UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Analyse von Repräsentationen für 3D-Modelle aus Sicht der Softwaretechnik

Diplomarbeit

vorgelegt von
Bozena Zdunczyk

Erstgutachter:	Prof. Dr. Jürgen Ebert (Institut für Softwaretechnik)
Zweitgutachter:	Dipl.-Inform. Kerstin Falkowski (Institut für Softwaretechnik)

Koblenz, im September 2006

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort, Datum Unterschrift

Inhaltsverzeichnis

1 Einleitung

Anlass zur Themenstellung dieser Arbeit ist das Gemeinschaftsprojekt *Enhanced Reality (ER)* verschiedener Arbeitsgruppen des FB4 der Universität Koblenz-Landau. Ziel dieses Projektes ist eine gemeinsame komponenten-basierte Systemarchitektur für ER-Systeme. Der Begriff *Enhanced Reality* wird als Bereicherung der Realität definiert. Als Grundlage hierzu dient die *Augmented Reality*, welche die Realität durch zusätzliche Informationen erweitert. Eine Beschreibung des Projektes ist unter [uni06] zu finden.

In dieser Arbeit wird die Augmented Reality betrachtet. Wichtige Daten stellen die dort verwendeten Objekte, insbesondere die *3D-Modelle*, dar. Diese Daten werden genauer betrachtet. Es werden verschiedene Darstellungsformen der 3D-Modelle vorgestellt und herausgestellt in wie weit sie sich innerhalb von AR-Anwendungen einsetzen lassen. Insbesondere wird das am häufigsten verwendete *Flächenmodell* betrachtet. Dies erfordert eine Untersuchung der verwendeten Daten, ihrer Datenstrukturen und der sich hier ergebenden Aufwände. Um die Untersuchung zu erleichtern und verständlicher gestalten zu können wurde ein Konzept entwickelt, welches die einzelnen Stufen der Untersuchung herausstellt.

In diesem Kapitel werden der Begriff Augmented Reality, das Themengebiet und der Aufbau dieser Arbeit vorgestellt.

1.1 Motivation

Augmented Reality (kurz: AR) steht im Allgemeinen für eine Erweiterung der Realität und stellt eine neue Form der Mensch-Technik-Interaktion dar. Im Wesentlichen versteht man laut [Mül06] unter AR die Einbettung zusätzlicher virtueller Informationen in die reale Welt. Dabei handelt es sich vor allem um Objekte, aber auch andere Informationen wie z.B. Text. Die Informationen werden in das reale Sichtfeld eines Betrachters oder einer Kamera perspektivisch korrekt eingeblendet und mit den realen Bildern in Deckung gebracht und überlagert. Die Einblendung geschieht kontextabhängig, d.h. passend und abgeleitet vom betrachteten Objekt und der Umgebung.

Ein typischer Benutzer ist mit einem tragbaren Computer und einer Datenbrille ausgestattet. Die Einblendung in diese Datenbrille kann auf zwei unterschiedliche Weisen erfolgen: Video-See-Through oder Optical-See-Through. Beim *Video-See-Through* wird die Realität durch ein Kamerabild repräsentiert und dieses überlagert, beim *Optical-See-Through* hat die Brille ein durchsichtiges Display, sodass der Betrachter die Realität direkt wahrnimmt und die virtuellen Informationen zusätzlich eingeblendet sieht.
Die Interaktion mit virtuellen und realen Gegenständen und Informationen erfolgt in

Echtzeit. Dazu gehören unter anderem Gesten, Text- und Spracheingabe mit Hilfe von Maus, Tastatur oder auch Datenhandschuh.

Anwendung findet AR in Bereichen wie Medizin, Design, Montage, Wartung, Industrie, Raumfahrt oder Militär. Beispiele sind Montageanleitungen, Spiele und Planung von komplizierten medizinischen Eingriffen.

Die eingeblendeten Objekte, meist 3D-Modelle, sind ein wichtiger Bestandteil von AR-Anwendungen und sollen sich möglichst nahtlos und zur richtigen Zeit in das Bild der realen Welt einfügen. Um dies effizient und in Echtzeit zu erreichen ist die Wahl der Darstellungsform und der verwendeten Datenstruktur von Bedeutung.

Das Themengebiet dieser Arbeit wird auf die 3D-Modelle, insbesondere das Flächenmodell, eingegrenzt und beschäftigt sich mit folgenden Zusammenhängen:
Es wird untersucht welche Arten von Modellen existieren, welche Daten sie verwalten und wie diese Daten effizient gespeichert werden können. In diesem Zusammenhang sind verschiedene Datenstrukturen, ihre Operationen und die benötigten Aufwände von Bedeutung. So werden die verschiedenen Modelldarstellungen vorgestellt, mögliche Operationen herausgearbeitet und existierende Lösungsansätze zur Speicherung vorgestellt. An dieser Stelle werden die Operationen mit den Speicherungsarten zusammengebracht und auf ihren Aufwand hin untersucht und anschließend bewertet. Ziel dieser Arbeit soll es sein die Eignung der unterschiedlichen Modelldarstellungen für die Augmented Reality herauszustellen, wobei der Schwerpunkt auf den bereits erwähnten Flächenmodellen liegt.

1.2 Aufbau der Arbeit

In Kapitel 2 wird das Konzept dieser Arbeit vorgestellt. Dieses bildet die Grundlage der Arbeit und dient zur Untersuchung und Erläuterung des Themengebietes. Ein Anwendungsbeispiel verdeutlicht die vorgenommene Unterteilung. Kapitel 3 beschäftigt sich mit Listen. Sie bilden die Grundlage der später vorgestellten Strukturen zur Speicherung der Daten eines Flächenmodells. Kapitel 4 stellt vier verschiedene Repräsentationensformen von 3D-Modellen vor, und dient gleichzeitig zur Einordnung des Flächenmodells. Kapitel 5 stellt den Kern der Arbeit dar und behandelt das Flächenmodell. Hier werden mögliche Operationen, Speicherungsarten und ihr Aufwand vorgestellt. In Kapitel 6 erfolgt eine Zusammenfassung und ein kurzer Ausblick.

2 Konzept dieser Arbeit

In diesem Kapitel soll das Konzept der vorgenommenen Unterteilung vorgestellt werden. Da sich die gesamte Arbeit auf 3D-Modelle bezieht soll dieser Begriff zunächst definiert werden.

Als *Modell* werden nach [Bro03] grafische Abbilder von realen Objekten bezeichnet. Ein Abbild verweist auf ein reales Bild bzw. einen realen Gegenstand in der Außenwelt. Das Abbild muss die Realität erkennen lassen. Im Bereich des Computerdesigns werden oft auch Modelle erstellt, die keine realen Abbilder haben. Sie sind vom Designer erdacht. Man unterscheidet die weniger speicher- und rechenintensiven *zweidimensionalen Darstellungen (2D-Modelle)* und die aufwendigeren *räumlichen Darstellungen (3D-Modelle).*

Um die verschiedenen Bereiche des Modellaufbaus und der Modellbearbeitung zu untersuchen wurde in Zusammenarbeit mit Kerstin Falkowski[1] eine Unterteilung in fünf Stufen vorgenommen. Die benötigten Bestandteile und Informationen werden vom Allgemeinen zum Speziellen aufgeführt. Die Stufen sollen helfen, das Themengebiet besser zu untersuchen und zu erläutern.

Nachfolgend werden die Stufen zunächst aufgeführt und anschließend einzeln vorgestellt. Die Begriffe der Stufen werden während der gesamten Arbeit verwendet. Um einen Gesamtüberblick zu geben, wird die Unterteilung in die einzelnen Stufen zusätzlich an einem Beispiel vorgestellt.

Die fünf Stufen sind:

1. Modell
2. Repräsentationsform
3. Abstrakte Datenstruktur
4. Speicherungsart
5. Implementation

2.1 Die 5 Stufen

2.1.1 Modell

Der Begriff *Modell* stellt die oberste und allgemeinste Stufe der Unterteilung dar. Er wurde gewählt, da sich diese Arbeit in ihrem Kern mit 3D-Modellen beschäftigt. Das

[1] falke@uni-koblenz.de

3D-Modell ist eine Spezialisierung des Begriffs Modell. 3D-Modelle beschreiben räumliche Darstellungen von Objekten. Sie werden in vielen Bereichen verwendet, unter anderem in der Computergraphik in Augmented Reality-Anwendungen. Weiterhin gehört das 1D-, 2D- und 2.5D-Modell zur Gruppe der Modelle. Da ausschließlich 3D-Modelle betrachtet werden, werden sie während dieser Arbeit oft nur Modell genannt.

Typische Beispiele für Modelle in AR-Anwendungen sind alte Gebäude oder Alltagsgegenstände wie Stühle und Tische. So ist es möglich nicht mehr existierende Gebäude einzublenden oder zerstörte Bauwerke in ihrer ursprünglichen Form darzustellen. Dadurch ist es dem Benutzer möglich sich Gebäude aus früherer Zeit besser vorzustellen.
Innerhalb von Räumen ist es möglich zusätzliche Möbelstücke anzuzeigen und diese im Raum zu bewegen. Dadurch kann ein Benutzer einen Raum verändern oder sich diesen mit neuen Gegenständen vorstellen.

2.1.2 Repräsentationsform

Die *Repräsentationsform* definiert die Beschaffenheit eines Modells. Jede Repräsentationsform zeichnet sich durch eine spezifische Darstellung aus. Sie ist kennzeichnend für das Aussehen, die Bestandteile und die Anwendung des Modells.

Eine bekannte Repräsentation ist das *Volumenmodell*. Die Bezeichnung Volumenmodell lässt sich auf die Darstellung zurückführen. Das Modell wird durch Volumen, also ganze Körper beschrieben. So kann ein Modell mit Hilfe von Kugeln, Quadern und weiteren Körpern beschrieben werden. Ein Legostein beispielsweise kann aus einem Quader und vier Zylindern erstellt werden.

2.1.3 Abstrakte Datenstruktur

Die Stufe *abstrakte Datenstruktur* beschreibt die verwendeten Daten, auf welche Art und Weise diese abgelegt werden können und welche Operationen auf diesen Daten möglich sind. Auf dieser Ebene erfolgt die Spezifikation der möglichen Operationen.

Unter *Daten* versteht man nach [Bro03] alles was zum Zweck der Verarbeitung aufgrund von bekannten Vereinbarungen Informationen darstellt. Diese können in maschinenlesbarer Form oder in einer Form, die nur vom Menschen lesbar ist oder beide Möglichkeiten bietet, vorliegen. Daten für den Computer müssen sich für diesen in einer erkennbaren Weise codieren, speichern und verarbeiten lassen. Damit sind alle Informationen gemeint, die abstrahiert und computergerecht aufbereitet sind.

Eine *Datenstruktur* beschreibt nach [Bro03] ein auf Daten anwendbares Ordnungsschema, d.h. die Art wie Daten verwaltet und miteinander verknüpft werden, um auf diese zuzugreifen und diese zu manipulieren. Mit ihrer Hilfe lassen sich Daten interpretieren und spezifische Operationen auf ihnen ausführen.
Der spezielle Aufbau einer Datenstruktur soll nach [Hof04] ermöglichen Funktionen effizient zu implementieren. Dabei werden Zwecke wie schneller Zugriff, Eindeutigkeit, Redundanzfreiheit, optimale Verarbeitung und Widerspruchsfreiheit verfolgt.

Der Begriff *abstrakte Datenstruktur* stellt heraus, dass die Datenstruktur anhand ihrer Operationen betrachtet wird. Diese Operationen gelten auf der gesamten Datenstruktur und sind von der späteren Implementierung unabhängig.

Ein Beispiel für Daten sind Kanten eines Drahtmodells, Beispiele für Datenstrukturen sind Arrays, Listen, Graphen und Bäume. Bekannte Operationen sind z.B. Einfügen, Löschen, Suchen.

2.1.4 Speicherungsart

Die Stufe *Speicherungsart* beschreibt die Organisation des Speichers. D.h. es wird beschrieben, wie der Speicher aufgebaut ist und wie die Elemente in diesem zusammenhängen. Sie konkretisiert die vorangehende Stufe abstrakte Datenstruktur.

Eine *sequentielle Speicherung* beschreibt beispielsweise das Ablegen von Elementen nacheinander in einem zusammenhängenden Bereich und beschreibt dadaurch den Speicheraufbau. Elemente können z.B. 3D-Koordinaten sein.
Ein weiteres Beispiel stellt die einfache Verkettung von Elementen dar. Hierbei ist der Speicherbereich nicht zusammenhängend. Der Zusammenhang der Elemente wird durch das Verweisen auf benachbarte Elemente beschrieben.

2.1.5 Implementation

Die Stufe der *Implementation* beinhaltet die Wahl der Programmiersprache, die Umsetzung der abstrakten Datenstruktur, der Speicherungsart, sowie den darauf möglichen Operationen. Sie richtet sich nach der Spezifikation, die in der Stufe der abstrakten Datenstruktur vorgenommen wurde und den vorgegebenen Möglichkeiten aus der Stufe Speicherungsart.

Eine Beispielumsetzung stellt die Operation Einfügen innerhalb einer doppelt verketteten Liste in C++ dar. Die abstrakte Datenstruktur ist eine Liste, die Speicherungsart ist doppelt verkettet und die Programmiersprache ist C++.

2.2 Anwendungsbeispiel

In diesem Abschnitt sollen die vorgestellten Stufen an einem zusammenhängenden Beispiel verdeutlicht werden. Vor allem soll sichtbar werden, wie die Stufen zueinander in Beziehung stehen, und dass sie aufeinander aufbauen.

Als Beispiel wurde der berühmte *Utah Teapot* gewählt. Seine Geschichte ist unter anderen in [Bak] zu finden.

Der Utah Teapot ist das bekannteste Objekt der Computergraphik und wurde 1975 von Martin Newell im Rahmen seiner computergrafischen Forschungsarbeit an der Universität von Utah entwickelt. Er entstand aus dem Bedürfnis heraus ein einfaches mathe-

matisches Modell mit für die damaligen Zwecke notwendigen Eigenschaften zu erstellen. Diese Eigenschaften waren eine runde Form, konkave Elemente (nach außen gewölbte Formen) und geringer Speicherbedarf. Die Idee für den Teapot stammte von seiner Frau. Diese hatte vorgeschlagen ein Teegeschirr zu modellieren. Das Modell wurde von anderen Forschungsgruppen wegen seiner Eigenschaften übernommen und wurde so zum Referenzmodell für die Computergrafik.

Abbildung 2.1: Utah Teapot, echte Teekanne; Quelle: [Bak]

Abbildung 2.1 zeigt eine reale Teekanne, die als Grundlage für das Modell diente. In Abbildung 2.2 ist das gesamte Teegeschirr als Drahtmodell und als gerendertes 3D-Modell zu sehen. Der Begriff *Rendern* beschreibt ein Verfahren zur Berechnung eines computergenerierten Bildes aus einer vorliegenden Szenenbeschreibung und stellt eine realitätsnahe Gestaltung dreidimensionaler Objekte durch Farb- und Lichteffekte dar. Als Ausgang für das Rendering kann z.B. ein Drahtmodell dienen.

Die Veranschaulichung der Aufteilung wird am Drahtmodell der Teekanne aus Abbildung 2.2 vorgenommen. Ein Drahtmodell ist ein vereinfachtes dreidimensionales Modell, welches durch Kanten zwischen Punkten beschrieben wird. Flächen sind hier nicht sichtbar.

Die Unterteilung beginnt bei der allgemeinsten Stufe Modell. Der Utah-Teapot ist ein 3D-Modell. Er stellt die Teekanne räumlich dar.

Das 3D-Modell wird repräsentiert als Drahtmodell. Diese Repräsentationsform wird im nächsten Kapitel genauer vorgestellt. Die Bezeichnung Drahtmodell lässt sich auf die Darstellung zurückführen. Das Modell wird durch Kanten beschrieben und wirkt wie ein Gebilde aus Draht. Ein Drahtmodell wird oft zur einfachen Darstellung von dreidimen-

Abbildung 2.2: Utah Teapot, ganzes Teegeschirr; Quelle: [Bak]

sionalen Gegenständen genutzt. Es liefert einen ersten Eindruck des späteren 3D-Modells.

Eine abstrakte Datenstruktur für Drahtmodelle ist eine Kantenliste. Eine Kantenliste speichert Informationen über Eckpunkte und zusätzlich über die Kanten. Sie ist für die Verwaltung von Eckpunkten und Kanten besonders gut geeignet, da sie genau diese Daten verwaltet. Eine Kantenliste besteht insgesamt aus zwei Listen, einer Punktliste und einer Kantenliste. Die Kantenliste verweist auf die Punktliste und eine Face (Fläche) wird als Liste von Zeigern in die Kantenliste definiert. Die abstrakte Datenstruktur wird durch die auf ihr möglichen Operationen beschrieben. Beispiele für Operationen auf einer Kantenliste sind kanteEinfügen und kanteEntfernen. Wobei jede Kante zwei Eckpunkte, den Start- und Endpunkt, beinhaltet.

Der Aufbau des Speichers innerhalb einer Liste, hier einer Kantenliste, kann entweder sequentiell, einfach verkettet oder doppelt verkettet erfolgen. Die Speicherungsart spezialisiert die Art der Liste. In diesem Beispiel wird für die Listen die einfache Verkettung gewählt. Bei dieser Art des Listenaufbaus sind die Elemente und die verwendeten Listen über Zeiger miteinander verkettet. Der Aufbau ist dynamisch und die Liste enthält jeweils nur so viele Elemente wie tatsächlich auch existieren. Gibt es 50 Kanten innerhalb des Modells, so hat die Liste 50 Elemente. Einfach verkettete Listen sind flexibel und ermöglichen jederzeit Veränderungen der Daten.

Die Implementation der vorgegebenen Datenstruktur Liste und der vogegebenen Speicherungsart einfach verkettet, sowie den Operationen kanteEinfügen und kanteEntfernen erfolgt in der Programmiersprache C++.

Abbildung 2.3 zeigt noch einmal alle fünf Stufen des Utah Teapots bildlich dar.

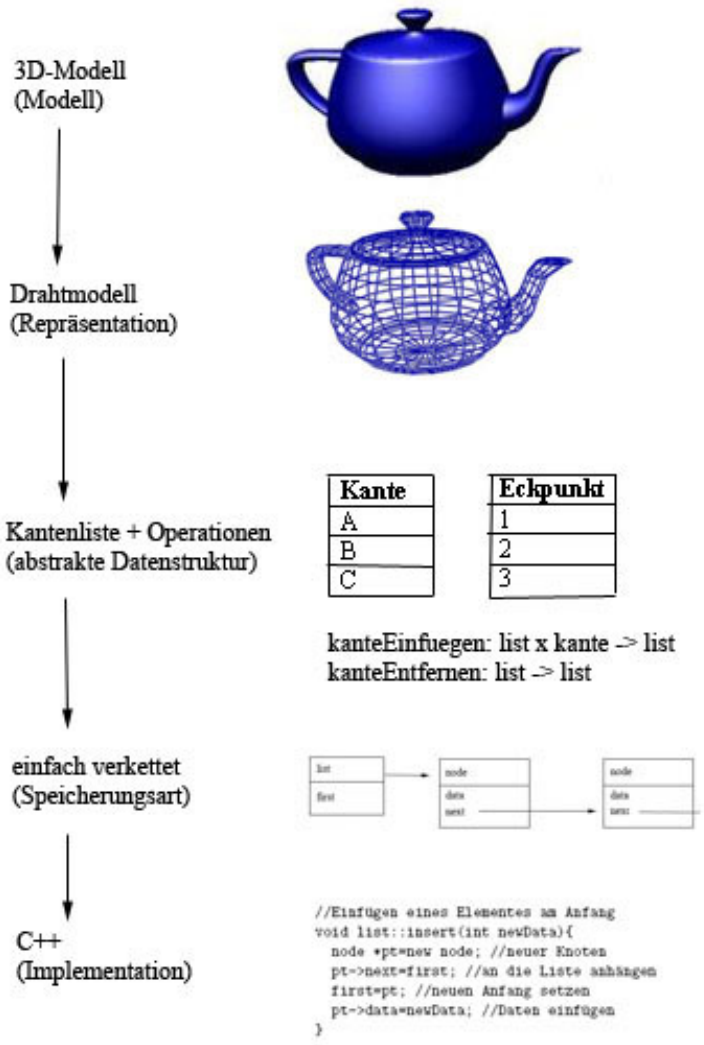


Abbildung 2.3: Die 5 Stufen am Beispiel des Utah Teapots

3 Listen und ihr Aufwand

Dieses Kapitel beschäftigt sich mit Listen und ihrem Aufwand. *Listen* bilden die Grundlage für die Operationen der meisten Repräsentationsformen. Sie werden hier behandelt, um später die Erkenntnisse dieses Kapitels innerhalb der Abstrakten Datenstrukturen der Repräsentationen von 3D-Modellen nutzen zu können. Zusätzlich dient dieses Kapitel nochmals zur Veranschaulichung der Unterteilung in die verschiedenen Stufen. Durch Anwendung der *Datenstruktur Liste* sollen die verwendeten Begriffe innerhalb der Unterteilung geklärt werden. Die Liste stellt die Stufe der Abstrakten Datenstruktur dar. Auf dieser Stufe geht es um die verwendeten Daten und die möglichen Operationen. Die Liste wird allgemein behandelt und ihr Aufbau nicht näher beschrieben. Die Stufen Modell und Repräsentation darüber existieren innerhalb dieser Darstellung nicht. Die Stufe Speicherungsart konkretisiert die allgemeine Liste und die Stufe Implementation setzt diese in einer Programmiersprache um.

Zunächst werden die Begriffe Liste, Aufwand, Signatur und Spezifikation definiert. Danach wird auf die möglichen Operationen auf Listen eingegangen. Diese werden zunächst vorgestellt und anschließend implementiert. Zuvor werden die Speicherungsarten innerhalb von Listen vorgestellt. Die Implementation der Operationen wird für jede der Speicherungsarten durchgeführt. Jede der Operationen wird kurz beschrieben und ihr Aufwand aufgeführt. Zum Schluß werden die Erkenntnisse dieses Kapitels zusammengefasst und kurz bewertet.

3.1 Abstrakte Datenstruktur

Die *abstrakte Datenstruktur Liste* ist nach [Bro03] eine lineare Datenstruktur, die so organisiert ist, dass sich den einzelnen Elementen Nummern zuordnen lassen. Jedes Element hat damit genau einen Vorgänger und einen Nachfolger, außer dem ersten Element und dem letzten Element. Das erste Element hat keinen Vorgänger und das letzte Element hat keinen Nachfolger. Die Zahl der Listenelemente ist variabel und im Prinzip beliebig groß. Listen lassen sich mit Hilfe der grundlegenden Operationen der Listenverarbeitung manipulieren. Dazu gehören Operationen wie das Einfügen, Entfernen und Suchen von Elementen und das Durchlaufen der Liste.

Eine Beipielliste von Integer-Werten wäre folgende Liste:

$$< 3, 9, 5, 1, 4, 10, 6, 12 >$$

3.1.1 Aufwand von Listen

Nach [Bal99] setzt sich der *Aufwand* aus dem *Speicheraufwand* und dem *Zeitaufwand* der Operationen zusammen. Der Speicheraufwand beschreibt den Speicherplatzbedarf der Daten. Der Zeitaufwand gibt an wie lange eine Operation dauert.

Der Speicherplatzbedarf von Listen wird hier in der *O-Notation* angegeben, und beträgt für eine abstrakte Datenstruktur O(n) für eine Liste mit n Elementen. Zusätzlich kann abhängig von der Speicherungsart Speicherplatz für die jeweils verwendete Liste hinzukommen. Der Zeitaufwand ist von der jeweiligen Operation und der Speicherungsart abhängig. Nachfolgend werden die Operationen Einfügen, Entfernen, Suchen und Testen vorgestellt. Bei der späteren Aufwandsbetrachtung wird der schlechteste Fall betrachtet.

3.1.2 Signatur und Spezifikation

Die *Signatur* einer Operation besteht nach [Bal99] aus dem Namen der Operation, den Typen aller Parameter und dem Ergebnistyp der Operation. Die Menge aller Signaturen, die von den Operationen einer Klasse definiert werden, nennt man die *Signatur der Klasse*. Die Signatur spezifiziert die Syntax der Operation. Die in dieser Arbeit verwendete Signatur ist eine Tiefensignatur. In ihr wird die Klasse explizit angegeben. Die Tiefensignatur wird in [Ebe] beschrieben.

Die *Spezifikation* beschreibt was die Operation tut, aber nicht wie sie das tut. Sie besteht in dieser Arbeit aus einer natürlichsprachlichen Beschreibung und ergänzend aus einer formalen Beschreibung (Signatur). Sie dient als Vorlage für die Implementierung.
Die Darstellung erfolgt zunächst durch eine Signatur, welche natürlichsprachlich ergänzt wird. Die Beschreibung erfolgt in kurzen Sätzen und ist in eine Vorbedingung, eine Nachbedingung und eine Fehlerbedingung gegliedert. In der Vorbedingung wird beschrieben was erfüllt sein muss, damit die Funktion funktioniert. Die Nachbedingung definiert was die Funktion bewirkt und formuliert welche Bedingungen auf der Datenstruktur gelten, nachdem die Funktion aufgerufen wurde. Außerdem wird hier gesagt wie die Ergebnisse der Funktion von der Eingabe vom Vorzustand abhängen. In den Fehlerbedingungen wird aufgeführt was passiert wenn die Vorbedingung nicht erfüllt ist.

Das Format der Beschreibung sieht wie folgt aus:

Signatur
Vorbedingung
Nachbedingung
Fehlerbedingung

3.1.3 Operationen auf Listen

In diesem Abschnitt werden die häufigsten Operationen auf Listen vorgestellt. Auf jeder Datenstruktur lassen sich spezifische Operationen durchführen. Die Operationen gelten auf der Datenstruktur an sich unabhängig von ihrem genauen Aufbau. Daher erfolgt

die Beschreibung innerhalb dieser Stufe und nicht erst auf der Stufe der Speicherungsart.

Die im Folgenden verwendeten Listen sind homogen und beinhalten ausschließlich Daten vom Typ Integer. Die Position einer Liste beginnt bei Null. Für die Elemente innerhalb der Liste wird ein Datentyp `data` und für die Position ein Datentyp `pos` definiert. `data` beschreibt die Listenelemente und `pos` definiert die Position innerhalb einer Liste. Sie dienen der Eindeutigkeit der Operationen.

Einfügen

insertAtBegin: list x data → list

Vorbedingung: —
Nachbedingung: Fügt das Element data am Anfang der Liste ein. Durch das neue Element wird die Liste verändert. Das neue Element stellt den neuen Anfang der Liste dar. Die bisher enthaltenen Daten bleiben unverändert. Die Position der Elemente ändert sich, da sie hinter dem neuen Element angeordnet werden.
Fehlerbedingung: —

insertAtEnd: list x data → list

Vorbedingung: Es muss mindestens ein Element in der Liste vorhanden sein.
Nachbedingung: Fügt das Element data am Ende der Liste ein. Durch das neue Element wird die Liste verändert. Die bisher vorhandenen Elemente und ihre Position bleiben unverändert erhalten.
Fehlerbedingung: Ist die Liste leer, ist das neue Element das erste Element der Liste.

insertAtPos: list x data x pos → list

Vorbedingung: Die Position muss größer oder gleich Null und kleiner als die Listenlänge sein.
Nachbedingung: Fügt das Element data an der Position pos in die Liste ein. Die Liste wird verändert. Das Element welches bisher sich an dieser Position befand und alle Elemente dahinter werden um eine Position nach hinten verschoben. Die Daten an sich werden nicht verändert, lediglich ihre Position innerhalb der Liste.
Fehlerbedingung: Ist die Position kleiner Null kann nicht in die Liste eingefügt werden. Ist die Position größer als die Listenlänge wird das Element am Ende der Liste angefügt.

Entfernen

deleteAtBegin: list → list

Vorbedingung: Die Liste muss mindestens ein Element beinhalten.
Nachbedingung: Löscht das erste Element in der Liste und ändert dadurch die Liste. Das zweite Element wird zum ersten Element und bildet den Anfang der Liste. Die Ele-

mente bleiben unverändert. Ihre Position ändert sich. Sie rücken um eine Position nach vorne.
Fehlerbedingung: Ist die Liste leer wird die Funktion nicht ausgeführt und das Element kann nicht entfernt werden.

deleteAtEnd: list → list

Vorbedingung: Es muss mindestens ein Element in der Liste existieren.
Nachbedingung: Löscht das letzte Element in der Liste. Die Liste wird verändert. Sie beinhaltet das bisher letzte Element nicht mehr. Die restlichen Elemente bleiben unverändert, wobei das vorletzte Element nun das Ende der Liste bildet.
Fehlerbedingung: Bei leerer Liste ist das Entfernen nicht möglich und die Funktion wird nicht ausgeführt.

deleteAtPos: list x pos → list

Vorbedingung: Die Liste muss mindestens ein Element beinhalten. Die Position muss größer oder gleich Null sein und kleiner als die Länge der Liste.
Nachbedingung: Löscht ein Element an der Position pos. Die veränderte Liste beinhaltet das gelöschte Element an der angegebenen Position nicht mehr. Die Elemente hinter dem gelöschten Element, falls vorhanden, werden um eine Position nach vorne verschoben um die entstandene Lücke zu schließen. Die Elemente an sich ändern sich nicht.
Fehlerbedingung: Ist kein Element in der Liste enthalten, dann kann kein Element gelöscht werden. Ist die Position negativ oder größer als die Länge der Liste, kann nicht entfernt werden. Auf eine falsche Position erfolgt keine Reaktion.

deleteData: list x data → list

Vorbedingung: Das Element data muss in der Liste vorhanden sein.
Nachbedingung: Löscht das Element data aus der Liste. Veränderte Liste enthält das Element data nicht mehr. Sind Elemente hinter dem gelöschten Element data vorhanden, so werden sie um eine Position nach vorne verschoben. Sind mehrere gleiche Elemente in der Liste vorhanden so wird das erste Vorkommen des Elementes gelöscht.
Fehlerbedingung: Ist das Element nicht in der Liste enthalten, kann es nicht aus der Liste gelöscht werden und die Funktion wird nicht ausgeführt.

Suchen

search: list x data → pos

Vorbedingung: —
Nachbedingung: Durchsucht die Liste nach dem Element data. Liefert die Position des gesuchten Elementes innerhalb der Liste. Die Liste und die Elemente werden durch die Funktion nicht verändert. Sind mehrere gleiche Elemente in der Liste vorhanden wird

das erste Vorkommen des Elementes angegeben.
Fehlerbedingung: Ist das Element nicht in der Liste vorhanden oder ist die Liste leer wird -1 als Positionsangabe zurückgeliefert.

Testen

isEmpty: list → bool

Vorbedingung: —
Nachbedingung: Überprüft die Liste, ob Elemente in dieser vorhanden sind. Liefert true, falls die Liste leer ist und false, falls die Liste nicht leer ist.
Fehlerbedingung: —

isIn: list x data → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob ein Element data innerhalb der Liste vorhanden ist. Liefert true, falls das Element in der Liste vorhanden ist und false, falls das Element nicht in der Liste vorhanden ist.
Fehlerbedingung: —

Tabelle 3.1 führt noch einmal alle vorgestellten Operationen zusammenfassend auf.

Operation
insertAtBegin: list x data ⟶ list insertAtEnd: list x data ⟶ list insertAtPos: list x data x pos ⟶ list
deleteAtBegin: list ⟶ list deleteAtEnd: list⟶ list deleteAtPos: list x pos ⟶ list deleteData: list x data ⟶ list
search: list x data ⟶ pos
isEmpty: list ⟶ bool isIn: list x data ⟶ bool

Tabelle 3.1: Gesamtübersicht der Listenoperationen

3.2 Speicherungsarten von Listen

In den Abschnitten zuvor wurde die abstrakte Datenstruktur Liste definiert. Dabei wurde die Liste mit ihren Operationen allgemein behandelt. In diesem Abschnitt wird auf die Speicherungsarten von Listen eingegangen. Dabei wird die Stufe der abstrakten Datenstruktur konkretisiert indem beschrieben wird, wie eine Liste aufgebaut sein kann.

Es werden die drei Speicherungsarten *sequentiell, einfach verkettet* und *doppelt verkettet* betrachtet. Anhand einer Beispielimplementation wird der Aufwand bestimmt. Die verschiedenen Arten der Implementation sollen die verschiedenen Aufwände verdeutlichen. Die Implementation bezieht sich auf die einfach und die doppelt verketteten Listen. Vollständigkeitshalber wird am Ende auch der Aufwand sequentieller Listen aufgeführt und diese kurz bewertet.

Sequentielle Listen sind Listen, die ihre Elemente in einem zusammenhängenden Speicherbereich ablegen. Auf die einzelnen Elemente wird über einen Index zugegriffen. Der Speicherplatz hat eine feste Größe, die am Anfang festgelegt wird. Reicht der Platz nicht aus müssen sie aufwendig erweitert werden.

Abbildung 3.1 zeigt den sequentiellen Aufbau einer Liste. Diese Liste hat eine Länge von 8 Elementen und ist komplett mit Integer-Werten gefüllt. Der Index beginnt bei Null.

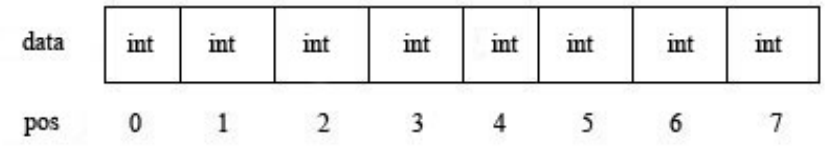


Abbildung 3.1: Sequentielle Liste

Die *Verkettung* ist eine Speicherungsart für Elemente, die durch Zeiger auf den Nachfolger (einfach verkettet) oder den Nachfolger und den Vorgänger (doppelt verkettet) verweist. Jedes Element enthält neben den Daten bzw. einem Zeiger auf die Daten mindestens einen weiteren Zeiger. Verkettete Listen besitzen eine dynamische Länge, d.h. die Liste ist nur so lang wie die Anzahl ihrer gespeicherten Elemente. Die Länge der Liste kann jederzeit an die Anzahl der Elemente angepasst werden.

Die Abbildung 3.2 zeigt eine einfach verkettete Liste und die Abbildung 3.3 zeigt eine doppelt verkettete Liste.

3.3 Implementation

In diesem Abschnitt werden die vorgestellten Speicherungsarten einfach verkettet und doppelt verkettet in der *Programmiersprache C++* implementiert. Die sequentielle Speicherung wird nur kurz vorgestellt. Die Stufe der Implementation ist die letzte Stufe. Sie richtet sich nach der Spezifikation aus der Stufe abstrakte Datenstruktur und beinhaltet

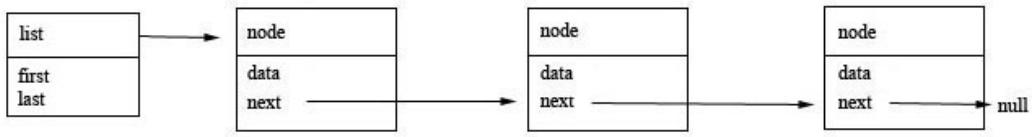


Abbildung 3.2: Aufbau einer einfach verketteten Liste

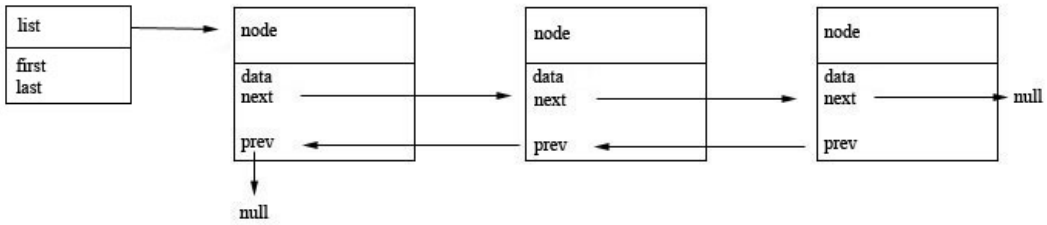


Abbildung 3.3: Aufbau einer doppelt verketteten Liste

die Seicherungsarten der vorangehenden Stufe.

Die in Abschnitt 3.1.3 aufgeführten Operationen und ihre Spezifikation werden in diesem Abschnitt realisiert. Die dort definierten Datentypen `data` und `pos` werden innerhalb der Listen verwendet und stellen Integer-Werte dar. Die Liste selbst wird durch die Klasse `list` bzw. `dlist` beschrieben. Die Operation Suchen wird durch den Aufbau verketteter Listen anders als in ihrer Spezifikation beschrieben implementiert. Dort wurde bei dieser Operation definiert, dass die Position `pos` als Integer-Wert zurückgeliefert wird. Bei der Umsetzung verketteter Listen ist es jedoch üblich den Knoten auszugeben. Dieser liefert sowohl die Position innerhalb der Liste als auch die enthaltenen Daten.

3.3.1 Sequentielle Liste

Sequentiell gespeicherte Elemente werden in C++ in einem zusammenhängenden Bereich als Array abgelegt. Die Größe dieses Bereiches wird am Anfang festgelegt. Die Operationen Einfügen, Entfernen und Suchen werden auf einem Array realisiert. Ein Array ist eine Datenstruktur, die es erlaubt Elemente vom gleichen Typ zu einer Einheit zusammenzufassen. Auf jedes Element wird über einen Index zugegriffen. Dieser wird im Allgemeinen erst zur Laufzeit berechnet. Der Aufwand dieser Operationen wird am Ende dieses Kapitels aufgeführt.

3.3.2 Einfach verkettete Liste

Um die abstrakte Datenstruktur Liste mit der Speicherungsart *einfach verkettet* zu implementieren sind zwei Klassen nötig. Die Klasse `node` beinhaltet die Knoten. Diese

bestehen aus den darin enthaltenen Daten `data` und einem Zeiger `next` auf den nächsten Knoten. Die Klasse `list` definiert die Liste und die darin enthaltenen Methoden. Der Knoten `first` beschreibt den Listenkopf und verweist auf den Anfang der Liste. Es werden die vorgestellten Operationen Einfügen, Entfernen, Suchen und Testen implementiert.

In der h-Datei `slist.h` werden die beiden Klassen `node` und `list`, sowie die benötigten Operationen deklariert. Zusätzlich wird die Hilfsoperation `searchPrev` implementiert.

```
 1 //slist.h
 2 #ifndef SLIST_H
 3 #define SLIST_H
 4
 5 class node{
 6 public:
 7   int data;
 8 private:
 9   node *next;
10 };
11
12 class list{
13 public:
14   list(void);
15   ~list( );
16   void insertAtBegin(int newData);
17   void insertAtEnd(int newData);
18   void insertAtPos(int newData, int pos);
19   void deleteAtBegin( );
20   void deleteAtEnd( );
21   void deleteAtPos(int pos);
22   void deleteData(int &oldData);
23   node *search(int sdata);
24   bool isEmpty( );
25   bool isIn(int iData);
26 private:
27   node *searchPrev(int sdata);
28   node *first;
29 };
30
31   #endif //SLIST_H
```

Sourcecode 3.1: Datei slist.h

In der cpp-Datei `slist.cpp` werden die verschiedenen Operationen implementiert. Diese werden einzeln mit ihrem jeweiligen Aufwand vorgestellt.

```
 1 //slist.cpp
 2 #include „slist.h“
 3
 4 list::list(void){ //Konstruktor
 5    first=0;
 6 }
 7
 8 list::∼list(){ //Destruktor
 9   while(first!=0){
10      node *pt=first;
11      first=first→next;
12      delete pt;
13      }
14 }
```

Sourcecode 3.2: Datei slist.cpp

Die Aufgabe eines Konstruktors ist es, ein Objekt zu erzeugen. Er alloziert den benötigten Speicherplatz und versetzt das Objekt in einen gültigen Zustand. Im Konstruktor wird der Knoten `first` mit dem Anfangswert Null belegt. Damit ist die Liste zu Begin leer. Der Aufwand hierfür ist konstant (O(1)).
Der Destruktor zerstört das Objekt. Er wird automatisch aufgerufen, wenn das Objekt nicht mehr benötigt wird. Im Destruktor werden die Knoten der Liste gelöscht. Bevor ein Block zu Ende geht und die Knoten nicht mehr benötigt werden, kann der für sie benötigte Speicherplatz wieder freigegeben werden. Der Aufwand ist O(n).

```
1 //Einfügen eines Elementes am Anfang
2 void list::insert(int newData){
3   node *pt=new node; //neuer Knoten
4   pt→next=first; //an die Liste anhängen
5   first=pt; //neuen Anfang setzen
6   pt→data=newData; //Daten einfügen
7 }
```

Sourcecode 3.3: Einfügen eines Elementes am Anfang

Beim Einfügen am Anfang der Liste wird ein neuer Knoten `pt` mit dem neuen Element `newData` eingefügt. Dieser verweist auf das ursprüngliche erste Element. Ist noch kein Element in der Liste vorhanden, so ist das neue Element das erste Element der Liste. Das Einfügen am Anfang ist in konstanter Zeit möglich, da die Liste vorher nicht durchlaufen

werden muss. Das neue Element kann direkt über den Knoten `first` eingefügt werden. Es müssen nur Zeiger manipuliert werden. Der Aufwand ist daher O(1).

```
 1 //Einfügen eines Elementes am Ende
 2 void list::insertAtEnd(int newData){
 3   if(first!=0){ //Liste nicht leer
 4     node *pt=first; //am Anfang beginnen
 5     while(pt→next!=0) //ans Ende laufen
 6       pt=pt→next;
 7     node *n = new node; //neuer Knoten
 8     pt→next=n; // ans Ende anhängen
 9     n→next=0; //Ende zeigt auf Null
10     n→data=newData; //Daten reinsetzen
11   }
12   else if(first==0){ //Liste leer
13     node *pt=new node; //neuer Knoten
14     pt→next=first; //an die Liste anhängen
15     first=pt; //neuen Anfang setzen
16     pt→data=newData; //Daten einfügen
17   }
18 }
```

Sourcecode 3.4: Einfügen eines Elementes am Ende

Beim Einfügen eines Elementes am Ende der Liste muss zunächst der letzte Knoten ermittelt werden. Dazu wird die Liste bis zum Ende durchlaufen. Innerhalb einer einfach verketteten Liste ist kein direkter Zugriff auf den letzten Knoten möglich, da die Zeiger stets alle in die gleiche Richtung zeigen. Die hier implementierte Liste beginnt beim ersten Knoten. Der Aufwand hierfür ist O(n).

```
 1 //Einfügen eines Elementes an bestimmter Position
 2 void list::insertAtPos(int newData, int pos){
 3   int number=0; //Anzahl der Elemente
 4   int length; //Länge der Liste
 5
 6   for(node *pt=first; pt!=0; pt=pt→next)
 7     number++;//Anzahl der Elemente zählen
 8   length = number;
 9
10   node *pt=first;
11   if(pos<0) //Position außerhalb der Liste
12     return;
13   if(pos>=length) //Position größer Listenlänge
14     pos=length; //am Ende anhängen
```

```
15  if(pos==0){ //an den Anfang der Liste setzen
16    node *pt=new node; //neuer Knoten
17    pt→next=first; //an die Liste anhängen
18    first=pt; //neuen Anfang setzen
19    pt→data=newData; //Daten einfügen
20  }
21  while(pos>1){ //bis zum Vorgänger gehen und davor einfügen
22    pt=pt→next;
23    pos - -;
24  }
25  node *p = new node; //neuer Knoten
26  p→next = pt→next; // neuer Knoten zeigt auf den
27                              // Knoten wo der alte hingezeigt hat
28  pt→next = p; //alter Knoten zeigt jetzt auf den neuen Knoten
29  p→data=newData; //Daten einfügen in den neuen Knoten
30
31 }
```

Sourcecode 3.5: Einfügen eines Elementes an bestimmter Position

Das Einfügen an beliebiger Position `pos` in der Liste erfordert die Kenntnis des Vorgängers. Der Zeiger des Vorgängers muss auf das neue Element gesetzt werden. Da in einer einfach verketteten Liste die Knoten jeweils nur auf den Nachfolger zeigen, muss der Vorgänger erst ermittelt werden. Dies geschieht indem man die Liste bis zu diesem durchläuft. Das erfolgt mit Hilfe der Variable `pos`. Der Aufwand ist O(n). Abbildung 3.4 verdeutlicht das Einfügen eines Elementes an einer beliebigen Position.

```
1 //Löschen am Anfang
2 void list::deleteAtBegin(){
3   if(first!=0){ //Liste ist nicht leer
4     node *pt=first; //am Anfang beginnen
5     first=first→next; //first zeigt auf den übernächsten Knoten,
6                       //damit der davor gelöscht werden kann
7     delete pt; //Knoten löschen
8   }
9 }
```

Sourcecode 3.6: Löschen am Anfang

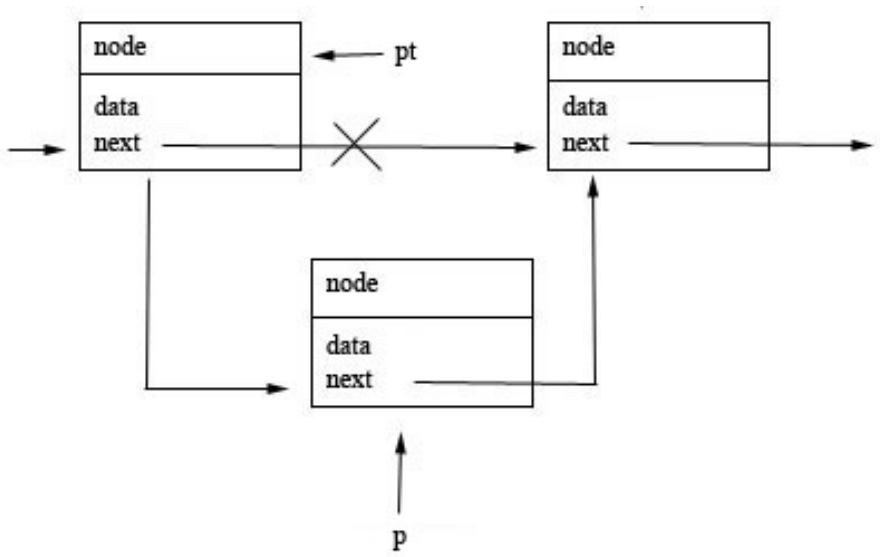


Abbildung 3.4: Einfach verkettete Liste: Einfügen eines Elements an beliebiger Position

Das Entfernen eines Elements am Anfang verursacht einen Zeitaufwand von O(1), da das Entfernen unabhängig von der Länge der Liste einen konstanten Aufwand verursacht. Es wird der Zeiger des Knotens `first` umgesetzt und der erste Knoten (`pt`) entfernt.

```
 1 //Löschen am Ende
 2 void list::deleteAtEnd(){
 3   if(first!=0){ //Liste ist nicht leer
 4     node *pt=first; //am Anfang beginnen
 5     if(pt→next!=0){//falls mehr als 1 Element
 6       while(pt→next→next!=0)//bis zum vorletzen Element gehen
 7         pt=pt→next;
 8       pt→next=0; //Vorletzer hat keinen Nachfolger mehr
 9       delete pt→next; //letzes Element löschen
10     }
11     else {//falls nur 1 Element
12       delete pt; //letztes Element löschen
13     }
14   }
15 }
```

Sourcecode 3.7: Löschen am Ende

Das Entfernen eines Elementes am Ende der Liste weist einen Aufwand von O(n) auf. Um das letzte Element entfernen zu können wird die Liste bis zum Vorgänger durchlaufen. Hier wird der Verweis auf den Nächsten auf Null gesetzt und das letzte Element

gelöscht. Wie beim Einfügen am Ende ist auch hier durch die vorgegebene Richtung der Zeiger kein direkter Zugriff auf das Ende der Liste möglich.

```
1 //Löschen an einer bestimmten Position
2 void list::deleteAtPos(int pos){
3   int anzahl=0; //Anzahl der Elemente
4   int length; //Listenlänge
5
6   for(node *pt=first; pt!=0; pt=pt→next)
7     anzahl++;//Anzahl der Elemente zählen
8   length = anzahl;
9
10  if(first!=0){ //Liste ist nicht leer
11    node *pt=first;
12    if(pos<0)//Position außerhalb der Liste
13      return;
14    if(pos>=length) //Position größer Listenlänge
15      return;
16    if(pos==0){ Position gleich Null
17      first=first→next;
18      delete pt; //ersten Knoten löschen
19    }
20    while ( pos> 0 ) {
21      pt = pt→next; //durchlaufen der Liste
22      pos - -;
23    }
24    pt=pt→next;
25    delete pt; //Knoten löschen
26  }
27 }
```

Sourcecode 3.8: Löschen an einer bestimmten Position

Um an einer gegebenen Position ein Element zu entfernen wird zu der gegebenen Position-1 traversiert. Damit ist der Vorgänger bekannt. Der `next`-Zeiger wird dann auf den neuen Nachfolger gesetzt und das Element an der gegebenen Position kann gelöscht werden. Die Operation erfordert das Durchlaufen der Liste und ist von deren Anzahl n der Elemente abhängig. Dadurch entsteht ein Aufwand von O(n).

```
1 //Löschen eines bestimmten Elementes
2 void list::deleteData(int &oldData){
3   if(first!=0){ //Liste ist nicht leer
4     node *prev = searchPrev(oldData); //Vorgänger suchen
5     node *old = prev→next;
```

```
 6     if(prev==0) //falls kein Vorgänger, also erstes Element
 7       node *pt=first; //am Anfang beginnen
 8       first=first→next; //first zeigt auf den übernächsten Knoten,
 9                         //damit der davor gelöscht werden kann
10       delete pt; //Knoten löschen
11     else {
12       if(old→next==0) //ist Element das entfernt wird das letzte
13         prev→next=0; // wenn ja dann next auf 0 setzen
14       else
15        prev→next=old→next; //sonst next auf Nachfolger von old setzen
16
17       delete old; //Element entfernen
18     }
19   }
20 }
```

Sourcecode 3.9: Löschen eines bestimmten Elementes

Um ein beliebiges Element aus der Liste entfernen zu können muss zunächst sein Vorgänger ermittelt werden. Dies geschieht mit der Funktion `searchPrevious`. Von dem Vorgänger wird dann der Zeiger `next` so geändert, dass er nun nicht mehr auf den nächsten, sondern den darauf folgenden Knoten zeigt. Damit kann der aktuelle Knoten gelöscht werden.
Der Zeitaufwand für das Entfernen eines Elementes ist durch die Ermittlung der Position und des Vorgängers O(n).
Abbildung 3.5 verdeutlich das Entfernen eines Elementes.

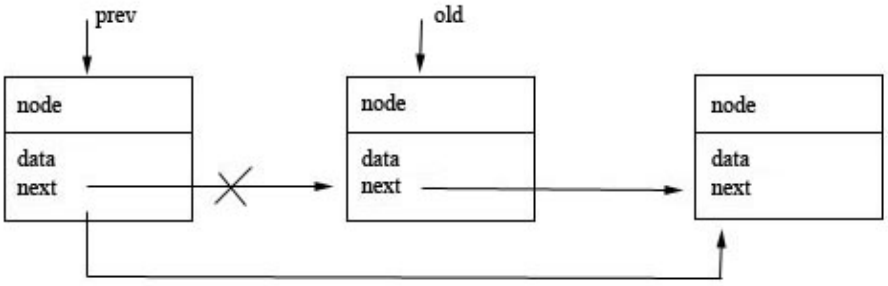


Abbildung 3.5: Einfach verkettete Liste: Entfernen eines gegebenen Elements

```
1 //Suchen eines Elementes
2 node *list::search(int sdata){ //search
3   node *pt = first; //am Anfang der Liste beginnen
4   do{
5     if(pt→data==sdata){ //Daten vergleichen
```

```
 6         return pt; //Daten ausgeben
 7       }
 8       pt = pt→next;//weiterlaufen
 9    }
10    while(pt→next!=0); //bis Liste zu Ende ist
11
12    return 0; //falls Element nicht gefunden
13 }
```

Sourcecode 3.10: Suchen eines Elementes

Um ein Element zu finden, muss dieses mit allen Elementen der Liste verglichen werden. Dazu muss die Liste durchlaufen werden. Im schlechtesten Fall ist das letzte Element das Gesuchte. Beim Durchlaufen der Liste ist der Aufwand abhängig von der Anzahl n der Elemente in der Liste und ist daher O(n).

```
1 //prüft ob die Liste leer ist
2 bool list::isEmpty( ){
3    if(first==0)
4       return true; //Liste ist leer
5    else
6       return false; //Liste ist nicht leer
7 }
```

Sourcecode 3.11: Überprüfen der Liste

Die Operation `isEmpty` überprüft die Liste auf Elemente indem sie den Listenkopft `first` testet. Diese Überprüfung erfordert nur einen Schritt und ist damit mit einem Aufwand von O(1) durchführbar.

```
 1 //prüft ob ein Element in der Liste ist
 2 bool list::isIn(int iData){
 3    node *pt = first; //am Anfang der Liste beginnen
 4       do{
 5          if(pt→data==iData){//schauen ob Element enthalten ist
 6          return true; //ja
 7          }
 8          pt = pt→next;
 9       }
10       while(pt→next!=0);//bis zum Ende der Liste
```

```
11
12      return false; //nein
13 }
```

Sourcecode 3.12: Überprüfen der Liste auf Elemente

Die Operation `isIn` überprüft ob ein gegebenes Element in der Liste enthalten ist. Diese Überprüfung erfordert das Durchlaufen der Liste und den Vergleich der Elemente. Dies verursacht einen Aufwand von O(n).

```
 1 //Suchen eines Vorgängerknotens
 2 node *list::searchPrev(int sdata){
 3   node *current=first; //aktueller Knoten ist der erste
 4   do
 5   {
 6     if(current→next!=0 && current→next→data==sdata){
 7       return current; //Vorgänger gefunden
 8     }
 9     current=current→next; //sonst weitersuchen
10   }
11   while (current→next!=0); //bis Liste zu ende ist
12
13   return 0; //Element nicht gefunden
14 }
```

Sourcecode 3.13: Suchen eines Vorgängerknotens

Die Operation `searchPrevious` stellt eine Hilfsfunktion dar und dient zur Suche des Vorgängers. Sie wird beim Entfernen an bestimmter Position benötigt und verursacht einen Aufwand von O(n).

3.3.3 Doppelt verkettete Liste

Eine doppelt verkettete Liste ist ähnlich wie eine einfach verkettete Liste aufgebaut. Sie hat ebenfalls zwei Klassen. Die Klasse `node` beinhaltet die Knoten. Diese bestehen aus den darin enthaltenen Daten `data` und zwei Zeigern `next und prev`. Der Zeiger `next` zeigt auf den Nachfolger und der Zeiger `prev` zeigt auf den Vorgänger. Dies ist auch der Unterschied zu einer einfach verketteten Liste. Hier wird ein zusätzlicher Zeiger auf den Vorgängerknoten verwendet. Die Klasse `dlist` definiert die Liste und die darin enthaltenen Operationen. Wie bei der einfach verketteten Liste werden auch hier die Operationen Einfügen, Entfernen, Suchen und Testen betrachtet.

In der h-Datei `dlist.h` werden die beiden Klassen `node` und `dlist`, sowie die benötigten Operationen deklariert.

```
 1 #ifndef DLIST_H
 2 #define DLIST_H
 3
 4 #include <cstdlib>
 5 using namespace std;
 6
 7 class node{
 8 public:
 9   int data;
10 private:
11   node *next, *prev;
12 };
13
14 class dlist{
15 public:
16   dlist(void);
17   ~dlist();
18   void insertAtBegin(int newData);
19   void insertAtEnd(int newData);
20   void insertAtPos(int newData, int pos);
21   void deleteAtBegin();
22   void deleteAtEnd();
23   void deleteAtPos(int pos);
24   void deleteData(int &oldData);
25   node *search(int sdata);
26   bool isEmpty();
27   bool isIn(int iData);
28 private:
29   node *first, *last;
30 };
31
32 #endif //DLIST_H
```

Sourcecode 3.14: dlist.h

In der cpp-Datei `dlist.cpp` werden die verschiedenen Operationen implementiert. Diese werden, wie in Abschnitt 3.1.3 spezifiziert, realisiert und einzeln mit ihrem jeweiligen Aufwand vorgestellt.

```
 1 #include „dlist.h“
 2
 3 dlist::dlist(void){
 4   first=0;
 5   last=0;
 6 }
 7
 8 dlist::∼dlist(){
 9   while(first!=0){
10     node *pt = first;
11     first=first→next;
12     delete pt; //Knoten löschen
13   }
14 }
```

Sourcecode 3.15: dlist.cpp

Im Konstruktor werden die Knoten `first` und `last` mit dem Anfangswert Null belegt. Damit ist die Liste zu Begin leer. Das Setzen der Anfangswerte verursacht einen Aufwand von O(1). Im Destruktor werden die Knoten der Liste gelöscht, wenn sie nicht mehr benötigt werden. Dadurch wird wieder Speicherplatz freigegeben. Dies erfordert einen linearen Aufwand.

```
 1 //Element am Anfang einfügen
 2 void dlist::insertAtBegin(int newData){
 3   node *pt = new node; //neuer Knoten
 4   if (first == 0){ //Liste leer
 5     last = pt; //hinten anfügen
 6   }
 7   else
 8     first → prev = pt; //sonst vorne
 9   pt → next = first; //an Liste anhängen
10   pt → prev = 0; //erstes Element hat keinen Vorgänger
11   first = pt; //pt ist neuer Anfang
12   pt → data = newData; //Daten einfügen
13 }
```

Sourcecode 3.16: Einfügen am Anfang

Beim Einfügen am Anfang einer doppelt verketteten Liste wird ein neuer Knoten am Anfang eingefügt. Dieser verweist auf das ursprüngliche erste Element. Dieses Element hat nun auch einen Vorgänger. Das neue Element hat keinen Vorgänger und verweist auf Null. Das Einfügen am Anfang ist in konstanter Zeit möglich, da die Liste vorher nicht

durchlaufen werden muss. Das neue Element kann direkt über den Listenkopft `first` eingefügt werden. Der Aufwand ist daher O(1).

```
1 //Element am Ende einfügen
2 void dlist::insertAtEnd(int newData){
3   node *pt=last→prev; //auf letzten Knoten zugreifen
4   node *n=new node; //neuer Knoten
5   pt→next=n;
6   n→next=0; //neuer Knoten hat keinen Nachfolger
7   n→prev=pt; //Vorgänger ist letzter Knoten vorher
8   n→data=newData; //Daten einfügen
9   last→prev = n;
10 }
```

Sourcecode 3.17: Einfügen am Ende

Das Einfügen eines Elementes am Ende der Liste kann direkt über den Verweis `last` auf das letzte Element erfolgen. Dies ist in einem Schritt möglich. Der Aufwand beträgt für das Einfügen am Ende einer doppelt verketteten Liste O(1).

```
1 //Element an bestimmter Position einfügen
2 void dlist::insertAtPos(int newData, int pos){
3   int number=0; //Anzahl der Elemente
4   int length; //Länge der Liste
5   for(node *pt=first; pt!=0; pt=pt→next)
6     number++; //zählen der Elemente
7   length = number;
8
9   node *pt=first; //erster Knoten
10  if(pos<0) //Positon außerhalb der Liste
11    return;
12  if(pos>=length) //Position größer Listenlänge
13    pos=length; //am Ende anhängen
14  if(pos==0){ //an den Anfang der Liste setzen
15    node *pt = new node; //neuer Knoten
16    if (first == 0){ //Liste leer
17      last = pt; //hinten anfügen
18    }
19    else
20      first→prev = pt; //sonst vorne
21    pt → next = first; //an Liste anhängen
```

```
22      pt → prev = 0; //erstes Element hat keinen Vorgänger
23      first = pt; //pt ist neuer Anfang
24      pt → data = newData; //Daten einfügen
25    }
26    while(pos>1){ //bis zum Vorgänger gehen und davor einfügen
27      pt=pt→next;
28      pos - -;
29    }
30    node *p = new node; //neuer Knoten
31    p→next = pt→next; // neuer Knoten zeigt auf den Knoten
32                      // wo der alte hingezeigt hat
33    pt→next = p; //alter Knoten zeigt jetzt auf den neuen Knoten
34    p→prev = pt;
35    pt→next→next→prev=p;
36    p→data=newData; //Daten einfügen in den neuen Knoten
37  }
```

Sourcecode 3.18: Einfügen an bestimmter Position

Das Einfügen an beliebiger Stelle in der Liste erfordert die Kenntnis des Vorgängers. Dieser wird ermittelt indem bis zu seiner Position traversiert wird. Sein `next`-Zeiger wird auf den neuen Knoten gesetzt. Die Zeiger des neuen Knotens verweisen auf den Vorgänger und den Nachfolger. Der Nachfolger des alten Knotens zeigt nun auf den neuen Knoten. Da die Operation das Durchlaufen der Liste mit n Elementen erfordert beträgt der Aufwand O(n).
Abbildung 3.6 verdeutlich das Einfügen an beliebiger Position in eine doppelt verkettete Liste.

```
1  //Element am Anfang löschen
2  void dlist::deleteAtBegin(){
3    if(first==0) //Liste ist leer
4      return;
5    node *pt=first; //am Anfang beginnen
6    first=first→next; //first zeigt auf den übernächsten Knoten,
7                //damit der davor gelöscht werden kann
8    if(first!=0)
9      first→prev=0; //first hat keinen Vorgänger
10   delete pt; //Knoten löschen
11 }
```

Sourcecode 3.19: Löschen am Anfang

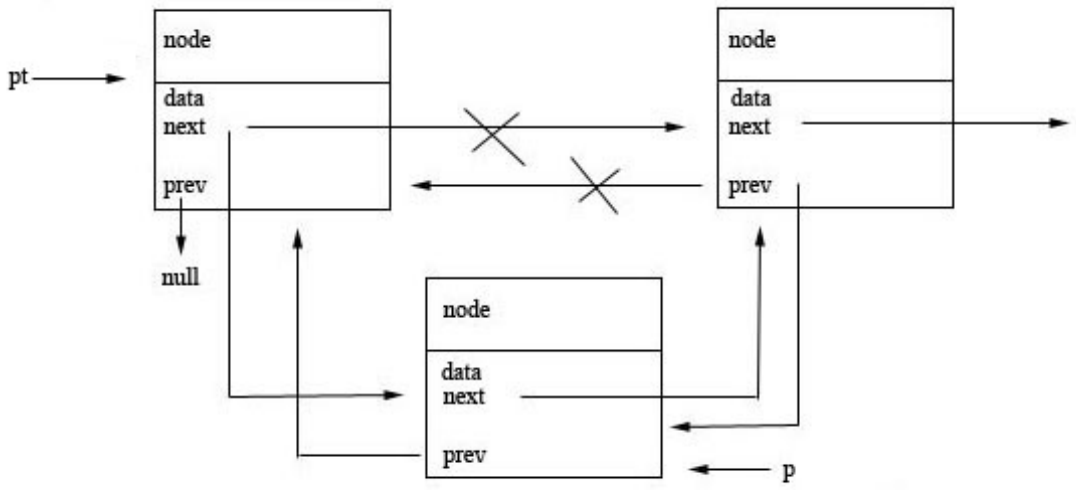


Abbildung 3.6: Doppelt verkettete Liste: Einfügen eines Elementes an beliebiger Stelle

Das Entfernen eines Elements am Anfang verursacht einen Zeitaufwand von O(1), da das Entfernen unabhägig von der Länge der Liste einen konstanten Aufwand verursacht. Es wird der Zeiger `first` umgesetzt und der erste Knoten entfernt.

```
1 //Element am Ende löschen
2 void dlist::deleteAtEnd(){
3   if(first==0) //Liste ist leer
4     return;
5   node *pt = last;//auf das Ende zugreifen
6   node *n = pt→prev;//Zeiger umsetzen
7   n→next=last;
8   last→prev=n;
9   delete pt; //Element löschen
10 }
```

Sourcecode 3.20: Löschen am Ende

Das Entfernen eines Elementes am Ende der Liste hat einen Aufwand von O(1). Über `last` kann direkt auf das letzte Element zugegriffen werden. Da auch die Vorgänger bekannt sind, können die Zeiger leicht umgesetzt werden.

```
1 //Element an bestimmter Position löschen
2 void dlist::deleteAtPos(int pos){
3   int number=0; //Anzahl der Elemente
4   int length; //Listenlänge
5
6   for(node *pt=first; pt!=0; pt=pt→next)
7     number++;//Anzahl der Elemente zählen
8   length = number;
9
10  if(first!=0){ //Liste ist nicht leer
11    node *pt=first;
12    if(pos<0) //negative Position
13      return;
14    if(pos>=length)//zu große Position
15      return;
16    while ( pos > 0 ) {
17      pt = pt→next; //durchlaufen der Liste
18    pos - -;
19    }
20    pt=pt→next; //Zeiger umsetzen
21    pt→next→prev=pt;
22    delete pt; //Element löschen
23  }
24 }
```

Sourcecode 3.21: Löschen an bestimmter Position

Um an einer gegebenen Position ein Element zu entfernen wird zur der gegebenen Position-1 traversiert. Damit ist der Vorgänger bekannt. Der `next`-Zeiger wird dann auf den neuen Nachfolger gesetzt und das Element an der gegebenen Position kann gelöscht werden. Zusätzlich muss bei einer doppelt verketteten Liste noch der Nachfolger auf den neuen Knoten verweisen. Die Operation erfordert das Durchlaufen der Liste und ist von deren Anzahl n der Elemente abhängig. Dadurch entsteht ein Aufwand von O(n).

```
1 //Löschen eines bestimmten Elementes
2 void dlist::deleteData(int &oldData){
3   if(first!=0){
4     node *p = old→prev; //Vorgänger
5     node *n = old→next; //Nachfolger
6
7     if(p==0){ //falls kein Vorgänger, also erstes Element
8       node *pt=first; //am Anfang beginnen
9       first=first→next; //first zeigt auf den übernächsten Knoten,
```

```
10                  //damit der davor gelöscht werden kann
11          delete pt; //Knoten löschen
12      }
13      else {
14          if(old→next==0) //schauen ob das Element das entfernt wird
15                          // das letzte ist
16              p→next=0; // wenn ja dann next auf 0 setzen
17          else{
18              p→next=old→next; //sonst next auf den Nachfolger von old setzen
19              n→prev = old→prev; //prev auf Vorgänger von old setzen
20          }
21      delete old; //Element entfernen
22      }
23  }
24 }
```

Sourcecode 3.22: Löschen eines bestimmten Elementes

Das Entfernen eines bestimmten Elementes erfordert einen Aufwand von O(n). Die Vorgänger und Nachfolger können direkt über die Zeiger abgelesen werden. Dadurch können diese leicht umgesetzt und das Element gelöscht werden. Zuvor muss zu dem zu löschenden Element traversiert werden.
Abbildung 3.7 verdeutlich das Entfernen eines Elementes aus einer doppelt verketteten Liste.

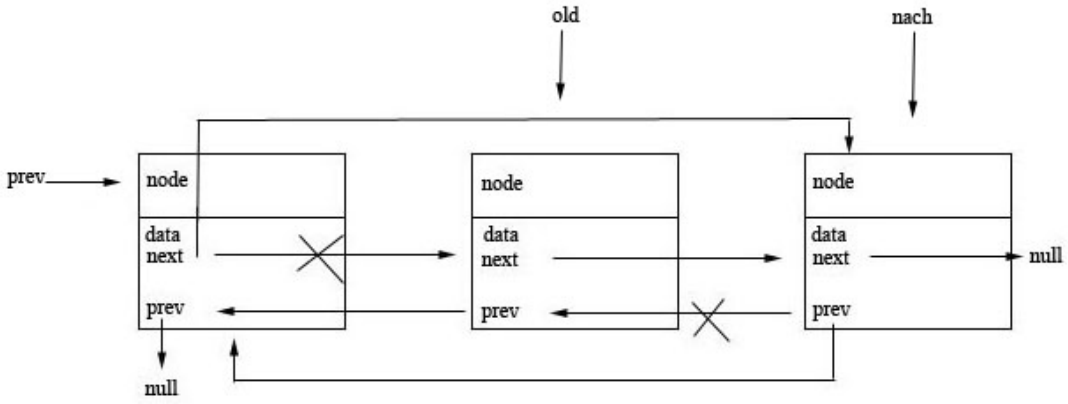


Abbildung 3.7: Doppelt verkettete Liste: Entfernen eines gegebenen Elements

```
1 //Suchen eines Elementes
2 node *dlist::search(int sdata){ //search
3   node *pt = first; //am Anfang der Liste beginnen
4   do{
5     if(pt→data==sdata){ //Elemente vergleichen
6       return pt; //Element ausgeben
7     }
8     pt = pt→next; //weiterlaufen
9   }
10  while(pt→next!=0); //bis Liste zu Ende
11
12  return 0; //Element nicht gefunden
13 }
```

Sourcecode 3.23: Suchen eines Elementes

Beim Suchen eines Elementes muss die Liste durchlaufen und das gesuchte Element mit den Elementen in der Liste verglichen werden. Bei einer Liste mit n Elementen sind n Schritte nötig. Es entsteht ein Aufwand von O(n).

```
1 //überprüft ob die Liste leer ist
2 inline bool dlist::isEmpty(){
3   if(first==0)
4     return true; //Liste leer
5   else
6     return false; //Liste nicht leer
7 }
```

Sourcecode 3.24: Prüfen der Liste

Die Operation `isEmpty` prüft, ob die Liste leer ist. Trifft `true` zu, dann ist die Liste leer, ansonsten ist die Liste nicht leer. Diese Überprüfung erfordert nur einen Schritt und ist damit mit einem konstanten Aufwand, also O(1), durchführbar.

```
1 //prüft ob ein Element in der Liste ist
2 bool dlist::isIn(int iData){
3   node *pt = first; //am Anfang der Liste beginnen
4     do{
5       if(pt→data==iData){//schauen ob Element enthalten ist
6       return true; //ja
7       }
8       pt = pt→next;
```

```
 9      }
10      while(pt→next!=0);//bis zum Ende der Liste
11
12      return false; //nein
13  }
```

Sourcecode 3.25: Prüfen der Liste auf Elemente

Die Operation `isIn` überprüft ob ein gegebenes Element in der Liste enthalten ist. Diese Überprüfung erfordert das Vergleichen der Elemente. Dabei wird die Liste mit einem linearen Aufwand (O(n)) durchlaufen.

3.4 Zusammenfassung und kurze Bewertung

Der Aufwand bei einer sequentiell gespeicherten Liste der Länge n beträgt für das Einfügen und Entfernen am Ende O(1). Dieser Aufwand ergibt sich durch den Aufbau sequentieller Listen. Die Elemente werden im Normalfall jeweils am Ende eingefügt. Das Einfügen und Entfernen am Anfang und an beliebiger Position verursacht einen Aufwand von O(n), obwohl über einen Index auf die jeweiligen Elemente zugegriffen werden kann. Es ist jedoch nötig alle Elemente um eine Stelle zu verschieben. Das Verschieben der Elemente ist auch beim Enfernen nötig. Das Suchen hat ebenfalls einen Aufwand von O(n). Hierbei müssen alle Elemente der Liste mit dem gesuchten Element verglichen werden. Die sequentielle Speicherung hat den Nachteil, dass die Anzahl der Speicherplätze vorher bestimmt wird. Wird das Feld dynamisch erweitert beträgt der Aufwand beim Einfügen und Entfernen auch am Ende der Liste O(n), da alle Elemente in ein neues größeres Feld kopiert werden müssen.

Einfach verkettete Listen haben den Vorteil gegenüber sequentiellen Listen, dass nur soviel Speicherplatz belegt wird, wie tatsächlich Elemente vorhanden sind. Das Einfügen und Entfernen am Anfang der Liste ist mit einer konstanten Zeit möglich. Hierfür muss das neue Element an den Anfang der Liste angehängt werden. Das Einfügen und Entfernen an beliebiger Stelle erfordert die Kenntnis des Vorgängers. Der Vorgänger erhält durch das Einfügen oder Entfernen eines Elementes einen neuen Nachfolger. Um den Vorgänger zu ermitteln muss die Liste durchlaufen werden. Dies erzeugt einen Aufwand von O(n). Dieser Aufwand ist auch für das Entfernen an einer beliebigen Stelle in der Liste nötig. Die gleiche Vorgehensweise gilt auch für das Entfernen am Ende der Liste, da kein direkter Verweis auf das Ende vorhanden ist.

Elemente innerhalb einer doppelt verketteten Liste zeigen jeweils auf den Vorgänger und den Nachfolger. Dadurch entfällt das aufwendige Ermitteln der Vorgänger wie bei der einfach verketteten Liste. Diese können direkt über die Zeiger abgelesen werden. Die Suche des Elementes bzw. das Ermitteln seiner Position verursachen einen linearen Aufwand.

Die in diesem Kapitel vorgestellten grundlegenden Operationen auf Listen und ihren Aufwand fasst Tabelle 3.2 zusammen.

Operation	sequentiell	einfach verkettet	doppelt verkettet
insertAtBegin: list x data ⟶ list	O(n)	O(1)	O(1)
insertAtEnd: list x data ⟶ list	O(1)	O(n)	O(1)
insertAtPos: list x data x pos ⟶ list	O(n)	O(n)	O(n)
deleteAtBegin: list ⟶ list	O(n)	O(1)	O(1)
deleteAtEnd: list⟶ list	O(1)	O(n)	O(1)
deleteAtPos: list x pos ⟶ list	O(n)	O(n)	O(n)
deleteData: list x data ⟶ list	O(n)	O(n)	O(n)
search: list x data ⟶ pos	O(n)	O(n)	O(n)
isEmpty: list ⟶ bool	O(1)	O(1)	O(1)
isIn: list x data ⟶ bool	O(n)	O(n)	O(n)

Tabelle 3.2: Aufwand von Operationen auf Listen

Je nach Art der benötigten Informationen sollte eine entsprechende Listenspeicherung gewählt werden. Sequentielle Listen eignen sich gut für kleine Datenbestände, deren Anzahl der zu speichernden Elemente bekannt ist, und das Einfügen und Entfernen hauptsächlich am Ende der Liste geschieht. Dadurch entfällt das aufwendige Verschieben der Elemente.

Verkettete Listen eignen sich gut für Daten, deren Anzahl nicht bekannt ist oder sich oft ändert, da die Listen dynamisch aufgebaut sind. Die Wahl zwischen einer einfach oder doppelt verketteten Liste hängt von den Operationen, die angewendet werden sollen, ab. Wird oft an verschiedenen Positionen eingefügt und entfernt, sollte eine doppelt verkettete Liste gewählt werden. Hierbei ist auch der Vorgänger bekannt, damit fällt das aufwendige Bestimmen des Vorgängers, wie bei der einfach verketteten Liste, weg. Werden diese beiden Operationen nur auf den Anfang der Liste angewendet, reicht auch eine einfach verkettete Liste aus. Zu bedenken ist bei der doppelt verketteten Liste, dass durch die zusätzlichen Zeiger auf den Vorgänger zusätzlicher Speicheraufwand entsteht.

4 Repräsentation von 3D-Modellen

Dieses Kapitel dient dazu die verschiedenen Repräsentationsformen von 3D-Modellen einzuführen. Bevor auf die einzelnen Repräsentation eingegangen wird, erfolgt eine kurze Beschreibung des Einsatzgebietes und der Erstellung dieser Repräsentationen. Danach wird jede Repräsentationsform einzeln vorgestellt. Die vier Repräsentationsformen sind die *Punktwolke, das Drahtmodell, das Flächenmodell und das Volumenmodell*. Den Schwerpunkt dieser Arbeit bildet das Flächenmodell. Dieses wird im nächsten Kapitel ausführlich behandelt. Um es einordnen zu können, wird in diesem Kapitel auf alle Modelle eingegangen.

3D-Modelle werden heute in vielen Bereichen eingesetzt. Unter anderem finden sie Anwendung in technischen Bereichen wie Maschinenbau, in der Medizin, Bildverarbeitung und Computergraphik.

3D-Modelle können auf verschiedene Arten entstehen. Die erste Möglichkeit ein Modell zu erstellen ist die mit Hilfe eines *CAD-Systems* wie Maya[1] oder 3ds Max[2]. CAD steht für Computer Aided Design und bedeutet „computergestütztes Entwerfen“. Mit Hilfe dieser Systeme können die Modelle in einer Benutzeroberfäche erstellt werden. Die nötigen Vorgänge für das Erstellen des Modells können über die Oberfläche ausgewählt werden. Die mathematische Berechnung wird automatisch vom System ausgeführt. Diese Art der Modell-Erstellung wird als *Modellierung* bezeichnet. Eine weitere Möglichkeit ein Modell zu generieren kann anhand von Messwerten erfolgen. Ein Beispiel herfür sind CT-Scans. Dies wird als *Visualisierung* bezeichnet. Die dritte Möglichkeit der Modellerstellung ist die *Rekonstruktion*. Diese kann aus von einer Stereo- oder 3D-Kamera aufgenommenen Bildern oder Daten eines 3D-Scanners erfolgen.

Die entstandenen Modelle können unterschiedlich repräsentiert werden. So entstehen bei der Modellierung meist Flächenmodelle, bei der Visualisierung Volumenmodelle und bei der Rekonstruktion Punktwolken.

Betrachtet man die nachfolgenden Modellarten, so ergibt sich eine *Reihenfolge*. Die Punktwolke stellt die einfachste Art der Repräsentation dar. Das Drahtmodell liefert mehr Informationen als die Punktwolke. Hier sind bereits die Zusammenhänge der Kanten bekannt. Das Flächenmodell erweitert diese Darstellung um Oberflächen zwischen den Kanten. Dadurch wirkt das Modell realistischer. Oberflächen können mit Farbe oder Textur belegt werden. Das Modell an sich ist jedoch noch hohl. Dies wird beim Volumenmodell ergänzt. Hier werden volle Körper verwendet. Von der Punktwolke bis zum

[1] http://www.alias-systems.de
[2] http://www.autodesk.com

Volumenmodell bilden die Modelle jeweils eine Erweiterung des Vorgängers. Einige der Modelle lassen sich ineinander überführen. So kann eine Punktwolke durch *Triangulierung* in ein Draht- oder sogar ein Flächenmodell überführt werden, und ein Drahtmodell kann zu einem Flächenmodell durch Bestimmung der Oberflächen zwischen den Kanten erweitert werden.

Abbildung 4.1 zeigt alle Modelle zusammenhängend in einem Klassendiagramm. Das Klassendiagramm beinhaltet nur die in dieser Arbeit behandelten Zusammenhänge.

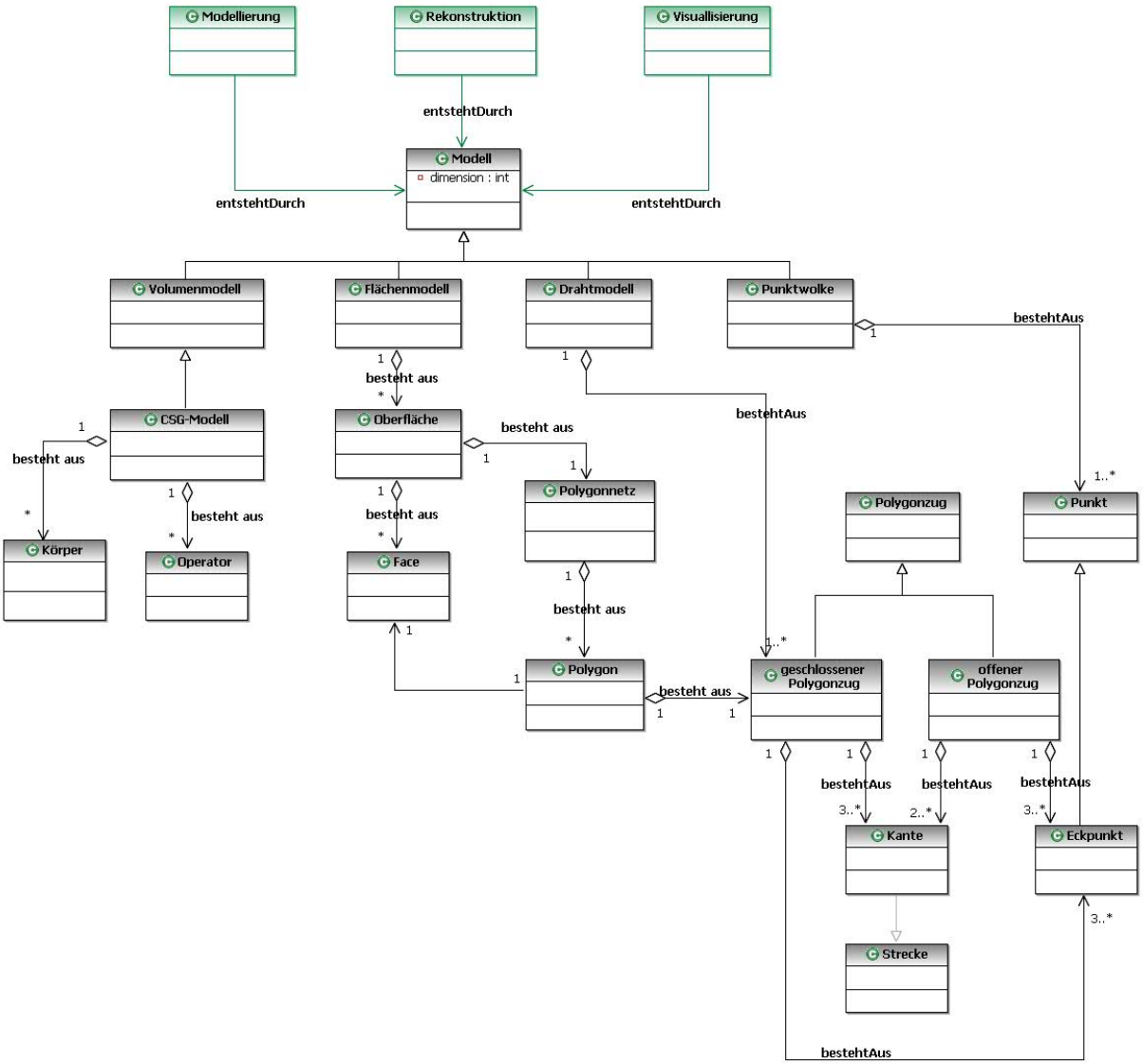


Abbildung 4.1: Klassendiagramm: Modelle im Überblick

4.1 Die Punktwolke

Die *Punktwolke* ist eine einfache Repräsentation dreidimesionaler Daten, bei der ein Objekt durch eine Menge von Punkten repräsentiert wird. Die Punkte sind im Raum verteilt und liefern keine direkte Information über die Oberflächen zwischen den Punkten oder das Volumen des Modells. Je dichter die Punktwolke, desto genauer kann das Objekt dargestellt werden. Dennoch ist *keine realitätstreue Darstellung* möglich. Aus einer Punktwolke kann eine Objektoberfläche, z.B. durch Aufspannen von Polygonen zwischen am nähesten benachbarten Punkten, ermittelt werden. Dieses Verfahren wird Triangulierung genannt und in einem späteren Abschnitt vorgestellt. Abbildung 4.2 zeigt ein Beispiel einer Punktwolke.

Punktwolken werden oft genutzt um Objekte in einer schnellen vereinfachten Form darzustellen. Hat man genug Punkte um eine geschlossene Oberfläche im Bild zu erzielen, lassen sich Objekte sogar exakt als Punktwolke darstellen.

Zur Speicherung dient eine Menge von Eckpunkten der Punktwolke. Diese Art der Speicherung kann verwendet werden, da eine bestimmte Reihenfolge der Punkte nicht erforderlich ist. Zudem werden keine Beziehungen zwischen den Eckpunkten verwaltet. Die Eckpunkte, aus denen ein Objekt besteht, werden mit ihren Koordinaten x, y und z abgelegt. Ein Objekt ist also eine Menge von Punkten und ein Punkt besteht aus seinen Koordinaten.

Nachfolgend wird zunächst auf die Möglichkeiten der Gewinnung einer Punktwolke eingegangen. Anschließend folgen die Epipolargeometrie und die Triangulierung. Die Epipolargeometrie dient zur Gewinnung einer Punktwolke aus Kamerabildern und die Triangulierung ist ein Verfahren zur Erstellung von Oberflächen zwischen den Punkten. Hier werden verschiedene Verfahren kurz vorgestellt. Zum Schluß werden die Vor- und Nachteile der Punktwolke aufgeführt.

Abbildung 4.2: Punktwolke

4.1.1 Gewinnung einer Punktwolke

Eine Punktwolke kann auf verschiedene Weisen entstehen. Die bekanntesten sind die Gewinnung aus Bildern einer Stereokamera und direkt durch einen 3D-Scanner.

Eine Stereokamera ist eine spezielle Kamera zur Aufnahme von Bilder zur Tiefenrekonstruktion. Stereokameras besitzen in der Regel zwei nebeneinander angebrachte Objektive, welche die gleichzeitige Aufnahme der erforderlichen beiden stereoskopischen Halbbilder ermöglichen. Stereoskopie bedeutet, dass zwei Bilder die gleiche räumliche Aufnahme leicht versetzt abbilden. Diese Art der Aufnahme ist den menschlichen Augen nachempfunden. Jedes Halbbild liefert ein Bild für jedes Auge. Auf den Bilder wird jeder Raumpunkt durch korrespondierende Bildpunkte auf jedem Halbbild abgebildet. D.h. jeder Raumpunkt ist auf den beiden Bildern vorhanden, allerdings durch die versetzte Aufnahme an einer anderen Stelle. Aus den beiden Halbbildern kann die Tiefe jedes Raumpunktes mathematisch ermittelt werden. Die Gesamtheit der Punkte wird dann als Punktwolke bezeichnet. Statt einer Stereokamera können auch zwei optische Kameras mit einem festen Abstand voneinander verwendet werden.

Eine weitere Methode zur Gewinnung der Punktwolke stellt der 3D-Scanner dar. Der 3D-Scanner (nach [Bro03] tastet reale Objekte ab. Das Objekt wird auf einen gleichmäßig rotierenden Drehteller gesetzt und von Lichtstrahlen oder von einem Laserstrahl abgetastet. Je nach Modell registrieren entweder Sensoren oder digitale Kameras, wie der Strahl reflektiert wird. Aus einer Vielzahl der Abtastungen ergibt sich ein räumliches Bild, welches durch eine Punktwolke repräsentiert wird.

Abbildung 4.3 zeigt alle erforderlichen Schritte zur Gewinnung eines Objektes aus einer Punktwolke. Wird die Punktwolke anhand eines 3D-Scanners ermittelt, so liefert dieser direkt die Punktwolke als Ergebnis. Je nachdem wie dicht abgetastet wurde wird die Punktwolke weiterverwendet. Bei einer sehr dichten Abtastung stellt die Punktwolke das Objekt exakt dar. Bei einer relativ niedrigen Dichte können Flächen zwischen den Punkten rekonstruiert und so das 3D-Modell dargestellt werden. Das Errechnen der Flächen geschieht anhand eines Triangulierungsverfahrens.

Steht eine Kamera zur Gewinnung der Punktwolke zur Verfügung so werden zunächst mit Hilfe der Epipolargeometrie die korrespondierenden Punkte der beiden Halbbilder ermittelt. Mit Hilfe eines weiteren Verfahrens, wie beispielsweise eines Stereo-Algorithmus, wird die Punktwolke, oder auch Tiefenkarte genannt, erstellt. Dieser Algoritmus beruht auf der Disparität der Bilder und dem Wissen über die Kameraparameter. Er wird in [Kon99] vorgestellt. Als Disparität wird der Unterschied zwischen zwei Bildern bezeichnet, welche das gleiche Objekt aus zwei verschiedenen Positionen darstellen. Die entstandene Punktwolke liefert die zur Rekonstruktion des 3D-Modells nötigen Tiefeninformationen. Anhand dieser Informationen werden mit Hilfe der Triangulierung die Flächen zwischen den Punkten bestimmt und das 3D-Modell erstellt.

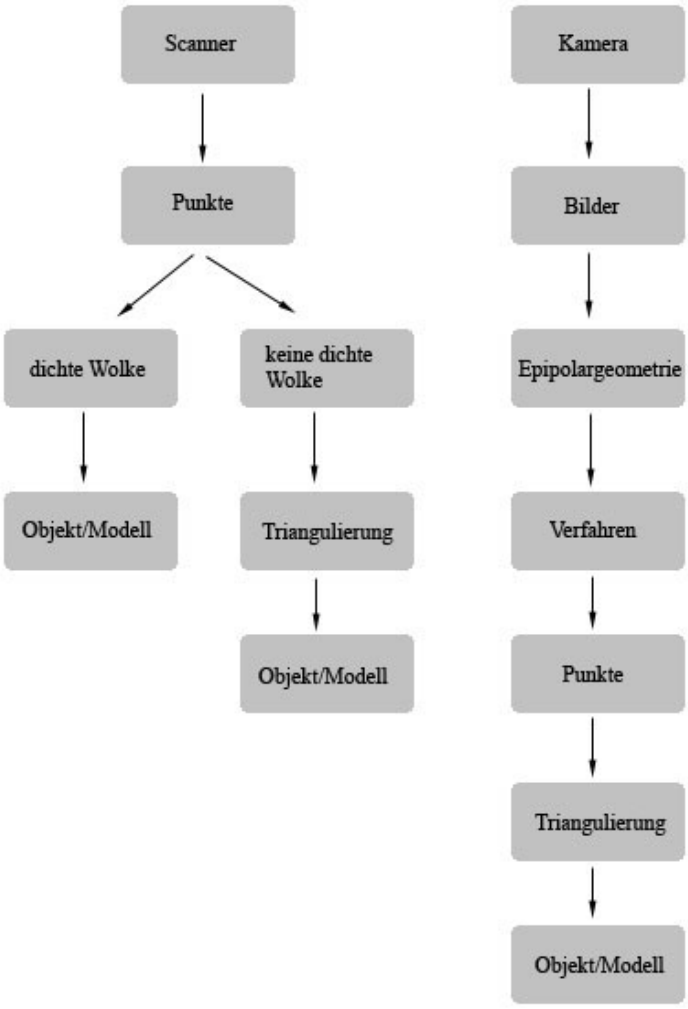


Abbildung 4.3: Schritte zur Gewinnung einer Punktwolke und eines Objektes aus dieser

4.1.2 Epipolargeometrie

In diesem Abschnitt wird die *Epipolargeometrie* erklärt. Diese bildet die Grundlage zur Gewinnung der Punktwolke aus Bildern. Die folgenden Informationen wurden [Pau05] entnommen.

Wichtigstes Ziel der *Rekonstruktion* von 3D-Objekten aus 2D-Bildern ist die Bestimmung *korrespondierender Punkte*. Ein korresepondierendes Punktepaar beschreibt den gleichen 3D-Punkt in der realen Welt. Dabei befinden sich die beiden Punkte in zwei verschiedenen Bildern, die die selbe Szene aus zwei verschiedenen Blickwinkeln betrachten. Mit Hilfe dieser Punkte können einzelne Objekte und ganze Szenen rekonstruiert werden. Die Epipolargeometrie dient als mathematisches Hilfsmittel zur Bestimmung der korrespondierenden Punkte und ist die Grundlage für alle Methoden der 3D-Rekonstruktion

aus Bildern.

Abbildung 4.4 zeigt das Basisszenario der Epipolargeometrie, welches die Grundlagen zur Ermittlung der Korrespondenzen liefert.

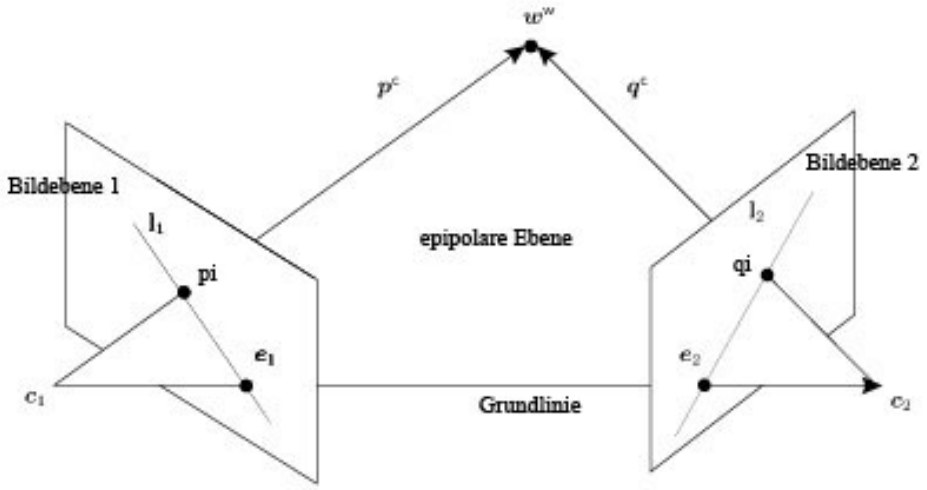


Abbildung 4.4: Basisszenario der Epipolargeometrie

Für dieses Szenario werden zwei Bilder verwendet, die die gleiche Szene aus zwei verschiedenen Blickwinkeln zeigen. Die Position und Orientierung der Kameras ist verschieden, aber bekannt. c_1 ist das Zentrum der ersten Kamera im *Kamerakoordinatensystem (3D)* und ist gleichzeitig der Ursprung des *Weltkoordinatensystems*. Dadurch gilt $w_w = p_c$. Das Weltkoordinatensystem beschreibt die Lage aller Objekte sowie die Position und Orientierung der Kamera in 3D. Sein Ursprung ist frei wählbar. Das Kamerakoordinatensystem beschreibt die Szene aus der Sicht der Kamera. Dabei entspricht die z-Achste der Blickrichtung der Kamera. c_2 ist das Zentrum der zweiten Kamera. w_w ist ein Punkt in der Welt. Dieser ist in Weltkoordinaten (3D) gegeben. Der Punkt p_c ist aus Sicht der ersten Kamera gleich w_w in Kamerakoordinaten (3D). q_c ist aus Sicht der zweiten Kamera w_w. p_i bildet w_w in *Bildkoordinaten (2D)* auf die erste Bildebe ab und q_i bildet w_w in Bildkoordinaten (2D) auf die zweite Bildebe ab.

Die Verbindungslinie zwischen c_1 und c_2 heißt *Grundlinie*. Alle Punkte sowie die Grundlinie liegen in einer Ebene. Diese wird *epipolare Ebene* genannt. Die Punkte e_1 und e_2 nennt man *Epipole*. Sie bilden die Schnittstelle der Grundlinie mit den *Bildebenen*. l_1 und l_2 sind die *epipolaren Linien*. Sie gehen jeweils durch das Epipol und den Punkt der jeweiligen Bildebene. Alle epipolaren Linien einer Bildebene treffen sich im Epipol. Die Epipolarlinien haben noch eine weitere wichtige Bedeutung. Sie sind die Linien auf

denen sich jeweils die korrespondierenden Punkte befinden. D.h. l_2 beihaltet die korrespondierenden Punkte zu p_i und l_1 die zu q_i.

Mit der Epipolargeometrie lassen sich die korrespondierenden Punkte nicht genau bestimmen, sondern nur eingrenzen. Warum das so ist soll anhand von Abbildung 4.5 erklärt werden.

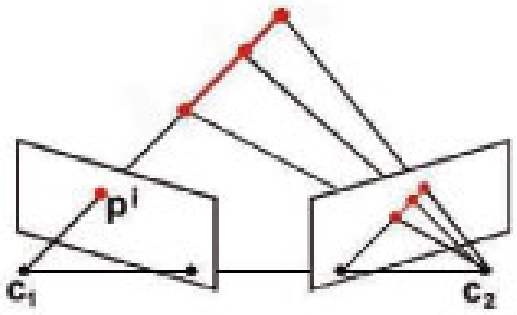


Abbildung 4.5: Basisszenario der Epipolargeometrie

Da bei 2D-Bildern die Tiefeninformation verloren geht werden Punkte die hintereinander liegen auf nur einen Punkt abgebildet. D.h. die Punkte die auf dem Vektor p_c liegen würden alle auf den Punkt p_i abgebildet. Eigentlich wird nur der vorderste Punkt, der mit der kleinsten z-Koordinate, abgebildet. Die restlichen Informationen gehen verloren. In der zweiten Bildebene ergeben diese hintereinander liegenden Punkte alle einen anderen Punkt. Alle diese Punkte bilden die epipolare Linie l_2. Die Epipolargeometrie liefert also zu jedem Punkt in der ersten Bildebene eine Linie, die die korrespondierenden Punkte enthält, und grenzt somit die Anzahl der korrespondierenden Punkte ein.

Um einen Punkt auf seine epipolare Linie abzubilden existieren in der Epipolargeometrie zwei Matrizen. Die *essentielle Matrix (E-Matrix)* und die *fundamentale Matrix (F-Matrix)*. Welche zur Berechnung benutzt wird entscheiden die bekannten Kameraparameter und die Koordinaten des Bildes, Bild- oder Pixelkoordinaten, wobei beide ineinander umgerechnet werden können. Beide Koordinatensysteme sind 2D. Sie unterscheiden sich nur in ihrem Ursprung und der Art des Bildes. *Pixelkoordinatensysteme* haben ihren Ursprung links oben, Bildkoordinatensysteme links unten. Bildkoordinaten beschreiben ein analoges Bild, Pixelkoordinaten ein Digitales. Hat man zwei Bilder und die benötigten Kameraparameter, kann man mit Hilfe einer der beiden Matrizen zu einem Punkt in der ersten Bildebene die epipolare Linie in der zweiten Bildebene berechnen. Die E-Matrix bildet einen Punkt in Bildkoordinaten auf seine epipolare Linie ab, die F-Matrix in Pixelkoordinaten. Für die E-Matrix reicht die Kenntnis über die *extrinsischen Kameraparameter* aus. Die F-Matrix benötigt die *intrinsischen und extrinsischen Kameraparameter*. Sie baut auf der E-Matrix auf. Zu den extrinsichen Parametern gehören

die Orientierung und die Position der Kamera, zu den intrinsischen Parametern zählen Brennweite, Verzerrungskoeffizienten, ein horizontaler Skalierungsfaktor und der Hauptpunkt, der den Schnittpunkt der optischen Achse mit der Bildebene darstellt. Durch Brennweite und den Hauptpunkt wird neben den optischen und geometrischen Eigenschaften der Kamera auch die Beziehung zwischen Kamera- und Bildkoordinatensystem beschrieben. Sie werden durch die Bewegung der Kamera nicht verändert.

Wie bereits erwähnt können die genauen Korrespondenzen und die Tiefenwerte beispielsweise mit Hilfe eines Stereo-Algorithmus ermittelt werden. Sind korrespondierende Bildpunkte gefunden, werden Kamerainformationen herangezogen, um die Tiefe der Bildpunkte zu bestimmen.

4.1.3 Triangulierungsverfahren

Die *Triangulierung* ist ein Verfahren zur *Dreiecksbildung*. Es existieren zwei Ansätze: die Triangulierung vorhandender Polygone und die Triangulierung einzelner Punkte. Bei einer Punktwolke werden Verfahren angewendet, welche aus einer gegebenen, unsortierten Punktwolke im 2- oder 3-dimensionalen Raum ein Netz modellieren und damit die Flächen der Objekte angeben. Drei dieser Verfahren sollen kurz vorgestellt werden. Sie sind unter anderem in [Pil02] zu finden. Das *TOREC-Verfahren* erzeugt sowohl Dreiecke als auch Rechtecke und stellt daher kein reines Triangulierungsverfahren dar.

4.1.3.1 TRIADS

TRIADS steht für *Triangulation of Dotted Surfaces* und erstellt aus einer Punktwolke ein Dreiecksnetz, welches das Objekt beschreibt. Das Vorgehen bei diesem Verfahren ist in Abbildung 4.6 dargestellt. Durch schrittweises Hinzufügen von Dreiecken werden alle Punkte in das Netz aufgenommen. Zwischen drei beliebigen Punkten, welche nah beieinander liegen wird das erste Dreieck gebildet. Dabei bildet die letzte Strecke eines Dreiecks die Basis für das nächste Dreieck. Die Dreiecksbildung erfolgt so lange bis keine Dreiecke mehr vorhanden sind. Das Verfahren ist nicht bei hoher Punktdichte anwendbar wie diese bei einem Scanner vorkommt. In diesem Fall können durch die Nähe der Punkte keine Dreiecke zwischen diesen gebildet werden.

4.1.3.2 TOREC

TOREC ist ein weiteres Verfahren zur Netzbildung aus Punktwolken und bedeutet *Triangle Or Rectangle*. Bei diesem Verfahren werden sowohl Dreiecke als auch Vierecke gebildet. Das Verfahren ist speziell für Scanner-Messdaten gedacht. Die Rekonstruktion der Flächen erfolgt aus den Abtaststrahlen eines Laserscanners. Dieser liefert die Punkte auf sogenannten Scanlinien. Auf diesen Linien haben alle Punkte die selbe x-Koordinate. Um die Flächen zwischen den Punkten zu bestimmen werden jeweils zwei Scanlinien betrachtet. Zwischen diesen zwei Scanlinien werden Dreiecke oder Vierecke, je nach Lage der Punkte gebildet.

Abbildung 4.7 zeigt die Schritte des TOREC-Verfahrens. P1, P2 und P3 bilden jeweils

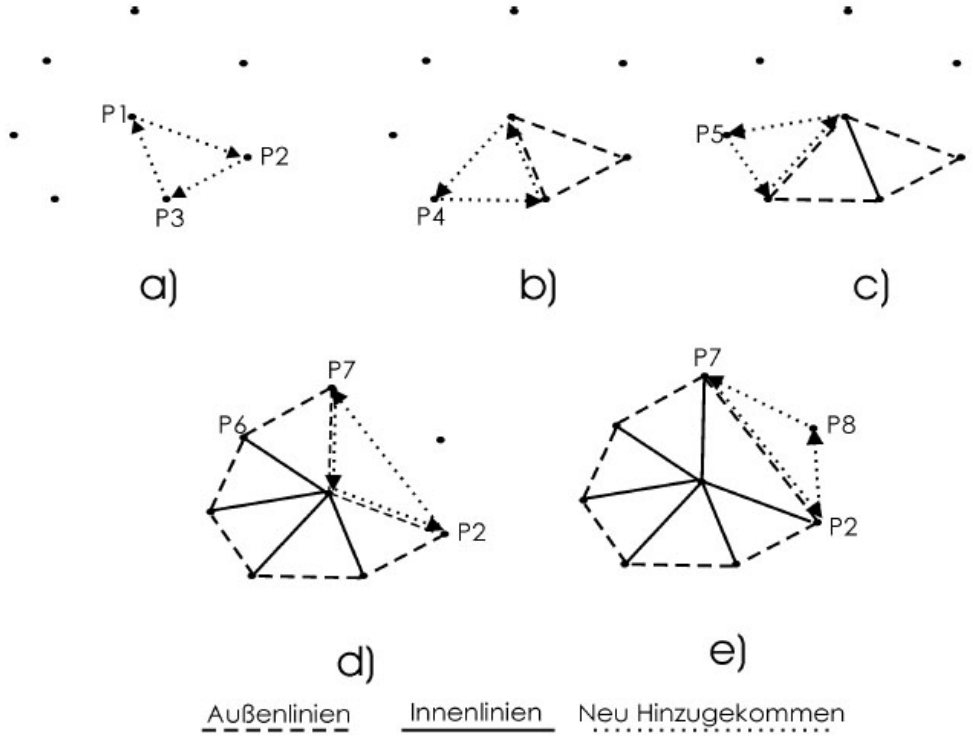


Abbildung 4.6: Prinzip des TRIADS-Verfahrens, Quelle:[Pil02]

ein Dreieck und P1, P2, P3 und P4 ein Viereck. P1 und P2 bleiben stets auf der selben Scanlinie. Sie bilden die erste Seite des Dreiecks. Ausgehend von der Strecke zwischen P1 und P2 ist P3 der erste Nachbar. Somit bilden P1, P2 und P3 das erste Dreieck. P3 wechselt nun die Seite, P2 rückt an die Position von P3. Damit entsteht das nächste Dreieck. Wiederrum wechselt P3 die Scanlinie. P1 wandert nun an die alte Stelle von P3. P2 bleibt an seiner Position. Nun wird ein Viereck mit P3 und P4 konstruiert. Vierecke entstehen immer dann, wenn die beiden Dreiecksflächen, die anstelle des Vierecks hätten konstruiert werden können, fast parallel zueinander liegen würden. Diese Vorgehensweise wird solange durchgeführt bis man am Ende der Scanlinien angekommen ist. Die Vorgehensweise wird für alle Scanlinien durchgeführt und führt am Ende zu einer Beschreibung des Objektes durch ein Dreiecks- und Viereckснetz. Dieses Verfahren ist nur dann anwendbar wenn die Scanlinien, also das Wissen über die Struktur, bekannt sind.

4.1.3.3 Delaunay

Die *Delaunay-Triangulierung* ist das bekannteste Verfahren zur Netzbildung und wird in [Mül05] sowie in [Ape] wie folgt beschrieben. Bei der Delaunay-Triangulierung werden die Eckpunkte mittels Kanten unter bestimmten Bedingungen zu Dreiecken verbunden. Wichtig ist vor allem spitze Dreiecke zu vermeiden. Um dies zu erreichen gelten folgende Bedingungen: Es existiert immer eine optimale Triangulierung, die den minimalen

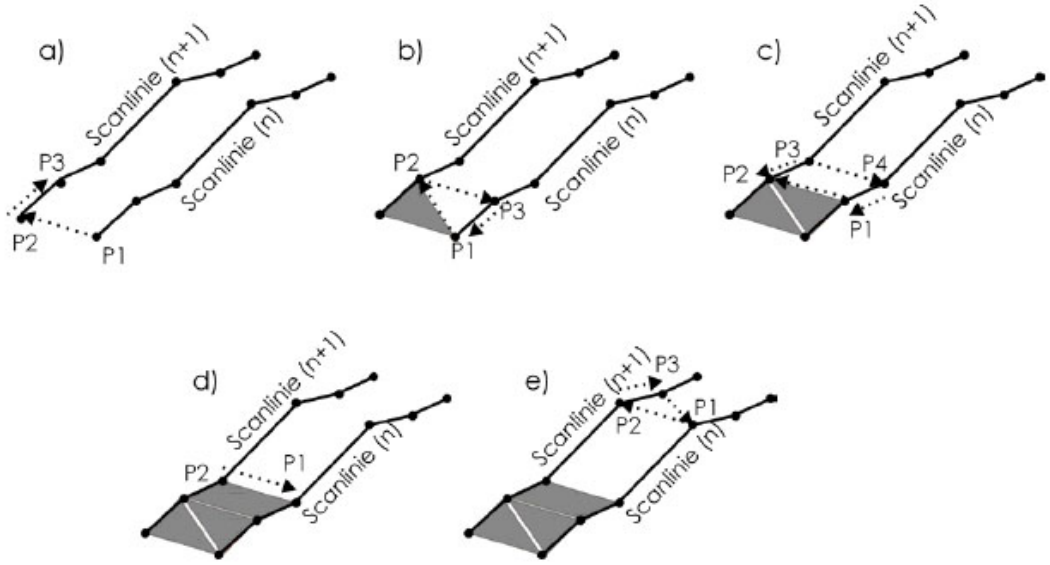


Abbildung 4.7: Prinzip des TOREC-Verfahrens, Quelle:[Pil02]

Winkel maximiert. Anders ausgedrückt: Für jedes Dreieck ist kein weiterer Eckpunkt im Umkreis des Dreiecks vorhanden. Diese Bedingung ist in Abbildung 4.9 dargestellt.

Die Delaunay-Triangulierung ist auf verschiedene Anfangsbedingungen anwendbar. Liegt ein Objekt als Punktewolke vor, so werden die Eckpunkte nach den oben erwähnten Bedingungen zu Dreiecken zusammengeschlossen. Existiert bereits ein Polygonnetz, welches nicht aus Dreiecken besteht, so wird dieses unter der oben genannten Bedingung in Dreiecke unterteilt. Besteht das Polygonnetz bereits ganz oder teilweise aus Dreiecken, so wird das Polygonnetz neu in Dreiecke aufgeteilt. Dies wird oft angewendet, wenn viele spitze Dreiecke sich innerhalb des Polygonnetzes befinden.

Das Vorgehen bei einer vorliegenden Punktwolke zeigt Abbildung 4.8. Links ist die Ausgangssituation, die einzelnen Eckpunkte, dargestellt, rechts das fertige Polygonnetz. Begonnen wird mit zwei Punkten mit dem kürzesten Abstand zueinander. Diese bilden die erste Kante des Dreiecks. In Abbildung 4.8 sind das die Punkte 1 und 2. Die beiden Eckpunkte werden anschließend mit allen umliegenden Nachbarpunkten zu einem Dreieck verbunden und der an der Spitze entstehende Winkel berechnet. Es wird das Dreieck gewählt welches den maximalen Winkel enthält. Es entsteht ein Dreieck mit den Punkten 1, 2 und 3. Dieses ist rot dargestellt. Über die Kanten dieses Dreiecks werden neue Dreiecke mit maximalen Winkeln ermittelt. So bilden die Punkte 1, 3 und 4 und die Punkte 2, 3 und 7 weitere Dreiecke. Diese Vorgehensweise wird so lange fortgeführt bis alle Punkte zu dem Polygonnetz gehören.

Die Delaunay-Triangulierung ist nicht eindeutig falls auf einem Umkreis mehr als drei Punkte liegen. In diesem Fall kann sich der Anwender aussuchen, welche drei Punkte zu

einem Dreieck verbunden werden. Der maximale Winkel ist jedoch immer eindeutig. Die hier beschriebene Vorgehensweise bezieht sich auf 2D. In 3D wird die Triangulierung mit Tetraedern ausgeführt. Hierbei wird der Winkel der Tetraeder-Spitze maximiert.

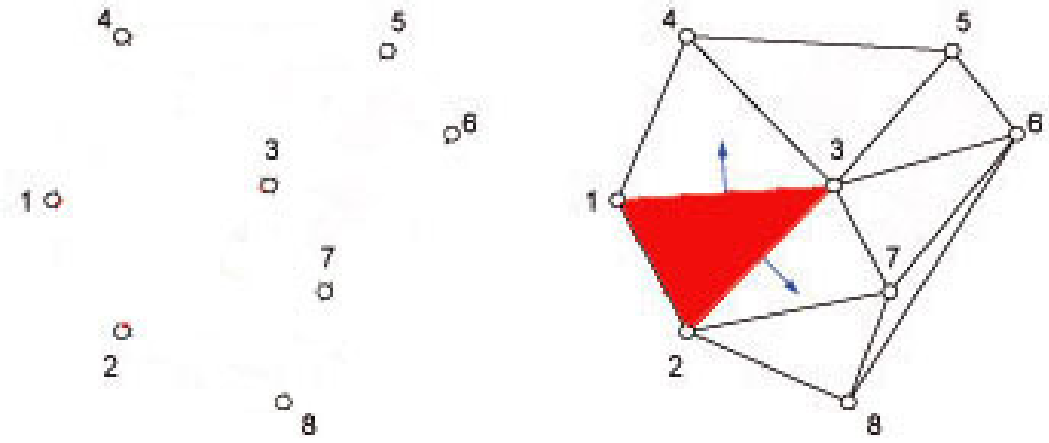


Abbildung 4.8: Delaunay-Triangulierung, Quelle: [Mül05]

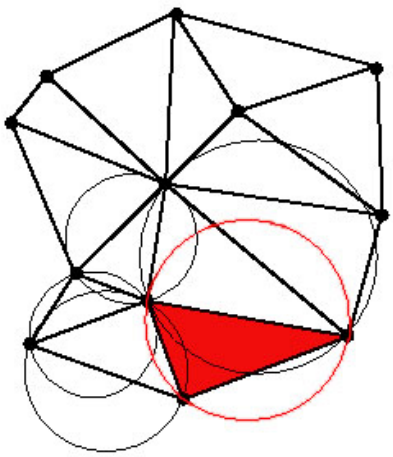


Abbildung 4.9: Kreisbedingung der Delaunay-Triangulierung, Quelle: [Ape]

4.1.4 Vor- und Nachteile

Vorteile der Punktwolke sind ihre schnelle und einfache Darstellung und ihr niedriger Speicherbedarf. Punkte können schnell auf eine Bildfläche projiziert werden. Bei einer Darstellung mit wenig Punkten lässt sich das Objekt noch erkennen und bietet einen geringen Speicherverbrauch.

Ein großer Nachteil der Punktwolke ist die realitätsuntreue Darstellung. Weitere Nachteile ergeben sich bei komplexeren Objekten, vor allem solchen, die viele gekrümmte Oberflächen haben. Um diese Flächen darzustellen, sind an der Stelle der Krümmung viele Punkte erforderlich. Soll ein Objekt sehr genau oder sogar exakt dargestellt werden, sind sehr viele Punkte erforderlich. Gekrümmte Flächen und exakte Darstellung haben einen hohen Speicherbedarf.

4.2 Das Drahtmodell

Das *Drahtmodell* wird auch als Kantenmodell bezeichnet und durch Kanten zwischen Punkten repräsentiert. Nimmt man einen Quader, so sind nur seine zwölf Kanten sichbar, nicht aber die Oberflächen. Geometrische Informationen über die Bereiche zwischen den Kanten sind nicht bekannt. Das Drahtmodell stellt eine *vereinfachte Darstellung* eines 3D-Modells dar. Abbildung 4.10 zeigt ein Beispiel eines Drahtmodells.

Da ein Drahtmodell nur die Umrisse eines Objektes skizziert ergibt sich eine *untreue Abbildung der Wirklichkeit*. Sie ist zwar für die Platzierung von Objekten ausreichend, kann aber zu Mehrdeutigkeiten führen, wenn man das Modell mit Flächen füllt. Dieses Problem ist in Abbildung 4.11 dargestellt. So können aus dem gleichen Drahtmodell verschiedene Objekte erstellt werden.

Zur Speicherung eines Drahtmodells kann die abstrakte Datenstruktur *Kantenliste* verwendet werden. Die Kantenliste besteht aus zwei Listen, die miteinander in Beziehung stehen. In der Kantenliste selbst werden die Kanten gespeichert. Zusätzlich verweist diese auf eine *Punktliste*. In der Punktliste werden die Eckpunkte des Drahtmodells verwaltet.

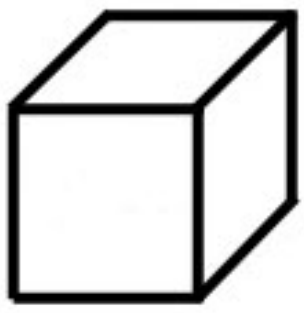


Abbildung 4.10: Drahtmodell

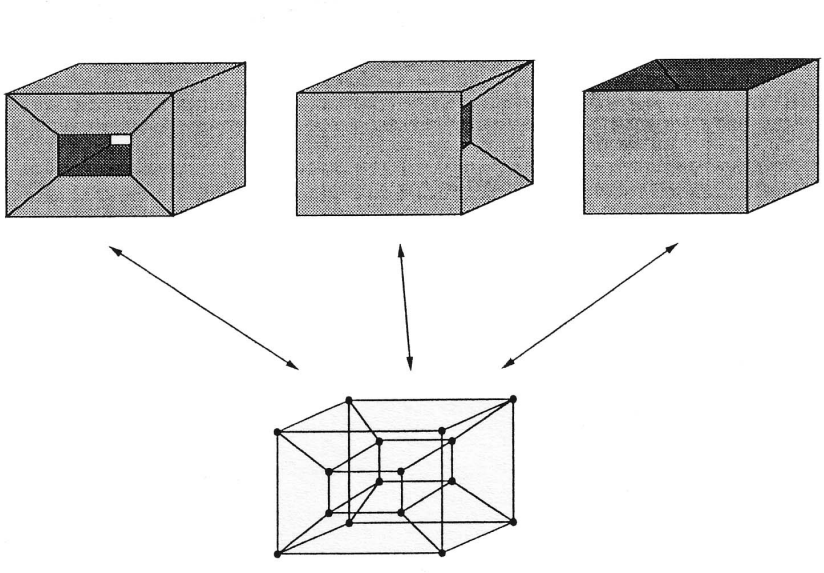


Abbildung 4.11: Mehrdeutigkeiten eines Drahtmodells; Quelle: [Zen99]

4.3 Das Flächenmodell

Beim *Flächenmodell* werden 3D-Modelle durch ihre Oberflächen repräsentiert. Die Flächen des Modells werden als Begrenzungsflächen bezeichnet. Das Modell selbst ist von innen hohl. Es ist bekannt zwischen welchen Kanten sich Flächen befinden und wie diese das Modell begrenzen. Zusätzlich werden *Geometrie- und Topologiedaten* abgespeichert. Zur Modellierung der Oberflächen werden z.B. Polygone verwendet. Abbildung 4.12 zeigt ein Beispiel eines Flächenmodells.

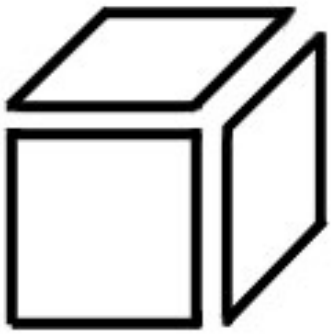


Abbildung 4.12: Flächenmodell

4.4 Das Volumenmodell

Ein *Volumenmodell* wird durch sein Volumen beschrieben. Die Beschreibung des Volumens kann entweder durch volle Körper oder durch Flächen erfolgen. Bei der Beschreibung durch Flächen umschließen diese einen Raum und definieren diesen dadurch.
Volumenmodelle werden häuftig in medizinischen oder technischen Bereichen eingesetzt.

Das Volumenmodell kann auf unterschiedliche Arten entstehen. Eine bekannte Repräsentation wird in diesem Abschnitt vorgestellt: die CSG.

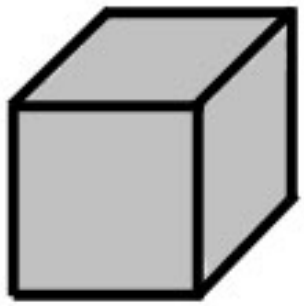


Abbildung 4.13: Volumenmodell

4.4.1 CSG

CSG steht für *Constructive Solid Geometry* und stellt ein Verknüpfungsmodell dar. Nach [Bri05] werden bei dieser Methode Modelle mit Hilfe von *Grundkörpern* und Operationen aus der Mengenlehre modelliert. Es können alle Grundkörper verwendet werden bei denen zwischen „innen“ und „außen“ unterschieden werden kann. Die wichtigsten Grundkörper sind Kugel, Quader, Würfel, Prisma und Zylinder. Dreiecke oder Ebenen sind nicht eindeutig und daher nicht anwendbar.

Die verschiedenen Grundkörper werden anhand von Operatoren der *Booleschen Algebra* miteinander verknüpft. Die Boolesche Algebra ist nach [Bro03] ein Zweig der Algebra (mathematische Theorie der Verknüpfungen), bei der die Elemente einer vorgegebenen Menge durch die Eigenschaften der *logischen Operatoren UND, ODER, NICHT* sowie den Eigenschaften der *mengentheoretischen Verknüpfungen Durchschnitt, Vereinigung und Komplement* verknüpft werden. Sie ist benannt nach George Boole, der sie im 19. Jahrhundert definierte, um algebraische Methoden in der *Aussagenlogik* anwenden zu können. Eine sprachliche oder mathematische *Aussage* ist ein Satz, zu dem der Begriff *wahr* oder *falsch* gehört. Die Boolesche Algebra beschreibt die Regeln mit denen man aus Aussagen neue Aussagen bilden kann.

Einige der Operationen sollen kurz anhand von Beispielen erklärt werden.

Betrachtet man die Grundkörper als Punktwolken, so umschließt die *Vereinigungsoperation* alle Punkte, die innerhalb der beiden ursprünglichen Körper liegen. Sie kann auch mit der Addition ausgeführt werden. Diese Operation ist mit zwei Kugeln in Abbildung 4.14 dargestellt.

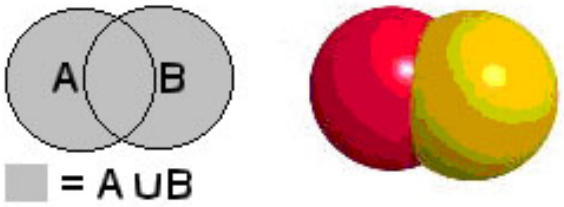


Abbildung 4.14: CSG: Vereinigungsoperation; Quelle: [Loh04]

Der *Durchschnittsoperator* erzeugt die Menge der Punkte, die in beiden Körpern enthalten sind. In Abbildung 4.15 schneiden sich zwei Kugeln, nur deren Schnitt ist bei dieser Operation sichtbar.

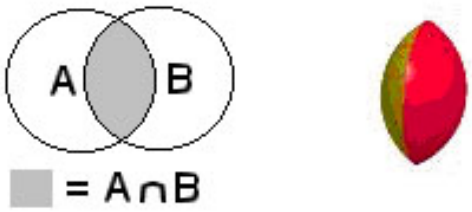


Abbildung 4.15: CSG: Durchschnittsoperation; Quelle: [Loh04]

Das *Komplement* beschreibt die NOT-Operation, d.h. dieses Objekt wird ausgeschlossen. In Abbildung 4.16 ist links die Anwendung dieser Operation auf die Menge A zu sehen. Das Komplement wird häufig im Zusammenhang mit anderen Operationen angewendet.

Der Durchschnitt von A und B ohne B liefert das gleiche Ergebnis wie die *Differenz* von A und B. Die Differenz kann auch mit Hilfe einer Subtraktion realisiert werden. Hier werden alle Punkte im zweiten Körper entfernt, die im ersten enhalten sind. Mit den vorgestellten Operationen sind viele weitere Modelliermöglichkeiten gegeben.

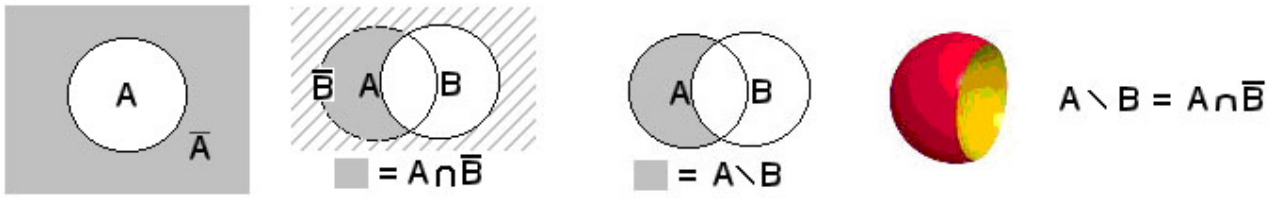


Abbildung 4.16: CSG: Komplement und Differenz; Quelle: [Loh04]

4.4.1.1 Abstrakte Datenstruktur der CSG

Die Modelldarstellung wird in einem *Baum* abgespeichert. In einem Baum hat jeder Knoten eine Folge von Nachfolgern. Dabei spielt die Reihenfolge der Elemente eine wichtige Rolle. Der Baum besteht aus *inneren Knoten*, Knoten die einen Nachfolger besitzen, und *Blättern*, Knoten die keinen Nachfolger besitzen. In den inneren Knoten werden der Typ des Operators, Transformationen und die räumliche Beziehung zwischen den von ihm kombinierten Unterknoten gespeichert. Die Blätter speichern den Typ des Grundkörpers (Primitive), sowie seine Position, Orientierung und Eigenschaften.

Jeder CSG-Baum beschreibt nur ein Objekt. Es ist möglich mehrere CSG-Bäume zu einem neuen Objekt miteinander zu verknüpfen. CSG-Bäume garantieren immer ein gültiges Objekt, seine Repäsentation ist jedoch nicht eindeutig. Ein Objekt kann verschieden dargestellt werden. Abbildung 4.17 zeigt einen CSG-Baum und das resultierende Objekt.

4.4.1.2 Implementierung der CSG

Ein CSG-Baum enthält einmal die Operatoren und einmal die Primitiven. Diese Zusammenhänge können beispielsweise mit Hilfe eines struct in C++ beschrieben werden. Sie werden in [IGD05] wie folgt aufgeführt und verdeutlichen die nötigen Information.

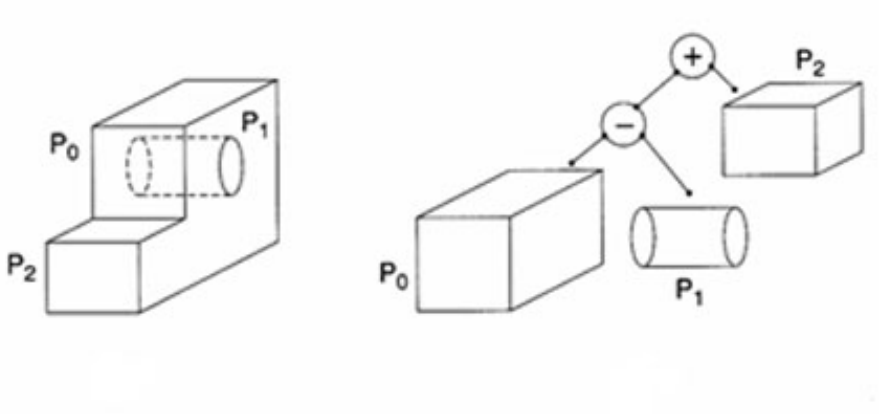


Abbildung 4.17: CSG; links: Modell; rechts: Baumstruktur; Quelle: [IGD05]

```
struct operator {
  int opType; // Vereinigung-, Durchschnitt-, Differenz-Operator
  int leftType; // Typ des linken Knotens: 0=Operator, 1=Primitive
  int rightType; // Typ des rechten Knotens: 0=Operator, 1=Primitive
  void *left; // linker Knoten
  void *right; // rechter Knoten
  void *parent; // Elternknoten
}
```

Wie bereits erwähnt beinhalten die Operatoren den Typ der Operation. Außerdem muss bekannt sein, ob der Nachfolger wieder ein Operator oder eine Primitive ist. Jeder Operator-Knoten verweist außerdem auf die Nachfolgerknoten links und rechts und auf den Elternknoten.

```
struct primitive {
  int primType; // Typ der Primitive
  double xPos, yPos, zPos; // x,y,z Position
  double xOri, yOri, zOri; // x,y,z Orientierung
  void *attribute; // Attribute der Primitive
                   wie Material, Dimension
}
```

Die Primitive enthält Informationen über den Typ der Primitive, ihre Position und Orientierung, sowie weitere Attribute. Da es sich bei Primitiven um Blätter handelt verweisen sie auf keine weiteren Knoten.

```
struct node {
  primitive prim; //Primitive
  operator op; //oder Operator
  node *left, *right; //linker oder rechter Knoten
}
```

Für die einzelnen Knoten wird jeweils entschieden, ob es sich um eine Primitive oder einen Operator handelt. Außerdem enthält jeder Knoten einen Verweis auf den linken und rechten Knoten, sofern dieser vorhanden ist.

4.4.1.3 Vor- und Nachteile von CSG

Die Vorteile von CSG sind ihre einfache Bedienbarkeit innerhalb von CAD-Systemen und die interaktiven und intuitiven Möglichkeiten innerhalb dieser. So können Modelle immer wieder durch Ändern von Operatoren oder Transformationen wie Skalieren, Rotieren und Translatieren verändert werden.

CSG hat aber auch eine Reihe von Nachteilen. So ist die Anzahl der vorhandenen Operationen eingeschränkt, und damit auch die Möglichkeiten der Modellierung. Bei detaillierten Modellen entstehen sehr komplexe logische Ausdrücke. Außerdem wirken sich die Booleschen Operationen global, also auf den gesamten Körper, aus. Lokale Operationen wie die detaillierte Veränderung einer einzelnen Fläche eines komplexen Objekts können nicht mit diesen Operationen umgesetzt werden.

Ein weiterer Nachteil wird beim *Rendern* sichtbar. Das Rendern beschreibt ein Verfahren zur realitätsnahen Darstellung dreidimensionaler Objekte durch Farb- und Lichteffekte. Hier sind *spezielle Renderingtechniken* oder die Umwandlung in ein *Polygonnetz* erforderlich, da die CSG-Datenstruktur keine geometrischen Informationen beinhaltet. Sie speichert lediglich die verwendeten Grundkörper und ihre Verknüpfungen. Dadurch ist die Darstellung sehr zeit- und rechenintensiv.

5 Das Flächenmodell

Das Flächenmodell ist das in der Computergraphik am häufigsten verwendete Modell. In diesem Kapitel wird diese Art des Modells im Zusammenhang mit den zu Anfang dieser Arbeit vorgestellten Stufen untersucht. Zu Beginn wird auf den Aufbau dieser Repräsentationsart eingegangen. Hier werden die Boundary Representation und Polygonnetze erläutert. Um die Beziehungen des Flächenmodells zu verdeutlichen werden diese im Zusammenhang mit einem Klassendiagramm und dem vef-Graphen noch einmal aufgegriffen. Da für den Aufbau eines Flächenmodells und die darauf ausführbaren Operationen Geometrie und Topologie eine große Rolle spielen wird diesem Thema ein eigener Abschnitt gewidmet. Dieser dient gleichzeitig als Grundlage für die darauf folgenden Abschnitte und die vorgestellten Operationen. Anschließend folgen die Stufen abstrakte Datenstruktur, innerhalb dieser werden Operationen auf Polygonnetzen eingeführt, und Speicherungsarten, hier wird Bezug genommen auf die Möglichkeiten der Speicherung eines Polygonnetzes. Zum Schluss wird der Aufwand der Operationen innerhalb zwei verschiedener Speicherungsarten untersucht. Ein abschließendes Fazit fasst jeden Abschnitt zusammen und stellt noch einmal die wichtigsten Erkenntnisse heraus.

5.1 Repräsentationsform

Die *Boundary Representation* (BReps) ist eine der am häufigsten verwendeten Techniken zum geometrischen Modellieren eines Modells über seine Oberfläche. Das Volumen des Modells wird dabei durch seine *Hülle* beschrieben und ist innen hohl. Da diese Beschreibung zunächst nicht eindeutig ist wird mit Hilfe von *Normalen* „innen“ und „außen“ bestimmt. Üblich ist dabei die Definition: wo die Normale hinzeigt ist „außen“.

Wichtig für die Darstellung des Modells sind nicht nur geometrische, sondern auch topologische Informationen. Erst sie bilden aus den einzelnen Bestandteilen das Modell, indem sie die Zusammenhänge der Bestandteile beschreiben.

Es gibt zwei Vertreter der Boundary Representation:

- die Oberflächenbeschreibung durch *Polygone*
- und die Oberflächenbeschreibung durch *Parameter-Oberflächen*

Im Folgenden wird auf Polygone eingegangen. Parameter-Oberflächen werden aus Gründen des Umfangs in dieser Arbeit nicht behandelt.

5.2 Das Polygonnetz

Polygone stellen eine bekannte Art der Darstellung eines Modells über seine Oberflächen dar. Die Oberflächen des Modells werden durch Flächen mit bestimmten Eigenschaften beschrieben, diese werden Polygone genannt. Mehrere Polygone bilden ein *Polygonnetz*. Die Definition eines Polygonnetzes ist die Repräsentationsform eines Flächenmodells.

Im Folgenden wird zunächst eine in der Computergraphik verwendete Definition des Begriffs Polygonnetz gegeben. Sie ist in [Bri05], sowie in [Lan] zu finden. Danach erfolgt noch einmal die Beschreibung der Bestandteile des Polygonnetzes und ihrer Beziehungen mit Hilfe eines Klassendiagramms und in diesem Zusammenhang die Einführung des vef-Graphen.

5.2.1 Definition eines Polygonnetzes

Für eine Menge von Punkten V_0,...,V_n wird Q = (V_0,V_1),(V_1,V_2),...,(V_{n-2},V_{n-1}) *Polygonzug* genannt. (V_n,V_m) beschreiben jeweils die Verbindungsstrecke zwischen zwei Punkten. Die Verbindungsstrecken werden als *Kanten* und die Punkte als *Eckpunkte* des Polygonzuges bezeichnet.

Ein *Eckpunkt* wird als ein nulldimensionales Objekt verstanden, welches keine Ausdehnung hat. Er ist ein Element eines zwei- oder dreidimensionalen Raumes und wird über seine Koordinaten x und y bzw. x, y und z beschrieben.

Stimmt der erste Eckpunkt mit dem letzten Eckpunkt überein, ist also $V_{n-1} = V_0$, so handelt es sich um einen *geschlossenen Polygonzug*. Ein geschlossener Polygonzug besteht aus mindestens drei Eckpunkten und mindestens drei Kanten und bildet im einfachsten Fall ein Dreieck.

Stimmt der erste Eckpunkt nicht mit dem Letzten überein spricht man von einem *offenen Polygonzug*. Für einen offenen Polygonzug sind mindestens zwei Kanten und drei Eckpunkte erforderlich.

Wird ein geschlossener Polygonzug mit einer Fläche ausgefüllt, so wird dieser als Polygon bezeichnet. Ein Polygon wird *einfach* genannt, falls sich keine zwei Kanten schneiden und jeder Endpunkt einer Kante höchstens zu zwei Kanten des Polygons gehört. Treffen die beschriebenen Eigenschaften nicht zu wird das Polygon *nicht-einfach* genannt. Abbildung 5.1 stellt ein einfaches und ein nicht-einfaches Polygon dar.

Ein Polygon wird außerdem durch weitere Eigenschaften beschrieben. Alle Eckpunkte und Kanten des Polygons liegen innerhalb einer Ebene, das Polygon wird als *planar* bezeichnet. Das Polygon kann sowohl *konvex* als auch *konkav* sein. Konvex bedeutet, dass alle Punkte auf einer Linie zwischen zwei beliebigen Eckpunkten sich innerhalb des Polygons oder auf seinen Kanten befinden. Bei Konkav trifft dies nicht zu. Dreiecke sind stets planar und konvex.

Abbildung 5.1: Links: einfaches Polygon, rechts: nicht-einfaches Polygon; Quelle:[Bri05]

Die Eckpunkte eines Polygons werden gegen den Uhrzeigersinn definiert. Diese Forderung resultiert aus dem verwendeten *Rechtssystem* und der *Orientierung* des Polygons. Die Orientierung des Polygons beschreibt das Außen und Innen eines Polygons anhand der Normale. Die *Normale* ist ein Vektor der senkrecht auf einer Ebene steht. Die Richtung der Normale bestimmt die Außenseite des Polygons. Dieses Wissen ist für die Verwendung von Polygonen innerhalb eines Polygonnetzes notwendig. Nur so kann festgestellt werden welche Seite des Polygons sich innen und welche Seite sich außen im Bezug auf das Polygonnetz befindet.

Bereits ein einfaches konvexes Polygon beschreibt ein *Polygonnetz*. Mehrere zusammenhängende einfache konvexe Polygone zeigt Abbildung 5.2.

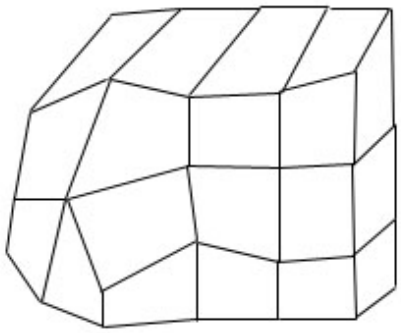


Abbildung 5.2: Ausschnitt eines Polygonnetzes

Ein Polygonnetz erfüllt mehrere Anforderungen. So haben zwei benachbarte Polygone eine Kante gemeinsam. Und jede Kante eines Polygons gehört zu einem oder höchstens zwei Polygonen. Gehört die Kante nur zu einem Polygon so handelt es sich um ein Randpolygon.

5.2.2 Aufbau und Beziehungen eines Polygonnetzes

In diesem Abschnitt wird noch einmal auf das Polygonnetz und seine Bestandteile eingegangen. Es werden noch einmal kurz die Bestandteile des Polygonnetzes vorgestellt und die gegenseitigen Beziehungen herausgestellt. Dies erfolgt anhand zwei verschiedener Sichtweisen: *dem Klassendiagramm und dem vef-Graphen.*

Das Polygonnetz wird durch seine Bestandteile Flächen (face), Kanten (edge) und Eckpunkte (vertex) beschrieben. Die Beschreibung enthält geometrische und topologische Informationen. Die Geometrie beschreibt die Lage der Bestandteile und die Topologie die Beziehungen zwischen den Bestandteilen. Beide Begriffe werden ausführlich in Abschnitt 5.2.3 behandelt.

- Die *Fläche* ist ein Teil des Polygonnetzes. Dieses beschreibt die Oberfläche des Objektes. Die Flächen werden durch Kanten und Eckpunkte begrenzt.
- Die *Kante* ist eine Strecke, die durch zwei Eckpunkte begrenzt wird. Sie zeigt außerdem die Grenzen zwischen den Flächen auf.
- Der *Eckpunkt* ist ein Punkt im 2D- oder 3D-Raum und begrenzt Kanten.

Laut der Definition aus Abschnitt 5.2.1 beschreibt ein Polygon eine umrandete Fläche und wird als Face bezeichnet, falls es ein Teil eines Objektes, z.B. eines Würfels, ist. Die Modell-Oberfläche wird durch das Polygonnetz beschrieben, wobei dieses im Normalfall aus mehreren Polygonen besteht. Dabei wird das Modell nur approximiert, also angenähert. Je feiner das Polygonnetz ist, desto genauer wird die Modell-Oberfläche dargestellt.

Das Polygonnetz und seine Bestandteile können wie bereits erwähnt auf verschiedene Weisen dargestellt werden. Abbildung 5.3 stellt die Zusammenhänge des Polygonnetzes in einem Klassendiagramm dar.

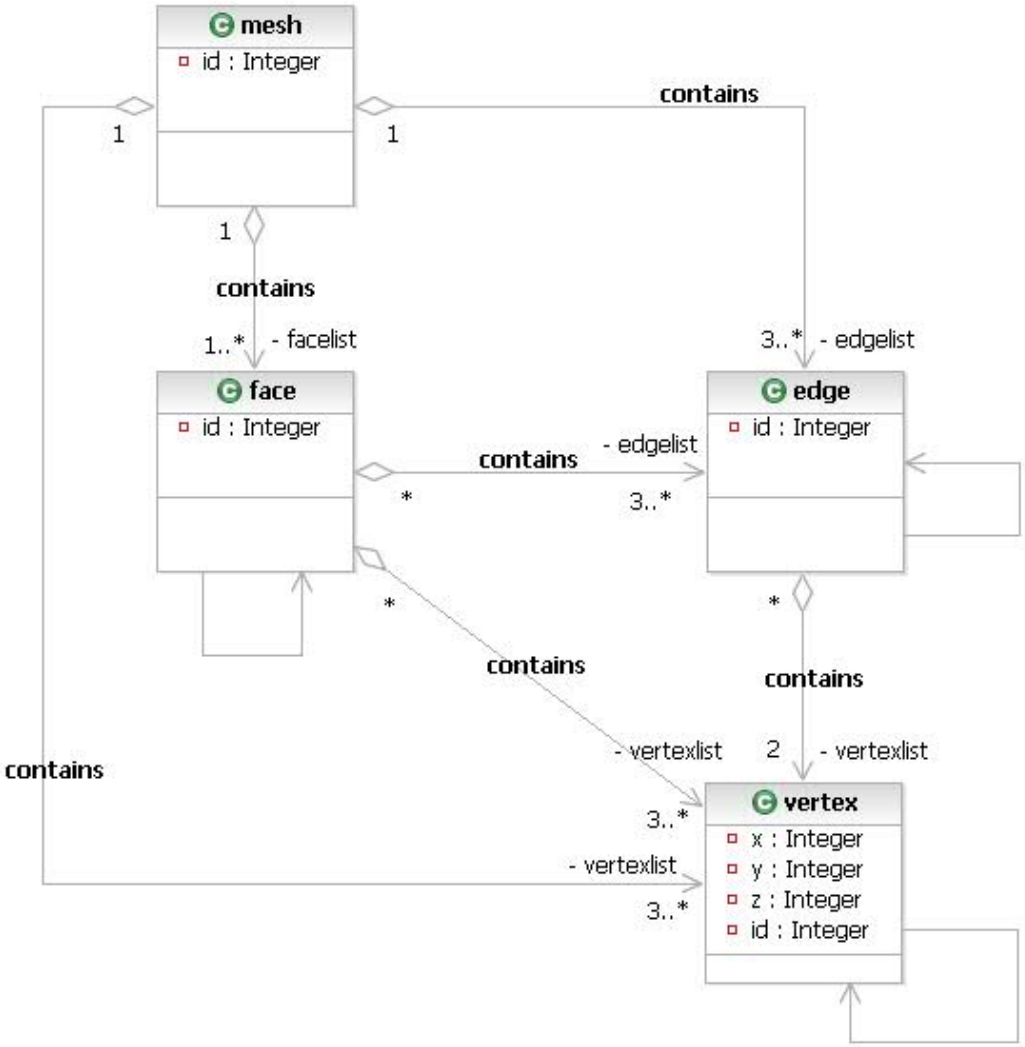


Abbildung 5.3: Klassendiagramm eines Polygonnetzes

Ein Polygonnetz besteht demnach aus mindestens einer Face. Eine Face besteht mindestens aus drei Eckpunkten und mindestens drei Kanten und bildet im einfachsten Fall ein Dreieck. Eine Kante hat genau zwei Eckpunkte, von denen sie begrenzt wird, und begrenzt mindestens eine und höchstens zwei Faces. Ein Eckpunkt wird durch seine Koordinaten beschrieben.

Ein Polygonnetz kann durch Faces, Kanten oder Eckpunkte beschrieben werden. Eine Face kann durch Kanten oder Eckpunkte beschrieben werden. Eine Kante wird durch ihre Endpunkte beschrieben. Außerdem ist es möglich die Eckpunkte einer Face über die Kanten, und die Eckpunkte eines Polygonnetzes über Kanten oder Faces zu ermitteln.

Diese Zusammenhänge können auch auf eine andere Weise beschrieben werden. Nachfolgend wird der vef-Graph vorgestellt. Er bezieht sich auf die gleichen Bestandteile eines Polygonnetzes und zeigt die bereits vorgestellten Beziehungen auf.

Beide Sichtweisen sind wichtig um zwischen dem Aufbau eines Polygonnetzes und den Beziehungen zwischen den Bestandteilen zu unterscheiden. Aus den vorhandenen Bestandteilen ergeben sich unterschiedliche Beziehungen zwischen ihnen. Diese können durch beide Sichtweisen beschrieben werden.

Der *vef-Graph* wird nach [Zen99] im Gegensatz zum Klassendiagramm zur Beschreibung rein topologischer Strukturen verwendet. Es werden nur die Beziehungen innerhalb des Polygonnetzes betrachtet. Das Klassendiagramm beinhaltet auch geometrische Informationen, indem es Koordinaten der Eckpunkte speichert. Dadurch wird die Lage des Polygonnetzes mitbetrachtet. Zusätzlich wird definiert welcher Aufbau eines Polygonnetzes möglich ist. Der vef-Graph besteht aus den Eckpunkten V, den Kanten E, den Flächen F und den Adjazenzrelationen R. Die geometrische Information wird bei dieser Darstellung ignoriert. Beim vef-Graphen sind alle Eckpunkte, Kanten und Flächen Knoten des Graphen und alle Kanten des Graphen sind verschiedene Adjazenzrelationen. Adjazenzrelationen beschreiben die Nachbarschaftsinformationen zwischen den Bestandteilen des Polygonnetzes. V beschreibt eine Menge von Eckpunkten, E eine Menge von Kanten, F eine Menge von Flächen und R die möglichen Relationen, also Beziehungen zwischen V, E und F.

Der vef-Graph wird wie folgt beschrieben:

$$G = (V, E, F; R)$$
$$V := v_1, ..., v_i$$
$$E := e_1, ..., e_j$$
$$F := f_1, ..., f_k$$
$$R := Adjazenzrelationen$$

Zwischen den Bestandteilen des Polygonnetzes können mit Hilfe des Graphen verschiedene Relationen hergestellt werden, die die Nachbarschaftsbeziehungen innerhalb eines Polygonnetzes beschreiben. Insgesamt sind 9 verschiedene Relationen möglich. Diese werden in Abbildung 5.4 dargestellt.

Im Gegesatz zum Klassendiagramm wird das Polygonnetz selbst nicht in der Darstellung verwendet. Die Beziehungen zwischen den Bestandteilen sind jedoch innerhalb beider Sichtweisen gleich. So entsprechen die Kanten des vef-Graphen den Assoziationen des Klassendiagramms und die Knoten den Klassen. Die im vef-Graphen beschriebenen Relationen stellen topologische Beziehungen dar und finden sich auch im Klassendiagramm wieder.

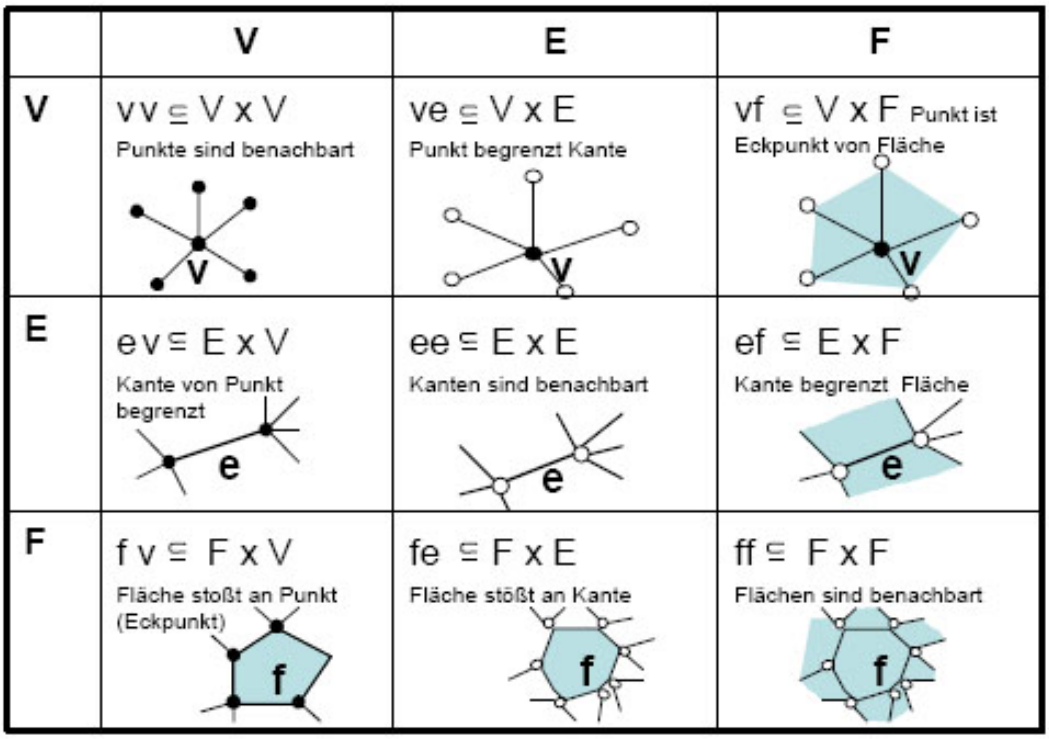


Abbildung 5.4: Mögliche Relationen, Quelle:[Nau]

Die Relationen des vef-Graphen werden kurz einzeln erläutert und in Bezug zum Klassendiagramm gesetzt. Als Beispiel-Objekt dient ein Tetraeder. Er ist in Abbildung 5.5 dargestellt. Für die Tetraeder-Topologie gilt n_V=4, n_E=6 und n_F=4. Er besteht demnach aus 4 Eckpunkten, 6 Kanten und 4 Flächen.

Relationen zwischen gleichen Bestandteilen sind in Abbildung 5.4 diagonal dargestellt.

Diese sind vv, ee und ff. Sie lassen sich im Klassendiagramm nur indirekt ablesen, sind aber dennoch vorhanden.

Die Relation vv beschreibt die Nachbarschaft zwischen Eckpunkten. Abbildung 5.5 zeigt den vef-Graphen für die Relation vv eines Tetraeders. v_1 ist mit v_2, v_3 und v_4 benachbart. v_2 ist Nachbar von v_3, v_4 und v_1. v_3 ist benachbart mit v_1, v_2 und v_4, und v_4 ist der Nachbar von v_1, v_2 und v_3. In einem Tetraeder sind alle Eckpunkte miteinander benachbart.

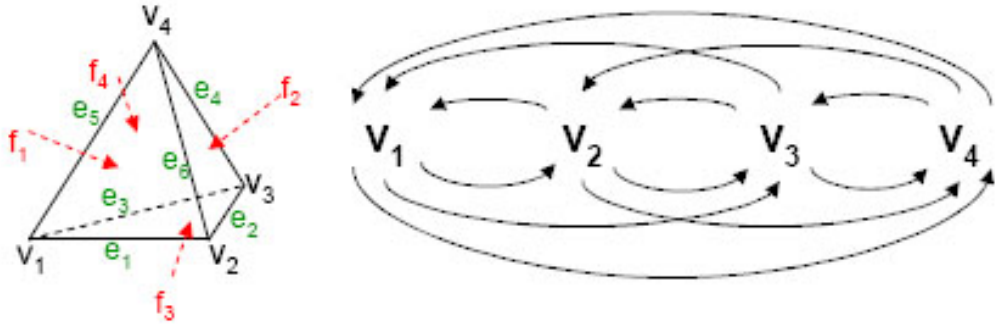


Abbildung 5.5: Tetraeder: vv-Relation, Quelle:[Nau]

Die Relation ee beschreibt die Nachbarschaft zwischen zwei Kanten. Folgende Relationenpaare sind gültig: (e_1,e_2), (e_1,e_5), (e_1,e_3), (e_1,e_6), (e_2,e_1), (e_2,e_6), (e_2,e_3), (e_2,e_4), (e_3,e_2), (e_3,e_1), (e_3,e_4), (e_3,e_5), (e_4,e_6), (e_4,e_2), (e_4,e_3), (e_4,e_5), (e_5,e_1), (e_5,e_3), (e_5,e_6), (e_5,e_4), (e_6,e_2), (e_6,e_4), (e_6,e_5), (e_6,e_1). Als Teilmenge dieser Relation existiert ee'. In diesem Fall sind die Kanten benachbart und begrenzen dieselbe Fläche. Im Tetraeder sind dies die gleichen Relationen wie bei ee.

Die Relation ff sagt aus, dass beide Flächen benachbart sind. Im Tetraeder sind dies gültige ff-Relationen: (f_1,f_2), (f_1,f_3), (f_1,f_4), (f_2,f_3), (f_2,f_1), (f_2,f_4), (f_3,f_1), (f_3,f_2), (f_3,f_4), (f_4,f_1), (f_4,f_2), (f_4,f_3).

Weitere Relationen sind ve, ev, vf, fv, ef und fe. Je zwei dieser Relationen bilden jeweils die Relation und die dazugehörige Umkehrrelation. Die Relation ve gibt an, dass ein Eckpunkt eine Kante begrenzt. Im Tetraeder begrenzt der Eckpunkt v_1 die Kanten e_1, e_3 und e_5, der Eckpunkt v_2 die Kanten e_1, e_6 und e_2, der Eckpunkt v_3 die Kanten e_2, e_3 und e_4 und der Eckpunkt v_4 die Kanten e_4, e_5 und e_6.

In der Relation ev wird die Kante e von einem Eckpunkt v begrenzt. So wird die Kante e_1 von v_1 begrenzt, aber auch von v_2. In diesem Fall sind die Relationen (e_1,v_1) und (e_1,v_2) möglich. Weitere Relationen sind: (e_2,v_3), (e_2,v_2), (e_3,v_1), (e_3,v_3), (e_4,v_3), (e_4,v_4), (e_5,v_1), (e_5,v_4), (e_6,v_2), (e_6,v_4).

Die Relationen ve und ev beschreiben im Klassendiagramm die Beziehung zwischen Kanten und Eckpunkten. So werden die Kanten von zwei Eckpunkten begrenzt und ein Eckpunkt begrenzt eine oder mehrere Kanten.

Die Relation vf sagt aus, dass der Punkt ein Eckpunkt der Fläche ist. So ist im Tetraeder der Eckpunkt v_1 ein Eckpunkt der Flächen f_1, f_3 und f_4, der Eckpunkt v_2 ein Eckpunkt der Flächen f_1, f_2 und f_3, der Eckpunkt v_3 ein Eckpunkt der Flächen f_2, f_3 und f_4 und der Eckpunkt v_4 ein Eckpunkt der Flächen f_1, f_2 und f_4.

Die Relation fv beschreibt den Zusammenhang zwischen einer Fläche und einem Eckpunkt. Hierbei stößt die Fläche an einen Eckpunkt. Gültige Relationen sind: (f_1,v_1), (f_1,v_2), (f_1,v_4), (f_2,v_2), (f_2,v_3), (f_2,v_4), (f_3,v_1), (f_3,v_2), (f_3,v_3), (f_4,v_3), (f_4,v_1), (f_4,v_4).

Die Relationen vf und fv entsprechen den Beziehungen zwischen Faces und Eckpunkten im Klassendiagramm. Faces werden durch Eckpunkte begrenzt und Eckpunkte gehören zu Faces.

Die Relation ef definiert, dass die Kante e die Fläche f begrenzt. Abbildung 5.6 zeigt den Graphen der Relation ef. Gültige Paare der Relation ef sind: (e_6,f_1), (e_6,f_2), (e_1,f_1), (e_1,f_3), (e_5,f_1), (e_5,f_4), (e_2,f_2), (e_2,f_3), (e_4,f_2), (e_4,f_4), (e_3,f_3), (e_3,f_4).

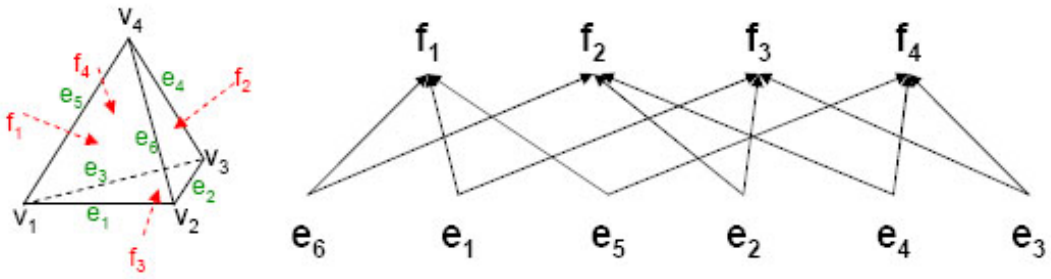


Abbildung 5.6: Tetraeder: ef-Relation, Quelle:[Nau]

Die Relation fe beschreibt die Nachbarschaftsbeziehungen zwischen einer Fläche und einer Kante. Die Fläche f stößt an die Kante e. Folgende Relationen sind innerhalb des Tetraeders gültig: (f_1,e_1), (f_1,e_5), (f_1,e_6), (f_2,e_2), (f_2,e_4), (f_2,e_6), (f_3,e_1), (f_3,e_2), (f_3,e_3), (f_4,e_3), (f_4,e_4), (f_4,e_5).

Auch die Relationen ef und fe können in das Klassendiagramm übertragen werden. Sie beziehen sich auf die Beziehungen zwischen Faces und Kanten.

Da es sehr speicherintensiv ist, alle Relationen abzuspeichern und außerdem Redundanzen vorhanden sind, ist es sinnvoll, nur einen Teil dieser Relationen abzuspeichern.

Bei der Wahl der zu speichernden Relationen muss überlegt werden welche Relationen schnell und günstig aus diesen errechnet werden können und welche somit abgespeichert werden müssen. Je nach der zu implementierenden Struktur wird eine passende Auswahl getroffen.

Der Speicheraufwand für die zu speichernden Paare ist von großer Bedeutung. Er wird daher näher betrachtet. Beim Betrachten der einzelnen Relationen stellt man schnell fest, dass die Angabe des Speicheraufwandes in Abhängigkeit von den Flächen und den Eckpunkten nicht direkt möglich ist. Die zu speichernde Anzahl an Paaren kann nur in einem konkreten Fall angegeben werden, denn die Anzahl der Nachbarflächen ist von der Form der Flächen und die Anzahl der Nachbarpunkte von der Anzahl angrenzender Flächen abhängig. Die Angabe des Speicheraufwandes in Abhängigkeit von den Kanten ist dagegen leicht. Aufgrund der Struktur einer Kante sind die Abhängigkeiten immer gleich. So führt jede Kante genau zu zwei Eckpunkten. Jede Kante wird genau von zwei Eckpunkten begrenzt. Jede Kante begrenzt genau zwei Flächen. Jede Kante dient genau zwei Flächen als Begrenzung.

Die Anzahl von auftretenden Paaren pro Kante e kann in jeder Relation vorherbestimmt werden. Der Speicheraufwand hat immer zwei Einheiten. Abbildung 5.7 zeigt die in Abhängigkeit von e entstehenden Speicheraufwände auf. So hat beispielsweise vv 2 Einheiten, da jede Kante zu genau zwei Paaren führt, (v_i,v_j) und (v_j,v_i), und bei ve wird jede Kante von genau zwei Punkten, v_i und v_j, begrenzt. Die Speicherung aller von e abhängigen Relationen verursacht einen Speicheraufwand von 20 Einheiten. Da auch Relationen in denen auf den ersten Blick keine Kante vorhanden ist Kanten enthalten, können für alle Relationen die Einheiten des Speicheraufwandes angegeben werden.

Relation	*vv*	*ve*	*vf*	*ev*	*ee'*	*ef*	*fv*	*fe*	*ff*
Komplexität	$2n_e$	$2n_e$	$2n_e$	$2n_e$	$4n_e$	$2n_e$	$2n_e$	$2n_e$	$2n_e$

Abbildung 5.7: Speicheraufwand abhängig von n_E der Kanten, Quelle:[Nau]

Da innerhalb der Relationen Redundanzen vorhanden sind, müssen nicht alle Relationen explizit gespeichert werden. Durch die vorhandenen Redundanzen lassen sich bestimmte Relationen durch Kombination anderer Relationen ausdrücken. Da der Speicheraufwand von Kanten bekannt ist, ist es sinnvoll über e zu verknüpfen. So kann der Aufwand auch für die restlichen Relationen angegeben werden.

Die Relation vf kann über die beiden Relationen ve und ef ermittelt werden. fv wird über fe und ev verknüpft, vv entsteht aus ve und ev, ff durch fe und ef und ee kann in die Relationen ev und ve oder ef und fe aufgebrochen werden. Die über e verknüpften Relationen vf, fv, vv, ff und ee lassen sich also aus ve, ev, fe und ef berechnen. Für

Verknüpfungen über v oder f gilt dies im Allgemeinen nicht. So setzt z.B. fv verknüpft mit ve im Gegensatz zu fe auch Flächen mit Kanten in Beziehung, die fv nicht begrenzt.

5.2.3 Die Begriffe Geometrie und Topologie

Damit aus den Bestandteilen ein Modell gebildet werden kann, wird neben der geometrischen auch die topologische Information benötigt. Erst beide Informationen beschreiben ein Modell vollständig. Die Begriffe *Geometrie* und *Topologie* bezogen auf ein Polygonnetz sind nach [Zen99] wie folgt definiert.

Unter dem Begriff Geometrie versteht man die Lage der Bestandteile eines Polygonnetzes Eckpunkt, Kante und Face und ihre Gestalt im Koordinatensystem. Dazu zählen auch die Form der Einzelflächen und die Lage der Eckpunkte, also ihre Koordinaten. Auch die Form des Polygonnetzes gehört zur Geometrie.

Die Topologie beschreibt das Beziehungsgeflecht aus Eckpunkten, Kanten und Faces. Sie beschreibt die gegenseitige Lage und Anordnung innerhalb des Polygonnetzes, also deren Nachbarschaftsbeziehungen.

Verschiedene Objekte mit unterschiedlicher Geometrie können dieselben topologischen Informationen besitzen. Denn die Topologie beschreibt nur die Zusammenhänge der Bestandteile, nicht ihre Lage. Andererseits existieren Objekte die weder eine gemeinsame Geometrie noch eine gemeinsame Topologie besitzen. Abbildung 5.8 zeigt zwei Objekte mit unterschiedlicher Geometrie und unterschiedlicher Topologie. In Abbildung 5.9 haben beide Objekte eine unterschiedliche Geometrie, aber die gleiche Topologie. Objekte, die die gleiche Geometrie, aber eine unterschiedliche Topologie besitzen existieren dagegen nicht. Es ist nicht möglich die Nachbarschaftsbeziehungen zu verändern ohne auch die Geometrie zu ändern. Sobald die Beschreibung der Nachbarschaften sich durch neue oder veränderte Bestandteile ändert, ändert sich auch die geometrische Beschreibung, da sich entweder Form des Polygonnetzes, die Lage der Eckpunkte oder die Beschreibung durch die zusätzlichen oder entfallenden Bestandteile ändert. Daher existieren auch keine Operationen mit solcher Auswirkung.

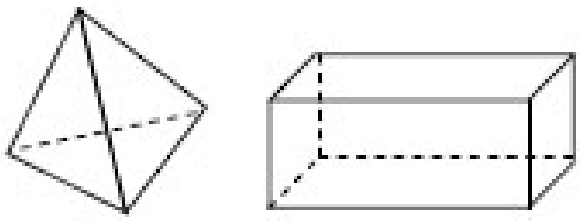


Abbildung 5.8: Verschiedene Topologie und Geometrie, Quelle:[Nau]

Es kann daher zwischen drei möglichen Kombinationen dieser Informationen unterschieden werden.

- verschiedene Geometrie und Topologie
- gleiche Geometrie und Topologie
- verschiedene Geometrie und gleiche Topologie

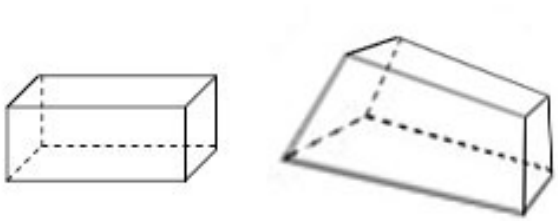


Abbildung 5.9: Gleiche Topologie, verschiedene Geometrie, Quelle:[Nau]

Demnach verhalten sich zwei Objekte mit den vier Kombinationen wie folgt:
Objekte mit unterschiedlicher Geometrie und unterschiedlicher Topologie haben eine unterschiedliche Form und unterschiedliche Nachbarschaftsbeziehungen. Objekte mit gleicher Geometrie und Topologie entsprechen sich in ihren geometrischen und topologischen Informationen, sind also gleich. Objekte mit unterschiedlicher Geometrie und gleicher Topologie besitzen zwar eine unterschiedliche Form, werden aber durch die gleichen Nachbarschaftsinformationen beschrieben.

Da ein Objekt sich stets aus geometrischen und topologischen Informationen zusammensetzt müssen diese auch bei der Ausführung von Operationen auf diesen beachtet werden. Die anschließend betrachteten Strukturen zur Speicherung der Daten eines Objektes beinhalten sowohl geometrische Informationen indem sie die Koordinaten der Eckpunkte abspeichern, als auch topologische Informationen indem sie auf andere Bestandteile verweisen. Die Unterscheidung zwischen diesen beiden Arten der Information erleichtert den Aufbau einer Datenstruktur und ermöglicht einen effizienten und schnellen Zugriff auf die Informationen. Gleichzeitig vereint die Struktur die Geometrie und Topologie.

Um die Wirkung dieser vier Kombinationen sichtbar zu machen werden die Operationen nach diesen unterteilt. Bei genauerer Betrachtung der Operationen wird deutlich, dass sie sich zusätzlich in raumbezogene und objektbezogene Operationen unterteilen lassen. Raumbezogen sind all die Operationen, die die Geometrie beeinflussen, objektbezogen alle Operationen die die Beziehungen innerhalb des Objektes verändern, also die Topologie.

Die verschiedenen Möglichkeiten werden nachfolgend kurz erläutert und im Hinblick auf die Operationen auf einem Polygonnetz betrachtet. Die dazugehörigen Operationen werden im nächsten Abschnitt behandelt.

Geometrie und Topologie verändernde Operationen

Geometrische Änderungen führen zu Formveränderungen, Lageveränderungen der Eckpunkte oder veränderter Beschreibung des Polygonnetzes durch neue Bestandteile. Die geometrischen Veränderungen haben eine Auswirkung auf die topologischen Informationen. Diese werden durch die neuen geometrischen Bedingungen verändert. Beispielsweise führt das Einfügen einer Kante innerhalb einer Face zu zwei Faces, also neuer Form der alten Face und neuen Nachbarschaftsbeziehungen zwischen den neuen Faces und den umliegenden Faces. Die hier zugehörigen Operationen ändern Geometrie und Topologie.

Geometrie verändernde Operationen

Operationen, die die Geometrie verändern, die Topologie jedoch unverändert belassen, verändern die Form der Flächen, des Polygonnetzes oder die Lage der Eckpunkte. Ihre topologischen Beziehungen bleiben jedoch erhalten. Ein Beispiel hierfür ist das Verschieben eines Außenpunktes. Dies verändert die Gestalt des Netzes, jedoch nicht die Beziehungen der Bestandteile um diesen Eckpunkt herum.

Geometrie und Topologie nicht verändernde Operationen

Bei gleicher Geometrie und Topologie bleiben die Informationen unverändert erhalten. Entsprechende Operationen stellen reine Abfragen von Informationen dar und ändern weder die Geometrie noch die Topologie. Auch das Polygonnetz selbst bleibt unverändert. Sie werden verwendet um Zusammenhänge innerhalb eines Polygonnetzes zu ermitteln.

5.3 Abstrakte Datenstrukturen für Polygonnetze

In diesem Abschnitt werden mögliche Operationen auf Polygonnetzen vorgestellt. Als Anregung diente die Vorlesung [Mül05] sowie die Arbeit von [Akl06]. Operationen gehören zur Stufe Abstrakte Datenstruktur. Sie legt fest wie Daten verwaltet und manipuliert werden. Die Operationen gelten unabhängig vom späteren spezifischen Aufbau der Struktur. Dieser wird im Abschnitt Speicherungsarten behandelt. Am Schluß fasst ein kurzes Fazit die wichtigsten Aussagen dieses Abschnitts noch einmal zusammen.

5.3.1 Operationen auf Polygonnetzen

Die Herleitung der Operationen erfolgt aus den in Abschnitt 5.2.3 erarbeiteten Zusammenhängen. So werden diese ebenfalls in die dort vorgestellten Gruppen eingeordnet. Es wird demnach unterschieden zwischen Operationen die Geometrie und Topologie verändern, die nur die Geometrie verändern und die Geometrie und Topologie unverändert lassen. Wie bereits erwähnt ist es nicht möglich nur die Topologie zu verändern.

Diese Unterscheidung verdeutlicht welche Auswirkungen die Operationen auf das Polygonnetz haben. So wird deutlich, ob sich beispielsweise nur die Lage des Netzes verändert, oder ob auch das Polygonnetz selbst verändert wird. Die Operationen werden jeweils mit ihrer Signatur und natürlichsprachlicher Spezifikation vorgestellt. Zusätzlich soll ein Beispiel (falls möglich) sie verdeutlichen.

Bei den hier vorgestellten Operationen werden planare, konvexe und einfache Polygone innerhalb des Polygonnetzes betrachtet. Das Ausführen einer Operation muss stets zu diesen Bedingungen führen, andernfalls ist die Operation nicht ausführbar. Innerhalb der Spezifikation wird oft zwischen äußeren und inneren Bestandteilen unterschieden. Innere Bestandteile sind von allen Seiten von weiteren Bestandteilen umgeben, äußere Bestandteile liegen am Rand des Polygonnetzes, d.h. sie sind im Geflecht es Polygonnetzes nicht von allen Seiten von weiteren Bestandteilen umgeben. Die Operationen sind sowohl in 2D als auch in 3D ausführbar. So werden entweder zweidimensionale Flächen wie z.B. Dreiecke oder dreidimensionale Körper wie z.B. Tetraeder gebildet. Die Abbildungen zu den Operationen sind wegen der Verständlichkeit und besseren Abbildbarkeit in 2D. Einige der vorgestellten Operationen beinhalten die Delaunay-Triangulierung aus Kapitel 4.1.3.3. Sie verwenden diese entsprechend der dort vorgestellten Definition.

Geometrie und Topologie verändernde Operationen

Die folgenden Operationen verändern sowohl die Geometrie als auch die Topologie eines Polygonnetzes. Sie wirken sich sowohl auf die Lage des Polygonnetzes wie auch auf das Polygonnetz selbst aus.

Einfügen und Entfernen eines Eckpunktes

insertVertex: mesh x vertex → mesh

Vorbedingung: Der neue Eckpunkt darf nicht auf einem bereits vorhandenen Eckpunkt liegen.
Nachbedingung: Fügt einen Eckpunkt in das Polygonnetz ein. Der Eckpunkt wird durch seine Koordinaten beschrieben. Es existieren drei verschiedene Fälle:
1. Liegt der Eckpunkt auf einer Kante, dann wird diese durch den Eckpunkt in zwei Kanten geteilt.
2. Liegt der Eckpunkt innerhalb einer Face, also innerhalb der gleichen Ebene wie die Eckpunkte, die die Face beschreiben, wird der neue Eckpunkt durch Kanten mit den Eckpunkten der Face, innerhalb welcher er liegt, verbunden. Es entstehen neue Flächen und Kanten.
3. Liegt der Eckpunkt außerhalb des Polygonnetzes wird er mit den Eckpunkten die am nächsten zu ihm liegen verbunden. Zum Verbinden des neuen Eckpunktes mit dem Polygonnetz wird die Delaunay-Triangulierung verwendet.
Bei allen Möglichkeiten wird das Polygonnetz durch den neuen Eckpunkt verändert. Durch den zusätzlichen Eckpunkt wird bei allen Fällen die Geometrie und die Topologie verändert. Die Geometrie, da durch den neuen Eckpunkt sich die Beschreibung des

Netzes ändert und die Topologie, da durch den neuen Eckpunkt neue Nachbarschaftsbeziehungen entstehen.
Fehlerbedingung: —

Die drei verschiedenen Fälle des Einfügens eines Eckpunktes werden in den Abbildungen 5.10, 5.11 und 5.12 aufgezeigt. Abbildung 5.10 zeigt das Einfügen eines Eckpunktes innerhalb einer Kante. Aus der alten Kante entstehen zwei neue Kanten. Abbildung 5.11 verdeutlicht das Einfügen eines Eckpunktes innerhalb einer Face. Der neue Eckpunkt wird über Kanten mit den vorhandenen Eckpunkten der Face verbunden. Das Einfügen des Eckpunktes außerhalb eines Polygonnetzes stellt Abbildung 5.12 dar. Der Eckpunkt wird über Kanten mit weiteren Eckpunkten verbunden, so dass eine neue Face entsteht.

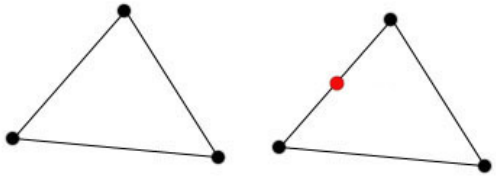


Abbildung 5.10: Eckpunkt innerhalb einer Kante einfügen

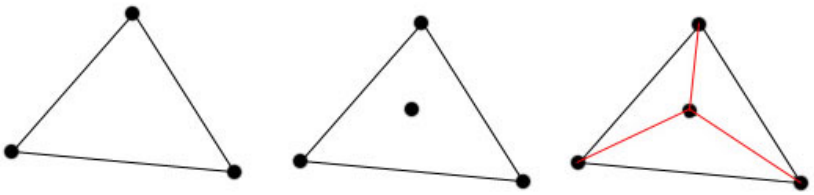


Abbildung 5.11: Eckpunkt innerhalb einer Face einfügen

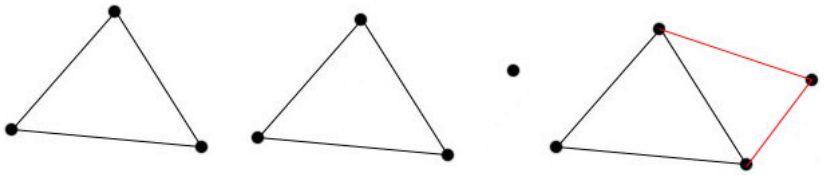


Abbildung 5.12: Eckpunkt außerhalb eines Polygonnetzes einfügen

deleteVertex: mesh x vertex → mesh

Vorbedingung: Der Eckpunkt muss in dem Polygonnetz vorhanden sein.
Nachbedingung: Löscht einen Eckpunkt aus dem Polygonnetz. Das Polygonnetz wird dadurch verändert. Es gibt zwei Fälle:
1. Der Eckpunkt wird am Rand des Polygonnetzes entfernt. Alle Kanten, die dieser Eckpunkt begrenzt, werden entfernt. Durch das Entfernen der Kanten müssen auch die Flächen, zu denen die Kanten gehören entfernt werden.
2. Der Eckpunkt liegt innerhalb eines Polygonnetzes, ist also von Faces umgeben. Alle Kanten, die durch diesen Eckpunkt begrenzt werden werden gelöscht. Es entsteht eine neue Face aus den Faces, die um diesen Eckpunkt lagen. Die neu entstehende Face muss planar, konvex und einfach sein.
Ändert die Geometrie durch das Entfernen der Eckpunkte. Falls es sich um äußere Eckpunkte handelt ändert sich die gesamte Form bzw. falls es sich um innere Eckpunkte handelt ändert sich die Form der Flächen innerhalb des Netzes. Dadurch ändert sich auch die Topologie. Die umliegenden Eckpunkte haben diesen Eckpunkt nicht mehr als Nachbarn. Sie erhalten neue Nachbarn. Die umliegenden Flächen erhalten neue Nachbarflächen.
Fehlerbedingung: —

Abbildung 5.13 zeigt das Entfernen eines Eckpunktes am Außenrand eines Polygonnetzes. In Abbildung 5.14 wird das Entfernen eines Eckpunktes im Inneren eines Polygonnetzes dargestellt.

Einfügen und Entfernen einer Kante

insertEdge: mesh x edge → mesh

Vorbedingung: Falls innerhalb einer Face eingefügt wird muss diese mehr als drei Eckpunkte haben. Die Eckpunkte dürfen in diesem Fall nicht nebeneinander liegen, also bereits durch eine Kante verbunden sein. Die Kante wird zwischen zwei vorhandenen Eckpunkten eingefügt.
Nachbedingung: Fügt eine Kante in das Polygonnetz ein und verändert das Polygon-

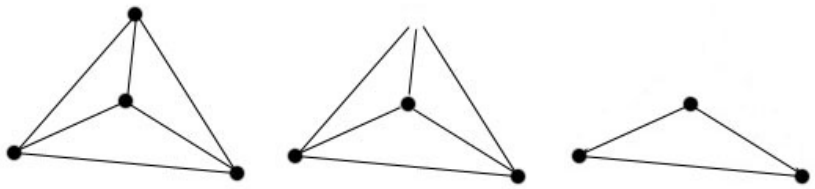


Abbildung 5.13: Eckpunkt außen entfernen

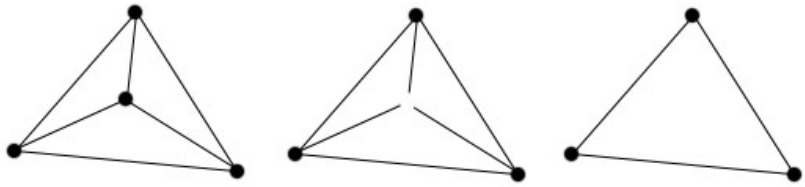


Abbildung 5.14: Eckpunkt innen entfernen

netz. Es wird zwischen zwei Fällen unterschieden: innen und außen.
1. Die Kante wird innerhalb eines Polygonnetzes zwischen bereits vorhandene Eckpunkte eingefügt.
a) Diese gehören zur gleichen Face. Die Face wird in zwei Faces aufgeteilt.
b) Die Eckpunkte gehören nicht zur gleichen Face. In diesem Fall wird in alle Kanten, die durch die neue Kante geschnitten werden ein Eckpunkt eingefügt. Die Kante wird in mehrere Kanten aufgeteilt und es entstehen neue Faces.
2. Die Kante wird außerhalb des Polygonnetzes eingefügt. Die Endpunkte der Kante werden mit den nächsten Eckpunkten durch Kanten verbunden. Um neue Faces zu erstellen wird an dieser Stelle die Delaunay-Triangulierung verwendet.
In allen Fällen wird die Form des Polygonnetzes durch neue Formen der Flächen beschrieben. Dadurch ändert sich die Geometrie.
Ändert auch die Topologie. Durch die neue Kante entstehen neue Nachbarschaftsbeziehungen zwischen der neuen Kante und den alten Bestandteilen des Polygonnetzes.
Fehlerbedingung: —

Abbildungen 5.15 bis 5.17 zeigen die verschiedenen Fälle beim Einfügen einer Kante auf.

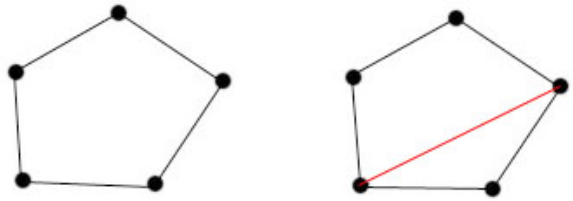


Abbildung 5.15: Einfügen einer Kante innerhalb einer Face, vorhandene Eckpunkte verwendet

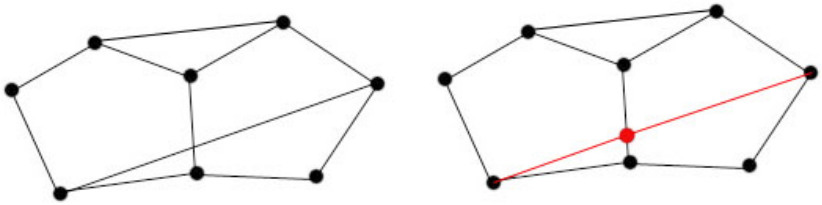


Abbildung 5.16: Einfügen einer Kante innerhalb mehrerer Faces, vorhandene Eckpunkte verwendet

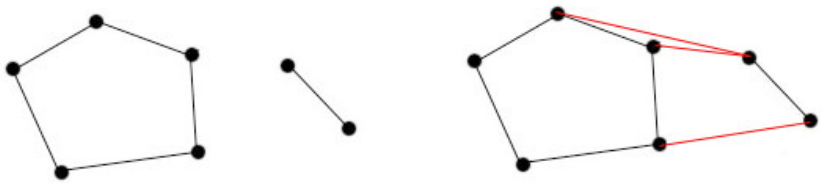


Abbildung 5.17: Einfügen einer Kante außerhalb des Polygonnetzes

deleteEdge: mesh x edge → mesh

Vorbedingung: Die zu entfernende Kante muss innerhalb des Polygonnetzes vorhanden sein.
Nachbedingung: Entfernt eine Kante innerhalb des Polygonnetzes. Das Polygonnetz wird verändert. Es existieren zwei Fälle zwischen denen unterschieden wird:
1. Die Kante liegt am Rand des Polygonnetzes. Die Face, die zu dieser Kante gehört wird mit entfernt.
2. Die Kante liegt innen. Das Entfernen einer Kante innerhalb des Polygonnetzes vereint

zwei Faces zu einer Face. Die neu entstehende Face muss planar, konvex und einfach sein.
In beiden Fällen gilt: Gehören die beiden Eckpunkte der Kante zu keiner weiteren Kante, so werden sie entfernt. Es werden Geometrie und Topologie verändert. Die Form der Flächen bzw. des Polygonnetzes wird verändert, also die Geometrie. Durch das Entfernen der Kante bzw. Kanten und Faces entstehen neue Nachbarschaftsbeziehungen zwischen den Faces, Kanten und Eckpunkten in der Umgebung der entfernten Kante.
Fehlerbedingung: —

In Abbildung 5.18 wird das Entfernen einer Außenkante dargestellt. Abbildung 5.19 zeigt ein Beispiel in dem eine Innenkante gelöscht wird. Es entsteht eine konkave Face. Weitere Kanten werden gelöscht. Es entsteht eine Face aus den drei zuvor.

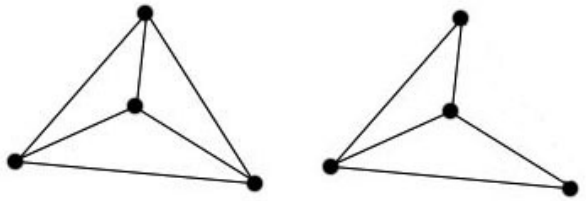


Abbildung 5.18: Außenkante entfernen

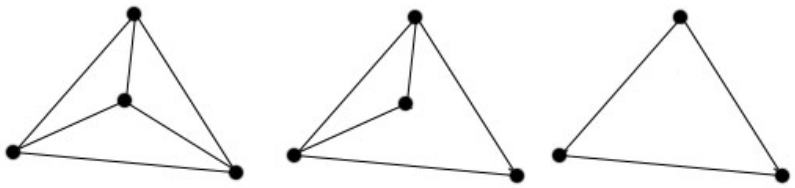


Abbildung 5.19: Innenkante entfernen

Einfügen und Entfernen einer Face

insertFace: mesh x face → mesh

Vorbedingung: —
Nachbedingung: Fügt eine Face in ein Polygonnetz ein. Das Polygonnetz wird verändert.

Die neue Face kann innerhalb oder außerhalb des Polygonnetzes eingefügt werden.
1. Die neue Face liegt außerhalb des Polygonnetzes. Die Eckpunkte der neuen Face werden mittels Kanten mit dem Polygonnetz verbunden. Das Verbinden wird mit Hilfe der Delaunay-Triangulierung durchgeführt.
2. Die neue Face liegt innerhalb des Polygonnetzes.
a) Die neue Face liegt innerhalb einer Face auf der gleichen Ebene. Die Eckpunkte der neuen Face werden durch Kanten mit den Eckpunkten der Face innerhalb welcher sich die neue Face befindet verbunden.
b) Die neue Face liegt innerhalb mehrerer Faces. Diese müssen planar sein. Die Eckpunkte der neuen Face werden mit den Eckpunkten der Faces innerhalb welcher sie sich befindet durch Kanten verbunden. Alle durch die neue Face geschnittenen Kanten erhalten an der Schnittstelle einen neuen Eckpunkt. Dadurch werden die Kanten aufgeteilt.
Verändert die Geometrie durch die Formänderung des Polygonnetzes bzw. seiner Flächen und die Topologie durch die neuen Nachbarschaftsbeziehungen zur neuen Face.
Fehlerbedingung: —

Abbildungen 5.20 bis 5.22 zeigen die drei Fälle beim Einfügen einer Face in ein Polygonnetz: das Einfügen außerhalb des Polygonnetzes, das Einfügen innerhalb einer Face und das Einfügen innerhalb mehrerer Faces.

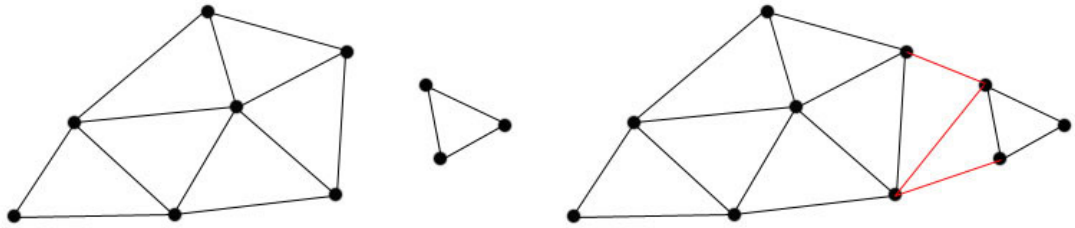


Abbildung 5.20: Einfügen einer Face außerhalb des Polygonnetzes

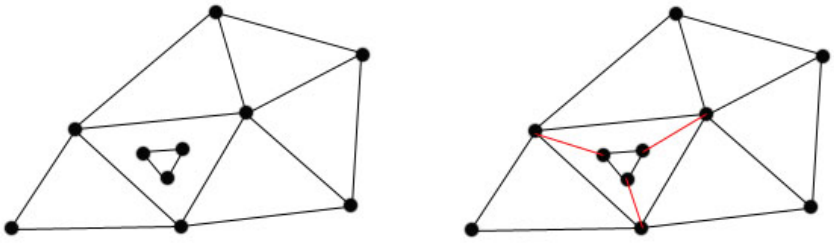


Abbildung 5.21: Einfügen einer Face innerhalb einer anderen Face

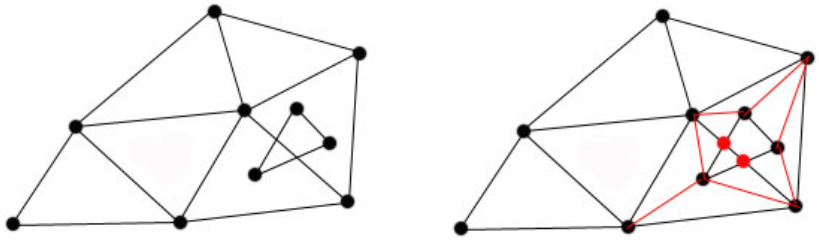


Abbildung 5.22: Einfügen einer Face innerhalb mehrerer Faces

deleteFace: mesh x face → mesh

Vorbedingung: Die zu entfernende Face muss innerhalb des Polygonnetzes vorhanden sein.
Nachbedingung: Entfernt eine Face aus dem Polygonnetz. Das Polygonnetz wird verändert.
1. Es wird eine Außenface entfernt. Alle Eckpunkte und Kanten dieser Face werden entfernt, falls sie zu keiner anderen Face gehören.
2. Es wird eine Innenface entfernt. Durch das Entfernen der Face werden die Nachbarfaces zu einer Face zusammengeschlossen. Die neu entstehende Face muss planar, konvex und einfach sein.
Ändert die Geometrie. Durch die fehlende Face ändert sich die Form des Polygonnetzes. Die Eckpunkte der Face bleiben erhalten. Ändert die Topologie. Die umliegenden Faces haben die gelöschte Face nicht mehr als Nachbarface. Dadurch ändern sich die Nachbarschaftsbeziehungen der umliegenden Faces.
Fehlerbedingung: —

In den Abbildungen 5.23 und 5.24 ist das Entfernen einer Face aus einem Polygonnetz dargestellt.

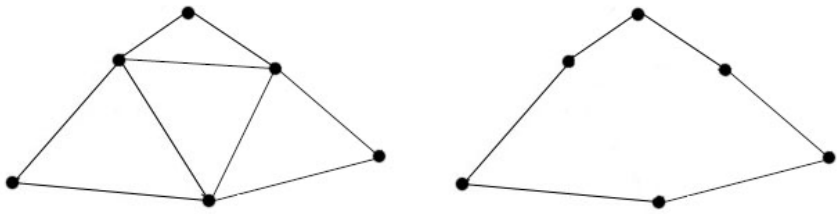


Abbildung 5.23: Entfernen einer Face innerhalb eines Polygonnetzes

Tabelle 5.1 führt noch einmal alle Operationen auf, die Geometrie und Topologie verändern.

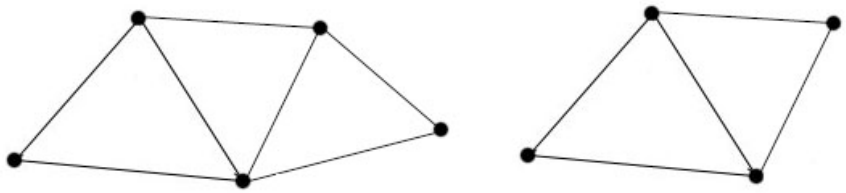


Abbildung 5.24: Entfernen einer Außenface

veränderte Geometrie und Topologie
insertVertex: mesh x vertex ⟶ mesh deleteVertex: mesh x vertex ⟶ mesh
insertEdge: mesh x edge ⟶ mesh deleteEdge: mesh x edge ⟶ mesh
insertFace: mesh x face ⟶ mesh deleteFace: mesh x face ⟶ mesh

Tabelle 5.1: Operationen, die Geometrie und Topologie verändern

Geometrie verändernde und Topologie nicht-verändernde Operationen

Diese Operationen bewirken eine Veränderung der Geometrie, aber nicht der Topologie. Die folgenden Operationen können nochmals unterteilt werden. Operationen, die sich nur auf bestimmte Bestandteile des Polygonnetzes beziehen werden als Deformationen bezeichnet. Diese Änderungen wirken sich lokal aus und ändern die Gestalt bzw. Form des Polygonnetzes. Deformationen werden mit Hilfe von transformierenden Operationen wie dem Rotieren, Translatieren (Verschieben) und Skalieren durchgeführt. Änderungen, die das gesamte Polygonnetz betreffen, werden Transformationen genannt. Sie wirken global. Die Deformationen und Transformationen werden in einem Weltkoordinatensystem durchgeführt. Weltkoordinaten beschreiben die Koordinaten eines Objektes bezüglich des Ursprungs in einem Koordinatensystem, d.h. alle Deformationen und Transformationen beziehen sich auf den Ursprung. Bei allen Operationen werden lediglich die Punktkoordinaten verändert. Die Topologie ist gegenüber Transformationen und Deformationen invariant (unveränderlich).

Rotieren

rotateVertex: mesh x vertex x double x double x double → mesh

Vorbedingung: Das Rotieren eines Eckpunktes ist nur innerhalb eines Polygonnetzes mit dreieckigen Faces möglich, da sonst die Rotation zu nicht planaren Faces führt. Ein Eckpunkt darf nur so weit rotiert werden, dass die an ihm hängenden Kanten keine anderen Kanten schneiden, d.h. keine nicht-einfachen Faces entstehen.
Nachbedingung: Rotiert einen Eckpunkt um die Achsen x, y und z um die angegebene Gradzahl in double. Für jede Achse kann eine Gradzahl angegeben werden. Durch die Rotation des Eckpunktes wird das Polygonnetz verändert. Die an dem Eckpunkt hängenden Kanten müssen an die veränderten Koordinaten des Eckpunktes angepasst werden. Dadurch ändern auch die Faces, die zu diesem Eckpunkt und den veränderten Kanten gehören ihre Form. Sie müssen hierbei planar und konvex bleiben. Ändert die Geometrie, da sich die Lage des Eckpunktes und die Form des Polygonnetzes ändern. Ändert nicht die Topologie. Die Beziehungen zwischen den Bestandteilen des Polygonnetzes bleiben erhalten.
Fehlerbedingung: —

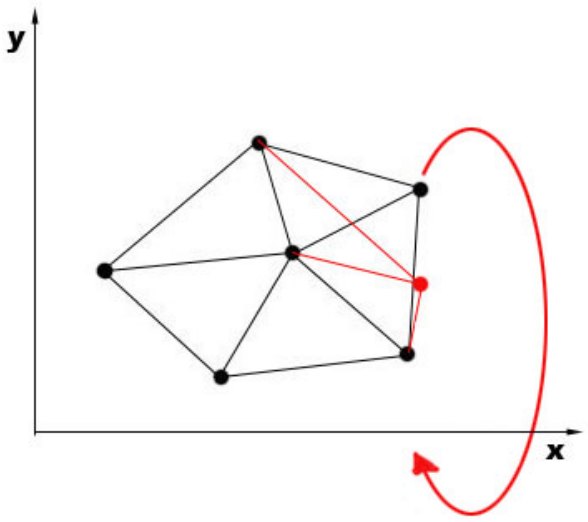


Abbildung 5.25: Rotation eines Eckpunktes um die x-Achse

rotateEdge: mesh x edge x double x double x double → mesh

Vorbedingung: Es können nur Kanten in Polygonnetzen rotiert werden, deren Faces Dreiecke sind. Das Rotieren einer Kante innerhalb eines Polygonnetzes mit Faces, die mehr als drei Kanten haben führt zu nicht planaren Faces. Die Kante darf nur so weit

rotiert werden wie sie keine weiteren Kanten schneidet, und somit keine nicht-einfachen Faces verursacht.
Nachbedingung: Rotiert eine Kante um die Achsen x, y und z um die angegebene Gradzahl in double. Für jede Achse kann eine Gradzahl angegeben werden. Durch die Rotation der Kante wird das Polygonnetz verändert. Das Rotieren der Kante wird ausgeführt indem ihre beiden Eckpunkte rotiert werden. Die Lage und Form der anliegenden Kanten und die Form der dazugehörigen Faces wird an die neue Lage der Kante angepasst. Es wird die Geometrie verändert, da sich die Lage der Bestandteile und die Form des Polygonnetzes verändern. Ändert nicht die Topologie. Die Beziehungen zwischen den Bestandteilen des Polygonnetzes bleiben erhalten.
Fehlerbedingung: —

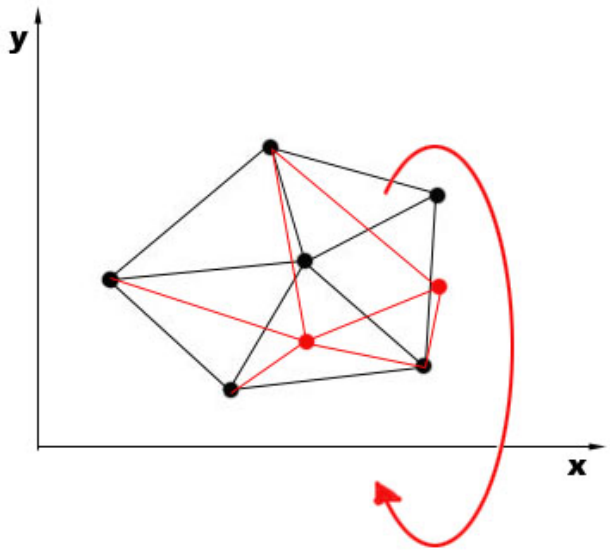


Abbildung 5.26: Rotation einer Kante um die x-Achse

rotateFace: mesh x face x double x double x double → mesh

Vorbedingung: Es können nur dreieckige Faces rotiert werden. Das Rotieren einer Face innerhalb eines Polygonnetzes mit nicht dreieckigen Faces führt zu nicht planaren Faces. Die rotierte Face darf keine anderen Faces schneiden.
Nachbedingung: Rotiert die Face um die Achsen x, y und z um die angegebene Gradzahl in double. Für jede Achse kann eine Gradzahl angegeben werden. Durch die Rotation der Face wird das Polygonnetz verändert. Durch die Rotation einer Face müssen die anliegenden Kanten an die neuen Eckpunkte angepasst werden. Nachbarfaces ändern dadurch ihre Form. Ändert die Geometrie, da sich die Lage der Bestandteile und die Form des Polygonnetzes verändern. Ändert nicht die Topologie. Die Beziehungen zwischen den

Bestandteilen des Polygonnetzes bleiben erhalten.
Fehlerbedingung: —

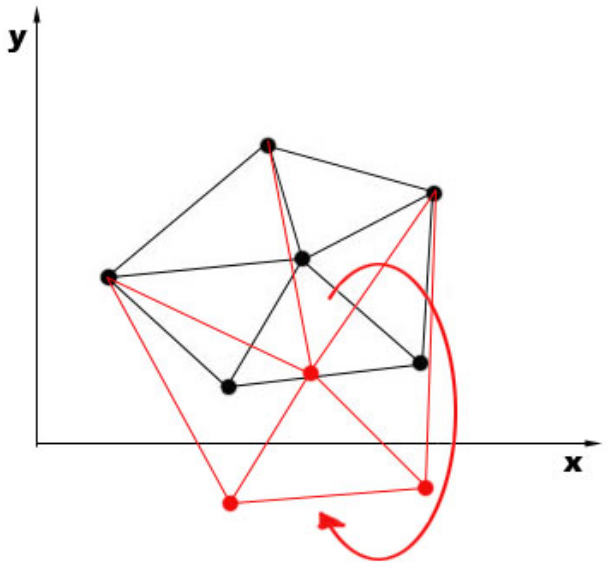


Abbildung 5.27: Rotation einer Face um die x-Achse

rotateMesh: mesh x double x double x double → mesh

Vorbedingung: —
Nachbedingung: Rotiert das gesamte Polygonnetz. Die Double-Werte geben die Gradzahl an um die rotiert wird. Die Reihenfolge der Gradangaben gibt gleichzeitig die Achsen an. Diese ist x, y und z. Durch die Rotation wird die Lage des Polygonnetzes mit seinen Bestandteilen im Koordinatensystem verändert. Da die Koordinaten der Eckpunkte, also ihre Lage sich verändert, wird die Geometrie des Polygonnetzes verändert. Die Topologie des Polygonnetzes bleibt unverändert. Die Beziehungen zwischen den Bestandteilen bleiben erhalten.
Fehlerbedingung: —

Skalieren

Für das Skalieren der Bestandteile werden Punkte innerhalb eines Bestandteils bzw. innerhalb des Polygonnetzes festgelegt. Diese werden als eine Art Gewicht genutzt. Je näher der Punkt zu einer Seite hinzeigt, desto stärker wird die Skalierung in diese Richtung ausgeführt. Liegt der Punkt in der Mitte erfolgt die Skalierung gleichmäßig zu allen Seiten.

Das Skalieren eines Eckpunktes ist nicht möglich, da ein Eckpunkt keine Dimension

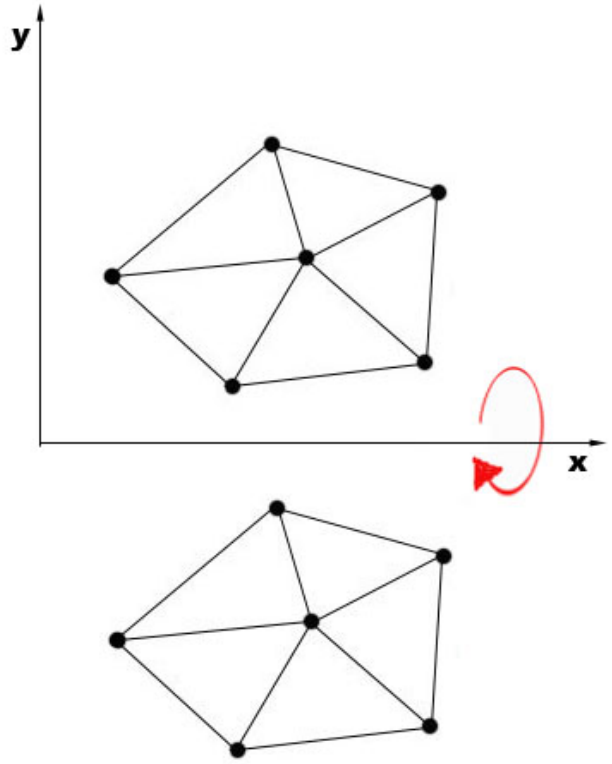


Abbildung 5.28: Rotation eines Polygonnetzes um die x-Achse

hat. Damit hat eine Skalierung keine Auswirkung auf den Eckpunkt.

scaleEdge: mesh x edge x vertex x double → mesh

Vorbedingung: Der Punkt, welcher als Gewicht dient, muss sich auf der zu skalierenden Kante befinden.
Nachbedingung: Skaliert in einem Polygonnetz eine Kante. Die Skalierung einer Kante erfolgt über ein Gewicht in Form eines Punktes welcher auf der Kante platziert wird und einen Faktor, der die Stärke der Skalierung angibt. Die Angabe des Faktors erfolgt durch einen double-Wert. Das Skalieren einer Kante führt zum Verschieben ihrer Endpunkte. Liegt das Gewicht genau in der Mitte der Kante wird die Kante auf beiden Seiten gleich skaliert. Liegt das Gewicht links auf der Kante wird mehr auf der rechten Seite skaliert, liegt das Gewicht rechts wird mehr auf der linken Seite skaliert. Die an dieser Kante angrenzenden Kanten ändern sich durch die veränderte Lage der Endpunkte der skalierten Kante. Das Skalieren einer Kante verändert die Geometrie, da sich die Lage der Eckpunkte und die Form des Polygonnetzes ändern. Die Topologie bleibt erhalten.
Fehlerbedingung: —

Abbildung 5.29 verdeutlicht das Prinzip der Skalierung. Der Faktor beträgt hier -2,

d.h. die Größe der Kante wird um die Hälfte reduziert. Das Gewicht liegt in der Mitte der Kante.

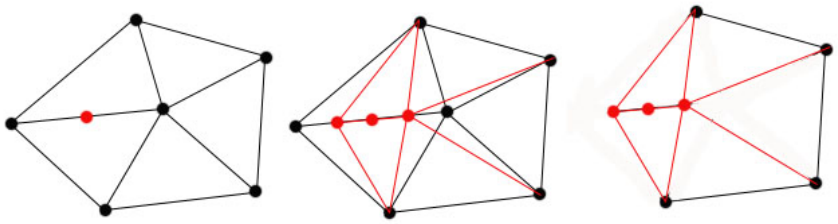


Abbildung 5.29: Skalieren einer Kante innerhalb eines Polygonnetzes

scaleFace: mesh x face x vertex x double → mesh

Vorbedingung: Der Punkt, welcher als Gewicht dient, muss sich innerhalb der zu skalierenden Face befinden.
Nachbedingung: Skaliert eine Face in einem Polygonnetz. Die Skalierung erfolgt über alle Eckpunkte der Face. Es wird ein Punkt angegeben, der als Gewicht dient und die Richtung der Skalierung beeinflusst. Ein Faktor bestimmt die Stärke der Skalierung. Er wird durch einen double-Wert angegeben. Befindet sich der Punkt in der Mitte der Face wird diese gleichmäßig zu allen Seiten skaliert. Befindet sich der Punkt beispielsweise in der linken oberen Ecke erfolgt die Skalierung in diese Richtung. Die an der Face angrenzenden Faces ändern ihre Form durch die veränderte Lage gemeinsamer Eckpunkte. Ändert die Geometrie durch Veränderung der Form der Face und dadurch des gesamten Netzes. Die topologischen Beziehungen bleiben erhalten.
Fehlerbedingung: —

In Abbildung 5.30 und Abbildung 5.31 wird die Skalierung einer Face dargestellt. In Abbildung 5.30 befindet sich das Gewicht in der Mitte. In Abbildung 5.31 befindet sich das Gewicht am linken oberen Rand.

scaleMesh: mesh x vertex x double → mesh

Vorbedingung: —
Nachbedingung: Skaliert das Polygonnetz um einen gegebenen Faktor. Dieser bestimmt wie stark das Polygonnetz skaliert wird. Zusätzlich wird ein Punkt als Gewicht angegeben. Dieser bestimmt die Richtung der Skalierung. Der Faktor wird durch einen double-Wert angegeben. Durch die Skalierung wird das Polygonnetz einschließlich seiner Bestandteile verändert und nimmt eine andere Form an. Die Skalierung verursacht eine Veränderung der Geometrie, da die Lage der Eckpunkte des Polygonnetzes sich ändert. Die topologischen Beziehungen bleiben unverändert.
Fehlerbedingung: —

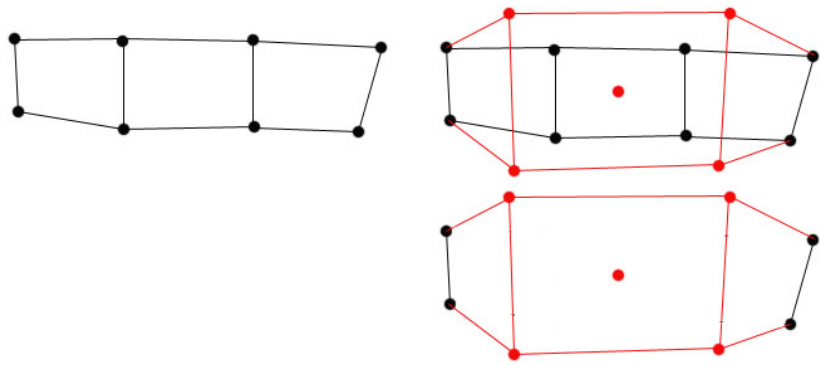


Abbildung 5.30: Skalieren einer Face innerhalb eines Polygonnetzes mit dem Gewicht in der Mitte

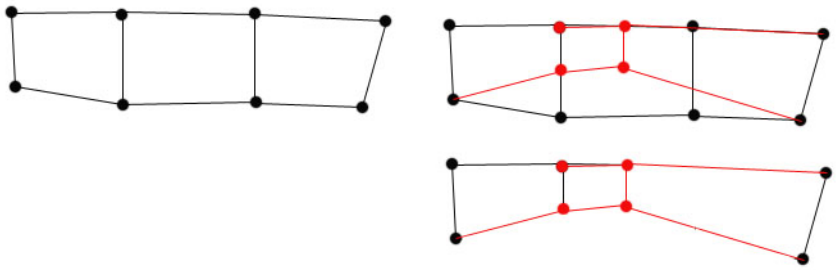


Abbildung 5.31: Skalieren einer Face innerhalb eines Polygonnetzes mit dem Gewicht links oben

Abbildung 5.32 zeigt eine Beispiel-Skalierung eines Polygonnetzes. Das Gewicht liegt links unten.

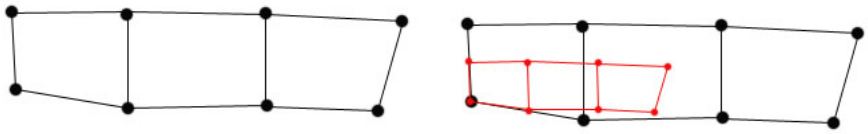


Abbildung 5.32: Skalieren eines Polygonnetzes mit dem Gewicht links unten

Translatieren (Verschieben)

moveVertex: mesh x vertex x double x double x double → mesh

Vorbedingung: Der verschobene Eckpunkt darf keine Kante kreuzen.
Nachbedingung: Verschiebt einen Eckpunkt des Polygonnetzes. Der Eckpunkt kann

in drei Richtungen verschoben werden. Diese werden durch die drei Double-Werte angegeben. Die Reihenfolge ist x, y, z. Durch das Verschieben des Eckpunktes werden die Kanten verändert und die Faces zu denen sie gehören in ihrer Form verändert. Ändert die Lage des Eckpunktes und die Form des Polygonnetzes. Es wird die Geometrie verändert. Die Beziehungen zwischen den Bestandteilen bleiben erhalten. Die Topologie bleibt unverändert.
Fehlerbedingung: —

Abbildung 5.33 zeigt das Verschieben eines Eckpunktes.

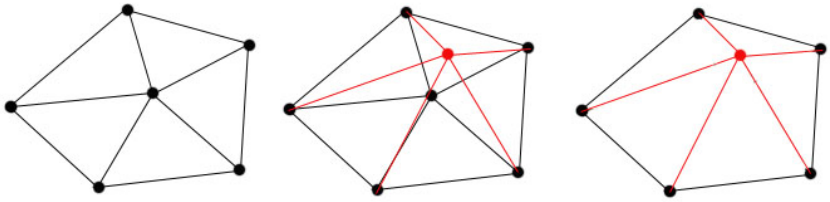


Abbildung 5.33: Verschieben eines Punktes

moveEdge: mesh x edge x double x double x double → mesh

Vorbedingung: Durch das Verschieben der Kante dürfen keine Faces entstehen deren Kanten sich kreuzen.
Nachbedingung: Verschiebt eine Kante im Polygonnetz. Es werden die beiden Endpunkte der Kante in Richtung x, y, z um einen Double-Wert verschoben. Die anliegenden Kanten werden an die neuen Eckpunkte angepasst. Es ändert sich die Form der Faces, die zu dieser Kante gehören und die die Eckpunkte der Kante gemeinsam haben. Die Position der Kante wird verändert. Dadurch ändert sich die Form des Polygonnetzes. Die Geometrie wird verändert. Die Topologie bleibt erhalten, da sich die Beziehungen zwischen den Bestandteilen nicht ändern.
Fehlerbedingung: —

In Abbildung 5.34 ist das Verschieben einer Kante dargestellt.

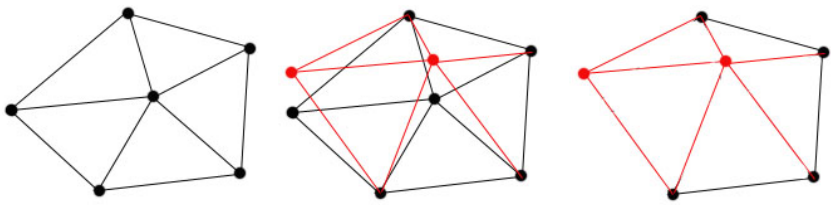


Abbildung 5.34: Verschieben einer Kante

moveFace: mesh x face x double x double x double → mesh

Vorbedingung: Das Verschieben einer Face darf nur erfolgen wenn die Kanten der Face keine anderen Kanten kreuzen.
Nachbedingung: Verschiebt eine Face in einem Polygonnetz. Das Verschieben erfolgt über die Eckpunkte der Face und kann in Richtung x, y, z erfolgen. Wie weit verschoben wird, wird durch die Double-Werte angegeben. Alle an der Face hängenden Kanten werden mittranformiert. Die anliegenden Faces ändern ihre Form. Es wird die Geometrie verändert, da sich die Lage der Eckpunkte und die Form des Polygonnetzes ändern. Die Topologie bleibt erhalten, da sich die Nachbarschaftsbeziehungen zwischen den Bestandteilen nicht ändern.
Fehlerbedingung: —

Abbildung 5.35 verdeutlicht das Verschieben einer Face.

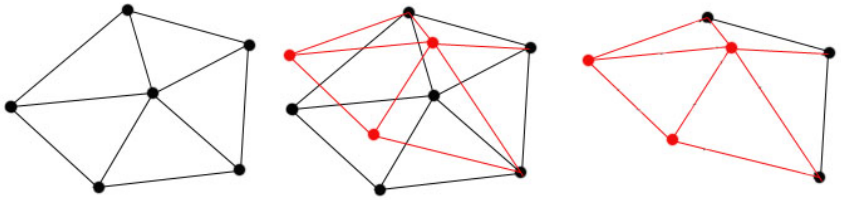


Abbildung 5.35: Verschieben einer Face

moveMesh: mesh x double x double x double → mesh

Vorbedingung: —
Nachbedingung: Verschiebt das Polygonnetz in die Richtungen x, y, z mit einem Double-Wert.
Es ändert sich die Lage des Polygonnetzes, also seine Geometrie, nicht aber die Topologie. Die Beziehungen der Bestandteile bleiben unverändert. Auch die Form des Polygonnetzes ändert sich nicht.
Fehlerbedingung: —

Abbildung 5.36 zeigt das Verschieben eines Polygonnetz. Zur Verdeutlichung ist das Verschieben innerhalb eines Koordinatensystems dargestellt.

Tabelle 5.2 führt noch einmal alle Operationen auf, die Geometrie verändern, aber Topologie erhalten.

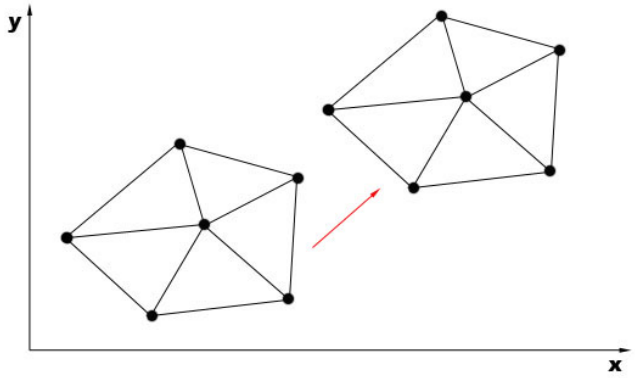


Abbildung 5.36: Verschieben eines Polygonnetzes

veränderte Geometrie und unveränderte Topologie
rotateVertex: mesh x vertex x double x double x double ⟶ mesh rotateEdge: mesh x edge x double x double x double ⟶ mesh rotateFace: mesh x face x double x double x double ⟶ mesh rotateMesh: mesh x double x double x double ⟶ mesh
scaleEdge: mesh x edge x double x double ⟶ mesh scaleFace: mesh x face x double x double ⟶ mesh scaleMesh: mesh x double x double ⟶ mesh
moveVertex: mesh x vertex x double x double x double ⟶ mesh moveEdge: mesh x edge x double x double x double ⟶ mesh moveFace: mesh x face x double x double x double ⟶ mesh moveMesh: mesh x double x double x double ⟶ mesh

Tabelle 5.2: Operationen, die Geometrie verändern und Topologie erhalten

Geometrie und Topologie nicht-verändernde Operationen

Die hier aufgeführten Operationen stellen reine Abfragen von Informationen dar und ändern weder die Geometrie noch die Topologie. Auch das Polygonnetz selbst bleibt unverändert. Außer den Operationen, bei denen das Vorhandensein eines Bestandteiles abgefragt wird und den Operationen, die eine Art von Bestandteilen ausgeben, beziehen sich die aufgeführten Operationen auf die Abfrage topologischer Informationen. Der in Abschnitt 5.2.2 vorgestellte vef-Graph beschreibt genau diese Operationen. Daher werden die hier aufgeführten Operationen zusammen mit den entsprechenden Relationen vorgestellt.

Die toplogischen Informationen beziehen sich auf die drei Bestandteile des Polygonnetzes. Verschiedene Abfragen bezüglich dieser Bestandteile sind möglich. Aus den Operationen können die Relationen und aus den Relationen die Operationen hergeleitet werden. Unterschieden wird zwischen get-Operationen, die Bestandteile liefern und zwischen is-Operationen, die das Vorhandensein der Bestandteile prüfen.

Die nachfolgenden Operationen lassen sich wie folgt den Relationen des vef-Graphen zuordnen: Jede Operation entspricht einer Relation. Zu jeder Relation exisitiert jeweils eine get- und eine is-Operation. Jede Operation und damit auch jede Relation fragt Beziehungen zwischen zwei Bestandteilen in zwei Richtungen ab. Das heißt es wird jeweils ermittelt wie beispielsweise ein Eckpunkt zu einer Kante und wie eine Kante zu einem Eckpunkt in Beziehung steht. Ausnahmen bilden die Operationen und die Relationen die die Beziehung zwischen gleichen Bestandteilen abfragen. Dies sind die Relationen vv, ee und ff. In Tabelle 5.3 wird zu jeder Operation die entsprechende Relation aufgeführt. Keine Relationen existieren zu den am Anfang aufgeführten Operationen.

get-Operationen

getVerticesWhichBelongToEdge: mesh x edge → set of vertices

Vorbedingung: —
Nachbedingung: Liefert alle Eckpunkte, die zu einer Kante gehören. Dies sind die beiden Endpunkte der Kante.
Fehlerbedingung: —
Diese Operation entspricht der Relation ve und beschreibt das ein Eckpunkt eine Kante begrenzt.

getEdgesWhichBelongToVertex: mesh x vertex → set of edges

Vorbedingung: —
Nachbedingung: Liefert Kanten, die zu einem bestimmten Eckpunkt gehören. Jede Kante wird durch ihre beiden Endpunkte beschrieben.
Fehlerbedingung: —
Die Operation entspricht der Relation ev. Es wird eine Kante von einem Eckpunkt begrenzt.

getVerticesWhichBelongToFace: mesh x face → set of vertices

Vorbedingung: —
Nachbedingung: Liefert alle Eckpunkte, die zu einer Face gehören.
Fehlerbedingung: —
Die Relation vf beschreibt diese Operation. Der Eckpunkt gehört zur Face.

getFacesWhichBelongToVertex: mesh x vertex → set of faces

Vorbedingung: —
Nachbedingung: Liefert alle Faces die einen Eckpunkt gemeinsam haben. Jede Face wird durch ihre Eckpunkte beschrieben.
Fehlerbedingung: —

Die Operation wird durch die Relation fv beschrieben. Die Face stößt an den Eckpunkt. Wird also von diesem begrenzt.

getEdgesWhichBelongToFace: mesh x face → set of edges

Vorbedingung: —
Nachbedingung: Liefert alle Kanten, die zu einer Face gehören. Jede Kante wird durch ihre Endpunkte beschrieben.
Fehlerbedingung: —
Die Relation ef beschreibt diese Operation. Die Kante begrenzt die Face.

getFacesWhichBelongToEdge: mesh x edge → set of vertices

Vorbedingung: —
Nachbedingung: Liefert alle Faces die eine Kante gemeinsam haben. Jede Face wird durch ihre Eckpunkte beschrieben.
Fehlerbedingung: —
Diese Operation wird durch die Relation fe beschrieben. Die Face stößt an die Kante, wird also von ihr begrenzt.

getAdjacentVerticesOfVertex: mesh x vertex → set of vertices

Vorbedingung: —
Nachbedingung: Liefert alle Eckpunkte, die Nachbarn eines gegebenen Eckpunktes sind.
Fehlerbedingung: —
Die Operation entspricht der Relation vv und beschreibt die Nachbarschaft zwischen zwei Eckpunkten.

getAdjacentEdgesOfEdge: mesh x edge ← set of edges

Vorbedingung: —
Nachbedingung: Liefert zu einer Kante alle Nachbarkanten. Jede Kante wird durch ihre Endpunkte beschrieben.
Fehlerbedingung: —
Diese Operation beschreibt die Nachbarschaft zwischen zwei Kanten und entspricht der Relation ee.

getAdjacentEdgesOfEdgeWithSameFace: mesh x edge → set of edges

Vorbedingung: —
Nachbedingung: Liefert zu einer Kante alle Nachbarkanten, die die gleiche Face begrenzen. Jede Kante wird durch ihre Endpunkte beschrieben.
Fehlerbedingung: —

Die Relation ee' beschreibt diese Operation. Zwei Kanten sind benachbart und gehören zur gleichen Face.

getAdjacentFacesOfFace: mesh x face → set of faces

Vorbedingung: Das Polygonnetz muss aus mindestens zwei Faces bestehen.
Nachbedingung: Liefert zu einer gegebenen Face alle Nachbarfaces. Jede Face wird durch ihre Eckpunkte beschrieben.
Fehlerbedingung: —
Die Operation wird durch die Relation ff beschrieben. Beide Faces sind benachbart.

getAllVerticesOfMesh: mesh → set of faces

Vorbedingung: —
Nachbedingung: Liefert alle Eckpunkte eines Polygonnetzes.
Fehlerbedingung: —

getAllEdgesOfMesh: mesh → set of faces

Vorbedingung: —
Nachbedingung: Liefert alle Kanten eines Polygonnetzes. Jede Kante wird durch ihre beiden Endpunkte beschrieben.
Fehlerbedingung: —

getAllFacesOfMesh: mesh → set of faces

Vorbedingung: —
Nachbedingung: Liefert alle Faces eines Polygonnetzes. Jede Face wird durch ihre Eckpunkte beschrieben.
Fehlerbedingung: —

is-Operationen

isVertexBoundingEdge: mesh x vertex x edge → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob der gegebene Eckpunkt die Kante begrenzt. Liefert true, falls dies zutrifft und false, falls dies nicht zutrifft.
Fehlerbedingung: —
Zu dieser Operation gehört die Relation ve. Ein Eckpunkt begrenzt eine Kante.

isEdgeContainingVertex: mesh x edge x vertex → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob die gegebene Kante den Eckpunkt enthält. Liefert true, falls dies zutrifft und false, falls dies nicht zutrifft.
Fehlerbedingung: —
Diese Operation kann auch durch die Relation ev beschrieben werden. Eine Kante wird von einem Eckpunkt begrenzt.

isVertexBoundingFace: mesh x vertex x face → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob der Eckpunkt die Face begrenzt. Ist dies der Fall wird true zurückgegeben, ist dies nicht der Fall wird false zurückgegeben.
Fehlerbedingung: —
Die Relation vf beschreibt diese Operation. Eckpunkt ist Teil der Face.

isFaceContainingVertex: mesh x face x vertex → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob die Face den Eckpunkt enthält. Ist dies der Fall wird true zurückgegeben, ist dies nicht der Fall wird false zurückgegeben.
Fehlerbedingung: —
Die Operation entspricht der Relation fv. Face wird von Eckpunkt begrenzt.

isEdgeBoundingFace: mesh x edge x face → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob die Kante die Face begrenzt. Trifft dies zu wird true zurückgeliefert. Trifft dies nicht zu wird false zurückgeliefert.
Fehlerbedingung: —
Die Operation kann auch durch die Relation ef beschrieben werden. Die Kante begrenzt die Face.

isFaceContainingEdge: mesh x face x edge → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob die Face die Kante enthält. Trifft dies zu wird true zurückgeliefert. Trifft dies nicht zu wird false zurückgeliefert.
Fehlerbedingung: —
Die Relation fe entspricht dieser Operation. Die Fläche wird von der Kante begrenzt.

isVertexAdjacentToVertex: mesh x vertex x vertex → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob die gegebenen Eckpunkte Nachbarn sind. Liefert true, falls der Eckpunkt enthalten ist und false, falls der Eckpunkt nicht enthalten ist.
Fehlerbedingung: —
Entspricht der Relation vv un beschreibt benachbarte Eckpunkte.

isEdgeAdjacentToEdge: mesh x edge x edge → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob die beiden Kanten benachbart sind. Trifft dies zu wird true zurückgeliefert. Trifft dies nicht zu wird false zurückgeliefert.
Fehlerbedingung: —
Durch die Relation ee werden benachbarte Kanten beschrieben.

isEdgeAdjacentToEdgeAndHaveSameFace: mesh x edge x edge → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob die beiden Kanten benachbart sind und zur gleichen Face gehören. Trifft dies zu wird true zurückgeliefert. Trifft dies nicht zu wird false zurückgeliefert.
Fehlerbedingung: —
Diese Operation entspricht der Relation ee' und beschreibt die Nachbarschaft zwischen Kanten, die zur gleichen Face gehören.

isFaceAdjacentToFace: mesh x face x face → bool

Vorbedingung: Das Polygonnetz muss mindestens zwei Faces enthalten.
Nachbedingung: Überprüft, ob die beiden Faces benachbart sind. Liefert true, falls der Eckpunkt enthalten ist und false, falls der Eckpunkt nicht enthalten ist.
Fehlerbedingung: —
Entspricht der Relation ff und beschreibt die Nachbarschaft zwischen Faces.

isVertexInsideMesh: mesh x vertex → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob der gegegebene Eckpunkt im Polygonnetz enthalten ist. Liefert true, falls der Eckpunkt enthalten ist und false, falls der Eckpunkt nicht enthalten ist.
Fehlerbedingung: —

isEdgeInsideMesh: mesh x edge → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob die Kante im Polygonnetz enthalten ist. Liefert true, falls die Kante im Polygonnetz enthalten ist und false, falls die Kante nicht im Polygonnetz enthalten ist.
Fehlerbedingung: —

isFaceInsideMesh: mesh x face → bool

Vorbedingung: —
Nachbedingung: Überprüft, ob die Face im Polygonnetz enthalten ist. Bei wahr wird true ausgegeben, bei unwahr wird false zurückgegeben.
Fehlerbedingung: —

Tabelle 5.3 führt noch einmal die wichtigsten topologischen Abfragen auf. Zusätzlich werden die Relationen des vef-Graphen aufgeführt, falls vorhanden.

unveränderte Geometrie und Topologie	**Relation**
getAllVerticesOfMesh: mesh ⟶ set of vertex getAllEdgesOfMesh: mesh ⟶ set of edge getAllFacesOfMesh: mesh ⟶ set of face	
isVertexInsideMesh: mesh x vertex ⟶ bool isEdgeInsideMesh: mesh x edge ⟶ bool isFaceInsideMesh: mesh x face ⟶ bool	
getVerticesWhichBelongToEdge: mesh x edge ⟶ set of vertex getEdgesWhichBelongToVertex: mesh x vertex ⟶ set of edge getVerticesWhichBelongToFace: mesh x face ⟶ set of vertex getFacesWhichBelongToVertex: mesh x vertex ⟶ set of face getEdgesWhichBelongToFace: mesh x face ⟶ set of edge getFacesWhichBelongToEdge: mesh x edge ⟶ set of face	ve ev vf fv ef fe
getAdjacentVerticesOfVertex: mesh x vertex ⟶ set of vertex getAdjacentEdgesOfEdge: mesh x edge ⟶ set of edge getAdjacentEdgesOfEdgeWithSameFace: mesh x edge ⟶ set of edge getAdjacentFaceOfFace: mesh x face ⟶ set of face	vv ee ee' ff
isVertexBoundingEdge: mesh x vertex x edge ⟶ bool isEdgeContainingVertex: mesh x edge x vertex ⟶ bool isVertexBoundingFace: mesh x vertex x face ⟶ bool isFaceContainingVertex: mesh x face x vertex ⟶ bool isEdgeBoundingFace: mesh x edge x face ⟶ bool isFaceContainingEdge: mesh x face x edge ⟶ bool	ve ev vf fv ef fe

isVertexAdjacentToVertex: mesh x vertex x vertex ⟶ bool	vv
isEdgeAdjacentToEdge: mesh x edge x edge ⟶ bool	ee
isEdgeAdjacentToEdgeAndHaveSameFace: mesh x edge x edge ⟶ bool	ee'
isFaceAdjacentToFace: mesh face x face ⟶ bool	ff

Tabelle 5.3: Bezug zwischen Geometrie und Topologie unverändernden Operationen und Relationen im vef-Graphen

5.3.2 Fazit

In diesem Abschnitt wurde die Stufe Abstrakte Datenstruktur behandelt. Es wurde aufgezeigt welche Operationen auf einem Polygonnetz durchführbar sind und wie sie von geometrischen und topologischen Informationen abhängen. Nur Geometrische und topologische Informationen zusammen bilden ein Modell. Die Geometrie beschreibt dabei die Lage und Form des Polygonnetzes, durch welches das Modell beschrieben wird, die Topologie definiert die Beziehungen im Polygonnetz. Aus diesen beiden Bereichen lassen sich Operationen herleiten. Bezugnehmend auf den Abschnitt 5.2.3 wurden die Operationen nach den verschiedenene Kombinationen aus Geometrie und Topologie unterteilt. Zusätzlich lassen sich die Operationen in raumbezogen und objektbezogen aufteilen. Raumbezogen sind alle Operationen, welche die Geometrie verändern. Sie ändern nicht nur die Form, sondern auch die Lage der Bestandteile. Als objektbezogen können alle Operationen betrachtet werden, die Veränderungen an den Objektbeziehungen, also der Topologie des Objektes, bewirken.

Die Operationen die beides verändern beziehen sich auf alle Bestandteile des Polygonnetzes und betreffen das Einfügen und Entfernen von diesen. Sie führen vor allem zu Formveränderungen der einzelnen Flächen und zur Veränderung der Form des Polygonnetzes. Dadurch ändern sie die topologischen Beziehungen zu Nachbarkanten und -faces. Operationen, die nur die Geometrie verändern wurden zusätzlich in Deformationen und Transformationen unterteilt. Diese Unterteilung sollte die lokale und globale Wirkung der Operationen unterstreichen. Die Deformationen verändern nur bestimmte Stellen innerhalb des Netzes und ändern damit seine Form. Sie unterliegen der Beschränkung, dass sie nicht zu nicht-einfachen Polygonen führen dürfen. Transformationen verändern gleichmäßig das gesamte Netz. Alle Operationen lassen die topologischen Informationen unverändert. Zuletzt wurden Operationen vorgestellt, die beides unverändert lassen und nur reine Abfragen darstellen. In diesem Zusammenhang ist auch der in Abschnitt 5.2.2 vorgestellte vef-Graph zu betrachten. Dieser wird verwendet um topologische Beziehungen darzustellen und beschreibt die hier aufgeführten Operationen durch Relationen. Operationen und Relationen können einfach auf einander abgebildet werden.

Operationen auf Polygonnetzen können also sowohl zur Erstellung eines Polygonnetzes, zur Veränderung als auch zur Abfrage von Informationen dienen. Die Aufteilung in Geometrie und Topologie dient dazu die verschiedenen Auswirkungen einzelner Operationen auf das Polygonnetz zu unterstreichen. Zudem kann entschieden werden welche Ope-

rationen bei welcher Anwendung benötigt werden. Nachbarschaftsbeziehungen können durch Operationen der letzten Gruppe erfolgen.

Tabelle 5.5 listet noch einmal alle möglichen Operationen auf. Die topologischen Abfragen beinhalten zudem die entsprechenden Relationen.

veränderte Geometrie und Topologie	
insertVertex: mesh x vertex ⟶ mesh deleteVertex: mesh x vertex ⟶ mesh	
insertEdge: mesh x edge ⟶ mesh deleteEdge: mesh x edge ⟶ mesh	
insertFace: mesh x face ⟶ mesh deleteFace: mesh x face ⟶ mesh	
veränderte Geometrie und unveränderte Topologie	
rotateVertex: mesh x vertex x double x double x double ⟶ mesh rotateEdge: mesh x edge x double x double x double ⟶ mesh rotateFace: mesh x face x double x double x double ⟶ mesh rotateMesh: mesh x double x double x double ⟶ mesh	
scaleEdge: mesh x edge x double x double ⟶ mesh scaleFace: mesh x face x double x double ⟶ mesh scaleMesh: mesh x double x double ⟶ mesh	
moveVertex: mesh x vertex x double x double x double ⟶ mesh moveEdge: mesh x edge x double x double x double ⟶ mesh moveFace: mesh x face x double x double x double ⟶ mesh moveMesh: mesh x double x double x double ⟶ mesh	
unveränderte Geometrie und Topologie	**Relation**
getAllVerticesOfMesh: mesh ⟶ set of vertex getAllEdgesOfMesh: mesh ⟶ set of edge getAllFacesOfMesh: mesh ⟶ set of face	
isVertexInsideMesh: mesh x vertex ⟶ bool isEdgeInsideMesh: mesh x edge ⟶ bool isFaceInsideMesh: mesh x face ⟶ bool	
getVerticesWhichBelongToEdge: mesh x edge ⟶ set of vertex getEdgesWhichBelongToVertex: mesh x vertex ⟶ set of edge getVerticesWhichBelongToFace: mesh x face ⟶ set of vertex getFacesWhichBelongToVertex: mesh x vertex ⟶ set of face getEdgesWhichBelongToFace: mesh x face ⟶ set of edge getFacesWhichBelongToEdge: mesh x edge ⟶ set of face	ve ev vf fv ef fe
getAdjacentVerticesOfVertex: mesh x vertex ⟶ set of vertex getAdjacentEdgesOfEdge: mesh x edge ⟶ set of edge getAdjacentEdgesOfEdgeWithSameFace: mesh x edge ⟶ set of edge getAdjacentFaceOfFace: mesh x face ⟶ set of face	vv ee ee' ff
isVertexBoundingEdge: mesh x vertex x edge ⟶ bool	ve

isEdgeContainingVertex: mesh x edge x vertex ⟶ bool	ev
isVertexBoundingFace: mesh x vertex x face ⟶ bool	vf
isFaceContainingVertex: mesh x face x vertex ⟶ bool	fv
isEdgeBoundingFace: mesh x edge x face ⟶ bool	ef
isFaceContainingEdge: mesh x face x edge ⟶ bool	fe
isVertexAdjacentToVertex: mesh x vertex x vertex ⟶ bool	vv
isEdgeAdjacentToEdge: mesh x edge x edge ⟶ bool	ee
isEdgeAdjacentToEdgeAndHaveSameFace: mesh x edge x edge ⟶ bool	ee'
isFaceAdjacentToFace: mesh face x face ⟶ bool	ff

Tabelle 5.4: Gesamttabelle aller Operationen auf Polygonnetzen

5.4 Speicherungsarten für Polygonnetze

Für die Speicherung von Polygonnetzinformationen bilden Listen die Grundlage der abstrakten Datenstruktur. Listen werden verwendet um die Bestandteile des Polygonnetzes abzulegen und Beziehungen zwischen diesen herzustellen. Listen wurden in Kapitel 3 behandelt. Innerhalb dieser Datenstruktur exitieren unterschiedliche Speicherungsarten. Sie werden in diesem Abschnitt vorgestellt. Als Hilfe zur bildlichen Darstellung dienten die Unterlagen von datCG.

Die Stufe Speicherungsart konkretisiert die vorangehende Stufe Abstrakte Datenstruktur und beschreibt das Ablegen der verwendeten Daten und die dazu genutzten Strukturen. Die verschiedenen Speicherungsarten werden einzeln beschrieben. Dabei werden bei jeder Speicherungsart die verwendeten Listen aufgezeigt. Die Speicherungsarten wurden in zwei Abschnitte unterteilt. Zunächst werden die *Topologischen Speicherungsarten* vorgestellt. Sie beschreiben die topologischen Zusammenhänge innerhalb des Polygonnetzes anhand der Verzeigerung innerhalb und zwischen den verwendeten Listen. Zudem enthalten sie auch geometrische Informationen. Danach folgen *Speicherungsarten in der Hardware*. Sie beschreiben besondere Strukturen, die innerhalb der Graphik-Hardware genutzt werden.

Um ein Polygonnetz effizient speichern zu können sind Strukturen nötig, welche eine gute Verbindung zwischen dem erforderlichen *Speicherplatz* und der benötigten *Rechenzeit* herstellen. Je nachdem welche der beiden Komponenten mehr Gewicht erhält muss eine passende Speicherungsart gewählt werden.

Auch innerhalb der Speicherungsarten kann auf den vef-Graphen aus Abschnitt 5.2.2 Bezug genommen werden. So kann für die einzelnen Speicherungsarten bestimmt werden, durch welche der Relationen sie beschrieben werden. Da der vef-Graph nur topologische Zusammenhänge beschreibt wird er nur innerhalb der topologischen Strukturen betrachtet. Bei der Wahl von Relationen muss darauf geachtet werden den Speicherplatz gering zu halten und möglichst wenig Zeit zum Berechnen von nicht gespeicherten Relationen zu verwenden. Die Wahl der gespeicherten Relationen bestimmt den Speicher- und Zeitaufwand, aber auch den Aufbau der Struktur, die zur Speicherung der Daten dient.

Je nach Wahl der Relationen werden unterschiedliche Informationen abgepeichert. Diese können direkt abgefragt werden. Weitere Informationen müssen über die restlichen Relationen, die erst dann berechnet werden, wenn sie gebraucht werden, ermittelt werden. Wichtig ist bei der Wahl auch, aus welchen Relationen sich die restlichen Relationen am einfachsten ermitteln lassen und ob der Aufwand vorhersehbar ist. Die Kanten-Relationen bieten hier einen sehr guten Ansatz.

Die durch die Relationen beschriebenen Nachbarschaftsbeziehungen, sowie die innerhalb der Speicherungsarten verwendeten Listen sind auch innerhalb des in Abschnitt 5.2.2 beschriebenen Klassendiagramms zu finden. Hier sind alle möglichen Bestandteile

des Polygonnetzes und die Beziehungen zwischen diesen aufgeführt. Zusätzlich sind die verwendeten Listen eingetragen. Durch Weglassen einiger Bestandteile, deren Beziehungen und einigen Listen, oder durch Hinzufügen weiterer Informationen entstehen die verschiedenen Speicherungsarten.

5.4.1 Topologische Speicherungsarten

Die folgenden Strukturen speichern topologische Informationen in ihrem Aufbau. Die Topologie wird unter anderem in den Verbindungen zwischen den verwendeten Listen sichtbar. Einige der Strukturen speichern zusätzlich Informationen wie benachbarte Kanten und Flächen. Diese dienen einer schnellen Abfrage benötigter Zusammenhänge.

Zu Beginn wird auf zwei Varianten Bezug genommen, welche in der Auswahl der verwendeten Relationen den typischen Tradeoff zwischen Speicherplatz und Rechenzeit zeigen. Sie nutzen dabei die Aufwände der Kanten-Relationen aus, welche in Abschnitt 5.2.2 vorgestellt wurden. Anschließend werden weitere bekannte Speicherungsarten vorgestellt und versucht ihnen Relationen zuzuordnen.

5.4.1.1 Variante 1

Die *erste Variante* speichert die Relationen *ev* und *ef* ab. Die Relation ev beschreibt die Beziehung zwischen einer Kante und ihren Eckpunkten, genauer von welchen Eckpunkten die Kante begrenzt wird. ef beschreibt, dass eine Kante eine Fläche begrenzt. Das Speichern von nur zwei Relationen benötigt nur einen geringen Speicherplatz. Das Nutzen von Relationen welche sich auf Kanten beziehen hat den Vorteil, dass die Anzahl der auftretenden Paare pro Kante in jeder Relation bestimmt werden kann. Dies lässt sich darauf zurückführen, dass eine Kante stets von zwei Eckpunkten begrenzt wird und höchsten zwei Flächen begrenzt. Wie in Abschnitt 5.2.2 vorgestellt kann hier von einem Speicheraufwand von 4 Einheiten ausgegangen werden. Je zwei Einheiten pro Relation. Alle anderen Relationen können berechnet werden. Der Zeitaufwand für die Berechnung der restlichen Relationen ist jedoch sehr hoch. Um z.B. zu einem Eckpunkt alle angrenzenden Flächen zu ermitteln, sind n Zugriffe auf die Datenstruktur erforderlich. Je größer die Anzahl der enthaltenen Elemente, desto zeitintensiver ist die Berechnung.

Die verwendeten Bestandteile innerhalb dieser Speicherungsart sind Eckpunkte, Kanten und Flächen. Zu jeder der beiden Relationen werden die entsprechenden Zusammenhänge der Bestandteile gespeichert. So müssen beispielsweise für jede Kante in ev nur zwei Eckpunkte und in ef nur zwei Flächen eingetragen werden. Die Speicherung der Informationen erfolgt in Listen.

5.4.1.2 Variante 2

Die *zweite Variante* speichert die Relationen *ev, ef, ve* und *fe*. ev beschreibt die Beziehung, dass eine Kante von einem Eckpunkt begrenzt wird. Bei ef begrenzt eine Kante eine Fläche, bei ve begrenzt ein Eckpunkt eine Kante und bei fe stößt eine Fläche an

eine Kante. Die Variante 2 stellt eine Erweiterung der ersten Variante um die Relationen ve und ef dar. Diese Erweiterung soll den Zeitaufwand für die Berechnung der restlichen Relationen senken. Durch das Speichern weiterer Relationen müssen weniger fehlende Relationen ermittelt werden. Durch die zusätzlichen Relationen steigt jedoch der Speicheraufwand. Durch die Kombination der Relationen können schnell die restlichen Relationen ermittelt werden. So führt das Verknüpfen der Relationen fe und ev zur Relation vv oder das Verknüpfen der Relationen ve und ef zur Relation vf. Wichtig bei den hier verwendeten Relationen ist das die Anzahl der Kanten bei ef verschieden von der Anzahl der Kanten bei fe ist. So begrenzt bei ef die Kante höchstens zwei Flächen. Eine Fläche kann bei fe jedoch von mehreren Kanten begrenzt werden, abhängig von der Form der Fläche. Die hier verwendeten Relationen werden in Listen abgelegt.

5.4.1.3 Explizite Speicherung

Bei der *expliziten Speicherung* wird jede Face durch eine Liste ihrer Eckpunktkoordinaten repräsentiert. Das Prinzip der Speicherung der Eckpunkte zeigt Abbildung 5.37. Die Kanten sind implizit dennoch vorhanden. Zwischen jedem Paar von Ecken existiert eine Kante, auch zwischen der letzten und ersten Ecke. Die Beziehung der hier verwendeten Bestandteile kann anhand der *Relationen fv und vf* beschrieben werden. Hierbei ist der Speicherplatz gering, jedoch die Rechenzeit für die Ermittlung der restlichen Relationen hoch.

5.4.1.4 Eckenlisten

Bei der *Eckenliste* werden die Koordinaten aller Eckpunkte in einer Punktliste abgelegt. Eine Face wird als Liste von Zeigern in die Punktliste definiert. Abbildung 5.38 zeigt ein Beispiel anhand dessen die nötigen Listen und das Ablegen der Daten verdeutlicht werden soll. Wie schon bei der Expliziten Speicherung werden auch hier nur die Bestandteile Eckpunkte und Faces verwendet. Sie können ebenfalls durch die *Relationen vf und fv* beschrieben werden.

5.4.1.5 Kantenlisten

Wie bei der Eckenliste werden auch bei der Kantenliste zunächst alle Ecken in einer Punktliste gespeichert. Zusätzlich werden die Kanten in einer eigenen *Kantenliste* abgelegt. Eine Face wird dann als Liste von Zeigern in die Kantenliste definiert.

Abbildung 5.39 zeigt die nötigen Daten und Zusammenhänge auf. Es wird eine Punktliste, eine Kantenliste und eine Faceliste benötigt. In Abbildung 5.39 wird zudem beschrieben, welche Informationen die Kantenliste benötigt. So wird zu jeder Kante ein Verweis auf den Anfangs- und den Endpunkt und die Faces links und rechts der Kante gespeichert. Bei Kante a wären das (A,B,1,0), also A und B als Anfangs- und Endpunkt und 1 und 0 für die anliegenden Faces. Falls es zu einer Kante keine anliegende Face gibt, wird dies mit einem N gekennzeichnet. Es wird jeweils die Face zuerst aufgezählt, die bei der durch die Ecken definierten Durchlaufrichtung links der Kante liegt. Dies gilt, da ein Rechtssystem verwendet wird.

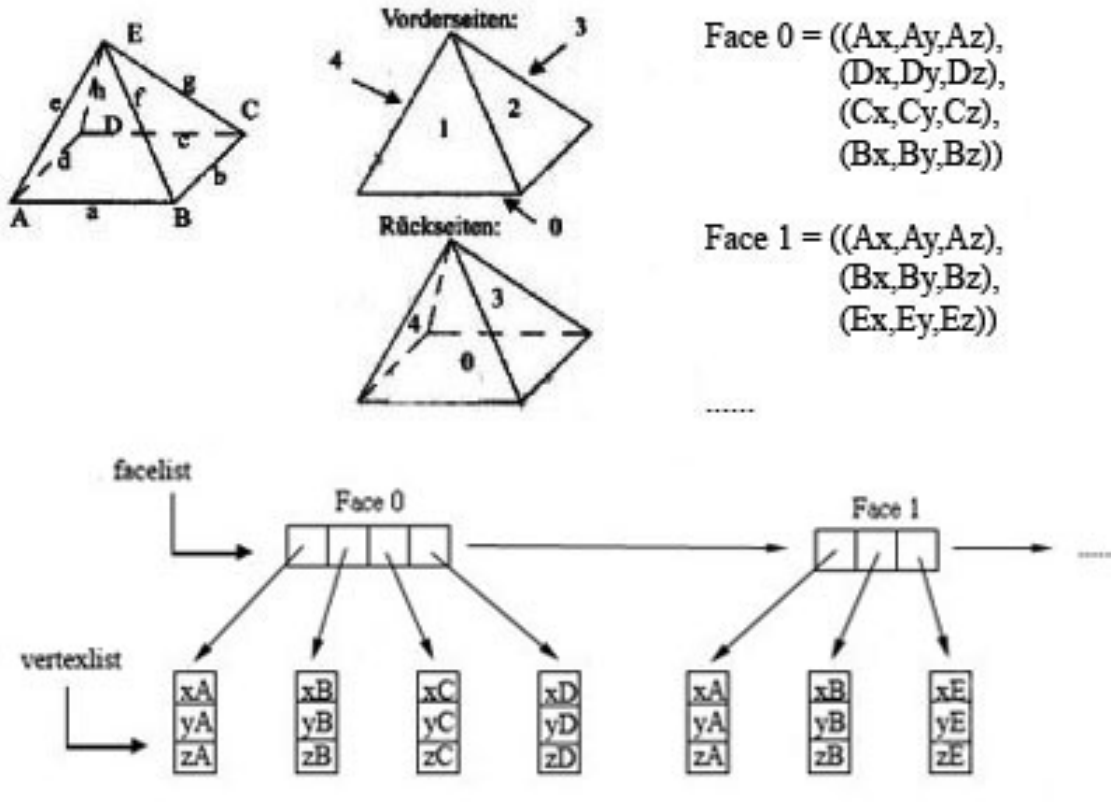


Abbildung 5.37: Beispiel einer expliziten Speicherung

Die Kantenliste kann mit der bereits beschriebenen Variante 2 verglichen werden. Sie bezieht sich auf alle Bestandteile. Da aber aus den Relationen, welche Informationen zur Kante beinhalten, auch die restlichen Relationen schnell ermittelt werden können, ist es sinnvoll nur diese abzulegen. So werden gleichzeitig Redundanzen in der Struktur vermieden. Folgende Relationen können gespeichert werden: *ev, ef, ve und fe.*

5.4.1.6 Half-Edge-Datenstruktur

Die *Half-Edge-Datenstruktur* ist ein *Flächen-orientierter Ansatz*. Sie wird in [IGD05] wie folgt beschrieben. Zunächst wird jede Kante in zwei Teile mit unterschiedlichen Richtungen aufgeteilt. Jede Face zeigt dann auf eine Liste von *Halbkanten*. Die Halbkanten sind gegen den Uhrzeigersinn orientiert. Jede Halbkante einer Kante verläuft in eine andere Richtung, also einmal von A nach B und einmal von B nach A. Das Prinzip der Halbkanten wird verwendet um die Doppelverwendung von Kanten aus Sicht der Faces zu vermeiden. Eckpunkte speichern jeweils ihre Koordinaten und einen Pointer auf die Halbkante, die diesen Eckpunkt als Startpunkt nutzt. Eine Face braucht nur auf eine der Halbkanten zu verweisen, und zwar auf die, die es umrandet. Abbildung 5.40 verdeutlicht diese Vorgehensweise und den Zusammenhang der verwendeten Listen.

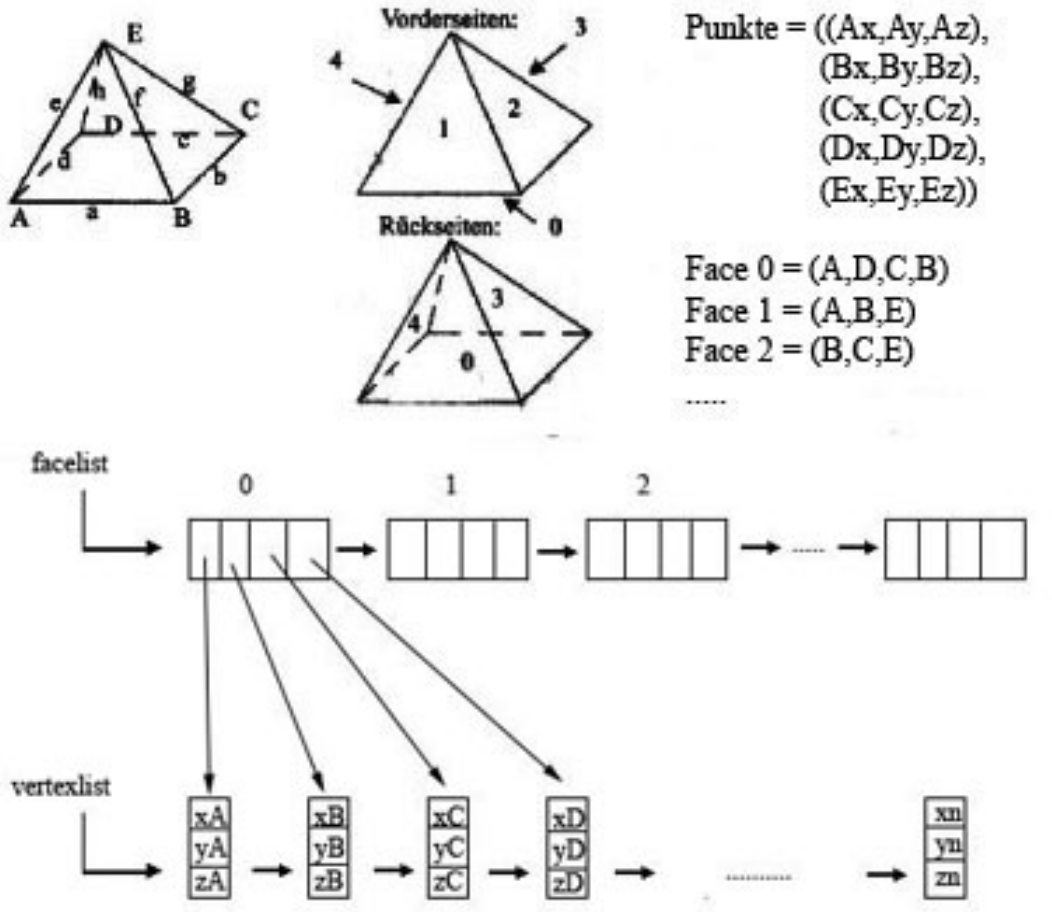


Abbildung 5.38: Beispiel einer Eckenlisten

Im Zusammenhang mit dem vef-Graphen kann die Half-Edge Struktur wie folgt beschrieben werden. Sie speichert die Relationen *ev, ef und ee"* und außerdem noch eine Punktliste und eine Flächenliste. In der Punktliste wird pro Eckpunkt ein Verweis auf eine von ihm begrenzte Kante gespeichert. In der Flächenliste wird pro Fläche ein Verweis auf eine sie begrenzende Kante abgelegt. Von den vier Kanten e1, e2, e3 und e4, die zu jeder Kante e in ee' gespeichert werden, begrenzen jeweils zwei dieselbe Fläche wie e. Abgespeichert wird jeweils die Kante, die bezüglich der entsprechenden Fläche auf e im Uhrzeigersinn folgt. Abbildung 5.41 zeigt dies. Hier sind es die Kanten e_2 und e_3. Die Relation ee" ersetzt die Relation ee', da innerhalb der Half-Edge Struktur Halbkanten gepeichert werden. Die Half-Edge Struktur braucht relativ wenig Speicherplatz im Bezug auf die Relationen. Der Zeitaufwand für die nötigen Berechnungen ist von der Form der Flächen abhängig und kann variieren.

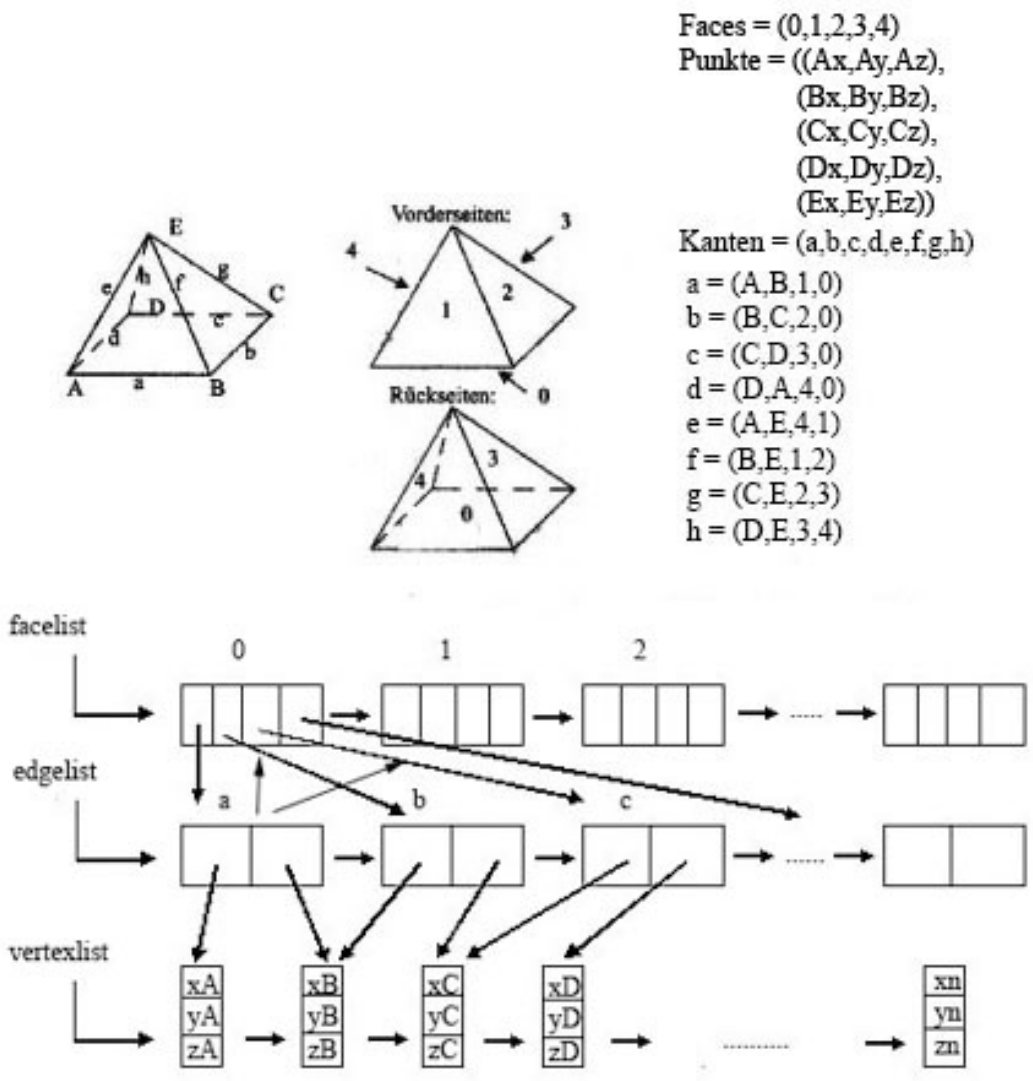


Abbildung 5.39: Beispiel zur Verdeutlichung der Kantenlisten

5.4.1.7 Winged-Edge-Datenstruktur

Die *Winged-Edge-Datenstruktur* ist ein *Kanten-orientierter Ansatz* und stellt eine *doppelt verkettete Kantenliste* dar. Zusätzlich zu den Zeigern auf Anfangs- und Endpunkt und die Face, in denen die Kante auftrifft, werden Zeiger auf die Kanten abgelegt, die von Anfangs- und Endpunkt der aktuellen Kante abgehen. D.h. genauer: zu jeder Kante gibt es einen Anfangs- und Endpunkt, die benachbarte Face links und rechts, nachfolgende Kanten links und rechts und vorausgehende Kanten links und rechts. Die Kante stellt die zentralle Struktur dar. Hier sind die meisten Verweise zu finden. Sie können über entsprechende Zeiger realisiert werden. Über diese können Nachbarschaftsinformationen schnell und einfach abgefragt werden.

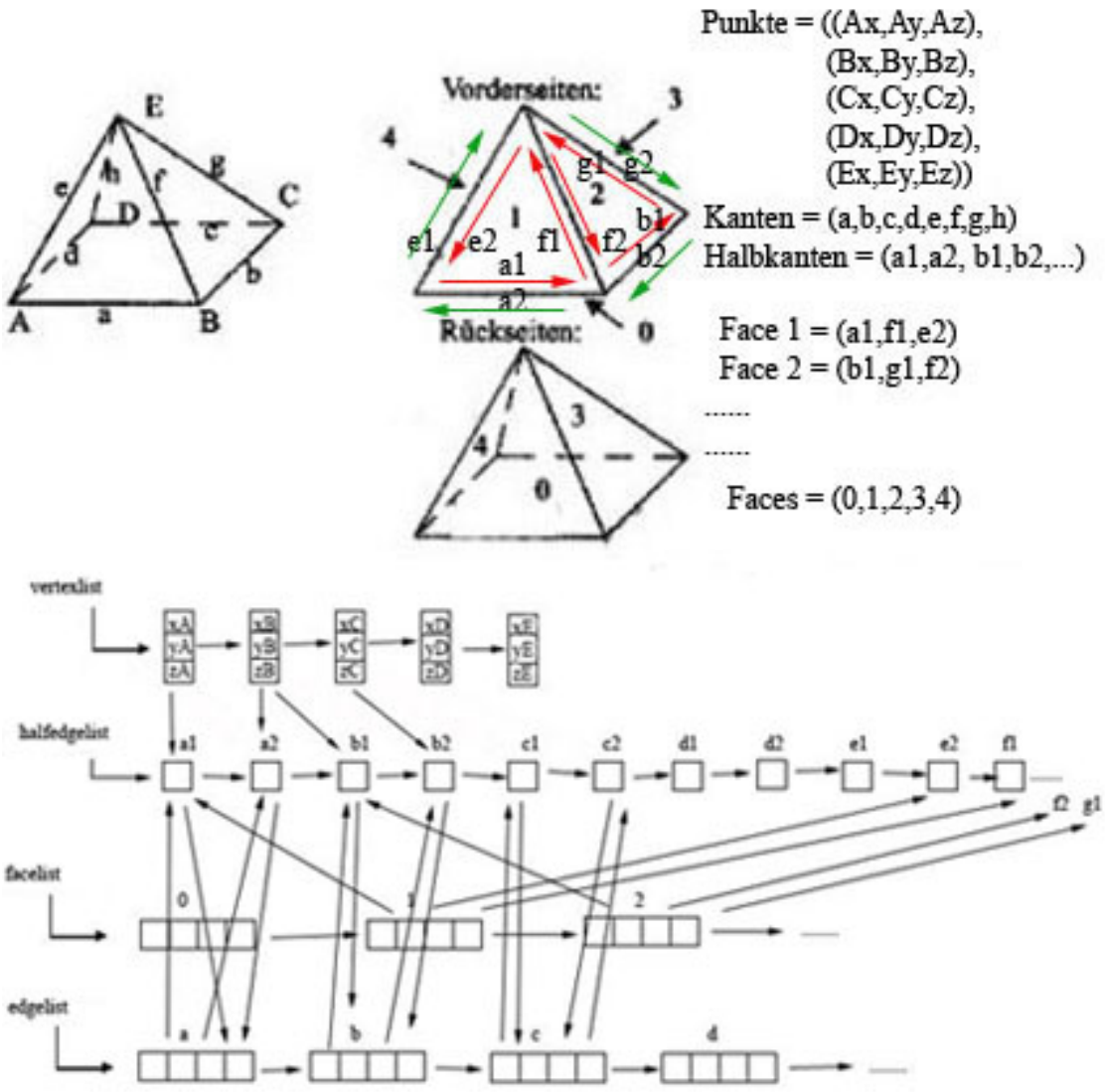


Abbildung 5.40: Beispiel einer Half-Edge Struktur

Wegen der Orientierung (gegen den Uhrzeigersinn) wird eine Kante einmal in positiver und einmal in negativer Richtung durchlaufen. Das Prinzip der Durchlaufrichtung wird in Abbildung 5.42 dargestellt. Hier wird Kante a in beiden Richtungen durchlaufen. Daraus ergibt sich die in der zugehörigen Tabelle aufgeführte Struktur. Die Kante verläuft von B nach A. Hier ist Face 0 links und Face 1 rechts. Folgt man links nun den Kanten gegen den Uhrzeigersinn so ergibt sich für die Kante a die Kante c als Vorgänger und die Kante b als Nachfolger. Rechts wird die Kante ebenfalls gegen den Uhrzeiger-

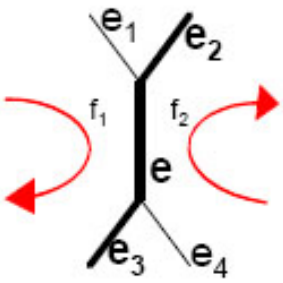
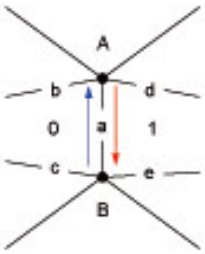


Abbildung 5.41: Half-Edge: Relation ee“, Quelle:[Nau]

sinn durchlaufen. Hier ist also der Vorgänger die Kante d und der Nachfolger die Kante e.



edge	vertex 1	vertex 2	face links	face rechts	Vorg. links	Nachf. links	Vorg. rechts	Nachf. rechts
a	B	A	0	1	c	b	d	e

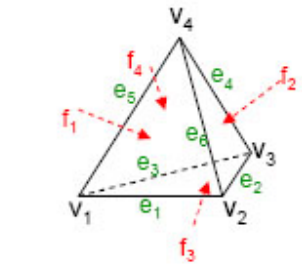
Abbildung 5.42: Windged-Edge Struktur

Die beschriebenen Zusammenhänge werden in Abbildung 5.44 verdeutlicht. Die in der Kantenliste gespeicherten Informationen sind in Abbildung 5.44 wie folgt dargestellt. Die ersten beiden Werte sind der Anfangs- und Endpunkt, danach folgen die benachbarten Flächen und anschließend die Vorgänger- und Nachfolgerkanten links und rechts.

Im vef-Graphen nutzt die Windged-Edge Struktur folgende Relationen: *ev, ef und ee'*. Zusätzlich werden eine Punktliste und eine Flächenliste verwaltet. Die Punktliste speichert pro Eckpunkt einen Verweis auf eine von ihm begrenzte Kante. Die Flächenliste speichert pro Fläche einen Verweis auf eine sie begrenzende Kante. Auch aus den Relationen wird deutlich, dass Kanten eine zentrale Rolle spielen.

Diese Struktur braucht für die verwendeten Relationen relativ wenig Speicherplatz. Jede Relation benötigt 2 Einheiten. Da eine Kante stets zwei auftretende Paare hat. Zustätzlich muss Speicherplatz für die Listen eingerechnet werden. Der Zeitaufwand kann va-

riieren, da er von der Form der Flächen abhängt. So ist die Anzahl der Kanten einer Fläche im Allgemeinen nicht konstant. Abbildung 5.43 zeigt noch einmal das Tetraeder aus Abschnitt 5.2.2 in der Winged-Edge Struktur.



ev

e_1	v_1	v_2
e_2	v_2	v_3
e_3	v_1	v_3
e_4	v_3	v_4
e_5	v_1	v_4
e_6	v_2	v_4

ef

e_1	f_1	f_3
e_2	f_2	f_3
e_3	f_3	f_4
e_4	f_2	f_4
e_5	f_1	f_4
e_6	f_1	f_2

ee'

e_1	e_2	e_3	e_5	e_6
e_2	e_1	e_3	e_4	e_6
e_3	e_1	e_2	e_4	e_5
e_4	e_2	e_3	e_5	e_6
e_5	e_1	e_3	e_4	e_6
e_6	e_1	e_2	e_4	e_5

v Liste

v_1	e_1
v_2	e_2
v_3	e_3
v_4	e_4

f Liste

f_1	e_5
f_2	e_2
f_3	e_2
f_4	e_4

Abbildung 5.43: Tetraeder: Winged-Edge, Quelle:[Nau]

5.4.2 Speicherungsarten in der Hardware

Auch in der *Graphik-Hardware* werden Polygonnetze verwendet. Eine bekannte Schnittstelle zur Graphik-Hardware stellt *OpenGL* dar und wird in [ope04] beschrieben. Es dient zur Bearbeitung und Darstellung interaktiver 3D-Anwendungen. OpenGL ist unabhängig von der verwendeten Hardware und nutzt diese zur Beschleunigung der Darstellung. Die Objekte bestehen aus einer bestimmten Anzahl an Eckpunkten und unterscheiden sich nur in der Art ihrer Verbindung. Diese werden *Primitive* genannt. Sie können in *Strips* und *Fans* gespeichert sein. Primitive wie Triangle Strips werden schneller verarbeitet als einfache Listen und können in OpenGL direkt angegeben werden. Innerhalb eines Strips oder Fans werden keine Eckpunkte doppelt verarbeitet. Um die Objekte darzustellen werden diese durch die *Graphikpipeline* geschickt. Die Graphik-Hardware (s. [Wös]) arbeitet am schnellsten wenn sich der Zustand der Graphikpipeline, also beispielsweise Farbe, Materialeigenschaften und Primitive, nicht ändert. Jede Änderung erfordert das Senden von Befehlen an die Graphik-Hardware und die Verarbeitung der Befehle durch die Graphik-Hardware.

Dreiecksnetze sind die am häufigsten verwendeten Netze. Sie erfüllen stets die Bedingungen der Einfachheit, Konvexität und Planarität. Die einfachste Möglichkeit diese

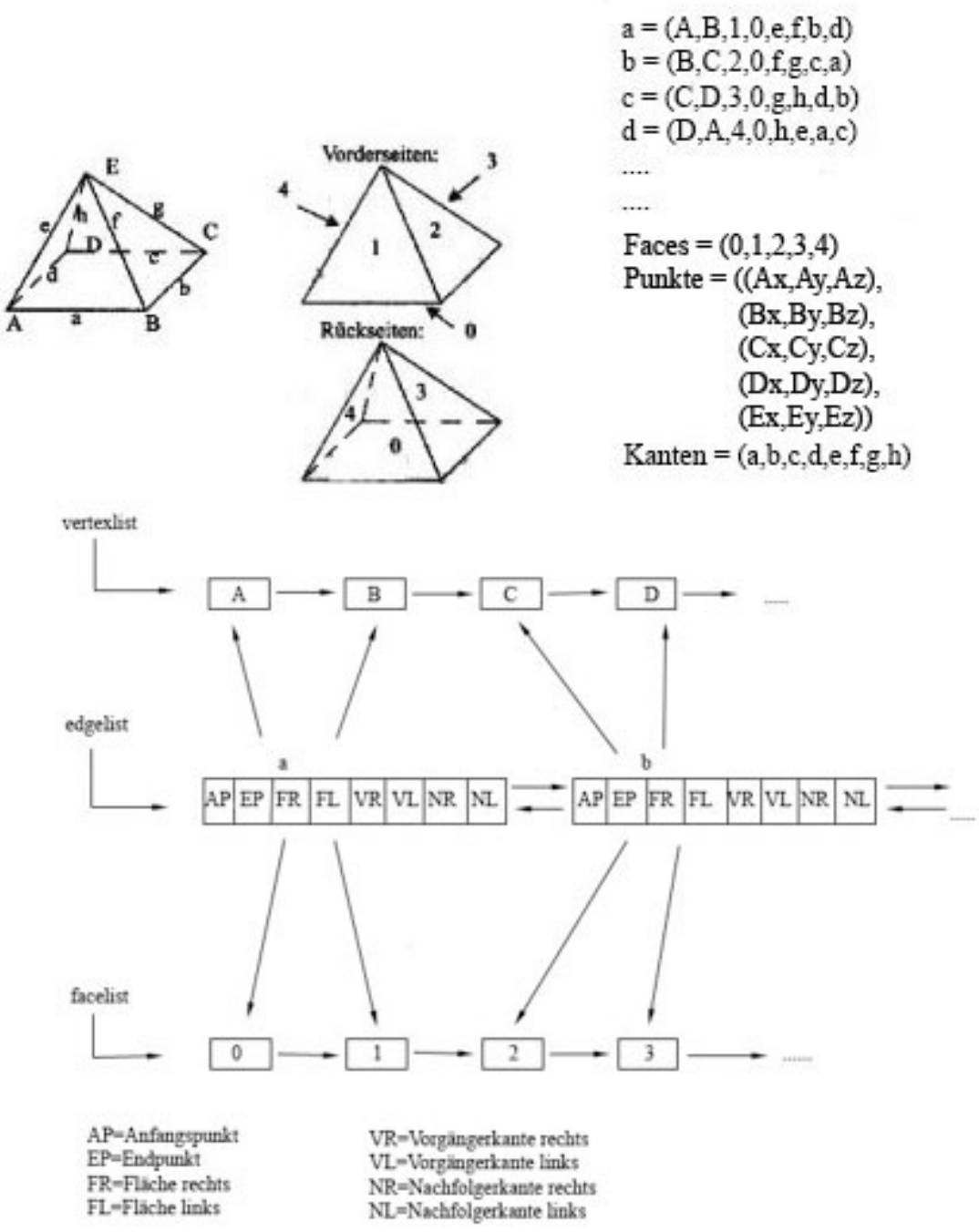


Abbildung 5.44: Beispiel der Winged-Edge Struktur

abzulegen sind Listen. Sie haben allerdings den Nachteil, dass sie jedes Dreieck in einer einzelnen Punktliste speichern und damit viele Redundanzen entstehen. Um diese

Redundanzen zu vermeiden existieren für Dreiecke und auch Vierecke besondere Speicherungsarten, die ihre Form mitberücksichtigen und diese effizient ablegen. Die Triangle Strips und Fans und die Quad Strips stellen eine Abwandlung der Eckenliste dar und werden nachfolgend beschrieben.

5.4.2.1 Dreiecks- und Viereckslisten

Die einfachste Möglichkeit Dreiecke oder Vierecke abzuspeichern ist jedes einzelne Dreieck oder Viereck extra abzulegen. Dies entspricht der Expliziten Speicherung in Abschnitt 5.4.1.3. Jedes Dreieck oder Viereck wird durch eine Punktliste repräsentiert. Dadurch werden jedoch Eckpunkte mehrfach abgespeichert.

5.4.2.2 Triangle Strips

Um die redundanten Informationen zu vermeiden gibt es mehrere Möglichkeiten. Eine davon sind *Triangle Strips*. Dies ist eine sortierte Dreiecksliste. Dabei wird darauf geachtet, dass gemeinsame Eckpunkte nur einmal gepeichert werden. Bei Triangle Strips werden jeweils drei Eckpunkte zusammengefasst. Der Anfangspunkt ist V_i. Darauf folgen V_{i-1} und V_{i-2}. Alle drei Eckpunkte bilden das Dreieck. Die beiden letzten Eckpunkte V_{i-2}, V_{i-1} werden iterativ gespeichert und mit dem darauf folgenden Eckpunkt zu einem neuen Dreieck (V_{i-2}, V_{i-1}, V_i) zusammengefügt. Anschließend wird der älteste Eckpunkt V_{i-2} verworfen und durch V_{i-1} ersetzt. V_i ersetzt V_{i-1}. Ein neuer Eckpunkt wird dann zu V_i. So bilden jeweils zwei vorhandene und ein neuer Eckpunkt, also drei aufeinander folgende Eckpunkte das Dreieck. Dies kann solange durchgeführt werden wie zusammenhängende Dreiecke vorhanden sind. Ein Triangle Strip ist in Abbildung 5.45 zu sehen.

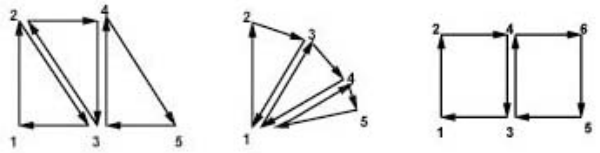


Abbildung 5.45: Links: Triangle Strip, Mitte: Triangle Fan, Rechts: Quad Strip

Abbildung 5.47 zeigt die bei Triangle Strips nötigen Listen und ihren Zusammenhang. Der Triangle Strip ist nicht für alle Formen von Dreiecksnetzen geeignet. Um dennoch die Funktion des Triangel Strip zu verdeutlichen wurde dieser auf die Faces 1 und 2 angewendet.

5.4.2.3 Triangle Fans

Eine weitere Möglichkeit des effizienten Abspeicherns von Dreiecken bieten *Triangle Fans*. Dabei wird im Prinzip wie beim Triangle Strip vorgegangen. Allerdings wird nicht

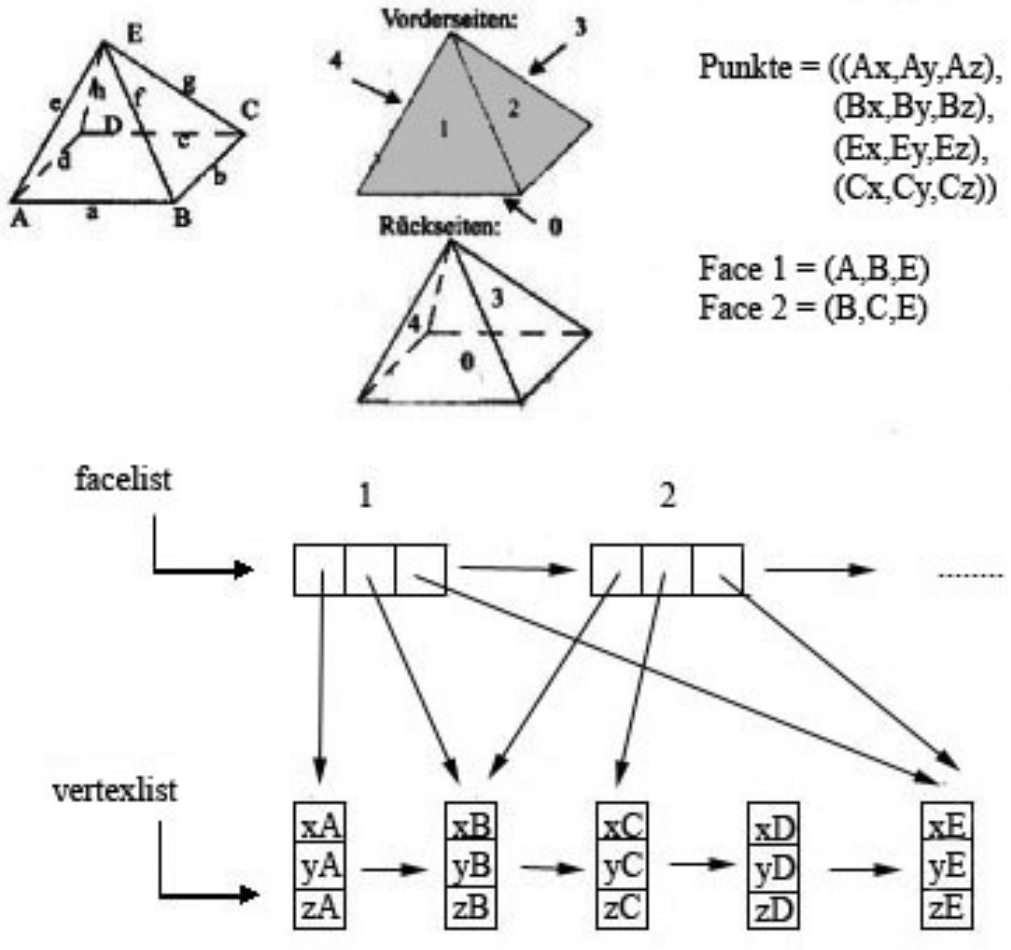


Abbildung 5.46: Triangle Strip: Struktur

der älteste Eckpunkt, sondern der Vorgänger des aktuellen Eckpunktes durch einen neuen Eckpunkt ersetzt. Dabei bleibt der zuerst übertragene Eckpunkt immer erhalten. Das Dreieck ist somit durch den ersten Eckpunkt und die jeweils zwei folgenden Eckpunkte definiert. Alle Dreiecke haben einen gemeinsamen Eckpunkt und jeweils benachbarte Dreiecke besitzen zwei gemeinsame Eckpunkte. Ein Beispiel wäre die Approximation einer Kreisfläche, wobei der gemeinsame Eckpunkt der Mittelpunkt des Kreises ist. Abbildung 5.45 stellt diese Variante der Dreiecksbildung dar.

In Abbildung 5.47 werden die benötigten Daten und Listen aufgezeigt. Als Beispiel dient eine Pyramide. Diese kann nicht komplett durch einen Triangle Fan realisiert werden. Die Fläche 0 kann nicht durch einen Triangle Fan beschrieben werden.

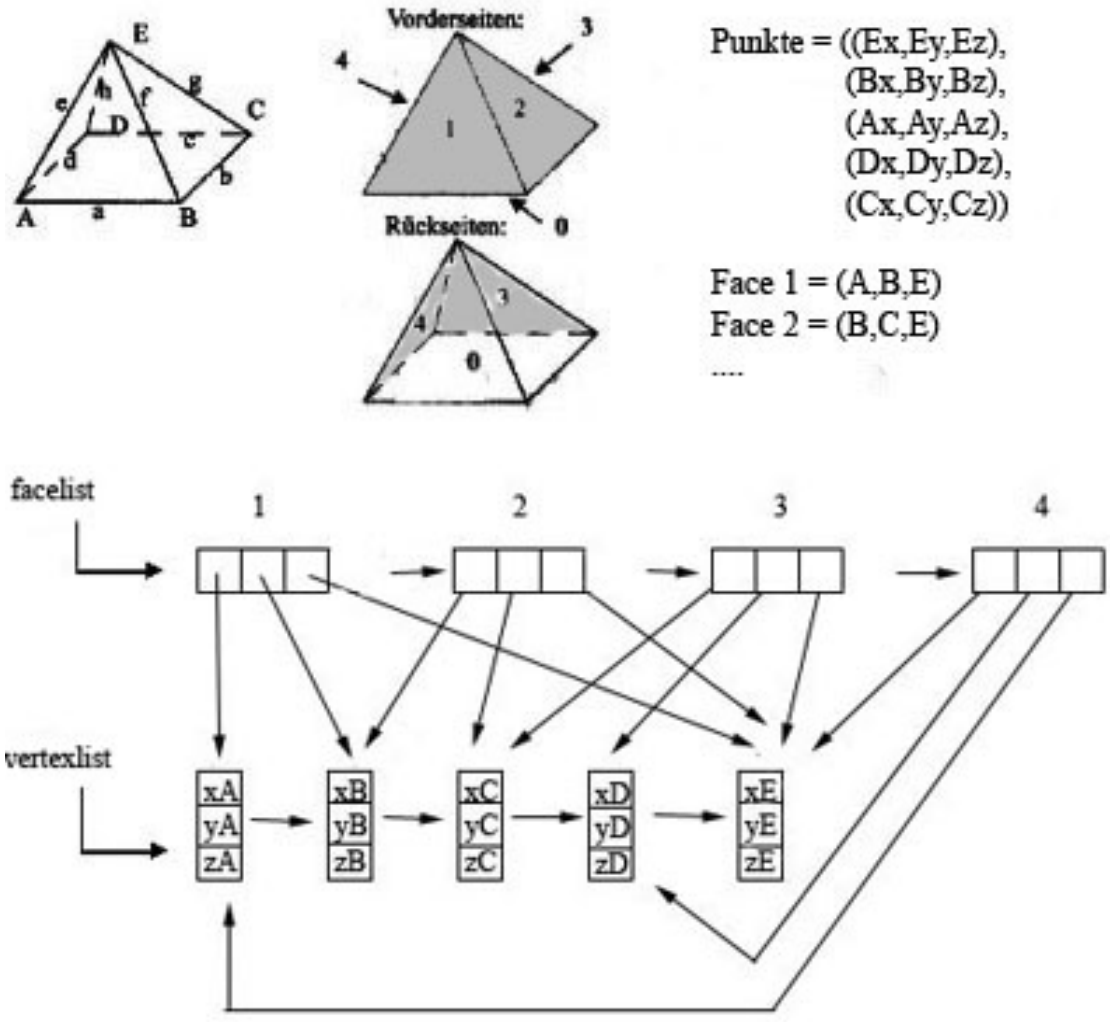


Abbildung 5.47: Triangle Fan: Struktur

5.4.2.4 Quad Strips

Quad Strips gehören zur gleichen Kategorie wie Triangle Strips und Fans. Dienen jedoch zur effizienten Speicherung von viereckigen Netzstrukturen. Quad Strips bilden eine Folge von Vierecken, wobei die letzten beiden Eckpunkte eines Vierecks gleichzeitig als die ersten beiden des nächsten Vierecks interpretiert werden. Liegt eine ungerade Zahl an Eckpunkten vor, so wird der letzte Eckpunkt ignoriert. Abbildung 5.45 zeigt einen Quad Strip. Abbildung 5.48 zeigt das Prinzip des Quad Strips. Bei einer Pyramide ist dieser nicht anwendbar. Nur die untere Fläche kann durch einen einzelnen Quad dargestellt werden.

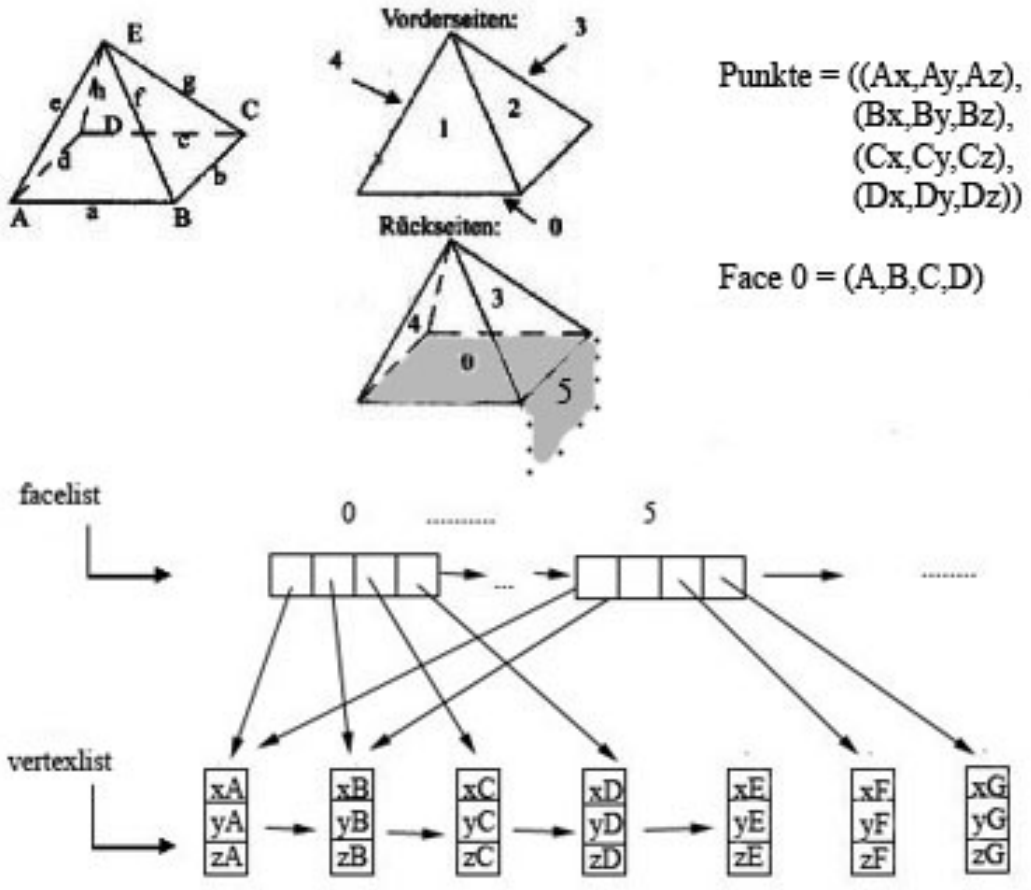


Abbildung 5.48: Quad Strip: Struktur

Wahl der Speicherungsart

Abbildung 5.49 zeigt eine Kugel anhand der die Wahl einer Speicherungsart sichtbar wird. So wird für die Kappe der Kugel ein Triangle Fan gewählt (hell grau). Für die restlichen Flächen kann ein Quad Strip genutzt werden (dunkel grau).

5.4.3 Fazit

In diesem Abschnitt wurden verschiedene Speicherungsarten vorgestellt. Diese bieten unterschiedliche Möglichkeiten der Speicherung von Daten und haben verschiedene Schwerpunkte. Die Explizite Speicherung und die Eckenliste speichern keine Informationen über Kanten. Auch die Strips und Fans gehören zu dieser Kategorie. Hier wird das Polygonnetz ausschließlich über die Eckpunkte und die Faces beschrieben. Innerhalb der Kantenliste, sowie der Half-Edge und Winged-Edge Struktur spielen Kanten dagegen eine zentrale Rolle. Zusätzlich verwaltet die Winged-Edge Struktur weitere Informationen wie Vorgänger- und Nachfolgerkanten. Dadurch ist ein schnellerer Zugriff auf diese Informa-

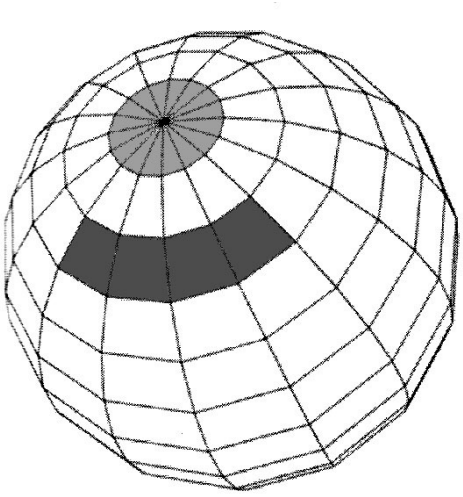


Abbildung 5.49: Kugel: Triangle Fan und Quad Srip; Quelle [Wat02]

tionen möglich. Sie können direkt abgelesen und müssen nicht erst ermittelt werden. Welche Speicherungsart gewählt wird hängt von verschiedenen Anforderungen ab. Die Hauptkriterien sind Speicherplatzbedarf, Rechenzeit und benötigte Informationen.

Bei der Wahl einer Speicherungsart ist es wichtig zu wissen welche Informationen direkt abgefragt werden sollen. Danach richtet sich welche Daten gespeichert werden und welche zunächst vernachlässigt werden. Außerdem muss darauf geachtet werden die fehlenden Informationen später ermitteln zu können. Hier ist die Rechenzeit von Bedeutung.

Vor der Wahl einer Speicherungsart sollte Folgendes geklärt werden: ist der Speicherplatz oder die Rechenzeit wichtiger und welche Informationen sind von Bedeutung.
Sind nur einfache Abfragen wie welche Eckpunkte gehören zu einer Face nötig, reicht eine Ecken- oder Kantenliste aus. Diese Informationen sind in der Struktur enthalten und können mit relativ geringem Zeitaufwand abgefragt werden. Der Speicherplatzbedarf steigt linear mit der Größe des Polygonnetzes.

Werden vor allem Nachbarschaftsinformationen wie gemeinsame Ecken und Kanten von Faces abgefragt, ist die Winged-Edge Struktur eine gute Wahl. Mit ihr lassen sich diese Informationen effizient abfragen. Sie sind über die Zeiger der Kantenliste leicht ermittelbar. Diese Struktur benötigt mehr Speicherplatz als eine Eckenliste, allerdings auch weniger Rechenzeit, da Informationen nicht erst errechnet werden müssen.

Weiterhin wurde in diesem Abschnitt der vef-Graph betrachtet. Er kann innerhalb topologischer Strukturen verwendet werden. Auch hier spielen Speicherplatz und Rechenzeit eine große Rolle. Sie richten sich nach der Art und der Anzahl der gespeicherten Relationen. Aus dem Aufbau der Speicherungsarten und den dazugehörigen Relationen lassen sich Speicherplatz und Rechenzeit ableiten. Werden nur wenige Relationen gespeichert

ist der Speicherbedarf gering. Dafür die Rechenzeit hoch. Bei vielen Relationen dreht sich dies um. Entscheidend für die Rechenzeit ist aber auch welche Relationen gespeichert werden. Die Wahl sollte auf Relationen fallen aus denen sich die restlichen Relationen leicht berechnen lassen und deren Aufwände vorhersehbar sind. Diese Relationen beziehen sich auf Kanten. Der Aufbau der Listen in welchen die Informationen gespeichert werden kann die Abfrage erleichtern indem bestimmte Zeiger gesetzt werden.

Effiziente Speicherungsarten nutzen Relationen, die auf Kanten basieren. Sie nutzen die Kenntnis über die gleichbleibenden Abhängigkeiten der Kanten aus. Auftretende Paare sind daher in jeder Relation vorherbestimmbar. So begrenzt eine Kante stets zwei Flächen und wird selbst stets von zwei Eckpunkten begrenzt. Dadurch ist der Speicheraufwand von der Anzahl der gespeicherten Relationen abhängig. Jede abgelegte Relation benötigt zwei Einheiten. Der Zeitaufwand hängt davon ab wie schnell die restlichen Relationen ermittelt werden können. Strukturen wie Winged-Edge basieren auf diesem Wissen. Hier bilden die Kanten die zentrale Strukur und verweisen auf weitere Bestandteile. Durch die vielen Verweise innerhalb der Struktur lassen sich Zusammenhänge schnell und leicht ablesen. Trotz der vielen Informationen bleibt der Speicherbedarf durch die Kantenstruktur überschaubar. Die Rechenzeit ist relativ gering, da die meisten Informationen direkt ablesbar sind.

5.5 Implementation

Zwei der im vorigen Abschnitt vorgestellten Speicherungsarten werden in diesem Abschnitt im Zusammenhang mit den Polygonoperationen aus Abschnitt 5.3.1 behandelt. Die betrachteten Speicherungsarten sind die *Explizite Speicherung* und die *Winged Edge-Struktur*. Diese beiden Speicherungsarten wurden gewählt, da die Explizite Speicherung wenige Informationen verwaltet und die Abfrage einiger Informationen sich aufwendig gestaltet. Im Gegensatz dazu speichert die Winged Edge-Struktur viele Informationen und ermöglicht durch gespeicherte Beziehungen zwischen den Bestandteilen schnelle Abfragen. Jede Operation wird innerhalb der Speicherungsart betrachtet und ihr Aufwand aufgeführt. So kann entschieden werden, welche Speicherungsart für welche Operation am besten geeignet ist.

Die in Kapitel 3 vorgestellten Operationen können zur Ermittlung des Aufwandes von Polygonoperationen herangezogen werden, da ihre Speicherungsarten ebenfalls auf Listen basieren. Die Operationen können miteinander verglichen und so der Aufwand für die Polygonoperationen ermittelt werden. Der Aufwand von Operationen, die nicht direkt den Listenoperationen entsprechen, kann aus den bereits behandelten Operationen, den verwendeten Listen, ihrem Aufbau, der gespeicherten Datenmenge oder der nötigen Ausführung abgeleitet werden. Es wird jeweils der schlechteste Fall betrachtet. Der Aufwand wird in der *O-Notation* angegeben. Die Anzahl n der Elemente in allen verwendeten Listen bestimmt den Aufwand wesentlich mit.

5.5.1 Explizite Speicherung

Die Explizite Speicherung speichert nur Informationen zu Eckpunkten und zu Faces. Die Kanten werden nicht mit verwaltet. Dennoch sind diese implizit vorhanden. Der Zusammenhang der hier verwalteten Bestandteile des Polygonnetzes, sowie der verwendeten Listen ist in Abbildung 5.51 dargestellt. So wird ein Polygonnetz durch die Faces beschrieben, die wiederum durch Eckpunkte beschrieben werden.

Die Explizite Speicherung verwaltet zwei Arten von Listen entsprechend den gespeicherten Informationen. So werden die Faces in einer *Flächenliste (facelist)* und die Eckpunkte in *mehreren Punktlisten (vertexlist)* abgelegt. Durch das Verwenden mehrerer Punktlisten entstehen Redundanzen bei der Speicherung der Eckpunkte, denn ein Eckpunkt gehört meistens zu mehreren Faces. Dadurch wird der Eckpunkt in mehreren Punktlisten abgelegt. Abbildung 5.50 verdeutlicht den Aufbau der einzelnen Listen und den Zusammenhang zwischen der Flächenliste und den Punktlisten. Eine Face in der Flächenliste wird durch die Eckpunkte in der Punktliste, auf welche sie verweist, beschrieben. Die in der Flächenliste gespeicherten Informationen über die Faces sehen wie folgt aus. Zu jeder Face wird eine `id` und ein Verweis auf die entsprechende Punktliste gespeichert. Die jeweilige Punktliste enthält die Eckpunkte der dazugehörigen Face, beschrieben durch ihre Koordinaten. Die `id` ist ein Integer-Wert, welcher die Face identifiziert. Die nicht gespeicherten Kanten können durch ein Paar von Eckpunkten beschrieben werden. Da die Eckpunkte in der Punktliste stets in der Reihenfolge abgelegt werden, wie sie auch gezeichnet werden, ist klar, dass sich zwischen zwei aufeinander folgenden Eckpunkten

und zwischen dem ersten und dem letzten Eckpunkt jeweils eine Kante befindet. Im Bezug auf die geometrischen und topologischen Informationen aus Abschnitt 5.2.3 wird durch die Punktlisten und die darin gespeicherten Koordinaten die Geometrie beschrieben und durch die Verbindungen zwischen den beiden Listen die Topologie.

Die Punktlisten und die Flächenliste können sowohl sequenziell, einfach verkettet als auch doppelt verkettet sein. In diesem Abschnitt wird nur eine Möglichkeit betrachtet. Diese sieht wie folgt aus. Die *Flächenliste* stellt eine *einfach verkettete Liste* dar. Die *Punktlisten* sind *sequentielle Listen*. Dieser Listenaufbau wurde aus folgenden Gründen gewählt. Einfach verkettete Listen sind dynamisch aufgebaut. Damit ist es möglich jederzeit ein Element in die Liste einzufügen und diese beliebig zu erweitern. Das Polygonnetz kann also stets um Faces erweitert werden. Das Abspeichern der Eckpunkte in sequenziellen Listen ist völlig ausreichend. Diese Listen sind statisch und haben eine festgelegte Länge. Da die Anzahl der Eckpunkte entweder gleich bleibt oder sich nur wenig ändert ist hier der nötige Speicherplatz bekannt.

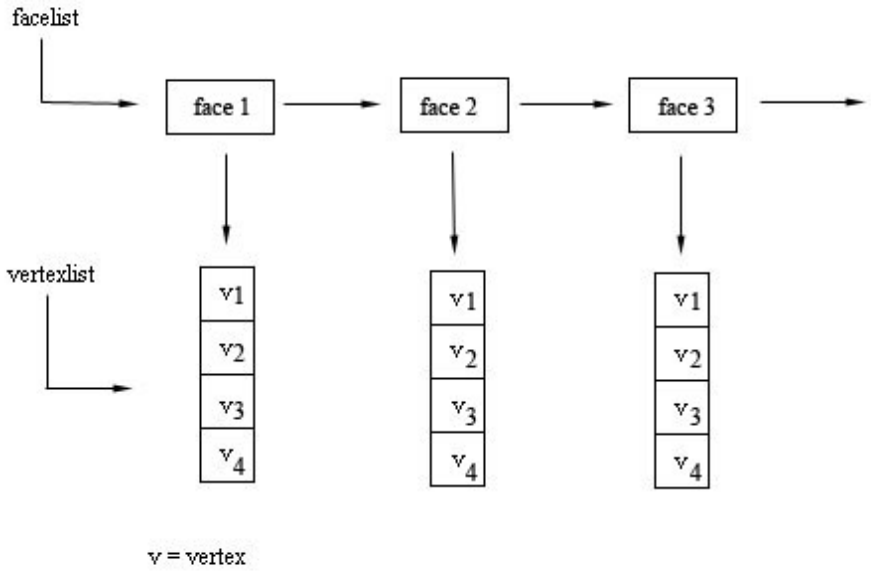


Abbildung 5.50: Explizite Speicherung: Struktur der Listen

Nachfolgend werden die in Abschnitt 5.3.1 erarbeiteten Operationen innerhalb der Expliziten Speicherung vorgestellt und ihr Aufwand ermittelt. Zu beachten ist nicht nur der jeweilige Aufbau der Listen und die Besonderheiten bei der Manipulation dieser, sondern auch die Bedingungen innerhalb eines Polygonnetzes. Hier wurden zuvor bei den Operationen verschiedene Fälle unterschieden. Diese müssen hier mitbetrachtet werden. Außerdem gelten die in Abschnitt 5.2.1 aufgeführten Bedingungen für Polygonnetze wie *Einfachheit, Planarität und Konvexität der Faces*. Innerhalb der Listen muss zusätzlich

auf die Beziehungen zwischen der Flächenliste und den Punktlisten eingegangen werden.

Die in der Signatur verwendeten Typen `vertex, edge` und `face`, sowie `set of vertex, set of edge` und `set of face` sind wie folgt definiert: `vertex` wird durch die Koordinaten x, y, z, `edge` durch ein Paar von `vertices` und `face` durch die `id` beschrieben. `set of vertex` ist eine Liste mit Eckpunkten, die durch ihre Koordinaten beschrieben werden. `set of edge` gibt jeweils eine Liste mit Paaren von Eckpunkten aus und `set of face` gibt eine Liste mit den `id's` der Faces aus.

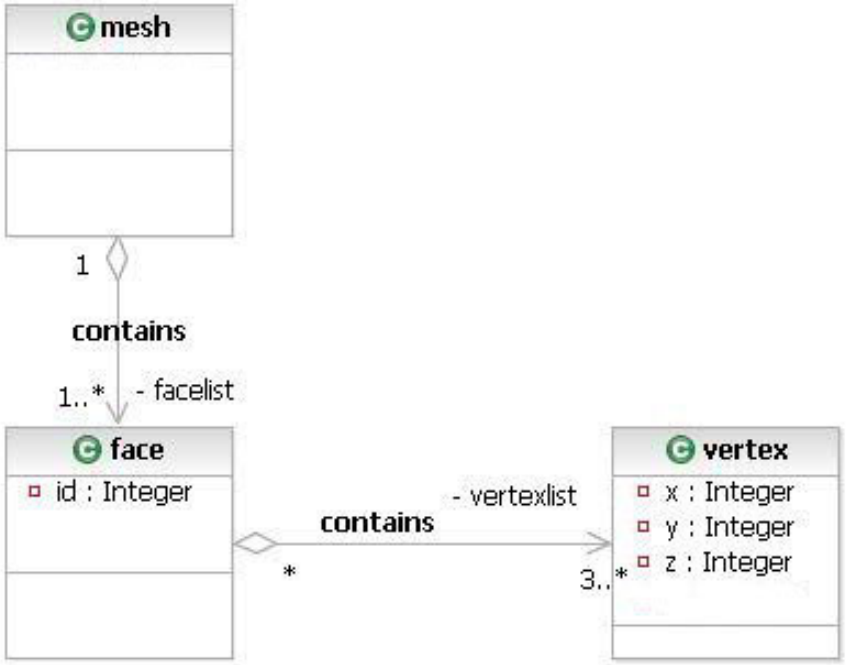


Abbildung 5.51: Explizite Speicherung: Klassendiagramm

Einfügen und Entfernen

insertVertex: mesh x vertex → mesh

In Kapitel 3 wurden verschiedene Möglichkeiten des Einfügens innerhalb einer Liste vorgestellt. So kann am Anfang der Liste, am Ende der Liste oder an einer bestimmten Position innerhalb der Liste eingefügt werden. Die Operation `insertVertex` kann mit diesen Operationen verglichen werden. Da für die Eckpunkte sequentielle Listen verwendet werden, werden diese Operationen und der entstehende Aufwand auf sequentielle Listen bezogen. Da Eckpunkte in einer bestimmten Reihenfolge gespeichert werden, ist hier nur die Operation `insertAtPos` von Bedeutung. Die restlichen Möglichkeiten werden nicht aufgelistet. Sie können jedoch in dem bereits erwähnten Kapitel über Listen nachgeschlagen werden. Diese Art des Einfügens wird am häufigsten verwendet, da einige

Anwendungen beim Zeichnen der Faces die Eckpunkte in einer bestimmten Reihenfolge benötigen. Ein Beispiel hierfür ist OpenGL. Hier werden die Eckpunkte gegen den Uhrzeiger benötigt, um anhand der Normale innen und außen der Faces zu bestimmen.

Das Einfügen eines Eckpunktes kann auf einer Kante, innerhalb einer Face oder außerhalb des Polygonnetzes erfolgen. Wird ein Eckpunkt innerhalb einer Face oder außerhalb des Polygonnetzes eingefügt, wird er mit Kanten mit dem restlichen Polygonnetz verbunden. In diesem Fall sind die Kanten nur implizit vorhanden. Das Einfügen des Eckpunktes ist nur dann möglich, wenn die dabei neu entstehenden Faces planar und konvex bleiben. Zudem muss beim Einfügen eines Eckpunktes innerhalb einer vorhandenen Face, diese Face gelöscht werden. Der Aufwand hierfür beträgt O(n) und wird später näher vorgestellt. Die neu entstandenen Faces werden in die Flächenliste eingefügt und neue Punktlisten mit ihren Eckpunkten erstellt. Das Einfügen der Eckpunkte in die Punktliste erfolgt mit einem Aufwand von O(1). Sie können hintereinander in der Liste abgelegt werden. Wird ein Eckpunkt auf einer Kante eingefügt, so wird diese Kante in zwei Kanten geteilt. Da bei der Expliziten Speicherung die Kanten nicht abgelegt werden, muss nur bekannt sein zu welchen Faces der neue Eckpunkt gehört. Er wird dann in die Punktlisten dieser Faces eingetragen. Dazu wird zunächst die Flächenliste nach den entsprechenden Faces durchsucht. Sobald eine Face gefunden wird, wird über den Verweis auf die Punktliste der Eckpunkt in diese eingefügt. Anschließend wird die Flächenliste weiter durchsucht. Dieser Vorgang wird bis zum Ende der Flächeliste fortgeführt. Die Eigenschaften der Faces werden nicht verändert.

Das Einfügen an einer bestimmten Position entspricht der Listenoperation `insertAtPos` und erfolgt über den Index der Liste. Der Eckpunkt an dieser Position und alle Eckpunkte hinter ihm werden um eine Position nach rechts verschoben und der neue Eckpunkt wird eingefügt. Durch das Verschieben der Elemente erfolgt das Einfügen an einer bestimmten Position mit einem Aufwand von O(n).

Die Operation verändert den Aufbau und den Inhalt der Punktlisten. Die Veränderungen haben jedoch keinen Einfluss auf die Flächenliste, denn jede Face verweist auf eine gesamte Punktliste. Die Flächenliste und die Beziehung zwischen den Punktlisten und der Flächenliste, also die Zeiger zwischen den beiden Listen, bleiben unverändert. Gehört der Eckpunkt zu mehreren Faces wird er in mehrere Punktlisten eingefügt, da jede Face ihre eigene Punktliste verwaltet. Der Aufwand ist beim Einfügen von dem Eckpunkt in eine Punktliste genauso hoch wie beim Einfügen in mehrere Punktlisten, da in beiden Fällen die Flächenliste durchlaufen wird und von dieser aus auf die Punktlisten zugegriffen wird.

deleteVertex: mesh x vertex → mesh

Beim Entfernen eines Eckpunktes aus den Punktlisten wird die Flächenliste durchlaufen. Bei jeder Face wird auf die dazugehörige Punktliste zugegriffen und diese auf den gegebenen Eckpunkt überprüft. Dies ist erforderlich da Eckpunkte mehrmals abgelegt werden.

Zusätzlich ist es wichtig zu wissen, ob es sich dabei um einen Eckpunkt am Rand des Polygonnetzes oder im Inneren des Polygonnetzes handelt. Bei einem Außenpunkt werden auch die Faces, die zu diesem Eckpunkt gehören mitentfernt. Das Entfernen einer Face verursacht in den meisten Fällen einen Aufwand von O(n). Bei einem Innenpunkt werden die Faces an diesem Eckpunkt zu einer zusammengefasst. Es müssen also alle Faces mit dem gegebenen Eckpunkt ermittelt werden und zu einer Face zusammengefasst werden. Dies kann durch Löschen der alten Faces und Einfügen der neuen Face geschehen. Der Aufwand hierfür beträgt O(n^2). Das Löschen innerhalb des Polygonnetzes ist jedoch nur dann möglich, wenn die neu entstehende Face planar und konvex ist.

Das Löschen eines Eckpunktes aus der Punktliste kann auf verschiedene Weisen erfolgen. In Kapitel 3 über Listen wurden an dieser Stelle vier verschiedene Möglichkeiten aufgeführt. An dieser Stelle wird das Löschen eines bestimmten Eckpunktes vorgestellt. Das Löschen eines bestimmten Eckpunktes ist im Zusammenhang mit Polygonnetzen die einzig sinnvolle Operation. Wird eine der anderen Operationen genutzt, ist nicht bekannt welcher Eckpunkt im Netz gelöscht wird.

Das Entfernen eines bestimmten Eckpunktes ist mit der Listenoperation `deleteData` vergleichbar. Es wird der zu löschende Eckpunkt angegeben, in der Punktliste gesucht und aus der Punktliste entfernt. Der Aufwand hierfür beträgt O(n).

Die Punktliste wird dabei verändert. Die Flächenliste ändert sich durch die Bedingungen innerhalb des Polygonnetzes. So müssen auch Flächen entfernt werden um das Polygonnetz zu erhalten. Wird eine Face entfernt, so wird die gesamte Punktliste mitentfernt. Das Entfernen einer Face aus der Flächenliste wird durch die Operation `deleteFace` beschrieben. Die Beschreibung dieser Operation folgt an einer späteren Stelle in diesem Abschnitt. Die Zeiger zwischen der Punktliste und der Flächenliste bleiben unverändert.

insertEdge: mesh x edge → mesh

Das direkte Einfügen von Kanten ist bei der Expliziten Speicherung nicht möglich, da keine Informationen über die Kanten verwaltet werden. Jeder Kante können aber zwei Eckpunkte, durch welche sie beschrieben wird zugeordnet werden. Kanten können entweder zwischen vorhandenen Eckpunkten im Polygonnetz eingefügt oder neu außerhalb des Polygonnetzes erstellt werden.

Beim Einfügen zwischen bereits vorhandene nicht benachbarte Eckpunkte entstehen neue Faces. Die alte Face wird geteilt. Um Faces zu teilen muss bekannt sein zu welcher Face die Eckpunkte der Kante gehören. Außerdem muss die Face mehr als drei Eckpunkte besitzen. Die Ermittlung der Face verursacht einen zusätzlichen Aufwand von O(n). Weiterhin muss die alte Face aus der Flächenliste gelöscht und die neuen Faces eingefügt werden. Das Löschen einer Face hat einen Aufwand von O(n). Das Einfügen beispielsweise am Anfang der Flächenliste hat einen Aufwand von O(1). Insgesamt ergibt sich ein Aufwand von O(n^2). Die neuen Faces erhalten die Eigenschaften der alten Face. Sie sind also ebenfalls planar und konvex.

Das Einfügen einer neuen Kante kann auch über zwei neue Eckpunkte oder über einen vorhandenen Eckpunkt und einen neuen Eckpunkt erfolgen. Dabei entstehen neue Faces, welche die neue Kante mit dem Rest des Polygonnetzes verbinden. Sie müssen planar und konvex sein. Hier müssen noch die restlichen Eckpunkte der Face ermittelt werden. Der Aufwand beträgt O(n). Die neue Face wird dann in die Flächenliste eingetragen. Das Einfügen einer Face kann beispielsweise am Anfang der Flächenliste erfolgen. In diesem Fall beträgt der Aufwand O(1). Durch das Einfügen der Face entsteht eine neue Punktliste. Das Erstellen der Punkliste erfolgt ebenfalls mit einem konstanten Aufwand, da innerhalb sequentieller Listen üblicherweise am Ende eingefügt wird. Der Aufwand beträgt dann insgesamt O(n).

Diese Operation bewirkt Änderungen innerhalb der Flächenliste und innerhalb der Punktlisten. In der Flächenliste werden die alten Faces entfernt und die neu entstandenen Faces eingefügt. Die Punktlisten der gelöschten Faces werden entfernt. Neu entstandene Faces führen zu neuen Punktlisten, da jede Face eine eigene Punktliste verwaltet.

deleteEdge: mesh x edge → mesh

Auch das direkte Entfernen von Kanten ist nicht möglich, da keine Speicherung dieser Informationen statt findet. Die Kante wird jedoch durch zwei Eckpunkte beschrieben. Es existieren zwei Fälle beim Entfernen einer Kante. Es kann entweder eine Kante am Rand des Polygonnetzes entfernt werden oder im Inneren des Polygonnetzes. Wird eine Kante innen entfernt führt das zur Verbindung von zwei Faces. Zwei benachbarte Faces mit einer gemeinsamen Kante können über zwei gemeinsame Eckpunkte ermittelt werden. Anschließend können diese beiden Faces zu einer zusammengeschlossen werden. Das Zusammenschließen von Faces ist nur dann möglich, wenn die entstehende Face planar ist. Auch wenn die Kante hier durch zwei Eckpunkte beschrieben wird bleiben diese erhalten. Dazu können die Operationen `deleteFace` und `insertFace` genutzt werden. Zuvor müssen die Eckpunkte der beiden Faces ausgelesen und die Redundanzen beseitigt werden, um sie anschließend neu speichern zu können. Das Ermitteln der Faces verursacht einen Aufwand von O(n). Das Löschen der Face hat ebenfalls einen Aufwand von O(n), beim Einfügen der neuen Face und beim Erstellen der neuen Punktliste entsteht je ein Aufwand von O(1). Insgesamt beträgt der Aufwand für das Entfernen einer Kante $O(n^2)$.
Um eine Kante am Rand des Polygonnetzes zu entfernen, muss bestimmt werden zu welcher Face die beiden Eckpunkte der Kante gehören. Beim Entfernen einer Randkante wird die Face, zu welcher sie gehört ebenfalls entfernt. Insgesamt entsteht ein Aufwand von $O(n^2)$.

Das Entfernen und Einfügen von Faces beinflusst bei der Expliziten Speicherung auch immer die Punktlisten, da die Faces durch diese beschrieben werden.

insertFace: mesh x face → mesh

Das Einfügen einer Face kann genau wie das Einfügen eines Eckpunktes auf verschiedene Weisen erfolgen. Am Anfang wurde die Flächenliste als einfach verkettete Liste festgelegt. Die Operationen werden daher auf diese Art der Listen bezogen. Das Einfügen einer Face erfolgt zunächst in die Flächenliste. Hier wird die id und der Verweis auf die Punktliste, welche die Face beschreibt, eingefügt. Durch das Einfügen einer neuen Face entsteht immer auch eine neue Punktliste. Hier werden die Eckpunkte der Face eingetragen. Die Anzahl der Eckpunkte richtet sich nach der Form der Face. Diese besteht aus mindestens drei Eckpunkten. Der Aufwand hierfür wurde bereits aufgeführt. Er muss beim Einfügen einer Face mitbeachtet werden.

Zusätzlich muss zwischen den drei verschiedenen Fällen unterschieden werden an denen die neue Face eingefügt werden kann. So kann die neue Face außerhalb des Polygonnetzes oder innerhalb des Polygonnetzes liegen. Innerhalb des Polygonnetzes kann die Face innerhalb einer Face oder innerhalb mehrerer Faces eingefügt werden. Eine Face innerhalb des Polygonnetzes teilt sich keine Eckpunkte mit den bestehenden Faces, sodass beim Einfügen in die Flächenliste nur eine neue Punktliste erstellt wird. Hier liegen die Eckpunkte der neuen Face entweder innerhalb einer anderen Face oder innerhalb mehrerer anderer Faces. Beim Einfügen innerhalb mehrerer Faces muss darauf geachtet werden, dass diese planar sind, sonst ist ein Einfügen nicht möglich. Die Eckpunkte müssen nicht in weitere Punktlisten eingetragen werden. Beim Einfügen einer Face außerhalb des Polygonnetzes entstehen weitere Faces, welche die neue Face mit dem Polygonnetz verbinden. Hier sind gemeinsame Eckpunkte verhanden, welche in mehrere Punktlisten eingetragen werden müssen. Vorher muss anhand einer zusätzlichen Operation bestimmt werden zu welchen Faces die gemeinsamen Eckpunkte gehören. Dies verursacht einen zusätzlichen Aufwand von O(n). Das Eintragen in mehrere Punktlisten hat den gleichen Aufwand wie das Eintragen in eine Punktliste, da auf die Punktliste über die Flächenliste zugegriffen wird. Diese wird durchlaufen und bei jeder betreffenden Face werden die Eckpunkte in die Punktliste eingetragen. Das Einfügen einer Face in die Faceliste kann auf verschiedene Weisen erfolgen.

Das Einfügen einer Face am Anfang der Flächenliste entspricht der Listenoperation `insertAtBegin`. Dabei wird vor den ersten Knoten in der Liste ein neuer Knoten eingefügt. Dies geschieht bei einer einfach verketteten Liste durch setzen eines Zeigers auf den bisher ersten Knoten. Diese Operation benötigt einen Aufwand von O(1).
Beim Einfügen einer Face am Ende der Flächenliste (`insertAtEnd`) muss die Flächenliste bis zum Ende durchlaufen werden. Da die Zeiger zwischen den Knoten immer nur in eine Richtung und zwar die gleiche Richtung verlaufen, kann nicht direkt auf den letzten Knoten zugegriffen werden. Dadurch ist der Aufwand hierfür O(n). Er ist von der Länge der Liste abhängig.
Das Einfügen an bestimmter Position kann mit der Listenoperation `insertAtPos` verglichen werden. Hierbei wird eine Position innerhalb der Flächenliste angegeben, an der die neue Face eingefügt werden soll. Für das Einfügen müssen die Position und der Vorgänger

bekannt sein. Die Liste wird durchlaufen und der Vorgänger ermittelt. Da nicht direkt über Zeiger auf den Vorgänger zugegriffen werden kann entsteht ein Aufwand von O(n).

Alle diese Operationen verändern die Flächenliste. Die bestehenden Punktlisten werden nicht verändert. Jedoch entsteht wie bereits erwähnt eine neue Punktliste, die die Eckpunkte der neuen Face enthält. Die bestehenden Zeiger zwischen der Flächenliste und den Punktlisten bleiben unverändert.
Bei der Flächenliste können alle Operationen gleichermaßen verwendet werden, da keine bestimmte Reihenfolge der Faces eingehalten werden muss. Wichtig ist nur die Reihenfolge ihrer Eckpunkte. Das Einfügen am Anfang der Liste verursacht den geringsten Aufwand.

deleteFace: mesh x face → mesh

Bei den Faces muss unterschieden werden, ob es sich um eine Außen- oder eine Innenface handelt. Außenfaces liegen am Rand des Polygonnetzes, Innenfaces im Inneren des Polygonnetzes. Die Eckpunkte einer Außenface werden nur gelöscht, falls sie zu keiner anderen Face gehören. Das Löschen einer Innenface führt zum Verschmelzen der Nachbarfaces. Dies kann jedoch nur dann ausgeführt werden, wenn die Planarität der Faces erhalten bleibt. Hier muss die neue Face in die Flächenliste eingetragen werden. Da bei der Expliziten Speicherung alle Eckpunkte mehrfach gespeichert werden, kann jede Face mit allen ihren Eckpunkten entfernt werden. Gehören die Eckpunkte zu weiteren Faces bleiben sie in deren Punktlisten gespeichert. Das Auffinden einer Face erfolgt über ihre id. Diese muss bekannt sein.

Das Löschen einer Face kann auf verschiedene Weisen erfolgen. Die am häufigsten verwendete Operation ist das Löschen einer bestimmten Face. Bei allen anderen Operationen ist nicht bekannt welche Face gelöscht wird.

Um eine bestimmte Face zu löschen wird die Face innerhalb der Flächenliste gesucht und entfernt. Auch hier müssen die Zeiger umgesetzt werden. Dazu sind Vorgänger und Nachfolger nötig. Der Aufwand dieser Operation beträgt O(n).

Die Operation verändert die Flächenliste. Da eine Face entfernt wird und eine Face auf eine Punktliste verweist, wird diese Punktliste gelöscht. Das Löschen der Punktliste verursacht zusätzlich noch einen Aufwand von O(n).

Rotieren, Skalieren, Translatieren

rotateVertex: mesh x vertex x double x double x double → mesh
rotateEdge: mesh x edge x double x double x double → mesh
rotateFace: mesh x face x double x double x double → mesh

Das Rotieren der Eckpunkte, Kanten und Faces geschieht indem die Koordinaten der Eckpunkte in der Punktliste geändert werden. Das Rotieren ist nur bei Dreiecksnetzen

möglich, da sonst die Planarität der Faces nicht mehr erhalten bleibt. Auch die restlichen Eigenschaften des Netzes bleiben erhalten, wenn die Einschränkungen aus Abschnitt 5.3.1 beachtet werden. Daher muss zuvor die Anzahl der Eckpunkte in den Punktlisten überprüft werden. Dies verursacht einen Aufwand von O(n).

Beim Rotieren von Eckpunkten wird die Flächenliste durchlaufen und alle Punktlisten auf welche sie verweist auf den angegeben Eckpunkt untersucht. In jeder Punktliste wird somit nach dem gegeben Eckpunkt gesucht und seine Koordinaten geändert. Der Aufwand hierfür beträgt O(n).

Das Rotieren von Kanten ist innerhalb der Expliziten Speicherung nicht direkt möglich, da Kanteninformationen nicht gespeichert werden. Da aber bekannt ist wo sich Kanten befinden und die Kanten durch ein Paar von Eckpunkten beschrieben werden, kann diese Operation wie die Operation `rotateVertex` angewendet werden. Jedoch werden hierbei zwei Eckpunkte gleichzeitig rotiert. Der Aufwand ist dann wie bei der Rotation von Eckpunkten O(n).

Das Rotieren einer Face wird über ihre Eckpunkte ausgeführt. Innerhalb der Flächenliste wird die zu rotierende Face über ihre id gesucht und über den Zeiger auf die Punktliste auf ihre Eckpunkte zugegriffen. Die Punktliste wird durchlaufen und die Koordinaten der Eckpunkte geändert. Da bei dieser Art der Speicherung die Eckpunkte mehrmals abgelegt werden, müssen auch die restlichen Punktlisten überprüft werden. Gehören die Eckpunkte zu weiteren Faces müssen auch deren Punktlisten geändert werden. Die Flächenliste wird durchlaufen und alle dazugehörigen Punktlisten auf den Eckpunkt überprüft. In diesem Fall ist der Aufwand O(n^2). Sind alle Faces bekannt zu denen der Eckpunkt gehört, so werden nur diese in der Flächenliste gesucht. Das Suchen und ändern der Koordinaten kann in diesem Fall in einem Schritt erfolgen. Dadurch beträgt der Aufwand nur O(n).

Bei allen diesen Operationen bleibt die Flächenliste und die Zeiger zwischen dieser und den Punktlisten unverändert. Der Aufbau der Punktlisten ändert sich ebenfalls nicht. Die in den Punktlisten enthaltenen Objekte werden dagegen geändert. Sie erhalten neue Koordinaten.

scaleEdge: mesh x edge x double x double → mesh
scaleFace: mesh x face x double x double → mesh

Das Skalieren eines Eckpunktes ist nicht möglich. Diese Operation würde keine Veränderung an dem Eckpunkt bewirken.

Das Skalieren einer Kante kann nicht direkt über diese Operation ausgeführt werden, da keine Kantenliste bzw. Kanteninformationen gespeichert werden. Die Operation kann durch die Operation `moveVertex` ersetzt werden, falls die Eckpunkte der Kante bekannt sind. Das Skalieren einer Kante wird durch das Verschieben von Eckpunkten ausgeführt. Diese Operation hat einen Aufwand von O(n).

Beim Skalieren einer Face wird zunächst die Face innerhalb der Flächenliste über ihre id gesucht und anschließend die Koordinaten ihrer Eckpunkte verändert, indem die neuen Koordinaten in die Punktliste eingetragen werden. Auf die Eckpunkte wird über den Zeiger auf die Punktliste zugegriffen. Da alle Eckpunkte einer Face in einer Punktliste gespeichert werden, kann die Punktliste durchlaufen und die Koordinaten der Eckpunkte geändert werden. Der Aufwand beträgt O(n). Gehört ein Eckpunkt zu mehreren Faces müssen auch in deren Punktlisten die Koordinaten dieser Eckpunkte geändert werden. Sind die Faces bekannt beträgt der Aufwand ebenfalls O(n). Müssen die Faces erst ermittelt werden erhöht sich der Aufwand auf O(n^2).

Die Zeiger zwischen der Flächenliste und den Punktlisten bleiben bei diesen Operationen unverändert. Es ändern sich nur die Daten der Objekte in den Punktlisten.

moveVertex: mesh x vertex x double x double x double → mesh
moveEdge: mesh x edge x double x double x double → mesh
moveFace: mesh x face x double x double x double → mesh

Das Verschieben eines Eckpunktes bewirkt die Änderung seiner Koordinaten. Um dies zu ändern werden alle Punktlisten nach dem gegebenen Eckpunkt durchsucht und die Koordinaten des Eckpunktes geändert. Der Zugriff auf die Punktlisten erfolgt über die Flächenliste. Diese wird durchlaufen und die Punktlisten der Reihe nach durchsucht. Dies verursacht einen Aufwand von O(n).

Das Verschieben einer Kante ist wegen fehlender Informationen nicht direkt ausführbar. Die Operation kann durch die Operation `moveVertex` ersetzt werden. Dazu muss man aber die Eckpunkte der Kante kennen. Sie wird dann auf die Endpunkte der Kante angewendet. Der Aufwand beträgt O(n).

Beim Verschieben einer Face wird zunächst die gegebene Face in der Flächenliste über ihre id ermittelt. Über den Zeiger wird auf die Punktliste zugegriffen. Dort werden die Koordinaten der Eckpunkte geändert. Der Aufwand beträgt O(n). Da ein Eckpunkt zu mehreren Faces gehören kann und diese dann mehrmals in den einzelnen Punktlisten gepeichert sind, müssen auch die restlichen Punktlisten überprüft und gegebenenfalls geändert werden. Sind alle Faces bekannt beträgt der Aufwand ebenfalls O(n). Müssen diese erst über die Eckpunkte ermittelt werden, so beträgt der Aufwand O(n^2).

Bei allen diesen Operationen bleiben die Flächenliste und die Zeiger zwischen der Flächenliste und den Punktlisten unverändert. Nur die gespeicherten Koordinaten der Objekte in den Punktlisten ändern sich.

rotateMesh: mesh x double x double x double → mesh
scaleMesh: mesh x double x double → mesh
moveMesh: mesh x double x double x double → mesh

Das Rotieren, Skalieren und Verschieben eines Polygonnetzes erfordert das Ändern aller Koordinaten der Eckpunkte. Dazu ist es nötig die Punktlisten zu durchlaufen. Sie sind über die Flächenliste erreichbar. Das Durchlaufen einer Liste und somit auch der Zeitaufwand sind von der Anzahl der Elemente abhängig. Auch der Speicheraufwand hängt von der Elementanzahl der Liste ab. Der Aufwand hierfür beträgt insgesamt O(n). Die Flächenliste wird bei diesen Operationen nicht verändert. Nur die Objektdaten der Punktlisten ändern sich. Auch die Zeiger zwischen den Listen bleiben unverändert erhalten.

get-Operationen

getAllVerticesOfMesh: mesh ⟶ set of vertex

Bei dieser Operation werden alle Punktlisten ausgegeben. Dazu wird die Flächenliste durchlaufen und alle Punktlisten über die Zeiger ermittelt. Dadurch, dass die Eckpunkte mehrmals abgelegt werden entstehen bei der Ausgabe Redundanzen. Diese müssen in einem zweiten Schritt eliminiert werden. Der Aufwand beträgt daher O(n^2). Die Flächenliste und die Punktlisten werden dabei nicht verändert. Auch die Zeiger zwischen den Listen bleiben unverändert.

getAllEdgesOfMesh: mesh ⟶ set of edge

Bei dieser Operation werden die Kanten durch Eckpunktpaare beschrieben. Die Punktliste werden als Paare von Eckpunkten ausgegeben. Es ist bekannt, dass zwischen den aufeinanderfolgenden Eckpunkten in jeder Punktliste sich eine Kante befindet. Da sich mehrere Faces eine Kante teilen können, entstehen bei der Ausgabe Redundanzen. Diese müssen entfernt werden. Die verwendeten Listen und die Zeiger zwischen den Listen bleiben unverändert. Der Aufwand für diese Operation beträgt O(n^2).

getAllFacesOfMesh: mesh ⟶ set of face

Bei dieser Operation wird die gesamte Flächenliste ausgegeben. Die Faces werden durch ihre id beschrieben. Die Flächenliste und die Punktlisten, sowie die Zeiger zwischen ihnen, bleiben unverändert erhalten. Die Operation erfordert einen Aufwand von O(n).

getVerticesWhichBelongToFace: mesh x face ⟶ set of vertex

Diese Operation sucht in der Flächenliste die entsprechende Face. Diese wird über ihre id angegeben. Über den Zeiger auf die Punktliste können die Eckpunkte abgelesen werden. Das Suchen und ausgeben der Eckpunkte erfordert einen Aufwand von O(n). Die Listen

und die Zeiger werden dabei nicht verändert.

getFacesWhichBelongToVertex: mesh x vertex ⟶ set of face

Diese Operation sucht in jeder Punktliste nach dem Eckpunkt. Dazu wird auf die Punktliste über die Flächenliste zugegriffen. Ist der Eckpunkt in der Punktliste enthalten wird die dazugehörige Face ausgegeben. Der Aufwand beträgt O(n). Die Flächenliste und die Punktlisten werden durch die Operation nicht verändert.

is-Operationen

isVertexInsideMesh: mesh x vertex ⟶ bool

Diese Überprüfung erfordert das Suchen innerhalb der Punktlisten und kann mit der Listenoperation `isIn` verglichen werden. Auf die Punktlisten wird über die Flächenliste zugegriffen. Die Operation ist von der Anzahl der Objekte in den Punktlisten abhängig und erfordert einen Aufwand von O(n). Die Punktlisten werden nur abgefragt, aber nicht verändert.

isEdgeInsideMesh: mesh x edge ⟶ bool

Eine Kante wird durch zwei Eckpunkte beschrieben. Zwischen zwei aufeinanderfolgenden Eckpunkten befindet sich eine Kante. Die Operation kann ausgeführt werden indem sie durch die Suche der beiden Eckpunkte ersetzt wird. Hier kann überprüft werden, ob die beiden Endpunkte der Kante vorhanden sind. Zusätzlich muss geprüft werden, ob sich die Eckpunkte hintereinander in der Liste befinden. Der Aufwand für das Auffinden der Eckpunkte beträgt O(n^2).

isFaceInsideMesh: mesh x face ⟶ bool

Diese Operation überprüft die Flächenliste auf das Vorhandensein der Face. Dazu wird die Liste durchlaufen und alle Faces in der Flächenliste mit der gegebenen Face verglichen. Der Aufwand beträgt hierfür O(n). Die Flächenliste und die Punktlisten werden nicht verändert. Auch die Zeiger zwischen den Listen bleiben erhalten.

isVertexBoundingFace: mesh x vertex x face ⟶ bool

Diese Operation sucht zunächst innerhalb der Punktlisten den Eckpunkt und überprüft, ob dieser Eckpunkt zu der gesuchten Face gehört. Die Punktlisten werden über die Flächenliste erreicht. Dies kann mit einem Aufwand von O(n) durchgeführt werden. Die Listen und die Zeiger bleiben dabei unverändert.

isFaceContainingVertex: mesh x face x vertex ⟶ bool

Diese Operation sucht zunächst in der Flächenliste nach der id der gegebenen Face und schaut dann in der Punktliste, die zu dieser Face gehört, ob der Eckpunkt in dieser enthalten ist. Auch hier ist der Aufwand O(n). Die Operation bewirkt keine Änderung der Flächenliste, der Punktlisten und der Zeiger.

Nachbarschaften

getAdjacentVerticesOfVertex: mesh x vertex ⟶ set of vertex
getAdjacentEdgesOfEdge: mesh x edge ⟶ set of edge
getAdjacentEdgesOfEdgeWithSameFace: mesh x edge ⟶ set of edge
getAdjacentFaceOfFace: mesh x face ⟶ set of face

isVertexAdjacentToVertex: mesh x vertex x vertex ⟶ bool
isEdgeAdjacentToEdge: mesh x edge x edge ⟶ bool
isEdgeAdjacentToEdgeAndHaveSameFace: mesh x edge x edge ⟶ bool
isFaceAdjacentToFace: mesh face x face ⟶ bool

Nachbarschaftsbeziehungen können über die Beziehung zu Kanten ermittelt werden. Nachbarpunkte haben eine gemeinsame Kante, Nachbarkanten haben einen gemeinsamen Eckpunkt und Nachbarfaces teilen sich eine Kante. Da innerhalb der Expliziten Speicherung keine Kanten gespeichert werden, können diese Operationen nicht direkt ausgeführt werden. Sie werden über die Paare der Kanten bestimmt.

Um die Nachbarpunkte eines Eckpunktes zu bestimmten müssen zunächst alle Kanten mit diesem Eckpunkt ermittelt werden. Anschließend muss in jeder Liste der Eckpunkt gefunden werden, und der Eckpunkt vor ihm in der Liste und nach ihm mit dem zweiten Eckpunkt der Kante verglichen werden. Dies erfordert einen Aufwand von O(n^2).
Gemeinsame Kanten werden bestimmt indem jede Punktliste nach den Endpunkten der gegebenen Kanten untersucht werden. Um gemeinsame Faces zu finden, muss jede Punktliste nach den beiden Eckpunkten der gegebenen Kante durchsucht werden. Beide Operationen verursachen einen Aufwand von O(n). Bei allen Operationen bleiben die Punktlisten, die Flächenliste und die Zeiger zwischen den Listen unverändert.

Kanten-Operationen

getVerticesWhichBelongToEdge: mesh x edge ⟶ set of vertex
getEdgesWhichBelongToVertex: mesh x vertex ⟶ set of edge
getEdgesWhichBelongToFace: mesh x face ⟶ set of edge
getFacesWhichBelongToEdge: mesh x edge ⟶ set of face

isEdgeContainingVertex: mesh x edge x vertex ⟶ bool
isEdgeBoundingFace: mesh x edge x face ⟶ bool
isVertexBoundingEdge: mesh x vertex x edge ⟶ bool
isFaceContainingEdge: mesh x face x edge ⟶ bool

Die direkten Abfragen bezüglich der Kanten sind nicht möglich, auch wenn diese implizit vorhanden sind. Die Explizite Speicherung verwaltet keine Informationen über die Kanten. Um die Operationen dennoch ausführen zu können, könnte man die Kanten als Punktpaare definieren und nach den Punkten der Kanten suchen.

In Tabelle 5.5 sind noch einmal alle Operationen mit ihrem Aufwand aufgeführt. Um die Tabelle übersichtlich zu halten sind nur die Aufwände der Operationen selbst aufgeführt. Aufwände, die zusätzlich durch die Bedingungen eines Polygonnetzes entstehen sind nicht aufgeführt.

Operation	**Aufwand**
Einfügen	
insertVertex: mesh x vertex ⟶ mesh	O(n)
insertEdge: mesh x edge ⟶ mesh (Pos)	O(n^2)
insertFace: mesh x face ⟶ mesh (Begin)	O(1)
insertFace: mesh x face ⟶ mesh (End)	O(n)
insertFace: mesh x face ⟶ mesh (Pos)	O(n)
Entfernen	
deleteVertex: mesh x vertex ⟶ mesh	O(n)
deleteEdge: mesh x edge ⟶ mesh	O(n^2)
deleteFace: mesh x face ⟶ mesh	O(n)
Rotieren, Skalieren, Verschieben	
rotateVertex: mesh x vertex x double x double x double ⟶ mesh	O(n)
rotateEdge: mesh x edge x double x double x double ⟶ mesh	O(n)
rotateFace: mesh x face x double x double x double ⟶ mesh	O(n^2)
rotateMesh: mesh x double x double x double ⟶ mesh	O(n)
scaleEdge: mesh x edge x double x double ⟶ mesh	O(n)
scaleFace: mesh x face x double x double ⟶ mesh	O(n^2)
scaleMesh: mesh x double x double ⟶ mesh	O(n)
moveVertex: mesh x vertex x double x double x double ⟶ mesh	O(n)
moveEdge: mesh x edge x double x double x double ⟶ mesh	O(n)
moveFace: mesh x face x double x double x double ⟶ mesh	O(n^2)
moveMesh: mesh x double x double x double ⟶ mesh	O(n)
get-Operationen	
getAllVerticesOfMesh: mesh ⟶ set of vertex	O(n^2)
getAllEdgesOfMesh: mesh ⟶ set of edge	O(n^2)
getAllFacesOfMesh: mesh ⟶ set of face	O(n)
getVerticesWhichBelongToFace: mesh x face ⟶ set of vertex	O(n)
getFacesWhichBelongToVertex: mesh x vertex ⟶ set of face	O(n)

is-Operationen	
isVertexInsideMesh: mesh x vertex ⟶ bool	O(n)
isEdgeInsideMesh: mesh x edge ⟶ bool	O(n^2)
isFaceInsideMesh: mesh x face ⟶ bool	O(n)
isVertexBoundingFace: mesh x vertex x face ⟶ bool	O(n)
isFaceContainingVertex: mesh x face x vertex ⟶ bool	O(n)
Nachbarschaften	
getAdjacentVerticesOfVertex: mesh x vertex ⟶ set of vertex	O(n^2)
getAdjacentEdgesOfEdge: mesh x edge ⟶ set of edge	O(n^2)
getAdjacentEdgesOfEdgeWithSameFace: mesh x edge ⟶ set of edge	O(n^3)
getAdjacentFaceOfFace: mesh x face ⟶ set of face	O(n^2)
isVertexAdjacentToVertex: mesh x vertex x vertex ⟶ bool	O(n^2)
isEdgeAdjacentToEdge: mesh x edge x edge ⟶ bool	O(n^2)
isEdgeAdjacentToEdgeAndHaveSameFace: mesh x edge x edge ⟶ bool	O(n^3)
isFaceAdjacentToFace: mesh face x face ⟶ bool	O(n^2)

Tabelle 5.5: Gesamttabelle aller Operationen auf Polygonnetzen innerhalb der Expliziten Speicherung und ihr Aufwand

5.5.2 Winged-Edge

Die Winged-Edge-Struktur speichert alle Bestandteile eines Polygonnetzes. Diese sind die Eckpunkte, die Kanten und die Faces. Die Kanten stehen in dieser Struktur an zentraler Stelle. Hier werden Zusatzinformationen wie Vorgänger- und Nachfolgerkanten, Faces, welche von der Kante begrenzt werden und Eckpunkte, welche die Kante begrenzen, abgespeichert. Die Bestandteile werden in verschiedenen Listen verwaltet. Diese sind eine *Eckpunktliste (vertexlist), eine Kantenliste (edgelist)* und eine *Flächenliste (facelist)*. Das Klassendiagramm in Abbildung 5.52 zeigt den Zusammenhang der Bestandteile und an welchen Stellen die Listen verwendet werden. Auch hier steht die Kantenliste im Mittelpunkt. Von ihr aus gehen Verweise zur Punktliste und zur Flächenliste. Es wird jeweils ein Verweis auf den Anfangs- und den Endpunkt jeder Kante und auf die Face links und rechts jeder Kante gespeichert. Zusätzlich werden pro Kante Verweise auf die Nachfolger- und Vorgängerkante links und rechts abgelegt. Der Aufbau der drei verwendeten Listen innerhalb der Winged-Edge-Struktur ist in Abbildung 5.53 dargestellt. Die hier dargestellten Beziehungen sind auch im Klassendiagramm erkennbar. So gehen alle Verweise von der Kante aus bzw. das Polygonnetz selbst verweist auf die zentrale Struktur, die Kante.

Die Bestandteile des Polygonnetzes werden folgendermaßen in den Listen abgelegt. Die Eckpunkte werden jeweils mit ihren Koordinaten und einer `id` in der Punktliste abgelegt. Die Faces erhalten eine `id` und eine Folge von Punktverweisen durch welche sie begrenzt werden. Die Kanten werden durch die Verweise ihrer Endpunkte, eine `id` und wie oben erwähnt weitere Informationen beschrieben. Die Punktliste enthält die geometrischen Informationen. Alle Listen, sowie die Verweise zwischen den Listen beschreiben

die topologischen Informationen, also die Zusammenhänge innerhalb des Polygonnetzes. Innerhalb der gesamten Struktur werden Geometrie und Topologie miteinander verbunden.

Die *Kantenliste* ist eine *doppelt verkettete Liste*, die *Eckpunktliste* und die *Flächenliste* sind *einfach verkettet*. Diese Listenarten sind dynamisch aufgebaut und können jederzeit beliebig erweitert werden. Zusätzlich erlaubt eine doppelt verkettete Liste den Zugriff auf Vorgänger und Nachfolger. So kann schnell innerhalb der Struktur navigiert und Informationen abgefragt werden. Dies ist innerhalb einer einfach verketteten Liste nicht möglich, da die Verzeigerung nur in einer Richtung aufgebaut ist. Sequenzielle Listen bieten zu wenig Spielraum, da sie nicht beliebig erweitert werden können.

Die nachfolgend vorgestellten Operationen werden im Zusammenhang mit der Winged-Edge-Struktur betrachtet. Es wird ihr Aufwand ermittelt und betrachtet in wie weit sich Veränderungen der Listen und der Zeiger zwischen den Listen ergeben. Wie bei allen Speicherungsarten müssen auch hier die Bedingungen des Polygonnetzes aus Abschnitt 5.2.1 beachtet werden. Hier ist vor allem darauf zu achten, dass die Faces innerhalb des Polygonnetzes *planar, einfach und konvex* bleiben. Zusätzlich müssen die Besonderheiten von Operationen auf Polygonnetzen beachtet werden. So ist es beispielsweise möglich die Bestandteile an verschiedenen Stellen des Polygonnetzes einzufügen.

Die in der Signatur verwendeten Typen `vertex, edge` und `face`, sowie `set of vertex, set of edge` und `set of face` sind wie folgt definiert: `vertex` wird durch die Koordinaten x, y, z, `edge` und `face` durch die `id` beschrieben. `set of vertex` stellt eine Liste mit Eckpunkten dar, welche durch ihre Koordinaten beschrieben werden. `set of edge` gibt jeweils eine Liste mit Kanten-`id's` aus und `set of face` gibt eine Liste mit den `id's` der Faces aus.

Einfügen und Entfernen

insertVertex: mesh x vertex → mesh

Das Einfügen eines Eckpunktes in die Punktliste kann wie in Kapitel 3 aufgeführt auf drei verschiedene Weisen erfolgen. Am Anfang der Liste entspricht der Listenoperation `insertAtBegin`, am Ende der Liste entspricht der Operation `insertAtEnd` und an einer bestimmten Position innerhalb der Liste der Operation `insertAtPos`. Da die Reihenfolge der Eckpunkte wichtig ist wird hier das Einfügen an einer bestimmten Position vorgestellt. Bei der Punktliste handelt es sich um eine einfach verkettete Liste.

Innerhalb eines Polygonnetzes kann ein Eckpunkt auf einer Kante, innerhalb einer Face oder außerhalb des Polygonnetzes eingefügt werden. Beim Einfügen eines Eckpunktes auf einer Kante entstehen zwei neue Kanten aus der alten Kante. Es muss bekannt sein auf welcher Kante sich der neue Eckpunkt befindet. Diese Kante wird aus der Kantenliste entfernt und dafür zwei neue Kanten eingefügt. Außerdem müssen die Verweise der alten Kante gelöscht und neue Verweise auf die Punktliste und die Flächenliste einge-

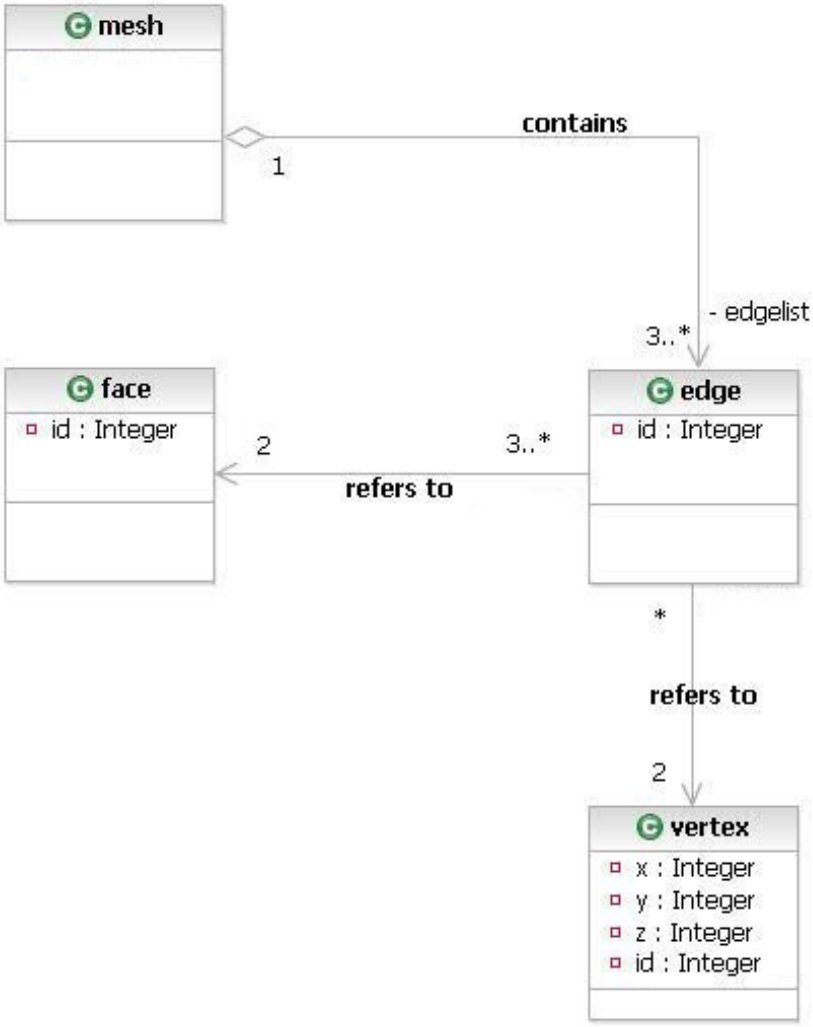


Abbildung 5.52: Aufbau der Windge-Edge-Struktur: Klassendiagramm

tragen werden. Das Löschen und Einfügen einer Kante wird später behandelt. In den meisten Fällen entsteht jedoch ein Aufwand von O(n). Die Bestandteile können über ihre id ermittelt werden.

Durch das Einfügen innerhalb einer Face oder außerhalb eines Polygonnetzes entstehen neue Faces und neue Kanten, da der eingefügte Eckpunkt mit Kanten verbunden wird. Die neu entstandenen Faces und Kanten müssen in die Kantenliste und die Flächenliste eingetragen werden. Die neuen Faces bleiben planar, wenn der eingefügte Eckpunkt in der gleichen Ebene liegt wie die Eckpunkte mit denen er verbunden wird. Die konvexe Form bleibt erhalten. Das Eintragen dieser Bestandteile geschieht normalerweise am Anfang der jeweiligen Liste. Der Aufwand hierfür beträgt jeweils O(1). Die hierzu nötigen Operationen werden später vorgestellt. Beim Einfügen innerhalb einer Face muss

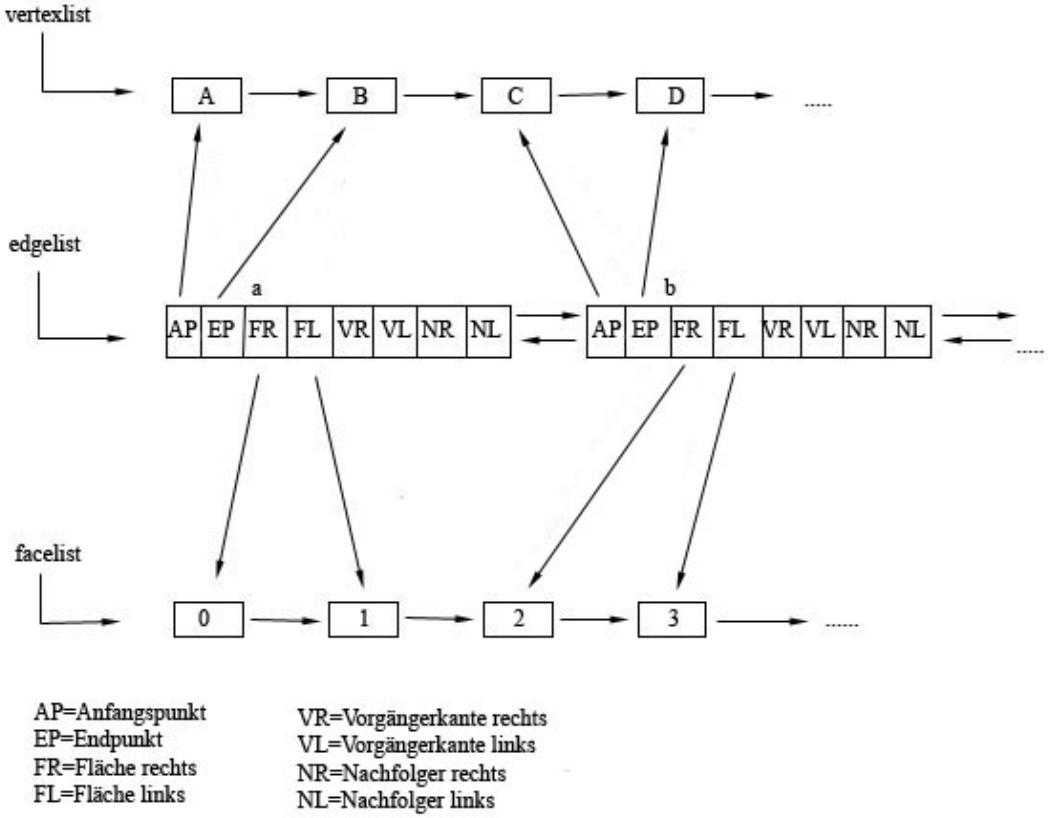


Abbildung 5.53: Winged-Edge: Struktur der Listen

zusätzlich die alte Face gelöscht werden. Zudem müssen neue Zeiger von der Kantenliste auf die Punktliste und die Flächenliste gesetzt werden. So müssen die neuen Kanten auf ihre Endpunkte und die Faces, welche sie begrenzen verweisen. Diese Verweise werden in der Kantenliste als Zusatzinformationen mitabgelegt.

Das Einfügen an einer bestimmten Position innerhalb der Punktliste benötigt die Angabe der Position in der Liste. Anschließend wird an diese Position traversiert und dort der neue Knoten mit dem Eckpunkt eingefügt. Zusätzlich müssen die Zeiger umgesetzt werden um den neuen Knoten in die Punktliste zu integrieren. Dazu muss der Vorgänger des neuen Knotens ermittelt werden. Die Operation benötigt einen Aufwand von O(n).

Die Operation verändert die Punktliste und die Kantenliste. Wird innerhalb einer Face oder außerhalb des Polygonnetzes eingefügt, so ändert sich auch die Flächenliste. Die neuen Eckpunkte müssen den Kanten zugeordnet und neue Verweise ausgehend von der Kantenliste gesetzt werden. Das Zuordnen der Eckpunkte erfordert nochmals einen Aufwand von O(n).

insertEdge: mesh x edge → mesh

Das Einfügen einer Kante kann innerhalb oder außerhalb eines Polygonnetzes erfolgen. Innerhalb des Polygonnetzes wird die Kante zwischen zwei bestehenden Eckpunkten eingefügt. Dadurch werden die alte Face bzw. mehrere Faces geteilt und neue Faces entstehen. Die neuen Faces haben wie die alte Face die Eigenschaften Planarität und Konvexität. Sie werden in die Flächenliste eingetragen. Die alten Faces werden gelöscht. Der Aufwand für das Einfügen und Entfernen einer Face wird später aufgelistet. Außerdem müssen Verweise von der neuen Kante zur Flächenliste und zur Punktliste gesetzt werden. Sie beschreiben die Endpunkte der Kanten und die angrenzenden Faces. Wird die Kante außerhalb des Polygonnetzes eingefügt, muss sie mit dem Polygonnetz verbunden werden. Dies geschieht über weitere Kanten. Dabei entsteht auch eine neue Face. Diese muss planar und konvex sein. Die neuen Kanten müssen ebenfalls in die Kantenliste und die Face in die Flächenliste eingetragen werden. Auch hier müssen die Verweise zwischen den Bestandteilen gesetzt werden. Jede Kante verweist auf ihre Eckpunkte und die Faces welche sie begrenzt. Sind die Eckpunkte der neuen Kante nicht in der Punktliste vorhanden, müssen sie neu eingefügt werden. Da die Eckpunkte an bestimmter Position eingefügt werden entsteht ein Aufwand von O(n).

Die Kantenliste ist eine doppelt verkettete Liste. Das Einfügen einer Kante kann an verschiedenen Stellen innerhalb der Kantenliste geschehen. Zusätzlich werden weitere Informationen wie Vorgänger- und Nachfolgerkanten und Nachbarfaces in Form von Verweisen eingetragen.

Das Einfügen einer Kante am Anfang der Kantenliste erfolgt mit einem Aufwand von O(1). Hierbei wird ein neuer Knoten der die neue Kante enthält an den Anfang der Kantenliste gesetzt und mit einem Zeiger an die bisherige Liste angefügt.
Das Einfügen am Ende der Kantenliste erfordert ebenfalls nur einen Aufwand von O(1). Bei einer doppelt verketteten Liste kann direkt auf das Ende der Liste zugegriffen werden, da die Zeiger in beide Richtungen verlaufen.
Das Einfügen an einer bestimmten Position innerhalb der Liste erfordert dagegen einen Aufwand von O(n). Hier wird bis zur gegebenen Position traversiert. Anschließend wird der neue Knoten an dieser Position eingefügt und die Zeiger umgesetzt.

Bei diesen Operationen wird die Kantenliste verändert. Da eine neue Kante aus zwei Eckpunkten besteht ändert sich auch die Punktliste, falls die Eckpunkte der Kante noch nicht vorhanden sind und neu eingetragen werden müssen. Das Eintragen der neuen Eckpunkte verursacht einen Aufwand von O(n). Sind sie bereits vorhanden werden nur Zeiger auf diese Eckpunkte gesetzt. Außerdem müssen Zeiger von der Kante auf die Flächenliste gesetzt werden, um die neue Kante einer Face zuzuordnen. Das Setzen der Zeiger erfolgt im gleichen Schritt mit dem Einfügen. Dadurch entsteht kein zusätzlicher Aufwand.

insertFace: mesh x face → mesh

Das Einfügen einer Face in ein Polygonnetz kann an verschiedenen Stellen erfolgen. So kann die neue Face innerhalb oder außerhalb des Polygonnetzes liegen. Innerhalb eines Polygonnetzes kann sie sich innerhalb einer Face oder innerhalb mehrerer Faces befinden. Innerhalb mehrerer Faces kann die neue Face nur eingefügt werden, wenn diese planar sind. In beiden Fällen werden Kanten gezogen, um die Face mit dem Polygonnetz zu verbinden. Dadurch entstehen neue Faces. Sie werden in die Flächenliste eingetragen. Die alte Face bzw. die alten Faces existieren nicht mehr. Sie müssen aus der Flächenliste gelöscht werden. Der Aufwand für das Eintragen und Entfernen der Faces beträgt O(n). Die entstandenen Kanten werden in die Kantenliste eingefügt. In den meisten Fällen ist dazu ein Aufwand von O(n) erforderlich.
Liegt die neue Face außerhalb des Polygonnetzes wird sie mit diesem verbunden. Hierbei entstehen neue Faces. Diese bestehen aus Kanten und Eckpunkten, welche ebenfalls in die Listen eingetragen werden müssen und planar und konvex sein müssen. Das Eintragen neuer Bestandteile verursacht im schlechtesten Fall einen Aufwand von O(n). Zusätzlich müssen neue Zeiger zwischen den Listen gezogen werden. Die neuen Kanten verweisen auf die Faces in der Flächenliste, welche sie begrenzen, und auf die Eckpunkte in der Punktliste von welchen sie begrenzt werden.

Das Einfügen am Anfang der Flächenliste verursacht einen Aufwand von O(1). Hierbei wird die neue Face an den Anfang der Liste mit Hilfe eines Zeigers angehängt.
Das Einfügen am Ende der Flächenliste erfordert das Durchlaufen der Flächenliste bis zum Ende der Liste, da nicht direkt auf das Ende der Liste zugegriffen werden kann. Die neue Face wird hinter der letzten Face in der Flächenliste eingefügt. Der Aufwand für diese Operation beträgt O(n).
Das Einfügen einer Face an einer bestimmten Position erfordert einen Aufwand von O(n). Hierbei wird die Flächenliste bis zur gegebenen Position durchlaufen und der neue Knoten mit der Face mittels Umsetzen von Zeigern eingefügt. Da in einer einfach verketteten Liste nicht direkt auf den Vorgänger zugegriffen werden kann, muss dieser ermittelt werden, um den Zeiger auf den neuen Knoten setzen zu können.

Alle diese Operationen verändern die Flächenliste. Da eine Face durch Kanten und Eckpunkte beschrieben wird muss geprüft werden, ob diese bereits vorhanden sind. Ist dies der Fall werden Zeiger auf diese gesetzt. Sind sie nicht vorhanden, müssen sie in die entsprechenden Listen eingefügt werden. Dadurch ändern sich auch die Kantenliste und die Punktliste. Das Eintragen neuer Eckpunkte und neuer Kanten verursacht jeweils einen Aufwand von O(n).

deleteVertex: mesh x vertex → mesh

Ein Eckpunkt, der gelöscht wird, kann am Rand oder im Inneren des Polygonnetzes liegen. Er kann über seine Koordinaten angegeben werden. Wird ein Eckpunkt am Rand entfernt, so wird auch die Face zu welcher er gehört mitentfernt. Das Entfernen einer Face verursacht einen Aufwand von O(n). Liegt der Eckpunkt im Inneren des Polygon-

netzes so werden alle Kanten, welche an ihm hängen entfernt. Dadurch entsteht aus den Faces an diesem Eckpunkt eine Face. Sie muss planar sein, um diese Operation ausführen zu können. Das Entfernen der Kanten, sowie das Löschen einer Face hat einen Aufwand von O(n). Das Eintragen einer neuen Face verursacht am Anfang der Flächenliste einen Aufwand von O(1). Beim Entfernen der Bestandteile werden alle Verweise dieser Bestandteile mitentfernt. Es werden neue Verweise auf die neue Face gesetzt. Das Löschen eines Eckpunktes kann auf verschiedene Weisen erfolgen. Hier wird das Löschen eines bestimmten Eckpunktes vorgestellt. Bei den anderen Operationen ist nicht bekannt welcher Eckpunkt gelöscht wird.

Das Löschen eines bestimmten Eckpunktes verursacht einen Aufwand von O(n). Hierbei wird die Punktliste nach dem Eckpunkt durchsucht und dieser entfernt.

Alle diese Operationen verändern die Punktliste, die Kantenliste und die Flächenliste. Je nachdem welche Bestandteile gelöscht bzw. eingefügt werden. Zusätzlich müssen die Zeiger entfernt bzw. an die neuen Bestandteile angepasst werden.

deleteEdge: mesh x edge → mesh

Eine Kante kann am Rand oder im Inneren des Polygonnetzes entfernt werden. Die Kante wird über ihre id angegeben. Wird eine Kante innen entfernt entsteht aus zwei Faces eine Face. Diese muss planar und konvex sein. Somit müssen die alten Faces aus der Flächenliste gelöscht (Aufwand O(n)) und die neue Face in die Flächenliste eingetragen werden (Aufwand O(1)). Die Faces können über ihre id ermittelt werden. Ist diese nicht bekannt müssen die Faces entweder über ihre Kanten oder Eckpunkte gefunden werden. Ist das Ermitteln der Faces erforderlich, entsteht ein zusätzlicher Aufwand von O(n). Außerdem werden die Verweise der entfernten Kante mitentfernt und neue Verweise gesetzt. Beim Entfernen einer Kante am Rand des Polygonnetzes wird auch die Face welche sie begrenzt entfernt. Das Entfernen einer Face aus der Flächenliste verursacht einen Aufwand von O(n). In beiden Fällen müssen auch die Eckpunkte der Kante, durch welche sie begrenzt wird, entfernt werden, falls sie zu keiner anderen Face gehören. Das Entfernen der Eckpunkte weist einen Aufwand von O(n) auf. Das Entfernen einer Kante ist auf vier verschiedene Weisen möglich wie in Abschnitt 3 bereits vorgestellt wurde. Hier wird das Entfernen einer bestimmten Kante vorgestellt, da sonst nicht bekannt ist welche Kante entfernt wird.

Das Entfernen einer bestimmten Kante erfordert einen Aufwand von O(n). Die Kante wird in der Kantenliste gesucht und aus dieser entfernt.

Die Operation verändert die Kantenliste und die Flächenliste. In den meisten Fällen auch die Punktliste. Und zwar immer dann wenn die Eckpunkte zu keiner anderen Kante gehören. Da eine Kante aus zwei Eckpunkten besteht, müssen die Verweise auf diese beiden Eckpunkte in der Punktliste mitentfernt werden. Auch die Verweise auf die Faces müssen an die neuen Gegebenheiten angepasst werden.

deleteFace: mesh x face → mesh

Die zu löschende Face kann sich entweder am Rand des Polygonnetzes befinden oder in seinem Inneren. Sie kann anhand der mitgespeicherten id angegeben werden. Bei einer Außenface, also am Rand liegend, muss geprüft werden, ob die Eckpunkte noch zu weiteren Faces gehören. Falls ja, wird die gesamte Face entfernt, aber die Eckpunkte, welche noch zu anderen Faces gehören, bleiben in der Punktliste erhalten. Die Überprüfung erfordert einen zusätzlichen Aufwand von O(n). Das Löschen der Eckpunkte hat ebenfalls einen Aufwand von O(n). Bei einer Innenface werden die anliegenden Faces zu einer zusammengeschlossen. Dies ist jedoch nur möglich, wenn die neuen Faces planar und konvex bleiben. Die angrenzenden Faces werden gelöscht. Sie können über ihre Kanten ermittelt werden (Aufwand O(n)). Die neue Face wird in die Flächenliste eingetragen. Am Anfang der Liste erfordert dies einen Aufwand von O(1).

Das Löschen einer bestimmten Face erfordert das Durchsuchen der Flächenliste nach dieser Face. Diese Operation hat einen Aufwand von O(n).

Die Operation verändert die Flächenliste. Aber auch die Kantenliste und die Punktliste. Da eine Face immer durch Kanten und Eckpunkte beschrieben wird müssen auch diese entfernt werden, falls sie zu keiner anderen Face gehören. Das Entfernen dieser Bestandteile verursacht einen Aufwand von O(n). Außerdem werden die Verweise zwischen den Listen entfernt bzw. neu gesetzt.

Rotieren, Skalieren, Translatieren

rotateVertex: mesh x vertex x double x double x double → mesh
moveVertex: mesh x vertex x double x double x double → mesh

Das Rotieren eines Eckpunktes ist nur innerhalb von Dreiecksnetzen möglich, da die Faces sonst nicht planar bleiben. Dies muss vorher überprüft werden und verursacht einen Aufwand von O(n).

Ein Eckpunkt wird rotiert indem seine Koordinaten verändert werden. Dazu wird der Eckpunkt in der Punktliste anhand seiner id bestimmt und seine Koordinaten verändert. Das Suchen innerhalb einer einfach verketteten Liste erfordert einen Aufwand von O(n). Dabei bleiben alle Listen und Zeiger zwischen den Listen unverändert. Nur die Objekte der Punktliste ändern sich.

Um einen Eckpunkt zu verschieben wird dieser innerhalb der Punktliste ermittelt und seine Koordinaten geändert. Dies verursacht einen Aufwand von O(n). Dabei werden die Objekte der Punktliste, aber nicht die Liste an sich verändert. Die Zeiger zwischen den Listen bleiben unverändert erhalten.

rotateEdge: mesh x edge x double x double x double → mesh
scaleEdge: mesh x edge x double x double → mesh
moveEdge: mesh x edge x double x double x double → mesh

Bevor rotiert wird, muss die Anzahl der Eckpunkte oder Kanten überprüft werden, da die Rotation nur innerhalb von Dreiecksnetzen möglich ist. Dies verursacht einen Aufwand von O(n).

Das Rotieren einer Kante wird über ihre Eckpunkte ausgeführt. Deren Koordinaten werden verändert. Dazu wird in der Kantenliste die Kante über ihre id ermittelt und über die Zeiger auf die Punktliste auf ihre Endpunkte zugegriffen. Dort werden dann die Koordinaten der Eckpunkte geändert. Die Operation ändert keine Listen und keine Zeiger. Es ändert sich nur der Inhalt der Punktliste. Der Aufwand dieser Operation beträgt O(n).

Das Skalieren einer Kante wird ebenfalls über ihre Endpunkte ausgeführt. Die Kante wird zunächst in der Kantenliste über ihre id bestimmt. Anschließend wird über die Zeiger auf ihre Endpunkte zugegriffen und die Koordinaten der Eckpunkte in der Punktliste geändert. Der Aufwand dieser Operation beträgt O(n). Alle Zeiger bleiben unverändert.

Auch das Verschieben einer Kante hat einen Aufwand von O(n). Hier wird ebenfalls zunächst die Kante in der Kantenliste ermittelt und anschließend die Koordinaten der Eckpunkte in der Punktliste geändert.

Die Operationen ändern keine der Listen. Nur die in der Punktliste gespeicherten Informationen werden verändert.

rotateFace: mesh x face x double x double x double → mesh
scaleFace: mesh x face x double x double → mesh
moveFace: mesh x face x double x double x double → mesh

Eine Face kann nur rotiert werden, falls sie ein Dreieck bildet. Dies kann anhand der Anzahl der Kanten oder der Eckpunkte überprüft werden und verursacht einen zusätzlichen Aufwand von O(n).

Um eine Face zu rotieren müssen die Koordinaten ihrer Eckpunkte in der Punktliste geändert werden. Die Face wird anhand ihrer id ermittelt. Da innerhalb der Kantenliste Informationen über die Faces gespeichert werden, können die Faces über diese ermittelt werden. Über die Kantenliste wird bestimmt welche Eckpunkte zu dieser Face gehören und die Koordinaten der Eckpunkte geändert. Der Aufwand hierfür beträgt O(n).

Das Skalieren einer Face verursacht einen Aufwand von O(n). Hierbei wird innerhalb der Flächenliste die Face über ihre id ermittelt und in der Punktliste die Koordinaten der zur Face gehörenden Eckpunkte verändert. Auf die Eckpunkte wird über die Kantenliste zugegriffen werden.

Auch das Verschieben einer Face geschieht über die Koordinaten der Eckpunkte. Hier wird ebenfalls die Face in der Flächenliste bestimmt und die Eckpunkte in der Punktliste verändert. Über die Kantenliste kann sowohl auf die Faces als auch auf die Eckpunkte zugegriffen werde. Der nötige Aufwand für diese Operation beträgt O(n).

Die Zeiger und die Listen bleiben bei allen Operationen unverändert. Die Objekte innerhalb der Punktliste ändern sich.

rotateMesh: mesh x double x double x double → mesh
scaleMesh: mesh x double x double → mesh
moveMesh: mesh x double x double x double → mesh

Das Rotieren, Skalieren und Verschieben eines Polygonnetzes wird über die Koordinaten der Eckpunkte ausgeführt. Hierbei wird die Eckpunktliste durchlaufen und die Koordinaten der Eckpunkte verändert. Das Durchlaufen einer einfach verketteten Liste erfordert einen Aufwand von O(n). Die Operation ändert keine Liste und keine Zeiger, nur die Objekte der Punktliste.

get-Operationen

getAllVerticesOfMesh: mesh ⟶ set of vertex
getAllEdgesOfMesh: mesh ⟶ set of edge
getAllFacesOfMesh: mesh ⟶ set of face

Um die Eckpunkte des Polygonnetzes zu erhalten wird die Punktliste durchlaufen und ausgegeben. Hierbei handelt es sich um eine einfach verkettete Liste. Der Aufwand dieser Operation beträgt O(n).

Die Ausgabe aller Kanten in einem Polygonnetz wird durch die Ausgabe der Kantenliste realisiert. Hier wird eine doppelt verkettete Liste benutzt. Der Aufwand der Operation beträgt O(n).

Alle Faces eines Polygonnetzes erhält man durch die Ausgabe der Flächenliste. Dies erfordert einen Aufwand von O(n).

Alle diese Operationen fragen nur die Listen ab. Keine der Listen und keiner der Zeiger werden verändert.

getVerticesWhichBelongToFace: mesh x face ⟶ set of vertex

Um die Eckpunkte einer Face zu erhalten kann auf beide Bestandteile über die Kantenliste zugegriffen werden. Sie verweist auf alle nötigen Informationen. So können über eine Kante eine Face und ihre Eckpunkte ermittelt werden. Die Eckpunkte können anhand der Zeiger auf die Punktliste abgelesen und ausgegeben werden. Der Aufwand

beträgt O(n). Bei dieser Operation bleiben die Listen und Zeiger unverändert.

getVerticesWhichBelongToEdge: mesh x edge ⟶ set of vertex

Bei dieser Operation wird die Kante in der Kantenliste über ihre id ermittelt und über die Zeiger auf die Punktliste die Eckpunkte dieser Kante abgelesen und ausgegeben. Das erfordert einen Aufwand von O(n). Die Operation verändert keine Liste. Es werden nur die Kantenliste und die Punktliste abgefragt.

getEdgesWhichBelongToVertex: mesh x vertex ⟶ set of edge

Hierbei müssen in der Kantenliste Kanten ermittelt werden, die den gegeben Eckpunkt enthalten. Es müssen also alle Kanten auf den Eckpunkt überprüft werden. Die Kanten welche zu diesem Eckpunkt gehören werden ausgegeben. Die Operation lässt alle Listen unverändert und erfordert einen Aufwand von O(n).

getEdgesWhichBelongToFace: mesh x face ⟶ set of edge

Bei dieser Operation werden Kanten in der Kantenliste gesucht, die die gegebene Face begrenzen. Die zur Kante gehörende Face kann über die Zeiger zur Flächenliste abgelesen werden. Die Kanten werden anschließend ausgegeben. Die Operation verursacht einen Aufwand von O(n). Sie lässt alle Listen unverändert.

getFacesWhichBelongToEdge: mesh x edge ⟶ set of face

Bei dieser Operation wird die gegebene Kante in der Kantenliste über ihre id ermittelt und die dazugehörigen Faces über die Zeiger abgelesen. Die Faces welche zu dieser Kante gehören werden ausgegeben. Dies verursacht einen Aufwand von O(n). Die Flächenliste und die Kantenliste bleiben unverändert. Die Punktliste wird bei dieser Operation nicht benötigt. Auch die Zeiger zwischen den Listen bleiben unverändert.

getFacesWhichBelongToVertex: mesh x vertex ⟶ set of face

Da alles über die Kantenliste geht muss dort zunächst eine Kante mit dem gegebenen Eckpunkt bestimmt werden. Ist die Kante gefunden kann auf die dazugehörige Face über den Zeiger zur Flächenliste zugegriffen und die Faces ausgegeben werden. Die Listen und Zeiger werden dabei nicht verändert. Der Aufwand beträgt O(n).

is-Operationen

Diese Operationen können mit der Listenoperation `isIn` verglichen werden. Sie überprüfen ob der gegebene Bestandteil innerhalb der entsprechenden Liste vorhanden ist.

isVertexInsideMesh: mesh x vertex ⟶ bool
isEdgeInsideMesh: mesh x edge ⟶ bool
isFaceInsideMesh: mesh x face ⟶ bool

Um festzustellen, ob ein Eckpunkt sich im Polygonnetz befindet wird die Punktliste durchsucht. Dies erfordert einen Aufwand von O(n).

Das Überprüfen des Polygonnetzes auf eine Kante erfordert das Durchsuchen der Kantenliste. Der Aufwand hierfür beträgt ebenfalls O(n).

Um zu sehen, ob eine Face sich im Polygonnetz befindet wird die Flächenliste nach dieser Face durchsucht. Diese Operation erfolgt mit einem Aufwand von O(n).

Alle Operationen fragen nur die Listen ab und bewirken keine Veränderungen der Listen oder der Zeiger.

isVertexBoundingFace: mesh x vertex x face ⟶ bool

Um zu überprüfen, ob ein Eckpunkt eine Face begrenzt wird zunächst in der Kantenliste die Kante mit diesem Eckpunkt ermittelt. Anschließend wird über den Zeiger von der Kante auf die Flächenliste die Face überprüft. Dies kann in einem Schritt erfolgen. Diese Operation verursacht einen Aufwand von O(n). Sie fragt die Listen nur ab, verändert sie aber nicht. Auch die Zeiger zwischen den Listen bleiben unverändert.

isEdgeContainingVertex: mesh x edge x vertex ⟶ bool

Um festzustellen, ob eine Kante einen Eckpunkt begrenzt wird die Kante in der Kantenliste über ihre id ermittelt und über die Zeiger zur Punktliste ihre Endpunkte mit dem gegebenen Eckpunkt verglichen. Das Suchen und Vergleichen verursachen einen Aufwand von O(n). Alle Liste und Zeiger bleiben unverändert.

isEdgeBoundingFace: mesh x edge x face ⟶ bool

Um zu überprüfen, ob eine Kante eine Face begrenzt wird die Kante in der Kantenliste über die id bestimmt und anschließend über die Zeiger auf die Flächenliste geprüft, ob der Zeiger auf die entsprechende Face verweist. Diese Operation verursacht einen Aufwand von O(n). Sie lässt alle Listen und Zeiger unverändert.

isVertexBoundingEdge: mesh x vertex x edge ⟶ bool

Diese Operation erfordert das Durchsuchen der Kantenliste nach der gegebenen Kante. Anschließend wird geprüft, ob die Endpunkte der Kante mit dem angegebenen Eckpunkt übereinstimmen. Die Eckpunkte der Kante können über die Zeiger zur Punktliste abgelesen werden. Die Operation betrifft nur die Kantenliste und die Punktliste und fragt

diese nur ab. Der Aufwand dieser Operation beträgt O(n).

isFaceContainingEdge: mesh x face x edge ⟶ bool

Um festzustellen, ob die Face die Kante beinhaltet wird die Kantenliste nach der Kante durchsucht. Über den Zeiger auf die Flächenliste werden die Faces verglichen. So kann überprüft werden, ob die Kante die gegebene Face begrenzt. Die Operation hat einen Aufwand von O(n). Alle Zeiger und Listen bleiben unverändert.

isFaceContainingVertex: mesh x face x vertex ⟶ bool

Die Überprüfung, ob eine Face einen Eckpunkt beinhaltet, kann über die Kantenliste erfolgen. Durch die dort gespeicherten Informationen kann sowohl die Face als auch ihre Eckpunkte gefunden und mit den Angaben verglichen werden. Die Operation hat einen Aufwand von O(n). Sie fragt die Listen nur ab ohne sie zu verändern.

Nachbarschaften

getAdjacentVerticesOfVertex: mesh x vertex ⟶ set of vertex

Um die Nachbarpunkte eines Eckpunktes zu ermitteln werden zunächst alle Kanten an diesem Eckpunkt bestimmt. Dazu wird die Kantenliste durchlaufen und über die Zeiger zur Punktliste Kanten mit diesem Eckpunkt bestimmt. Anschließend bestimmt man die Endpunkte dieser Kanten, gibt diese aus und eliminiert die Redundanzen. Diese Operation benötigt einen Aufwand von $O(n^2)$ und fragt die Listen nur ab ohne sie zu verändern.

getAdjacentEdgesOfEdge: mesh x edge ⟶ set of edge

Um die Nachbarkanten einer Kante zu bestimmen werden an jedem Endpunkt der Kante die angrenzenden Kanten ermittelt. Die Endpunkte der Kante werden über die Punktliste bestimmt. Anschließend werden alle Kanten an diesen beiden Eckpunkten in der Kantenliste bestimmt. Die Punktliste und die Kantenliste werden nur abgefragt. Die Flächenliste wird bei dieser Operation nicht benötigt. Die Operation erfordert einen Aufwand von $O(n^2)$.

getAdjacentEdgesOfEdgeWithSameFace: mesh x edge ⟶ set of edge

Um benachbarte Kanten zu bestimmen, die auch eine gemeinsame Face besitzen, wird zunächst das gleiche wie bei der Operation `getAdjacentEdgesOfEdge` durchgeführt. Anschließend wird überprüft auf welche Faces die Kanten verweisen. Die Face, die bei beiden dieser Kanten vorkommt wird ausgegeben. Die Operation erfordert einen Aufwand von $O(n^3)$. Sie fragt die Listen nur ab und lässt diese und die Zeiger unverändert.

getAdjacentFaceOfFace: mesh x face ⟶ set of face

Da innerhalb der Winged-Edge-Struktur die Faces rechts und links einer Kante gespeichert werden, können die Nachbarfaces über die Kanten der angegebenen Face ermittelt werden. Dazu werden in der Kantenliste alle Kanten die zu dieser Face gehören herausgesucht und die dazu gespeicherten Faces ausgegeben. Eine der Faces ist immer die gegebene Face. Die Redundanzen können anschließend aus der Ausgabeliste entfernt werden. Die Operation betrifft nur die Kantenliste und die Flächenliste. Beide Listen werden nur abgefragt. Der Aufwand hierfür beträgt $O(n^2)$.

isVertexAdjacentToVertex: mesh x vertex x vertex ⟶ bool

Um zu bestimmen ob zwei Eckpunkte benachbart sind werden zunächst die beiden Eckpunkte in der Punktliste ermittelt. Anschließend wird überprüft ob diese Eckpunkte zu einer Kante gehören. Die Operation hat eine Aufwand von $O(n^2)$ und lässt alle Listen unverändert.

isEdgeAdjacentToEdge: mesh x edge x edge ⟶ bool

Zwei Kanten sind benachbart wenn sie einen gemeinsamen Eckpunkt besitzen. Um zu prüfen, ob dies zutifft werden die beiden Kanten in der Kantenliste über ihre id ermittelt und geprüft, ob sie auf einen gemeinsamen Eckpunkte zeigen. Diese Operation lässt alle Listen und Zeiger unverändert. Der Aufwand beträgt $O(n^2)$.

isEdgeAdjacentToEdgeAndHaveSameFace: mesh x edge x edge ⟶ bool

Zunächst wird vorgegangen wie bei der Operation `isEdgeAdjacentToEdge`. Um dann zu überprüfen, ob beide Kanten zur gleichen Face gehören werden beide Kanten in der Kantenliste ermittelt und über die Zeiger zur Flächenliste überprüft, ob sie auf die gleiche Face verweisen, d.h. die Verweise werden miteinander verglichen. Der Aufwand beträgt $O(n^3)$. Die Listen werden nicht verändert, sondern nur abgefragt. Auch die Zeiger bleiben unverändert.

isFaceAdjacentToFace: mesh face x face ⟶ bool

Zwei Flächen sind benachbart wenn sie eine gemeinsame Kante besitzen. Dazu werden zunächst die beiden Faces bestimmt. Anschließend wird überprüft, ob beide eine gemeinsame Kante enthalten. Dazu wird die Kantenliste nach den Kanten der Faces durchsucht. Die Operation erfordert einen Aufwand von $O(n^2)$. Die Operation fragt die Listen nur ab ohne sie zu verändern. Die Zeiger bleiben ebenfalls unverändert.

In Tabelle 5.6 sind noch einmal alle Operationen mit ihrem Aufwand aufgeführt. Zusätz-

liche Aufwände wie sie beispielsweise für das Ermitteln zusätzlicher Bestandteile erforderlich sind, sind nicht aufgeführt.

Operation	**Aufwand**
Einfügen	
insertVertex: mesh x vertex ⟶ mesh	O(n)
insertEdge: mesh x edge ⟶ mesh (Begin)	O(1)
insertEdge: mesh x edge ⟶ mesh (End)	O(1)
insertEdge: mesh x edge ⟶ mesh (Pos)	O(n)
insertFace: mesh x face ⟶ mesh (Begin)	O(1)
insertFace: mesh x face ⟶ mesh (End)	O(n)
insertFace: mesh x face ⟶ mesh (Pos)	O(n)
Entfernen	
deleteVertex: mesh x vertex ⟶ mesh	O(n)
deleteEdge: mesh x edge ⟶ mesh	O(n)
deleteFace: mesh x face ⟶ mesh	O(n)
Rotieren, Skalieren, Verschieben	
rotateVertex: mesh x vertex x double x double x double ⟶ mesh	O(n)
rotateEdge: mesh x edge x double x double x double ⟶ mesh	O(n)
rotateFace: mesh x face x double x double x double ⟶ mesh	O(n)
rotateMesh: mesh x double x double x double ⟶ mesh	O(n)
scaleEdge: mesh x edge x double x double ⟶ mesh	O(n)
scaleFace: mesh x face x double x double ⟶ mesh	O(n)
scaleMesh: mesh x double x double ⟶ mesh	O(n)
moveVertex: mesh x vertex x double x double x double ⟶ mesh	O(n)
moveEdge: mesh x edge x double x double x double ⟶ mesh	O(n)
moveFace: mesh x face x double x double x double ⟶ mesh	O(n)
moveMesh: mesh x double x double x double ⟶ mesh	O(n)
get-Operationen	
getAllVerticesOfMesh: mesh ⟶ set of vertex	O(n)
getAllEdgesOfMesh: mesh ⟶ set of edge	O(n)
getAllFacesOfMesh: mesh ⟶ set of face	O(n)
getVerticesWhichBelongToEdge: mesh x edge ⟶ set of vertex	O(n)
getEdgesWhichBelongToVertex: mesh x vertex ⟶ set of edge	O(n)
getVerticesWhichBelongToFace: mesh x face ⟶ set of vertex	O(n)
getFacesWhichBelongToVertex: mesh x vertex ⟶ set of face	O(n)
getEdgesWhichBelongToFace: mesh x face ⟶ set of edge	O(n)
getFacesWhichBelongToEdge: mesh x edge ⟶ set of face	O(n)
is-Operationen	
isVertexInsideMesh: mesh x vertex ⟶ bool	O(n)
isEdgeInsideMesh: mesh x edge ⟶ bool	O(n)
isFaceInsideMesh: mesh x face ⟶ bool	O(n)
isVertexBoundingEdge: mesh x vertex x edge ⟶ bool	O(n)

isEdgeContainingVertex: mesh x edge x vertex ⟶ bool	O(n)
isVertexBoundingFace: mesh x vertex x face ⟶ bool	O(n)
isFaceContainingVertex: mesh x face x vertex ⟶ bool	O(n)
isEdgeBoundingFace: mesh x edge x face ⟶ bool	O(n)
isFaceContainingEdge: mesh x face x edge ⟶ bool	O(n)
Nachbarschaften	
getAdjacentVerticesOfVertex: mesh x vertex ⟶ set of vertex	O(n)
getAdjacentEdgesOfEdge: mesh x edge ⟶ set of edge	$O(n^2)$
getAdjacentEdgesOfEdgeWithSameFace: mesh x edge ⟶ set of edge	$O(n^3)$
getAdjacentFaceOfFace: mesh x face ⟶ set of face	$O(n^2)$
isVertexAdjacentToVertex: mesh x vertex x vertex ⟶ bool	$O(n^2)$
isEdgeAdjacentToEdge: mesh x edge x edge ⟶ bool	$O(n^2)$
isEdgeAdjacentToEdgeAndHaveSameFace: mesh x edge x edge ⟶ bool	$O(n^3)$
isFaceAdjacentToFace: mesh face x face ⟶ bool	$O(n^2)$

Tabelle 5.6: Gesamttabelle aller Operationen auf Polygonnetzen innerhalb der Winged-Edge-Struktur und ihr Aufwand

5.5.3 Fazit

In diesem Abschnitt wurden Operationen innerhalb von zwei Speicherungsarten, der Expliziten Speicherung und der Winged Edge-Struktur vorgestellt. Beide Speicherungsarten sind sehr gegensätzlich. Die Explizite Speicherung verwaltet nur Informationen über die Eckpunkte und die Faces. Kanten werden nicht abgespeichert, sind aber dennoch implizit vorhanden. Die Winged Edge-Struktur betrachtet alle Bestandteile eines Polygonnetzes, also auch Kanten. Zudem stellen hier die Kanten die zentrale Struktur dar. Von hier aus gehen alle Verweise zu weiteren Bestandteilen und Informationen. Abfragen sind sehr unterschiedlich ausführbar. Dies hat auch Auswirkungen auf den Aufwand einiger Operationen.

Die Informationen werden jeweils in Listen abgelegt. Die Explizite Speicherung verwaltet eine Flächenliste und mehrere Punktlisten. Dadurch entstehen Redundanzen, da Eckpunkte, die zu mehreren Faces gehören, mehrmals abgelegt werden. Die Winged Edge-Struktur enthält drei Listen: die Punktliste, die Kantenliste und die Flächenliste. Sie enthält keine Redundanzen. Auf Informationen, die zu mehreren Bestandteilen gehören, wird über Verweise zugegriffen.

Die Speicherung der Bestandteile erfolgt bei beiden Strukturen bei den Eckpunkten über deren Koordinaten, bei den Faces über die id und Verweise. Die Winged Edge-Struktur speichert zusätzlich id's für die Eckpunkte und Informationen zu Kanten ab. Die Kanten werden durch id's und Verweise beschrieben. Die nicht gespeicherten Kanten innerhalb der Expliziten Speicherung können durch ein Paar von Eckpunkten beschrieben werden. Da die Eckpunkte in der Punktliste stets in der Reihenfolge abgelegt werden, wie sie auch gezeichnet werden, ist klar, dass sich zwischen zwei aufeinander folgenden Eckpunkten und zwischen dem ersten und dem letzten Eckpunkt jeweils eine Kante befindet.

Die in Kapitel 3 vorgestellten Operationen wurden zur Ermittlung des Aufwandes herangezogen. Außerdem wurde Bezug genommen auf die verschiedenen Fälle, die bei den Polygonoperationen betrachtet werden müssen, und die Bedingungen innerhalb eines Polygonnetzes, wie Einfachheit, Planarität und Konvexität der Faces.

Die Operationen wurden in diesem Abschnitt wie folgt aufgeteilt: Einfügen, Löschen, Rotieren, Skalieren und Translatieren, get-Operationen, is-Operationen, Nachbarschaften und bei der Expliziten Speicherung zusätzlich Kantenoperationen. Die Operationen Einfügen und Löschen beeinflussen den Inhalt und die Zusammenhänge der Listen. Die Operationen Rotieren, Skalieren und Translatieren ändern nur den Inhalt der Punktlisten, die Beziehungen zwischen den Listen bleiben erhalten. Die get- und is-Operationen, sowie die Nachbarschafts- und Kantenoperationen fragen die Listen nur ab.

Innerhalb der Expliziten Speicherung muss bei den Nachbarschaften beachtet werden, dass keine Kanten gespeichert werden. Hier werden die Operationen über die Paare von Punkten der Kanten bestimmt. Auch die Kantenoperationen können nur durchgeführt werden, wenn die Kanten als Punktpaare definiert werden.

Bei allen Operationen müssen die Bedingungen, die innerhalb eines Polygonnetzes gelten, beachtet werden. Sie können nur dann ausgeführt werden, wenn die entstehenden Faces planar, konvex und einfach bleiben. Besonders wichtig sind diese bei folgenden Operationen: insertVertex, deleteVertex, deleteEdge, insertFace, deleteFace. Sie führen zum Verschmelzen von Faces. Das Rotieren aller Bestandteile kann nur innerhalb von Dreieckesnetzen durchgeführt werden, da sonst die Planarität der Faces nicht mehr gewährleistet ist. Auch das Verschieben der Bestandteile muss wieder zu diesen Bedingungen führen.

Einige der vorgestellten Operationen haben einen unterschiedlichen Aufwand innerhalb der vorgestellten Speicherungsarten. Sie werden noch einmal aufgezeigt. So ist der Aufwand beim Löschen von Kanten innerhalb der Expliziten Speicherung höher. Das Löschen dieser kann hier nur über ihre Eckpunkte erfolgen. Dazu müssen die Faces mit diesen Eckpunkten ermittelt und gelöscht werden. Anschließend muss eine neue Face in die Flächenliste eingefügt werden. Der Aufwand beträgt O(n^2). Bei Winged Edge kann direkt auf die Kante zugegriffen werden. Durch Umsetzen der Zeiger können die neuen Faces erstellt werden. Der Aufwand beträgt hier nur O(n). Das Rotieren, Skalieren und Translatieren von Faces ist bei der Expliziten Speicherung ebenfalls aufwendiger, da Eckpunkte, die zu mehreren Faces gehören in mehreren Punktlisten abgegelegt werden. Hier muss zunächst ermittelt werden zu welchen Faces welcher Eckpunkt gehört, bevor die Koordinaten der Eckpunkte geändert werden können. Die get-Operationen `getAllVerticesOfMesh` und `getAllEdgesOfMesh` verursachen innerhalb der Expliziten Speicherung einen Aufwand von O(n^2). Hier müssen noch die Redundanzen behoben werden, welche durch die Speicherung der Eckpunkte in mehreren Punktlisten entstehen. Dies entfällt bei der Winged Edge-Struktur. Hier beträgt der Aufwand O(n). Zudem gestalten sich Operationen bezüglich der Kanten innerhalb der Expliziten Speicherung

schwieriger. Sie sind nur implizit vorhanden und werden durch ihre Eckpunkte beschrieben. Hintereinander liegende Eckpunkte innerhalb einer Punktliste bilden eine Kante. Außerdem befindet sich eine Kante zwischen dem ersten und dem letzten Eckpunkt der Punktliste. Hier muss meisten noch geprüft werden, ob die Eckpunkte wirklich zu einer Kante gehören, also hintereinander abgelegt sind.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurden 3D-Modelle im Bezug auf die Verwendung innerhalb von Augmented Reality-Anwendungen untersucht. In der Augmented Reality bilden eingeblendete Objekte einen wichtigen Bestandteil. Davon stellen die 3D-Modelle einen großen Anteil dar. Sie sollen sich in die umgebende Welt nahtlos und zur richtigen Zeit einfügen. Um dies zu erreichen spielen die Wahl des 3D-Modells und die verwendete Datenstruktur zur Speicherung des Modells eine große Rolle. Sie entscheiden darüber mit welchem Aufwand und in welcher Zeit das Einblenden erfolgen kann. Die vier bekanntesten Modelle sind die Punktwolke, das Drahtmodell, das Flächenmodell sowie das Volumenmodell. Den Schwerpunkt dieser Arbeit bildete das Flächenmodell.

Für die Untersuchung wurde ein Konzept, bestehend aus den fünf Stufen Modell, Repräsentationsform, Abstrakte Datenstruktur, Speicherungsart und Implementation, erstellt. Das Konzept dieser Stufen diente zur besseren Verständlichkeit des Themengebietes, und zur Trennung der verschiedenen Abstufungen, welche das gesamte Modell beschreiben.

Verlauf dieser Arbeit

Zu Anfang wurde das erarbeitete Konzept inklusive eines Beispiels zur Verdeutlichung vorgestellt. Als Beispielobjekt diente der berühmte Teapot. Anschließend wurde das Konzept innerhalb der Abstrakten Datenstruktur Liste angewendet. Hier wurde die Datenstruktur mit ihren Operationen und ihren Speicherungsarten vorgestellt. Die Implementierung der Operationen erfolgte innerhalb einer einfach verketteten und einer doppelt verketteten Liste in C++. Das Kapitel über Listen diente als Basis für die später vorgestellten Speicherungsarten im Bezug auf das Flächenmodell. Im weiteren Verlauf der Arbeit wurden die vier häufigsten Repräsentationsformen von 3D-Modellen vorgestellt. Sie dienten vor allem zur Einordnung des Flächenmodells.

Das darauf folgende Kapitel über das Flächenmodell bildete den größten Teil der Arbeit. Hier wurde das Flächenmodell nach dem zu Anfang beschriebenen Konzept untersucht. Zu Begin wurde das Modell vorgestellt. Das Flächenmodell gehört zu den 3D-Modellen und kann durch unterschiedliche Repräsentationsformen beschrieben werden. Die Arbeit bezog sich auf die Darstellung durch Polygone. Hier erfolgte zunächst eine Definition des Polygonnetzes aus der Sicht der Computergraphik. Der Aufbau und die Beziehungen innerhalb eines Polygonnetzes wurden anhand eines Klassendiagramms und des vef-Graphen dargestellt. Beide Sichtweisen wurden vorgestellt und in Beziehung zueinander gesetzt. Danach folgten die Abstrakte Datenstruktur, welche alle ausführbaren Operationen beinhaltete, die Speicherungsarten und die Implementation. Die aufgeführten

Operationen stellten einen großen Teil der Betrachtungen dar. An dieser Stelle wurde überlegt welche Operationen möglich und sinnvoll sind. Eine besondere Herausforderung boten die Bedingungen des Polygonnetzes. Um die Operationen und ihre Auswirkungen zunächst einzuschränken, wurde beispielsweise vorausgesetzt, dass die Flächen konvex, planar und einfach sind. Um eine spätere Umsetzung der Operationen innerhalb einer geeigneten Speicherungsart zu ermöglichen, mussten Vorüberlegungen getroffen und eine genaue Spezifikation erstellt werden. Vor allem die Spezifikation der Operationen ermöglicht eine schnellere Umsetzung dieser. Daraufhin wurden verschiedene bekannte Speicherungsarten vorgestellt und anschließend in Bezug zu den Operationen gesetzt. Hier wurde ein möglicher Aufwand bestimmt. Wichtig ist hierbei welche Daten gespeichert werden und wie auf diese zugegriffen werden kann. Auch hier müssen die Bedingungen des Polygonnetzes beachtet werden. Sie schränken die Operationen ein. Einige der Operationen sind unter bestimmten Umständen nicht ausführbar.

Erkenntnisse dieser Arbeit

Um ein geeignetes Modell zu finden, muss eine Repräsentation gefunden werden mit der möglichst viele Objekte schnell und korrekt darstellbar sind. Hierbei spielt der Aufbau der Struktur und die darin gespeicherten Daten eine große Rolle. Sie bestimmen den Speicherplatzbedarf und die nötige Rechenzeit.

Die Punktwolke stellt eine einfache Abbildung eines Objektes durch Eckpunkte dar. Damit ist allerdings keine realitätstreue Darstellung möglich. Zudem erfordern Krümmungen innerhalb des Objektes sehr viele Eckpunkte. Diese verursachen einen sehr hohen Speicherbedarf und erhöhen die Zeit die zur Projektion auf eine Bildfläche nötig ist. Das Drahtmodell zeigt nur die Umrisse des Objektes. Dadurch entsteht eine untreue Abbildung der Wirklichkeit. Außerdem kann es zu Mehrdeutigkeiten kommen, wenn das Modell mit Flächen gefüllt wird. Hier können verschiedene Objekte entstehen. Für die Platzierung von Objekten ist das Drahtmodell jedoch sehr gut geeignet. Es werden nur wenige Informationen gespeichert, dadurch ist der Speicherbedarf überschaubar. Es müssen nur Kanten dargestellt werden, sodass der Rechenaufwand nur von der Anzahl der Kanten abhängt. Das Volumenmodell kann auf unterschiedliche Weisen beschrieben werden. Eine bekannte Darstellung ist die CSG. Sie beschreibt das Modell durch volle Körper wie Kugeln und Quader. Da diese Art der Darstellung keine geometrischen Informationen verwaltet ist ein spezielles Rendering-Verfahren oder die Umwandlung in ein Polygonnetz zur Darstellung nötig. Dadurch ist die Darstellung sehr zeit- und rechenintensiv.

Das Flächemodell repräsentiert ein Objekt durch seine Oberfläche. Außerdem wird das Objekt sowohl durch geometrische als auch durch topologische Informationen beschrieben. Erst die topologischen Informationen bilden aus den einzelnen Flächen das Modell. Die Oberflächenbeschreibung kann entweder durch Polygone oder Parameter-Oberflächen erfolgen. Die in dieser Arbeit betrachtete Art der Darstellung waren die Polygone. Polygone ermöglichen eine schnelle und einfache Darstellung. Zudem wird die Darstellung durch Schnittstellen und Auslagerungen auf die Graphik-Hardware unterstützt.

Um zu entscheiden welche der Speicherungsarten geeignet ist und welche Operationen nötig sind, muss eine genaue Aufgabenstellung vorhanden sein. Anhand der vorgenommenen Untersuchung lässt sich jedoch schneller auswählen welche Struktur geeignet sein könnte. Durch den ermittelten Aufwand kann entschieden werden, welche Speicherungsart wann einzusetzen ist. Das Flächenmodell ist durch die zahlreichen vorhandenen Möglichkeiten und die einfache Speicherung und Darstellung dieses Modells für die Nutzung innerhalb von Augmented Reality-Anwendungen gut geeignet.

Operationen auf Polygonnetzen können sowohl zur Erstellung eines Polygonnetzes, zur Veränderung als auch zur Abfrage von Informationen dienen. So können Bestandteile eingefügt, entfernt oder verändert werden. Außerdem können Informationen über Beziehungen innerhalb des Polygonnetzes und das Vorhandensein von Bestandteilen abgefragt werden. Wichtig ist hier die Beachtung der Bedingungen des Polygonnetzes. Die Ausführung der Operationen muss stets zu planaren, einfachen und konvexen Flächen innerhalb des Polygonnetzes führen. Durch diese Bedingungen sind einige Operationen, wie beispielsweise das Entfernen von Kanten, welche zum Zusammenschmelzen von Flächen führen nicht ausführbar. Die verschiedenen Bestandteile des Polygonnetzes können unterschiedlich abgelegt werden. Hierzu wurden verschiedene Speicherungsarten vorgestellt. Diese speichern unterschiedliche Bestandteile und Informationen in unterschiedlichen Listen und vereinen zudem die nötigen geometrischen und topologischen Informationen. So enthalten die Speicherungsarten die Koordinaten der Eckpunkte und speichern durch Verweise Beziehungen im Polygonnetz. Die Bestandteile eines Typs werden in der gleichen Liste gespeichert. So werden Eckpunkte in einer Punktliste, Kanten in einer Kantenliste und Faces in einer Flächenliste abgelegt. Einige der Speicherungsarten wie die Explizite Speicherung speichern mehrere Listen zum gleichen Bestandteil. Dadurch entstehen Redundanzen. Sie verursachen einen höheren Speicheraufwand und einen höheren Rechenaufwand bei der Ausgabe und Ermittlung dieser Bestandteile, da hier die Redundanzen entfernt werden müssen. Speicherungsarten wie die Explizite Speicherung und die Eckenliste speichern keine Informationen über Kanten. Auch die in der Hardware verwendeten Speicherungsarten der Strips und Fans gehören zu dieser Kategorie. Hier wird das Polygonnetz ausschließlich über die Eckpunkte und die Faces beschrieben. Innerhalb der Kantenliste, sowie der Half-Edge und Winged-Edge Struktur spielen Kanten dagegen eine zentrale Rolle. Zusätzlich verwaltet die Winged-Edge Struktur weitere Informationen wie Vorgänger- und Nachfolgerkanten. Dadurch ist ein schnellerer Zugriff auf diese Informationen möglich. Sie können direkt abgelesen und müssen nicht erst ermittelt werden. Grundsätzlich ist jede Speicherungsart geeignet. Sie sollte aber stets an die gestellten Anforderungen angepasst sein. Vor allem wenn viele Informationen benötigt werden, ist es sinnvoll eine Speicherungsart zu wählen welche diese in ihrer Struktur enthält. Der benötigte Speicherplatz ist geringer als der benötigte Rechenaufwand zur Ermittlung der fehlenden Informationen. Speicherungsarten welche keine Kanteninformationen speichern verursachen bei der Abfrage von diesen Informationen einen höheren Aufwand. Die Kanten können hier zwar durch die Beschreibung anhand von Eckpunkt-Paaren ersetzt werden, jedoch muss stets sichergestellt sein, dass sich dort auch tatsächlich Kanten befinden.

Ausblick

Ansätze für Erweiterungen und Verbesserung sind vor allem die Betrachtung weiterer Bedingungen innerhalb von Polygonnetzen wie beispielsweise die Orientierbarkeit der Flächen beim Flächenmodell. Oder auch die Betrachtung konkaver Flächen. Daraus können sich auch weitere Operationen und Einschränkungen innerhalb dieser ergeben. Um noch genauer festlegen zu können welches Modell unter welchen Bedingungen die geeignetste Wahl darstellt, ist eine genauere Untersuchung der restlichen Modelle nötig. Zudem können innerhalb verschiedener Anwendungen unterschiedliche Modelle geeignet sein, daher sollten die Modellarten innerhalb von konkreten Anwendungen betrachtet werden.

Literaturverzeichnis

[Akl06] Ergun Akleman. Topological Mesh Modeling. http://www-viz.tamu.edu/faculty/ergun/research/topology/papers.html, 2006.

[Ape] Jens Apel. Rekonstruktion von Oberflächen aus Punktewolken/Delaunay Netzwerke. http://www.stud.tu-ilmenau.de/~japel/vortrag/gdv/delaunay.htm. aufgerufen 2006.

[Bak] Steve Baker. The History of The Teapot. http://www.sjbaker.org/teapot/. aufgerufen 2006.

[Bal99] Helmut Balzert. *Lehrbuch Grundlagen der Informatik*. Spektrum Akademischer Verlag, 1999.

[Bar] Moritz Bartl. Geometric Representation & Processing, Proseminar Computer Graphics. http://wwwcg.in.tum.de/Teaching/SS2004/ProSem/Workouts/bartlmo/Folien.pdf. aufgerufen 2006.

[Bri05] Michael Bender; Manfred Brill. *Computergraphik*. Carl Hanser Verlag, 2nd edition, 2005.

[Bro03] Brockhaus. Fachlexikon Computer, 2003.

[Ebe] Jürgen Ebert. Softwaretechnik, Vorlesungsunterlagen. http://www.uni-koblenz.de/FB4/Institutes/IST/AGEbert/Teaching.

[Hof04] Peter Fischer; Peter Hofer. *Lexikon der Informatik*. SmartBooks Publishing AG, 6th edition, 2004.

[IGD05] Fraunhofer IGD. Geometrische Methoden des CAD/CAE - Vorlesung WS 04/05. http://www.igd.fhg.de/igd-a2/staff/stork/Vorlesung/CAD_CAE_lectures.html, 2005.

[Kob] Stephan Bischoff; Leif Kobbelt. Teaching meshes, subdivision and multiresolution techniques. http://www-i8.informatik.rwth-aachen.de/publications/downloads/teaching_meshes_preprint.pdf. Computer Graphics Group, RWTH Aachen.

[Kon99] Kurt Konolige. Stereo Geometry, Disparity and the Horopter. http://www.ai.sri.com/~konolige/svs/disparity.htm, 1999.

[Lan] Hans Werner Lang. Grundlagen: Polygon. http://www.iti.fh-flensburg.de/lang/algorithmen/geo/polygon.htm. Institut für Medieninformatik und Technische Informatik, FH Flensburg, aufgerufen 2006.

[Loh04] Friedrich A. Lohmüller. CSG - Constructive Solid Geometry. http://www.f-lohmueller.de/pov_tut/csg/povcsg1.htm, 2004.

[Mül05] Stefan Müller. Computergraphik 2. Universität Koblenz - Institut für Computervisualistik, 2004/2005. Semesterunterlagen; http://www.uni-koblenz.de/FB4/Institutes/ICV/AGMueller/Teaching/WS0405/CG2/Materialien.

[Mül06] Stefan Müller. Virtuell Realität und Augmented Reality. http://www.uni-koblenz.de/FB4/Institutes/ICV/AGMueller/Teaching/WS0506/VRAR/Materialien, 2005/2006. Semesterunterlagen.

[Nau] Jens Nausedat. Geometrische Modellierung, Seminar Computational Steering. http://www.computational-steering.org/seminar/Geometrische_Modellierung_Topo.pdf. aufgerufen 2006.

[ope04] Opengl 2.0 Specification. http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf, 2004.

[Pau05] Dietrich Paulus. Medizinische Bildverarbeitung, Struktur aus Bewegung. http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/Teachings, 2004/2005. Vorlesungsunterlagen.

[Pil02] Tony Pilz. Proseminar: Simulation und VR in der Medizin SS02, Universität Karlsruhe. http://wwwiaim.ira.uka.de/Teaching/ProseminarMedizin/Ausarbeitungen/SS2002/05_Vernetzung.pdf, 2002.

[R.D] R.Dumke. Komplexität, Aufwand. http://ivs.cs.uni-magdeburg.de/~dumke/EAD/Skript20.html. Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Institut für Verteilte Systeme, AG Softwaretechnik.

[Rei06] Prof. Dr. U. Reif. Mathematik für Maschinenbau. http://www.mathematik.tu-darmstadt.de:8080/Math-Net/Lehrveranstaltungen/Lehrmaterial/WS2005-2006/MatheI_MB/skript.pdf, 2006.

[The06] E. Gröller; T. Theußl. 3D-Datenstrukturen. http://www.cg.tuwien.ac.at/courses/CG2/slides.html, 2006. Technische Universität Wien, Institut für Computergraphik und Algorithmen, Vorlesungsunterlagen.

[tuK04] Kompaktkurs Computergraphik und CAD / Multimediasysteme. http://www.icsy.de/studium/vorlesung/ss04/vorlesung.shtml, SS 2004. TU Kaiserslautern, Fachbereich Informatik.

[uni06] Projekt Enhanced Reality. http://er.uni-koblenz.de/, 2006. Projektbeschreibung.

[uR04] Bayerischer Forschungsverbund Abfallforschung und Reststoffverwertung. BayFORREST - Modellarten. http://www.inf.bauwesen.tu-muenchen.de/forschung/bayforrest/f248/index.php?sub=projekt/modelle, 2004.

[Wat02] Alan Watt. *3D-Computergraphik*. Pearson Studium, 3rd edition, 2002.

[Wös] Uwe Wössner. Szenengraphen. http://www.hlrs.de/organization/vis/vorlesung/VorlesungVis05.pdf. Visualisierung technisch wissenschaftlicher Daten, Uni Stuttgart.

[Zen99] Hans-Joachim Bungartz; Michael Griebel; Christoph Zenger. *Einführung in die Computergraphik*. Verlag Vieweg, 1999.

Abbildungsverzeichnis

Tabellenverzeichnis