UNIVERSITÄT OSNABRÜCK

Institut für Informatik
AG Wissensbasierte Systeme

Masterarbeit

Automatische Rekonstruktion Planarer 3D-Umgebungen

Thomas Wiemann

Oktober 2007

Erstgutachter: Prof. Dr. Joachim Hertzberg
Zweitgutachter: Prof. Dr. Oliver Vornberger

Zusammenfassung

Ein aktueller Trend in der mobilen Robotik ist es, die Umgebung eines Roboters mit 3D−Laserscannern zu erfassen. Die Tatsache, dass befahrbare Gebiete in der Regel eben sind, spiegelt sich auch in den aufgenommenen Punktwolken wieder. Diese sind ohne weitere Aufbereitung nur schwer zu interpretierbar. Ein gängiger Ansatz zur Weiterbearbeitung ist die Erstellung polygonaler Netze aus den gewonnenen Daten. Diese haben den Vorteil, dass sie effizient mit gängigen Grafikbibliotheken gerendert werden können.

In der vorliegenden Arbeit wird ein Verfahren zur Modellerzeugung und -optimierung aus solchen Daten vorgestellt. Dabei wird zunächst mit Hilfe des bekannten Marching-Cubes-Algorithmus ein initiales Umgebungsmodell erzeugt. In einem zweiten Schritt wird dieses Modell optimiert, indem die planaren Anteile im Modell detektiert und zusammengefasst werden. Des Weiteren werden die so gefundenen Flächen klassifiziert und mit passenden Texturen belegt. Das Hauptaugenmerk bei allen Schritten liegt auf der Minimierung der benötigten Rechenzeit.

Abstract

A current trend in mobile robotics is to use 3D laser scanners to capture the surroundings of a robot. The fact, that drivable areas are mostly flat, is reflected in the collected data. But 3D point clouds are hard to interpret by humans. Therefore fast and reliable algorithms are needed to automatically generate more suitable models. One common approach is to create triangle meshes approximating the collected data. These meshes can be rendered efficiently by modern graphics systems.

This theses describes a method for model generation and optimization based on 3D laser scanner data collected by mobile robots. In a first step an inital model is built based on this data using the well known Marching Cubes algorithm. Optimization is done by detecting and combining the planar regions in the initial model. These regions are classified and textured with appropriate textures. Special attention is turned on reducing the computational costs.

Danksagung

Besonders bedanken möchte ich mich bei Prof. Dr. Hertzberg für die freundliche Betreuung meiner Masterarbeit. Die Gespräche mit ihm und seine vielen Denkanstöße haben mir geholfen, einige Probleme zu erkennen und zu lösen.

Mein weiterer Dank gilt Andreas Nüchter und Kai Lingemann, die viele große und kleine Fehler in meiner Arbeit gefunden haben und immer geholfen haben, wenn „der LaTeX" mal wieder nicht das gemacht hat, was ich wollte.

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Algorithmen

Kapitel 1

Einleitung

Eine häufige Aufgabenstellung in der mobilen Robotik ist es, aus den vorhandenen Sensordaten eines Roboters ein Computermodell seiner Umgebung zu erstellen. Dazu werden in der Regel Laserscannerdaten und/oder Kamerabilder ausgewertet. Die mit dem Laserscanner des Roboters Kurt3D aufgenommenen 3D-Punktwolken ergeben allerdings nur einen relativ vagen Eindruck von der gescanten Umgebung. Zudem fallen große Datenmengen an. Daher ist eine Modellerzeugung per Hand aus solchen Daten in der Regel sehr zeitaufwändig oder gar unmöglich. Deshalb werden Algorithmen benötigt, die diese Aufgabe zuverlässig automatisch erledigen.

Eine Möglichkeit, algorithmisch aus den Rohdaten des 3D-Scanners ein besseres Umgebungsmodell zu gewinnen, bietet der bekannte Marching-Cubes-Algorithmus [25]. Mit Hilfe dieses Algorithmus lassen sich aus einer 3D-Punktwolke Dreiecksnetze erstellen, die leicht mit gängigen Grafikbibliotheken gerendert werden können. Diese Darstellung erlaubt es, einen erheblich natürlicheren Eindruck von der Umgebung des Roboters zu gewinnen. Allerdings enthalten die auf diese Art und Weise erzeugten Netze wesentlich mehr Dreiecksflächen als zu einer genauen Approximation der gescanten Oberfläche notwendig sind. Dieser Effekt tritt besonders dann hervor, wenn die gescante Umgebung überwiegend aus ebenen Gebieten besteht, wie es in der Regel in Umgebungen der Fall ist, die von mobilen Robotern befahrbar sind. Die unnötig häufige Unterteilung entsteht dadurch, dass der von den Messdaten eingenommene Raum bei der Verwendung des Marching-Cubes-Algorithmus in gleich große Zellen eingeteilt wird, in denen lokal bis zu vier Dreiecke erzeugt werden.

Die Herausforderung besteht nun darin, die Anzahl der Dreiecksflächen zu reduzieren, ohne die Geometrie des zuvor generierten Modells zu verändern. Für komplexe Objekte, die wenige planare Anteile haben, wurden bereits in der Vergangenheit Optimierungsverfahren entwickelt. Die meisten basieren darauf, iterativ die Elemente aus dem Modell herauszunehmen, deren Entfernung den geringsten Fehler an der Geometrie verursacht. Da nach jedem Iterationsschritt Kosten der am Modell vorgenommenen Änderungen zumindest lokal berechnet werden müssen, sind solche Verfahren relativ rechenaufwändig.

In der vorliegenden Arbeit wird ein neues Verfahren zur Modelloptimierung entwickelt. Es basiert auf der Annahme, dass die vorhandenen Datensätze viele Punkte enthalten, die in Ebenen

liegen. Mit Hilfe geeigneter Algorithmen und Datenstrukturen sollen zunächst in einem initialen Marching-Cubes-Modell die Konturen dieser Flächen gefunden werden. Die Dreieckskanten, die diese Begrenzungen bilden, werden anschließend zu mit Hilfe eines Konturverfolgungsalgorithmus zusammengefasst, so dass eine Polygondarstellung der Konturen erzeugt wird. Diese Konturpolygone werden in einem dritten Schritt trianguliert, so dass wieder eine Approximation durch Dreiecke vorliegt. Ein weiteres Ziel ist es, diese Flächen anhand ihrer Lage im Raum zu klassifizieren und mit geeigneten Standardtexturen zu belegen, so dass ein realistischer Eindruck von der gescanten Umgebung entsteht. Das Hauptaugenmerk bei der Entwicklung dieser Algorithmen liegt auf der Reduzierung der Rechenzeit, die benötigt wird, um ein Umgebungsmodell aus Laserscannerdaten zu erstellen, da die zur Verfügung stehenden Rechenkapazitäten im Bereich der mobilen Robotik begrenzt sind.

1.1 Aufbau der Masterarbeit

Kapitel 1: Kapitel 1 gibt einen Überblick über die der Arbeit zugrunde liegende Aufgabenstellung. Des Weiteren wird der Aufbau der Arbeit vorgestellt und der wissenschaftliche Beitrag erörtert.

Kapitel 2: Kapitel 2 arbeitet den aktuellen Stand der Forschung auf. Es werden verschiedene Arten von 3D-Laserscannern und ihre Anwendungsgebiete vorgestellt sowie Techniken zur Auswertung der gewonnenen Punktwolken angesprochen.

Kapitel 3: Im dritten Kapitel wird gezeigt, wie sich mit Hilfe einer speziellen Datenstruktur, dem HashGrid, eine effiziente Implementierung des bekannten Marching-Cubes-Algorithmus zur Erzeugung von Dreiecksnetzen in verschiedenen Repräsentationen realisieren lässt.

Kapitel 4: In Kapitel 4 wird ein Verfahren zur Optimierung der Marching-Cubes-Modelle eingeführt. Die Grundannahme dabei ist, dass es sich bei diesen Modellen um Rekonstruktionen planarer Umgebungen handelt, wie sie häufig im Berech der mobilen Robotik auftreten.

Kapitel 5: Um beim Rendern der Dreiecksnetze eine möglichst realistische Wiedergabe der Umgebungsrekonstruktionen zu erreichen, sollen diese mit Texturen versehen werden. Im 5. Kapitel wird gezeigt, wie die dazu benötigten Texturkoordinaten berechnet und den Polygonen im Dreiecksnetz durch Analyse der Marching-Cubes-Konfigurationen Standardtexturen zugewiesen werden können.

Kapitel 6: Im 6. Kapitel wird das in den Kapitel 3 bis 5 entwickelte Verfahren anhand verschiedener Testdatensätze evaluiert. Dazu werden die Laufzeiten und die erreichten Kompressionsraten bei der Modelloptimierung mit iterativen Verfahren verglichen.

Kapitel 7: Kapitel 7 gibt eine Zusammenfassung der Ergebnisse dieser Arbeit, und es werden weitergehende zukünftige Anwendungen vorgeschlagen.

1.2 Wissenschaftlicher Beitrag

Die meisten bekannten Vereinfachungsalgorithmen für polygonale Netze gehen davon aus, dass es sich bei den modellierten Objekten um komplexe Strukturen handelt. Dies liegt daran, dass der Marching-Cubes-Algorithmus ursprüglich zur Visualisierung der Messwerte von Computer-Tomographien entwickelt wurde.

In dieser Arbeit wird ein neuartiges Verfahren zur Optimierung von Polygonnetzen vorgestellt, welches auf der Annahme basiert, dass es sich bei den vorhandenen Daten um Laserscans von mobilen Robotern handelt, die viele planare Anteile enthalten. Das wird bei der Modelloptimierung ausgenutzt, indem diese Gebiete zusammengefasst und neu trianguliert werden. Im Gegensatz zu anderen Verfahren wird dabei wesentlich weniger Rechenzeit benötigt und gleichzeitig die Form der ursprünglich erzeugten Modelle nicht verändert. Auf diese Art und Weise wird ein guter Kompromiss zwischen Modellqualität und Rechenaufwand erreicht.

Kapitel 2

Stand der Forschung

2.1 3D Laser Scanning

Zur präzisen Vermessung dreidimensionaler Objekte kommen in der Regel Laserscanner zum Einsatz. Diese Sensoren schicken einen Laserstrahl in einer bestimmten Richtung aus und messen das reflektierte Signal. Aus der Laufzeitdifferenz zwischen ausgesendetem und empfangenem Signal wird der Abstand zu der Oberfläche berechnet, die den Strahl reflektiert hat. Zur Bestimmung der Entfernung gibt es zwei verschiedene Methoden. Bei gepulsten Systemen (PW-Systemem) werden kurze Laserblitze ausgesandt und die Laufzeit Δt des Lichts gemessen. Aus der Lichtgeschwindigkeit c kann dann direkt die Entfernung L gemessen werden. Bei der anderen Variante werden kontinuierliche Lichtwellen ausgesandt (CW-Systeme). Dabei wird die Objektentfernung aus der Phasendifferenz zwischen ausgesandtem und empfangenem Licht berechnet.

Laserscanner, die mit der CW-Methode arbeiten, erreichen vergleichsweise hohe Auflösungen. Aufgrund des großen technischen Aufwands sind sie allerdings sehr teuer. Meistens kommen sie in sogenannten Architekturscannern zum Einsatz, mit denen die Fassaden von Gebäuden vermessen werden. Bekannte Hersteller solcher Systeme sind Riegl, CYREX, Leica und Zoller+Fröhlich. PW-Systeme werden aufgrund der vergleichsweise geringen Kosten häufig in der Industrie oder der mobilen Robotik eingesetzt. Bekannte Hersteller sind z.B. SICK und Schmersal. Typische Auflösungen liegen je nach Gerät zwischen 2 mm(Riegl) und 20 mm(SICK). Die

Abbildung 2.1: Laserscanner verschiedener Hersteller. (a) SICK [6], (b) Schmersal [5], (c) Zoller+Froehlich [7], (d) Riegl [4], (e) Leica [3]

Abbildung 2.2: Der Roboter Kurt3D

messbaren Maximaldistanzen betragen bis zu 200 m(Riegl). Einige Laserscanner verschiedener Hersteller sind in Abbildung 2.1 gezeigt.

Die meisten dieser Geräte vermessen dreidimensionale Umgebungen durch Schwenken eines 2D-Laserscanners. Ein 2D-Scan liefert das Entfernungsprofil in einer Ebene. Dazu wird der ausgesandte Laserstrahl mittels eines rotierenden Spiegels abgelenkt und die Objektentfernug in der entsprechenden Richtung gemessen. Diese Polarkoordinaten können anschließend in karthesische Koordinaten umgerechnet werden. Typische Winkelbereiche eines 2D-Scans liegen zwischen 180^o und 270^o. Werden solche Profile unter verschiedenen Blickwinkeln erstellt, entsteht ein dreidimensionales Abbild der Umgebung.

Ein solches System kommt auch auf dem Roboter Kurt3D zum Einsatz (s. Abbildung 2.2). Dabei wird ein SICK-Laserscanner mit Hilfe eines Servos geschwenkt. Auf diese Art und Weise kann ein maximaler Winkelbereich von $180°(\text{h}) \times 120°(\text{v})$ vor dem Roboter in unterschiedlichen Auflösungen erfasst werden. Horizontal können Auflösungen von 181, 361 und 721 Messpunkten eingestellt werden. In der Horizontalen sind 128, 176, 256, 400 oder 500 Messungen möglich. Der Scan einer Ebene mit 181 Punkten benötigt dabei 13 ms. Höhere Auflösungen verdoppeln bzw. vervierfachen diese Zeit. Ein kompletter Scan mit 361×176 Punkten benötigt z.B. 4.5 s [24,43].

Alternativ zu Laserscannern können auch sogenannte Projektionsscanner eingesetzt werden, um 3D-Szenen zu erfassen. Auf der Grundlage solcher Daten wurden u.a. die Modelle im „Digitalen Micheangelo"-Projekt der Stanford University erstellt [2]. Dabei wurde ein Lasermuster auf die zu vermesseenden Objekte projeziert und mit einer Kamera detektiert. Aus den Verzerrungen des Musters wurden dann die Oberflächen rekonstruiert. Weitere Alternativen bieten 3D-Stereo-Kameras. Diese sind allerdings derzeit noch nicht ausgereift und liefern nur sehr unpräzise Daten.

2.2 Modellerzeugung

Die mit dem Laserscanner aufgenommenen Punktwolken können ohne weitere Aufbereitung auf mobilen Robotern nur schwer ausgewertet werden, da sehr große Datenmengen anfallen. Zudem liegen nur diskrete Punkte vor. Da es sich bei den gescanten Objekten in der Regel

um Flächen handelt, müssen Methoden gefunden werden, die aus der Punktdarstellung eine Flächenrepräsentation erzeugen.

Ein verbreiteter Ansatz ist es, eine implizite Funktion der Form $F(x, y, z) = 0$ mittels verschiedener Varianten von Least-Squares-Fits aus den Messdaten zu approximieren. Für Punkte, die auf der Oberfläche eines Objektes liegen, soll diese Funktion den Wert Null annehmen. Für Punkte, die sich innerhalb eines Objektes befinden, soll sich ein negativer Funktionswert ergeben, für Punkte, die außerhalb liegen, ein positiver. Auf diese Art und Weise kann für jeden Punkt des Raumes entschieden werden, ob er sich auf dem gescanten Objekt befindet. Durch zusätzliche Randbedingungen, wie z.B. Stetigkeit, kann garantiert werden, dass sich zwischen den diskret gemessenen Punkten ein kontinuierlicher Verlauf der Nullstellen ergibt, die die Objektoberfläche repräsentieren.

Zur Bestimmung einer solchen Funktion wurden in den vergangenen Jahren verschiedene Methoden vorgestellt. Alexa et al. und Levien berechnen eine globale Approximation, die sich aus lokal angefitteten Polynombasen zusammensetzt, die mittels eines gewichteten Least-Squares-Fits unter verschiedenen Randbedingungen berechnet werden (*Moving Least Squares, MLS*) [1, 23, 26]. Ein anderes Verfahren wurde bereits 1992 von Hoppe vorgestellt. Er berechnet eine Distanzfunktion zur Objektoberfläche, deren Vorzeichen Innnen oder Außen bestimmt. Dazu werden lokal Ebenen an benachbarte Punkte angefittet. Der Funktionswert ergibt sich dann aus dem Abstand des Raumpunktes zur nächstgelegenen Ebene. Weitere Details zu diesen Methoden finden sich in Kapitel 3.

Alle diese Verfahren erzeugen zwar eine relativ kompakte Repräsentation der Objektoberflächen, sie können aber nur sehr ineffizient visualisiert werden. Ein Standardverfahren zur Lösung dieses Problems ist es, polygonale Netze (vor allem Dreiecksnetze) aus den mathematischen Beschreibungen zu generieren. Dazu wird nahezu immer der Marching-Cubes-Algorithmus angewendet. Dieses Verfahren wird ebenfalls in Kapitel 3 näher beschrieben.

Eine Alternative zur Approximation impliziter Funktionen ist es, die Delaunay-Triangulation der Punktwolke zu berechnen. Das ist eine Zerlegung der kompakten Hülle der Messdaten in planare Dreiecke, so dass sich innerhalb des Umkreises eines Dreiecks keine weiteren Messdaten mehr befinden. Da die konvexe Hülle einer Punktwolke aber nicht unbedingt die Form der gescanten Objekte repräsentiert, ist das Verfahren zur Visualisierung nur bedingt geeignet. Eine Erweiterung dieses Ansatzes bieten die so genannten α-Shapes [16]. Das α-Shape wird bestimmt, indem aus dem Volumen der konvexen Hülle der Messpunkte die Volumenanteile entfernt werden, in die eine Kugel mit dem Radius α passt, ohne dass sie Messpunkte beinhaltet oder berührt. Für $\alpha = 0$ bleiben demnach nur die Messwerte selbst übrig. Für $\alpha = \infty$ bleibt die konvexe Hülle erhalten, da kein Raum entfernt werden kann. Bei der Implementierung dieses Verfahrens werden aus praktischen Gründen alle Elemente der Delaunay-Triangulation entfernt, deren Umkreis einen Radius kleiner als α hat. Abbildung 2.3 zeigt ein Beispiel für dieses Verfahren.

Als Alternative zum Meshing wurde 2000 von Pfister et al. und Rusinkiewicz u. Levoy zeitgleich das sogenannte Point-Splatting vorgeschlagen [36,39]. Dabei werden die Messpunkte durch kleine Scheiben ersetzt, deren Radius so gewählt ist, dass sie sich gegenseitig überlappen. Dadruch ergibt sich beim Rendern der Objekte ein flächiger Eindruck ergibt. Im Gegensatz zum Meshing wird allerdings keine Information über den Verlauf der Flächen hinzugewonnen.

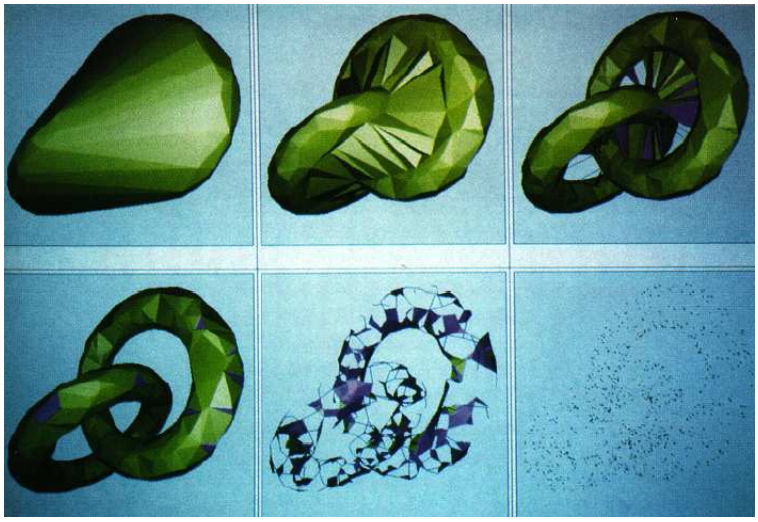


Abbildung 2.3: Beispiel für ein α-Shape bei Variation des Schwellwertes α. Oben links ist $\alpha = \infty$, unten rechts ist $\alpha = 0$. Mann kann erkennen, dass nur bei passendem Schwellwert ein gutes Ergebnis erzielt wird (Bild entnommen aus [16]).

2.3 Modelloptimierung

Die mit dem Marching-Cubes-Verfahren erzeugten Dreiecksnetze enthalten viele redundante Dreiecke, die mit geeigneten Verfahren entfernt werden müssen. Viele Verfahren basieren darauf, mit Hilfe verschiedener Metriken die Elemente zu entfernen, die den gerigsten Fehler verursachen (vgl. Kapitel 6.3).

Ein weiterer Ansatz besteht darin, neben den redundanten Elementen auch die Vertexpositionen nachträglich zu optimieren. Hoppe berechnet dazu eine globale „Energiefunktion", die durch Entfernen von Elementen und Verschieben von Vertices minimiert wird [18, 19, 21]. Diese Funktion besteht aus drei Termen:

$$E = E_{Dist} + E_{Rep} + E_{Spring}.$$

E_{Dist} ist der quadratische Abstand aller Vertices von den ursprünglichen Messpunkten. Der Term E_{Rep} ist proportional zur Anzahl der Vertices im Netz und bestraft Repräsentationen mit vielen Vertices. Der Term E_{Spring} entspricht einer Art „Federenergie" mit einer vom Benutzer festgelegten Federkonstanten, die garantieren soll, dass sich das Netz tendenziell „zusammenzieht". So wird verhindert, dass Vertices weit nach außen verlagert werden und sichergestellt, dass ein Minimum existiert.

Kapitel 3

Modellgenerierung

In diesem Kapitel wird eine Methode aufgezeigt, mit der effizient polygonale Modelle aus 3D-Laserscannerdaten erstellt werden können. Dazu wird u.a. der bekannte Marching-Cubes-Algorithmus verwendet. Ausgangspunkt bildet eine Implementation dieses Algorithmus, die bereits in meiner Bachelor-Arbeit dargelegt wurde [46]. In diesem Abschnitt wird eine wesentlich verbesserte Version vorgestellt, die eine neue Datenstruktur names *HashGrid* verwendet.

Im Folgenden sollen zunächst die Eigenschaften von Polygonnetzen beschrieben werden. Anschließend werden verschiedene Datenstrukturen diskutiert, mit denen sie repräsentiert werden können. Zuletzt wird gezeigt, wie sich mit Hilfe der HashGrid-Datenstruktur und des Marching-Cubes-Algorithmus Polygonnetze in unterschiedlichen Darstellungen generieren lassen.

3.1 Polygonnetze und ihre Repräsentation

Polygonnetze sind eine in der Computergrafik weit verbreitete Datenstruktur zur Repräsentation dreidimensionaler Objekte (vgl. [37, 49, 52]). Ein Polygonnetz ist eine endliche Menge von miteinander verbundenen planaren Polygonen, die die Oberfläche eines Objektes approximieren. Die Polygone im Netz werden als Faces oder Facetten bezeichnet.

Polygone werden durch Angabe von Vertices (Eckpunkten) und Kanten (Verbindung von zwei Vertices) definiert. Üblicherweise werden die Vertices eines Polygons gegen den Uhrzeigersinn

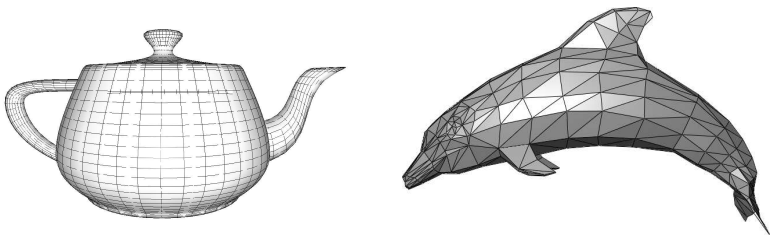


Abbildung 3.1: Beispiele für Polygonnetze. Links der Utah-Teapot [33] durch Vierecke approximiert, rechts ein durch Dreiecke repräsentierter Delfin (entnommen aus [49])

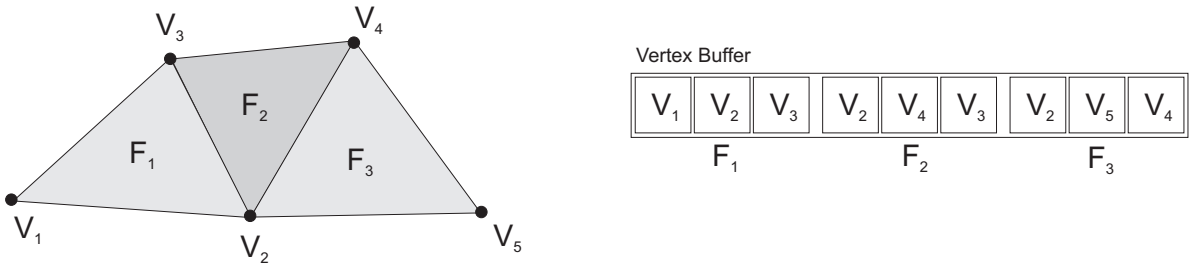


Abbildung 3.2: Ein Dreiecksnetz mit der dazugehörigen einfachen Datenstruktur bei sequenzieller Definition der Vertices. Alle Vertices, die zu mehreren Dreiecken gehören, sind mehrfach abgespeichert.

durchnummeriert. Die Verbindung zwischen zwei aufeinander folgenden Vertices bildet eine Kante. Ein Polygonnetz heißt geschlossen, falls jede Kante im Netz zu genau zwei Polygonen gehört.

Polygonnetze werden in den meisten Fällen aus Dreiecken oder Vierecken aufgebaut, da diese mit relativ geringem Aufwand gerendert werden können. Je nach Anwendung können sie aber auch aus komplexeren Faces bestehen. Zwei Beispiele für Polygonnetze zeigt Abbildung 3.1. Zur Verwaltung von Polygonnetzen gibt es verschiedene Datenstrukturen. Die gebräuchlichsten Repräsentationen werden im Folgenden kurz erläutert.

3.1.1 Einfache Datenstrukturen

Einfache Datenstrukturen zur Repräsentation polygonaler Netze enthalten keine oder nur wenige Informationen über Nachbarschaftbeziehungen zwischen Flächen, Kanten und Vertices im Netz.

Sequenzielle Definition von Vertices

Ein naiver Ansatz zur Definition eines Polygonnetzes ist, sequenziell alle Vertices eines jeden Polygons anzugeben. Vorteil dieser Datenstruktur ist, dass bei der Generierung von Oberflächen die erzeugten Vertices unabhängig von den bisher vorhandenen Polygonen direkt abgespeichert werden können. Ein Beispielnetz mit der dezugehörigen Datenstruktur zeigt Abbildung 3.2.

Der Nachteil dieser Methode ist, dass Vertices redundant im Netz gespeichert werden. Da in geschlossenen Netzen jeder Vertex zu mehreren Flächen gehört, würde es eigentlich ausreichen, ihn nur einmal zu speichern und auf ihn zu verweisen. Genau diese Idee wird bei der Eckenliste verfolgt.

Indizierter Vertex-Buffer

Ein indizierter Vertex-Buffer besteht aus zwei getrennten Listen. In der ersten Liste sind die Positionsdaten der Vertices abgespeichert (Vertex-Buffer). Die zweite Liste definiert die Polygone und besteht aus Zeigern auf die Positionen der Vertices im Vertex-Buffer (Index-Buffer, s. Abbildung 3.3). Es wird also die Geometrie (Vertexpositionen) von der Topologie (Beziehungen zwischen den Vertices) des Netzes getrennt.

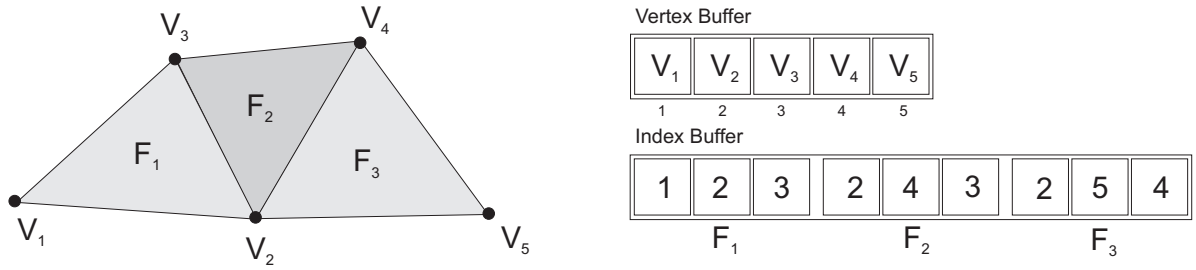


Abbildung 3.3: Beispiel für eine Eckenliste. Das Modell wird dabei durch Index- und Vertexbuffer repräsentiert. Jeder Vertex muss nur einmal gespeichert werden.

Die Vertexdefinitionen können je nach Anwendung neben den reinen Positionsdaten auch noch weitere Informationen wie Normalenvektoren oder Farbinformationen enthalten. Da im Indexbuffer nur die Zeiger auf die Vertices abgelegt werden müssen, kann viel Speicher eingespart werden, indem auf bereits im Vertexbuffer vorhandene Vertices verwiesen wird.

Kantenliste

In einer Kantenliste werden die Faces im Polygonnetz durch Zeiger auf ihre Kanten definiert. Kanten bestehen in dieser Repräsentation aus Zeigern auf die beiden beteiligten Vertices und die beiden angrenzenden Faces. Es werden also bei dieser Datenstrukter drei Listen benötigt: Ein Vertex-Buffer, der die Vertexdefinitionen enthält, ein Edge-Buffer, in dem die Kanten des Netzes abgelegt sind, und ein Face-Buffer, der die Faces durch Zeiger auf die Begrenzungskanten definiert (s. Abbildung. 3.4)

Eine Kantenliste ermöglicht es, in linearer Zeit (in der Anzahl der Kanten eines Faces) alle Nachbarn zu ermitteln, indem die Kanten einer Fläche traversiert und die Zeiger auf den anderen Nachbarn der Kante ausgewertet werden. Zudem können Randkanten in einem Netz schnell gefunden werden, da diese nur einen Zeiger auf ein zugehöriges Face haben. Komplexe Abfragen (z.B. alle Faces, die sich einen Vertex teilen) sind aber, wie in der Eckenliste, nur sehr ineffizient möglich.

3.1.2 Komplexe Datenstrukturen

In komplexen Datentrukturen werden neben den Defintionen der Faces im Polygonnetz noch weitere Informationen über die Topologie abgelegt. Dafür werden die Kanten und Polygone geschickt verzeigert. Bekannte Datenstrukturen sind die Winged-Edge-Datenstruktur und die Half-Edge-Datenstruktur.

Die Winged-Edge-Datenstruktur

Die Winged-Edge-Datenstruktur ist die älteste Datenstruktur mit der Nachbarschaftsbeziehungen in Polygonalen Netzen abgespeichert werden können. Sie wurde bereits 1975 von Baumgart

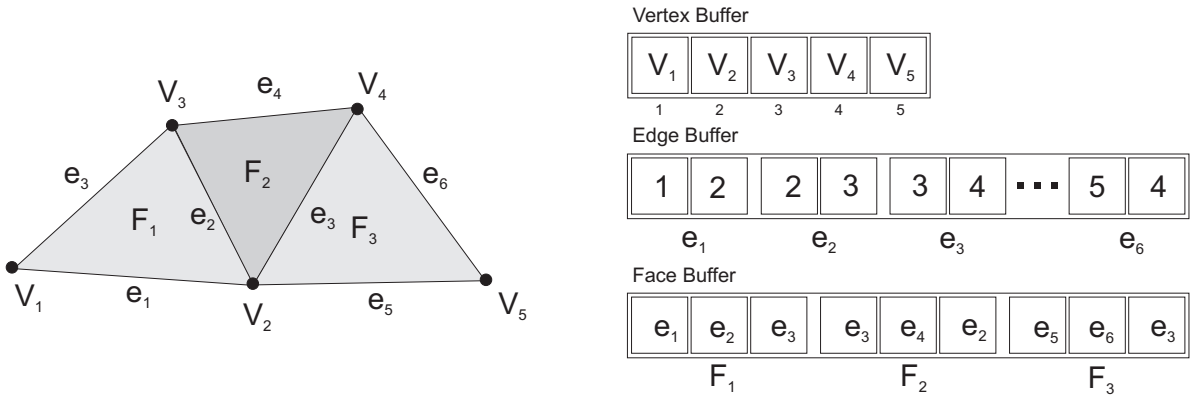


Abbildung 3.4: Ein Dreiecksnetz mit den Datenstrukturen für eine Kantenliste. Im Vertexbuffer werden die Vertexdefinitionen gespeichert. Im Edge-Buffer sind die Kanten abgespeichert. Die Polygone im Netz werden durch die Zeiger auf die begrenzenden Kanten im Face-Buffer definiert.

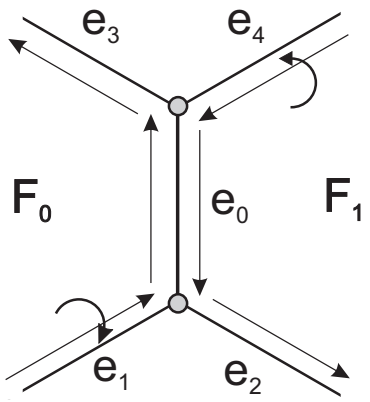


Abbildung 3.5: Kanten, Vertices und Faces in der Winged-Edge-Datendtruktur

beschrieben [8].

Die Grundidee dieser Datenstruktur ist, dass neben Zeigern auf die Vertices und Nachbarn einer Kante auch Zeiger auf ihre Vorgänger und Nachfolger abgelegt werden. Werden die Vertices aller Polygone im Netz gegen den Uhrzeigersinn definiert, wird bei der Traversierung aller Faces im Netz jede Kante zweimal durchlaufen, allerdings in entgegengesetzten Richtungen. Daher müssen insgesamt vier Zeiger pro Kante abgelegt werden, nämlich die Vorgänger und Nachfolger, die beim Durchlaufen der Kanten eines Faces gegen den Uhrzeigersinn auf die aktuelle Kante folgen. Die Faces im Netz benötigen nur noch einen Zeiger auf eine beliebige Begrenzungskante. Von dieser aus können sie durch Aufruf der entsprechenden Nachfolgerkante gegen den Uhrzeigersinn traversiert werden.

Abbildung 3.5 soll diese Tatsachen verdeutlichen. Die Kante e_0 wird bei der Traversierung der Fläche F_0 von unten nach oben durchlaufen. Die Vorgängerkante ist e_1. Bei der Traversierung von F_1 wird e_0 hingegen von oben nach unten durchlaufen. Vorgänger und Nachfolger sind nun

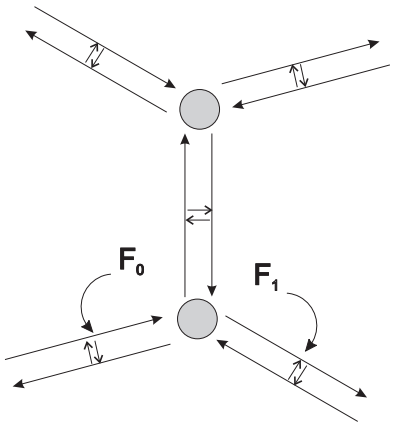


Abbildung 3.6: Verzeigerung der Kanten und Faces in der Half-Edge-Datenstruktur

e_4 bzw. e_2. Daher müssen Zeiger auf diese vier Kanten in e_0 abgelegt werden. Sie bilden die *„Flügel"* dieser Kante. Daher der Name Winged-Edge. Zusätzlich erhält jeder Vertex Zeiger auf eine von ihm ausgehende Kante. So können insgesamt neun verschiedene Abfragen realisiert werden: Welche Kante, Ecke oder welches Face gehört zu jeder Kante, Ecke oder jedem Face [49]?

Die Half-Edge-Datenstruktur

Die Half−Edge−Datenstruktur speichert dieselben Informationen wie die Winged−Edge−Datenstruktur. Allerdings können einige Abfragen etwas effizienter erfolgen. In dieser Darstellung wird die Tatsache, dass Kanten im Polygonnetz von benachbarten Faces in verschiedenen Richtungen durchlaufen werden, berücksichtigt, indem jede Kante der Länge nach in zwei gerichtete Halbkanten aufgespalten wird (s. Abbildung 3.6). Jede dieser Halbkanten enthält einen Zeiger auf das Face, zu dem sie bei der Traversion gegen den Uhrzeigersinn gehört. Damit die Nachbarschaftsbeziehungen zwischen den Faces nicht verloren gehen, besitzt jede Halbkante einen Zeiger auf die jeweilige Partnerkante. Zusätzlich enthalten die Vertices im Polygonnetz eine Liste mit Zeigern auf alle Halbkanten, die aus ihnen herauslaufen.

3.1.3 Vergleich der Datenstrukturen

Einfache Datenstrukturen haben den Vorteil, dass sie leicht zu generieren sind und wenig Speicher verbrauchen. Dafür liegen aber nur wenige Informationen über die Topologie vor. Zur reinen Repräsentation eines Polygonnetzes sind Eckenlisten optimal, da keine Vertices redundant gespeichert werden. Sollen aber Operationen wie z.B. das Entfernen eines Faces im Netz durchgeführt werden, benötigt man verzeigerte Datenstrukturen wie Winged-Edge oder Half-Edge, um die beteiligten Objekte bestimmen zu können. Tabelle 3.1 zeigt die Komplexitäten verschiedener topologischer Abfragen in den unterschiedlichen Datenstrukturen.

Man kann sehen, dass alle Abfragen, die die Kanten betreffen, in der Eckenliste nicht verfügbar sind, da dort keine Kanten definiert sind. Abfragen nach Nachbarschaftsbeziehungen haben in

Tabelle 3.1: Übersicht über die Komplexitäten topologischer Abfragen in verschiedenen Repräsentationen von Polygonnetzen. Dabei bezeichnet k die Anzahl aller Kanten im Netz, k_f die Anzahl der Kanten eines Faces, k_ν die Anzahl der Kanten, die einem Vertex entspringen, und f die Anzahl der Faces im Netz (entnommen aus [49]).

Abfrage	Ind. Vertex-Buffer	Kantenliste	Winged-Edge	Half-Edge
Kante → Ecken	nicht möglich	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante → Faces	nicht möglich	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante → angrenzende Kanten	nicht möglich	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(k_\nu)$
Eckpunkt → Kanten	$\mathcal{O}(f \cdot k_f)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$O(k_\nu)$
Eckpunkt → Faces	$\mathcal{O}(f \cdot k_f)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(k_\nu)$
Face → Kanten	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$
Face → Eckpunkte	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$
Face → angrenzende Faces	$\mathcal{O}(f \cdot k_f^2)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$

dieser Datenstruktur eine Laufzeit, die linear von der Anzahl der Faces abhängt, da alle Faces einmal überprüft werden müssen, um Nachbarn zu finden.

Die anderen vorgestellten Datenstrukturen erlauben alle Abfragen, die Laufzeiten variieren aber. Kantenliste und Winged-Edge haben dabei dieselben Laufzeiten, die Winged-Edge benötigt aber weniger Speicher, denn für die Definition der Faces muss nur ein Zeiger auf eine Kante abgelegt werden. Die Half-Edge-Datenstruktur ist aufgrund der Tatsache, dass die Vertices Listen mit Zeigern auf alle von ihnen ausgehenden Kanten haben, von der Laufzeit her am effizientesten. Allerdings ist der Speicherdedarf höher, da jedes Objekt in der Datenstruktur mehrere Zeiger enthält.

3.2 Der Marching-Cubes-Algorithmus

Der Marching-Cubes-Algorithmus wurde 1987 von Wiliam E. Lorensen und Harvey E. Cline [25] vorgestellt. Es handelt sich dabei um ein Verfahren, das ein Dreiecksnetz aus einer impliziten Flächenbeschreibung generiert (*Isosurface-Approximation*). Eine implizite Flächenbeschreibung hat die Form $f(\mathbf{p}) = c$, wobei c eine Konstante und $\mathbf{p} = (x, y, z)^T$ ein Punkt im $\mathbb{R}^3$ ist. Anhand der Konstante c kann bestimmt werden, wo sich ein Punkt relativ zur Fläche befindet. Es gilt: $f(\mathbf{p}) < c$, falls sich der Punkt unterhalb der Fläche befindet, $f(\mathbf{p}) = c$, falls sich der Punkt auf der Fläche befindet, und $f(\mathbf{p}) > c$, falls sich der Punkt oberhalb der Fläche befindet.

Dieses Verfahren wurde ursprünglich zur Visualisierung der Ergebnisse von Computertomografien verwendet, da die CT-Daten für unterschiedliche Gewebeschichten andere Werte erfassen. So lassen sich z.B. die verschiedenen Gewebeschichten im menschlichen Körper durch Veränderung des Schwellenwertes darstellen. Durch geeignete Interpolationsverfahren lässt sich dieses Verfahren aber auch auf Laserscanner-Daten anwenden.

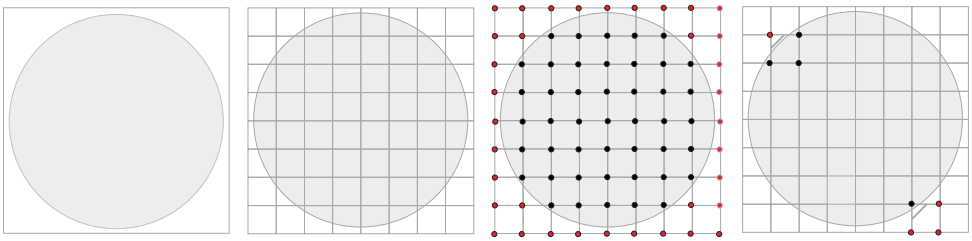


Abbildung 3.7: Beispiel für Linienapproximation eines Kreises mit Hilfe des Marching-Squares-Algorithmus. Die Ebene wird in quadratische Zellen unterteilt, und für jeden Eckpunkt wird bestimmt, ob er sich außerhalb der Fläche befindet (rot = außen). Ganz links sind zwei komplementäre Zellbelegungen mit der entstehenden Approximationslinie zu sehen.

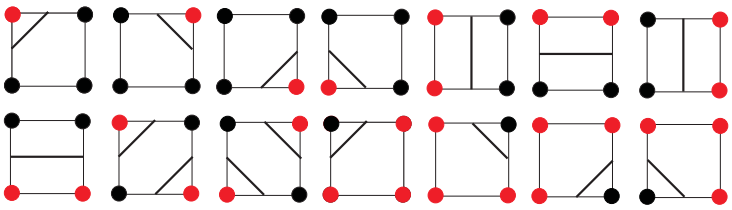


Abbildung 3.8: Mögliche Linienzüge beim Marching-Squares-Algorithmus. Man kann erkennen, dass komplementäre Belegungen die gleichen Linien erzeugen.

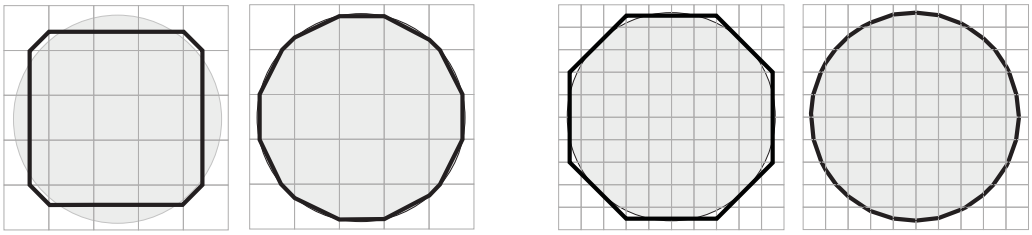
3.2.1 Marching-Squares

Zur Veranschaulichung des Grundprinzips soll das Verfahren zunächst nur in zwei Dimensionen beschrieben werden. Dabei werden aus Punktdaten zusammenhängende Linienzüge erzeugt.

Im zweidimensionalen Fall liegen alle Datenpunkte in einer Ebene. Diese wird in ein regelmäßiges Gitter aus Quadraten unterteilt (daher *Marching-Squares*). Jedes Quadrat hat vier Ecken, für die bestimmt wird, ob sie sich innerhalb oder außerhalb der zu approximierenden Fläche befinden. Für jede Belegungskombination lässt sich nun der Verlauf der Fläche innerhalb der Zellen durch Geraden approximieren. Liegt zum Beispiel nur ein Eckpunkt außerhalb der Fläche, wird eine Gerade durch die Mittelpunkte der an diese Ecke des Quadrates angrenzenden Kanten gezogen (s. Abbildung 3.7). Insgesamt ergeben sich im Zweidimensionalen $2^4 = 16$ mögliche Kombinationen von Belegungen. Da komplementäre Belegungen gleiche Approximationen ergeben, können insgesamt nur 8 Linienzüge entstehen. Diese sind in Abbildung 3.8 dargestellt. Befinden sich alle Ecken innerhalb oder außerhalb der Fläche, werden keine Linien erzeugt.

3.2.2 Erweiterung auf drei Dimensionen: Marching-Cubes

Im dreidimensionen Fall wird der von den Messdaten eingenommene Raum in würfelförmige Zellen eingeteilt. Wie im zweidimensionalen Fall wird dann ermittelt, welche der acht Ecken eines Würfels sich außerhalb der Isofläche befinden. Insgesamt können dabei 256 verschiedene Belegungskombinationen vorkommen. Allerdings reichen 16 Grundmuster zur Flächenapproximation



(a) (b)

Abbildung 3.9: Verbesserung der Approximation im zweidimensionalen Fall durch Berechnung der Kantenschnittpunkte für zwei verschiedene Gitterkonstanten. Links sind jeweils die Muster zu sehen, die bei Verwendung des Kantenmittelpunktes verwendet werden. Selbst bei relativ grober Gitterauflösung kann so noch ein gutes Ergebnis erreicht werden s. Bild (a) rechts.

aus (s. Abbildung 3.14), da komplementäre Belegungskombinationen die gleichen Approximationsmuster liefern und es zusätzlich auch noch Rotationssymmetrien gibt.

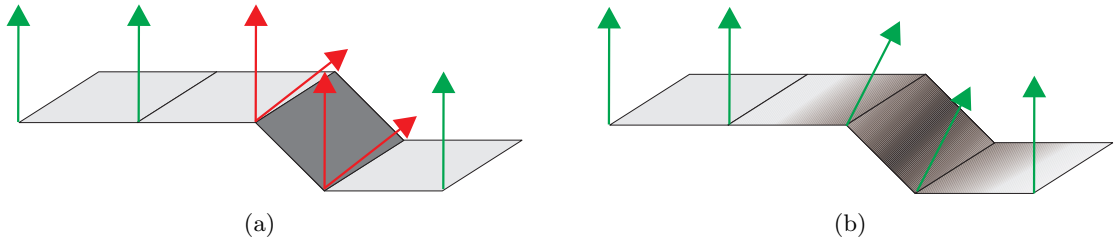
Die Approximationsmuster, die den Verlauf der Fläche innerhalb einer Zelle approximieren, bestehen aus bis zu vier Dreiecken, die Punkte auf den Würfelkanten als Vertices haben. Welche Punkte bei welchem Belegungsmuster verwendet werden, steht in einer zuvor berechneten Tabelle. Jede Zeile der Tabelle repräsentiert ein Muster und besteht aus mehreren Indizes, die auf Punkte auf den zwölf Würfelkanten verweisen. Jeweils drei aufeinanderfolgende Indizes definieren dabei ein Dreieck. Auf diese Art und Weise können Approximationen in konstanter Zeit bestimmt werden.

Die einzige Operation, die zur Bestimmung des zu einer Belegungskombination gehörenden Musters nötig ist, ist die richtige Zeile in der Tabelle aufzufinden. Dazu wird jeder Konfiguration ein Index zugewiesen, der diese Zeile repräsentiert. Dieser Index besteht aus einem 8-Bit Integer-Wert, in dem jedem Bit eine Ecke im Volumenelement zugeordnet ist. Befindet sich eine Ecke außerhalb der Fläche, wird das entsprechende Bit auf 1 gesetzt, ansonsten auf 0.

Im Standardfall werden die Mittelpunkte der Würfelkanten zur Definition der Vertices genommen. Dies hat den Nachteil, dass durch die vorgegebene Kantenlänge des Würfels eine Art "Treppeneffekt" entsteht. Eine Möglichkeit, dies zu verhindern, ist, den Schnittpunkt der Fläche mit der Kante durch lineare Interpolation der impliziten Funktionswerte an den Würfelecken zu ermitteln. Dadurch entstehen weichere Übergänge zwischen den Approximationsmustern und das erzeugte Modell wird deutlich besser. Zwei Beispiele im zweidimensionalen Fall zeigt Abbildung 3.9.

3.2.3 Berechnung der Vertexnormalen

Um ein Polygonnetz beleuchtet rendern zu können, wird für jeden Vertex ein Normalenvektor benötigt. Das ist ein Vektor, der senkrecht auf der zum Vertex gehörenden Fläche steht und nach außen zeigt (s. Abbildung 3.10(a)). Die Länge der Normalen wird dabei auf den Betrag 1 normiert.



(a) (b)

Abbildung 3.10: Interpolation der Vertexnormalen. An den Übergängen zwischen verschiedenen Ebenen gibt es pro Vertex zwei mögliche Normalen (rot). Es ist daher nur möglich, das Modell mit Flat-Shading zu rendern (a). Nach Mittelung der roten Normalen kann das Modell mit Gouraud-Shading gerendert werden (b). Es entsteht ein weicherer Farbübergang und somit auch ein plastischerer Eindruck.

Eine einfache Möglichkeit, den Vertices Normalen zuzuweisen, ist, diese mit Hilfe des Kreuzproduktes aus zwei Kanten eines Dreieckes zu berechnen und anschließend zu normieren. Dazu wird der zweite Vertex, der in der Marching-Cubes-Tabelle angegeben ist, als Referenzpunkt verwendet. Die so berechnete Normale wird anschließend allen zum Dreieck gehörenden Vertices zugewiesen. Da die Vertices der Dreiecke in der Tabelle immer gegen den Uhrzeigersinn nummeriert sind, wird gewährleistet, dass die Normale immer nach außen zeigt.

$$\boldsymbol{n} = (\boldsymbol{v}_1 - \boldsymbol{v}_2) \times (\boldsymbol{v}_1 - \boldsymbol{v}_3)$$

Mit Hilfe dieses Verfahrens können die Normalen nur einzeln für jedes Dreieck berechnet werden. Es werden bei der Beleuchtung aber keine stetigen Farbverläufe erziehlt, da Vertices, die an einem Knick im Netz liegen, von den angrenzenden Dreiecken verschiedene Normalen zugewiesen bekommen. Dieses Problem kann gelöst werden, indem man die Normalen an den Knicken mittelt. So wird ein kontinuierlicher Übergang zwischen verschiedenen Flächen erreicht (s. Abbildung 3.10).

Ist die implizite Funktion $F(x.y, z)$, die approximiert werden soll, bekannt können die Normalen auch direkt durch Gradientenbildung in den einzelnen Vertices berechnet werden. Dies kann je nach Komplexität der Funktion allerdings sehr aufwändig sein. Zudem muss die genaue Formel der zu approximierenden Funktion bekannt sein. Dies ist in der Regel aber nicht der Fall.

$$\boldsymbol{n} = \left(\frac{\partial F}{\partial x}, \frac{\partial F}{\partial y}, \frac{\partial F}{\partial z} \right)^T$$

3.2.4 Inkonsistente Flächenkonfigurationen

Durch die Kombination der verschiedenen Muster entsteht in der Regel ein geschlossenes Dreiecksnetz. Allerdings können durch ungünstige Konfigurationen Löcher im Netz auftreten (s. Abbildung 3.11). Dieses Problem kann durch zusätzliche Approximationsmuster gelöst werden, die im Falle einer ungünstigen Kombination das Loch beseitigen (s. Abbildung 3.18).

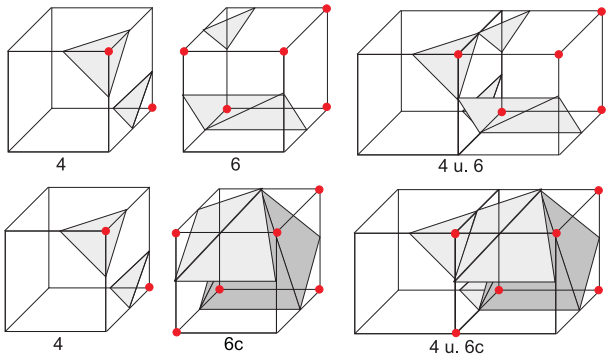


Abbildung 3.11: Beispiel für eine inkonsistente Musterkombination beim Marching-Cubes-Algorithmus. Durch Kombination der Grundmuster 4 und 6 entsteht ein Loch im Polygonnetz (oben). Mit Hilfe des zusätzlichen Musters 6c kann dieses geschlossen werden (unten)

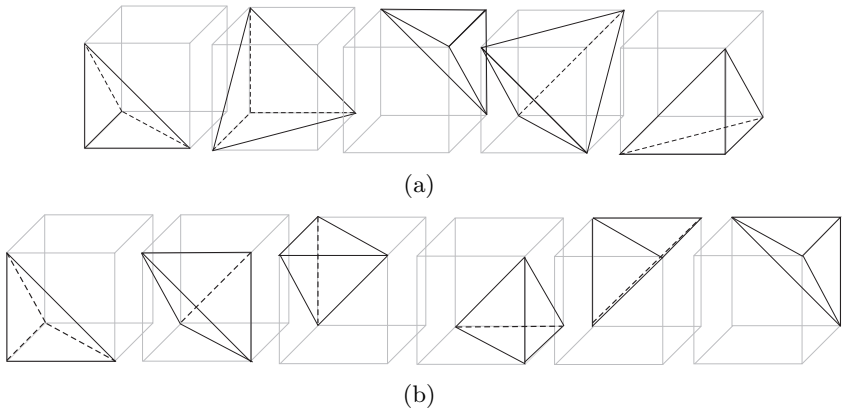


Abbildung 3.12: Tetraederzerlegungen eines Würfels. Oben ist die Zerlegung in 5 Zellen gezeigt. Das untere Bild zeigt die Zerlegung in 6 Tetraeder, die beim Marching-Tetrahedra-Algorithmus benutzt wird.

Eine andere Möglichkeit ist, statt einer würfelförmigen Zerlegung eine Tetraederzerlegung des Raums zu wählen. Dabei werden die ursprünglichen Würfel weiter in 5 oder 6 Tetraeder zerlegt (s. Abbildung 3.12). Üblicherweise wird die Zerlegung in 6 Tetraeder gewählt, da diese dann immer gemeinsame Kanten mit den Nachbarn haben. In einer solchen Zerlegung treten nur sieben Approximationsmuster (s. Abbildung 3.13) auf, die in allen Kombinationen immer geschlossene Flächen liefern. Dieses Verfahren wird *Marching-Tetrahedra* genannt [35, 45]. Es hat jedoch den Nachteil, dass mehr Dreiecke generiert werden als bei der Unterteilung in Würfel, da jeder Tetraeder innerhalb des Würfels ein Dreieck erzeugen kann. Es können also pro Würfel anstatt vier bis zu sechs Dreiecke entstehen. Im Mittel werden etwa doppelt so viele Dreiecke erzeugt wie bei einer Würfelzerlegung.

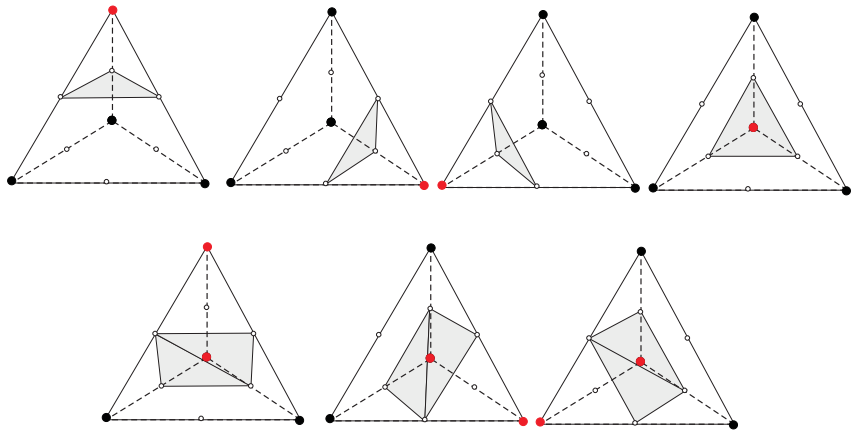


Abbildung 3.13: Grundmuster des Marching-Cubes-Algorithmus

3.2.5 Bestimmung der Marching-Cubes-Indizes

Damit der Marching-Cubes-Algorithmus angewendet werden kann, muss nach der Unterteilung des Raumes in gleich große Zellen festgestellt werden, welche Ecken sich innerhalb und außerhalb der zu modellierenden Fläche befinden. Da die Laserscannerdaten keine Flächenbeschreibung, sondern nur einzelne Punkte auf der Oberfläche eines Objektes liefern, werden Verfahren benötigt, die den Verlauf der Oberfläche approximieren.

Ein mögliches Verfahren ist es, eine implizite Funktion an die Messdaten anzufitten und so den Flächenverlauf zu bestimmen. Ein weiteres Verfahren wurde von Hughes Hoppe [21] vorgestellt. Er approximiert die Fläche lokal durch kleine tangentiale Ebenen und berechnet einen mit Vorzeichen versehenen Abstand der Eckpunkte zu der nächsten Ebene. Da dies die Standardverfahren sind, sollen sie im Folgenden kurz vorgestellt werden. Im Anschluss soll gezeigt werden, wie die Messdaten ohne das Interpolieren eines Flächenverlaufs direkt ausgewertet werden können.

Interpolation einer impliziten Funktion

Ziel ist es, eine implizite Funktion zu finden, die in den Messdaten $\boldsymbol{p}_i$ den Wert Null annimmt:

$$f(\boldsymbol{p}_i) = 0$$

Zur Interpolation werden in der Regel trivariante Polynombasen mittels geeigneter Verfahren (Least-Squares-Fit, Weighted Least Squares, usw.) numerisch bestimmt [26]. Einen guten Überblick über verschiedene Interpolationsverfahren liefert [32]. Damit nicht die triviale Lösung $f(\boldsymbol{p}_i) = \mathbf{0}$ gefunden wird, werden noch weitere Randbedingungen benötigt. Diese werden unter Verwendung der Normalen $\boldsymbol{n}_i$ in den Messpunkten definiert:

$$f(\boldsymbol{p}_i \pm \alpha \cdot \boldsymbol{n}_i) = \pm\alpha$$

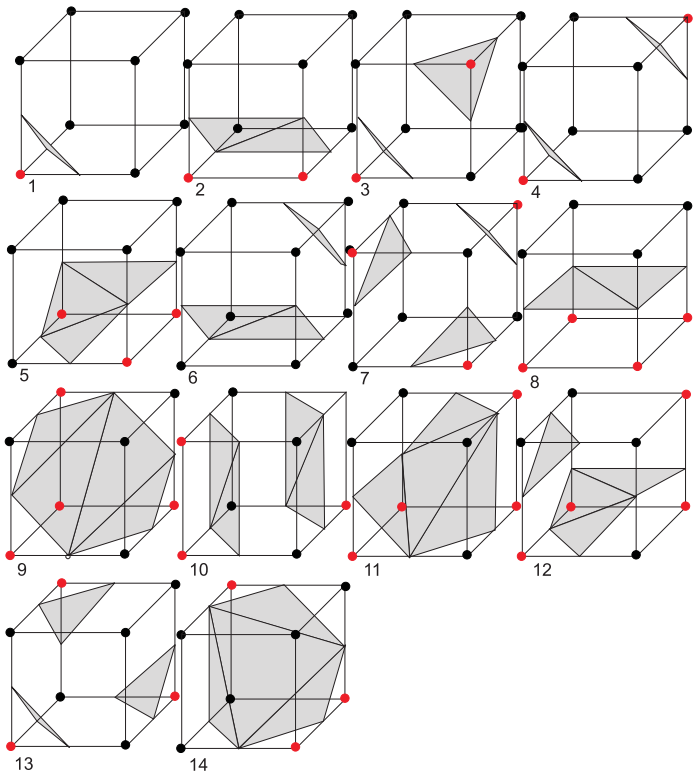


Abbildung 3.14: Grundmuster des Marching-Cubes-Algorithmus

Diese Randbedingungen sorgen dafür, dass sich der Betrag der Fit-Funktion im Bereich von $\pm\alpha'$ linear mit der Entfernung von der Oberfläche des abgetasteten Objektes vergrößert. Das Vorzeichen sorgt dafür, dass die Information zwischen Innen und Außen erhalten bleibt. Zudem werden dadurch unvorhergesehene Funktionsverläufe, wie sie bei einer polynomialen Approximation auftreten können, verhindert. Auf diese Art und Weise kann bei einer späteren Anwendung des Marching-Cubes-Algorithmus der tatsächliche Schnittpunkt einer Würfelkante mit einer angefitteten Fläche genau durch lineare Interpolation bestimmt werden. Die benötigten Normalen werden entweder direkt beim Scannen der Objekte ermittelt oder im Nachhinein unter Einbeziehung der Nachbarpunkte berechnet.

So entsteht bei n Messpunkten unter Berücksichtigung dieser Nebenbedingungen ein Gleichungssystem mit $3n$ Gleichungen. Daher wird bei vielen Messwerten der Aufwand zur Lösung des Systems sehr hoch. Eine Methode, das System zu reduzieren ist, zufällig eine bestimmte Anzahl von Messpunkten auszuwählen. Anschließend werden die Punkte, an denen die Approximation nicht genau genug ist, iterativ dazugenommen, bis die Approximationsgüte hoch genug ist. Ist eine Funktion gefunden, die den oben genannten Randbedingungen genügt, kann der Marching-Cubes-Index einer Raumzelle direkt durch Auswertung dieser Funktion an den Ecken des Würfels bestimmt werden.

Verfahren nach Hoppe

Der Ansatz von Hoppe besteht darin, anstelle einer impliziten Funktion eine *vorzeichenbehaftete Distanzfunktion* $d_U(\boldsymbol{p}) = s(\boldsymbol{p}) \cdot d(\boldsymbol{p}, U)$ zu bestimmen, wobei $d(\boldsymbol{p}, U)$ der Abstand des Punktes $\boldsymbol{p}$ zur Oberfläche U des Objektes ist und $s(\boldsymbol{p}) = \pm 1$, je nachdem, auf welcher Seite der Fläche $\boldsymbol{p}$ liegt [21].

Um d_U zu ermitteln, werden für jeden Messpunkt tangentiale Ebenen $Tp(p_i)$ aus den umgebenden Messpunkten mittels eines Least-Squares-Fits berechnet. Diese werden durch den Schwerpunkt o der betrachteten Umgebungspunkte und der Flächennormalen definiert:

$$Tp(o_i) = \boldsymbol{o}_i \cdot \boldsymbol{n}_i$$

Die Größe der betrachteten Umgebung wird dabei von der Dichte der Messpunkte ρ und dem Rauschen des Scanners δ bestimmt. Diese Parameter müssen vom Benutzer abgeschätzt werden. Der Abstand des Punktes von der zu approximierenden Fläche wird berechnet, indem die Entfernung zwischen $\boldsymbol{p}$ und der Projektion dieses Punktes in die nächstgelegene Tangentialebene bestimmt wird.

$$d_U = (\boldsymbol{p} - \boldsymbol{o}_i) \cdot \boldsymbol{n}_i$$

Zur Bestimmung des Vorzeichens $s(\boldsymbol{p})$ wird die Orientierung der Flächennormalen ausgewertet. Zeigt sie in Richtung $\boldsymbol{p}$, ist $s(\boldsymbol{p})$ positiv, ansonsten negativ. Die Berechnung der Normalen erfolgt durch Komponentenanalyse der Umgebungspunkte. Da die Reihenfolge, in der die Punkte betrachtet werden, von Bedeutung ist, müssen die Vorzeichen benachbarter Ebenen konsistent zugewiesen werden.

Auf diese Art und Weise können analog zur Verwendung einer impliziten Funktion die Belegung der Eckpunkte der Marching-Cubes-Boxen und der Schittpunkt der Approximationsfläche mit den Würfelkanten bestimmt werden.

Direktes Bestimmen aus den Laser-Messdaten

Die vorab vorgestellten Methoden zur Bestimmung der Belegungskonfigurationen der Marching-Cubes-Zellen haben zwar den Vorteil, dass der Schnittpunkt der vermessenen Oberfläche mit den Würfelkanten exakt bestimmt werden kann, allerdings wird zur Interpolation dieser Fläche sehr viel Rechenzeit benötigt (je nach Verfahren und Anzahl der Messpunkte bis zu mehreren Stunden).

Bei Anwendungen in der mobilen Robotik steht allerdings die direkte Auswertung der Daten, im besten Fall online auf dem Roboter, im Vordergrund. Um Rechenzeit zu sparen wird daher im Folgenden darauf verzichtet, eine globale Approximation der Messdaten zu berechnen. Vielmehr sollen die Zellenkonfigurationen direkt beim Einlesen der Messpunkte bestimmt werden. Dazu wird zu jedem Messpunkt die dazugehörige Zelle bestimmt. Anschließend wird der Wert der Ecke, die dem Messpunkt am nächsten ist, auf „außerhalb" gesetzt. Die Punkte werden also quasi auf

den nächstgelegenen Raumgitterpunkt verschoben. Bei der Anwendung des Marching-Cubes-Algorithmus werden die Kantenmittelpunkte als Vertices der Approximationsmuster verwendet.

Die so entstehenden Modelle weisen zwar den zuvor angesprochenen Treppeneffekt auf, jedoch kann bei diesem Verfahren mit hinreichend kleiner Gitterkonstante ein gutes Ergebnis erzielt werden. Auf diese Art und Weise können auch aus großen Datensätzen innerhalb weniger Sekunden akzeptable Rekonstruktionen erzeugt werden (vgl. Kapitel 6).

```
procedure MarchingCubes
    for all Cells do
        index ← index of corner configuration
        i ← 0
        while Marching-Cubes-Table[index][i] != −1 do
            for j = 0 to 3 do
                EdgeNumber ← Marching-Cubes-Table[index][i+j]
                VertexBuffer ← Intersection on Edge[EdgeNumber]
            end for
            calculate Normal for generated Face
            i ← i + 3
        end while
    end for
end procedure
```

Algorithmus 3.1: Der Marching-Cubes-Algorithmus in der einfachen Form. Dabei wird nur ein nicht indizierter Vertexbuffer erstellt.

3.3 Aufbau eines Polygonnetzes

Im Folgenden wird beschrieben, wie effizient zwei verschiedene Darstellungen eines Polygonnetzes direkt bei der Erzeugung durch den Marching-Cubes-Algorithmus generiert werden können. Zum einen soll eine Darstellung durch Index- und Vertexbuffer gewonnen werden, die zum direkten Rendern des Modells verwendet werden kann. Zum anderen soll eine Halbkantendarstellung erzeugt werden, die topologische Abfragen im Netz ermöglicht. Dazu wird die *HashGrid-Datenstruktur* eingeführt, die die Raumaufteilung für den Marching-Cubes-Algorithmus verwaltet.

3.3.1 Die HashGrid-Datenstruktur

Damit der Marching-Cubes-Algorithmus angewendet werden kann, muss der durch die Messdaten eingenommene Raum in würfelförmige Zellen unterteilt werden. Da nicht der gesamte Raum ausgefüllt ist, sollen nur die Zellen erzeugt werden, die auch wirklich Messpunkte enthalten.

Für die spätere Erzeugung des Polygonnetzes werden zusätzlich auch Informationen über die Nachbarschaftsbeziehungen zwischen den Würfeln benötigt. Die einzelnen Zellen müssen also entsprechend verzeigert werden. Dies geschieht in der HashGrid-Datenstruktur.

Beschreibung der Datenstruktur

Die HashGrid-Datenstruktur ist eine Repräsentation eines dreidimensionalen Gitters und besteht aus würfelförmigen Zellen, die die Messdaten umgeben. Die Position jeder Zelle wird durch die Angabe von drei ganzzahligen Indizes i, j und k eindeutig im Gitter identifiziert (analog zu einem dreidimensionalen Array). Allerdings werden die Zellen in einer STL-Hashmap gespeichert. Das hat den Vorteil, dass man nur die Zellen speichern muss, die auch wirklich Daten enthalten. Diese können dann mit Hilfe eines Iterators sequentiell durchlaufen werden. Es wird also beim Anwenden des Marching-Cubes-Algorithmus eine lineare Laufzeit in der Anzahl der vorhandenen Zellen erreicht. Ein weiterer Vorteil ist, dass die Zellen durch Angabe eines Hashwertes in konstanter Zeit angesprochen werden können. Der Hashwert berechnet sich folgendermaßen:

$$H(i,j,k) = i \cdot maxIndex_x^2 + j \cdot maxIndex_y + k$$

Die Konstanten $maxIndex_x$ und $maxIndex_y$ sind dabei die in der entsprechenden Raumrichtung maximal möglichen Indizes. Solange $maxIndex_x$ und $maxIndex_y$ verschieden sind, liefert diese Funktion für jede Kombination von i, j und k einen eindeutigen Wert. Dies ist in der Regel der Fall. Sollte aber zufällig $maxIndex_x = maxIndex_y$ gelten, wird $maxIndex_y$ um 1 erhöht, um eine ideale Hashfunktion zu erhalten.

Zellen im Gitter werden durch eine eigene Klasse repräsentiert. Sie werden untereinander verzeigert, so dass Nachbarschaftsinformationen zwischen den Zellen ohne Neuberechnung des Hashwertes abgefragt werden können. Dabei werden nicht nur die acht direkt benachbarten Zellen verlinkt, sondern alle Zellen, die mindestens eine gemeinsame Kante mit der aktuellen Zelle haben. Maximal kann jede Zelle also 26 Nachbarn haben, die geeignet verlinkt werden müssen (s. Abbildung 3.15). Die Nachbarschaftsinformationen könnten zwar auch über die Indizes abgefragt werden; es hat sich aber gezeigt, dass die Zeigerrepräsentation eine bessere Kodierung ermöglicht (vgl. Kapitel 3.3.2).

Neben den Nachbarschaftbeziehungen werden in den Zellen auch alle weiteren Informationen gespeichert, die zum Erzeugen eines Polygonnetzes unter Verwendung des Marching-Cubes-

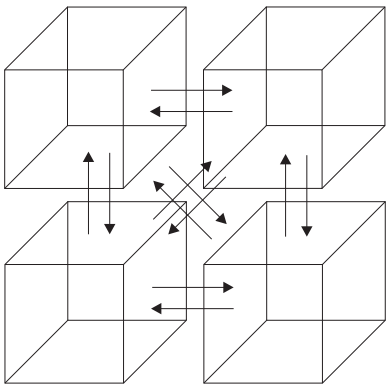


Abbildung 3.15: Beispiel einer Verlinkung von vier Hashgrid-Zellen. Es werden dabei nicht nur die Zellen verlinkt, die eine geminsame Fläche haben, sondern auch die mit gemeinsamen Kanten.

Algorithmus erforderlich sind: u.a. die Positionen der Schnittkanten, die Belegung der Eckkanten und einen Index-Buffer, der das Approximationsmuster innerhalb der Zelle repräsentiert.

Aufbau des Raumgitters

Um die für die Berechnung der Hashwerte benötigten Konstanten $maxIndex_x$ und $maxIndex_y$ zu bestimmen, wird beim Einlesen der Messdaten die maximale Ausdehnung des eingenommenen Raumes bestimmt. Dazu werden die größten und kleinsten auftretenden Komponenten in jeder Raumrichtung gespeichert (min_x, min_y, min_z bzw. max_x, max_y, max_z). Der von den Messdaten eingenommene Raum wird in Zellen mit einer vorgegebenen Kantenlänge l unterteilt. Nun kann für jeden Messpunkt $\boldsymbol{p}$ ermittelt werden, in welcher Zelle des Gitters er sich befindet:

$$i = \text{floor}\left(\frac{p_x - min_x}{l}\right)$$

Analog werden die Indizes j und k unter Verwendung der y- bzw. z-Komponenten bestimmt. Mit ihrer Hilfe wird überprüft, ob die Zelle, die den Messpunkt umgibt, bereits im Gitter existiert. Ist dies der Fall, wird der Punkt der Zelle hinzugefügt. Ansonsten muss eine neue Zelle an der entsprechenden Stelle erzeugt und korrekt verzeigert werden. Dazu werden die Indizes im Bereich von -1 bis 1 variert und die entprechenden Positionen abgefragt. Zeiger auf alle Zellen, die bereits im Gitter existieren, werden in der Zelle gespeichert.

Sobald eine Zelle neu in das Gitter eingefügt wird, müssen auch die Zeiger der bereits vorhandenen Nachbarzellen aktualisiert werden. Dies geschieht mit Hilfe einer Tabelle, die angibt, welcher Zeiger in Abhängigkeit der Position der neuen Zelle im Nachbarn gesetzt werden muss.

Ein Beispiel: Eine neue Zelle wird rechts neben einer bereits bestehenden Zelle eingefügt. Wird bei der Verzeigerung die alte, rechts von ihr liegende Zelle gefunden, ist klar, dass die Referenz auf diese Zelle im Zeiger „Links" liegen muss. In der alten Zelle muss aber der Zeiger „Rechts" angepasst werden. Genau diese Verhältnisse stehen in der Aktualisierungstabelle.

Vorteile des HashGrids

Im Gegensatz zu einfachen Strukturen wie z.B. dreidimensionalen Arrays werden im HashGrid nur die Zellen erzeugt und verwaltet, die wirklich Datenpunkte enthalten. Dies könnte zwar auch durch andere Datenstrukturen, z.B. Octrees, erreicht werden, allerdings ist dort die Abfrage von Nachbarbeziehungen aufwändiger. Da diese aber bei der späteren Erstellung eines Polygonnetzes benötigt werden, wäre eine Implementierung im Octree schwierig. Ein weiterer Vorteil des HashGrids ist die Tatsache, dass alle vorhandenen Zellen bei Bedarf direkt durch einen Iterator traversiert werden können. Dies ist durch rekursive Aufrufe zwar auch im Octree möglich, die Implementierung der Algorithmen ist mit Iteratoren jedoch leichter verständlich.

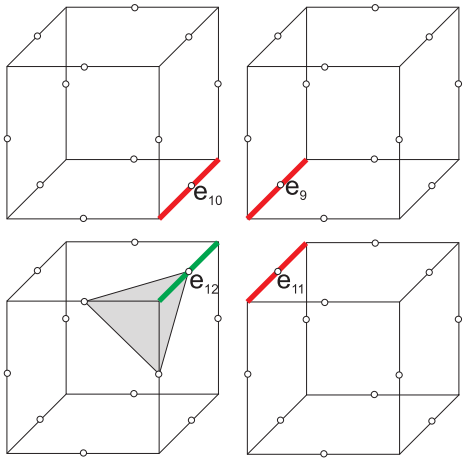


Abbildung 3.16: Suchen nach bereits im Modell vorhandenen Vertices. Bei der Generierung der grauen Fläche wird der Kantenvertex 12 benötigt. Um zu überprüfen, ob er bereits im Vertexbuffer vorhanden ist, müssen die Indexarray-Einträge 11, 9 und 10 in den benachbarten Zellen überprüft werden (die Nummerierung entspricht der von Lorensen und Cline [25]).

3.3.2 Erzeugung von Index- und Vertex-Buffer

Ziel ist es, einen Index- und Vertex-Buffer aufzubauen, so dass jeder Vertex mit der dazugehörenden Normalen nur einmal abgespeichert wird. Dazu ist es nötig zu wissen, ob ein Vertex bereits im Netz vorhanden ist. Um diese Abfrage effizient durchführen zu können, wird die Verzeigerung der Zellen im HashGrid ausgenutzt.

Jede Zelle erhält dazu ein 12-stelliges Indexarray, das die Indizes der Kantenmittelpunkte im Vertex-Buffer enthält. Die Felder im Array werden mit dem Wert -1 initialisiert, um anzuzeigen, dass die Vertices noch nicht genutzt wurden. Sobald ein Vertex von einem Marching-Cubes-Approximationsmuster verwendet wird, wird die Position der Vertexdaten im Vertex-Buffer an der entsprechenden Stelle im Indexarray der Zelle abgelegt.

Da alle Vertices auf den Zellkanten liegen, müssen bei der Anwendung des Marching-Cubes-Verfahrens nur die passenden Kanten in den maximal drei Zellen, die sich diese Kante teilen, überprüft werden, um herauszufinden, ob der Punkt bereits im Vertexarray vorhanden ist. Die Beziehungen, welche Nachbarn und welche Indexeinträge überprüft werden müssen, werden wiederum in zwei Tabellen kodiert (s. Abbildung 3.16).

Wird in den drei Nachbarn kein Verweis auf eine Position im Vertex-Buffer gefunden, wird der vom Approximationsmuster verwendete Punkt sowie die dazugehörende Normale neu in den Puffer eingetragen und der neue Index in der aktuellen Zelle und den drei Nachbarn abgelegt. Sollte einer der drei Nachbarn bereits einen passenden Indexeintrag haben, wird die Normale im Vertex-Buffer mit der neu berechneten Normalen gemittelt. Damit eine konsistente Belegung der Zell-Indizes gewährleistet ist, werden anschließend alle beteiligten Zellen entsprechend aktualisiert. Auf diese Art und Weise ist sichergestellt, dass jeder Kantenpunkt nur einmal im Vertex-Buffer abgelegt wird. Das Verfahren ist schematisch in Algorithmus 3.3 dargestellt.

Vorteil dieser Methode ist, dass vor dem Einfügen eines Vertex in den Vertex-Buffer nicht überprüft werden muss, ob dieser sich bereits darin befindet. Dies würde zu einem Algorithmus der Komplexität $\mathcal{O}(n^2)$ führen. Bei dieser Vorgehensweise müssen maximal $3n$-Zellen überprüft werden, die durch die Verzeigerung und die Verwendung einer Tabelle in konstanter Zeit abgefragt werden können. Die Laufzeit dieses Verfahrens betragt also $\mathcal{O}(n)$. Es entsteht lediglich ein kleiner Speicheroverhead, da jede Zelle ein 12-stelliges Integer für die Positionen im Buffer enthalten muss. Im Vergleich zu den ansonsten auftretenden Redundanzen im Vertex-Buffer (mit je mindestens drei Floats bzw. Doubles) ist der zusätzlich erforderliche Speicher aber relativ gering.

Der komplizierteste Teil bei der Implementierung dieses Verfahrens ist das Aufstellen der Tabellen mit den Nachbarschaftsbeziehungen. Sie müssen aber nur ein einziges Mal berechnet werden, so dass sich der Aufwand in Grenzen hält.

3.3.3 Erzeugung einer Halbkantendarstellung

Zur Erzeugung einer Halbkantendarstellung werden Datenstrukturen zur Repräsentation von Vertices, Kanten und Faces benötigt. Diese sind wie folgt definiert:

HalfEdgeVertex. Ein HalfEdgeVertex repräsentiert einen Knoten im Polygonnetz. Er enthält neben den benötigten Vertexinformationen (Position, Normale, Farbe etc.) eine Liste mit den Kanten, die auf ihn zeigen bzw. aus ihm herauslaufen.

HalfEdge. Repräsentation einer Kante. Sie besteht aus zwei Vertices und Zeigern auf die nächste Kante bei der Traversion eines Faces gegen den Uhrzeigersinn. Des Weiteren werden Zeiger auf das Face und die Partnerkante gespeichert.

HalfEdgeFace. Repräsentation eines Faces. Es enthält lediglich einen Zeiger auf eine begrenzende Kante.

Der Aufbau des Polygonnetzes geschieht folgendermaßen: Als erstes werden die drei zu einem Face gehörenden Vertices und die entsprechenden Indizes wie zuvor beschrieben mit der Marching-Cubes-Methode erzeugt. Anschließend wird es in die HalfEdge-Datenstruktur eingebunden.

Um das zu erreichen werden in einem ersten Schritt die drei Kanten generiert, die dieses Face umrunden. Für jeden Vertex wird dabei überprüft, ob es bereits eine Kante gibt, die auf ihn zeigt, und deren Partnerkante kein Face zugeordnet ist. Dazu wird die Liste der eingehenden Kanten untersucht. Ist eine solche Kante vorhanden, wird sie in die Liste der umgebenden Kanten aufgenommen, und der Face-Zeiger dieser Kante wird auf das aktuelle Dreieck gesetzt. Ansonsten wird ein neues Halbkantenpaar erzeugt und die Ein- uns Ausgangslisten der beiden beteiligten werden Vertices aktualisiert. Sind alle Halbkanten eines Faces gefunden, werden sie verkettet. Zum Schluss erhält das aktuelle Face einen Zeiger auf die erste Kante der Liste. Der komplette Prozess ist in Abbildung 3.17 und Algorithmus 3.2 dargestellt.

Auf diese Art und Weise lässt sich effizient ein korrekt verzeigertes Netz aufbauen. Fast alle dazu benötigten Abfragen können in konstanter Zeit erfolgen. Lediglich das Durchsuchen der Listen mit den eingehenden Kanten erzeugt eine lineare Laufzeit in der Anzahl der auf ihn zeigenden

Kanten. Da in der Regel aber nur wenige Kanten auf einen Vertex zeigen, verlangsamt die Suche den Prozess nicht wesentlich.

```
procedure AddFaceToMesh(vertexIndices, mesh)
    edges ← empty list of half edges
    faces ← empty half edge face
    for i = 0 to 3 do
        currentVertex ← vertexIndices[i]
        nextVertex ← vertexIndices[(i + 1) mod 3]
        if edge to current vertex exists then
            edges[i] ← pair-edge of edge to vertex
            edges[i].face ← currentFace
        else
            create new edge with correnspondingpair and link them
            update in and out lists of edge vertices
            edges[i] ← newly created half edge
        end if
    end for
    for i = 0 to 3 do
        edges[i].next ← edges[(i + 1) mod 3]
    end for
    face ← edges[0]
    add face to mesh
end procedure
```

Algorithmus 3.2: Erzeugung einer HalfEdge Darstellung. Der Methode werden die Indizes der drei Vertices des vom Marching-Cubes-Algorithmus erzeugten Dreiecks und ein Zeiger auf eine Liste mit den Faces im Dreiecksnetz übergeben.

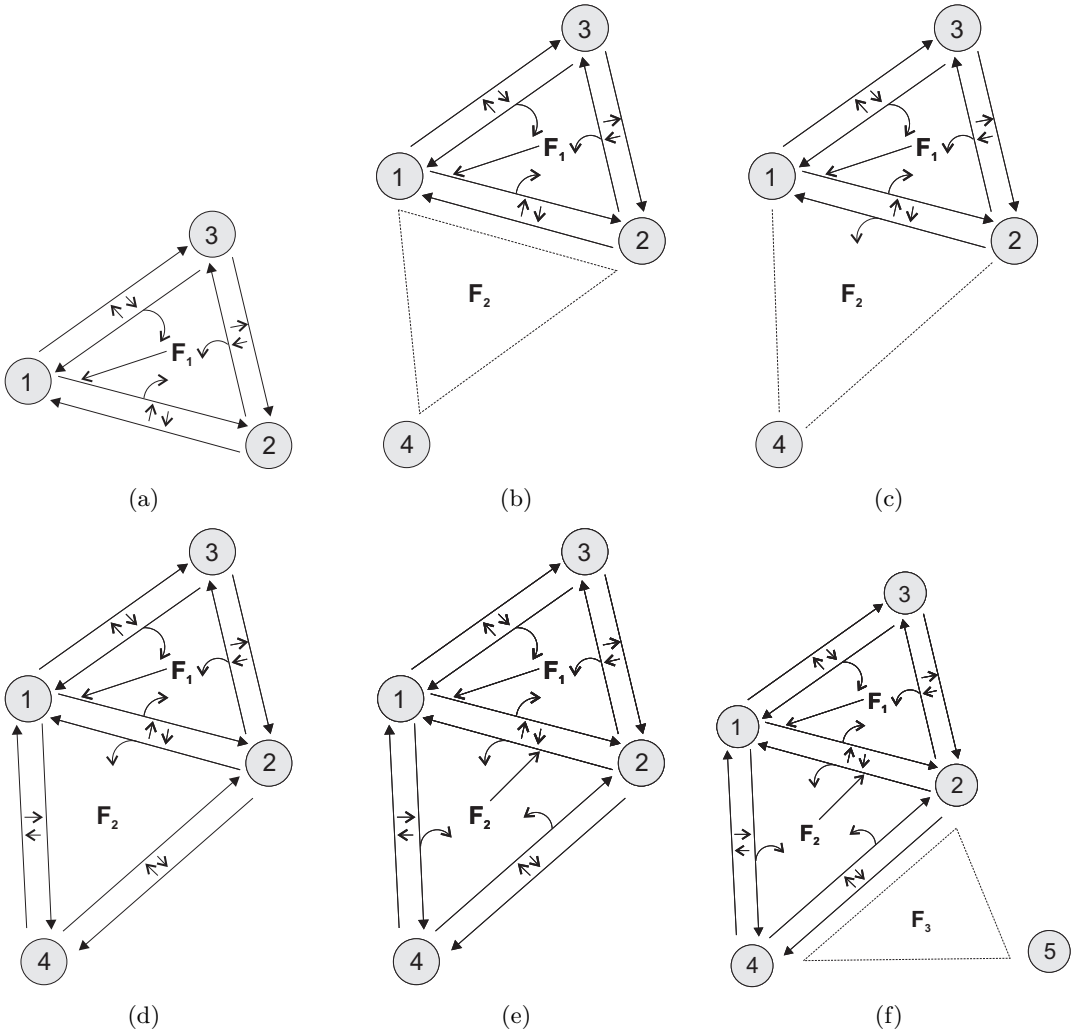


Abbildung 3.17: Erzeugung eines Halbkantennetzes. Zunächst werden das erste Face erzeugt und alle Objekte korrekt verzeigert (a). Ein neu erzeugtes Face ist noch nicht eingebunden (b). Im nächsten Schritt werden alle Halbkanten gesucht, die zuvor Randkanten ware und auf einen der der Vertices des neu entstandenen Faces zeigen; die Zeiger dieser Kanten werden auf das neue Dreieck gesetzt. Es ist nun in das Netz eingebunden (c). Werden keine weiteren Kanten gefunden, die das Face bereits vorher begrenzt haben, werden die fehlenden Kantenpaare neu generiert (d) und anschließend verzeigert (e). Sobald ein weiteres Face entsteht, startet der Prozess von vorn.

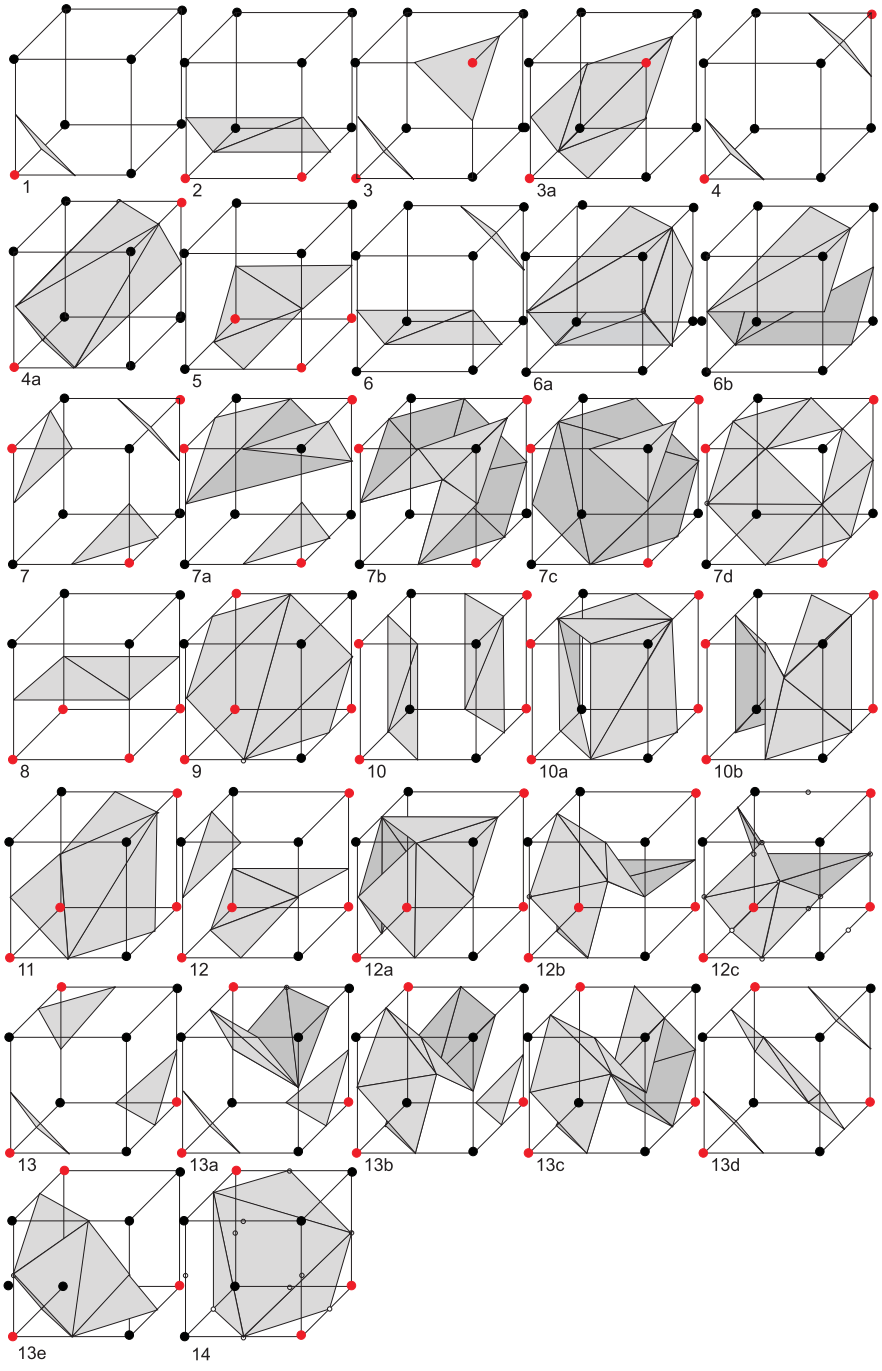


Abbildung 3.18: Alle Grundmuster, die zur Erzeugung geschlossener Polygonnetze erforderlich sind

```
procedure GenerateBuffers
    indexBuffer ← empty
    vertexBuffer ← empty
    gloabalIndex ← 0
    for all Cells do
        index ← index of corner configuration
        i ← 0
        vertexIndex ← -1
        while Marching-Cubes-Table[index][i] != −1 do
            for j = 0 to 3 do
                edgeNumber ← [Marching-Cubes-Table[index][i+j]]
                vertexIndex ← indices[EdgeNumber]
                if vertexIndex = -1 then
                    search for index in the cells that share edge[edgeNumber]
                    if no index found then
                        vertexBuffer ← intersection on edge[edgeNumber]
                        vertexIndex ← globalIndex
                        globalIndex ← globalIndex + 1
                        update indices in cells with shared edge
                    else
                        indices[EdgeNumber] ← found index
                        vertexIndex ← found index
                    end if
                end if
                indexBuffer ← vertexIndex
            end for
            calculate Normal for generated Face
            i ← i + 3
        end while
    end for
end procedure
```

Algorithmus 3.3: Erweiterter Marching-Cubes-Algorithmus zur Erzeugung von Index- und Vertexbuffer. Dabei wird den Schnittpunkten auf den Würfelzellen ein Eintrag in einem lokalen Index-Array zugeordnet, der die Position des Vertex' im VertexBuffer angibt.

Kapitel 4

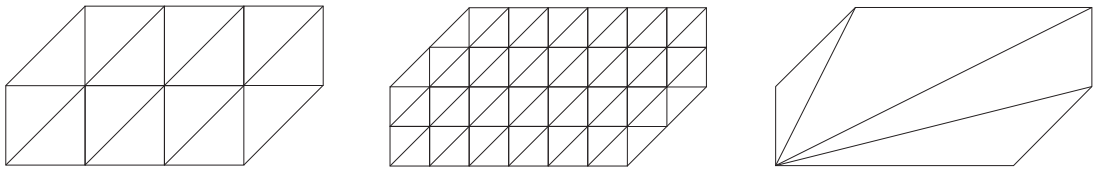
Modelloptimierung

4.1 Motivation

Bei der Generierung eines Polygonmodells aus Laserscannerdaten erzeugt der Marching-Cubes-Algorithmus lokal bis zu vier Dreiecke pro Zelle. Daher werden auch planare Flächen, die normalerweise durch wenige Faces dargestellt werden könnten, unnötig oft unterteilt. Da die Anzahl der erzeugten Faces direkt von der Gittergröße abhängt, werden mit kleinerer Zellengröße, also besserer Approximation der Ausgangsdaten, immer mehr Dreiecke generiert (s. Abbildung 4.1).

Eine solche Repräsentation ist sehr speicherintensiv und kann nur ineffizient gerendert werden. Es werden daher Verfahren benötigt, die die redundanten Dreiecke entfernen, ohne die Geometrie des Modells zu beeinflussen. Viele Reduktionsalgorithmen berechnen den Fehler, den das Verändern eines Vertices oder einer Kante am vorhandenen Modell verursachen würde. Das Modell wird bei der Optimierung iterativ modifiziert bis, je nach Verfahren, ein vorgegebener Maximalfehler erreicht wird [27] oder die Iteration konvergiert [19]. Solche Methoden haben allerdings den Nachteil, dass nach jeder Änderung des Meshes neue Fehlerwerte für das veränderte Modell berechnet werden müssen. Die Verfahren sind also sehr rechenintensiv.

In diesem Kapitel soll ein effizienter Algorithmus vorgestellt werden, mit dessen Hilfe die Anzahl der Dreiecksflächen in Modellen mit vielen ebenen Gebieten drastisch verringert werden



(a) (b) (c)

Abbildung 4.1: Typische Approximationsmuster des Marching-Cubes-Algorithmus und eine optimale Triangulation. Das in (a) dargestellte Polygon wird in 14 Dreiecke unterteilt, das unter (b) gezeigte bei halbierter Zellgröße sogar in 56. Eine Aufteilung in vier 4 Grundflächen wäre für die gleiche Darstellung aber ausreichend (c).

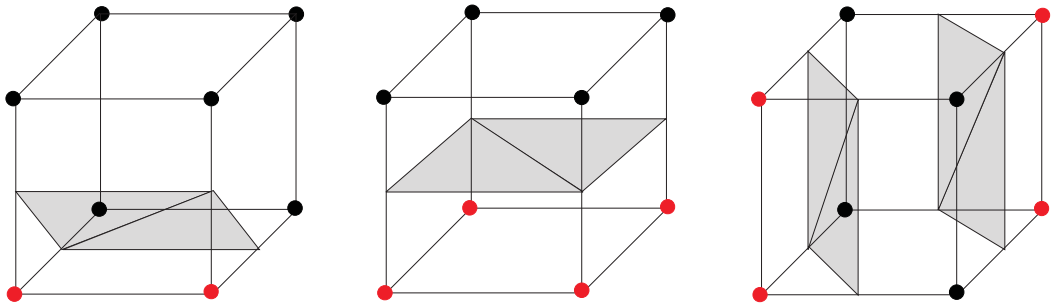


Abbildung 4.2: Drei Approximationsmuster, die zusammengesetzt ebene Flächen ergeben.

kann, ohne dass Fehlerterme berechnet werden müssen. Die Geometrie des urprünglichen Polygonnetzes wird dabei nicht verändert. Zur Erkennung redundanter Faces werden lediglich die vom Marching-Cubes-Algorithmus bestimmten Flächenindizes analysiert, um zusammenhängende Gebiete zu erkennen. Dabei wird die zuvor generierte Halbkantendarstellung ausgenutzt, um Nachbarschaftsbeziehungen der Faces innerhalb des Polygonnetzes abzufragen.

Das Verfahren besteht im Wesentlichen aus drei Schritten: Zunächst werden zusammenhängende planare Flächen ermittelt. Anschließend werden die begrenzenden Halbkanten dieser Flächen zu Polygonen zusammengefasst. In einem dritten Schritt werden die Polygone neu trianguliert, so dass wieder eine Dreiecksrepräsentation vorliegt.

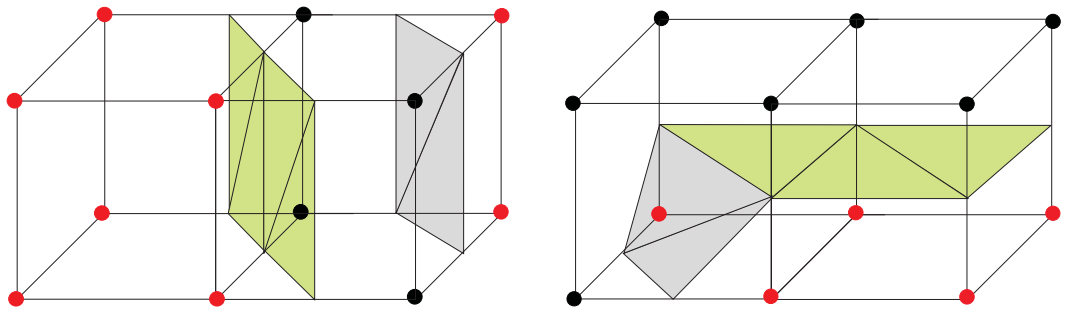
4.2 Detektion zusammenhängender Flächen

Zusammenhängende Flächen können erkannt werden, indem die Marching-Cubes-Indizes analysiert werden. Einige Grundmuster ergeben, wenn sie nebeneinander wiederholt vorkommen, ebene Flächen. Die drei Grundmuster, die diese Eigenschaft immer erfüllen, zeigt Abbildung 4.2. Auch Kombinationen dieser drei Muster können größere planare Flächen bilden.

So kann z.B. eine rotierte Form von Muster (a) aus Abbildung 4.2 zusammen mit Muster (c) eine zusammenhängende Fläche ergeben. Bei den anderen Mustern können durch geeignete Kombinationen von Teilflächen auch planare Regionen entstehen. Diese setzen sich allerdings in der Regel nur aus wenigen Faces zusammen und können daher vernachlässigt werden (Abbildung 4.3).

Um für eine Vereinfachung relevante Fälle erkennen zu können, werden bei der Erzeugung des HalfEdge-Netzes die Indizes der Approximationmuster, aus denen die Faces erzeugt wurden, gespeichert. Nach der Generierung des Oberflächenmodells werden sequenziell alle Dreiecke überprüft, ob sie zu einem für die Vereinfachung relevanten Approximationsmuster gehören.

Sobald eine solche Fläche ermittelt wurde, werden die Indizes aller Nachbarflächen getestet. Wird ein angrenzendes Face gefunden, das denselben oder einen passenden anderen Index hat, so dass eine ebene Fläche entsteht, wird die Suche von dieser Fläche aus rekursiv neu gestartet. Um Mehrfachprüfungen zu vermeiden, werden bereits besuchte Faces als überprüft markiert. Besitzt ein Nachbarface einen ungültigen Index, bildet die Kante zu diesem Dreieck eine Begrenzung



(a) (b)

Abbildung 4.3: Kombinationen von Approximationsmustern, die zusammenhängende ebene Flächen ergeben. In (a) ist eine Kombination der Muster (a) und (c) aus Abbildung 4.2 zu sehen. Die vier grünen Dreiecke könnten durch zwei neue ersetzt werden. Teil (b) zeigt eine Kombination aus einem anderen Muster und (b) aus Abbildung 4.2. Hier wird durch die Kombination nur ein einzelnes neues Dreieck zur grünen Fläche hinzugefügt, das zudem durch nicht zur Ebene gehörende Flächen begrenzt wird. Die Einsparmöglichkeiten sind in solchen Kombinationen nur gering. Daher werden sie bei der Zusammenfassung nicht behandelt.

der aktuellen Ebene. Sie wird in eine Menge von Begrenzungskanten aufgenommen und die Rekursion wird abgebrochen.

Auf diese Art und Weise werden alle Begrenzungskanten bestimmt, die zu der Ebene gehören, in der auch das erste gefundene Dreieck lag. Im zweiten Schritt des Vereinfachungsalgorithmus sollen diese Kanten zu Polygonen zusammengefasst werden. Um später effizient auf sie zugreifen zu können, werden die gefundenen Kanten in einer Hashmap gespeichert. Die Speicheradressen (Zeiger) der Startvertices liefern die Hashwerte. Da diese Speicheradressen eindeutig sind, sind Kollisionen beim Hashing ausgeschlossen.

Nachdem alle Begrenzungskanten einer Ebene gefunden wurden, wird ausgehend vom letzten Startdreieck ein neues noch nicht geprüftes Startface gesucht. Sobald keine unberücksichtigten Faces mehr gefunden wurden, terminiert der Algorithmus.

Dieses Vorgehen hat den Vorteil, dass jedes Face nur einmal getestet werden muss, um zu bestimmen, ob es zusammen mit einem seiner Nachbarn eine ebene Fläche bildet. Zudem werden alle relevanten ebenen Flächen gefunden und zusammengefasst, da sequenziell alle Faces überprüft werden, ob sie als Startfläche in Frage kommen. Die Laufzeit des Algorithmus beträgt daher $\mathcal{O}(n)$ in der Anzahl der generierten Faces. Eine Terminierung ist auch in jedem Fall gewährleistet, da alle planaren Gebiete durch unpassende Nachbarfaces definierte Grenzen haben.

4.3 Zusammenfassen der Begrenzungskanten zu Polygonen

Im ersten Schritt der Vereinfachung wurden die Begrenzungskanten von ebenen Regionen gefunden. Diese liegen allerdings nicht geordnet vor. Ziel des zweiten Vereinfachungsschrittes ist es, aus diesen Daten Polygondarstellungen der Begrenzungen zu gewinnen.

```
function GenerateLists
    for all Faces do
        currentFace ← used
        currentIndex ← index of current Face
        if currentIndex is relevant then
            checkNeighbours(currentIndex, currentFace, currentList)
        else
            borderLists ← currentList
            currentList ← empty
        end if
    end for
end function


function CheckNeighbours(checkIndex, currentFace, listOfBorders)
    currentFace ← used
    for all Neighbours of current Face do
        if index of currentNeighbour = checkIndex ∧ currentNeighbour not used then
            CheckNeighbours(checkIndex, currentNeighbour, listOfBorders)
        else
            listOfBorders ← border to currentNeighbour
        end if
    end for
end function
```

Algorithmus 4.1: Zusammenfassen planarer Regionen in der HalfEdge-Datenstruktur

Dazu wird ein Konturverfolgungsalgorithmus angewendet: Ausgehend von einer Startkante wird diejenige Kante gesucht, deren Startpunkt mit dem Endpunkt der Ausgangskante übereinstimmt. Diese Suche wird so lange fortgesetzt, bis der Endpunkt der letzten gefundenen Kante wieder mit dem Startpunkt der ersten Kante übereinstimmt und die Kontur geschlossen ist (s. Abbildung 4.4).

Auf diese Art und Weise werden gerade Strecken allerdings unnötig oft unterteilt, da die maximale Länge der Halbkanten durch die Größe der Marching-Cubes-Boxen vorgegeben ist. Es muss also ein Verfahren angewendet werden, das die auf einer Geraden liegenden zusammengehörenden Kanten erkennt und zusammenfasst.

Eine einfache Möglichkeit, im Zweidimensionalen solche Konfigurationen zu erkennen, stellen die sog. Freeman-Kettencodes dar [12]. Ursprüglich wurden diese Codes benutzt, um Konturen in Pixelbildern zu kodieren. Der Freeman-Code gibt in acht Richtungen quantisiert an, wo vom aktuellen Pixel aus gesehen sich das nächste Pixel einer Kontur befindet. Pixel, die auf einer Geraden liegen, erhalten dabei den selben Kettencode (s. Abbildung 4.5).

Eine auf drei Dimensionen erweiterte Variante dieser Idee lässt sich auf die Kanten der Begrenzungspolygone anwenden. Da die Vertices der Halbkanten immer auf Kantenmittelpunkten der

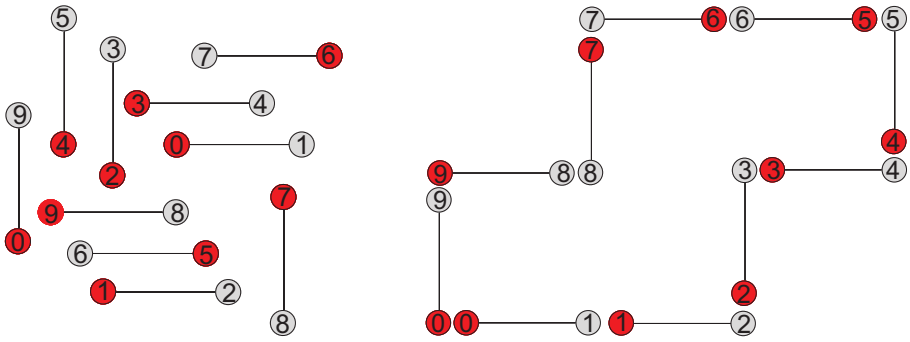


Abbildung 4.4: Ordnen der unsortierten Begrenzungskanten. Links ist die Menge der unsortierten Begrenzungskanten mit Start-Vertices (rot) und End-Vertices (grau) zu sehen. Rechts erkennt man die Kontur des Begrenzungspolygons, nachdem die Kanten geordnet wurden. Die Nummern entsprechen Zeigern, die auf die dazugehörigen Vertices verweisen.

Marching-Cubes-Boxen liegen, kommen auch hier nur diskrete Richtungen vor. Im Dreidimensionalen kann es insgesamt 26 verschiedene Richtungen geben, denen Codes zugeordnet werden müssen. Dies geschieht wie folgt: Der Start-Vertex einer Kante wird als Referenzpunkt festgelegt und der 2D-Freeman-Code gemäß Abbildung 4.5 unter Vernachlässigung der z-Komponente der Vertices berechnet. Damit der Wert 0 nicht vorkommen kann, werden die ursprünglichen Fälle allerdings nicht von 0 bis 7, sondern von 1 bis 8 durchnummeriert. Anschließend wird die Differenz der z-Komponenten betrachtet. Beträgt sie 0, werden die 2D-Codes beibehalten. Bei einer positiven Differenz wird der Code mit 10 multipliziert, bei einer negativen mit -10. Auf diese Art und Weise lassen sich die Richtungen der Kanten eindeutig identifizieren. Haben aufeinanderfolgende Kanten denselben Kettencode, werden sie zusammengefasst (Beispiel s. Abbildung 4.6). In der Polygondarstellung werden abschließend nur noch die Zeiger auf die Startvertices abgespeichert.

Mit Hilfe dieses Verfahrens wird aus der unsortierten Kantenmenge eine optimale Polygondarstellung der Kontur einer ebenen Fläche erzeugt. Allerdings müssen die betrachteten Flächen nicht zwangsläufig nur eine einzige Kontur besitzten. Innerhalb der äußeren Kontur können weitere Umrisse vorhanden sein, die nicht zur Fläche gehörende Bereiche abgrenzen (s. Abbildung

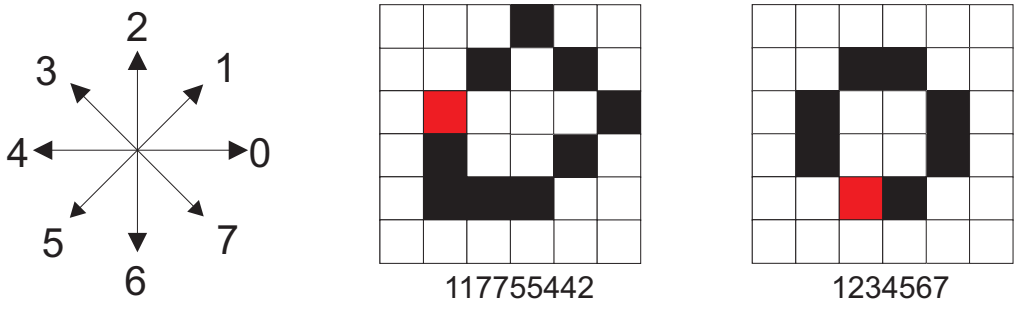


Abbildung 4.5: Freeman-Codes mit Beispielen. Die roten Pixel markieren jeweils den Startpunkt der Kodierung.

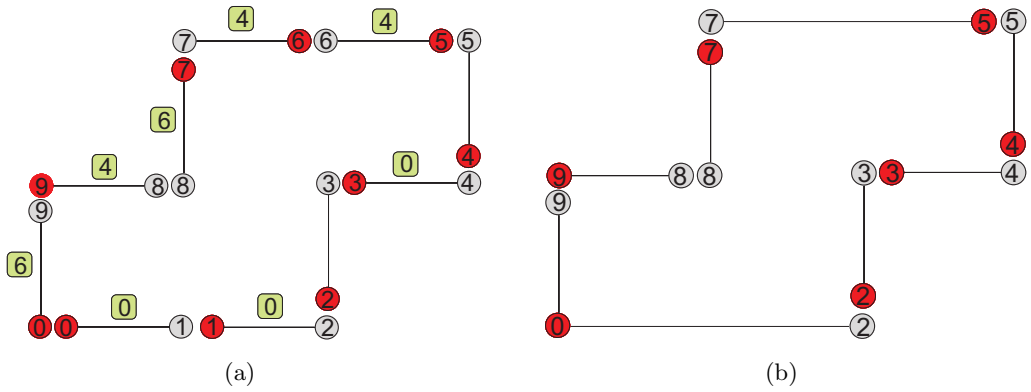


Abbildung 4.6: Zusammenfassen der Kanten mit Hilfe des Kettencodes. Die Kettencodes sind in (a) grün dargestellt. Durch vergleich der Codes konnten die Vertices 1 und 6 eingespart werden (b).

4.7). Diese können z.B. durch Fenster, Säulen, Türen oder andere Objekte in den Scans hervorgerufen werden. Da für eine korrekte Triangulation alle Konturen zu berücksichtigen sind, muss sichergestellt werden, dass alle zuvor gefundenen Begrenzungskanten überprüft worden sind. Das geschieht, indem die Anzahl der zusammengefassten Kanten mitgezählt wird. Sind noch nicht alle verwendet worden, wird eine neue Kontur mit einer bislang unbenutzten Halbkante begonnen. Das Ergebnis einer Reduktion nach Anwendung der ersten beiden Schritte ist in Abbildung 4.8 dargestellt.

4.4 Triangulation der Konturpolygone

Da die meisten Grafik-APIs wie OpenGL oder DirectX nur Dreiecke oder konvexe Polygone rendern können, müssen die durch die gefundenen Konturen begrenzten Flächen in entsprechende Primitive zerlegt werden. Dieser Prozess wird in der Computergrafik als Tesselation bezeichnet. Das kleinste einfache und konvexe Polygon ist ein Dreieck. Daher sind Dreiecke zum Approximieren komplexer Strukturen am besten geeignet. Der Prozess der Zerlegung von Polygonen in Dreiecke wird als Triangulation bezeichnet.

Im Folgenden werden einige Triangulationsalgorithmen kurz vorgestellt. Obwohl sie für Polygone mit zweidimensionalen Vertices beschrieben sind, lassen sich die Verfahren auch auf planare

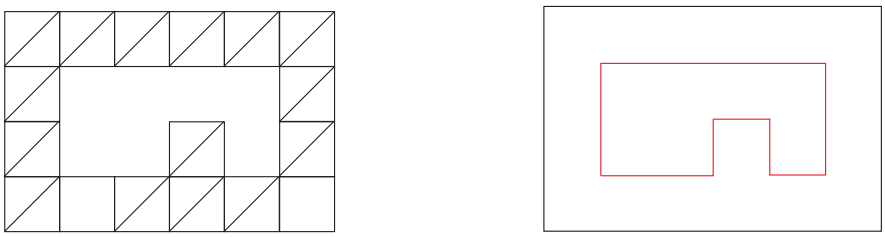


Abbildung 4.7: Eine Flächenkonfiguration, die mehrere Konturen erzeugt.

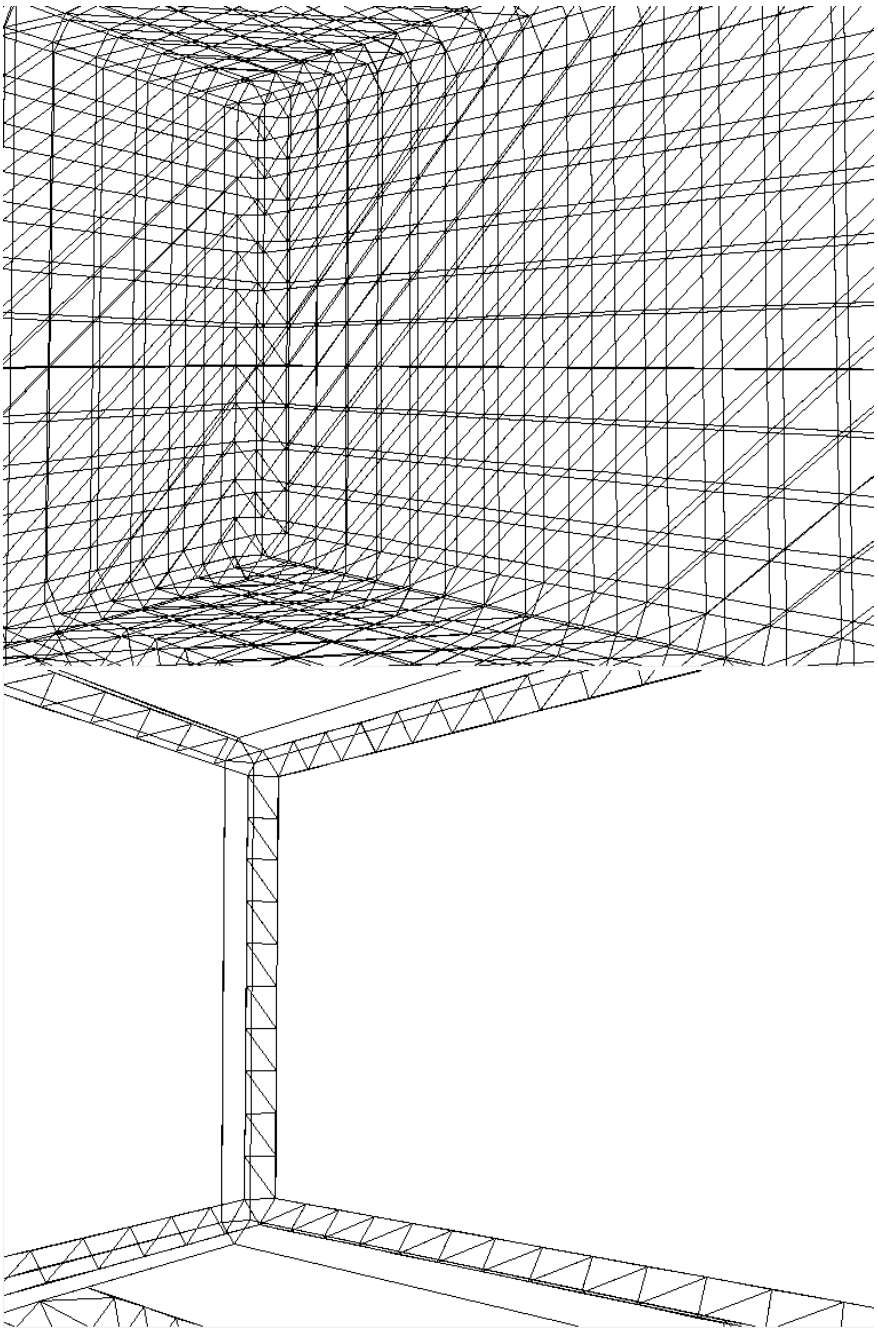
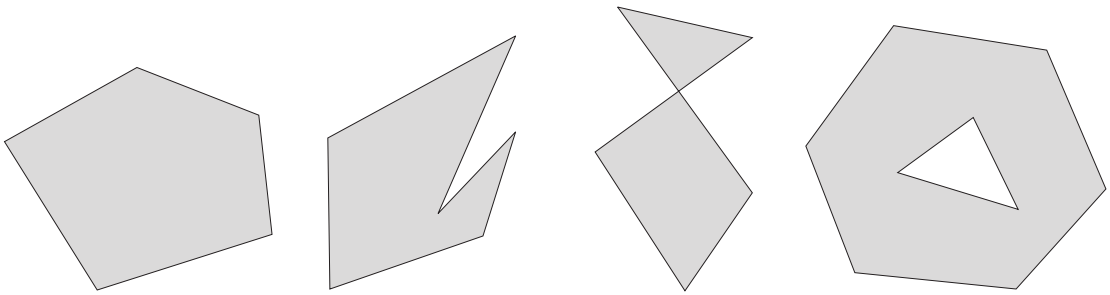


Abbildung 4.8: Erzeugte Konturpolygone nach der Zusammenfassung. Das obere Bild zeigt das Ausgangsmesh ohne Reduktion. Man kann deutlich erkennen, dass ebene Gebiete nicht mehr unterteilt werden und von Konturpolygonen begrenzt sind.



(a) Konvex (b) Konkav (c) Komplex (d) Polygon mit Loch

Abbildung 4.9: Verschiedene Klassen von Polygonen

Polygone mit dreidimensionalen Punktkoordinaten anwenden, indem man diese vor der Triangulation in die $x - y$-Ebene projiziert.

4.4.1 Eigenschaften von Polygonen

Polygonzüge und Vertices in Polygonen können unterschiedliche Eigenschaften haben, anhand derer sie klassifiziert werden. Im Folgenden sollen die für die Beschreibung von Triangulationsalgorithmen relevanten Begriffe erläutert werden.

Komplexe Polygone

Komplexe Polygone sind Polygone, die eine oder mehrere sich schneidende Kanten oder Löcher haben. Ein Loch ist ein Polygon, das vollständig in einem anderen liegt. Löcher werden manchmal auch als Inseln bezeichnet.

Einfache Polygone

Einfache Polygone sind Polygone, die keine Löcher oder sich schneidende Kanten haben. Ein einfaches Polygon wird als konvex bezeichnet, wenn es nur konvexe Vertices besitzt. Ein Vertex v_i eines Polygons wird als konvex bezeichnet, wenn bei einer Traversierung des Polygons entgegen dem Urzeigersinn beim Übergang von v_{i-1} nach v_{i+1} an der Stelle v_i ein Knick nach links entsteht. Entsteht ein Knick in die andere Richtung, wird der Vertex als konkav bezeichnet. Einfache Polygone, die mindestens einen konkaven Vertex haben, nennt man konkav, die anderen konvex.

Konvexe und konkave Polygone können auch anhand der zur innenfläche liegenden Winkel klassifiziert werden. Bei konvexen Polygonen sind alle Innenwinkel kleiner als 180^o.

Diagonalen

Eine Diagonale im Polygon ist eine Verbindungslinie zweier Vertices v_i und v_j, die vollständig im Polygon liegt und keine Kanten schneidet.

Monotone Polygone

Ein monotones Polygon ist ein Polygon, dessen Kontur aus zwei monotonen Ketten besteht. Eine Folge von Vertices heisst x- bzw. y-monoton, wenn die entsprechenden Koordinaten der

(a) (b)

Abbildung 4.10: Einbinden von Löchern in Polygonzüge. In Bild (a) ist ein einfaches Polygon mit Löchern gezeigt. Durch Verbinden der Vertices, die den kürzesten Abstand zur umschließenden Kontur haben, werden die Konturen der Löcher in den Polygonzug mit eingebunden und ein einfaches Polygon entsteht (b).

Punkte in der Kette aufsteigend sind.

4.4.2 Triangulierung von konvexen Polygonen

Die Triangulation von konvexen Polygonen ohne Löcher ist trivial: Man wählt einen beliebigen Vertex aus und zieht Diagonalen zu allen anderen Punkten. Die Komplexität dieses Verfahrens beträgt $\mathcal{O}(n)$.

Die Frage, ob ein Polygon konvex ist, kann mit linearem Zeitaufwand entschieden werden. Dazu wird für jeden Vertex v_i überprüft, ob er konvex ist, d.h. ob er sich links von der durch v_{i-1} und v_{i-2} definierten Geraden befindet.

4.4.3 Triangulierung von komplexen Polygonen

Bei komplexen und konkaven einfachen Polygonen werden aufwändigere Algorithmen benötigt. Befinden sich Löcher in den Polygonen, müssen die Konturen zunächst verbunden werden, indem die kürzeste Verbindung zwischen der äußeren Kontur und dem Loch als neue Kante eingefügt wird [29]. Diese Strecke wird doppelt eingefügt, so dass sie beim Traversieren des Polygons einmal als „Hinweg" und einmal als „Rückweg" vorhanden ist. Auf diese Art und Weise entsteht wieder eine einzelne geschlossene Kontur (Abbildung 4.10).

Ear Cutting

Einfache Polygone lassen sich immer triangulieren, wie H. G. Meisters 1975 gezeigt hat [15]. Er bewies, dass jedes Polygon mit mehr als drei Vertices mindestens zwei sich nicht überlappende Ohren besitzt (*Two Ears Theorem*). Ein Vertex wird als Ohr eines Polygons bezeichnet, falls die Verbindungsstrecke der Nachbarn v_{i+1} und v_{i-1} eines Polygonvertexes v_i komplett im Polygon liegt. Das aus diesen drei Vertices gebildetete Dreieck wird ebenfalls als Ohr bezeichnet.

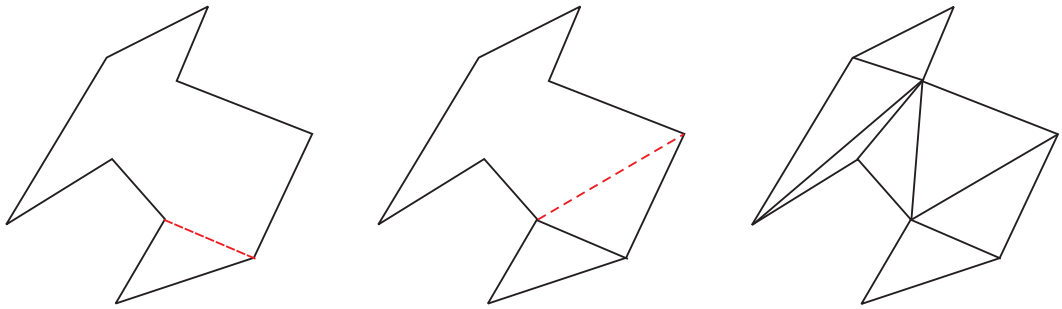


Abbildung 4.11: Ear-Cutting. Das Polygon wird durch wiederholtes Abtrennen von Ohren trianguliert.

Jedes einfache Polygon lässt sich triangulieren, indem sukzessive alle Ohren abgetrennt werden. Dabei entstehen immer mindestens $n-2$ Dreiecke. Dieses Verfahren wird *Ear Cutting* genannt. Das Hauptproblem bei dieser Vorgehensweise ist das Auffinden der Ohren.

Algorithmus 4.2 zeigt einen einfachen Algorithmus zum Auffinden eines Ohrs [51]. Die Laufzeit des Algorithmus beträgt $\mathcal{O}(kn)$, wobei k die Anzahl der konkaven Vertices im Polygon ist. Im Worst-Case beträgt die Laufzeit $\mathcal{O}(n^2)$, da insgesamt $\mathcal{O}(n)$ konkave Vertices auftreten können.

```
function FindEar(Polygon)
    ear not found ← true
    i ← 0
    while ear not found do
        if p_i is convex then
            if Triangle formed by v_{i-1}, v_i and v_{i+1} contains no concave vertex then
                ear not found ← false
            end if
            if ear not found then
                i ← i + 1
            end if
        end if
    end while
    return v_i
end function
```

Algorithmus 4.2: Finden eines Ohrs in $\mathcal{O}(n^2)$

Einen Algorithmus zum Auffinden eines Ohrs in linearer Zeit zeigt Algorithmus 4.3 [17]. Dieser zerteilt ein Polygon rekursiv in „Gute Unterpolygone" (*Good Sub Polygons, GSP*), bis ein Ohr gefunden wurde. Ein GSP ist ein Unterpolygon, das mit dem Rest des ursprünglichen Polygons nur eine gemeinsame Kante hat (die sog. *Cutting Edge*). Da jedes Unterpolygon höchstens $n-2$ Vertices haben kann, ergibt sich eine lineare Laufzeit.

Ear-Cutting Algorithmen haben den Vorteil, dass sie leicht verständlich und einfach zu implementieren sind. Allerdings haben sie hohe Worst-Case-Laufzeiten. Da jedes Polygon in $n-2$

Dreiecke unterteilt werden kann, muss der Auffinde-Algorithmus beim Ear-Cutting $(n-2)$-mal aufgerufen werden. Es entstehen demnach Worst-Case Laufzeiten von $\mathcal{O}(n^3)$ bei Verwendung von Algorithmus 4.2 bzw. $\mathcal{O}(n^2)$ mit Alg.4.3. Die Laufzeiten werden allerdings in beiden Fällen von der Anzahl der konkaven Vertices im Polygon bestimmt. Da in der Praxis mehr konvexe als konkave Vertices auftreten, liegen die tätsachlichen Laufzeiten in der Regel unter dem Worst-Case.

```
function FindEar(GSP, v_i)
    if v_i is an ear then
        return v_i
    end if
    Find a vertex p_j such that (p_i, p_j) is a diagonal of GSP.
    GSP' ← The good sub-polygon of GSP formed by (p_i, p_j).
    k ← Number of vertices in GSP'
    Re-label the vertices of GSP' so that p_i ← p_0 and p_j ← p_{k-1}
    (or p_j ← p_0 and p_i ← p_{k-1} as appropriate)
    FindEar(GSP', v_{k/2})
end function
```

Algorithmus 4.3: Finden eines Ohrs in $\mathcal{O}(n)$

Triangulation nach Seidel

Ein ebenfalls relativ einfach zu implementierender Triangulationsalgorithmus mit einer Worst-Case-Laufzeit von $\mathcal{O}(n \cdot log(n))$ wurde 1995 von Narkhede und Manocha [31] vorgestellt und basiert auf dem mit gleicher Laufzeit arbeitenden Algorithmus von Seidel [40]. Narkhedes und Manochas Algorithmus besteht aus drei Schritten: Zunächst werden die Polygone in Trapezoide zerlegt. Anschließend werden die Trapezoide in monotone Polygone zerlegt, die danach durch Entfernen konvexer Vertices trianguliert werden.

Um das Polygon in Trapezoide zu zerlegen, werden die Kanten des Polygons zufällig durchnummeriert. Ausgehend von den Endpunkten der Segmente wird eine horizontale Line nach links und rechts gezogen, bis sie auf ein weiteres Segment trifft. Die Strecken zwischen Endpunkt und Schnittpunkt, die innerhalb des Polygons liegen, bilden die horizontalen Begrenzungslinien der Trapezoide (s. Abbildung 4.12(a)). Diese Zerlegung geschieht in $\mathcal{O}(n \cdot log(n))$ Zeit. Die monotonen Polygone werden aus der Trapezzerlegung gewonnen, indem, wann immer möglich, Diagonalen zwischen zwei Vertices der ursprünglichen Kanten gezogen werden, die sich auf verschiedenen Seiten des Trapezoiden befinden (s. Abbildung 4.12b). Die dafür benötigte Zeit beträgt $\mathcal{O}(n)$.

Monotone Polygone können mit einem Scanline-Algorithmus in $\mathcal{O}(n)$ trianguliert werden. Dieser benutzt einen Stack, um die noch zu verarbeitenden Vertices zu verwalten. Die Funktionsweise soll hier für y-monotone Polygone gezeigt werden (nach [34,42]). Für x-monotone Polygone muss die Formulierung angepasst werden. Zunächst wird der Stack mit den oberen beiden Vertices initialisiert. Dann werden die Vertices in absteigender Reihenfolge durchlaufen. Liegen der oberste Vertex des Stacks und der aktuelle Vertex auf derselben Seite des Polygons, werden so oft wie

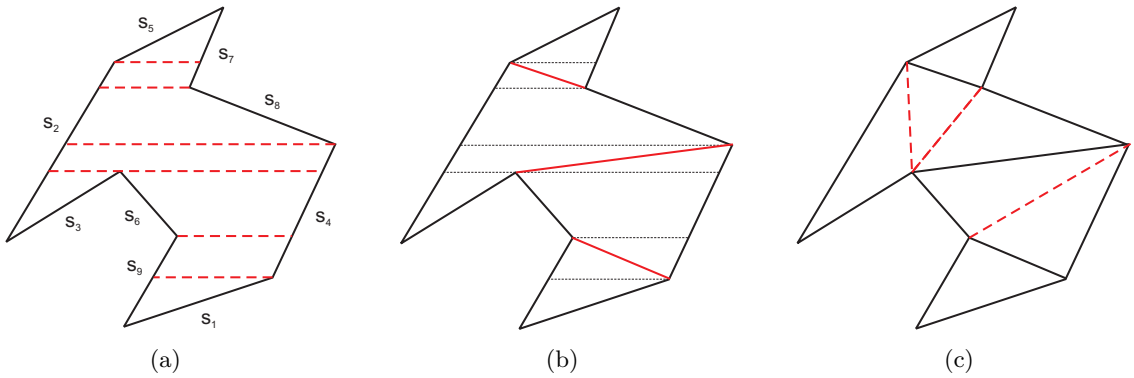


Abbildung 4.12: Triangulation nach Seidel. Zunächst werden die Kanten des Polygons zufällig durchnummeriert (s_1 bis s_9). Danach wird durch Ziehen von horizontalen Linien eine Zerlegung in Trapezoide hergestellt (a). Anschließend werden duch Einfügen von Diagonalen monotone Polygone erzeugt (b), die im nächsten Schritt trianguliert werden (c).

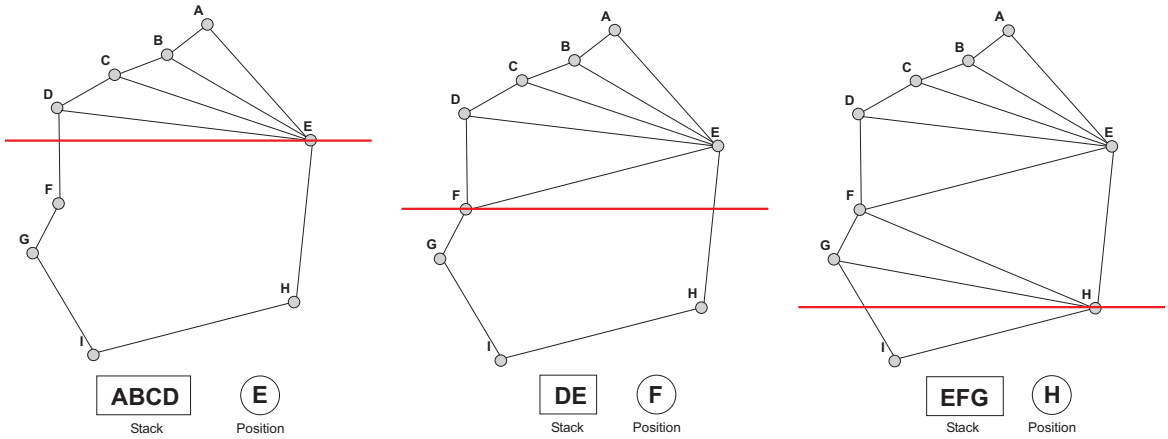


Abbildung 4.13: Triangulation monotoner Polygone. Hier sind die Situationen gezeigt, in denen der zuvor aufgebaute Stack abgearbeitet wird.

möglich Diagonalen eingefügt und die entsprechenden Vertices vom Stack entfernt. Anschließend wird der aktuelle Vertex abgelegt. Liegt der oberste Vertex auf der gegenüberliegenden Seite des aktuellen Vertexes, werden Diagonalen zu allen Vertices auf dem Stack gezogen und die Elemente gelöscht. Das oberste Element wird allerdings gespeichert und nach dem Ziehen der Diagonalen zusammen mit dem aktuellen Vertex neu abgelegt. Dieses Schritte werden fortgesetzt, dis der letzte Vertex erreicht ist (s. Algorithmus 4.4). Abbildung 4.13 verdeutlicht den Ablauf. Jeder Vertex, der nicht mehr verwendet wird, wird sofort aus dem Stack gelöscht. Daraus ergibt sich iene Laufzeit von $\mathcal{O}(n)$.

Die meiste Zeit nimmt das Erstellen der Trapezdekomposition in Anspruch, so dass sich eine fast lineare Laufzeit von $\mathcal{O}(n \cdot log(n))$ ergibt. Da dieser Algorithmus schnell arbeitet und leicht zu implementieren ist, findet er in der Praxis häufig Verwendung.

```
stack ← first two vertices
v ← third Vertex
while v != bottom vertex do
    if v is on opposite chain then
        build diagonals to all Vertices on Stack
        clear Stack
        push v_-1 and v
    else
        if v is convex then
            push v
        else
            draw all possible diagonals to stack vertices
            pop all these vertices
            save bottom vertex of stack
        end if
    end if
    v ← v_+1
end while
```

Algorithmus 4.4: Triangulation y-monotoner Polygone. Vorraussetzung ist, dass die Vertices bereits absteigend nach y-Koordinaten sortiert sind.

Triangulation in linearer Zeit

Den bisher einzigen deterministischen Algorithmus zur Triangulation in linearer Zeit hat Chazelle 1990 vorgestellt [10]. Er ist allerdins dermaßen kompliziert, dass eine Implementation praktisch unmöglich ist. Ein einfacherer propabilistischer Algorithmus, der eine erwartete Laufzeit von $\mathcal{O}(n)$ hat, wurde 2001 von Amato et. al [30] veröffentlicht. Frei verfügbare Implementationen von solchen Algorithmen gibt es meines Wissens bisher nicht, so dass auf die genauen Funktionsweisen hier nicht weiter eingegangen werden soll.

4.4.4 Triangulation mit OpenGL

Ein effektives, frei verfügbares Werkzeug zum Triangulieren von Polygonen bietet der Tesselator der OpenGL Utility Library. Diese Bibliothek benötigt keinen Renderkontext, so dass sie unabhängig vom verwendeten Grafiktreiber ist. Sie lässt sich daher im Gegensatz zu den anderen OpenGL-Modulen auch auf Systemen ohne grafische Benutzeroberfläche verwenden und zusammen mit anderen APIs, z.B. DirectX, verwenden. Des Weiteren ist der OpenGL-Tesselator auch in der Lage, komplexe Polygone zu triangulieren. Sich schneidende Kanten oder Löcher bereiten keine Probleme.

Im Folgenden soll gezeigt werden, wie sich diese Bibliothek verwenden lässt, um die zuvor gewonnenen Konturen der planaren Gebiete im Umgebungsmodell zu triangulieren. Nach der Triangulierung soll das vereinfachte Modell nur durch einen Index- und Vertexbuffer repräsentiert

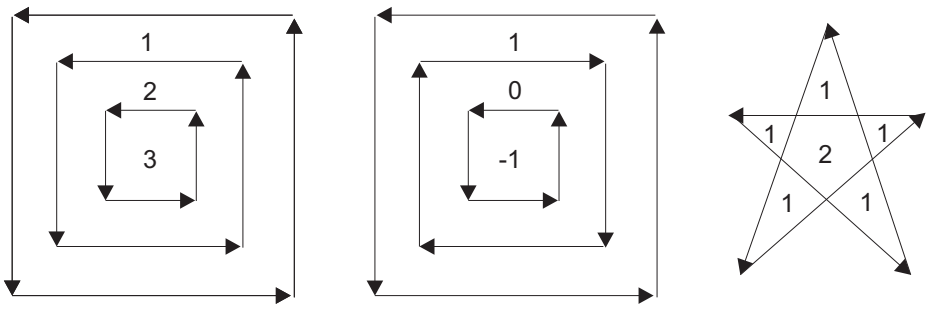


Abbildung 4.14: Windungszahlen für verschiedene Polygone (nach [11]).

werden. Dies hat den Vorteil, dass sich das Modell leicht in verschiedene Dateiformate exportieren lässt.

Festlegung der Windungsregel

Damit die Triangulation die gewünschten Ergebnisse liefert, muss bei der Initialisierung des Tesselators festgelegt werden, wie Löcher in Polygonen behandelt werden sollen. Dies geschieht durch Festlegung einer so genannten *Windungsregel* (engl. *Winding Rule*), die Gebiete innerhalb eines Polygons anhand von Windungszahlen (*Winding Numbers*) klassifiziert.

Die Windungszahl legt für jeden Punkt im Polygon fest, wie oft er umrundet wird, wenn man einer Kontur des Polygons folgt. Durchläuft man die Kontur gegen den Uhrzeigersinn, ist die Windungszahl positiv, ansonsten negativ. Besitzt ein Polygon mehrere Konturen, werden in innen liegenden Gebieten die Windungszahlen aller umgebenden Konturen addiert. Beispiele für die Zuweisung von Windungszahlen zeigt Abbildung 4.14. Mit Hilfe dieser Zahlen kann anhand einer Windungsregel entschieden werden, welche Gebiete innerhalb eines Polygonzuges durch Dreiecke approximiert werden sollen. Die in OpenGL definierten Windungsregeln und die Auswirkungen ihrer Anwendung zeigt Abbildung 4.15.

In den Löchern der Konturpolygone des Polygonnetzes können keine weiteren Konturen mehr liegen, da bei der Flächenextraktion nur zusammenhängende Gebiete vereinigt wurden. Es können also nur die Windungszahlen eins oder zwei auftreten, wobei die durch eins gekennzeichneten Gebiete diejenigen sind, die trianguliert werden müssen. Regionen mit der Windungszahl zwei liegen innerhalb von Löchern und müssen nicht beachtet werden. Es muss also die Regel "ODD" aus Abbildung 4.15 angewendet werden, um das gewünschte Resultat zu erzielen.

Definition der Polygone

Um die Daten eines Polygons an den Tesselator weiterzugeben, werden insgesamt fünf Methodenaufrufe benötigt: `gluTessBeginPolygon`, `gluTessBeginContour`, `gluTessVertex`, `gluTessEndContour` und `gluTessEndPolygon` [13].

Mit gluTessBeginPolygon wird ein neues Polygon eingeleitet. Anschließend müssen die Konturen des Polygons definiert werden. Eine neue Kontur wird mit `gluTessBeginContour` begonnen.

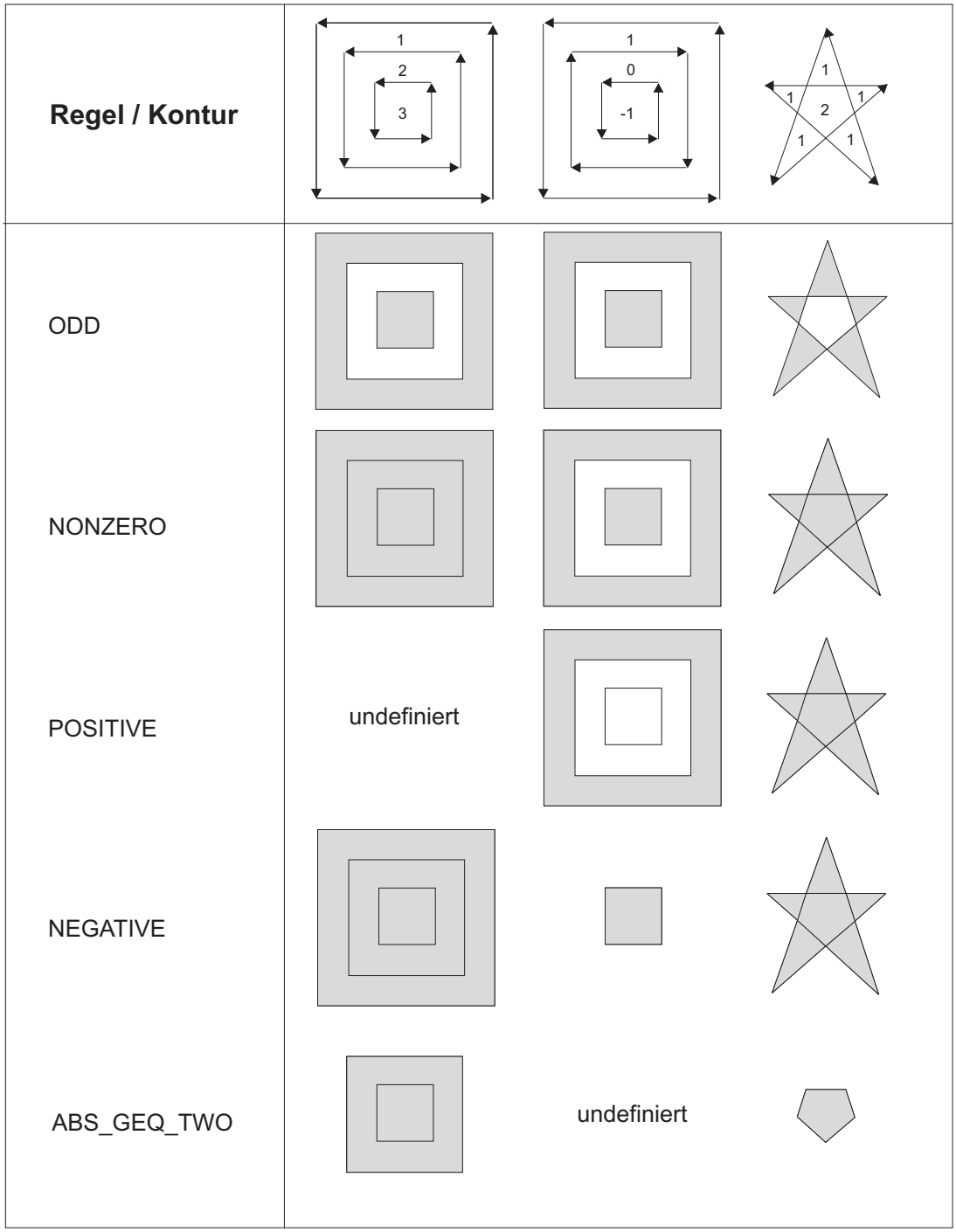


Abbildung 4.15: Drei verschiedene Konturen mit ihren Windungszahlen und den Auswirkungen der verschiedenen Windungsregeln.

Danach werden alle Vertices der aktuellen Kontur durch Aufruf von gluTessVertex definiert und diese zuletzt mit *gluTessEndContour* geschlossen. Nach Übergabe aller Konturen wird die Definition des Polygons mit *gluEndPolygon* beendet.

Die Methode gluTessVertex ist in C wie folgt deklariert:

```
void gluTessVertex(
    GLUtesselator * tess,
    GLdouble coords[3],
    void * data
);
```

Der Aufruf benötigt drei Parameter: Einen Zeiger auf das Tesselator-Objekt, ein Double-Array mit drei Vertex-Koordinaten und einen Zeiger auf eine beliebige Datenstruktur. Mit Hilfe dieses Zeigers können zusätzliche Vertex-Informationen wie Normalen oder Texturkoordinaten, die für die Triangulation aber nicht von Bedeutung sind, übergeben werden. In meiner Implementation werden dazu die Zeiger auf den entsprechenden HalfEdge-Vertex übergeben.

```
for all polygons do
    gluTessBeginPolygon(tesselator)
    for all contours in polygon do
        gluTessBeginContour
        for all vertices in contour do
            gluTessVertex(tesselator, vertexPosition, vertexData)
        end for
        gluTessEndContour
    end for
    gluTessEndPolygon
end for
```

Algorithmus 4.5: Eingabe der Konturpolygone in den OpenGL Tesselator

Bestimmung der Datenstruktur

Nach der vollständigen Definition eines Polygons wird die Triangulation gestartet. Im Verlauf der Verarbeitung werden vom Tesselator verschiedene Callback-Methoden aufgerufen. Mit Hilfe dieser Methoden können u.a die erzeugten Dreiecke gespeichert und auftretende Fehler erkannt werden.

Der Tesselator liefert die Dreiecke im Verlauf der Triangulation in verschiedenen Datenstrukturen, wie sie auch beim Rendern mit OpenGL verwendet werden. Möglich sind einzelne Dreiecke (Triangles), Triangle-Fans und Triangle-Strips. Bei einer Triangle-List bilden jeweils drei aufeinander folgende Vertices ein Dreieck. Ein Triangle-Fan ist eine zusammenhängende Fläche, in der alle Dreiecke einen gemeinsamen Vertex haben. Sobald mehr als zwei Vertices vorhanden sind, bilden die Vertices 1, i und $i-1$ ein Dreieck, wobei i die Nummer des aktuellen Vertexes ist. Ein Triangle-Strip ist ebenfalls eine zusammenhängende Fläche. Dort haben aufeinanderfolgende Dreiecke eine gemeinsame Kante. Bei gerader Vertexnummer und $i > 3$ bilden die Vertices i,

$i-1$ und $i-2$ ein Dreieck. Andernfalls wird das Dreieck durch i, $i-2$ und $i-1$ definiert. Durch diese Art der Nummerierung wird gewährleistet, dass die Dreiecke immer entgegen dem Uhrzeigersinn definiert werden. Bei Triangle-Fan und Triangle-Strip wird mit jedem neuen Vertex auch ein neues Dreieck definert. Abbildung 4.16 zeigt die verschiedenen vom Tesselator erzeugten Datenstrukturen.

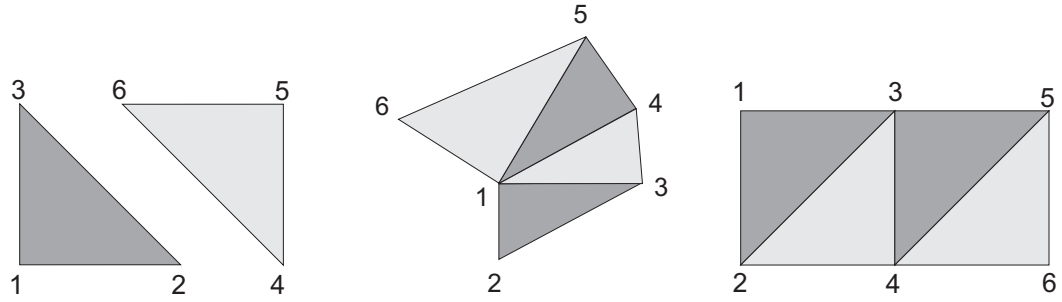
Während der Tesselierung werden bevorzugt Triangle-Fans [41] generiert. Das liegt vermutlich daran, dass der Tesselator versucht, konvexe Unterpolygone zu erzeugen, die dann durch Ziehen von Diagonalen trianguliert werden. Dabei entstehen automatisch Triangle-Fans. Leider gibt es aber keine Veröffentlichung, in der genau beschrieben ist, wie der Tesselator die Polygone zerlegt.

Der Beginn einer neuer Datenstruktur wird durch den Aufruf der unter dem GL_TESS_BEGIN-Callback registrierten Methode angezeigt. Dieser wird als Parameter der Typ der neuen Datenstruktur übergeben. Damit die Vertices der folgenden Dreiecke in der richtigen Reihenfolge abgespeichert werden können, wird die Art der aktuellen Datenstruktur in einer globalen Variablen gespeichert. Bevor eine neue Datenstruktur angefangen wird, ruft der Tesselator den GL_TESS_END-Callback auf.

Aufbau von Index- und Vertex-Buffer

Wird vom Tesselator ein mittels gluTessVertex definierter Punkt des Polygons einem Dreieck in der Triangulation zugewiesen, wird der GL_TESS_VERTEX- bzw. GL_TESS_VERTEX_DATA-Callback aufgerufen. Der GL_TESS_VERTEX_DATA-Callback erhält neben den Koordinaten des aktuellen Vertexes auch den bei der Definition angegebenen Zeiger auf zusätzliche Daten. Mit Hilfe dieses Zeigers ist es möglich, die Position des Vertexes im Vertex-Buffer weiterzugeben und somit einen neuen Index-Buffer aufzubauen.

Dazu werden die Indizes der beiden zuvor benutzten Vertices zwischengespeichert. Wird ein Triangle-Fan oder ein Triangle-Strip generiert, erzeugt jeder Aufruf des Callbacks ein neues Dreieck gemäß der zuvor genannten Nummerierung, dessen Vertex-Indizes in den Index-Buffer eingefügt werden. Bei der Erzeugung von einzelnen Dreiecken, wird bei jedem dritten Callback-Aufruf ein neues Dreieck abgelegt. Auf diese Art und Weise können alle vom Tesselator erzeugten



(a) Triangles (b) Triangle-Fan (c) Triangle-Strip

Abbildung 4.16: Typen von Dreiecks-Listen in OpenGL

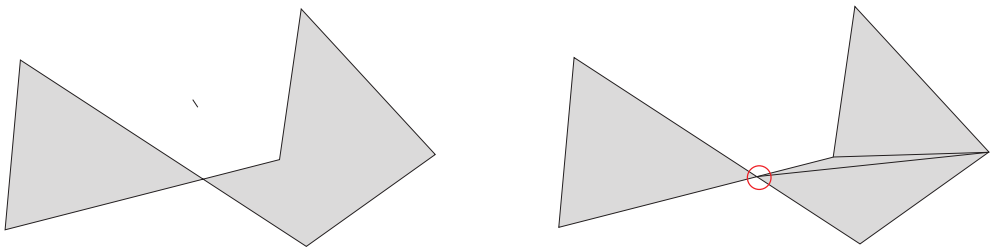


Abbildung 4.17: Triangulierung eines konplexen Polygons mit sich schneidenden Kanten. An der rot markierten Stelle mus während der Triangulation ein neuer Vertex eingefügt werden.

Dreiecke in einer einzigen Bufferstruktur gespeichert werden.

Behandlung neuer Vertices

Gelegentlich kommt es vor, dass der Tesselator einen neuen Vertex generiert. Das geschieht z.B. dann, wenn sich zwei Kanten eines Polygons schneiden (s. Abbildung 4.17). Ist dies der Fall, wird der GL_TESS_COMBINE_DATA-Callback aufgerufen. Die Koordinaten des neuen Vertexes werden vom Tesselator berechnet. Er kann aber nicht wissen, wie der Vertex weiter verwendet werden soll und wie die übergebenen Zusatzinformationen gehandhabt werden müssen. Es ist daher notwendig, die fehlenden Informationen in der Callback-Methode zu generieren.

Die zum GL_TESS_COMBINE_DATA-Callback gehörende Methode ist wie folgt deklariert:

```
void __stdcall combine(GLdouble coords[3],
  void* vertex_data[4],
  GLfloat weight[4],
  void** outData
);
```

Der erste Parameter enthält ein Array mit den neuen Vertex-Koordinaten. Der zweite liefert Zeiger auf die den vier Nachbar-Vertices übergebenen Zusatzdaten. Im dritten Parameter werden vier Gewichte geliefert, die angeben, welchen Einfluss die vier Nachbarn auf den neuen Vertex haben. Mit Hilfe dieser Gewichte können z.B. Texturkoordinaten oder Normalen für den neuen Vertex aus den Daten der Nachbarn interpoliert werden.

Nach der Interpolation werden die neu erzeugten Zusatzinformationen mit dem neuen Vertex assoziiert indem mit Hilfe des Parameters outData dem Tesselator ein Zeiger auf Daten übergeben wird. Anschließend wird der neue Vertex in den Vertexbuffer eingefügt und mit einem neuen Index versehen. Dazu wird dem Vertex der aktuelle globale Index zugewiesen, der dann erhöht wird.

Ergebnisse

Beispiele für einige Polygone und für mit Hilfe des Tesselators erzeugte Triangulationen zeigen Abbildungen 4.18 und 4.19:

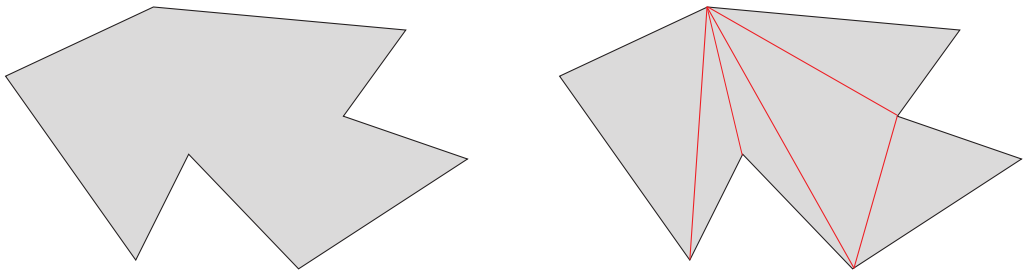


Abbildung 4.18: Triangulierung eines konkaven Polygons

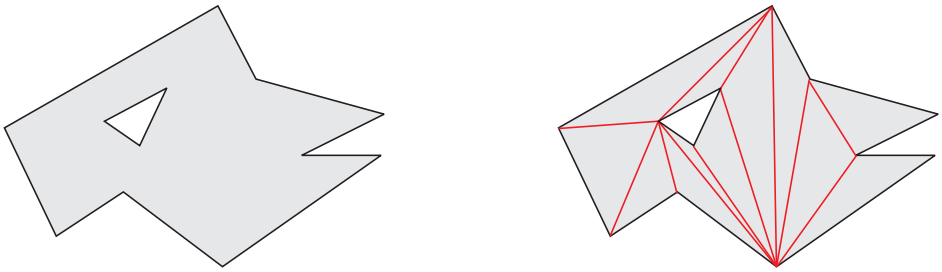


Abbildung 4.19: Triangulierung eines Polygons mit Loch

4.5 Fazit

Mit Hilfe der HalfEdge-Datenstruktur des Ausgangsmodells können die Konturen planarer Gebiete effizient in eine Polygondarstellung überführt werden. Die gesamte Laufzeit des Extraktionsalgorithmus beträgt $\mathcal{O}(n)$ in der Anzahl der im Modell vorhandenen Faces.

Ein komplexes Problem ist die anschließende Triangulation dieser Polygone. Dazu werden ausgefeilte Algorithmen benötigt, die diese Aufgabe bewältigen können. Eine Möglichkeit bietet die Verwendung des OpenGL-Tesselators. Leider liegen keine Informationen über die Konplexitäten der dort verwendeten Algorithmen vor. Umfangreiche Tests haben allerdings gezeigt, dass die Performance des Tesselators hoch ist. Daher wird er in der Implementation zur Triangulierung der Konturpolygone verwendet.

Ein Nachteil des vorgestellten Reduktionsverfahrens ist die Tatsache, dass die vereinfachten Daten nicht mehr in einer verlinkten Datenstruktur mit Nachbarschaftsinformationen zwischen den einzelnen Faces vorliegen. Dies liegt daran, dass durch das Zusammenfassen zusammenhängender Gebiete Grenzkanten entstehen können, die mehrere Nachbarn haben (sog. *T-Kanten*, s. Abbildung 4.20). Dieser Effekt ist beim Rendern der Modelle allerdings nicht zu sehen und kann daher vernachlässigt werden.

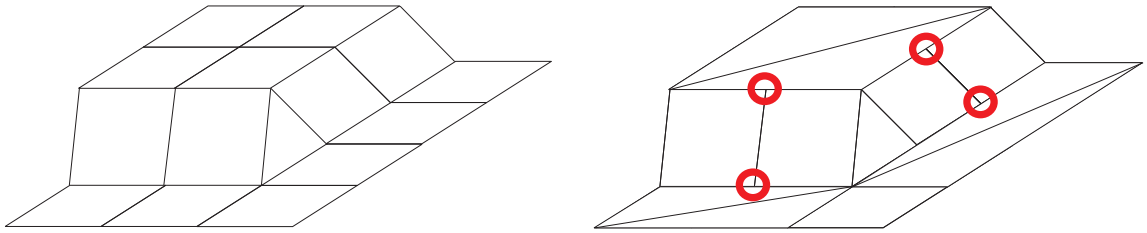


Abbildung 4.20: Entstehung von T-Kanten. Sobald große Flächen neu trianguliert wurden, können die Begrenzungskanten dieser Gebiete mehrere Nachbarfaces haben.

Kapitel 5

Texture Mapping

Im Folgenden soll beschrieben werden, wie die aus den 3D-Laserdaten gewonnenen Modelle mit Texturen versehen werden können, um sie realistischer wirken zu lassen. Dabei werden Eigenschaften wie Oberflächenmuster oder die Rauheit der Objektoberflächen, die sich mit Polygonen nur sehr aufwändig modellieren ließen, unter Verwendung spezieller Bitmaps, sog. Texturemaps, festgelegt.

Zunächst soll ein kurzer Überblick über die Technik des Texture-Mappings gegeben werden. Anschließend wird gezeigt, wie den Vertices der Konturpolygone Texturkoordinaten zugewiesen werden können. Im dritten Teil dieses Abschnitts wird ein Verfahren vorgestellt, mit dem sich anhand der Marching-Cubes-Indizes Flächen klassifizieren lassen, denen anschließend passende Standardtexturen zugeordnet werden.

Ein Beispiel für ein texturiertes Objekt zeigt Abbildung 5.1. Als Textur wurde eine Aufnahme der Erdoberfläche verwendet, die mit Hilfe einer sphärischen Projektion auf eine Kugel übertragen wurde, wodurch ein virtueller Globus entstanden ist.

Abbildung 5.1: Beispiel zum Texture-Mapping. Links eine einfach beleuchtete Kugel, die mit der in der Mitte gezeigten Textur versehen wurde. Das Ergebnis ist im rechten Bild zu sehen (entnommen aus [50]).

(a) Repeat (b) Clamp (c) Mirror

Abbildung 5.2: Randwiederholungsfunktionen beim Texture Mapping.

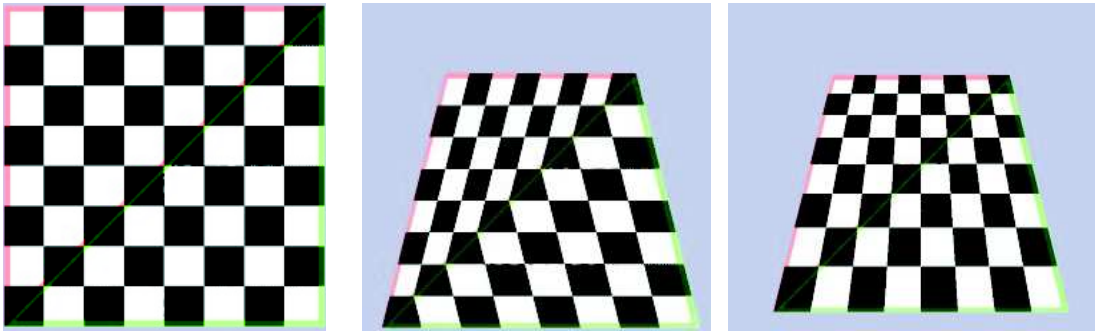


Abbildung 5.3: Perspektivenverzerrungen beim Texture-Mapping. Links die verwendete Textur, die auf zwei Dreiecke gemappt wird. Das Bild in der Mitte zeigt das Ergebnis der Interpolation ohne Perspektivenkorrektur. Rechts sieht man das Ergebnis unter Berücksichtigung der z-Komponente. (entnommen aus [50])

5.1 Grundlagen

Beim Texture-Mapping werden die Oberflächen der Objekte ähnlich einer Tapete mit Bildern (Texturen) versehen, die die Struktur und Farbe der Oberflächen wiedergeben. Dazu werden den Vertices der Polygone zweidimensionale Texturkoordinaten (u, v) zugeordnet, die die Position dieses Punktes innerhalb der Textur wiedergeben. Der untere linke Punkt der Textur erhält üblicherweise die Koordinaten $(0, 0)$, der obere rechte die Koordinaten $(1, 1)$. Es ist möglich, den Vertices der Polygone auch Koordinaten außerhalb des Intervalls $[0, 1]$ zuzuweisen. In solchen Fällen wird eine geeignete Randwiederholungsfunktion, z.B. Wiederholung oder Spiegelung der Textur, durchgeführt. Die von vielen Grafik-APIs unterstützten Randwiederholungseffekte zeigt Abbildung 5.2.

Beim Rendern der Polygone werden im einfachsten Fall zunächst die Koordinaten linear entlang der Begrenzungskanten interpoliert. Zwischen den so gewonnenen Werten wird zeilen- oder spaltenweise entlang des Bildschirms interpoliert und die Farbe des korrespondierenden Pixels in der Textur übernommen. Bei diesem Verfahren können aber perspektivische Verzerrungen auftreten,

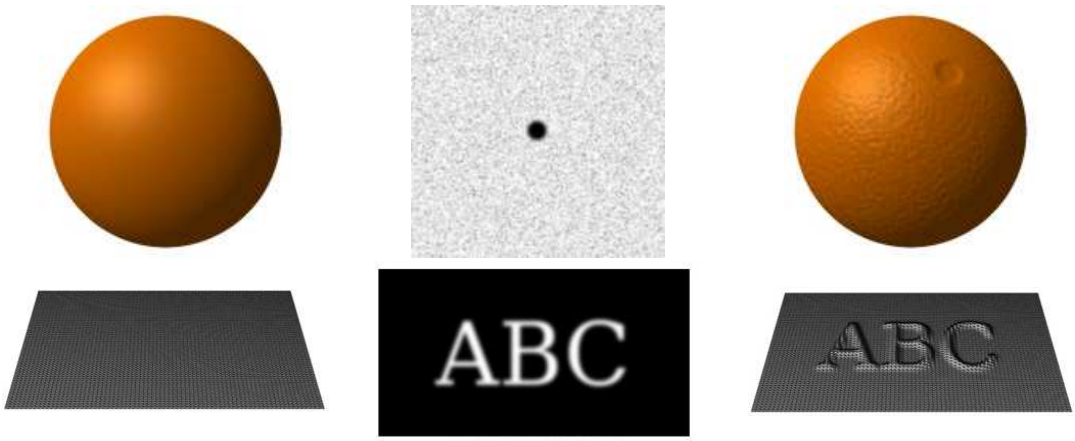


Abbildung 5.4: Bump-Mapping (oben) und Displacement-Mapping (unten). Links sind jeweils die Ausgangsmodelle gezeigt. In der Mitte stehen die benutzten Maps. Die rechte Spalte zeigt die Modelle, nachdem die Maps angewendet wurden (entnommen aus [47, 48]).

da die Interpolation erst durchgeführt wird, nachdem die Polygone in den zweidimensionalen Texturraum projiziert wurden (s. Abbildung 5.3). Daher muss auch auch die z-Komponente der Polygonvertices berücksichtigt werden, um eine perspektivisch korrekte Wiedergabe zu erreichen [20]:

$$\begin{aligned} u &= (u/z) \\ v &= (v/z) \end{aligned}$$

Ein weiteres Problem ist das korrekte Übernehmen der Pixelfarben. Da bei der Interpolation der Texturkoordinaten auch Positionen bestimmt werden können, die zwischen zwei Pixeln der Textur liegen, müssen die zu verwendenden Farbwerte mit geeigneten Verfahren bestimmt werden. Die einfachste Variante ist es, die Farbe des nächstegelegenen Pixels zu übernehmen. Da Farbwerte eines Pixels beim Rendern der Polygone mehrfach übernommen werden können, wirken die Texturen auf den Objekten sehr „pixelig". Um diesen Effekt zu vermindern werden die Farbwerte daher in der Regel mit einem geeigneten Filter interpoliert. Eine weit verbreite Methode ist die bilineare Filterung.

Texturen können auch angewendet werden, um andere Eigenschaften als die Farben der Polygone festzulegen. So werden beim sog. Bump-Mapping beim Rendern vorher festgelegte Normalenwerte zugewiesen, um Beleuchtungseffekte zu erzielen. Beim Displacement-Mapping werden mit Hilfe von in Texturen gespeicherten Informationen die Vertexpositionen beeinflusst (s. Abbildung 5.4).

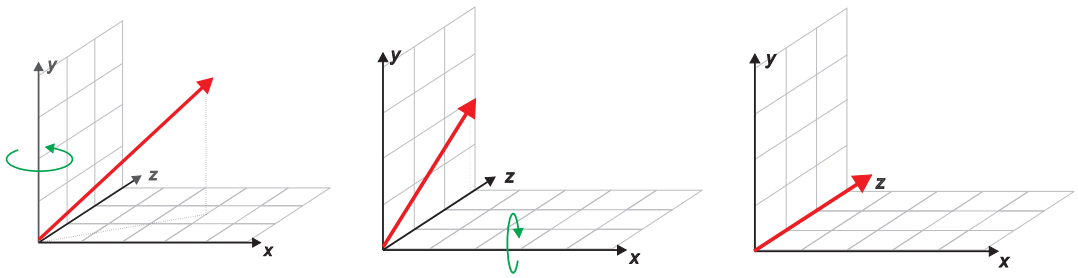


Abbildung 5.5: Projektion der Polygone in die x-y-Ebene. Dazu wird die Flächennormale zunächst um die y-Achse in die y-z-Ebene und anschließend auf die z-Achse gedreht.

5.2 Berechnung der Texturkoordinaten

Sollen die Polygone mit sich wiederholenden Texturen, z.B. bestimmten Mustern, versehen werden, genügt es, die Polygone in die x-y-Ebene zu projezieren. Dann können die x- und y-Komponenten der Vertices direkt als Texturkoordinaten verwendet werden.

Im Folgenden soll beschrieben werden, wie man den Vertices eines Polygons Texturkoordinaten so zuweisen kann, dass alle Vertices Koordinaten im Intervall [0,1] haben. Es können also keine Wiederholungseffekte auftreten. Dies ist nützlich, wenn man die gefundenen Konturpolygone im Modell mit bestimmten Daten, wie z.B. aufbereiteten Fotos der Szene, korrelieren will. Zur Berechnung der Koordinaten sind dann drei Schritte nötig: Projektion des Polygons in die x-y-Ebene, Verschiebung der unteren rechten Vertexposition in den Ursprung und Zuweisung der Texturkoordinaten.

Die Grundidee zur Berechnung der Projektion ist es, die Flächennormale eines Polygons in Richtung der z-Achse zu drehen. Dazu wird zunächst der Schwerpunkt S berechnet und in den Ursprung verschoben (baryzentrische Koordinaten):

$$S = \frac{1}{N} \sum_{i=1}^{N} V_i$$

In einem zweiten Schritt werden alle Polygonvertices in die $x-y$-Ebene gedreht. Die Drehwinkel werden ermittelt, indem die Flächennormale des Polygons zunächst um die y-Achse in die y-z-Ebene und danach um die $x-$Achse auf die z-Achse gedreht wird (s Abbildung 5.5). Es gilt:

$$\begin{aligned} \tan \alpha_y &= n_x / n_z \\ \tan \alpha_x &= n_z / n_y. \end{aligned}$$

Anschließend werden die Vertices mit Hilfe der folgenden Drehmatrizen transformiert:

$$\mathrm{RotX} = \begin{pmatrix} \cos\alpha_x & -\sin\alpha_x & 0 \\ \sin\alpha_x & \cos\alpha_x & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad \mathrm{RotY} = \begin{pmatrix} \cos\alpha_z & 1 & -\sin\alpha_z \\ 0 & 1 & 0 \\ \sin\alpha_z & 0 & \cos\alpha_z \end{pmatrix}$$

Um allen Vertices Texturkoordinaten zuweisen zu können, werden die größten und kleinsten auftretenden Koordinatenwerte bestimmt. Mit diesen lässt sich ein Rechteck bestimmen, das das Polygon komplett umschließt (die sog. *Bounding Box*). Damit nur positive Werte auftreten können, wird das Polygon so verschoben, dass der untere rechte Punkt der Bounding Box im Ursprung liegt. Für die Texturkoordinaten gilt dann:

$$\begin{aligned} u &= v_x/l_x \\ v &= 1-(v_y/l_y) \end{aligned}$$

Hierbei bezeichnen v_x und v_y die x- und y-Komponenten eines Vertices, l_x und l_y die Längen der Bounding Box. Der gesamte Transformationsprozess ist in Abbildung 5.6 dargestellt. Abbildung 5.7 zeigt Beispiele einer Szene, die mit und ohne Festlegen der Texturkoordinaten auf das Intervall [0,1] gerendert wurden. Man kann erkennen, dass in beiden Fällen die Texturen perspektivisch korrekt gemappt werden. Bei Beschränkung der Koordinaten wird der Bereich der Textur, der nach der Projektion von der Kontur eingenommen wird, auf das Polygon übertragen (s. Abbildung 5.7(a)). Bei direkter Übernahme der Vertexpositionen wird die voreingestellte Randwiederholungsfunktion *Repeat* angewendet (s. Abbildung 5.7(b)).

5.3 Automatische Zuweisung von Texturen

Datensätze, die in von Menschen gebauten Umgebungen aufgenommen werden, enthalten in der Regel viele planare Flächen, die sekrecht zueinander stehen, wie z.B. Fußboden, Wände und Decken in Gebäuden. Dieses Wissen kann ausgenutzt werden, um die verschiedenen Strukturen in solchen Umgebungen mit Hilfe der Marching-Cubes-Indizes zu klassifizieren. Nach der Klassifizierung können ihnen dann vom Benutzer vorgegebenen Standardtexturen zugewiesen werden. Diese können z.B. durch Fotos, die in der Szene gemacht wurdenv gewonnen werden.

Die Marching-Cubes-Grundmuster, die bei der Vereinfachung der Modelle verwendet wurden, erzeugen immer zusammenhängende Flächen. Je nach dem, in welcher Rotation sie vorkommen, kann abgeschätzt werden, ob es sich bei der generierten Fläche um Fußböden, Wände, Rampen, Decken usw. handelt. Muster 8 erzeugt z.B. immer vertikale oder horizontale Flächen. Vertikale Flächen sind in der Regel Wände, horizontale Decken oder Fußböden. Ähnliches gilt für von Muster 10 erzeugte Flächen, diese stehen aber geneigt in der Szene (z.B. Türen, Rampen). Je nach Ausrichtung der Normalen kann dabei zwischen Fußböden und Decken unterschieden werden: Fußböden haben nach oben zeigende Normalen, die Normalen von Decken zeigen nach unten.

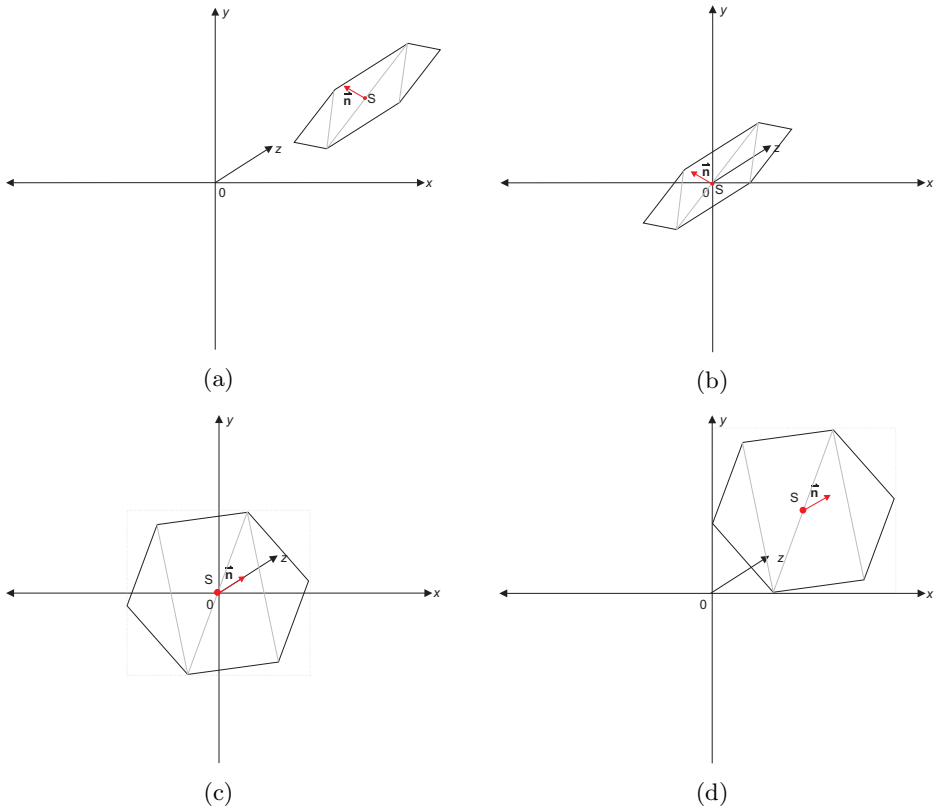
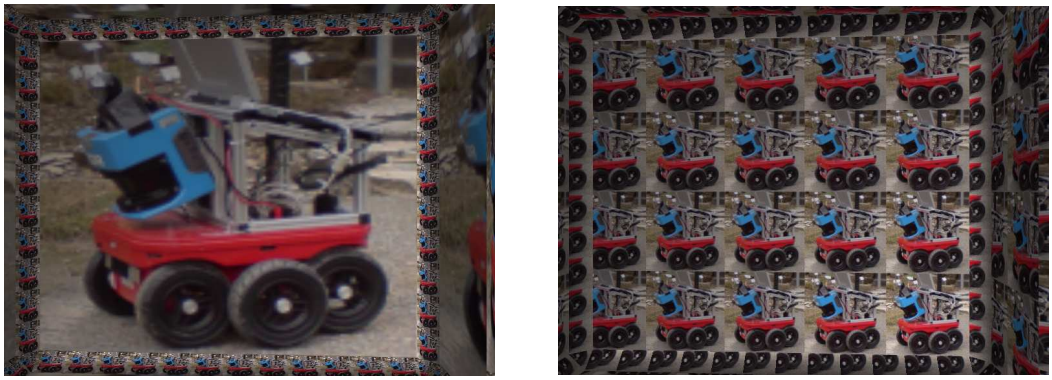


Abbildung 5.6: Projektion zur Berechnung der Texturkoordinaten. (a) Ein Polygon mit Normalen und Schwerpunkt. (b) Der Schwerpunkt des Polygons wird in den Ursprung verschoben. (c) Die Normale wird auf die z-Achse gedreht. Das Polygon liegt nun in der x-y-Ebene. (d) Verschiebung des unteren rechten Punktes der Bounding Box in den Ursprung.

Bei der Erzeugung der Modelle wird jedes Face anhand seines Marching-Cubes-Indices in eine der folgenden Klassen eingeteilt: Decke, Fußboden, Wand, Tür, Unbekannt. Jeder dieser Klassen wird dann beim Rendern eine vorgegebene Textur zugewiesen.

5.4 Fazit

Abbildung 5.8 zeigt ein Beispiel für eine Szene, der automatisch Texturen zugewiesen wurden, vor und nach dem Texture Mapping. Man kann deutlich erkennen, dass Wände, Fußböden und Decken gut erkannt und die Texturen perspektivisch korrekt gerendert wurden. Es zeigt sich aber auch, dass Flächen, die von unten gesehen wurden, immer als Decken klassifiziert wurden. Ebenso wurden von oben gesehene immer als Fußböden erkannt. Die Klassifizierung ließe sich noch verbessern, wenn z.B. die y-Koordinaten mit einbezogen würden: Erst ab einer gewissen Höhe werden Flächen als Decke erkannt. Zudem könnten weitere Randbedingungen betrachtet werden. So treffen z.B. Wände in der Regel senkrecht auf Füßboden und Decken. Diese und ähnliche Überlegungen werden in Zukunft in die Verbesserung des Klassifizierungsalgorithmus einfließen.



(a) (b)

Abbildung 5.7: Wiederholung von Texturen und vollständige Projektion. Im linken Bild wurden die Texturkoordinaten der Polygonvertices auf das Intervall [0,1] beschränkt. Die Textur bei Rechtecken vollständig auf das Polygon gemappt. Im rechten Bild wurden als Texturkoordinaten die x- und y-Kompomenten der Vertices nach der Projektion übernommen. Als Randwiederholungsfuktion wurde *Repeat* gewählt.

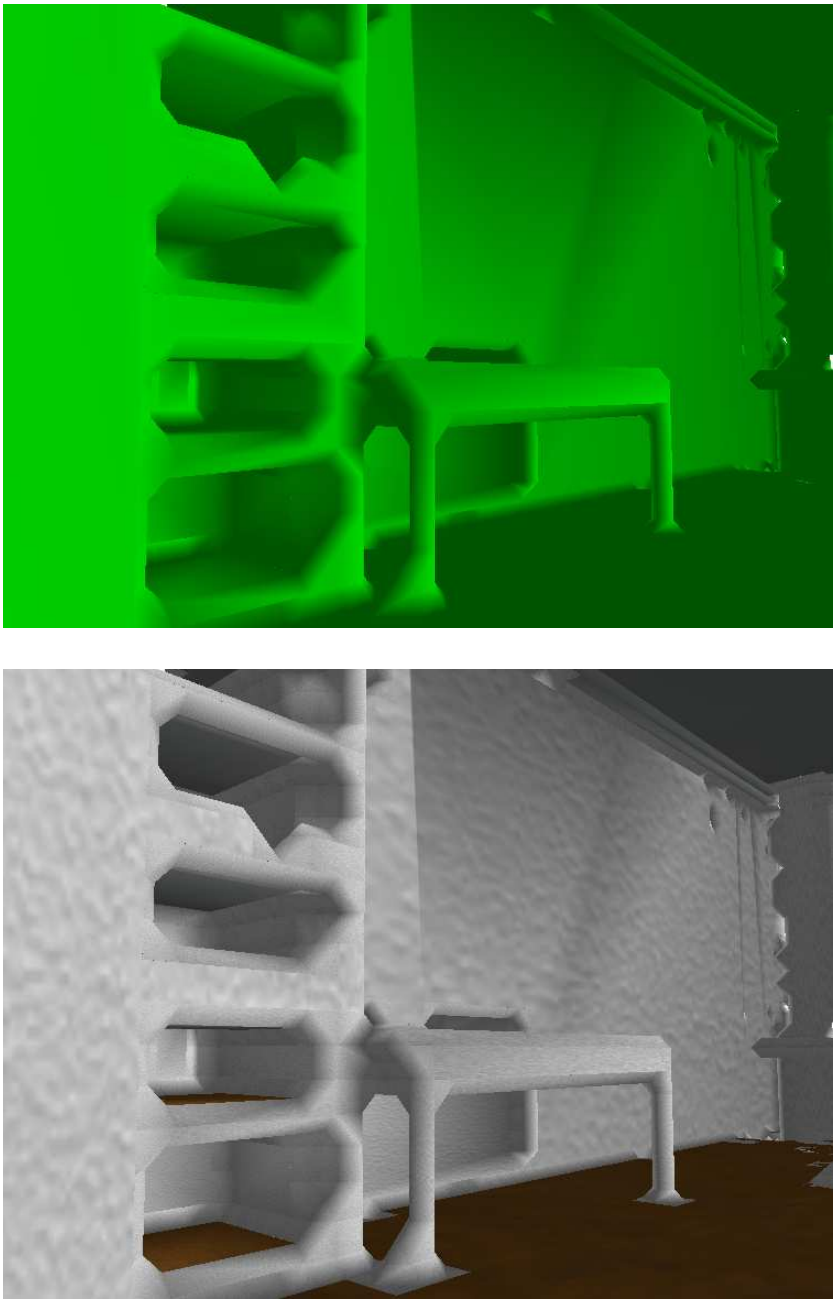


Abbildung 5.8: Beispielszene vor und nach der automatischen Zuweisung von Texturen. Der zugrunde liegende Datensatz ist eine Simulation eines Laserscans, der mit einem mobilen Roboter aufgenommen wurde, und zeigt ein Laborraum mit Tisch und Regalen. Die texturierte Version dieser Szene (unten) wirkt durch die Verwendung von geeigneten Texturen die nach der Klassifikation der Flächen zugewiesen wurden deutlich realistischer.

Kapitel 6

Experimentelle Ergebnisse

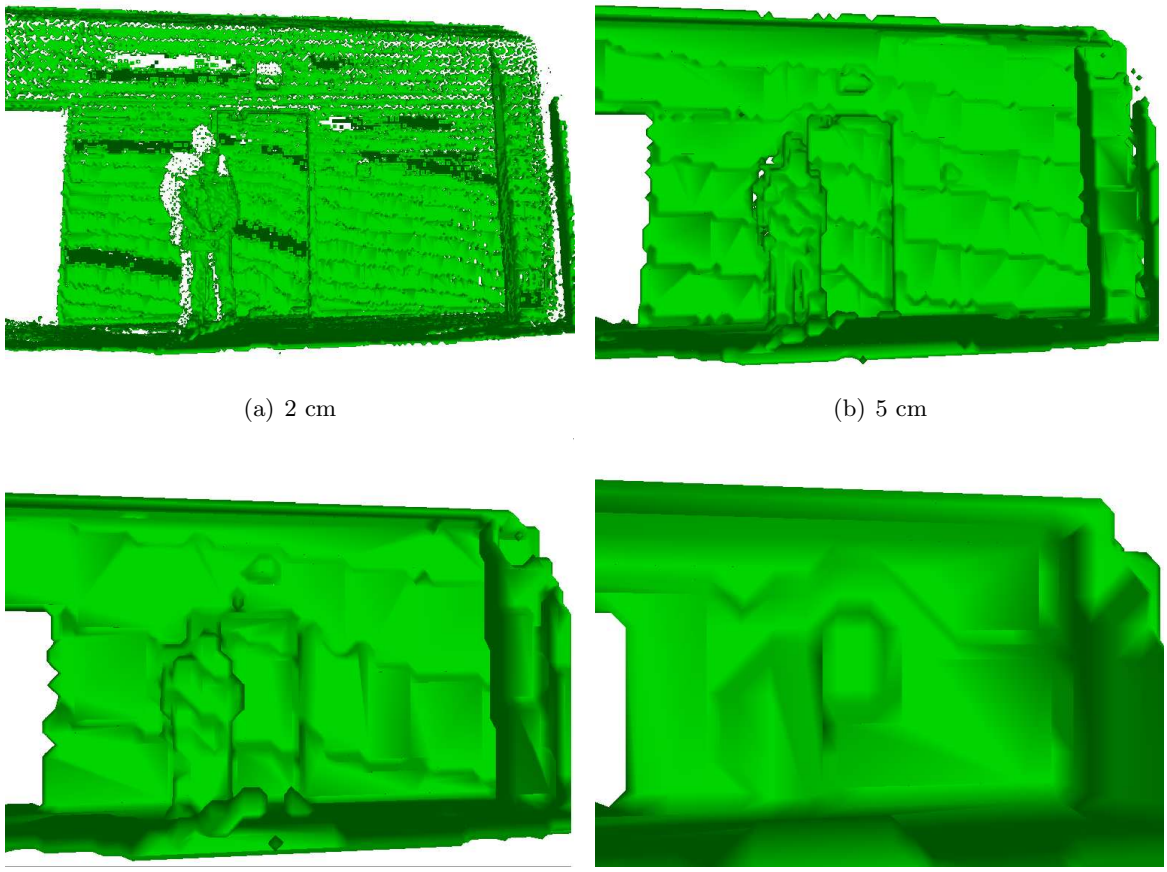
In diesen Kapitel sollen die Ergebnisse des Modellerzeugungs- und Optimierungsprozesses evaluiert werden. Dazu wird zunächst diskutiert, welchen Einfluss die Gitterkonstante des HashGrids auf die Ergebnisse des Modellerzeugungsprozesses hat. Anschließend wird die gerenderte Darstellung verschiedener Modelle untersucht und der neue Vereinfachungsalgorithmus mit bekannten iterativen Verfahren verglichen.

6.1 Einfluss der Zellengröße

Abbildung 6.1 zeigt den Einfluss der gewählten Marching-Cubes-Zellengröße auf die Form des erzeugten Modells anhand eines 3D-Laserscans, der auf dem Flur im 5. Stock des AVZ der Universität Osnabrück von unserem Kurt3D Roboter aufgenommen wurde (Punktauflösung ca. 2 cm). Der Scan zeigt eine Person, die vor einer geschlossenen Tür steht, sowie eine der Säulen im Flur.

Man kann deutlich erkennen, dass sich die Form des Modells stark mit der Zellengröße ändert. Bei einer Gitterkonstanten, die etwas kleiner ist als die Auflösung des Scanners, werden die Flächen, an denen die Punktedichte zu klein ist, aufgebrochen und es bilden sich kleine Strukturen, die im Extremfall einzelne Messpunkte diamantförmig umschließen (a). Bei Zellengrößen, die größer sind als die Scanneraufl̈osung, werden größtenteils zusammenhängende Flächen erzeugt. Mit wachsender Zellengröße gibt es immer weniger Löcher im Modell, allerdings sind aufgrund der groben Raumeinteilung weniger Details zu sehen. Auch der ursprünglich vorhandene Laserschatten hinter der Person wird überdeckt. Die äußeren Konturen des Modells lassen sich aber noch deutlich erkennen (s. (b) und (c)). Steigt die Gitterkonstante weiter an, verschmelzen die Konturen ineinander und die ursprüngliche Form des Modells lässt sich nur noch erahnen.

Die Qualität des aus den Laserdaten rekonstruierten Modells hängt stark von der gewählten Gitterkonstante ab. Bei geschickter Wahl kann die Streuung der Messdaten herausgefiltert werden, und es bleiben gleichzeitig viele Details erhalten. In verschiedenen Versuchen hat sich gezeigt, dass gute Ergebnisse erzielt werden, wenn die gewählte Zellengröße ca. 1.5 bis zweimal so groß ist wie die Scanneraufl̈osung. Probleme ergeben sich bei Scans, in denen die Auflösung variiert.



(a) 2 cm (b) 5 cm

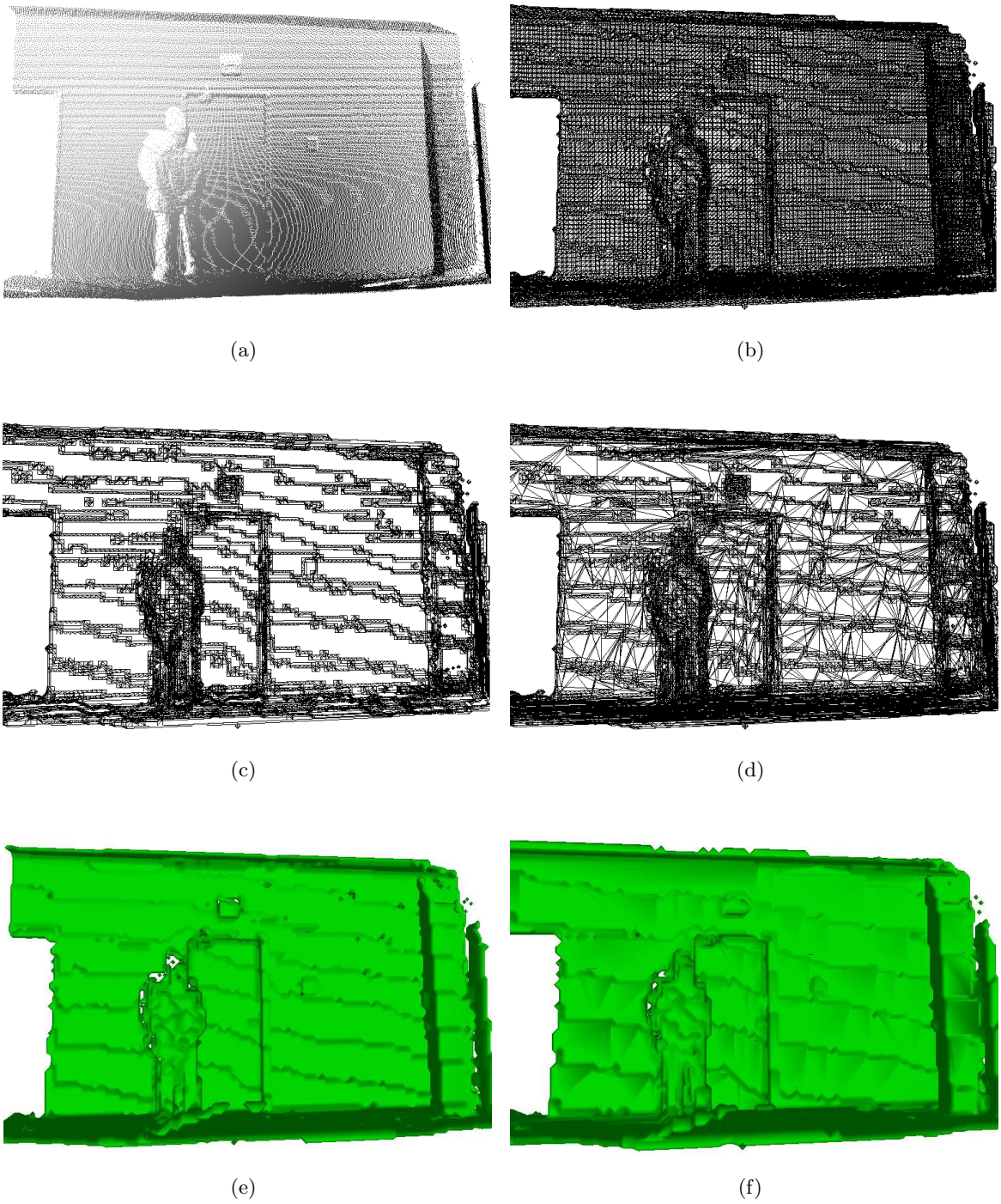
(c) 10 cm (d) 30 cm

Abbildung 6.1: Einfluss der Zellengröße auf die Gestalt des Modells. Bei zu geringer Größe beginnen die Konturen zu zerfallen, und es bilden sich Löcher (a). Mit zunehmender Zellengröße werden die Flächen zusammenhängender, allerdings werden Details verwaschen (b) bis (d).

Hier kann es in Regionen, in denen die Auflösung geringer ist, zu Löchern in Flächen kommen. Ist die Streuung des Sensors allerdings bekannt, kann diese in die Überlegungen mit einbezogen werden, um Messfehler auszugleichen.

6.2 Visuelle Darstellung

Abbildung 6.2 zeigt die Zwischenergebnisse der einzelnen Schritte der Modellerzeugung und Optimierung anhand des AVZ Laserscans. Die erste Zeile des Bildes zeigt die mit dem Laserscanner aufgenommene 3D-Punktwolke (a) und das mit Hilfe des Marching-Cubes-Algorithmus erzeugte initiale Dreiecksnetz (b). Aufgrund von Kalibrierungsfehlern ist der Laserscan leicht um die x-Achse des Koordinatensystems geneigt. Diese Neigung zeigt sich deutlich in einem „Treppen-



(a) (b)

(c) (d)

(e) (f)

Abbildung 6.2: Zwischenschritte im Prozess der Modellgenerierung. Bild (a) zeigt die mit dem Laserscanner aufgenommene Punktwolke. Aus dieser wird im zweiten Schritt mit Hilfe des Marching-Cubes-Verfahrens ein Dreiecksnetz erstellt(b). In den Dreiecksnetz werden dann die Konturpolygone der planaren Flächen extrahiert (c) und anschließend trianguliert (d). Die letzte Zeile zeigt das Ausgangsnetz (e) und das optimierte Netz ohne Texturen gerendert.

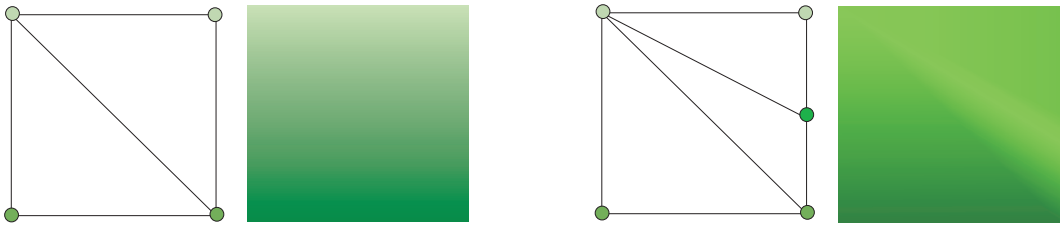


Abbildung 6.3: Entstehung des Machband-Effekts bei der Neutriangulierung. Links die Schattierung bei einer Unterteilung, wie sie im ursprünglichen Netz auftritt. Da die oben- und untenliegenden Kanten jeweils die gleichen Farbwerte an den Vertices haben, entsteht bei der Interpolation nach dem Gouraud-Verfahren ein kontinuierlicher Farbverlauf. Das rechte Bild zeigt dieselbe Zelle, die aber in drei Dreiecke unterteilt wurde. An der rechten Kante wurde ein neuer Vertex eingefügt. Durch die Interpolation von Vertex zu Vertex in einem Dreieck entstehen unterschiedliche Verläufe, die als Linie wahrgenommen werden.

effekt" im Modell, der ensteht, wenn eine gescannte Fläche eine Ebene von Würfelzellen verlässt. Des Weiteren kann man einen deutlichen Laserschatten in der Punktwolke erkennen, der von der vor der Wand stehenden Person erzeugt wird. Dieser Schatten wird in der Rekonstruktion weitestgehend durch die relativ große Kantenlänge der Marching-Cubes-Zellen(hier 5 cm) ausgeglichen.

Die zweite Zeile zeigt die Zwischenergebnisse des Reduktionsverfahrens. Links ist das Netz nach der Extraktion der Begrenzungskanten der planaren Regionen zu sehen. Man kann deutlich erkennen, dass, wie zu erwarten, bevorzugt im Bereich der Wände solche Gebiete gefunden wurden. Im Bereich, in dem die Person die Wand verdeckt und an den Übergängen zwischen den Ebenen werden kaum Dreiecke entfernt, da diese eine hohe Krümmung aufweisen und daher vom Vereinfachungsalgorithmus nicht zusammengefasst werden können. Das rechte Bild zeigt das Ergebnis der Triangulation. Dabei werden die ebenen Gebiete wieder in Dreiecke zerlegt.

In der unteren Reihe sind das Ausgangsmodell (a) und das reduzierte Modell (b) mit Gouraud-Shading gerendert. Man kann deutlich erkennen, dass im vereinfachten Modell einige Schattierungsfehler vorkommen. Dieser Effekt ist auf die Triangulation in Kombination mit dem Gouraud-Shading zurückzuführen. Bei diesem Schattierungsverfahren werden zunächst die Farbwerte an den drei Ecken gemäß eines Beleuchtungsmodells bestimmt. Anschließend wird entlang der Dreieckskanten von einem Farbton zum anderen linear interpoliert. Zuletzt wird das Dreieck gefüllt, indem entlang einer horizontalen Scanline die Farbwerte zwischen den Schnittpunkten dieser Linie mit den Dreieckskanten interpoliert werden.

Im ursprünglichen Marching-Cubes-Modell wurden diese Farbverläufe lokal innerhalb einer Zelle interpoliert, da die Vertices in benachbarten Dreiecken durch die Mittelung die gleichen Normalen hatten und das Beleuchtungsmodell somit dort die gleichen Farbwerte ermittelt. Nach der Triangulation werden die Farbverläufe allerdings über große Anteile der planaren Regionen interpoliert. Es kann dabei passieren, dass benachbarte Dreiecke mit unterschiedlichen Farbverläufen gerendert werden, wenn die Vertices, die nicht auf der gemeinsamen Kante liegen, verschiedene Normalen haben. Die Grenze zwischen den Flächen fällt dabei besonders auf, da das menschliche Auge Unstetigkeiten bis zur zweiten Ableitung erkennem kann und diese Kontraste verstärkt wahrgenommen werden (*Machband-Effekt*, s. Abbildung 6.3 [38]).

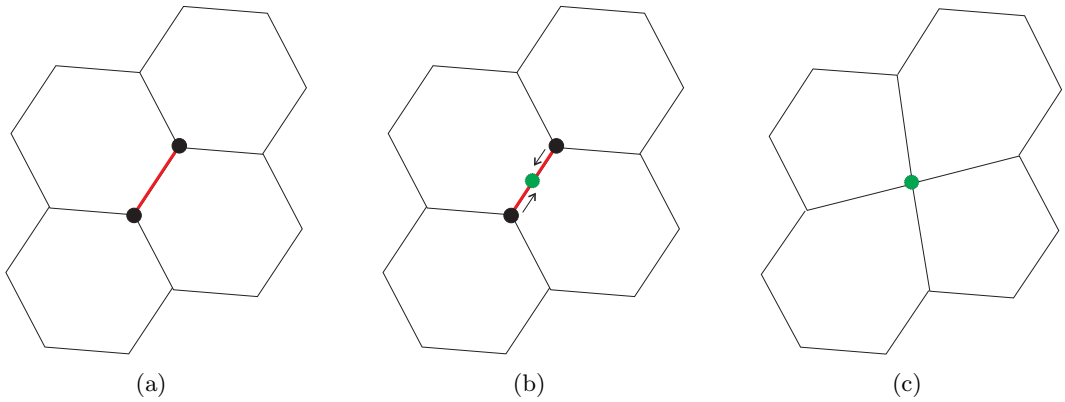


Abbildung 6.4: Der Edge-Collapse-Prozess. Zunächst wird die zu entfernende Kante des Polygonnetzes bestimmt (a). Anschließend wird ein neuer Vertex in der Mitte dieser Kante erzeugt (b), und die beiden Endvertices werden auf diesen Punkt gezogen (c).

Dieser Effekt ließe sich verhindern, wenn allen Vertices im Polygon die gleiche Normale zugewiesen würde. Man könnte das Modell dann allerdings nur noch mit Flat-Shading (d.h. die gesamte Fläche wird in der selben Farbe ausgefüllt) rendern. Eine Alternative wäre es, den beiden Normalen der Vertices einer Kante der Begrenzungspolygone denselben Wert zuzuordnen. Es ist allerdings sehr schwierig, dabei eine konsistente Belegung zu erreichen, da es im reduzierten Netz in der Regel T-Kanten an den Übergängen von planaren zu gekrümmten Flächen gibt. Da der Machband-Effekt nach der Texturierung wesentlich weniger ins Auge fällt und die Berechnung einer konsistenten Normalenzuweisung sicherlich aufwändig wäre, wurde bis jetzt darauf verzichtet, entsprechende Lösungen zu entwickeln.

6.3 Vergleich mit anderen Verfahren

Im Folgenden sollen die Ergebnisse der Modelloptimierung durch Triangulations der ebenen Flächen mit anderen Methoden verglichen werden. Als Referenz dient hierbei ein von Jeff Somers frei zur Verfügung gestelltes Softwarepaket, in dem verschiedene Methoden zur iterativen Netzoptimierung implementiert sind. Dazu wird mit verschiedenen Strategien versucht, die Kanten im Polygonnetz zu finden, deren Entfernung die geringste Änderung der Geometrie hervorruft. Die Kanten werden durch so genannte *Edge-Collapses* entfernt. Dabei im wird Mittelpunkt der zu entfernenden Kanten ein neuer Vertex erzeugt. Die Endvertices der Kante werden dann auf diesen Punkt verschoben (s. Abbildung 6.4).

Zur Detektion der zu entfernenden Kanten stehen vier verschiedene Methoden zur Verfügung. Die einfachste ist das sukzessive Löschen der kürzesten Kanten. Zwei weitere berechnen mit der von Garland und Heckbert vergestellten quadratischen Fehlermetrik [27] den Fehler, der durch die Entfernung einer Kante am Modell verursacht wird. Diese Methode ist in zwei Varianten implementiert: In der einfachen Variante wird die Kante, die den Vertex mit dem geringsten Fehler hat, entfernt. Die zweite Veriante summiert die Fehler über alle Vertices eines Faces

und entfernt das Dreieck mit dem kleinsten Wert. Bei beiden Methoden wird der beim Edge-Collapse neu erzeugte Vertex so positioniert, dass der resultierende Fehler minimiert wird. Die vierte Methode implementiert das von S. Melax vorgestellte Verfahren [28]. Sein Fehlerwert berücksichtigt sowohl die Länge einer Kante als auch den Unterschied der Normalen, also die Krümmung der Faces, die an eine Kante angrenzen.

Abbildung 6.10 und 6.11 zeigen die Ergebnisse der verschiedenen Varianten für zwei verschiedene Modelle nach Reduktion auf 50% der urspünglichen Vertices. Zum Vergleich wurde zum einen der Flurscan aus Abbildung 6.2 und zum anderen ein Scan des bekannteen Stanford-Bunnys [22] verwendet. Diese Modelle wurden ausgewählt, da sie unterschiedliche Arten von Flächen beinhalten. Der Flurscan repräsentiert eine typische Umgebung, wie sie von einem mobilen Roboter erfasst wurde und enthält viele planare Gebiete, während der Scan der Hasen-Skulptur beinahe ausschließlich aus gekrümmten Flächen besteht.

Auffällig ist, dass nach der Reduzierung bei den meisten Methoden aufgrund der tendenziell größeren Dreiecke ein deutlich sichtbarer Machband-Effekt auftritt. Eine Ausnahme bildet das Shortest-Edge-Verfahren. Dabei werden vornehmlich die Kanten entfernt, die die Länge der Marching-Cubes-Zellen haben. Später werden die Kanten an den Übergängen zwischen ebenen Flächen zusammengezogen, so dass die mit dieser Methode bearbeiteten Modelle etwas „verwaschen" wirken, da scharfe Konturen herausgemittelt werden. Durch die Entfernung der kürzesten Kanten werden jedoch weiterhin hauptsächlich relativ kleine Dreiecke erzeugt, wodurch der Machband-Effekt vermindert wird.

Bei den Modellen, die mit Garland und Heckberts oder Melax' Methode optimiert wurden, haben sich in planaren Gebieten Unebenheiten gebildet, die durch die Berechnung der Position des neuen Vertexes beim Edge-Collapse-Prozess entstanden sind. Da diese Verfahren versuchen, bei der Bestimmung der Vertexposition ihr Fehlermaß zu minimieren, können die neuen Positionen auch aus der ursprüglichen Ebene herausragen. Ein positiver Nebeneffekt dieser Tatsache ist, dass die im ursprünglichen Modell vorhandenen harten Übergänge zwischen verschiedenen Regionen abgemildert werden und so ein etwas natürlicherer Eindruck des gescanten Objektes entsteht (s. jeweils (d) bis (f) in Abbildung 6.11 und 6.10). In den Polygonnetzen, die durch das Triangulieren der Begrenzungspolygone der planaren Regionen vereinfacht wurden, wird die Geometrie des initialen Netzes nicht verändert. Deshalb ist dieser Effekt in diesen Rekonstruktionen nicht zu beobachten.

Die oben genannten Effekte treten in allen Datensätzen unabhängig von der Beschaffenheit der gescanten Objekte auf. Die iterativen Verfahren liefern für alle Arten von Datensätzen annehmbare Ergebnisse, da ihre Fehlermetriken verhindern, dass das Modell zu stark modifiziert wird. Auch die Neutriangulation der Ebenen in den Modellen liefert visuell akzeptable Ergebnisse, jedoch kann hier die Reduktionsrate nicht vorgegeben werden, weil sie von der Geometrie der Modelle abhängt.

6.4 Analyse der Reduktionsraten

Abbildung 6.5 zeigt die mit dem Neutriangulationsalgorithmus erreichten Reduktionsraten für verschiedene Datensätze. Benutzt wurden der bereits gezeigte AVZ-Datensatz, ein mit USAR-

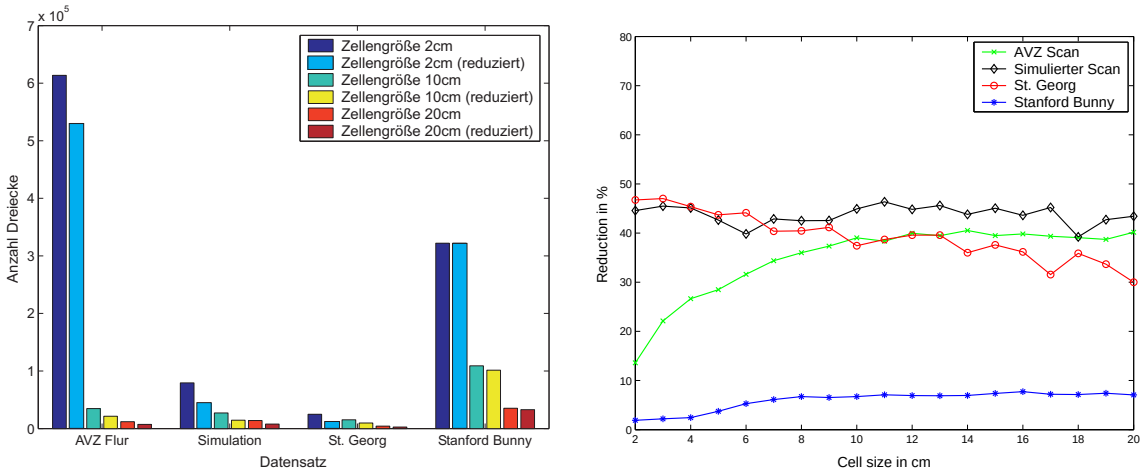


Abbildung 6.5: Reduktionsraten für verschiedene Datensätze. Links ist die Anzahl der Dreiecke in den generierten Modellen für 2, 10 und 20 cm Zellengröße vor und nach der Reduzierung zu sehen. Damit die Datensätze vergleichbar sind, wurden die Objekte jeweils auf die gleiche Größe skaliert. Die rechte Grafik zeigt die prozentualen Reduktionen der Modelle für verschiedene Gitterkonstanten.

SIM simulierter Datensatz (vgl. Kapitel 6.6.3), ein Architekturscan der St. Georg Kirche in Horn (vgl. Kapitel 6.6.2) sowie der Standord-Bunny Datensatz. Man kann deutlich erkennen, dass bei Datensätzen, die viele ebene Anteile haben, hohe Reduktionsraten zwischen 40% und 50% erreicht werden. Bei Scans von Objekten mit vielen Krümmungen ist die Reduktionsrate wesentlich geringer. Beim Stanford-Bunny-Datensatz konnten lediglich ca. 6% bis 8% der Dreiecke eingespart werden.

Der Datensatz, der mit dem dem Kurt3D Roboter auf dem Flur im 5. Stock des AVZ aufgenommen wurde, zeigt eine weitere Besonderheit. Ist die Marching-Cubes-Zellengröße nicht viel größer als die maximale Auflösung des Laserscanners, beginnen die Flächen im Modell zu „zerfallen" und können nicht weiter zusammengefasst werden. Daher fällt die Reduktionsrate in diesem Datensatz bei Zellengrößen unterhalb von 10 cm stark ab. Die anderen Datensätze, die eine höhere Punktdichte haben, sind von diesem Effekt nicht betroffen.

6.5 Vergleich der Laufzeiten

Abbildung 6.6(a) zeigt die Laufzeiten der verschiedenen Reduzierungsverfahren und die Laufzeiten der einzelnen Schritte bei der Modellerzeugung und Optimierung mit Hilfe des Triangulationsverfahrens. Alle Tests wurden mit einem Pentium-4-System mit 3.0 GHz und 1 GB Arbeitsspeicher durchgeführt. Damit die Laufzeiten der iterativen Algorithmen mit dem Triangulationsverfahren verglichen werden konnten, wurde für jede Zellgröße zunächst die erreichte Reduktionsrate ermittelt. Diese wurde dann als Zielwert für die iterativen Verfahren vorgegeben. Als Beispiel wird hier der AVZ-Flurscan gezeigt. Weitere Tests wurden auch mit anderen Datensätzen durchgeführt, die Ergebnisse waren aber im Wesentlichen identisch. Lediglich bei

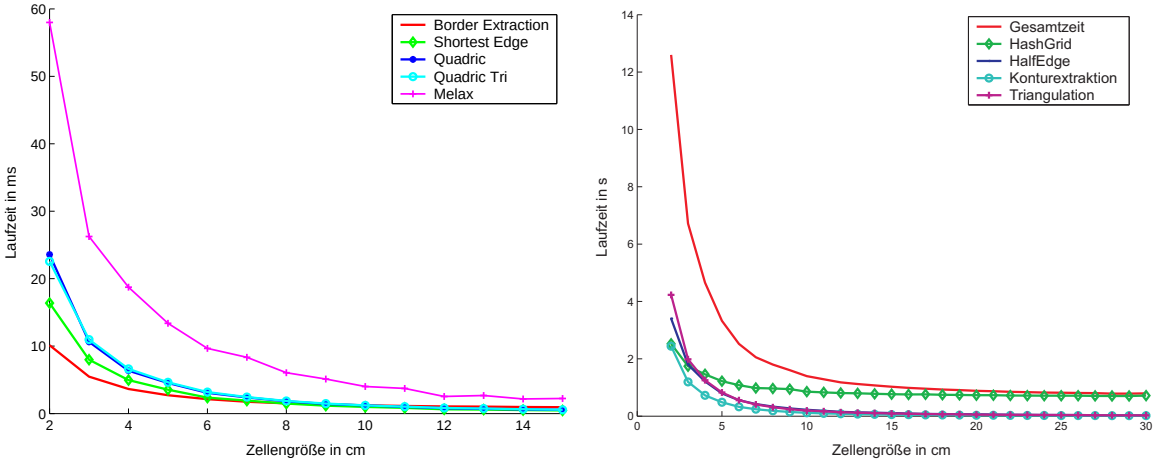


Abbildung 6.6: Laufzeitenvergleich der Algorithmen. Links wurden die Laufzeiten der verschiedenen Optimierungsverfahren am Beispiel des AVZ-Scans verglichen. Das rechte Bild zeigt die Laufzeiten der einzelnen Schritte bei der Modellerzeugung und -optimierung durch Triangulation der planaren Flächen im Modell.

Scans mit wenig ebenen Anteilen zeigte sich eine weitere leichte Leistungssteigerung beim Triangulationsverfahren, da der Schritt der Neutriangulierung weitestgehend wegfiel.

Man kann deutlich erkennen, dass der Triangulationsalgorithmus bei kleinen Zellgrößen deutlich schneller arbeitet als die iterativen Verfahren. Das liegt daran, dass die Berechnung der Fehlermaße bei diesen Algorithmen einen erheblichen Einfluss auf die Laufzeit hat. So ist das Shortest-Edge Verfahren nur wenig schneller als das Triangulationsverfahren. Die Laufzeiten der beiden Varianten mit der quadratischen Fehlermetrik sind beinahe gleich schnell, weil die Aussummierung über alle Vertices eines Faces nur wenig mehr Rechenzeit verlangt. Die ungünstigste Variante ist das Verfahren von Melax, denn dort müssen die komplexesten Berechnungen durchgeführt werden.

Insgesamt zeigt sich, dass sich die Laufzeiten bei größeren Zellen annähern, da mit der Zellgröße auch die Anzahl der erzeugten Faces abnimmt und deshalb weniger Fehlerwerte berechnet werden müssen. Das Triangulationsverfahren liegt in der Laufzeigt aber fast immer etwas unter den iterativen Verfahren, von denen die Melax-Methode immer am schlechtesten abschneidet.

In Abbildung 6.6(b) wird gezeigt, welchen Anteil die einzelnen Schritte des Triangulationsverfahrens an der Gesamtlaufzeit haben. Am meisten Zeit nimmt dabei in der Regel das Erstellen der HashGrid-Datenstruktur in Anspruch. Bei sehr kleinen Gitterkonstanten benötigen die Neutriangulation und die Erzeugung der HalfEdge-Datenstruktur mehr Rechenzeit. Die Triangulation wird aufwändiger, da durch die entstehende Fragmentierung sehr viele kleine Konturpolygone trianguliert werden müssen. Mit steigender Zellgröße entstehen mehr zusammenhängende Regionen. Die Polygone werden dann zwar größer, ihre Anzahl nimmt aber ab. Dadurch verringert sich die Zeit, die für die Triangulation aufgewendet werden muss, denn wenige große Polygone haben im Schnitt weniger Vertices als viele kleine. Die Zeit, die für die Generierung des HalfEdge-Netzes benötigt wird, sinkt mit der Zellengröße aufgrund der Tatsache, dass weniger

Abbildung 6.7: Modell einer byzantinischen Kirche. Anzahl der Messpunkte: ca. 2.200.000. Anzahl generierter Faces nach Reduktion: 185.429, Reduktionsfaktor 48%. Dauer des Modellerzeugungsprozesses: 10.7 s.

Faces in die Datenstruktur eingefügt werden müssen. Bei kleinen Gitterkonstanten nimmt die Erzeugung des HashGrids im Vergleich zu den anderen Schritten übermäßig viel Zeit ein, daher müsste dieser Schritt in Zukunft noch weiter optimiert werden.

6.6 Weitere Beispiele

Im Folgenden sollen noch einige weitere Beispieldatensätze vorgestellt werden, die teilweise auch von anderen Forschungsgruppen zur Verfügung gestellt wurden.

6.6.1 Scan einer byzantinischen Kirche auf Kreta

Abbildung 6.7 zeigt ein Modell der Kirche „Pana Kera" auf Kreta. Der zugrunde liegende Datensatz enthält insgesamt ca. 2.2 Mio Messpunkte und wurde von Jan Böhm vom Institut für

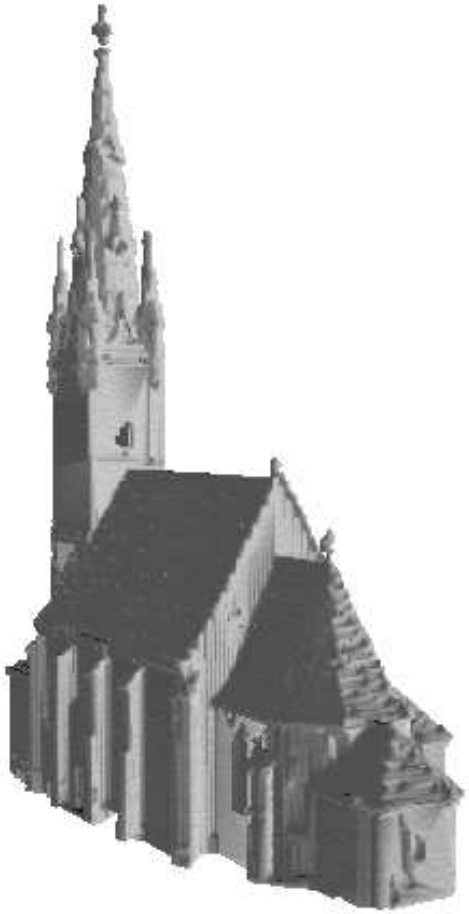


Abbildung 6.8: Scan der St.Georg-Kirche in Horn (Österreich). Anzahl Messpunkte: ca. 3.000.000. Anzahl generierter Faces nach Reduktion: 96.617, Reduktionsfaktor 47%. Dauer des Modellerzeugungsprozesses: 6.7 s.

Photometrie der Universität Stuttgart zur Verfügung gestellt [14]. Er wurde mit einem Leica HDS3000 Scanner aufgenommen, dessen Punktgenauigkeit unter 6 mm liegt. Die Modellerzeugung wurde mit einer Zellengröße von 8 mm durchgeführt [9]. Der Modellerzeugungsprozess hat insgesamt 10.7 s in Anspruch genommen und es wurden insgesamt 185.429 Dreiecke erzeugt. Der Reduktionsfaktor lag bei 48%.

Man kann im Bild deutlich die äußere Form der Kirche mit ihren verschiedenen Gebäudeteilen erkennen, die alle von Gewölben und Kuppeln bedeckt sind. Bei den Kuppeln sticht der vom Marching-Cubes-Algorithmus erzeugte Treppeneffekt besonders ins Auge. Trotz des hohen Anteils an gekrümmten Flächen wurde dennoch ein Rekduktionsfaktor von beinahe 50% erreicht. Das liegt daran, dass sowohl an den geraden Wandflächen als auch an den Stufen, die sich an den gekrümmten Flächen gebildet haben, viele Dreiecke eingespart werden konnten, da sich die Stufen über weite Teile des Modells gebildet haben (besonders an den Gewölben).

Das Modell gibt insgesamt einen guten Eindruck von der Gestalt der Kirche wieder. Leider ließen sich durch den Treppeneffekt die Texturen nicht fehlerfrei automatisch zuweisen, so dass das Modell ohne Texturen gerendert wurde.

6.6.2 Scan der Kirche St. Georg in Horn

Abbildung 6.8 zeigt einen Scan der St. Georg-Kirche in Horn in Östereich. Er wurde uns zusammen mit einer Software zum Registrieren von Architekturscans von der Firma RIEGL [44] zur Verfügung gestellt und enthält ca. 3 Mio. Messpunkte. Das gezeigte Modell besteht aus 96.617 Faces und wurde in 6.7 s erzeugt (Reduktionsrate: 47%).

Auch in diesem Modell zeigen sich ähnliche Effekte wie im Modell der Pana Kera-Kirche. Allerdings werden sie hier nicht hauptsächlich durch Gewölbe, sondern durch die schrägen Dachflächen verursacht. Wenn diese die Marching-Cubes-Zellen ungünstig schneiden, entstehen Stufen im Modell. Daher konnte auch dieses Modell nicht automatisch mit Texturen belegt werden und wurde nur einfarbig gerendert.

6.6.3 Umgebungsrekonstruktionen aus Roboterscans

Abbildung 6.9 zeigt Umgebungsscans, die in unserer Arbeitsgruppe angefertigt wurden. Die Datensätze für die Modelle in der oberen Zeile wurden mit dem 3D-Laserscanner unseres Kurt3D-Roboters aufgenommen. Die unteren beiden wurden in der Simulationssoftware USARSIM simuliert. Die Punktauflösung der Scans beträgt ca. 2 cm. Alle Modelle wurden mit einer Zellengröße von 10 cm erzeugt. Die erreichten Reduktionsraten betrugen zwischen 40% und 50%. Der Modellerzeugungsprozess benötigte bei allen Beispielen weniger als 2 s.

Da diese Scans im Inneren von Gebäuden aufgenommen wurden, treten hauptsächlich senkrecht aufeinandertreffende Flächen in den Scans auf. Deshalb konnte diesen Flächen in den Beispielen automatisch Texturen zugewiesen werden. Die Texturen wurden dabei durch Fotografieren der entsprechenden Flächen gewonnen.

Man kann deutlich erkennen, dass die Rekonstruktionen aus den simulierten Daten wesentlich realistischer wirken als die aus den realen Daten erzeugten Modelle. Dies liegt hauptsächlich daran, dass die in der Simulation gewonnenen Daten korrekt kalibriert sind, d.h. alle Flächen stehen wirklich senkrecht in der Szene. Außerdem ist kein Rauschen vorhanden. In den anderen Modellen treten die bekannten Treppen auf, die durch leicht schief gesehene Wände entstehen. Insgesamt kann durch die Zuweisung von Texturen im optimierten Modell ein sehr guter Eindruck der Szene, in der sich der Roboter befindet, gewonnen werden.

6.7 Zusammenfassung

Die durchgeführten Experimente haben gezeigt, dass das in dieser Arbeit vorgestellte Verfahren gut geeignet ist, mit Hilfe des Marching-Cubes-Algorithmus aus 3D Laserscannerdaten erstellte Polygonmodelle zu vereinfachen, sofern sie vornehmlich planare Anteile haben. In solchen Fällen

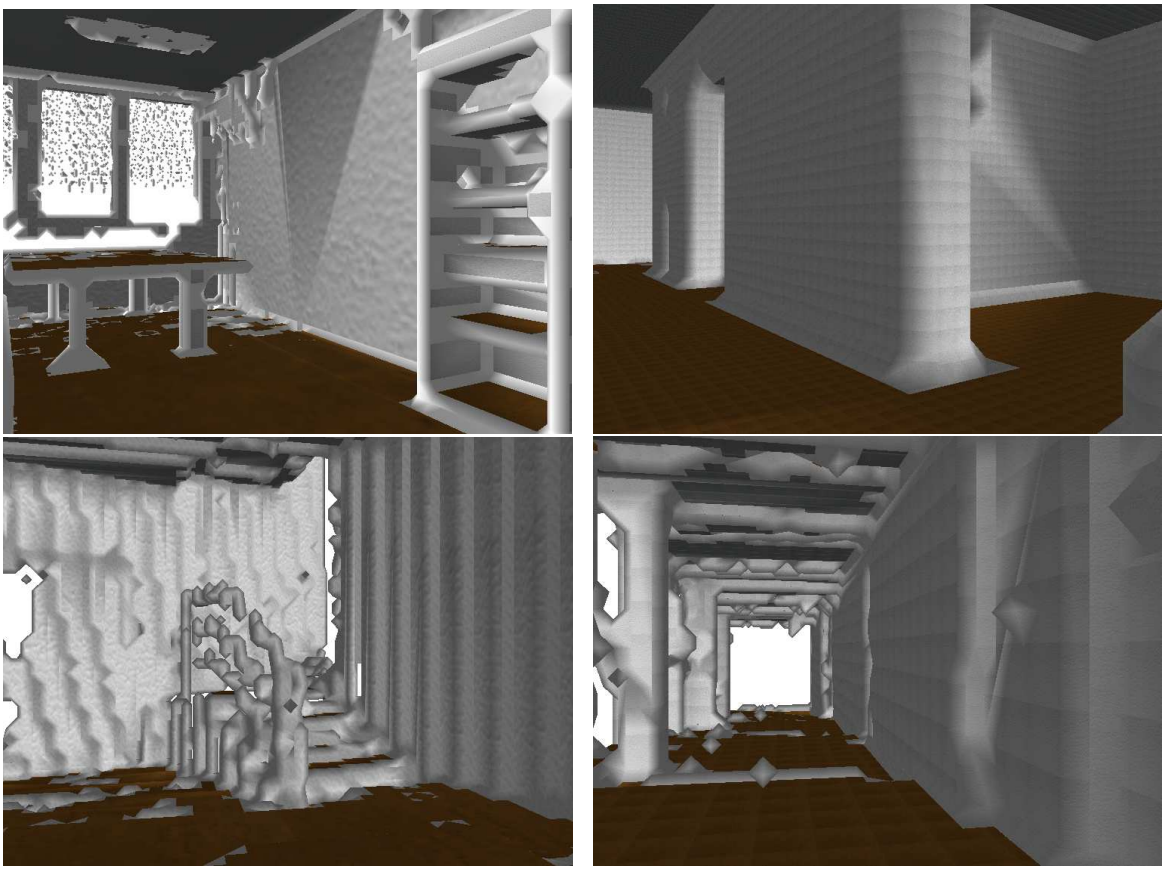
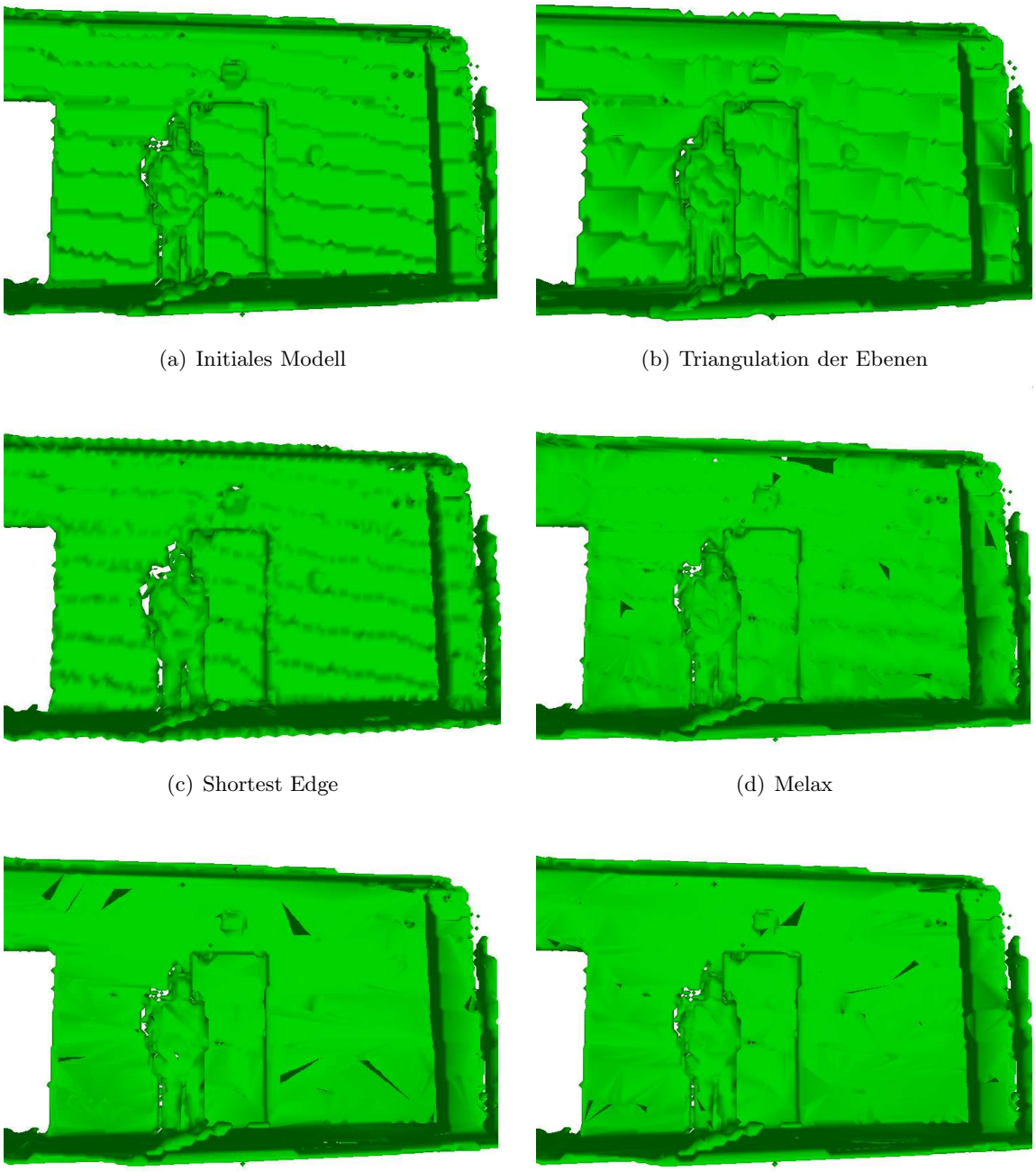


Abbildung 6.9: Umgebungsrekonstruktionen aus Roboterscans. Den oberen beiden Rekonstruktionen liegen in der USARSIM-Software simulierte Scans zugrunde. Die unteren beiden wurden aus Laserscans, die mit dem Kurt3D-Roboter im AVZ aufgenommen wurden, erstellt.

konnten zwischen 40% und 50% der ursprünglich im Modell enthaltenen Dreiecke eingespart werden, ohne dass die Geometrie des Ausgangsmodells geändert wurde. Der Reduktionsprozess lief dabei erheblich schneller ab als bei iterativen Verfahren. Aufgrund der besseren Laufzeit und der Tatsache, dass von autonomen Robotern befahrbare Gebiete in der Regel eben sind, ist das vorgestellte Verfahren gut für die Verwendung in der mobilen Robotik geeignet.

Der wesentliche Nachteil des Triangulationsalgorithmus ist die Tatsache, dass die erreichbaren Reduktionsraten ausschließlich von der Beschaffenheit der Ausgangsdaten abhängig sind. Bei Scans, die kaum planare Anteile haben, werden nur sehr geringe Reduktionsraten erzielt. Zur Vereinfachung von Modellen, die aus solchen Daten erstellt wurden, sind die iterativen Verfahren wesentlich besser geeignet, da die Anzahl der zu entfernenden Kanten vorgegeben werden kann.



(a) Initiales Modell

(b) Triangulation der Ebenen

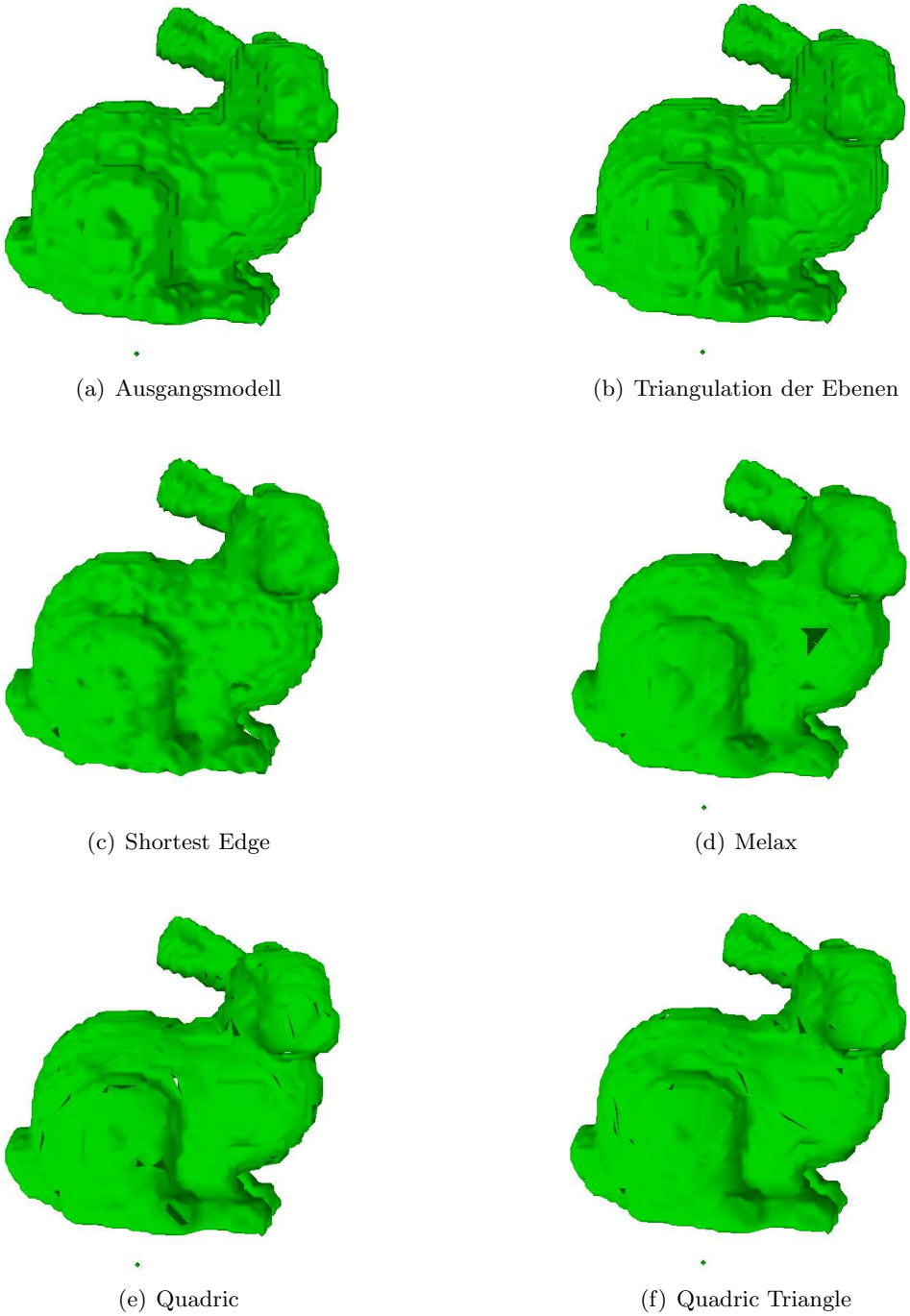
(c) Shortest Edge

(d) Melax

(e) Quadric

(f) Quadric Triangle

Abbildung 6.10: Ergebnisse verschiedener Reduktionsverfahren am Beispiel des AVZ-Flurscans



(a) Ausgangsmodell

(b) Triangulation der Ebenen

(c) Shortest Edge

(d) Melax

(e) Quadric

(f) Quadric Triangle

Abbildung 6.11: Ergebnisse verschiedener Reduktionsverfahren am Beispiel des Scanford-Bunny-Datensatzes

Kapitel 7

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein Verfahren zur Modellerzeugung und -optimierung aus 3D-Laserscannerdaten entwickelt. Die Modellerzeugung basiert auf dem bekannten Marching-Cubes-Algorithmus, der mit Hilfe der sogenannten HashGrid-Datenstruktur implementiert wurde. Diese Datenstruktur ermöglicht eine effiziente Überführung der erzeugten Dreiecksnetze in einen indizierten Vertex-Buffer und eine Halbkantendarstellung. Das zur Modelloptimierung entwickelte Verfahren geht von der Annahme aus, dass es sich bei den vorliegenden Datensätzen um Umgebungsscans handelt, die von mobilen Robotern aufgenommen wurden und daher viele ebene Flächen beeinhalten. Diese Annahme wird ausgenutzt, um die initial erzeugten Marching-Cubes-Modelle zu vereinfachen. Dazu werden die Konturen der ebenen Flächen im Ausgangsmodell detektiert und in eine Polygondarstellung überführt. Die so gefundenen Polygone werden in einem dritten Schritt mit Hilfe des OpenGL-Tesselators neu trianguliert, so dass wieder eine Dreiecksdarstellung vorliegt. Mit Hilfe dieses Verfahrens konnten in solchen Modellen Kompressionsraten von ca. 50% gegenüber den Ausgangsmodellen erreicht werden, ohne deren Geometrie zu verändern. Der Vergleich mit bekannten iterativen Optimierungsverfahren hat gezeigt, dass die Triangulationsmethode deutlich weniger Rechenzeit benötigt. Bei Modellen, die wenig planare Anteile hatten (z.B. Scans von Skulpturen), wurde kaum eine Reduzierung erreicht. Diese entsprechen allerdings auch nicht der Grundannahme des Verfahrens. Des Weiteren wurden die großen Flächen in den Modellen mit Hilfe eines einfachen Klassifizierungsverfahrens, das die Marching-Cubes-Grundmuster analysiert, in die Klassen „Wand", „Fußboden" und „Decke" eingeteilt. Diesen Klassen wurden Texturen zugewiesen, so dass sich ein realistisches Bild der gescanten Umgebung erstellen ließ.

Die relativ geringe Rechenzeit, die von den in dieser Arbeit vorgestellten Verfahren benötigt wird, um aus Laserscans effiziente polygonale Darstellungen der gescanten Objekte zu gewinnen, legt eine weitere Verwendung in der mobilen Robotik nahe. Scans, die mit Hilfe des Kurt3D-Roboters aufgenommen werden, können, je nach gewählter Auflösung, bereits in wenigen Sekunden ausgewertet werden. Diese Modelle könnten mit aus Kamerabildern des Roboters gewonnenen Texturen versehen werden, um eine detailgetreue Wiedergabe der Roboterumgebung zu erzielen. Neben der reinen Visualisierung der könnten in zukünftigen Anwendungen die Polygone der Konturen der ebenen Anteile in den Laserscans ausgewertet werden, um z.B. die Selbstlokalisierung eines autonomen Roboters zu verbessern oder Hindernisse bzw. befahrbare Bereiche zu erkennen.

Literaturverzeichnis

[1] *Geometric Modelling for Data Visualization*, chapter *Mesh Independent Surface Interpolation*. Springer, 2003.

[2] The Digital Michelangelo Project, `http://graphics.stanford.edu/projects/mich/`, 2004.

[3] Leica HDS3000 Laserscanner, `http://leica.loyola.com/products/hds/pdf/hds3000.pdf`, 2007.

[4] Riegl Laserscanner, `http://www.riegl.com/terrestrial_scanners`, 2007.

[5] Schmersal Laserscanner, `http://produkte.schmersal.de/585/538/group.html?lang=de`, 2007.

[6] SICK Laserscanner, 2007. `http://www.sick.de/de/products/categories/safety/espe/laserscanner/`.

[7] Z+F Laserscanner, `http://www.zf-laser.com/d_profilgebende.html`, 2007.

[8] B. G. Baumgart. Winged Edge Polyhedron Representation. Technical report, Stanford, CA, USA, 1972.

[9] J. Böhm and M. Pateraki. From Point Samples to Surfaces - On Meshing and Alternatives. In *International Archives on Photogrammetry and Remote Sensing IAPRS*, volume XXXVI, Com. V, 2006.

[10] B. Chazelle. Triangulating a Simple Polygon in Linear Time. *Foundations of Computer Science*, 1:220–230, 1990.

[11] D. Shreiner, M. Woo and Jackie Neider. *OpenGL Programming Guide. The Official Guide to Learning OpenGL, Versin 2 (OpenGL).* Addison-Wesley Longman, Amsterdam, 5th edition, 2005.

[12] H. Freeman. On the Encoding of Arbitrary Geometric Configurations. *IRE Trans. Electronic Computers*, EC-10:260–268, 1961.

[13] Michael Fötsch. Triangulating and Extruding Arbitrary Polygons. May 2001. `http://www.geocities.com/foetsch/extrude/extrude.htm`.

[14] Institut für Photometrie der Universität Stuttgart. Laser Splatting. `http://www.ifp.uni-stuttgart.de/publications/software/laser_splatting/i%ndex.html`.

[15] G. H. Meister. Polygons Have Ears. *American Mathematical Monthly*, 85:648–651, 1975.

[16] H. Edelsbrunner, E.P. Mücke. Three-Dimensional Alpha Shapes. *ACM Transactions on Graphics*, 13:43–72, Jan. 1994.

[17] H. ElGindy, H. Everett and G.T. Toussaint. Slicing an Aar using Prune-and-Search. *Pattern Recognition Letters*, pages 719–722, 1993.

[18] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald and W. Stuetzle. Mesh optimization. *SIGGRAPH*, pages 19–26, 1994.

[19] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald and Werner Stuetzle. Surface Reconstruction from Unorganized Points. *Computer Graphics*, 26(2):71–78, 1992.

[20] T. Hammersley. Perspective Corrected Texture Mapping. *GameDev.net*, 2007. `http://www.gamedev.net/reference/articles/article331.asp`.

[21] H. Hoppe. *Surface reconstruction from unorganized points*. PhD thesis, U. of Washington, Seattle, 1994.

[22] Stanford Computer Graphics Laboratory. The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`.

[23] D. Levien. The Approximation Power of Moving Least Squares. *Math. Comp. 67*, pages 1517–1531, 1998.

[24] Andreas Nüchter Kai Lingemann and Joachim Hertzberg. *Kurt3D – A Mobile Robot for 3D Mapping of Environments*. 2006. ELROB Technical Paper.

[25] W. E. Lorensen and H. E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In *Proceedings of ACM SIGGRAPH'87*, volume 21, pages 163–169, 1987.

[26] A. Adamson M. Alexa. Interpolatory Point Set Surfaces - Convexity and Hermite Date. *ACM Transactions on Graphics*, 2007.

[27] M. Garland and P. S. Heckbert. Surface Simplification Using Quadric Error Metrics. *Computer Graphics*, 31(Annual Conference Series):209–216, 1997.

[28] S. Melax. A Simple, Fast, and Effective Polygon Reduction Algorithm. *Game Developer Mag.*, pages 47–49, November 1998.

[29] T. Moller and E. Haines. *Real-Time Rendering*. AK Peters, Ltd., second edition, 2002.

[30] N. M. Amato, M. T. Goodrich and E. Ramos. Linear-Time Triangulation of a Simple Polygon Made Easier Via Randomization. *Discrete and Computational Geometry*, 26:245–265, 2001.

[31] A. Narkhede and D. Manocha. Fast Polygon Triangulation Based on Seidel's Algorithm. *Graphic Gems V*, 1995.

[32] A. Nealen. An As-Short-As-Possible Introduction to the Least Squares, Weighted Least Squares and Moving Least Squares Methods for Scattered Data Approximation and Interpolation. Technical report, Technische Universität Berlin, 2004. `http://www.nealen.com/projects/mls/asapmls.pdf`.

[33] The University of Utah. The Utah Teapot. `www.cs.utah.edu/school/whyteapot.htm`.

[34] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1998.

[35] P. Bourke. Polygonising a Scalar Field Using Tetrahedrons. Jun. 1997. `http://astronomy.swin.edu.au/~pbourke/modelling/polytetra/`.

[36] H. Pfister, M. Zwicker, J. van Baar, and M. H. Gross. Surfels: Surface elements as rendering primitives. In *Proc. of SIGGRAPH*, 2000.

[37] J. Pöpsel. *Computergrafik: Algorithmen und Implementierung*. Springer, 1994.

[38] R. Lotto , S. Williams and Purves D. Mach Bands as Empirically Derived Associations. In *Proceedings of the National Academy of Sciences*, 1999.

[39] S. Rusinkiewicz and M. Levoy". Qsplat: A multiresolution point rendering system for large meshes. In *Proc. of ACM SIGGRAPH*, 2000.

[40] R. Seidel. A Simple and Fast Incremental Randomized Algorithm for Computing Trapedozional Decompositions and for Triangulating Polygons. *Computational Geometry: Theory and Applications*, 1(1):51–64, 1991.

[41] D. Shreiner. *OpenGL Reference Manual. The Official Reference Document to OpenGL, Version 1.4*. Addison-Wesley Longman, Amsterdam, 4th edition, 2004.

[42] S. Suri. Polygon Triangulation. `http://www.cs.ucsb.edu/~suri/cs235/Triangulation.pdf`.

[43] H. Surmann, K. Lingemann, A. Nüchter, and J. Hertzberg. Fast acquiring and analysis of three dimensional laser range data. In *Proc. VMV*, 2001.

[44] RIEGL Laser Measurement Systems. `http://www.riegl.com`.

[45] G. M. Treece, R. W. Prager, and A. H. Gee. Regularised marching tetrahedra: Improved iso-surface extraction. *Computers and Graphics*, 23(4):583–598, 1999.

[46] T. Wiemann. *Berührungslose Vermessung Dreidimensionaler Objekte und deren Modellierung am Rechner*. Universität Osnabrück, 2005. Bachelor's Thesis.

[47] Wikipedia. Bump-mapping, 2007. `http://de.wikipedia.org/wiki/Bumpmapping`.

[48] Wikipedia. Displacement-mapping, 2007. http://en.wikipedia.org/wiki/Displacement_mapping.

[49] Wikipedia. Polygonnetz. 2007. http://de.wikipedia.org/wiki/Polygonnetz.

[50] Wikipedia. Texture mapping, 2007. http://en.wikipedia.org/wiki/Texture_mapping.

[51] X. Kong, H. Everett and G.T. Toussaint. The Graham Scan Triangulates Simple Polygons. *Pattern Recognition Letters*, pages 713–716, 1990.

[52] K. Zeppenfeld. *Lehrbuch der Grafikprogrammierung.* Spektrum Akad. Verl., 2004.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Osnabrück, Oktober 2007