des Supertetraeder besitzen. Das Ergebnis ist eine Delaunay Triangulation in 3D und ist in Abbildung 39 ersichtlich. Der Aufwand dieser Berechnungen beträgt $O(n^2)$.

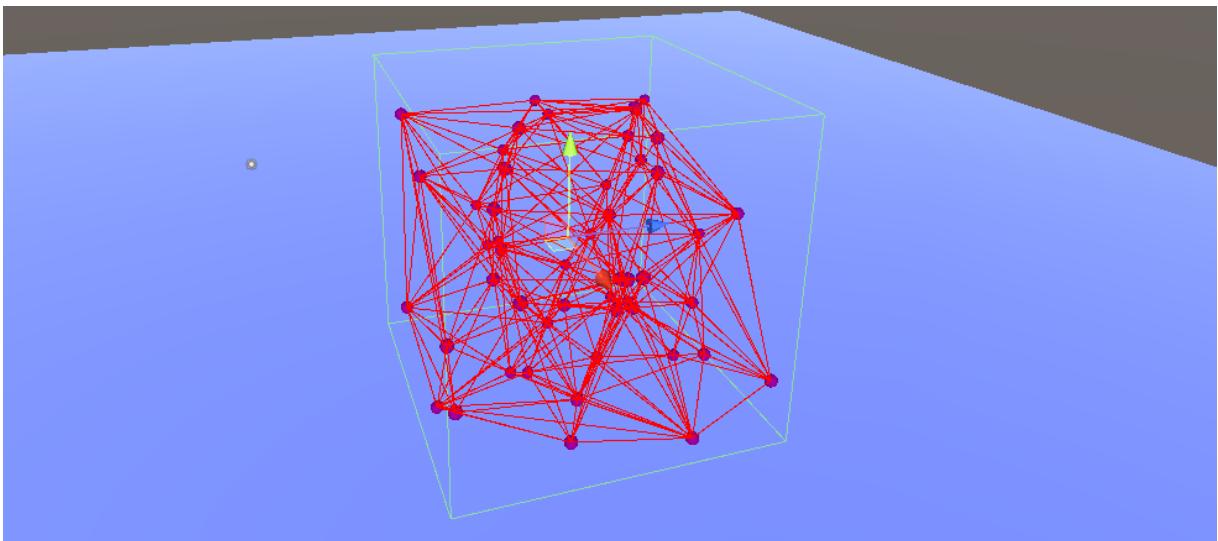


Abbildung 39: Resultat der Delaunay Triangulation in 3D mit 50 Punkten. Quelle: Eigene Darstellung

Nachdem die Delaunay Triangulation berechnet wurde, folgt die Umwandlung zum Voronoi Diagramm. Hierfür werden die ursprünglich generierten Punkte der Punktmenge durchlaufen. Für jeden Punkt werden die benachbarten Tetraeder in der Liste der Delaunay Triangulation gesucht und gespeichert. Anschließend wird für jede Kante, die an diesem Punkt angrenzt eine Ebene berechnet, die die Kante in ihrem Mittelpunkt rechtwinkelig teilt. Für jedes benachbarte Tetraeder wird der Mittelpunkt der Umkugel berechnet und jene Mittelpunkte, die sich auf der zuvor berechneten Ebene befinden, bilden eine gemeinsame Fläche einer Voronoi-Zelle. Nachdem dies mit jeder Kante durchgeführt wurde, ist das Resultat eine Liste an Flächen, die gemeinsam eine Zelle binden. Wurden diese Berechnungen zu einer Zelle für jeden Punkt durchgeführt, bilden diese infolgedessen das Voronoi Diagramm. Der Aufwand für diese Umwandlung ist ebenfalls $O(n^2)$. Das Resultat mit der gleichen Punktmenge wie in Abbildung 39 wird in Abbildung 40 gezeigt.

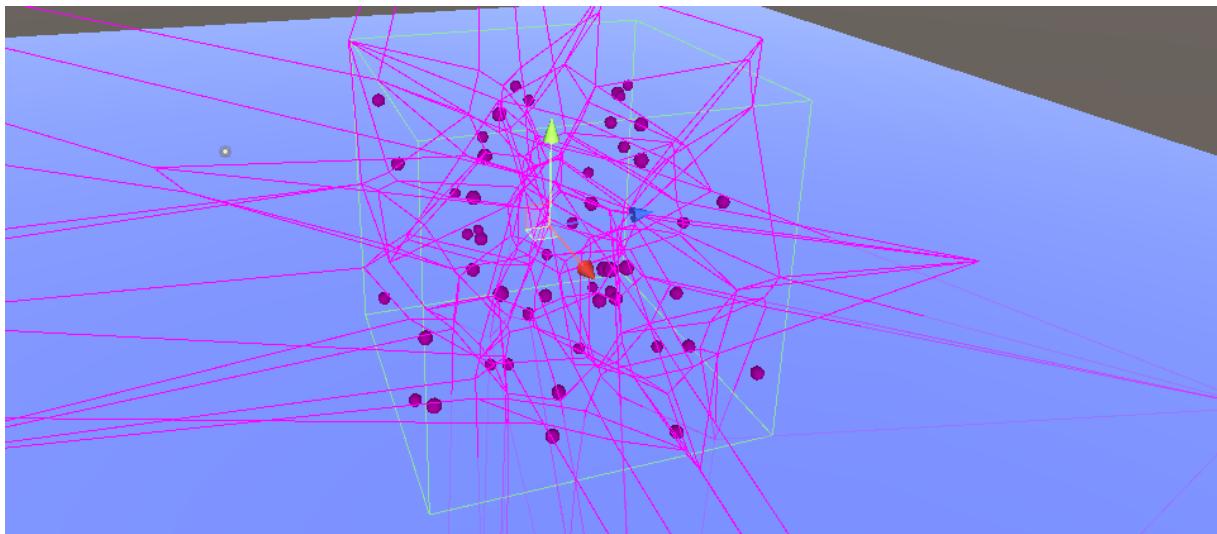


Abbildung 40: Voronoi Diagramm in 3D mit 50 Punkten. Quelle: Eigene Darstellung

Um die Voronoi Zellen besser und einfacher auf Fehler zu untersuchen wurde eine Methode implementiert die, die Voronoi Zellen zu Meshes umwandelt und anschließend Unity-Gameobjects für jede Zelle erstellt. Dies wird erreicht, indem für jede Zelle jede Fläche mithilfe von Bowyer Watson 2D trianguliert wird, da für die Darstellung eines Meshes in Unity Dreiecke benötigt werden. Jedoch ist eine einfache Liste von Dreiecken nicht ausreichend. Diese wird noch weiter umgewandelt in eine Liste von Punkten und eine Liste von Indizes. Die Liste der Indizes verweist auf die Liste von Eckpunkten und definiert in Dreierschritten welche Punkte gemeinsam ein Dreieck bilden.

Der letzte Schritt ist das Berechnen des Durchschnitts jeder Voronoi Zelle mit dem zu zerstörenden Mesh, da dieser die jeweiligen Fragmente bildet. Dafür werden zuerst die Informationen über das Mesh gelesen, was so viel bedeutet, als dass die Liste von Punkten und die Liste von Indizes in eine Liste von Dreiecken umgewandelt wird. Dies erleichtert das Manipulieren der Liste der vorhandenen Dreiecke während des Schneidens des Meshes. Danach wird mit jeder Fläche einer Zelle jedes Dreieck des Meshes geschnitten, wie bereits in Kapitel 3.4.2 erläutert wurde. Dabei wird eine Fläche einer Zelle als eine Ebene betrachtet. Daraus ergeben sich drei mögliche Fälle:

1. Alle Punkte des Dreiecks befinden sich auf der positiven Seite der Ebene
2. Alle Punkte des Dreiecks befinden sich auf der negativen Seite der Ebene
3. Das Dreieck wird von der Ebene geschnitten

Beim ersten Fall befindet sich das gesamte Dreieck auf der positiven Seite der Ebene, was so viel bedeutet, als dass das Dreieck außerhalb der Zelle ist und demnach nicht Teil des Durchschnitts ist. Daher wird dieses aus der Liste entfernt und mit dem nächsten Dreieck fortgesetzt.

Befindet sich das gesamte Dreieck auf der negativen Seite der Fläche, ist dieses möglicherweise innerhalb der Voronoi Zelle und wird demnach nicht aus der Liste entfernt, sondern es wird ohne Weiteres mit dem nächsten Dreieck fortgesetzt.

Die dritte Option ist die komplexeste der drei. Hier wird das Dreieck von der Ebene geschnitten, was bedeutet, dass zwei Punkte auf einer Seite der Ebene sind und der Dritte auf der anderen Seite. Hierfür wird zuerst dieses Dreieck aus der Liste entfernt, da dieses aufgetrennt werden muss. Abhängig von der Anzahl der Punkte innerhalb der Zelle werden anschließend ein oder zwei neue Dreiecke hinzugefügt (siehe Abbildung 32, Kapitel 3.4.2).

Wurde jedes Dreieck von dieser Fläche geschnitten und der dementsprechende Fall durchgeführt, wird bevor mit der nächsten Ebene fortgesetzt wird das entstandene Loch mit Dreiecken gefüllt, um zum Schluss ein geschlossenes Fragment zu erhalten. Nachdem die Dreiecke, die das Loch füllen hinzugefügt wurden, wird diese Liste der aktuellen Dreiecke als Basis für die nächste Ebene verwendet. Sind alle Flächen einer Voronoi Zelle betrachtet worden, ist der Durchschnitt der Zelle und des Meshes berechnet und eine Liste von Dreiecken ist das Resultat. Der benötigte Aufwand für diese Berechnungen beträgt $O(n)$, wobei n in diesem Fall die Anzahl der Zellen beschreibt und nicht die Anzahl der Punkte der Punktmenge. Anschließend wird dies wiederum zu einem Unity-Gameobject umgewandelt, indem diese Liste wieder in eine Liste von Punkten und eine Liste von Indizes aufgeteilt wird, um eine Darstellung in Unity zu ermöglichen.

Eine wichtige Optimierung wurde bei dem Füllen der Löcher durchgeführt. Da beim Schneiden der Ebene mit Dreiecken gearbeitet wird, entstehen rasch viele schlecht geformte Dreiecke, wie in Abbildung 41 zu sehen ist.

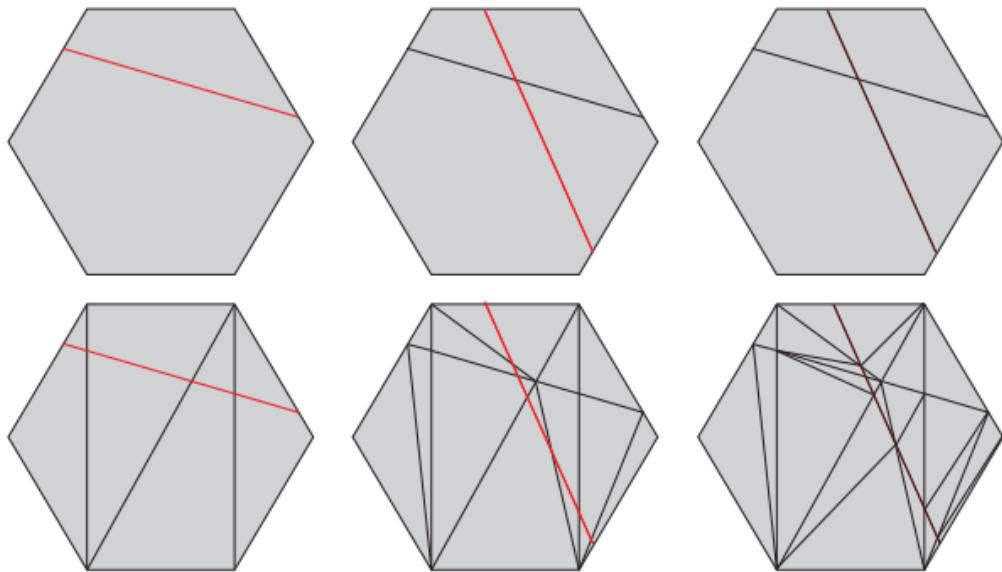


Abbildung 41: Unterschied zwischen dem Verwenden von Polygonen und Dreiecken. Hier wird ein Sechseck zweimal geteilt. In der oberen Reihe wird dieses als Polygon behandelt und in der unteren Reihe als triangulierte Fläche. Im letzteren Fall entstehen nach nur zwei Schnitten viele schlecht geformte Dreiecke. [50]

Da jedoch Unity-Meshes aus Dreiecken bestehen und es keine Informationen darüber gibt, welche Dreiecke eine Fläche bilden, muss weiter mit Dreiecken gearbeitet werden. Um nichtsdestotrotz das Schneiden nun effizienter zu machen, wurde das Füllen der Löcher optimiert, da hier sehr viele redundante Punkte vorhanden sind. Ein Vergleich des Ergebnisses mit und ohne dieser Optimierung wird in Abbildung 42 gezeigt.

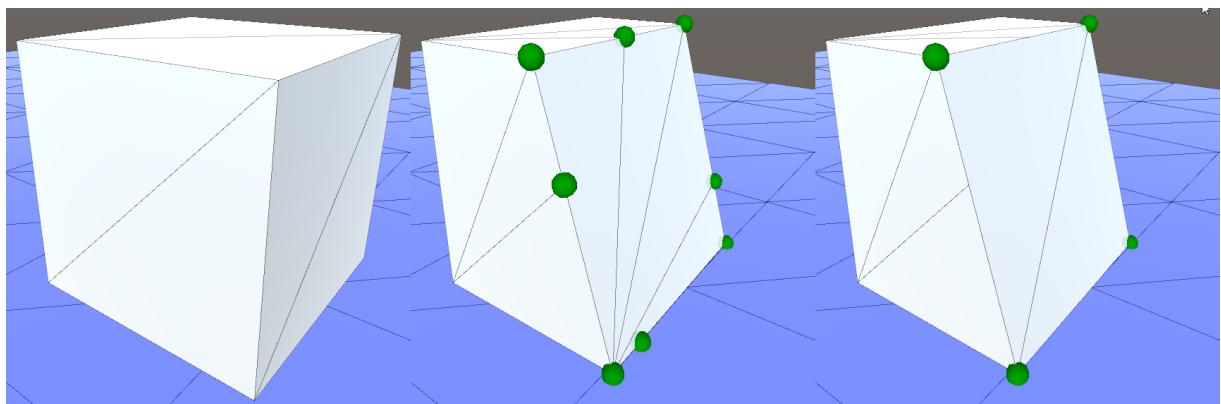


Abbildung 42: Entfernen von redundanten Punkten beim Füllen der Löcher verringert die Anzahl der Dreiecke und die benötigte Zeit. Quelle: Eigene Darstellung

Hier werden die Punkte, die zum Füllen des Loches verwendet werden, zuerst im Uhrzeigersinn sortiert. Dies ist möglich, da alle Punkte auf derselben Ebene liegen, nämlich jene die zum Schneiden verwendet wird. Anschließend werden jeweils drei Punkte untersucht und sind diese

kollinear, also liegen auf einer Geraden, wird der mittlere Punkt entfernt. Dadurch wird für die Triangulierung des Loches die minimale Anzahl an Punkten verwendet.

5 Ergebnisse und Diskussion

In diesem Kapitel werden die Ergebnisse des Vergleichs der beiden implementierten Methoden aus Kapitel 4 vorgestellt, um zu ermitteln, ob es einen Unterschied in der Leistung der beiden getesteten Methoden gibt. Weiters wird die Qualität der Fragmente von beiden Methoden evaluiert. Zuletzt werden die Methoden in der Komplexität der Implementierung gegenübergestellt.

5.1 Leistung

Der Vergleich der Performanz der beiden Methoden, Austausch mit einem vorgebrochenen Gegenstück und voronoibasierte dynamische Zerstörung, wurde durch eine quantitative Analyse der Leistung abgeschlossen. Die Messungen wurden mit dem in Kapitel 4.1.1 Setup durchgeführt. Um die Leistung der beiden Methoden zu testen, wurde die Zerstörung der jeweiligen Methode nach dem Starten des Programms ausgeführt und die Zeit die benötigt wurde gemessen. Diese Messungen wurden mit unterschiedlicher Anzahl an Fragmenten durchgeführt. Das Messen für eine bestimmte Anzahl von Fragmenten wurde ebenfalls mehrmals ausgeführt, um einen umfangreichen Datensatz für die Analyse zu erhalten und um Fehler zu vermeiden. Das wiederholte Testen der Methoden wurde erreicht durch das Schreiben eines Skriptes welches die Szene nach der Zerstörung neu lädt. Bei jedem neuen Durchgang werden die Zeiten in eine Textdatei aufgezeichnet. Dieser Vorgang wurde für jede Anzahl von Fragmenten mindestens zehnmal wiederholt um genügend Daten analysieren zu können.

Für die erste der beiden Methoden, dem Austausch des Objekts mit einem vorgebrochenen Gegenstück, wurde die Zeit mit mehreren Modellen gemessen. Dafür wurden wie in Kapitel 4.2.2 beschrieben, in Blender mehrere Modelle mit unterschiedlicher Anzahl von Fragmenten erstellt und anschließend in Unity importiert. Die Messungen wurden mit Modellen mit einer Anzahl von 10, 25, 50, 100, 150 und 200 Fragmenten durchgeführt.

Bei dieser Methode wurde mit zwei unterschiedlichen Möglichkeiten das Objekt auszutauschen gearbeitet. Eine Möglichkeit ist das Objekt beim Zeitpunkt der Kollision mit der *Instantiate()*-Funktion zu erstellen und das originale Objekt mittels *Destroy()* zu entfernen. Bei der zweiten wird das Objekt bereits zu Programmstart instanziert, jedoch auf inaktiv gesetzt. Tritt die Kollision auf, wird die Position des gebrochenen Gegenstücks aktualisiert und auf aktiv gesetzt.

Anschließend wird das originale Objekt wieder aus der Szene entfernt. Die Ergebnisse der Messungen werden in Abbildung 43 dargestellt.

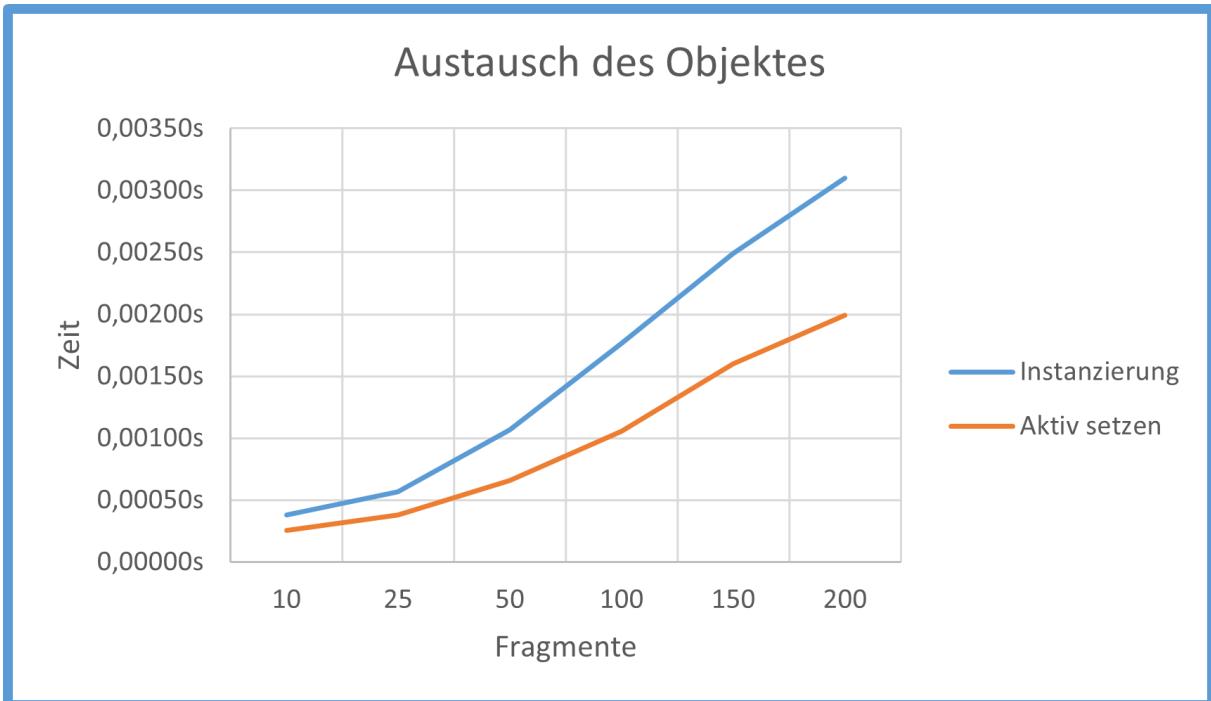


Abbildung 43: Austausch des Objekts mit zwei unterschiedlichen Methoden. Quelle: Eigene Darstellung

Bei beiden Arten ist ein linearer Aufwand erkennbar, da über die Fragmente einmal iteriert wird, um die entsprechende Kraft auf die Fragmente anzuwenden. Bei der Methode, wo das Objekt bereits vorher instanziert wird und zur Kollision auf aktiv gesetzt wird, ist das Austauschen effizienter, da das Objekt bereits zu Beginn des Levels oder bei Programmstart geladen wird. Da jedoch beide Methoden eine extrem geringe Zeit in Anspruch nehmen ist dieser Unterschied nicht erkennbar.

Die zweite implementierte Methode zum Zerstören von Objekten ist eine voronoibasierte, prozedurale Methode. Hier wurde wie bei der ersten Methode die Zerstörung nach Programmstart initiiert und die benötigte Zeit bis zum Erstellen der Fragmente gemessen. Bei dieser Methode wurde die Zeit zusätzlich in drei Teile eingeteilt: das Erstellen des Delaunay Triangulation, das Umwandeln zu dem Voronoi Diagramm und das Schneiden des Meshes. Die Messungen werden in Abbildung 44 gezeigt.

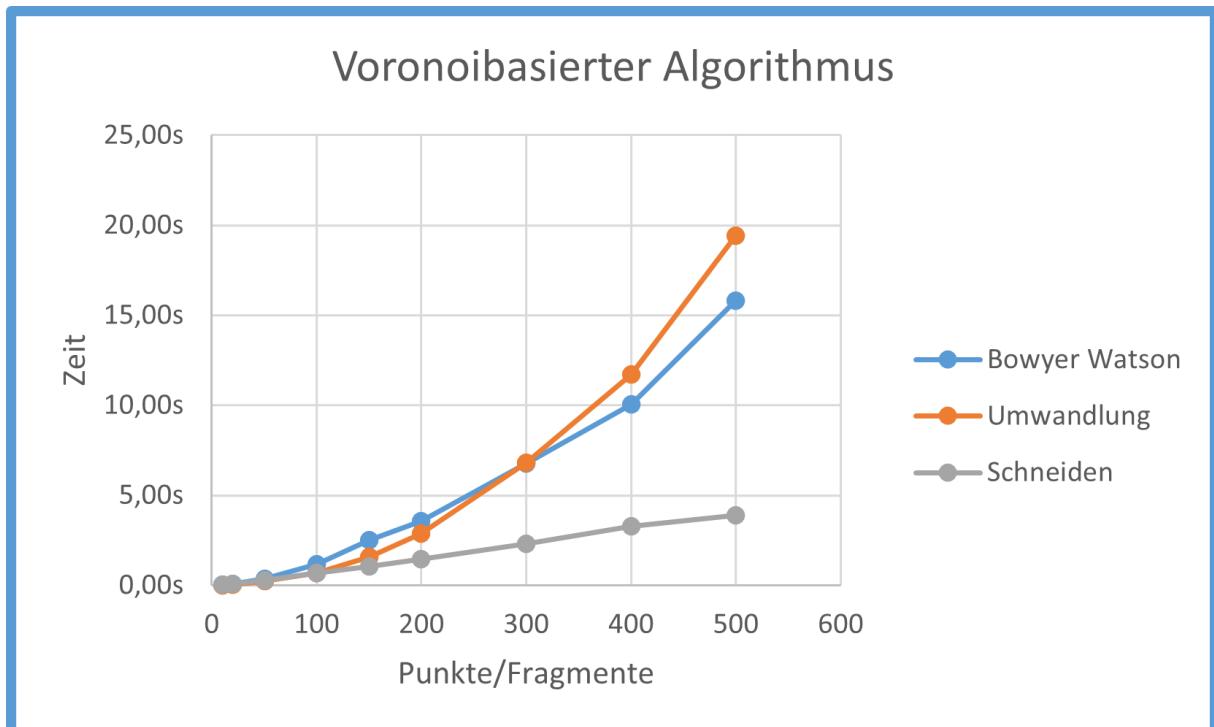


Abbildung 44: Messungen der einzelnen Teile des voronoibasierten Algorithmus. Quelle: Eigene Darstellung

In dieser Grafik ist erkennbar, dass im Vergleich zu der ersten Methode nur bei einer geringen Anzahl von Fragmenten wie beispielsweise zehn eine Zerstörung in Echtzeit möglich ist. Um eine höhere Anzahl an Fragmenten effizient genug erzeugen zu können, müsste vor allem das Konstruieren des Voronoi Diagramms optimiert werden, da sowohl der Bowyer-Watson Algorithmus als auch die Umwandlung einen quadratischen Aufwand besitzen. Eine weitere Möglichkeit wäre das Generieren des Voronoi Diagramms zur Laufzeit auszulassen und stattdessen vorgefertigte Bruchmuster zu verwenden wie in der Arbeit von Müller et al. [50]. Dadurch ist es trotzdem möglich dynamisch Fragmente zu erschaffen, indem das Bruchmuster mit der Aufprallstelle abgestimmt wird, jedoch ohne aufwendige Berechnungen des Voronoi Diagramms. Um diese Bruchmuster zur Entwicklungszeit zu erzeugen, kann das Voronoi Diagramm verwendet werden, da die Größe und Form der Fragmente definiert werden kann. Dadurch können verschiedene Bruchmuster für unterschiedliche Materialien verwendet werden.

5.2 Qualität

Um die beiden Methoden in ihrer Qualität zu analysieren, wurden folgende Kriterien definiert:

- Die Anzahl der Fragmente

- Möglichkeit eines mehrmaligen Zerstörens beziehungsweise einer lokalen Fragmentierung des Objektes
- Bestimmung von Größe und Form der Fragmente

Die mögliche Anzahl der Fragmente ist ein wichtiges Kriterium für die Qualität der Zerstörung, da mit einer höheren Anzahl spektakulärere und detailliertere Zerstörungen simuliert werden können. Dieses Kriterium steht im direkten Zusammenhang mit der Leistung der Algorithmen und dementsprechend spiegeln sich die Ergebnisse hier wider. Auch wenn die voronoibasierte Methode optimiert wird und eine höhere Anzahl an Fragmenten in Echtzeit zulässt, ist es nicht möglich, an die Anzahl der Fragmente der Methode des Austauschens heranzukommen. Infolgedessen stellt sich hier der Austausch des Objektes als die bessere Variante heraus.

Das nächste Qualitätskriterium gibt an, ob es möglich ist ein Objekt mehrmals zu zerstören, beziehungsweise ob eine lokale Fragmentierung möglich ist. Eine lokale Fragmentierung bedeutet, das nicht das gesamte Objekt zerstört wird, sondern nur kleine Fragmente abbröckeln. Bei der Methode des Austauschens des Objekts mit einem vorgebrochenen Gegenstück, ist es zwar möglich mehrere Stufen zu modellieren, bevor das Objekt bricht, jedoch kann die eigentliche Fragmentierung nur einmal durchgeführt werden. Ebenso ist eine lokale Fragmentierung bei dieser Methode nicht möglich, da hierfür die Einschlagstelle zur Berechnung benötigt wird. Die einzige Ausnahme dafür sind Zwischensequenzen in Videospielen, da diese einem vorher festgelegten Skript folgen. Bei der voronoibasierten Methode hingegen können die Objekte mehrfach zerstört werden und eine lokale Fragmentierung ist ebenfalls möglich, da die Fragmente zur Laufzeit berechnet werden. Dadurch können diese abhängig von einer Einschlagstelle konstruiert werden.

Das letzte Kriterium gibt an, ob die Fragmente in ihrer Größe und Form mithilfe von Parametern verändert werden können, um unterschiedliche Materialien simulieren zu können. Beispielsweise sollten die Fragmente von Glas, Beton und Holz jeweils unterschiedliche Formen, Größen und Kanten haben um ein realistisches Ergebnis zu erhalten. Bei der Methode des Austauschens kann das entsprechende gebrochene Gegenstück entweder von einem 3D-Artist erstellt werden oder wie in dem Projekt dieser Arbeit durch zur Verfügung stehende Software wie das *Cell Fracture Add-On* von Blender. Der 3D-Artist kann die Fragmente so modellieren, wie diese im Spiel benötigt werden. Das *Cell Fracture Add-On*, welches hier verwendet wurde, bietet eine Vielzahl an Parameter um die gewünschten Fragmente zu erhalten. Dadurch erfüllt diese Methode das Kriterium. Ein ähnliches Ergebnis findet sich bei der voronoibasierten Methode wieder. Hier kann ebenfalls die Größe und Form der Fragmente variiert werden, da diese abhängig von der generierten Punktmenge erstellt werden. Allerdings müssen die Parameter, die das Erstellen der Punktmenge modifizieren, zuerst implementiert und hinzugefügt werden.

Nachdem beide Methoden mithilfe der genannten Kriterien analysiert wurden, lässt sich nicht eindeutig feststellen, welche Methode eine bessere Qualität besitzt. Bei beiden Methoden kann

die gewünschte Form und Größe der Fragmente definiert werden. Die Methode des Austauschens lässt zwar eine deutlich höhere Anzahl an Fragmenten zu, jedoch kann diese Methode keine dynamische Zerstörung simulieren, wie es bei der voronoibasierten Methode der Fall ist.

5.3 Komplexität

Ein dritter wichtiger Aspekt für die Analyse eines Algorithmus ist die Komplexität der Implementierung. Algorithmen, die komplex zu implementieren sind, erfordern qualifizierte Entwickler*innen, mehr Zeit für die Implementierung und weisen ein höheres Risiko von Implementierungsfehlern auf. Um die Komplexität zu messen und zu vergleichen wurden zwei bestehende Kriterien ausgewählt: Cyclomatic Complexity [48] und Cognitive Complexity [13].

Die Cyclomatic Complexity Zahl beschreibt die Komplexität des Programmablaufs des Algorithmus. Diese wird für jede Methode berechnet und entspricht der Anzahl der Verzweigungen in der Methode. Außerdem erhöht sich diese Zahl für jeden logischen „Und“ (`&&`) und „Oder“ (`||`) Operator. Je größer diese Zahl ist, desto mehr Ausführungspfade gibt es in der Funktion. Dabei ist zu beachten, dass die Cyclomatic Complexity unabhängig von der Komplexität der verwendeten Datenstruktur ist. Während die Cyclomatic Complexity die Testbarkeit und Wartbarkeit genau misst, liefert sie bei der Messung der Verständlichkeit keine akkurate Ergebnisse. Um dieses Problem zu lösen wurde die Cognitive Complexity entwickelt und wird mithilfe folgender drei Regeln berechnet:

- Ignorieren von Strukturen, die es ermöglichen, mehrere Anweisungen lesbar zu einer einzigen zusammenzufassen, wie beispielsweise das Aufteilen in Methoden
- Erhöhung um eins für jede Unterbrechung im linearen Ablauf des Codes. Dazu zählen: Schleifen, Verzweigungen und `try-catch`-Anweisungen
- Erhöhung, wenn die eben genannten Strukturen verschachtelt sind

Für den Vergleich der beiden Algorithmen wird die Cyclomatic Complexity und die Cognitive Complexity jeder verwendeten Methode eines Algorithmus berechnet und summiert. Die Ergebnisse werden in folgender Tabelle 1 dargestellt:

	Austausch des Objektes	Voronoibasierter Algorithmus
Cyclomatic Complexity	5	76
Cognitive Complexity	3	207
Gesamt	8	283

Tabelle 1

In dieser Tabelle ist ersichtlich, dass der voronoibasierte Algorithmus sowohl eine deutlich höhere Cyclomatic Complexity als auch Cognitive Complexity besitzt und infolgedessen eine komplexere Methode zu implementieren ist. Dies ist vor allem darauf zurückzuführen, dass die Fragmente zur Laufzeit berechnet und erstellt werden müssen und nicht wie beim Austausch des Objektes bereits vorher konstruiert werden. Bei der Methode des Austauschens ist die Implementierung des Austauschens an sich keine komplizierte Aufgabe und besitzt demnach nur eine geringe Komplexität.

Die Ergebnisse zeigen, dass sich von den evaluierten Methoden der Austausch des Objektes als die bessere Methode herausstellt. Diese liefert für die meisten Szenarien vielversprechende Ergebnisse in der Leistung und Qualität der Zerstörung und ist mit wenig Komplexität und Aufwand verbunden. Ist jedoch die Zerstörung von Objekten eine Grundmechanik eines entwickelten Spiels, sollte eine dynamische Methode in Betracht gezogen werden. Hierfür muss allerdings genügend Zeit mit einberechnet werden, um ein effizientes System implementieren zu können.

6 Zusammenfassung und Ausblick

Ziel dieser Masterarbeit war es, einen Überblick über aktuell verwendete Methoden zum Zerstören von 3D-Objekten in Videospielen zu geben. Weiters sollte ein Einblick in die Implementierung dieser Methoden gegeben werden. Dies wurde durch eine extensive Literaturrecherche und das Entwickeln eines Unity-Projekts erreicht.

In dieser Arbeit wurden zwei Methoden, um Objekte in Echtzeitanwendungen zerstören zu können gegenübergestellt. Dies erfolgte durch die Implementierung dieser zwei Methoden, nämlich dem Austausch des Objektes mit einem vorgebrochenen Gegenstück und einer dynamischen voronoibasierten Methode. Die Leistung der beiden Methoden wurde durch eine quantitative Analyse verglichen, indem die Zeit von Kollisionsbeginn bis zum Erstellen aller Fragmente gemessen wurde. Dies wurde mehrfach wiederholt, um umfangreiche Daten zu erhalten. Weiters wurden die implementierten Methoden in der Qualität der Zerstörung und der Komplexität der Implementierung verglichen.

Die Untersuchung hat gezeigt, dass in der Leistung ein erheblicher Unterschied zwischen den beiden Methoden existiert und, dass der Austausch des Objektes um ein Vielfaches schneller ist. Dies liegt aber auch daran, dass im Rahmen dieser Arbeit nur eine grundlegende Version der voronoibasierten Methode implementiert werden konnte und Optimierungen benötigt werden. Beispiele für diese Optimierungen wären eine verbesserte Suche der schlechten Dreiecke des Bowyer-Watson Algorithmus um einen Aufwand von $O(n \log n)$ zu erhalten und das Parallelisieren des Schneidens des Meshes. Bei der Qualität der Zerstörung der Algorithmen gab es keine vorherrschende Methode, vielmehr ist jede Methode für unterschiedliche Szenarien besser geeignet. Zuletzt wurden die Algorithmen in der Komplexität der Implementierung verglichen und dabei stellt sich heraus, dass die voronoibasierte Methode wesentlich komplexer und schwieriger zu implementieren ist. Dies liegt vor allem daran, dass die Fragmente zur Laufzeit berechnet und erstellt werden müssen und dass das Mesh geschnitten werden muss.

Zukünftige Forschung könnte an dieser Arbeit anknüpfen und einen Vergleich mit einer physikalisch basierten Methode, wie dem Mass-Spring Model oder der Finite-Elemente-Methode durchführen. Weiters waren die in dieser Arbeit verwendeten Methoden auf der Simulation von spröden Materialien beschränkt. Durch das Implementieren einer physikalisch basierten Methoden wäre es möglich verformbare Materialien zu simulieren und dabei stellt sich die Frage, ob dies die Qualität der Zerstörung verbessern würde.

Literaturverzeichnis

- [1] DMM Engine. <https://www.pixelux.com/DMMengine.html>. Abgerufen: 12. Jänner 2022.
- [2] Quantum Break. <https://www.remedygames.com/games/quantumbreak/>, Nov 2021. Abgerufen: 12. Jänner 2022.
- [3] 2KLIKSPHILIP: *Red Faction's Destruction - Best Ever?*. <https://youtu.be/S7txd9QI8eg>, 2019. Abgerufen: 06. September 2021.
- [4] ANSARI, M.: *Game Development Tools*. EBL-Schweitzer. CRC Press, 2016.
- [5] ASHBY, M. F., H. SHERCLIFF und D. CEBON: *Materials: engineering, science, processing and design*. Butterworth-Heinemann, 2018.
- [6] AURENHAMMER, F.: *Voronoi diagrams—a survey of a fundamental geometric data structure*. ACM Computing Surveys (CSUR), 23(3):345–405, 1991.
- [7] AURENHAMMER, F. und H. EDELSBRUNNER: *An optimal algorithm for constructing the weighted voronoi diagram in the plane*. Pattern Recognition, 17(2):251–257, 1984.
- [8] AURENHAMMER, F., R. KLEIN und D.-T. LEE: *Voronoi Diagrams and Delaunay Triangulations*. WORLD SCIENTIFIC, 2013.
- [9] BAUMGART, B. G.: *A Polyhedron Representation for Computer Vision*. In: *Proceedings of the May 19-22, 1975, National Computer Conference and Exposition*, AFIPS '75, S. 589–596, New York, NY, USA, 1975. Association for Computing Machinery.
- [10] BHATTACHARYA, P. und M. GAVRILOVA: *Voronoi diagram in optimal path planning*. Proceedings - ISVD 2007 The 4th International Symposium on Voronoi Diagrams in Science and Engineering 2007, S. 38–47, 07 2007.
- [11] BLENDER-DOCUMENTATION: *Cell Fracture*. https://docs.blender.org/manual/en/2.93/addons/object/cell_fracture.html, 2021. Abgerufen: 15. November 2021.
- [12] BOWYER, A.: *Computing Dirichlet tessellations**. The Computer Journal, 24(2):162–166, 01 1981.
- [13] CAMPBELL, G. A.: *Cognitive Complexity: An Overview and Evaluation*. In: *Proceedings of the 2018 International Conference on Technical Debt*, TechDebt '18, S. 57–58, New York, NY, USA, 2018. Association for Computing Machinery.

- [14] DAVIDSON, J.: *A Comparison of Fracture Techniques For Use in Video Games*. Masterarbeit, Heriot-Watt University, 2018.
- [15] DE BERG, M., M. VAN KREVELD, M. OVERTMARS und O. SCHWARZKOPF: *Computational geometry*. Springer, 1997.
- [16] DEEP-SILVER-VOLITION: *Red Faction*. <https://www.dsvolition.com/games/red-faction-guerrilla>, 2001. Abgerufen: 09. August 2021.
- [17] DEUSSEN, O.: *Aesthetic Placement of Points Using Generalized Lloyd Relaxation*. In: DEUSSEN, O. und P. HALL (Hrsg.): *Computational Aesthetics in Graphics, Visualization, and Imaging*. The Eurographics Association, 2009.
- [18] DOBRANSKÝ, M.: *Efficient simulation of environment destruction in games*. Bachelorarbeit, Charles University, 2017.
- [19] DOLBILIN, N.: *Boris Nikolaevich Delone (Delaunay): Life and work*. Proceedings of the Steklov Institute of Mathematics, 275, 12 2012.
- [20] FISHER, J.: *Visualizing the Connection Among Convex Hull , Voronoi Diagram and Delaunay Triangulation*. Proceedings of the Thirty-seventh Midwest Instruction and Computing Symposium, 2004.
- [21] FOLEY, J. D., A. VAN DAM, S. FEINER und J. F. HUGHES: *Computer graphics - principles and practice, 2nd Edition*. Addison-Wesley, 1990.
- [22] FORTUNE, S.: *A Sweepline Algorithm for Voronoi Diagrams*. In: *Proceedings of the Second Annual Symposium on Computational Geometry*, SCG '86, S. 313–322, New York, NY, USA, 1986. Association for Computing Machinery.
- [23] FRAZIER, R.: *Rendering constructive solid geometry with python*. <https://www.fotonixx.com/posts/efficient-csg/#the-full-function>, Mar 2021. Abgerufen: 15. März 2022.
- [24] FREY, P. J. und H. BOROUCHAKI: *Geometric evaluation of finite element surface meshes*. Finite Elements in Analysis and Design, 31(1):33–53, 1998.
- [25] GAHEGAN, M. und I. LEE: *Data structures and algorithms to support interactive spatial analysis using dynamic Voronoi diagrams*. Computers, Environment and Urban Systems, 24(6):509–537, 2000.
- [26] GLONDU, L.: *Physically-based and Real-time Simulation of Brittle Fracture for Interactive Applications*. Theses, École normale supérieure de Cachan - ENS Cachan, Nov. 2012.
- [27] GONG, Y., G. LI, Y. TIAN, Y. LIN und Y. LIU: *A vector-based algorithm to generate and update multiplicatively weighted Voronoi diagrams for points, polylines, and polygons*. Computers & Geosciences, 42:118–125, 2012.

- [28] GRÖNBERG, A.: *Real-time Mesh Destruction System for a Video Game*. Bachelorarbeit, Luleå University Of Technology,, 2017.
- [29] GÄRTNER, B. und M. HOFFMANN: *Computational Geometry Lecture Notes HS 2013*. 2014.
- [30] GUANGMING, L., T. JIE, Z. MINGCHANG, H. HUIGUANG und Z. XIAOPENG: *A new mesh simplification algorithm combining half-edge data structure with modified quadric error metric*. In: *2002 International Conference on Pattern Recognition*, Bd. 2, S. 659–662 vol.2, 2002.
- [31] HENRIQUES, A. und B. WÜNSCHE: *Real-Time Interaction Techniques for Meshless Deformation Based on Shape Matching*. 03 2022.
- [32] HUBBARD, P. M.: *Constructive Solid Geometry for Triangulated Polyhedra*, 1990.
- [33] IMAGIRE, T., H. JOHAN und T. NISHITA: *A fast method for simulating destruction and the generated dust and debris*. The Visual Computer, 25:719–727, 2009.
- [34] JÜNGER, M., V. KAIBEL und S. THIENEL: *Computing Delaunay Triangulations in Manhattan and Maximum Metric*. Techn. Ber., INSTITUT FÜR INFORMATIK, UNIVERSITÄT ZU KÖLN, 1995.
- [35] JU, L., T. RINGLER und M. GUNZBURGER: *Voronoi Tessellations and Their Application to Climate and Global Modeling*, Bd. 80, S. 313–342. 02 2011.
- [36] KAMIKAZETUTOR: *Good Idea, Bad Idea: Destructible Environments*. <https://www.destructoid.com/good-idea-bad-idea-destructible-environments-68780.phtml>, Feb 2008. Abgerufen: 9. August 2021.
- [37] KIRKPATRICK, D. G.: *Efficient computation of continuous skeletons*. 20th Annual Symposium on Foundations of Computer Science (sfcs 1979), S. 18–27, 1979.
- [38] KLITGAARD, N. und R. LOLL: *Introducing Quantum Ricci Curvature*. Physical Review D, 97, 12 2017.
- [39] LEDOUX, H.: *Computing the 3D Voronoi Diagram Robustly: An Easy Explanation*. In: *4th International Symposium on Voronoi Diagrams in Science and Engineering (ISVD 2007)*, S. 117–129, 2007.
- [40] LEDOUX, H. und C. GOLD: *Modelling three-dimensional geoscientific fields with the Voronoi diagram and its dual*. International Journal of Geographical Information Science, 22:547–574, 05 2008.
- [41] LEE, D. T. und R. L. DRYSDALE, III: *Generalization of Voronoi Diagrams in the Plane*. SIAM Journal on Computing, 10(1):73–87, 1981.

- [42] LEHMANN, D. J. und H. THEISEL: *The LloydRelaxer: An Approach to Minimize Scaling Effects for Multivariate Projections*. IEEE Transactions on Visualization and Computer Graphics, 24:2424–2439, 2018.
- [43] LUIS, M., S. SALHI und G. NAGY: *A Constructive Method and a Guided Hybrid GRASP for the Capacitated Multi-Source Weber Problem in the Presence of Fixed Cost*. Journal of Algorithms & Computational Technology, 9:215–232, 06 2015.
- [44] MAHANKALI, R.: *Mesh-Plane Clipping / GeomAlgoLib*. <https://www.youtube.com/watch?v=t6VvtW8y9q4>, 2020. Abgerufen: 01. März 2022.
- [45] MARINELLI, N. D.: *Hardware Accelerated Environment Deformation*. 2007.
- [46] MAUL, P., M. MUELLER, F. ENKLER, E. PIGOVA, T. FISCHER und L. STAMATOGIANNAKIS: *BeamNG.tech Technical Paper*, 2021.
- [47] MAZARAK, O., C. MARTINS und J. AMANATIDES: *Animating Exploding Objects*. In: *Proceedings of the Graphics Interface 1999 Conference, June 2-4, 1999, Kingston, Ontario, Canada*, S. 211–218, June 1999.
- [48] McCABE, T.: *A Complexity Measure*. IEEE Transactions on Software Engineering, SE-2(4):308–320, 1976.
- [49] MEGURO, K. und M. HAKUNO: *Fracture Analyses of Concrete Structures by the Modified Distinct Element Method*. Doboku Gakkai Ronbunshu, 1989:113–124, 1989.
- [50] MÜLLER, M., N. CHENTANEZ und T. KIM: *Real Time Dynamic Fracture with Volumetric Approximate Convex Decompositions*. ACM Transactions on Graphics, 32, 07 2013.
- [51] MÅRTENSSON, P. und S. KENNEDY: *Gamereactor Interview: Red Faction: Guerilla*. <https://www.gamereactor.de/video/5338/INTERVIEW+Red+Faction+Guerilla/>, 2009. Abgerufen: 20. Oktober 2021.
- [52] MUGUERCIA, L., C. BOSCH und G. PATOW: *Fracture modeling in computer graphics*. Computers & Graphics, 45:86–100, 2014.
- [53] MULLER, D. und F. PREPARATA: *Finding the intersection of two convex polyhedra*. Theoretical Computer Science, 7(2):217–236, 1978.
- [54] NAJIM, Y. A. H., G. TRIANTAFYLLOUDIS und G. PALAMAS: *DYNAMIC FRACTURING OF 3D MODELS FOR REAL TIME COMPUTER GRAPHICS*. In: *2018 - 3DTV-Conference: The True Vision - Capture, Transmission and Display of 3D Video (3DTV-CON)*, S. 1–4. IEEE, 2018.
- [55] NASROLLAHI, S.: *Real-time Physics Simulation and Machine Learning*. <https://www.unifiq.com/blog/real-time-physics-simulations-and-ml>, May 2021.

- [56] NEPERUD, B., J. LOWTHER und C.-K. SHENE: *Education: Visualizing and Animating the Winged-Edge Data Structure*. Comput. Graph., 31(6):877–886, dec 2007.
- [57] NIEVERGELT, J. und K. HINRICH: *Algorithms and Data Structures With Applications to Graphics and Geometry*. The Global Text Project, 2011.
- [58] NING, J.-F. und I.-K. LI: *A fast approach to simulate fracture of rigid body*. In: *2010 International Conference on Audio, Language and Image Processing*, S. 1301–1305, 2010.
- [59] NORTON, A., G. TURK, B. BACON, J. GERTH und P. SWEENEY: *Animation of Fracture by Physical Modeling*. Vis. Comput., 7(4):210–219, jul 1991.
- [60] O'BRIEN, J. F. und J. K. HODGINS: *Graphical Modeling and Animation of Brittle Fracture*. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, S. 137–146, USA, 1999. ACM Press/Addison-Wesley Publishing Co.
- [61] OHYA, T., M. IRI und K. MUROTA: *IMPROVEMENTS OF THE INCREMENTAL METHOD FOR THE VORONOI DIAGRAM WITH COMPUTATIONAL COMPARISON OF VARIOUS ALGORITHMS*. Journal of The Operations Research Society of Japan, 27:306–337, 1984.
- [62] OKABE, A., B. BOOTS und K. SUGIHARA: *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams, 2nd Edition*. John Wiley & Sons, Inc., USA, 2000.
- [63] PARKER, E. G. und J. F. O'BRIEN: *Real-Time Deformation and Fracture in a Game Environment*. In: *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, S. 165–175, New York, NY, USA, 2009. Association for Computing Machinery.
- [64] PAWEL, B.: *An overview on boundary representation data structures for 3D models representation*. 2013.
- [65] PREGUN, B.: *THREE-DIMENSIONAL MESH CUTTING IN VIRTUAL SCENE*, 2017.
- [66] RGB-LABS-TEAM: *Blender Cell Fracture Tutorial*. <https://www.rgb-labs.com/blender-198-cell-fracture-blender-addon/>, 2019. Abgerufen: 15. November 2021.
- [67] RIOFRIO, D.: *Applications of Voronoi partitions in particle therapy*. 2012.
- [68] SHAMOS, M. I.: *Geometric Complexity*. In: *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing*, STOC '75, S. 224–233, New York, NY, USA, 1975. Association for Computing Machinery.
- [69] SHAMOS, M. I. und D. HOEY: *Closest-point problems*. In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, S. 151–162, 1975.

- [70] SMITH, A.: *Storytelling Industries: Narrative Production in the 21st Century*. Springer International Publishing, 2018.
- [71] STANFORD-UNIVERSITY: *The Stanford 3D Scanning Repository*. <http://graphics.stanford.edu/data/3Dscanrep/>. Abgerufen: 22. Juli 2021.
- [72] SU, J., C. SCHROEDER und R. FEDKIW: *Energy Stability and Fracture for Frame Rate Rigid Body Simulations*. In: *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '09, New York, NY, USA, 2009. Association for Computing Machinery.
- [73] TÄHT, M.: *Real-time Cave Destruction Using 3D Voronoi*. 2018.
- [74] TANG, X., J. LIU, X. WANG und J. XIONG: *Electric vehicle charging station planning based on weighted Voronoi diagram*. In: *Proceedings 2011 International Conference on Transportation, Mechanical, and Electrical Engineering (TMEE)*, S. 1297–1300, 2011.
- [75] TERZOPOULOS, D. und K. FLEISCHER: *Modeling inelastic deformation: viscoelasticity, plasticity, fracture*. In: *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, S. 269–278, 1988.
- [76] THE CGAL PROJECT: *CGAL User and Reference Manual*. CGAL Editorial Board, 5.4 Aufl., 2022.
- [77] TORRES, L. M.: *Fracture Modeling in Computer Graphics, A survey*. Diplomarbeit, Universitat de Girona, 2011.
- [78] UNITY-TECHNOLOGIES-DOCUMENTATION: *Handles*. <https://docs.unity3d.com/ScriptReference/Handles.html>. Abgerufen: 10. März 2022.
- [79] UNITY-TECHNOLOGIES-DOCUMENTATION: *Renderer.bounds*. <https://docs.unity3d.com/ScriptReference/Renderer-bounds.html>. Abgerufen: 28. März 2022.
- [80] UNITY-TECHNOLOGIES-DOCUMENTATION: *Rigidbody.AddExplosionForce*. <https://docs.unity3d.com/ScriptReference/Rigidbody.AddExplosionForce.html>. Abgerufen: 25. März 2022.
- [81] UNREAL-ENGINE-DOCUMENTATION: *Chaos Destruction*. <https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Physics/ChaosPhysics/ChaosDestruction/>, 2021. Abgerufen: 15. November 2021.
- [82] VALVE: *Steam*. <https://store.steampowered.com/tags/en/Destruction/>. Abgerufen: 26. Juli 2021.
- [83] VORONOI, G.: *Nouvelles applications des paramètres continus à la théorie des formes quadratiques. Premier mémoire. Sur quelques propriétés des formes quadratiques po-*

- sitives parfaites..* Journal für die reine und angewandte Mathematik (Crelles Journal), 1908(133):97–102, 1908.
- [84] WATSON, D. F.: *Computing the n-dimensional Delaunay tessellation with application to Voronoi polytopes**. The Computer Journal, 24(2):167–172, 01 1981.
- [85] WORMSER, C.: *Generalized Voronoi Diagrams and Applications*. Doktorarbeit, Université Nice Sophia Antipolis, 2008.
- [86] YANG, X., M. YIP und X. XU: *Visual Effects in Computer Games*. Computer, 42(7):48–56, 2009.
- [87] ZEMEK, M.: *Regular Triangulation in 3D and Its Applications*. The State of the Art and Concept of Ph.D. Thesis, University of West Bohemia, 2010.

Abbildungsverzeichnis

Abbildung 1	Voronoi Diagramm mit acht Punkten in der Ebene und mit dem euklidischen Abstand berechnet [6]	3
Abbildung 2	Ein Voronoi Diagramm mit Ordnung 2. Im Vergleich zum regulären Voronoi Diagramm enthält nicht jede Zelle einem Punkt der Punktmenge. [43]	4
Abbildung 3	Berechnung des Voronoi Diagramms mit dem euklidischen Abstand, der Manhattan-Distanz und der Maximumsnorm. [34]	5
Abbildung 4	Voronoi Muster eines Schildkrötenpanzers [85]	6
Abbildung 5	Zellsystem eines Blattes ähnelt dem Voronoi Diagramm [73]	6
Abbildung 6	Getrocknete rissige Erde. Da dies in der Realität einem Voronoi Diagramm ähnelt, kann es für Videospiele verwendet werden, um realistische Risse zu modellieren [73]	6
Abbildung 7	a) Reguläres Voronoi Diagramm. b) Gewichtetes Voronoi Diagramm mit gleicher Punktmenge. Die Zahlen geben das Gewicht des jeweiligen Punktes an. [27]	7
Abbildung 8	Eine Delaunay Triangulation in der Ebene, zusammen mit den Umkreisen der gebildeten Dreiecke. Per Definition enthält kein Umkreis einen beliebigen Eckpunkt der Triangulation. [38]	8
Abbildung 9	Dualität des Voronoi Diagramms (blaue Linien) und der Delaunay Triangulation (grüne Linien). [85]	9
Abbildung 10	Dualität zwischen den Elementen des Voronoi Diagramms und der Delaunay Triangulation in 3D [39].	10
Abbildung 11	Unterschiedliche Punktmengen: a) zufällig generierte Punkte; b) Punktmenge mit Clustern. [62, S. 496]	11
Abbildung 12	Iterationen des Lloyd Algorithmus führen zu einer gleichmäßigeren Punkteverteilung. [42]	12
Abbildung 13	Eine Kante der Winged-Edge Datenstruktur [10]	15
Abbildung 14	Aufbau der Doubly-Connected Edge List [76]	16
Abbildung 15	Schritte für die Berechnung einer einzelnen Zelle des Voronoi Diagramms mithilfe eines Brute Force Ansatzes [73]	19

Abbildung 16 Aufbau des Fortune Algorithmus. Die rote strichlierte Line ist die Sweep-line, die sich entlang der Ebene bewegt und die Beachline setzt sich aus den blauen Parabeln zusammen. Gefüllte Kreise sind Punkte, die bereits von der Sweep-line erreicht worden sind, während nicht gefüllte Kreise noch hinzugefügt werden müssen. Die schwarzen Linien sind bereits Kanten des Voronoi Diagramms. [35]	20
Abbildung 17 Ablauf eines Kreisereignisses. Eine Parabel verschwindet von der Beachline und bildet eine neue Kante des Voronoi Diagramms. [15, S. 154]	21
Abbildung 18 Endprodukt des Divide & Conquer Algorithmus. Die polygonale Trennlinie bildet die neuen Kanten des gesamten Voronoi Diagramms. [8]	22
Abbildung 19 Grundlegende Typen von Flips. [87]	23
Abbildung 20 Ablauf beim Einfügen eines Punktes beim Bowyer-Watson Algorithmus. [87]	24
Abbildung 21 Turm bleibt nach einer Zerstörung hängen und zerfällt nicht in kleinere Stücke [3].	27
Abbildung 22 Links: Das Stanford-Bunny, eines der am häufigsten verwendeten Testmodelle in der Computergrafik [71]. Rechts: Das darunterliegende Mesh [24].	28
Abbildung 23 Oben: Ergebnis eines Bruchs eines spröden Materials Unten: Ergebnis eines dehnbaren Materials, erkennbar an der Verformung des Objektes [60].	30
Abbildung 24 a) Model eines Bechers b) Das gebrochene Model des Bechers mit Standardeinstellungen des <i>Cell Fracture Add-Ons</i> . c) Model einer Trinkschale d) Das gebrochene Gegenstück, welches mithilfe des <i>Annotation Pencil</i> erstellt wurde. Dabei werden durch das Zeichnen von Punkten und Kreisen die Fragmente erstellt. [66]	31
Abbildung 25 Das Tauschen eines Türmodells in der <i>Source Engine</i> . Die aufeinanderfolgende Beschädigung der Tür im Spiel erzeugt immer diese Abfolge, unabhängig davon, an welcher Stelle genau der Schaden angewandt wurde [18].	32
Abbildung 26 Das Ergebnis von zwei verschiedenen 3D-Objekten die während der Laufzeit gebrochen wurden, jedoch wurden die entstandenen Löcher nicht gefüllt [54].	33
Abbildung 27 Ein Glasfenster welches durch ein Spinnennetzmuster an mehreren Stellen realistisch zerbrochen wurde [50].	34
Abbildung 28 1) Ein Würfel mit den zugehörigen Verbindungen. Jeder der 12 Kanten des Würfels, sowie jeder der vier inneren Diagonalen ist eine Feder zugeordnet. 2) Anderer Aufbau des Würfels mit Flächendiagonalen. Wieder ist jeder der 12 Kanten des Würfels eine Feder zugeordnet, jedoch diesmal nicht den inneren Diagonalen, sondern den sechs Flächendiagonalen. 3) Ein Zylinder wird mit einem Netz aus deformierten Würfeln modelliert. [59]	37
Abbildung 29 Verbundene Voxel des Mass-Spring Models [47]	37

Abbildung 30	Oben: Aufbau eines tetraedrischen Elements. Unten: Tetraedrisches Mesh eines einfachen Objekts. In (a) sind nur die äußereren Flächen gezeichnet, in (b) wird die innere Struktur gezeigt [60].	39
Abbildung 31	CSG-Operationen eines Würfels und einer Kugel und die daraus resultierenden Objekte. [23]	41
Abbildung 32	Schneiden eines Dreiecks mit einer Ebene. [31]	43
Abbildung 33	Entstandenes Loch nach dem Schneiden des Würfels mit einer horizontalen Ebene. Die Unterseite und die beiden hinteren Flächen werden aufgrund von Backface Culling nicht gerendert. Quelle: Eigene Darstellung	43
Abbildung 34	Delaunay Triangulation einer zufälligen Punktmenge. Quelle: Eigene Darstellung	47
Abbildung 35	Voronoi Diagramm einer zufälligen Punktmenge. Quelle: Eigene Darstellung	47
Abbildung 36	Übersicht der Komponente zur Visualisierung der Delaunay Triangulation und des Voronoi Diagramms. Quelle: Eigene Darstellung	48
Abbildung 37	Einstellungen des <i>Cell Fracture Add-Ons</i> in Blender, die zum Erzeugen von unterschiedlichen Fragmenten gesetzt werden können. [11]	49
Abbildung 38	Oben links: Das 3D-Model wird aus einer Höhe fallengelassen. Oben rechts: Beim Aufprall mit dem Boden wird dieses mit dem gebrochenen Gegenstück ausgetauscht. Unten links: Fragmente zersplittern aufgrund der Geschwindigkeit des Aufpralls. Unten rechts: Ergebnis einer Fragmentierung eines Objektes mit ungefähr 50 Fragmenten. Quelle: Eigene Darstellung	51
Abbildung 39	Resultat der Delaunay Triangulation in 3D mit 50 Punkten. Quelle: Eigene Darstellung	54
Abbildung 40	Voronoi Diagramm in 3D mit 50 Punkten. Quelle: Eigene Darstellung	55
Abbildung 41	Unterschied zwischen dem Verwenden von Polygonen und Dreiecken. Hier wird ein Sechseck zweimal geteilt. In der oberen Reihe wird dieses als Polygon behandelt und in der unteren Reihe als triangulierte Fläche. Im letzten Fall entstehen nach nur zwei Schnitten viele schlecht geformte Dreiecke. [50]	57
Abbildung 42	Entfernen von redundanten Punkten beim Füllen der Löcher verringert die Anzahl der Dreiecke und die benötigte Zeit. Quelle: Eigene Darstellung	57
Abbildung 43	Austausch des Objekts mit zwei unterschiedlichen Methoden. Quelle: Eigene Darstellung	60
Abbildung 44	Messungen der einzelnen Teile des voronoibasierten Algorithmus. Quelle: Eigene Darstellung	61

Tabellenverzeichnis

Tabelle 1	63
---------------------	----

Quellcodeverzeichnis

1 Aufbau der Winged-Edge Datenstruktur	14
2 Abfrage der angrenzenden Eckpunkte und Flächen einer Halbkante	17
3 Iteration der an einer Fläche angrenzen Halbkanten.	17

Abkürzungsverzeichnis

Geo-mod Geometry Modification Technology

GPU graphics processing unit

CPU central processing unit

VACD Volumetric Approximate Convex Decomposition

EDEM Extended Distinct Element Method

FEM Finite-Elemente-Methode

DCEL Doubly-Connected Edge List

CSG Constructive Solid Geometry