University of Applied Sciences Wedel, Faculty of Media Information Science

Master-Thesis

Polygonisierung von impliziten Feldfunktionen

Henry G. Kleta
Dipl.-Ing.(FH), PgC

11. September 2002

Betreuer: Prof. Dr. Andreas Kolb
Fachhochschule Wedel
Feldstrasse 143
D-22880 Wedel

dedicated to the wumpus that was sacrificed for this thesis...

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Allgemein

Bei dem Prozess der Produktentwicklung kommt es heutzutage immer häufiger vor, dass bereits bevor auch nur ein Prototyp hergestellt wird, das Produkt als Modell auf einem Computer erstellt wird. Ein immer wichtiger werdendes Werkzeug hierfür ist die Virtual Reality (VR).
Bei der VR wird teils mit immensem Aufwand an Hard- und Software versucht, das zukünftige Produkt für den Benutzer[1] so realistisch wie möglich immersiv darzustellen. Das heisst, dass mit entsprechenden Mitteln versucht wird, das Produkt in dessen typischer Umgebung so darzustellen, dass es dem Benutzer erscheint, als tauche er in die dargestellte Szene ein.
Um einen hohen Grad an Immersion zu erreichen, wird meist auch mit Mitteln der Interaktion gearbeitet. Hiermit soll es dem Benutzer ermöglicht werden, sich "frei" in der dargestellten Szene zu bewegen, um sich z.B. nur die Bereiche, die ihn interessieren anzuschauen. Die Szene liegt also nicht bereits berechnet (gerendert) vor und wird nur angezeigt, sondern wird in Echtzeit gerendert und dargestellt.
Das Optimum an Immersion ist erreicht, wenn es dem Benutzer nicht mehr möglich ist, zwischen der Realität und der virtuellen Realität zu unterscheiden.

Abbildung 1.1.: Optimum an Immersion [Illiad]

Dieses Optimum ist in Abbildung 1.1 auf satirische Art und Weise dargestellt: Die Akteure

[1] Alle geschlechterspezifischen Bezeichnungen in diesem Text wurden so gewählt, dass sie den meiner Meinung nach üblichen neutralen Formen entsprechen. Sie gelten selbstverständlich gleichberechtigt für alle Lebensformen, die in der Lage sind diesen Text zu lesen.

fragen sich, wie viele Polygone zur Darstellung der Umgebung - die Ihrer eigenen Aussage nach besser aussieht als eine vergleichbare Szene mit einer sehr hohen Auflösung bei sehr hoher Farbtiefe - benötigt wurden, obwohl sich die Akteure in der freien Natur aufhalten.

Die Forderungen nach einer hohen Immersion und einer Darstellung in Echtzeit sind fast widersprüchlich, da auch heutige Computersysteme noch entsprechende Grenzen aufweisen. Eine dieser Grenzen ist die maximale Anzahl der darstellbaren Polygone pro Sekunde. Diese Anzahl ist für verschiedene übliche VR Computer-Systeme in Tabelle 1.1 dargestellt.

System[2]	max Pols/sek
Indigo2 (MaxImpact)[3]	1.800.000
O2 (R10k 250SC)[3]	394.000
Octane (R10k 250 SI+T)[3]	655.000
Onyx2 (IR2 single pipeline)[4]	13.100.000
Onyx2 (IR2 16 pipelines)[4]	210.000.000

Tabelle 1.1.: Maximale Anzahl darstellbarer Polygone pro Sekunde

Die angegebenen Werte sind als absolute Grenzwerte zu betrachten. Allein durch die Tatsache, dass in der VR meist stereoskopisch dargestellt wird, und dass für eine flüssige Darstellung eine Bildwiederholrate (Framerate) von mehr als 12 frames pro Sekunde benötigt wird, verringern sich die angegebenen Werte mindestens um den Faktor 24. Dies scheint gerade bei den durchaus üblichen leistungsstärkeren Onyx2 Systemen noch eine grosse Anzahl von darstellbaren Polygonen pro Sekunde zu sein, jedoch hängt die Framerate auch noch von zusätzlichen Faktoren wie Komplexität der Szene und Berechnungsaufwand ab (z.B. bei der für realistische Darstellungen unerlässlichen Schattierung).

Ein weiteres Problem ist, dass bei nahezu allen Modellen CAD-Daten zu Grunde liegen. Da die CAD-Daten auch für die Produktion benötigt werden, zeichnen sie sich durch eine sehr grosse Genauigkeit aus. Dieses führt zu sehr komplexen Polygon-Modellen. So ist es z.B. üblich, dass eine CAD-Software ein einzelnes Rad eines PKWs mit mehreren 100k Polygonen darstellt.
Wenn man in diesem Zusammenhang darzustellende Objekte in der Grössenordnung und Komplexität eines Verkehrsflugzeuges betrachtet, wird sehr schnell klar, dass die Polygon-Modelle für die Nutzung in der VR speziell bearbeitet oder aber speziell generiert werden müssen.

1.2. Problemstellung

Bei VR-Anwendungen ist - wie in Abschnitt 1.1 beschrieben - die Anzahl der darzustellenden Polygone aus Gründen der Performance von zentraler Bedeutung.
Um die Anzahl der Polygone zu verringern, bieten sich diverse Möglichkeiten an. Viele dieser Möglichkeiten wurden in der Diplomarbeit [John 01] dargestellt.

Ein zusätzlicher Aspekt (ebenfalls in [John 01] dargestellt) von grosser Wichtigkeit bei der Erstellung von Polygon-Modellen für VR-Anwendungen ist die Tatsache, dass bei vielen Polygon-Modellen Informationen darüber fehlen, wie benachbarte Flächen miteinander verbunden sind. Der Körper ist also aus mehreren einzelnen Flächen zusammengesetzt und somit nicht "wasserdicht" geschlossen. Durch diese fehlenden Informationen über die Verbindung

[2] Die fehlende Angabe des Prozessortyps bei der Indigo2 und den Onyx2-Systemen ist durch die Hardware-Struktur dieser Systeme zu erklären: die gesamte Grafikleistung entsteht im Grafik-Subsystem.
[3] Daten sind [FutureTech] entnommen.
[4] Daten sind [SGI] entnommen.

von einzelnen Modell-Teilen kann es bei dem Schritt der Polygon-Reduktion, der so genannten Simplifikation, dazu kommen, dass Risse entstehen.
Diese Risse sind bei vielen VR-Anwendungen sehr störend. Neben der Tatsache, dass das Hindurchblicken durch eigentlich geschlossene Körper für Benutzer irritierend und störend wirken kann, können solche Risse sogar bestimmte Anwendungen komplett unmöglich machen. So ist es z.B. für eine Raumklima-Simulation unerlässlich, dass alle in einem Körper simulierten Luftpartikel auch dort bleiben und nicht durch einen Riss entweichen können.

Kolb und John stellen in [Kolb 01] eine auf dem Ray-Stabbing von Nooruddin und Turk (siehe [Nooruddin 99]) basierende Methode vor, mit der aus einem initialen Polygon-Modell (über einen Zwischenschritt der Umwandlung in ein Volumen-Modell) ein in sich geschlossenes Polygon-Modell erzeugt werden kann, dass nur noch aus den äusseren sichtbaren Teilen des ursprünglichen Modells besteht.
Ein weiterer wesentlicher Vorteil hierbei ist, dass viele visuell unwichtige Details des Modells hierdurch wegfallen (z.B. ist der Motor bei einem PKW mit geschlossener Motorhaube für den visuellen Eindruck des PWKs unerheblich und kann somit weggelassen werden), womit das sich ergebende Modell weniger (teils sogar wesentlich weniger) Polygone aufweist als das initiale Modell.

Das Ziel dieser Arbeit ist die Generierung eines gleichmässigen, geschlossenen Polygon-Modells aus einem Volumen-Modell in Form einer impliziten Feldfunktion.
Für das Erreichen dieses Zieles wird die Darstellung eines Modells in Form impliziter Feldfunktionen untersucht und beschrieben. Ausserdem werden die notwendigen Grundlagen und Termini ebenso behandelt wie unterschiedliche Algorithmen die zum Ziel - einem gleichmässigen, geschlossenen Polygon-Modell - führen.

1.3. Überblick

Wie zuvor beschrieben, beschäftigt sich diese Arbeit mit der Generierung eines Polygon-Modells aus einem Volumen-Modell. Dazu werden die einzelnen Themen-Bereiche nacheinander in mehreren Kapiteln dargestellt.

Kapitel 2 analysiert die Grundlagen dieser Arbeit. Es werden die mathematischen Grundlagen, das bereits erwähnte Ray-Stabbing, sowie diverse Algorithmen zur Extraktion von Isoflächen beschrieben, analysiert und bewertet.

In Kapitel 3 wird die eigene Implementierung des Marching Triangles Algorithmus sowie der Umgang mit Sonderfällen erläutert. Desweiteren ist in diesem Kapitel das Benutzer-Handbuch des im Rahmen dieser Arbeit erstellten Programmes zu finden.

Das Kapitel 4 beinhaltet die Abschlussbetrachtung. Dabei wird Erreichtes und Unerreichtes ebenso dargestellt, wie auch Verbesserungsvorschläge als Hilfestellung für zukünftige auf dieser Arbeit basierenden Modifikationen und Erweiterungen.

Im Anhang befinden sich diverse Ergebnisse von polygonisierten Feldfunktionen, Vergleiche von vorgegebenen und eigenen Ergebnissen sowie das Programmierer-Handbuch.

Abschliessend werden im Literaturverzeichnis die für die Bearbeitung dieser Arbeit genutzten Quellen, sowie unter Weitere Quellen weitere, für die Gestaltung dieser Ausarbeitung genutzten Quellen aufgelistet.

2. Analyse

Dieses Kapitel befasst sich mit den Grundlagen dieser Arbeit.
Zuerst (Abschnitt 2.1) werden die mathematischen Grundlagen dieser Arbeit beschrieben. Anschliessend (Abschnitt 2.2) wird das Ray-Stabbing erläutert, gefolgt (Abschnitt 2.3) von einer Analyse der wichtigsten Algorithmen zur Extraktion von Isoflächen[1].

2.1. Mathematische Grundlagen [Bloomenthal 97]

In diesem Abschnitt werden die mathematischen Grundlagen dieser Arbeit beschrieben.
Zuerst (Abschnitte 2.1.1 und 2.1.2) werden die Grundlagen der impliziten und der parametrischen Flächen beschrieben, gefolgt (Abschnitt 2.1.3) von einem Vergleich der beiden Flächen-Typen. Abschliessend (Abschnitt 2.1.4) erfolgt eine Beschreibung der Grundlagen der impliziten Feldfunktionen.

2.1.1. Implizite Flächen

Implizite Flächen sind zweidimensionale geometrische Formen, die im dreidimensionalen Raum existieren. Dieses Konzept lässt sich am einfachsten durch eine zweidimensionale Analogie erklären: Ein Tropfen Öl auf feuchtem Asphalt wird mehrere, konzentrisch angeordnete, schillernde Farbringe erzeugen. Jeder einzelne Farbton wird eine geschlossene Kontur erzeugen. In Abbildung 2.1 wird diese Analogie am Beispiel des Farbtones Aquamarin verdeutlicht.

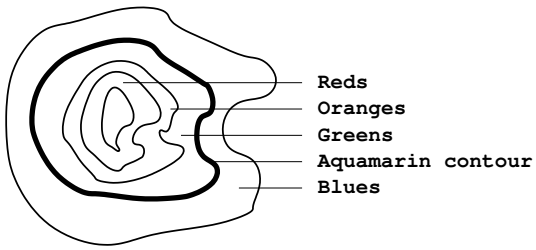


Abbildung 2.1.: Analogie: Öltropfen auf nassem Asphalt

Diese Analogie lässt sich auch auf den dreidimensionalen Raum erweitern: Man stelle sich einen gefüllten Wassertank vor, in den ein Tropfen Farbe eingebracht wird. Dieser Tropfen Farbe wird sich durch Diffusion nach und nach im gesamten Volumen verteilen. Zu einem beliebigen Zeitpunkt, bevor der Tropfen gleichmässig im gesamten Volumen verteilt ist, wird eine ganz bestimmte Farbnuance eine geschlossene Fläche in des Wassertanks beschreiben.

Eine implizite Fläche ist also eine Fläche mit einer bestimmten Eigenschaft. Im eben genannten Beispiel ist diese Eigenschaft die Farbe. Auf der impliziten Fläche ist der Wert der Eigenschaft immer konstant. Im Volumen hingegen variiert der Wert der Eigenschaft.

[1] Isoflächen werden im Abschnitt 2.1.1 beschrieben.

Die Eigenschaft lässt sich durch eine Funktion beschreiben. Üblicherweise wird die Funktion f genannt. Das Argument der Funktion f ist ein Punkt P innerhalb des betrachteten Volumens. Flächen mit der Eigenschaft $f(P) = const$ werden Isoflächen genannt. Isoflächen sind vor allem dann von Interesse, wenn unterschiedliche Isowerte ausgewertet werden sollen. Implizite Flächen hingegen werden per Definition durch die Eigenschaft $f(P) = 0$ beschrieben. Eine implizite Fläche ist also die Isofläche mit dem Wert 0.

2.1.2. Parametrische Flächen

Bei parametrischen Flächen werden die einzelnen Koordinaten durch Ausdrücke mit geometrischen Dimensionen des Objektes ersetzt. Es finden also folgende Formeln Anwendung: $x = f_x(s,t)$, $y = f_y(s,t)$ und $z = f_z(s,t)$. Siehe hierzu als Beispiel Abbildung 2.2.

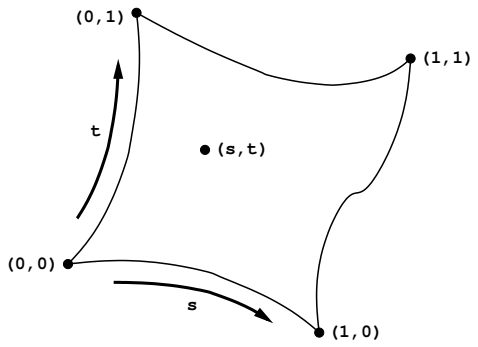


Abbildung 2.2.: Parametrische Fläche

Jeder 3D-Punkt auf der Fläche in Abbildung 2.2 kann durch ein geordnetes Paar (s,t) ausgedrückt werden. Diese Art der Abbildung nennt sich Parametrisierung, und findet vor allem bei der Texturierung, dem so genannten Texture-Mapping, Anwendung.

2.1.3. Vergleich parametrischer und impliziter Flächen

Dieser Arbeit liegen implizite Flächen zugrunde, da diese gegenüber parametrischen Flächen folgende Vorteile haben:

- Implizite Flächen beschreiben auch das Innere eines Objektes. Parametrische Flächen hingegen bestehen meist nur aus einzelnen, zusammengesetzten Flächenteilen.
  Klassifikationen von Punkten (innerhalb oder ausserhalb des Objektes) sind dadurch wesentlich leichter durchzuführen. Es muss lediglich das Vorzeichen des Funktionswertes $f(P)$ ausgewertet werden.
  Gerade die Erkennung von Kollisionen wird dadurch vereinfacht.
- Das Verschmelzen und Überblenden mehrerer Modell-Teile ist bei impliziten Flächen wesentlich einfacher.
- Implizite Darstellungen sind wesentlich übersichtlicher. Siehe hierzu Tabelle 2.1.

trigonometrisch:	$f(\alpha,\beta) = (cos\alpha \cdot cos\beta, sin\alpha, cos\alpha \cdot sin\beta), \alpha \in [0,\pi], \beta \in [0,2\pi]$
rational:	$x = 4st, y = 2t(1-s^2), z = (1-t^2)(1+s^2), w = (1+s^2)(1+t^2)$
implizit:	$f(x,y,z) = x^2 + y^2 + z^2 - 1$

Tabelle 2.1.: Verschiedene Darstellungsformen einer Kugel

2.1.4. Implizite Feldfunktionen

Der ein dreidimensionales Modell umgebende Raum wird durch ein Skalarfeld repräsentiert. Für jeden Punkt $P(x, y, z)$ in diesem Feld lässt sich der dazugehörige Funktionswert $f(P) = f(x, y, z)$ berechnen. Wegen der Ähnlichkeit mit physikalischen Feldern werden diese das Skalarfeld beschreibende Funktionen Feldfunktionen genannt.
Ein Funktionswert $f(P)$ kann z.B. über das Vorzeichen darüber Auskunft geben, ob sich der Punkt P innerhalb oder ausserhalb des Modells befindet.

Gibt es eine Menge von Punkten P, die alle den Funktionswert $f(P) = 0$ haben (sich also auf der Oberfläche des Modells befinden), wurde eine das Modell beschreibende Feldfunktion gefunden.

Dabei kann die Funktion f jede Art von mathematischen Ausdrücken beinhalten.

- Ein Ausdruck der ausschliesslich polynomiale Anteile hat, wird algebraisch genannt. Ein Beispiel hierfür ist die implizite Darstellung der Kugel in Tabelle 2.1.
- Hat der Ausdruck auch nicht polynomiale Anteile (z.B. trigonometrische, exponentielle, logarithmische oder hyperbolische) wird er transzendental genannt.
- Ausserdem kann die Funktion f durch einen Algorithmus beschrieben werden. In diesem Falle können Eigenschaften nur durch numerische Auswertung gewonnen werden. Ein Beispiel hierfür ist das Ray-Stabbing von Kolb und John (siehe Abschnitt 2.2.2).

2.1.5. Zusammengesetzte Feldfunktionen

Um sehr komplexe Feldfunktionen umsetzen zu können, besteht die Möglichkeit eine Feldfunktion aus mehreren einzelnen zusammen zu setzen. Zwischen den einzelnen Feldfunktionen wird übergeblendet. Dabei gibt es zwei verschiedene Verfahren:

- Das "harte" Überblenden. Dabei wird abhängig von der Position (x, y, z) des Punktes P immer nur eine Feldfunktion ausgewertet.
- Das "weiche", einer Funktion entsprechende Überblenden. Dabei werden alle Feldfunktionen f_i ausgewertet. Der Funktionswert $f(P)$ wird entsprechend einer Überblendfunktion aus den einzelnen Funktionswerten $f_i(P)$ berechnet.

Das "harte" Überblenden stellt die wesentlich einfachere Möglichkeit dar, um komplexe Feldfunktionen umzusetzen. Dabei wird entsprechend der Position im 3D-Raum die dort gültige Feldfunktion ausgewertet. Siehe als Beispiel Abbildung 2.3. Diese Abbildung stellt einen Schnitt durch die im Rahmen dieser Arbeit umgesetzte Hot-Dog-Feldfunktion[2] dar.

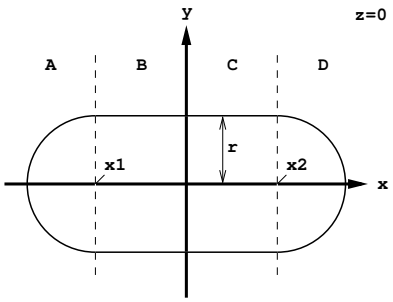


Abbildung 2.3.: Beispiel einer zusammengesetzten Feldfunktion

[2]Die Namen der Feldfunktionen in dieser Arbeit wurden entsprechend der optischen Erscheinung ihrer polygonisierten Darstellung gewählt.

Die Hot-Dog-Feldfunktion setzt sich aus drei verschiedenen Feldfunktionen zusammen:

- Bereich A: $f(P) = (x - x_1)^2 + y^2 + z^2 - r$
  Entspricht einer Kugel mit Radius r um den Mittelpunkt $(x_1, 0, 0)$.
- Bereiche B und C: $f(P) = y^2 + z^2 - r$
  Entspricht einem Zylinder mit Radius r um die X-Achse mit dieser als Mittellinie.
- Bereich D: $f(P) = (x - x_2)^2 + y^2 + z^2 - r$
  Entspricht einer Kugel mit Radius r um den Mittelpunkt $(x_2, 0, 0)$.

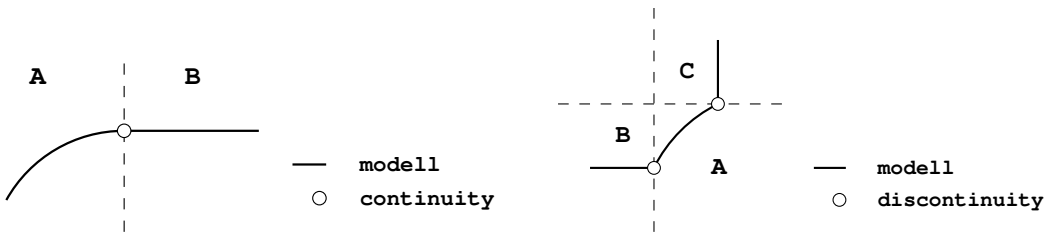
Bei folgenden im Rahmen dieser Arbeit umgesetzten Feldfunktionen handelt es sich ebenfalls um Feldfunktionen, bei denen "hart" zwischen mehreren Feldfunktionen geblendet wurde:

- Boje-Feldfunktion (siehe Abbildung A.1(e)):
  Sie besteht aus zwei Kegel-Feldfunktionen und einer Zylinder-Feldfunktion.
- Atom-Feldfunktion (siehe Abbildung A.1(f)):
  Sie besteht aus sieben Kugel-Feldfunktionen und sechs Zylinder-Feldfunktionen.

Der Vorteil der einfachen Nutzung des harten Überblendens wird oft durch den Nachteil der eventuell vorkommenden Unstetigkeiten an den Schnittstellen der einzelnen Feldfunktionen relativiert.

Bei der Hot-Dog-Feldfunktion konnte ein stetiger Übergang zwischen den einzelnen Feldfunktionen geschaffen werden. Abbildung 2.4(a) zeigt einen Ausschnitt der Hot-Dog-Feldfunktion. In diesem Ausschnitt ist ein Übergang von der Zylinder-Feldfunktion auf eine der Kugel-Feldfunktionen dargestellt. Da die Radien von Kugel und Zylinder gleich sind, der Mittelpunkt der Kugel sich auf der Mittellinie des Zylinders befindet und da genau im Mittelpunkt der Kugel (nur eine Halbkugel wird verwendet) geblendet wird, tritt keine Unstetigkeit auf.

Bei der Atom-Feldfunktion hingegen konnten die Unstetigkeiten nur im äusseren Bereich verhindert werden, da die einzelnen Arme der Hot-Dog-Feldfunktion entsprechen. Abbildung 2.4(b) zeigt einen Ausschnitt des Zentrums der Atom-Feldfunktion. Dort treffen sechs Zylinder-Feldfunktionen aufeinander. Um zu verhindern, dass durch dieses Auftreffen senkrechte Sprünge in der Feldfunktion auftreten, wurde zum Abschwächen des Überganges eine Kugel-Feldfunktion eingeführt. Die auftretenden Unstetigkeiten konnten jedoch nicht verhindert werden.

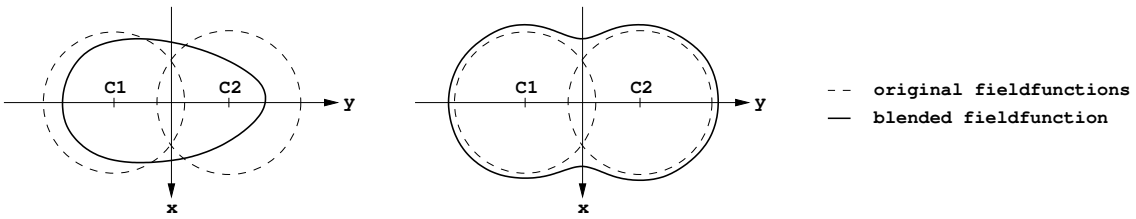


(a) stetiger Übergang (b) unstetiger Übergang

Abbildung 2.4.: Stetigkeit und Unstetigkeit beim Blenden

Um Unstetigkeiten zu verhindern, muss das zweite, "weiche" Verfahren zum Überblenden zwischen einzelnen Feldfunktionen verwendet werden. Dabei wird für jede einzelne Feldfunktion der Funktionswert für den Punkt P bestimmt. Der endgültige Funktionswert $f(P)$ wird dann entsprechend einer Überblendfunktion aus den einzelnen Funktionswerten ermittelt. Dabei gibt es zwei verschiedene Verfahren:

- Konstanter Einfluss aller Feldfunktionen.
  Die einzelnen Funktionswerte $f_i(P)$ werden mit einem Faktor g_i multipliziert und zum endgültigen Funktionswert $f(P) = \sum f_i(P)$ aufaddiert.
  Ein Beispiel hierfür ist die Ei-Feldfunktion (siehe Abbildung 2.5(a) und A.1(h)), die aus zwei Kugel-Feldfunktionen mit gleichen Radien r aber unterschiedlichen Zentren C_i besteht. Die einzelnen Faktoren g_i sind in diesem Falle 0.4 und 0.6.
- Abstandsabhängiger Einfluss aller Feldfunktionen.
  Entsprechend des Abstandes vom Punkt P zum Zentrum C_i der einzelnen Feldfunktion wird der Funktionswert $f_i(P)$ bestimmt. Anschliessend werden die einzelnen Funktionswerte $f_i(P)$ zu dem endgültigen Funktionswert $f(P) = \sum f_i(P)$ aufaddiert.
  Ein Beispiel hierfür ist die Erdnuss-Feldfunktion (siehe Abbildung 2.5(b) und A.1(i)), die aus zwei Kugel-Feldfunktionen mit gleichen Radii r aber unterschiedlichen Zentren C_i besteht. Die einzelnen Faktoren g_i sind in diesem Falle $\frac{1}{|P-C_i|}$.



(a) Ei-Feldfunktion (b) Erdnuss-Feldfunktion

Abbildung 2.5.: Beispiele von geblendeten Feldfunktionen

Beim Überblenden mehrerer Feldfunktionen können verschiedenste Überblendfunktionen zum Einsatz kommen. Üblich sind einfache, quadratische sowie exponentielle Wärmeleitfunktionen.

Ein genereller Nachteil bei Verfahren dieser Art ist, dass jede einzelne Feldfunktion f_i immer Einfluss auf den endgültigen Funktionswert $f(P)$ hat. Dieses kann bei sehr komplexen Modellen zu unerwünschten Effekten führen. So haben die Gebrüder Wyvill in [Wyvill 89] unerwünschte Überblend-Effekte bei mensch-ähnlichen Modellen beschrieben. Als Ziel ihrer Veröffentlichung geben die Gebrüder Wyvill an, dass in einem bestimmten Abstand d vom Zentrum C_i einer einzelnen Feldfunktion f_i der Einfluss dieser Feldfunktion gleich Null wird[3].

Durch Kombination beider vorgestellten Überblendverfahren lässt sich ein Ansatz erzeugen, der sowohl Unstetigkeiten beim "harten" Überblenden, als auch den Einfluss jeder einzelnen Feldfunktion f_i an jedem Punkt P des Volumens verhindert:

$$f_i(P) = \begin{cases} 0 & \text{wenn } P - C_i \geq d \\ f_i(P) & \text{sonst} \end{cases}$$

Da der Fokus dieser Arbeit hauptsächlich auf die Polygonisierung von Feldfunktionen gerichtet ist, und nicht auf die Darstellung aller denkbaren Feldfunktionen, wurde der zuletzt beschriebene Ansatz nicht umgesetzt.

[3]Wie dieses erreicht werden soll ist leider nicht genau ersichtlich.

2.2. Ray-Stabbing

In diesem Abschnitt wird das Ray-Stabbing beschrieben. Zuerst wird das Original von Nooruddin und Turk beschrieben. Anschliessend folgt die Beschreibung der Modifikationen von Kolb und John.

2.2.1. Das Original [Nooruddin 99]

Die Grundidee beim Ray-Stabbing ist es, ein vollständig geschlossenes Polygon-Modell zu erzeugen. Dieses wird erreicht, indem ein initiales Polygon-Modell (meist durch Export aus einer CAD Software gewonnen) in ein Volumen-Modell gewandelt wird. Aus diesem Volumen-Modell wird dann mit Methoden der Extraktion von Isoflächen das neue, nun vollständig geschlossene Polygon-Modell erzeugt. Dieser Workflow ist in Abbildung 2.6 dargestellt.

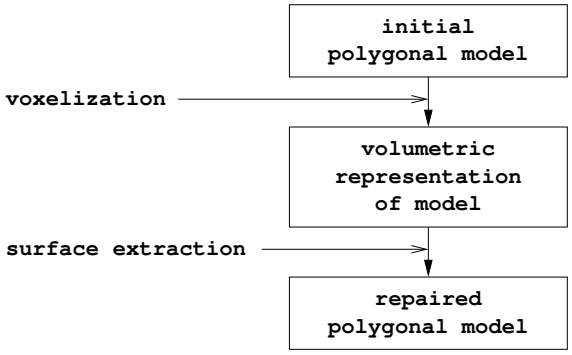


Abbildung 2.6.: Ray-Stabbing: Prinzipdarstellung

Die Umwandlung eines Polygon-Modells in ein Volumen-Modell mit der Auflösung $N \cdot N \cdot N$ umfasst folgende Schritte:

1. Es werden parallele, äquidistante Strahlen (r_i mit $i \in \mathbb{N} : i = [1..N]$) durch das die gegebene Geometrie G umgebende Bounding-Volume gesendet.
   Die Strahlen werden nacheinander aus bis zu 13 verschiedenen Richtungen gesendet. Hierdurch erhält man mehrere Strahlenmengen.
   Bei normalen Modellen werden nur Strahlen aus den drei Richtungen gesendet, die durch die Hauptachsen des Modells beschrieben werden. Bei problematischen Modellen[4] werden zusätzlich Strahlen aus den zehn durch die Oberflächennormalen eines Ikosaeders beschriebenen Richtungen gesendet.
2. Ein Strahl r wird als gültig klassifiziert, wenn er eine gerade Anzahl von Schnittpunkten mit G aufweist. Andernfalls wird er als ungültig klassifiziert.
3. Ein Voxel P entlang eines Strahls r (die Zentren der Voxel liegen auf den Strahlen) wird als intern klassifiziert, wenn r gültig ist und das Voxel P sich zwischen dem ersten und dem letzten Schnittpunkt von r mit G befindet.
   Andernfalls wird das Voxel P als extern klassifiziert.
4. Ein Voxel P wird als endgültig extern klassifiziert, wenn es bei mindestens einem Strahl als extern klassifiziert wurde.

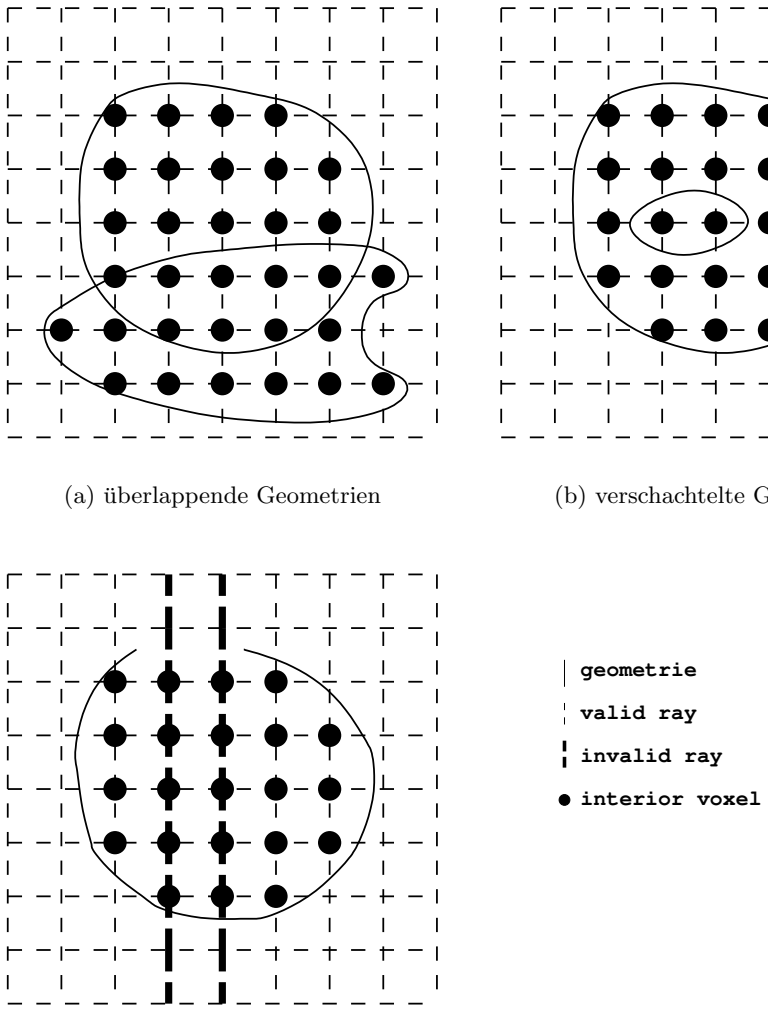
[4]Leider geben Nooruddin und Turk keine Auskunft darüber, was sie unter "troublesome models" verstehen.

5. Den Voxeln P wird entsprechend ihrer Klassifikation ein Wert zugewiesen:

$$f(P) \in \mathbb{R} = \begin{cases} 0 & \text{bei externen Voxeln} \\ 1 & \text{bei internen Voxeln} \\ [0,1] & \text{bei Voxeln nahe der Oberfläche} \end{cases}$$

6. Den Voxeln im Ergebnisraum wird mit Hilfe des "majority voting" aus den unterschiedlichen Werten der verschiedenen Strahlenmengen der endgültige Wert zugewiesen.

Abbildung 2.7 zeigt anhand von drei 2D-Beispielen die Funktionsweise des Ray-Stabbings, wobei die Zentren der als intern klassifizierten Voxel hervorgehoben dargestellt sind. Die Strahlen wurden hierbei aus den beiden Richtungen der Hauptachsen gesendet.



(a) überlappende Geometrien

(b) verschachtelte Geometrien

(c) nicht geschlossene Geometrie

Abbildung 2.7.: Ray-Stabbing: Verschiedene Geometrien

Nachdem das Volumen-Modell wie zuvor beschrieben erzeugt wurde, kann mit Methoden der Extraktion von Isoflächen (siehe Abschnitt 2.3) eine Isofläche als Polygon-Modell erzeugt werden. Dabei werden die Voxel als Eckpunkte von Zellen betrachtet.

2.2.2. Modifikationen von Kolb und John [Kolb 01]

Die wichtigsten Unterschiede der Implementation von Kolb und John zum Original Nooruddins und Turks sind folgende:

1. Die Strahlen werden aus sieben verschiedenen Richtungen (den drei Hauptachsen und den vier Raumdiagonalen) gesendet.
2. Den auf den Schnittpunkten der Strahlen liegenden Eck-Punkten P der Zellen[5] wird ein gerichteter Wert (der euklidische Abstand[6] zur Modell-Oberfläche) zugewiesen:

$$f(P) \in \mathbb{R} = \begin{cases} > 0 & \text{bei externen Punkten} \\ < 0 & \text{bei internen Punkten} \end{cases}$$

   Das Modell wird also durch die Isofläche $f(P) = 0$ beschrieben.
3. Den Eck-Punkten der Zellen des Ergebnisraumes wird nach folgendem Schema ein Wert aus den für den jeweiligen Punkt P_i ermittelten Werten (pro Strahl ein Wert) zugewiesen:
   a) Initialisierung des Punktes mit $-\infty$ (Wert der fernen Clipping Ebene).
   b) Wenn keiner der verfügbaren Werte positiv ist (also der Punkt bei allen Strahlen als intern klassifiziert wurde), wird der grösste Wert genommen.
   c) Wenn mindestens einer der sieben Werte positiv ist (der Punkt also als endgültig extern klassifiziert wurde), wird der kleinste positive Wert als Ergebnis genommen, auch wenn es einen negativen Wert gibt, der betragsmässig kleiner ist.

   Durch dieses Vorgehen wird garantiert, dass den Punkten entsprechend ihrer Klassifikation der geringste gemessene Abstand zur Modell-Oberfläche zugewiesen wird,

Obwohl das Ray-Stabbing durch den Abstand zweier benachbarter Strahlen r_i und r_{i+1} prinzipiell diskret ist, erreichen Kolb und John mit ihrer Methode durch bilineare Interpolation der entsprechenden, benachbarten Werte (z.B. wenn die Schnittpunkte der verschiedenen Strahlen nicht exakt mit Punkt P_i zusammen fallen) eine diskrete, dreidimensionale und vorzeichenbehaftete Feldfunktion.
Daher lässt sich das Modell implizit als Feldfunktion $f(P) = 0$ ansehen.

Da sich diese Arbeit mit der Polygonisierung von impliziten Feldfunktionen beschäftigt, und nicht mit der Umsetzung aller denkbaren Feldfunktionen, wurde das Ray-Stabbing lediglich analysiert und beschrieben, jedoch nicht implementiert.

[5] Kolb und John erwähnen zwar Voxel, jedoch wird aus dem Kontext klar, dass stattdessen Zellen gemeint sind.

[6] Kolb und John sprechen zwar vom euklidischem Abstand, jedoch wird bei Ihrer Implementierung aus praktikablen Gründen der geringste Abstand in Richtung eines Strahles verwendet, was nicht dem euklidischen Abstand entspricht, diesem aber sehr nahe kommt.

2.3. Algorithmen zur Extraktion von Isoflächen

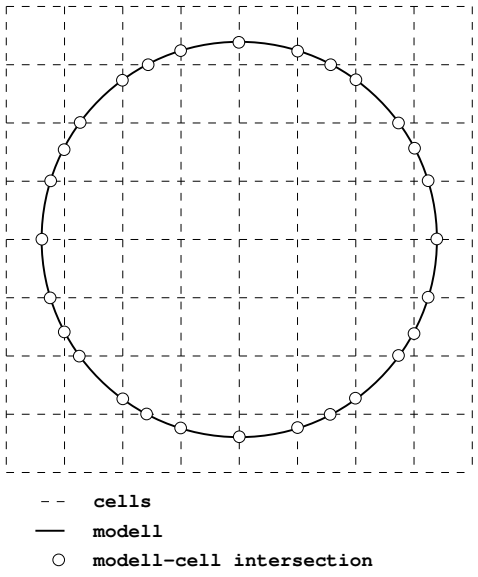
Für die Extraktion von Isoflächen (also die Polygonisierung von impliziten Feldfunktionen) gibt es eine Vielzahl von Algorithmen, die hier nicht alle dargestellt werden können.

Es werden lediglich die wichtigsten Vertreter der verschiedenen Ansätze sowie einige wichtige Optimierungen der Grund-Algorithmen analysiert. Die Abschnitte 2.3.1 - 2.3.4 behandeln volumenbasierte Algorithmen. Abschnitte 2.3.5 und 2.3.6 behandeln dann den flächenbasierten Marching Triangles Algorithmus und dessen Modifikationen.

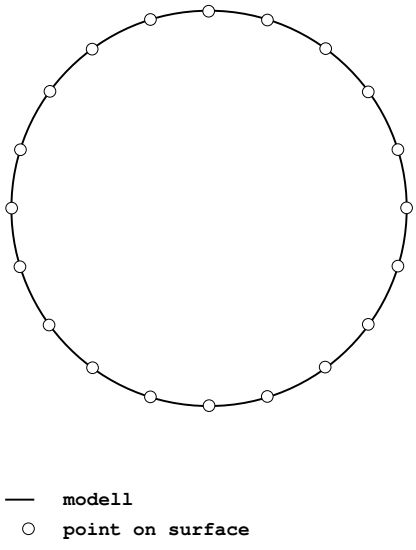
Bei der volumenbasierten Polygonisierung wird das das Modell umgebende Volumen in Zellen aufgeteilt. Zwischen den Schnittpunkten der Zellen-Kanten mit dem Modell werden Polygone erzeugt und das Modell somit approximiert.
Der Vorteil hierbei ist, dass das zu verarbeitende Volumen algorithmisch relativ einfach abgearbeitet werden kann.

Bei der flächenbasierten Polygonisierung wird ausgehend von einem initialen Polygon auf der Modell-Oberfläche (d.h., dass alle Punkte des Polygons sich auf der Modell-Oberfläche befinden) ein weiteres Polygon auf derselben generiert und so weiter.
Der Vorteil hierbei ist, dass die erzeugten Polygone wesentlich gleichmässiger sind, und dass nicht das gesamte das Modell umgebende Volumen abgearbeitet werden muss.

Bei den 2D-Beispielen in Abbildung 2.8 zeigt sich bereits, dass die Polygone bei der flächenbasierten Polygonisierung gleichmässiger sind. Ausserdem werden anstelle von 28 Polygonen bei der volumenbasierten Polygonisierung nur 20 Polygone bei der flächenbasierten Polygonisierung generiert, also ca. 30% weniger, womit 30% Rechenleistung eingespart werden.



(a) volumenbasierte Polygonisierung



(b) flächenbasierte Polygonisierung

Abbildung 2.8.: Volumen- und flächenbasierte Polygonisierung

2.3.1. Marching Cubes [Lorensen 87]

Der Marching Cubes Algorithmus ist wohl der bekannteste volumenbasierte Algorithmus zur Visualisierung von Volumendaten.

Das zu visualisierende Volumen wird in gleichmässige Datenwürfel (Zellen) zerlegt. Nun "wandert" ein Würfel (bzw. ein Quader) iterativ von einer Zelle zur nächsten, bis alle Zellen bearbeitet wurden. Siehe hierzu Abbildung 2.9.

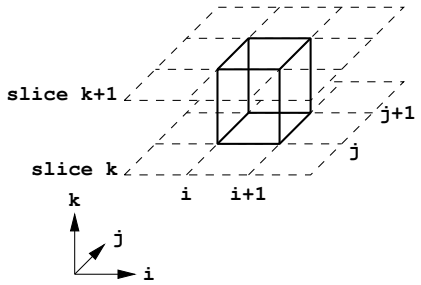


Abbildung 2.9.: Marching Cubes: Prinzipdarstellung

Beim Wandern des Würfels durch das Volumen wird für jede Zelle festgestellt, ob sich die gesuchte Isofläche mit den Kanten des Würfels schneidet. Dieses wird festgestellt, indem die Eckpunkte des Würfels als "innerhalb" bzw. "ausserhalb" der Isofläche (also $f(P) \leq c$ bzw $f(P) > c$) klassifiziert werden. Werden die Eckpunkte des Würfels unterschiedlich klassifiziert, schneidet der Würfel die gesuchte Isofläche. Der genaue Schnittpunkt der Isofläche mit den Kanten des Würfels wird durch Interpolation der Funktionswerte der Eckpunkte der jeweiligen Kante ermittelt. Zwischen diesen Schnittpunkten werden Dreiecke generiert und die gesuchte Isofläche somit stückweise approximiert.

Da der Würfel 8 Ecken hat, ergeben sich somit theoretisch 256 mögliche Klassifikationen eines einzelnen Würfels. Durch Ausnutzung von Symmetrien (so ist es z.B. unerheblich, ob sich alle Punkte innerhalb oder ausserhalb befindet, da das Ergebnis bis auf eine einfache Negation identisch ist) ergeben sich nur noch 15 zu betrachtende Fälle. Diese werden in Abbildung 2.10 dargestellt. Trotz der ausnutzbaren Symmetrien kann es bei einzelnen Fällen (3, 4, 6, 7, 10, 12, und 13 - siehe als Beispiel Abbildung 2.11 sowie [Abramowski 91, Abbildung 10.4] für eine vollständige Darstellung) mehrere mögliche Polygonisierungen geben. Diese möglichen Mehrdeutigkeiten müssen durch Betrachtung der benachbarten Zellen und deren Polygonisierungen gelöst werden.

Der Marching Cubes Algorithmus ist ein relativ alter Algorithmus. Dadurch kann man mit Sicherheit eine passende Implementation finden, die direkt einsetzbar ist. Allerdings birgt er den Nachteil, dass das gesamte Volumen schrittweise bearbeitet wird. Somit werden auch Zellen vom Typ 0 bearbeitet. Gerade bei einer schlechten Wahl des Bounding-Volumes (siehe Abbildung 2.12(a)) bzw. bei einem schwierigen Modell (siehe Abbildung 2.12(b)) werden so wesentlich mehr Zellen bearbeitet als eigentlich notwendig.

Desweiteren werden beim Marching Cubes Algorithmus Polygone verschiedenster Grösse erzeugt, was zu nicht effektiv darstellbaren Polygon-Modellen führen kann. Die unterschiedlichen Polygon-Grössen sind durch die unterschiedlichen Abstände der Schnittpunkte der Isofläche mit dem Würfel zu erklären und sind der Grund dafür, dass die meisten durch den Marching Cubes Algorithmus erzeugten Polygon-Modelle mit entsprechendem Aufwand simplifiziert werden müssen, bevor sie verwendet werden können.
Ein weiterer Nachteil ist, dass die Schnittpunkte nur durch Interpolation der Funktionswerte der benachbarten Eckpunkte der Zelle gewonnen werden, wodurch Ungenauigkeiten auftreten können.

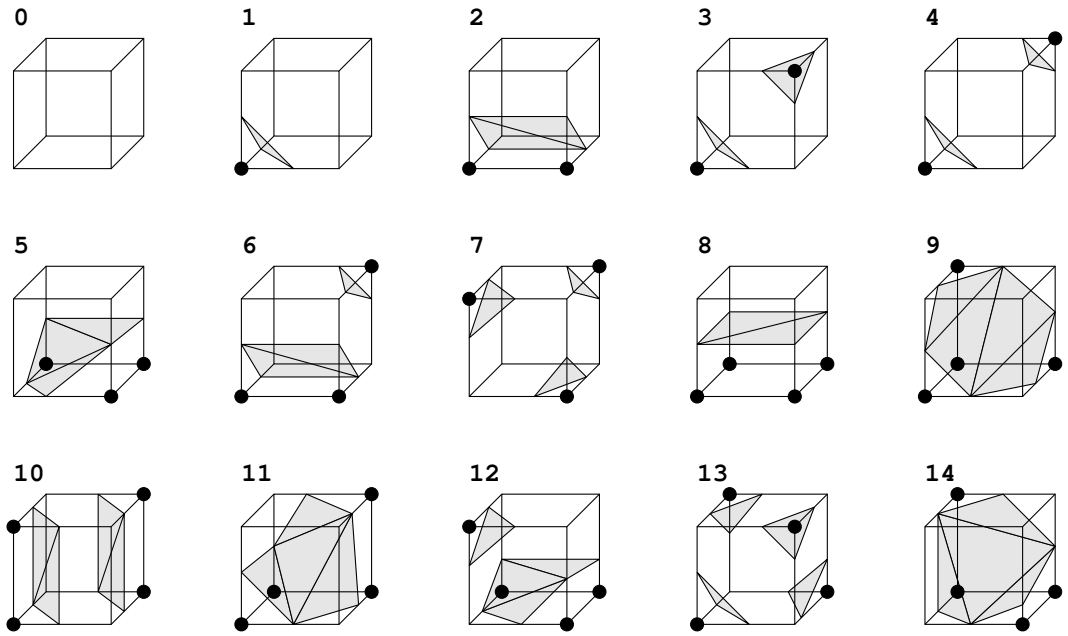


Abbildung 2.10.: Marching Cubes: Die verschiedenen Fälle

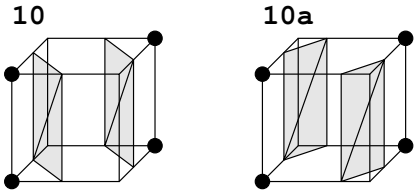
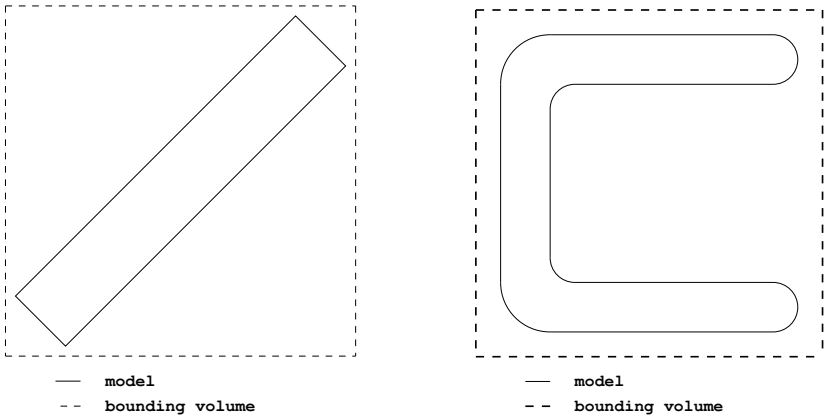


Abbildung 2.11.: Marching Cubes: Mehrdeutigkeiten



(a) unglückliche Wahl (b) schwieriges Modell

Abbildung 2.12.: Probleme bei der Erstellung von Bounding-Volumes

2.3.2. Marching Tetrahedra [Schumann 00]

Eine Optimierung des Marching Cubes Algorithmus stellt der Marching Tetrahedra Algorithmus dar. Hierbei wandert kein Würfel durch die einzelnen Zellen des Volumens, sondern ein Tetraeder. Der Vorteil hierbei ist, dass es nur drei verschiedenen zu betrachtende Fälle gibt (siehe Abbildung 2.13). Ausserdem werden maximal zwei Dreiecke pro Tetraeder generiert und es treten keine Mehrdeutigkeiten auf.

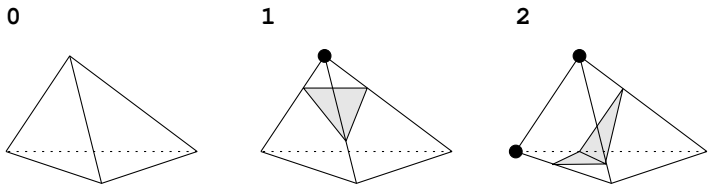


Abbildung 2.13.: Marching Tetrahedra: Die verschiedenen Fälle

2.3.3. Adaptive Subdivision [Bloomenthal 88]

Sowohl beim Marching Cubes als auch beim Marching Tetrahedra Algorithmus werden alle Zellen durchlaufen, unabhängig vom Typ der Zelle. Wenn die Zelle vom Typ 0 ist (wenn sich also kein Teil der Isofläche in der Zelle befindet) bedeutet dieses, dass die Zelle unnötigerweise bearbeitet wurde.

Genau hier setzt die adaptive Aufteilung an. Das Volumen wird nicht in gleich grosse Zellen zerlegt. Ausgehend von einer groben, initialen Aufteilung, werden nur die Zellen in 8 kleinere Zellen zerlegt, die nicht vom Typ 0 (siehe Abbildung 2.10) sind. Dieses wird solange wiederholt, bis die maximale Auflösung, also die minimale Zellengrösse erreicht ist. In Abbildung 2.14 wird dieses Vorgehen anhand eines 2D Beispieles dargestellt. Wenn das Volumen komplett in Zellen aufgeteilt ist, werden die kleinsten Zellen dann wieder auf Schnittpunkte mit der Isofläche untersucht. Zellen, die nicht unterteilt wurden, sind vom Typ 0 (siehe Abbildung 2.10). Sie haben also keine Schnittpunkte mit der gesuchten Isofläche und müssen somit nicht bearbeitet werden.

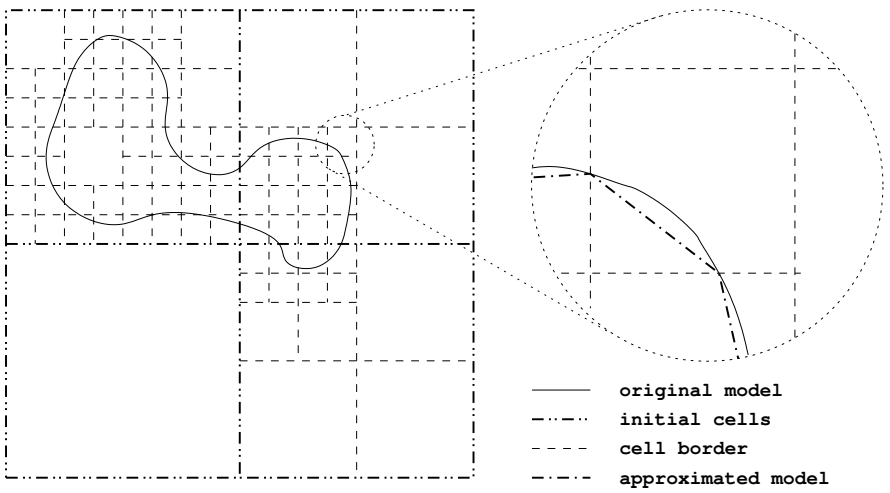


Abbildung 2.14.: Adaptive Subdivision: Prinzipdarstellung

2.3.4. Picewise-Linear Continuation [Bloomenthal 97]

Einen ähnlichen Ansatz wie die adaptive Aufteilung verfolgt die stückweise lineare Fortsetzung. Hierbei werden ausgehend von einer initialen Zelle, die einen Teil des Modells beinhalten muss[7], anhand der Kontur des Modells die Zellen markiert, die einen Teil des Modells beinhalten. Anschliessend werden nur die markierten Zellen bearbeitet.

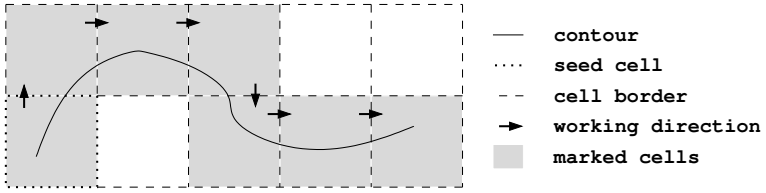


Abbildung 2.15.: Picewise-Linear Continuation: Prinzipdarstellung

2.3.5. Marching Triangles - das Original [Hilton 97]

Bei den bisher beschriebenen Ansätzen zur Extraktion von Isoflächen wurde das Volumen anfangs immer in Zellen aufgeteilt. Einen gänzlich anderen Ansatz verfolgt Hilton mit seinem Marching Triangles Algorithmus.

Das Modell wird durch eine Liste aller aussen liegenden Kanten des Polygon-Modells repräsentiert. Ausgehend von einem initialen Teil des Modells[8] wird eine Liste mit allen aussen liegenden Kanten generiert. Der Algorithmus ist als ein einfaches Durchlaufen der Liste beschrieben. Neue, durch den Algorithmus generierte, Kanten werden an das Ende der Liste angefügt, bearbeitete Kanten werden aus der Liste gelöscht. Das heisst, dass das erste Element der Liste immer die als nächstes zu bearbeitende Kante enthält. Die Kante, die gerade bearbeitet wird, wird aktive Kante genannt. Der Algorithmus wird so lange ausgeführt, bis alle Elemente der Liste bearbeitet wurden.

Die Bearbeitung eines Elementes der Liste sieht folgende Schritte vor:

1. Es wird ausgehend von der aktiven Kante i_{active}, j_{active} ein neuer Punkt x_p generiert. Hierzu wird der Mittelpunkt der aktiven Kante in Richtung der nach aussen weisenden, in der Ebene des Dreieckes i_{active}, j_{active}, k_{active} liegenden Senkrechten der aktiven Kante um die Projektionslänge l verschoben. Siehe hierzu Abbildung 2.16.
2. Der Punkt x_p wird auf die Isofläche projiziert.
   Hierzu wird der Punkt x_p entlang des Gradienten der Feldfunktion $f(P)$ im Punkt x_p um den Wert der Feldfunktion $f(P)$ im Punkt x_p verschoben.
3. Das neu generierte Dreieck wird dem 3D-Delaunay Test unterzogen.
   Dieser Test gilt als bestanden, wenn sich kein Teil des bereits existierenden Modells[9] innerhalb der Umkugel des zu testenden Dreieckes befindet. Die Umkugel ist die die Dreieckspunkte schneidende Kugel mit dem in der Ebene des Dreieckes liegenden Schnittpunkt der Mittelsenkrechten des Dreieckes als Mittelpunkt. Siehe hierzu Abbildung 2.17 sowie [Bronstein, Abbildung 2.14].

[7] Diese initiale Zelle kann durch einen einfachen Such-Algorithmus gefunden werden.
[8] Dabei kann es sich um ein einzelnes Dreieck oder aber um ein vorab generiertes Modell-Teil handeln.
[9] Leider gibt Hilton keine Auskunft darüber, was er unter "Teil des Modells" versteht.

4. Wenn der 3D-Delaunay Test bestanden ist:
   a) Das neue Dreieck wird dem Modell hinzugefügt.
   b) Die neuen Kanten des Dreieckes werden der Kanten-Liste hinzugefügt.
   c) Die aktive Kante wird aus der Kanten-Liste entfernt.
5. Wenn der 3D-Delaunay Test nicht bestanden ist, wird geprüft, ob ein neu generiertes Dreieck mit $x_p = j_{left}$ bzw. mit $x_p = i_{right}$ den Test besteht.
   Wird der Test bestanden, werden die beteiligten Kanten aus der Liste entfernt. Dieser Schritt erlaubt das lokale Verbinden von Modell-Teilen. Siehe hierzu Abbildung 2.18(a).
6. Wenn auch diese 3D-Delaunay Tests nicht bestanden sind, wird geprüft, ob ein neu generiertes Dreieck mit $x_p = i_{any}$ bzw. mit $x_p = j_{any}$ den Test besteht.
   Dieser Schritt erlaubt das Schliessen von eventuell noch vorhandenen Lücken im Modell und ist bei den meisten Modellen (z.B. einem Torus) sogar notwendig. Siehe hierzu Abbildung 2.18(b).
7. Der Algorithmus terminiert, sobald die Liste der noch zu bearbeitenden Kanten leer ist.

Der Marching Triangles Algorithmus hat gegenüber den zuvor beschriebenen Algorithmen den Vorteil, dass die Feldfunktion $f(P)$ nur dann ausgewertet wird, wenn ein Punkt auf der Isofläche $f(P) = 0$ generiert werden soll. Dieser Punkt befindet sich dann exakt auf der Isofläche und ist nicht - wie z.B. beim Marching Cubes Algorithmus - durch Interpolation gewonnen.
Ein Abarbeiten des gesamten Volumens - und somit auch von Zellen vom Typ 0 (komplett innerhalb oder ausserhalb des Modells) - ist nicht notwendig.
Desweiteren ist die Grösse der generierten Polygone wesentlich gleichmässiger als beim Marching Cubes Algorithmus. Daher kann unter Umständen sogar auf eine Modell-Simplifikation verzichtet werden.
Ausserdem ist die Wahl eines Bounding-Volumes nicht notwendig, da der Algorithmus sich nur an der Oberfläche des Modells orientiert.

Nach Abwägung der Vor- und Nachteile der zuvor beschriebenen Algorithmen wurde aufgrund der scheinbar überwiegenden Vorteile bei Hiltons Ansatz der Marching Triangles Algorithmus für diese Arbeit gewählt.
Da in der Veröffentlichung von Hilton einige Details fehlen, andere unerklärbar erscheinen, wurde eine vollständig eigene Umsetzung angestrebt. Diese wird in Abschnitt 3.1 beschrieben. Sie basiert zum Grossteil auf Hilton sowie auf einigen Konzepten der modifizierten Version von Charlot, die im nächsten Abschnitt beschrieben wird.

Hiltons Beschreibung der Betrachtung des Gradienten der Fläche, um auch gekrümmte Flächen bearbeiten zu können, konnte leider nicht erfolgreich umgesetzt werden. Aus Gründen der Übersichtlichkeit wurde daher die Beschreibung dieses Konzeptes weggelassen.

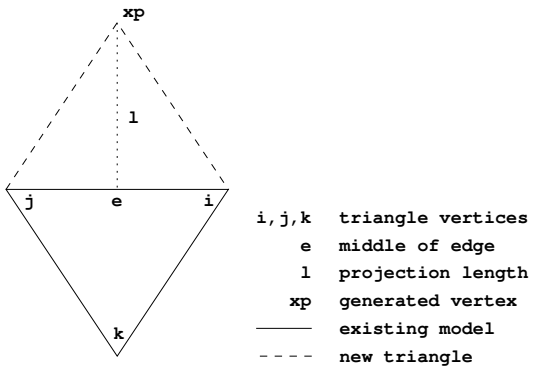


Abbildung 2.16.: Marching Triangles: Generierung von neuen Dreiecken

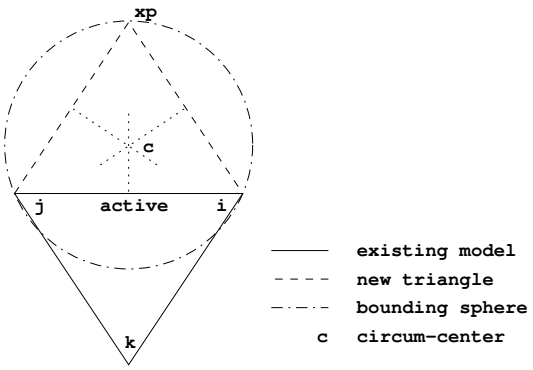
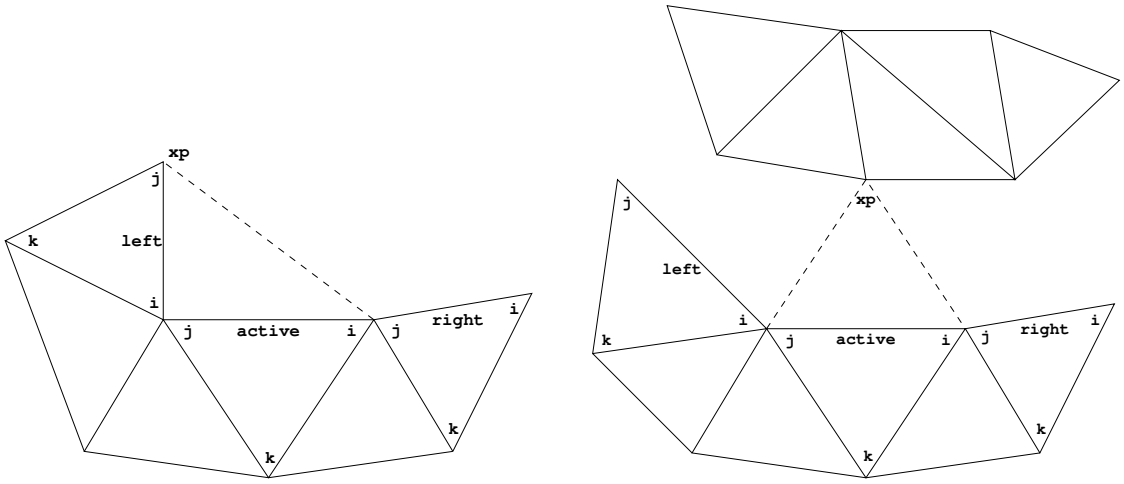


Abbildung 2.17.: Marching Triangles: 3D-Delaunay Test



(a) left / right

(b) overlap

Abbildung 2.18.: Marching Triangles: Schliessen mit existierenden Modell-Teilen

2.3.6. Marching Triangles - modifiziert von Charlot [Charlot 98]

Charlot stellt in [Charlot 98] eine eigene Umsetzung des Marching Triangles Algorithmus vor. Dabei sollte eine Möglichkeit geschaffen werden, um durch Simulation gewonnene geologische Modelle von Erdschichten zu polygonisieren. Die Modelle der Erdschichten können Verwerfungen und Brüche beinhalten, wodurch die zu polygonisierenden Flächen C^0-Unstetigkeiten aufweisen können. Diese Problematik entspricht überhaupt nicht der Aufgabenstellung dieser Arbeit, jedoch weist die Beschreibung von Charlot einige interessante Detaillösungen auf, die schliesslich auch Ihren Weg in die zu dieser Arbeit gehörende Umsetzung des Marching Triangles Algorithmus gefunden haben.

Die Änderungen gegenüber dem Original:

- Das Modell wird nicht durch eine Kanten-Liste, sondern durch einen Kanten-Stack repräsentiert[10].
- Durch die möglichen Unstetigkeiten des Modells kann es einen Modell-Rand geben, der gesondert behandelt wird.

[10]Charlot spricht zwar von einem Kanten-Stack, jedoch weist Abbildung 4 seiner Veröffentlichung darauf hin, dass doch eine Kanten-Liste verwendet wurde. Dieser Widerspruch konnte leider nicht aufgelöst werden.

- Ein Schliessen mit bereits existierenden Modell-Teilen ist nur für benachbarte Kanten vorgesehen.
- Bestimmte generierte Dreiecke werden unterteilt.
- Die Reihenfolge der Bearbeitungsschritte der aktiven Kante wurde gegenüber Hiltons Original verändert.

Die Bearbeitung eines Elementes des Kanten-Stacks sieht folgende Schritte vor:

1. Befindet sich die aktive Kante in der Nähe des eventuell existierenden Modell-Randes, wird die aktive Kante durch einen nicht näher beschriebenen iterativen Prozess auf den Rand verschoben.
   Die aktive Kante wird als BORDER markiert und vom Kanten-Stack entfernt.
2. Befindet sich die aktive Kante nicht in der Nähe des eventuell existierenden Modell-Randes, wird versucht, ob ein neues Dreieck durch lokales Schliessen mit benachbarten Kanten generiert werden kann.
   Dabei sind nur die benachbarten linken und rechten Kanten zugelassen, die einen Winkel kleiner 135° mit der aktiven Kante bilden.
   Das neu generierte Dreieck wird auf Überlappung mit existierenden Modell-Teilen getestet[11].
3. Wenn das Schliessen mit benachbarten Kanten nicht erfolgreich war, wird versucht ein neues Dreieck zu generieren.
   Um zu garantieren, dass es weder zu Überlappungen mit bereits existierenden Modell-Teilen kommt, noch dass zu kurze Kanten generiert werden, wird zuerst geprüft, ob sich ein Element des Modells[12] in der Umkugel des neuen Dreieckes befindet. Abweichend von Hiltons Original Algorithmus wird diese Umkugel wie folgt beschrieben: Es wird ausgehend von der aktiven Kante ein nur für die Dauer des Tests existierendes, gleichseitiges Dreieck generiert, dass in der Ebene des Dreieckes liegt, zu der die aktive Kante gehört. Die beiden Punkte der aktiven Kante werden von der Umkugel geschnitten. Der dritte Punkt des gleichseitigen Dreieckes ist das Zentrum der Umkugel (siehe hierzu Abbildung 2.19).
4. Wenn sich innerhalb der Umkugel ein Element des Modells befindet, gilt der Test als nicht bestanden.
   In diesem Falle wird die aktive Kante an das Ende des Kanten-Stacks verschoben und wieder bei Schritt 1 begonnen.
5. Befindet sich kein Element des Modells in der Umkugel, gilt der Test als bestanden.
   In diesem Falle wird ein neues Dreieck wie folgt generiert:
   a) Es wird eine Reihe von Punkten $x_{p,i}$ innerhalb der Kugel und auf deren Mittellinie liegend generiert (ausgehend von der Mitte der aktiven Kante und das Zentrum der Kugel schneidend). Der Abstand der Punkte zum Mittelpunkt der aktiven Kante wird durch eine geometrische Reihe bestimmt[13].
   b) Beginnend mit dem Punkt, der am weitesten von der aktiven Kante entfernt ist, werden die Punkte auf die implizite Fläche projiziert.

[11] Es bleibt leider unerwähnt, wie dieser Test genau umgesetzt wurde.

[12] Charlot versteht unter "Element des Modells" sowohl einen Punkt, als auch Teile von Kanten, wobei Kanten zugelassen sind, die adjazent zu der gerade aktiven Kante sind.

[13] Wie viele andere Details wird auch diese Reihe nicht näher beschrieben.

c) Die so gewonnenen Punkte werden wie folgt getestet:
   - Der Punkt muss innerhalb der Umkugel liegen.
   - Der Punkt muss innerhalb des Volumens liegen. Er darf also nicht ausserhalb eines eventuell existierenden Randes liegen.
   - Der Punkt muss ausserhalb der linken und rechten angrenzenden Dreiecke liegen.
   - Der Winkel zwischen dem Dreieck i_{active}, j_{active}, k_{active} und dem neuen Dreieck darf einen gewissen Wert[14] nicht überschreiten.

d) Wenn diese Bedingungen für einen Punkt x_p erfüllt sind, werden folgende Schritte ausgeführt:
   - Das neu entstandene Dreieck i_{active}, x_p, j_{active} wird dem Modell hinzugefügt.
   - Die neu entstandenen Kanten i_{active}, x_p und x_p, j_{active} werden dem Kanten-Stack hinzugefügt.
   - Die aktive Kante wird vom Kanten-Stack entfernt.

e) Wenn diese Bedingungen für keinen Punkt x_p erfüllt sind, werden statt dessen folgende Schritte ausgeführt:
   - Das Dreieck i_{active}, j_{active}, k_{active} wird in vier Dreiecke aufgeteilt. Siehe hierzu Abbildung 2.21.
     Dieser Schritt erlaubt das Bearbeiten von Modellen mit starker lokaler Krümmung.
   - Die neu entstandenen Dreiecke werden dem Modell hinzugefügt.
   - Das ursprüngliche Dreieck i_{active}, j_{active}, k_{active} wird gelöscht.
   - Die neu entstandenen Kanten i_{active}, n und n, j_{active} werden dem Kanten-Stack hinzugefügt.
   - Die aktive Kante wird vom Kanten-Stack entfernt.

6. Der Algorithmus terminiert sobald der Kanten-Stack leer ist[15].

Wie bei Hilton fehlen auch in dieser Beschreibung des Marching Triangles Algorithmus wesentliche Details. So fehlen zum Beispiel Beschreibungen der Tests auf Überlappung mit bestehenden Modell-Teilen.
Dafür gaben einige Konzepte von Charlot die Grundidee für eigene Modifikationen. So ist zum Beispiel das Verschieben einer zur Zeit nicht zu bearbeitenden Kante an das Ende des Kanten-Stacks ein Konzept, das - vorausgesetzt es wird richtig umgesetzt - zu guten und schnellen Ergebnissen beitragen kann.

Charlots Konzept des Teilens von Dreiecken um auch stark gekrümmte Flächen bearbeiten zu können erschien auf den ersten Blick sehr vielversprechend. Nachdem eigene Versuche mit einem sich an die Krümmung der Fläche anpassenden Algorithmus nicht erfolgreich verliefen, wurde dieses Konzept jedoch verworfen.

[14] Dieser Wert ist ebenfalls ein Detail, dass im Verborgenen bleibt.

[15] Abweichend von der Beschreibung von Charlot ist davon auszugehen, dass der Algorithmus nicht immer terminiert, da der Stack durch das beschriebene Verschieben von Elementen an dessen Ende nicht leer werden muss.

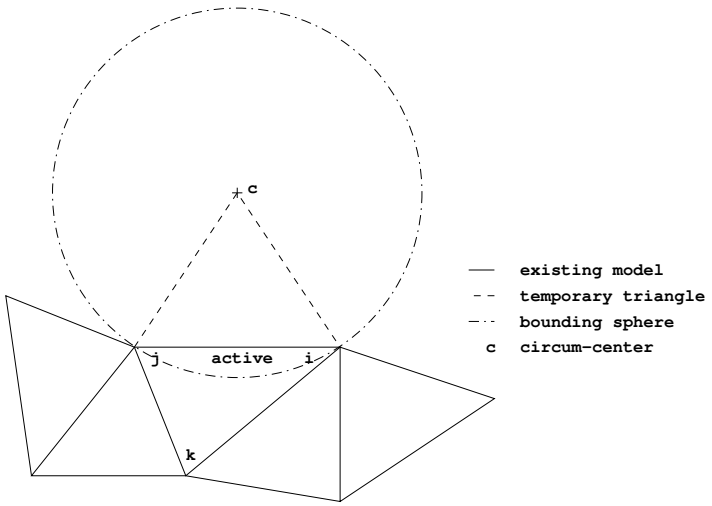


Abbildung 2.19.: Charlots Marching Triangles: Test auf Gültigkeit

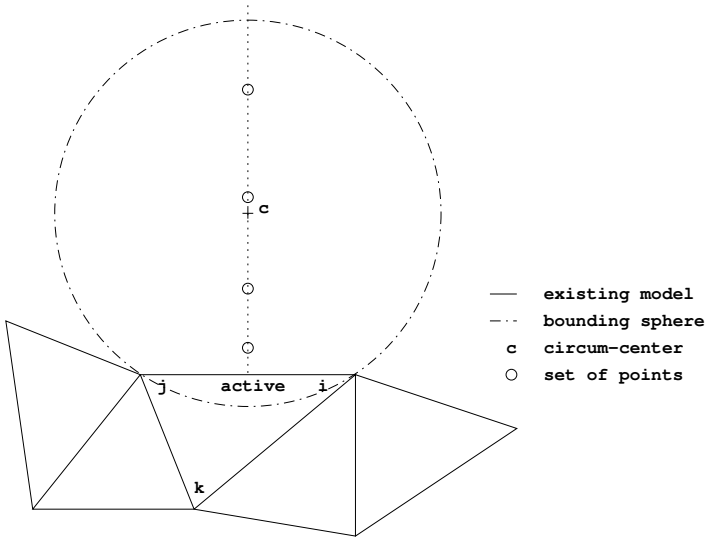


Abbildung 2.20.: Charlots Marching Triangles: Generierung von neuen Dreiecken

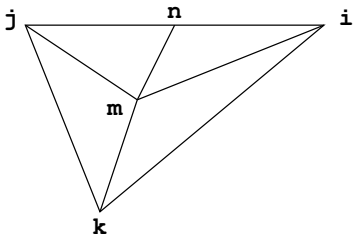


Abbildung 2.21.: Charlots Marching Triangles: Teilung von Dreiecken

3. Umsetzung

3.1. Marching Triangles Algorithmus

Nach Analyse des in Abschnitt 2.3.5 beschriebenen Original Algorithmus sowie der Modifikationen durch Charlot (siehe Abschnitt 2.3.6) wurde ein eigener, modifizierter Marching Triangles Algorithmus implementiert. Dabei wurden sowohl aus Hiltons als auch aus Charlots Beschreibung Konzepte übernommen und der Aufgabenstellung entsprechend angepasst.

Die Änderungen gegenüber dem Original und der Modifikation von Charlot im einzelnen:

- Ähnlich wie bei Charlot wurde ein Stack-Ansatz verfolgt. Dieser wurde wegen der Verwendungsart der Kanten gegenüber dem Listen-Ansatz bevorzugt.
- Der 3D-Delaunay Test wurde basierend auf dem Vorgehen von Hilton umgesetzt. Dabei wurden Modifikationen umgesetzt, die ähnlich wie bei Charlot dafür sorgen, dass keine zu kleinen Kanten generiert werden.
- Anders als bei Charlot wird beim 3D-Delaunay Test nur getestet, ob sich bereits bestehende Punkte in der zu testenden Umkugel befinden. Ein Test auf Überlappung mit Kanten-Teilen findet nicht statt.
- Abweichend von Hilton und Charlot sind einige wenige Punkte des Modells bei bestimmten 3D-Delaunay Tests in einer Umkugel des Tests zugelassen. Siehe hierzu Abschnitt 3.2.
- Kanten, die den 3D-Delaunay Test nicht bestanden haben, werden wie bei Charlot an das Ende des Kanten-Stacks verschoben.
  Die Kanten werden also zu einem späteren Zeitpunkt wieder zur Verarbeitung vorliegen. Dieses Verschieben der Kante an das Ende des Kanten-Stacks wird - anders als bei Charlot - in einem Zähler in der Datenstruktur der Kanten vermerkt. Dieser Zähler beschreibt also das Alter der Kante. Eine Kante, die das erste Mal zur Verarbeitung vorliegt hat das Alter 1. Jedesmal, wenn eine Kante zur Bearbeitung vom Stack genommen wird, wird das Alter um eins inkrementiert.
  Das Alter einer Kante wurde eingeführt, um zu gewährleisten, dass der Algorithmus - im Gegensatz zu der Beschreibung von Charlot - auch terminiert.
- Ähnlich wie bei Charlot werden bestimmte generierte Kanten geteilt. Jedoch werden - abweichend von der Beschreibung von Charlot - nur zu lange Kanten geteilt, um zu gewährleisten, dass relativ gleichmässige Polygone generiert werden. So werden zwei Polygone anstelle von nur einem Polygon generiert.
  Diese Teilung betrifft aber nicht wie bei Charlot Polygone, die bereits generiert wurden, sondern nur solche, die gerade generiert werden.
- Die Reihenfolge der Bearbeitungsschritte der aktiven Kante wurde gegenüber Hiltons Original und der Modifikation von Charlot verändert.

Die Reihenfolge der Bearbeitung eines Elementes des Kanten-Stacks wurde vor allem durch empirische Versuche bestimmt. Die Reihenfolge, die die schnellsten und besten Ergebnisse

lieferte, wurde bevorzugt umgesetzt. Dabei wurde verstärkt darauf geachtet, dass die Bearbeitung eines Elementes des Kanten-Stacks schnell erfolgreich abgeschlossen oder aber so schnell als möglich abgebrochen wird.

Die Bearbeitung eines Elementes des Kanten-Stacks sieht folgende Schritte vor:

1. Das Alter der aktiven Kante wird um eins inkrementiert.
2. Ausgehend von der aktiven Kante i_{active}, j_{active} wird der Winkel mit der angrenzenden, linken Kante i_{left}, j_{left} überprüft.
3. Wenn der Winkel kleiner als 120° ist, werden folgende Schritte ausgeführt:
   a) x_p wird gleich i_{left} gesetzt. Siehe hierzu Abbildung 2.18(a).
   b) Die Länge der eben generierten Kante wird überprüft. Wenn diese Länge die Projektionslänge l um den Faktor 1.4 überschreitet, wird die Kante halbiert.
   Dadurch werden zwei Dreiecke generiert, wodurch verhindert wird, dass sehr unterschiedliche Dreiecke generiert werden. Siehe hierzu Abbildung 3.1.
   c) Es wird überprüft, ob der Winkel auch kleiner als 90° ist. Das Ergebnis dieser Überprüfung wird in einem flag zur späteren Auswertung gespeichert.
4. Wenn der Winkel grösser als 120° ist, werden die vorausgegangenen Schritte für den Winkel zwischen der aktiven Kante und der angrenzenden, rechten Kante i_{right}, j_{right} wiederholt.
5. Wenn weder der linke noch der rechte Winkel kleiner als 120° ist, wird ausgehend von der aktiven Kante ein neuer Punkt x_p generiert.
   Hierzu wird der Mittelpunkt der aktiven Kante in Richtung der nach aussen weisenden, in der Ebene des Dreieckes i_{active}, j_{active}, k_{active} liegenden, Senkrechten der aktiven Kante um die Projektionslänge l verschoben. Siehe hierzu Abbildung 2.16.
6. Das neu generierte Dreieck wird dem 3D-Delaunay Test unterzogen.
   Dieser Test gilt als bestanden, wenn sich kein Punkt oder nur erlaubte Punkte des Modells in einer Umkugel befinden. Die genaue Beschreibung eventuell erlaubter Punkte in einer Umkugel des 3D-Delaunay Tests befindet sich in Abschnitt 3.2.
   Es kommen je nach Typ und Alter der aktiven Kante verschiedene 3D-Delaunay Tests zur Anwendung:
   a) Jedes neue Dreieck mit dem Alter 1 der aktiven Kante:
   Es wird der Standard 3D-Delaunay Test verwendet. Dieser unterscheidet sich von Hiltons 3D-Delaunay Test (siehe Abbildung 2.17) dadurch, dass - ausser wenn mit vorhandenen Modell-Teilen geschlossen wird - eine zweite Umkugel mit dem Radius $r = 0.7 \cdot l$ mit x_p als Zentrum verwendet wird.
   Hierdurch wird - ähnlich wie bei Charlot - die Generierung von sehr kurzen Kanten verhindert. Siehe hierzu Abbildung 3.2(a).
   b) Ein Dreieck mit einem linken oder rechten Winkel kleiner als 90° und mit einem Alter der aktiven Kante grösser als 1:
   Es wird der 3D-Delaunay Test für sehr kleine Winkel verwendet. Hierbei werden zwei Umkugeln überprüft, die je als Durchmesser den Wert der Länge einer der beteiligten Kanten (*active* und $left$ oder *active* und *right*) und den Mittelpunkt der jeweiligen Kante als Zentrum haben.
   Durch diese Art von Test wird wesentlich weniger Raum überprüft als mit dem Standard 3D-Delaunay Test, wodurch das Dreieck eher als gültig akzeptiert wird. Siehe hierzu Abbildung 3.2(b).
   c) Ein Dreieck, dessen neue Kante wegen zu grosser Länge geteilt wurde und mit einem Alter der aktiven Kante grösser als 1:
   Es wird der 3D-Delaunay Test für zu lange Kanten verwendet. Hierbei wird eine Umkugel mit der Länge der ungeteilten zu langen Kante als Durchmesser und dem Mittelpunkt der ungeteilten zu langen Kante als Zentrum verwendet.

Durch diese Art von Test wird wesentlich weniger Raum überprüft als mit dem Standard 3D-Delaunay Test, wodurch das Dreieck eher als gültig akzeptiert wird. Siehe hierzu Abbildung 3.2(c).

7. Wenn der 3D-Delaunay Test bestanden ist:
   a) Wird das neue Dreieck dem Modell hinzugefügt.
   b) Werden die neuen Kanten des Dreieckes dem Kanten-Stack hinzugefügt.
8. Wenn der 3D-Delaunay Test nicht bestanden ist, hängen die weiteren Schritte von dem Alter der aktiven Kante ab:
   a) Alter der aktiven Kante kleiner als 3:
   Die Kante wird an das Ende des Kanten-Stacks verschoben. Es wird das oberste Element des Kanten-Stacks als neue aktive Kante genommen und wieder bei Schritt 1 begonnen.
   b) Alter der aktiven Kante grösser als oder gleich 3:
   x_p wird an den Punkt verschoben, der das Nicht-Bestehen des 3D-Delaunay Tests verursacht hat. Das hierdurch neu entstandene Dreieck wird erneut dem Standard 3D-Delaunay-Test unterzogen. Hierdurch wird das Schliessen mit existenten Modell-Teilen ermöglicht. Siehe hierzu Abbildung 3.2(d).
9. Der Algorithmus terminiert sobald:
   a) der Kanten-Stack leer ist,
   b) die vom Benutzer vorgegebene, maximal zu erzeugende Anzahl von Dreiecken erreicht ist, oder
   c) die auf dem Kanten-Stack verbleibenden Kanten ein gewisses Alter überschritten haben.
   In diesem Falle wird abgebrochen, da davon auszugehen ist, dass die Kanten, die bislang nicht bearbeitet werden konnten, auch in Zukunft nicht bearbeitet werden können. Um in diesem Falle eine Endlosschleife zu verhindern, wird der Programmablauf abgebrochen.
   d) der gesamte Stack durchlaufen wurde, ohne, dass auch nur ein Element bearbeitet werden konnte.

Diese durch empirische Versuche bestimmte Reihenfolge bietet gute bis sehr gute Ergebnisse bei der Polygonisierung unterschiedlicher Feldfunktionen. Zum Testen wurden algebraische Feldfunktionen, zusammengesetzte algebraische Feldfunktionen sowie Algorithmen getestet.

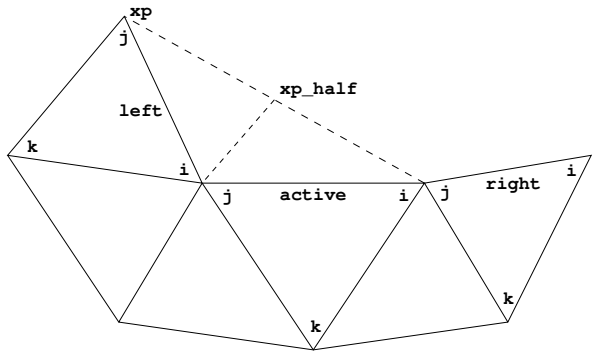
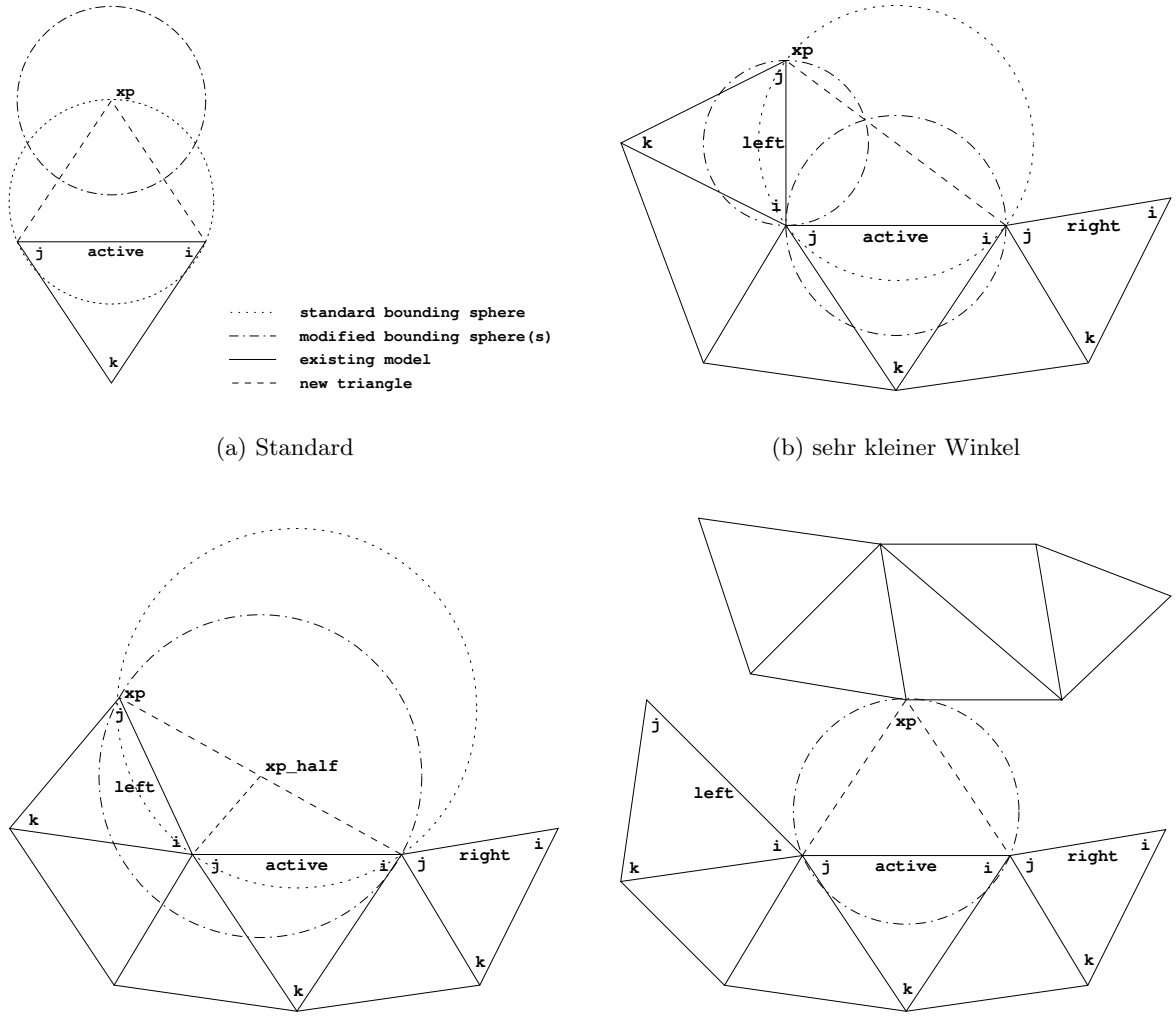


Abbildung 3.1.: Marching Triangles: Teilen einer zu langen Kante



(a) Standard

(b) sehr kleiner Winkel

(c) lange Kante

(d) overlap

Abbildung 3.2.: Marching Triangles: Modifizierte 3D-Delaunay Tests

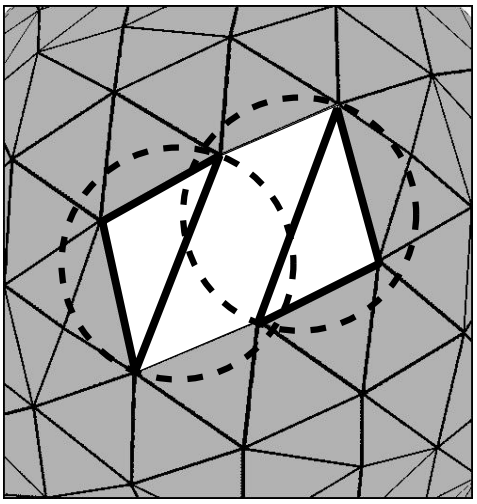
3.2. Sonderfälle beim Marching Triangles Algorithmus

Bei der im Rahmen dieser Arbeit erstellten Marching Triangles Algorithmus Implementierung sind einige Sonderfälle bei der Generierung von Polygonen umgesetzt worden. Diese Sonderfälle sind durch empirische Tests bei der Polygonisierung der verschiedenen Feldfunktionen gefunden worden:

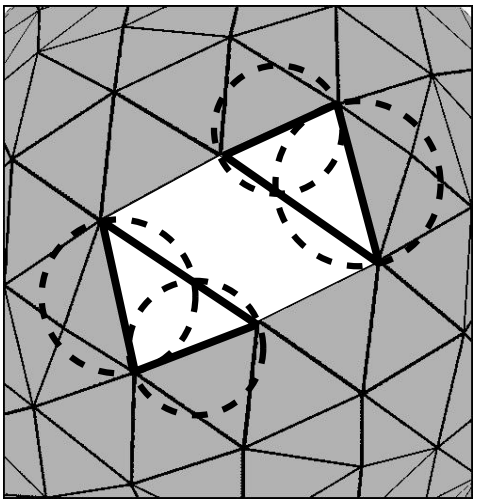
- Bei der Polygonisierung der Kugel-Feldfunktion ist ein Fall aufgetreten, bei dem das Modell nicht ganz geschlossen werden konnte.
  Ein solcher Fall ist beispielhaft in Abbildung 3.3 dargestellt. Dieses Problem liess sich durch Modifikation eines 3D-Delaunay Tests lösen.
- Bei der Polygonisierung der Torus-Feldfunktion ist ein Fall aufgetreten, bei dem das Modell nicht ganz geschlossen werden konnte.
  Dieser Fall ist in Abbildung 3.4 dargestellt. Er basiert auf dem Schliessen mit bereits bestehenden Modell-Teilen und konnte durch eine zusätzliche Abfrage beim Schliessen von kleinen Winkeln aufgelöst werden.

Sonderfall: 3D-Delaunay Test für sehr kleine Winkel

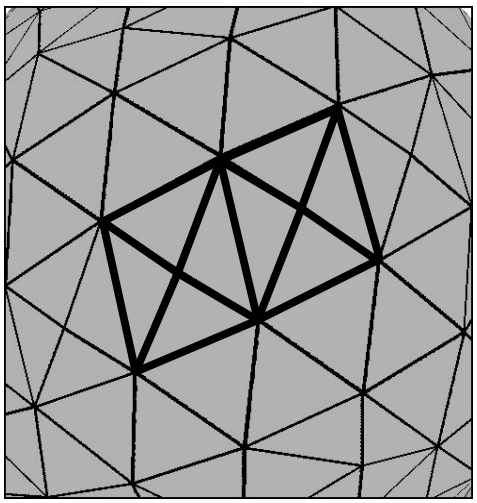
Abbildung 3.3(a) stellt das Nicht-Bestehen von zwei der vier möglichen 3D-Delaunay Tests dar. Diese beiden Tests werden wie erwartet nicht bestanden, da sich jeweils gegenüber der eventuellen neuen Kante ein Punkt des bereits bestehenden Modells in der Umkugel befindet.
Abbildung 3.3(b) stellt die beiden noch verbleibenden der vier möglichen 3D-Delaunay Tests dar. Diese Tests werden nicht bestanden, da sich jeweils der dritte Punkt des zu einer der beteiligten Kanten gehörenden Dreieckes in der Umkugel befindet.
Abbildung 3.3(c) zeigt die Polygonisierung derselben Kugel-Feldfunktion, nachdem beim 3D-Delaunay Test für sehr kleine Winkel alle Punkte der zu den beteiligten Kanten gehörenden Dreiecke in den Umkugeln zugelassen wurden.



(a) Sonderfall: sehr kleiner Winkel #1



(b) Sonderfall: sehr kleiner Winkel #2



(c) gelöster Sonderfall

Abbildung 3.3.: Sonderfall: Sehr kleiner Winkel

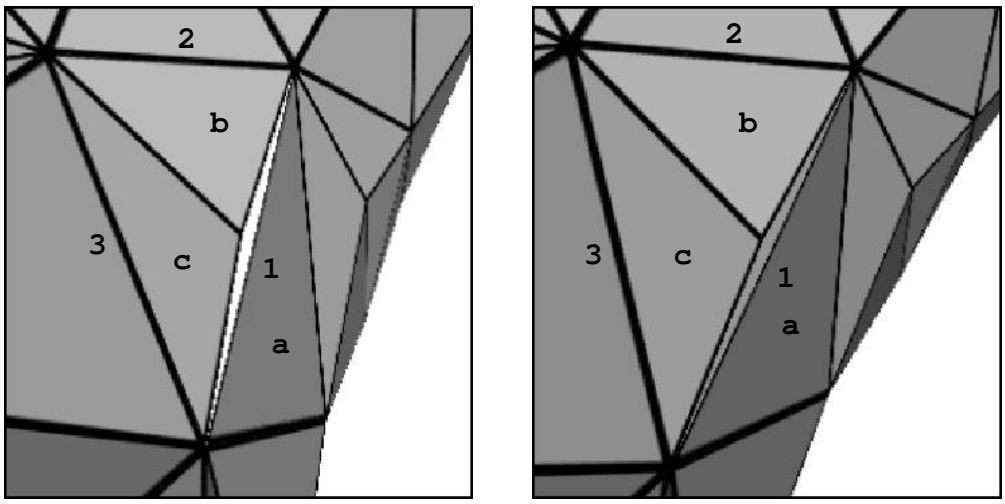
Ein ähnlicher Sonderfall ist auch bei dem 3D-Delaunay Test für lange Kanten denkbar, konnte aber nicht beobachtet werden.

Sonderfall: Overlap

In Abbildung 3.4 ist der zuvor erwähnte Sonderfall dargestellt, der bei der Polygonisierung der Torus-Feldfunktion aufgetreten ist. Dieser Fall tritt nur sehr selten unter folgenden Randbedingungen auf:

- Mindestens eine der Kanten eines durch Schliessen mit bestehenden Modell-Teilen - dem sogenannten Overlap (siehe Abbildung 2.18(b)) - generierten Dreiecks überschreitet die maximal zulässige Kantenlänge von $1.4 \cdot l$.
  Da in diesem Falle meist beide Kanten zu lang sind, macht ein Teilen einer einzelnen Kante, wie in Abbildung 3.1 dargestellt, keinen Sinn. Es ist also ein Dreieck mit zu langen Kanten entstanden. Im Beispiel in Abbildung 3.4(a) ist dieses das Dreieck a.
- Eine der beiden zuvor generierten zu langen Kanten bildet den Rand einer Lücke, die durch Generierung eines Dreieckes geschlossen werden würde. Im Beispiel ist dieses die Kante 1.
- Die Kante (im Beispiel Kante 2), durch deren Bearbeitung eine Lücke (im Beispiel durch die Kanten 1, 2 und 3 begrenzt) geschlossen werden würde, die an eine der beiden zuvor generierten zu langen Kanten grenzt (im Beispiel Kante 1), bildet mit einer dieser Kante gegenüberliegenden Kante (im Beispiel Kante 3) einen Winkel kleiner als 120°.

Wenn alle diese Bedingungen zutreffen, wird die Lücke durch ein Dreieck geschlossen, dessen neu generierte Kante mit der zuvor generierten zu langen Kante übereinstimmt (im Beispiel mit Kante 1). Da im Falle des Schliessens eines zu kleinen Winkels die Länge der neu generierten Kante überprüft wird, diese durch die zuvor erwähnte Übereinstimmung zu lang ist, wird diese Kante geteilt. Im Beispiel werden durch diesen Schritt die beiden Dreiecke b und c generiert. Da normalerweise alle am Schliessen einer Lücke beteiligten Kanten vom Kanten-Stack entfernt werden, würde in diesem Falle eine Lücke im Modell verbleiben. Diese Lücke ist in Abbildung 3.4(a) zu erkennen.
Abbildung 3.4(b) zeigt den gleichen Fall, nachdem beim Schliessen von Lücken getestet wird, ob eine der beteiligten Kanten die maximal zulässige Länge von $1.4 \cdot l$ überschreitet. Ist dieses der Fall, wird die entsprechende Kante nicht vom Kanten-Stack entfernt. Ausserdem werden die durch Teilung des generierten Dreieckes entstandenen Kanten auf den Kanten-Stack gelegt. Durch diesen Schritt ist die entstandene Lücke durch drei Kanten auf dem Kanten-Stack beschrieben, womit sie beim weiteren Bearbeiten geschlossen werden kann.



(a) Sonderfall: Overlap (b) gelöster Sonderfall

Abbildung 3.4.: Sonderfall: Overlap

3.3. Benutzer-Handbuch

In diesem Abschnitt wird erklärt, wie das zu dieser Arbeit gehörende Programm kompiliert, gestartet und bedient werden kann.

3.3.1. Systemvoraussetzungen

Das zu dieser Arbeit gehörende Programm wurde für ein Computer-System unter SGI-IRIX bzw. PC-Linux[1] entwickelt. Es wird eine voll funktionsfähige OpenGL (mit GLUT) Installation benötigt. Eine Grafikkarte mit Hardware-Texturing[2] ist empfohlen, aber nicht unbedingt notwendig.

Sollte Ihnen die grafische Ausgabe des Programmes zu langsam erscheinen, lässt sich durch Ändern der Konstante *INTERPOL_METHOD* in der Datei *scene.h* von *GL_LINEAR* auf *GL_NEAREST* die Geschwindigkeit der Darstellung erhöhen, was allerdings eine etwas niedrigere Darstellungsqualität zur Folge hat. Nach dieser Änderung muss das Programm neu kompiliert werden. Siehe hierzu Abschnitt 3.3.2.
Dieser Schritt ist bei weniger gut ausgestatteten Systemen sehr zu empfehlen.

Für Systeme ohne jeglichen Textur Support[3] steht das target *notexture* zur Verfügung. Siehe hierzu Abschnitt 3.3.2.

3.3.2. Kompilierung

Dieses Programm wurde für den *cc* Compiler in Verbindung mit *make* entwickelt[4]. Zu dem Programm gehören zwei verschiedene Makefiles: Makefile.irix für Systeme unter SGI-IRIX und Makefile.linux für PCs unter Linux. Sie unterscheiden sich lediglich in einigen systemspezifischen Punkten (Pfade zu den benötigten Bibliotheken und die genutzten Optionen des jeweiligen Compilers). Sollte das Kompilieren mit dem passenden Makefile nicht zum Erfolg führen, wird die Überprüfung der Pfadangaben zu den unterschiedlichen Bibliotheken empfohlen.

Um beim Aufruf von *make* auf die Angabe des passenden Makefiles verzichten zu können, wird empfohlen, vor dem ersten Kompilieren einen symbolischen Link mit Namen *Makefile* auf das passende Makefile einzurichten. Dieses kann durch Eingabe des folgenden Befehls in einer shell in dem Verzeichnis, das die Dateien des Programmes enthält, gemacht werden:

Bei SGI-IRIX Systemen: *ln -s Makefile.irix Makefile*
Bei PC-Linux Systemen: *ln -s Makefile.linux Makefile*

Danach kann das Programm einfach durch folgenden Aufruf kompiliert werden:

make target

[1] Getestet wurden SuSE und Debian Installationen.
[2] z.B. SGI Indigo2 MaxImpact oder Nvidia GeForce
[3] z.B. SGI Octane SI
[4] Aus Kompatibilitätsgründen wurde dabei auf den Einsatz von GNU-Tools verzichtet, da diese nicht zu einer Standard SGI-IRIX Installation gehören.

Dabei ist *target* durch eines der folgenden, zur Verfügung stehenden targets zu ersetzen:

all	-	für eine Standard Kompilation
debug	-	es wird eine ausführbare Datei erzeugt, bei der debug-Meldungen generiert werden
printable	-	es wird eine ausführbare Datei erzeugt, die druckbare Ergebnisse (dunkel auf hell ohne Koordinatenachsen) generiert
noaxis	-	wie all, jedoch ohne Darstellung der Koordinatenachsen
notexture	-	wie all, jedoch ohne Nutzung von Texturen
help	-	Ausgabe aller möglichen targets
tar	-	es wird ein tar-Archiv mit allen benötigten Dateien erzeugt
clean	-	es werden alle nicht benötigten Dateien gelöscht

Tabelle 3.1.: Verfügbare make-targets

3.3.3. Programmstart

Das Programm kann durch einfachen Aufruf gestartet werden. Standardmässig wird ein executable mit dem Namen *gnasl* erzeugt. Dieser Name lässt sich im Makefile durch Setzen der Konstante *EXEC* einstellen.

Optionen und Parameter

Beim Programmstart stehen folgende Optionen zur Verfügung:

-f	-	Angabe der zu polygonisierenden Feldfunktion Parameter: die Nummer der gewünschten Feldfunktion (siehe Tabelle 3.3) Default-Wert: Feldfunktion 1 (Kugel)
-n	-	Angabe der maximalen Anzahl der zu generierenden Dreiecke Parameter: maximale Anzahl der zu generierenden Dreiecke Default-Wert: -1 (keine Begrenzung)
-r	-	Angabe der Projektionslänge l für den Marching Triangles Algorithmus Parameter: l Default-Wert: 1.0
-h	-	Ausgabe der zugelassenen Optionen

Tabelle 3.2.: Optionen und Parameter des Programmes

Folgende Feldfunktionen stehen als Parameter zur Option *-f* zur Verfügung:

1	-	Kugel	siehe Abbildung A.1(a)
2	-	Ellipsoid	siehe Abbildung A.1(b)
3	-	Super-Ellipsoid	siehe Abbildung A.1(c)
4	-	Hot-Dog	siehe Abbildung A.1(d)
5	-	Boje	siehe Abbildung A.1(e)
6	-	Atom	siehe Abbildung A.1(f)
7	-	Torus	siehe Abbildung A.1(g)
8	-	Ei	siehe Abbildung A.1(h)
9	-	Erdnuss	siehe Abbildung A.1(i)

Tabelle 3.3.: Zur Verfügung stehende Feldfunktionen

3.3.4. Bedienung mit der Maus

Nach der Polygonisierung der Feldfunktion wird das generierte Polygon-Modell in einem eigenen Fenster dargestellt. In diesem Fenster kann mit der Maus wie folgt navigiert werden:

Taste	Funktion der Mausbewegung
links	Rotation des Modells um Betrachter-X- und Y-Achse
mitte	Translation des Modells in Betrachter-X- und Y-Richtung
links + mitte	Translation des Modells in Betrachter-Z-Richtung

Tabelle 3.4.: Bedienung mit der Maus

3.3.5. Beenden des Programmes

Beendet wird das Programm durch Drücken der *ESC* Taste. Hierbei muss der Input-Fokus bei dem Fenster mit der grafischen Darstellung der polygonisierten Feldfunktion sein. Dieses wird durch einfaches Anklicken des Fensters mit der Maus erreicht.

Nach Beenden des Programmes wird eine Abschluss-Meldung vom Programm ausgegeben.

3.3.6. Programm Ausgaben

Standard Ausgaben

Nach dem Start des Programmes wird der Benutzer über folgende Dinge informiert:

- Start der Initialisierungsphase und des Marching Triangles Algorithmus
- Feldfunktion, die polygonisiert wird
- Nummer des zur Zeit in Bearbeitung befindlichen Dreieckes

Nach Ablauf der Polygonisierung werden folgende Daten standardmässig von dem Programm berechnet und über *stdout*[5] ausgegeben:

# edges left	-	Anzahl der noch auf dem Stack befindlichen Kanten
# triangles	-	Anzahl der generierten Dreiecke
#edge age 1	-	Anzahl der nur einmal bearbeiteten Kanten
#edge age 2	-	Anzahl der zweimal bearbeiteten Kanten
#edge age 3	-	Anzahl der dreimal bearbeiteten Kanten
#elder edges	-	Anzahl der Kanten, die mehr als dreimal bearbeitet wurden
mid edge age	-	durchschnittliche Anzahl Bearbeitungsschritte pro Kante
max edge age	-	maximale Anzahl Bearbeitungsschritte
# steps	-	Anzahl Bearbeitungsschritte insgesamt
# rejects	-	Anzahl zurückgewiesener Kanten
# removes	-	Anzahl vom Stack gelöschter Kanten
# long edges	-	Anzahl der wegen zu grosser Kantenlänge geteilter Kanten (siehe Abbildung 3.1)
# overlaps	-	Anzahl genutzter overlaps mit existierenden Modell-Teilen (siehe Abbildung 2.18(b))

Tabelle 3.5.: Standard Ausgaben

[5] Als Standard ist dies die shell, aus der das Programm gestartet wurde.

Zusätzliche Ausgaben im Debug-Modus

Wenn das Programm im Debug-Modus kompiliert wurde (durch Aufruf von *make debug*), werden ergänzend zu den Standard-Ausgaben diverse Status- und Fortschrittsmeldungen ausgegeben. Diese werden durch bedingtes Kompilieren von Programmteilen erzeugt, die bei einer anderen Kompilation ausser acht gelassen werden.

Beispielhaft werden im Folgenden ein Teil der Ausgaben aufgeführt, die das Programm im Debug-Modus nach dem Aufruf von *gnasl -f 7 -n -1 -r 1.0* (es soll die Torus-Feldfunktion in der Auflösung 1.0 vollständig polygonisiert werden) generiert:

```
working on triangle 932, worked on this edge: 1 times
calculating new point...
distance of point to isosurface: 0.004
calculating bounding sphere (standard)...
checking bounding sphere(s)...
xp invalid: edgeOverLap (standard)!
overlap: moving edge to end...

working on triangle 932, worked on this edge: 3 times
left angle < MAX_ANGLE...
new right edge too long: 1.515 times l
old edge: using long edge bounding sphere...
left angle < SMALL_ANGLE...
old edge: using small angle bounding sphere...
calculating bounding sphere (small angle)...
checking bounding sphere(s)...
overLapLeftEdge...

distance of point to isosurface: 0.001

working on triangle 934, worked on this edge: 1 times
```

Es ist zu sehen, dass jeder einzelne Schritt des Marching Triangles Algorithmus kommentiert wird. Zusätzlich werden Daten über die aktiven Kanten sowie über die generierten Punkte und Kanten ausgegeben. Gerade bei der Fehlersuche bzw. der Optimierung des Programmes sind diese Ausgaben von sehr grosser Hilfe.

Fehlermeldungen

Sollte ein Fehler auftreten, wird dieser über *stderr* ausgegeben. In der folgenden Tabelle sind mögliche Fehlermeldungen, deren Ursachen, sowie Massnahmen zur Beseitigung des Fehlers aufgeführt:

Fehlermeldung	Ursache / Massnahme
unknown option or missing arg:	Es wurde versucht, eine ungültige Option zu nutzen. Für gültige Optionen siehe Tabelle 3.2
unknown option:	Es wurde versucht, eine ungültige Option zu nutzen. Für gültige Optionen siehe Tabelle 3.2
error while reading parameter...	Es wurde ein ungültiger Parameter angegeben. Für gültige Parameter siehe Tabelle 3.2
error while initialising graphics... aborting...	Bei der Initialisierung Ihres Grafik-Adapters ist ein Fehler aufgetreten. Vergewissern Sie sich, dass Ihr System die Systemvoraussetzungen erfüllt.
error while allocating memory...	Es wurde versucht, nicht vorhandenen Speicher zu allozieren. Weniger Polygone erzeugen lassen, mehr Arbeitsspeicher bereitstellen.
error while creating edgeElement...	Es konnte aufgrund nicht allozierbaren Speichers kein Kanten-Listen-Element erzeugt werden.
error while creating pointElement...	Es konnte aufgrund nicht allozierbaren Speichers kein Punkte-Listen-Element erzeugt werden.
error while creating triangleElement...	Es konnte aufgrund nicht allozierbaren Speichers kein Dreiecks-Listen-Element erzeugt werden.
error... element not part of list!	Es wurde versucht, ein Element aus einer Liste zu löschen, dass kein Element der Liste ist. Dieser Fehler weist auf einen Programmier-Fehler hin.[6]
type of bounding sphere not set...	Es wurde versucht, einen nicht existierenden 3D-Delaunay Test auszuführen. Dieser Fehler weist auf einen Programmier-Fehler hin.[6]
error in local dependencies: activeLeft wrong... active: activeLeft: activeRight:	Es ist ein Fehler in der lokalen Datenstruktur des Polygon-Modells aufgetreten. Die Polygonisierung sollte abgebrochen werden und mit neuen Parametern erneut gestartet werden.
error in local dependencies: activeRight wrong... active: activeLeft: activeRight:	Es ist ein Fehler in der lokalen Datenstruktur des Polygon-Modells aufgetreten. Die Polygonisierung sollte abgebrochen werden und mit neuen Parametern erneut gestartet werden.

Tabelle 3.6.: Fehlermeldungen

[6] Bei keinem der Tests ist einer dieser Fehler aufgetreten. Sollte einer dieser Fehler dennoch auftreten, ist dieser auf fehlerhafte Nutzung der zur Verfügung gestellten Funktionen zur Nutzung einer der implementierten Listen-Strukturen zurück zu führen.

4. Abschlussbetrachtung

4.1. Erreichtes und Unerreichtes

Erreichtes

In dieser Arbeit wurden die Grundlagen der Darstellung von Modellen in Form von impliziten Feldfunktionen, sowie Methoden für die Polygonisierung dieser Feldfunktionen dargestellt und analysiert. Desweiteren wurden unterschiedliche Typen von Feldfunktionen beschrieben.

Die dargestellten Nachteile der volumenbasierten Algorithmen und die Vorteile des flächenbasierten Marching Triangles Algorithmus bestätigen die Wahl des Marching Triangles Algorithmus für diese Arbeit.
Die Analyse der Beschreibungen von Hiltons und Charlots Marching Triangles Umsetzung lassen die Wahl einer eigenen, auf den Konzepten von Hilton und Charlot basierenden Implementierung ebenfalls sinnvoll erscheinen.

Das im Rahmen dieser Arbeit erstellte Programm zeigt bei verschiedensten Feldfunktionen, dass der flächenbasierte Marching Triangles Algorithmus eine relativ gleichmässige Polygonisierung erreicht und sich somit - wie von Hilton beschrieben - hervorragend für die Polygonisierung impliziter Feldfunktionen eignet.

Vergleiche der Geschwindigkeit bei der Polygonisierung zwischen dem Marching Triangles Algorithmus und dem Marching Cubes Algorithmus wurden nicht vollzogen, da die von Hilton dargestellten Daten in [Hilton 97] plausibel erschienen.

Unerreichtes

Eine allgemein gültige Lösung, die auch Unstetigkeiten in den Feldfunktionen bearbeiten kann, wurde nicht gefunden. Als Lösungsvorschlag wird hier die Verwendung von stetigen Feldfunktionen, im Speziellen die Nutzung des Ray-Stabbing von Kolb und John (siehe Abschnitt 2.2.2) empfohlen.

Die von Hilton vorgeschlagene Betrachtung der Gradienten der zu polygonisierenden Fläche, sowie Charlots Konzept des adaptiven Vorgehens, um auch stark gekrümmte Flächen polygonisieren zu können, konnten leider nicht erfolgreich umgesetzt werden.

Bei dem Überprüfung auf Überlappung mit bereits bestehenden Modell-Teilen wurde aus Gründen der Geschwindigkeit lediglich eine Überprüfung der Punkte des Modells, nicht aber eine Überprüfung der Kanten des Modells implementiert. Gerade im Bezug auf das zuvor beschriebene Konzept des adaptiven Algorithmus ist diese Überprüfung unverzichtbar.

4.2. Verbesserungsvorschläge

Das von Charlot vorgeschlagene adaptive Vorgehen erscheint gerade für stark gekrümmte Flächen von sehr grossem Vorteil zu sein. Einige von Kolb und John in [Kolb 01] vorgestellten Modelle sind für solch ein Vorgehen gerade zu prädestiniert, da sie nur an einigen Stellen sehr kleine, aber dennoch wichtige Details aufweisen. Die momentane Implementierung hat eine feste, durch die Projektionslänge l vorgegebene, Auflösung, die durch ein adaptives Vorgehen umgangen werden könnte.

Bei der Überprüfung auf Überlappung mit bestehenden Modell-Teilen sollte neben der bereits implementierten Überprüfung der Modell-Punkte auch eine Überprüfung der Modell-Kanten, und eventuell auch der Modell-Flächen (der einzelnen Polygone), implementiert werden. Diese Überprüfung ist für ein adaptives Verhalten des Algorithmus zwingend erforderlich.

Um eine noch gleichmässigere Polygonisierung zu erreichen, sollte über die Implementierung zusätzlicher Bedingungen bei den verschiedenen 3D-Delaunay Tests nachgedacht werden. Eine Betrachtung der benachbarten Polygone bietet sich hierfür an.

4.3. Anmerkungen des Verfassers

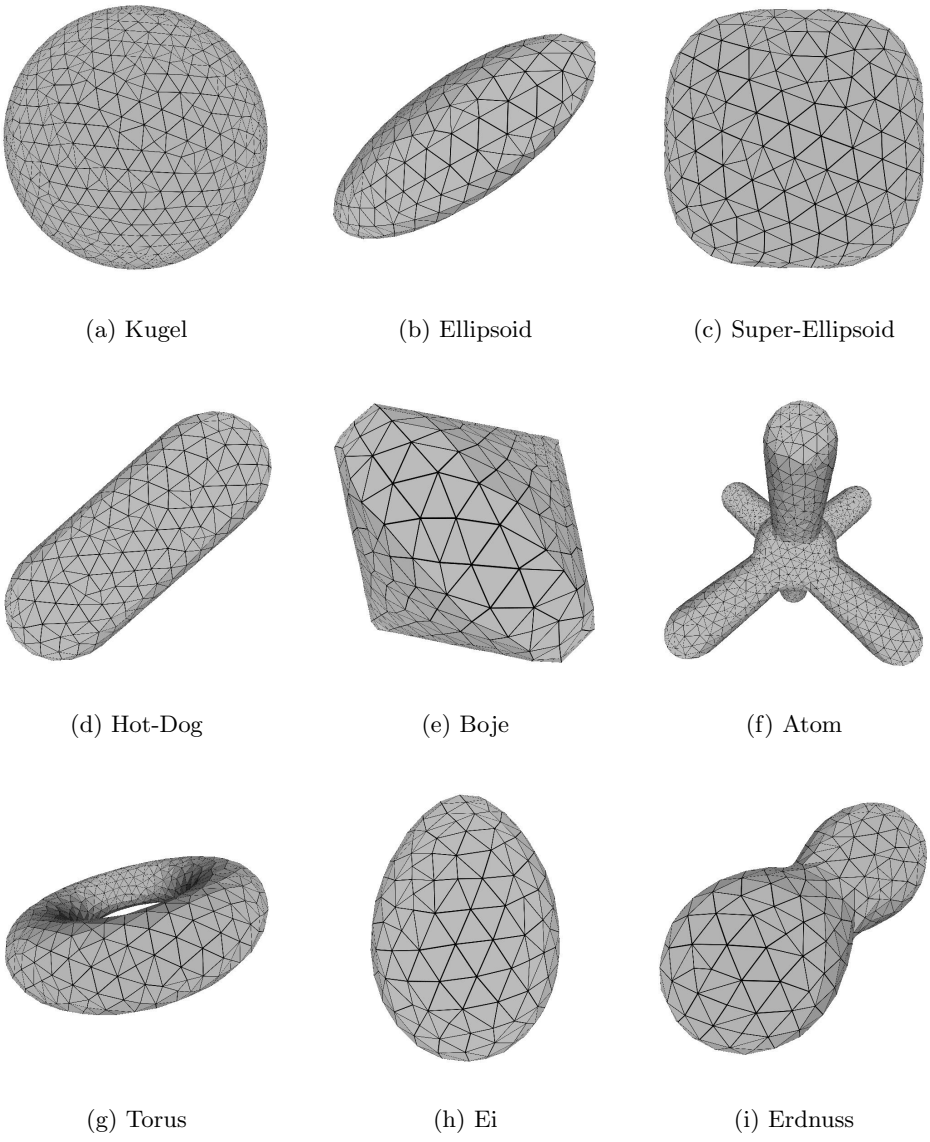
Als ein teilweise sehr grosses Problem bei der Bearbeitung dieser Arbeit hat sich leider die kaum zu erkennende Detail-Liebe einiger Autoren von genutzten Quellen herausgestellt.
So wurden elementare Begriffe verwechselt, wichtige Details ausgelassen oder schlicht und einfach Fehler gemacht, die bei einem Korrekturlesen durch eine dritte, fachkundige Person zu vermeiden gewesen wären. Die Feststellung und Korrektur dieser Ungereimtheiten hat leider sehr viel Zeit in Anspruch genommen.

A. Anhang

In diesem Anhang sind eigene und vergleichende Ergebnisse der zu dieser Arbeit gehörenden Marching Triangles Implementierung, sowie das dazugehörige Programmierer-Handbuch zu finden.

A.1. Ergebnisse

A.1.1. Verschiedene Feldfunktionen



(a) Kugel (b) Ellipsoid (c) Super-Ellipsoid

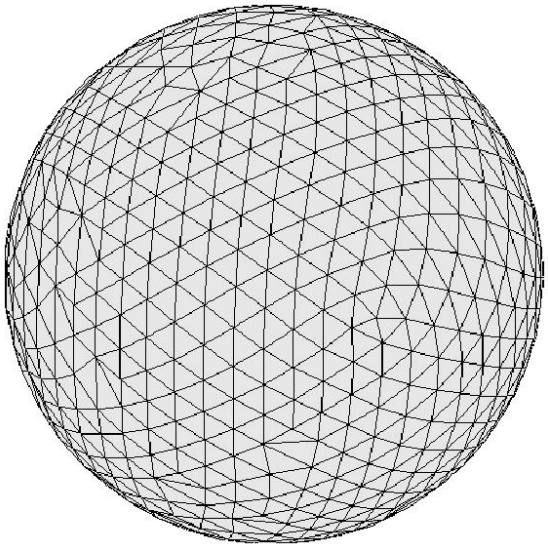
(d) Hot-Dog (e) Boje (f) Atom

(g) Torus (h) Ei (i) Erdnuss

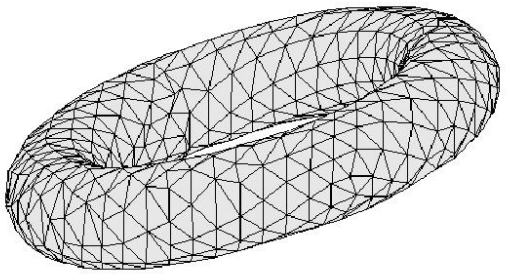
Abbildung A.1.: Ergebnisse: Verschiedene polygonisierte Feldfunktionen

A.1.2. Ergebnisse im direkten Vergleich

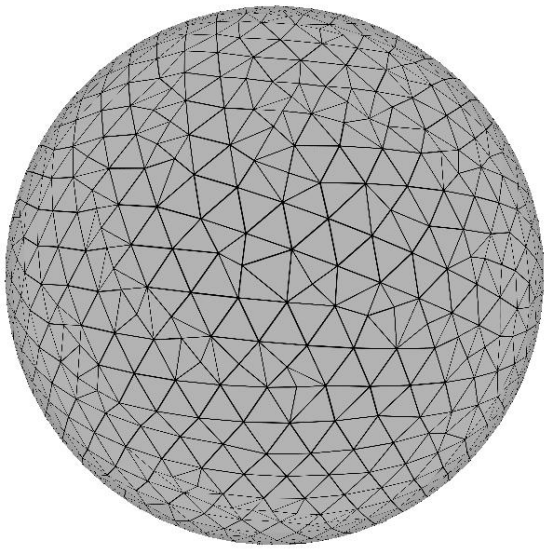
In Abbildung A.2 sind Ergebnisse von polygonisierten Feldfunktionen von Hiltons Implementierung und von der eigenen Implementierung des Marching Triangles Algorithmus dargestellt.



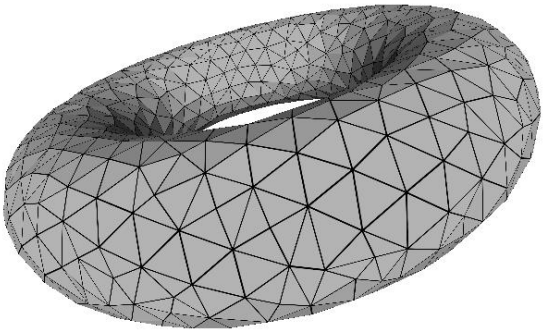
(a) Hiltons Kugel



(b) Hiltons Torus



(c) Kletas Kugel



(d) Kletas Torus

Abbildung A.2.: Ergebnisse: Direkter Vergleich

Die Ergebnisse von Hilton wurden von dessen Website an der University of Surrey [UoS] kopiert.

Da die Parameter des Marching Triangles Algorithmus, die zu Hiltons Ergebnissen geführt haben nicht bekannt waren, wurde durch rein optischen Vergleich versucht, ähnliche Ergebnisse zu erzielen.

Hiltons Kugel weist grössere gleichmässige Flächen auf. Dieses ist durch die unterschiedlichen Datenstrukturen (Liste bzw. Stack) zu erklären. Gerade bei dem Vergleich der Tori ist zu sehen, dass sich die eigene Implementierung durchaus mit der von Hilton messen kann.

A.2. Programmierer-Handbuch

A.2.1. Dateien

Das im Rahmen dieser Arbeit implementierte Programm besteht aus mehreren Dateien. Folgende Dateien gehören zu einer vollständigen Version:

README.TXT	-	Auflistung aller benötigten Dateien
Makefile.irix	-	Makefile für SGI-IRIX
Makefile.linux	-	Makefile für PC-Linux
mt_types.h	-	C-Header: eigene Daten-Typen und -Strukturen
global.h	-	"globaler" Speicher für Parameter-Übergabe der OpenGL Callback-Funktionen
main.c	-	C-Sourcen: Hauptprogramm
callback.h	-	C-Header: OpenGL callback Funktionen
callback.c	-	C-Sourcen: OpenGL callback Funktionen
scene.h	-	C-Header: OpenGL Szene
scene.c	-	C-Sourcen: OpenGL Szene
mt.h	-	C-Header: Marching Triangles Implementation
mt.c	-	C-Sourcen: Marching Triangles Implementation
edge_list.h	-	C-Header: Kanten-Liste
edge_list.c	-	C-Sourcen: Kanten-Liste
point_list.h	-	C-Header: Punkte-Liste
point_list.c	-	C-Sourcen: Punkte-Liste
triangle_list.h	-	C-Header: Dreiecks-Liste
triangle_list.c	-	C-Sourcen: Dreiecks-Liste

Tabelle A.1.: Programm: Quell-Dateien

Um das Programm kompilieren zu können (siehe Abschnitt 3.3.2), werden alle (ausser der Datei README.TXT) in Tabelle A.1 genannten Dateien benötigt. Sollte eine der Dateien fehlen, empfiehlt es sich, die Homepage des Autors unter *http://www.indigo2.de/* aufzusuchen, um von dort eine vollständige Version zu laden.

A.2.2. Programm Exit-Codes

Wenn das Programm beendet wird, wird an die shell, aus der das Programm gestartet wurde, ein Exit-Code übergeben. Der Wert dieses Codes gibt Ausschluss darüber, weshalb das Programm beendet wurde. In Tabelle A.2 sind alle möglichen Exit-Codes und Ihre Bedeutung aufgeführt.

Exit-Code	Wert	Bedeutung
OK	0	Normaler Programmablauf, kein Fehler aufgetreten.
GL_ERROR	1	Fehler bei der Initialisierung des Grafik-Adapters.
OPTION_ERROR	2	Fehler beim Lesen einer Option.
PARAM_ERROR	3	Fehler beim Lesen eines Parameters.

Tabelle A.2.: Programm: Exit-Codes

A.2.3. Programm Ablauf

In diesem Abschnitt wird der grobe Programmablauf beschrieben, um einen Überblick über das Programm zu geben.

Der Programmablauf umfasst folgende Schritte:

- Initialisierung des Marching Triangles Algorithmus.
- Ablauf des Marching Triangles Algorithmus wie in Abschnitt 3.1 beschrieben.
- Initialisierung des Grafik-Adapters.
- Initialisierung der darzustellenden Objekte:
  - Berechnung der grafischen Darstellung der Koordinatenachsen (ausser bei einer Kompilierung im *noaxis* oder *printable* Modus).
  - Umwandlung der vom Marching Triangles Algorithmus generierten Dreiecks-Liste in texturierte Polygone.
    Dabei werden zwei Polygon-Modelle erzeugt: das erste stellt das Modell von aussen, das zweite von innen dar.
- Darstellen der Objekte bis der Benutzer das Programm beendet.

Der Marching Triangles Algorithmus und die Darstellung des generierten Polygon-Modells wurden als sequentielle Schritte implementiert, um jederzeit den Workflow ändern zu können. So ist z.B. die Integration einer Modell-Simplifikation jederzeit möglich.

Die Aufteilung des Polygon-Modells in Aussen- und Innenansicht wurde wegen der besseren Übersichtlichkeit gewählt, da durch die Nutzung der teilweise transparenten Texturen für die Innenansicht ein Blick aus dem Modell heraus möglich ist.

A.2.4. Datenfluss

Beim zuvor beschriebenen Programm Ablauf, wird durch den Marching Triangles Algorithmus eine Feldfunktion $f(P)$ mit $P = (x, y, z)$ der Form

```
Coord f(Coord x, Coord y, Coord z)
```

in eine Dreiecks-Liste mit Elementen vom Typ *TriangleListElement* umgewandelt[1].

Diese Dreiecks-Liste wird während der Initialisierung der darzustellenden Objekte in zwei OpenGL Objekte - eins für die Aussenansicht, eins für die Innenansicht - umgewandelt. Diese Objekte werden in einer OpenGL Display-List verwaltet, und durch je eine ID in dieser Liste beschrieben.

A.2.5. Datenstrukturen

Für das im Rahmen dieser Arbeit erstellte Programm wurden diverse, der Aufgabenstellung angepasste, Daten-Typen, -Strukturen und -Listen benötigt. Der Beschreibung dieser Datenstrukturen ist dieser Abschnitt gewidmet.
Bei der Implementierung der einzelnen Typen, Strukturen und Listen wurden vor allem auf die Wartbarkeit geachtet.

[1] Die Beschreibung der verschiedenen Datentypen erfolgt in Abschnitt A.2.5.

Typen

Es wurden verschiedene eigene Typen implementiert. Diese wurden vor allem aus Gründen der Wartbarkeit umgesetzt. So ist z.B. die Änderung der Genauigkeit bei der Berechnung von Fliesskommazahlen durch einfaches Ändern des passenden Typen in der definierenden Datei *mt_types.h* für das ganze Programm machbar.

Eine Koordinate ist eine Zahl, die die Lage eines Punktes im Raum bestimmt.

```
typedef float Coord;
```

Ein Punkt ist ein beliebiger Punkt im 3D-Raum, beschrieben durch drei Koordinaten.

```
typedef Coord Point3D[POINT_3D_SIZE];
```

Ein Vektor ist eine Grösse, die durch Angriffspunkt und Richtung beschrieben wird. Wird hier durch drei Koordinaten beschrieben.

```
typedef Coord Vector3D[VECTOR_3D_SIZE];
```

Strukturen

Passend zur Aufgabenstellung wurden Daten-Strukturen für die Speicherung von Dreiecken und Kanten benötigt.

Ein Dreieck ist ein Polygon, dass aus drei Punkten i, j, k besteht. Dabei sind die drei Punkte von oben (ausserhalb des Modells) betrachtet mathematisch positiv (also entgegen des Uhrzeigersinnes) angeordnet.

```
/* triangle, orientation of points is counterclockwise (viewed from top) */
typedef struct
{
  Point3D i;
  Point3D j;
  Point3D k;
} Triangle;
```

Eine Kante ist eine Seite eines Dreieckes. Im Sinne des Marching Triangles Algorithmus besteht eine Kante aus den beiden Dreiecks-Punkten i und j. Für die Berechnung eines neuen Punktes xp wird allerdings auch der dritte Punkt k des Dreiecke benötigt. Zusätzlich zu diesen Punkten wird ein Zähler für die Speicherung des Alters der Kante, sowie eine Kennung zum Erkennen eines erfolglosen Stack-Durchlaufes[2] benötigt.

```
/* edge */
typedef struct
{
  int id;
  int age;
  Point3D i;
  Point3D j;
  Point3D k;
} Edge;
```

[2]Unter einem "erfolglosen Stack-Durchlauf" ist folgendes zu verstehen: Es wurden alle auf dem Stack befindlichen Elemente nacheinander gelesen, und - da sie nicht erfolgreich bearbeitet werden konnten - an das Ende des Stacks verschoben. Dieser Mechanismus wurde eingeführt, um ein frühzeitiges Terminieren des Algorithmus gewährleisten zu können.

Listenelemente

Für die Verwaltung der Daten (Punkte, Dreiecke und Kanten) wurden doppelt verkettete Listen gewählt.

Punkt:

```
/* List element consists of Point, and two pointers
   to next / previous element */
typedef struct PointListElement_
{
  Point3D point;
  struct PointListElement_ *next;
  struct PointListElement_ *previous;
} PointListElement;
```

Dreieck:

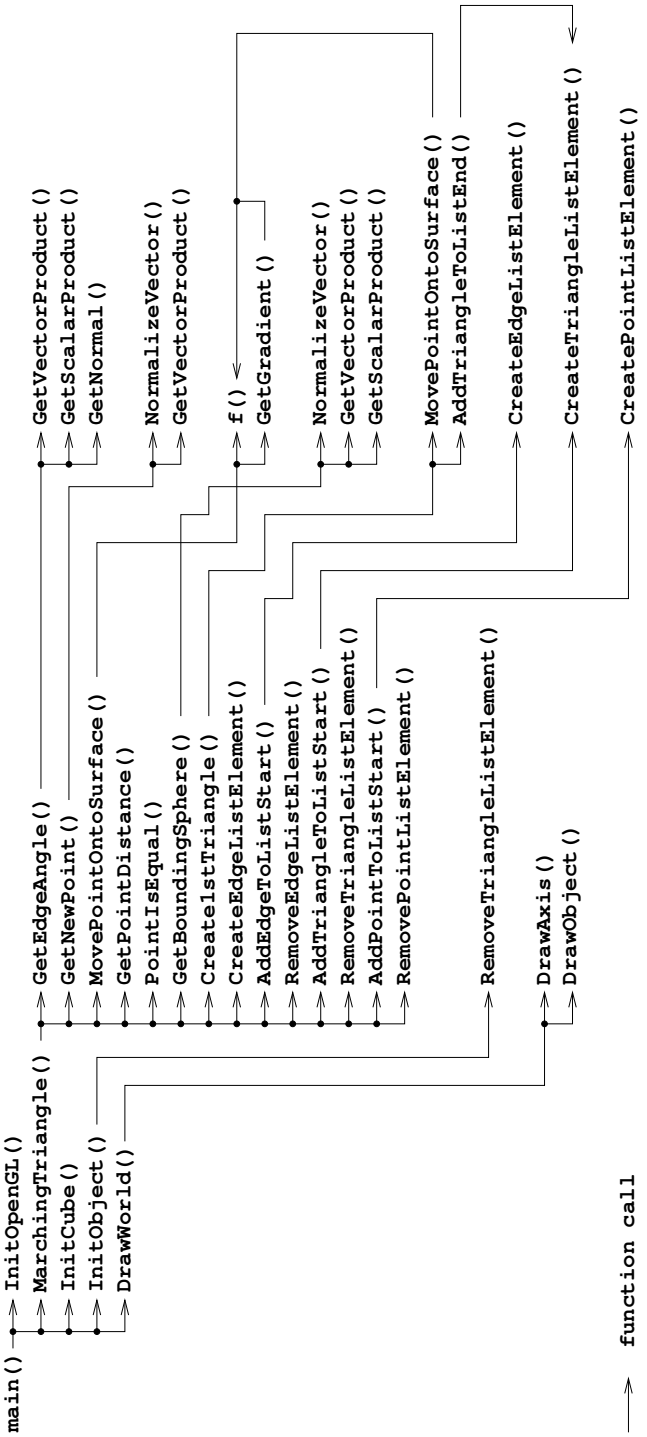
```
/* List element consists of triangle, and two pointers
   to next / previous element */
typedef struct TriangleListElement_
{
  Triangle triangle;
  struct TriangleListElement_ *next;
  struct TriangleListElement_ *previous;
} TriangleListElement;
```

Bei der Kanten-Liste wurde neben der doppelten Verkettung eine weitere doppelte Verkettung implementiert, um die lokalen Abhängigkeiten (linke bzw. rechte Kante) abbilden zu können.

```
/* List element consists of edge,
   two pointers to next / previous element (for list) and
   two pointers to left / right edge for local dependencies */
typedef struct EdgeListElement_
{
  Edge edge;
  struct EdgeListElement_ *next;
  struct EdgeListElement_ *previous;
  struct EdgeListElement_ *left;
  struct EdgeListElement_ *right;
} EdgeListElement;
```

A.2.6. Programm Organisations Plan

Die folgende Abbildung zeigt grafisch, welche Funktionen welche anderen Funktionen aufrufen. Dabei wurde auf die Darstellung der OpenGL Callback-Funktionen verzichtet, da diese einem Standard-OpenGL Programm entsprechen.



Die Funktion $f()$ steht für die jeweils gültige Feldfunktion. Für verfügbare Feldfunktionen siehe Tabelle 3.3.

Literaturverzeichnis

[Abramowski 91] S.Abramowski; H.Müller: *Geometrisches Modellieren*, BI Wissenschaftsverlag, ISBN 3-441-14491-2, 1991

[Bloomenthal 88] J.Bloomenthal: *Polygonization of implizit surfaces*, Computer Aided Geometric Design 5, Elsevier Science Publishers B.V., 1988

[Bloomenthal 97] J.Bloomenthal: *Introduction to Implicit Surfaces*, Morgan Kaufman, ISBN 1-55860-233-X, 1997

[Bronstein] I.N.Bronstein; K.A.Semendjajew: *Taschenbuch der Mathematik*, B.G.Teubner Verlagsgesellschaft, ISBN 3-8154-2000-8, 1991

[Charlot 98] J.Charlot; F.Schmitt; M.Perrin: *Simplified triangulated meshes for reliable representation of folded surfaces by the POLYPLI kinematic modeller*, GOCAD ENSG Conference, 3D Modeling of Natural Objects: A Challange for the 2000's, 1998

[FutureTech] Future Technology Research Index: *SGI Graphics Performance Comparison Tables*, http://www.futuretech.vuurwerk.nl/gfxtables.html

[Hilton 97] A.Hilton; A.J.Stoddart; J.Illingworth; T.Windeatt: *Marching Triangles: Delaunay Implicit Surface Triangulation*, T.CVSSP Technical Report 01, 1997

[Illiad] Illiad: *Cartoon for Jun 15, 2000*, http://www.userfriendly.org/, 2000

[John 01] L.John: *Performance-Optimierung der Echtzeitdarstellung komplexer dreidimensionaler Szenen durch Modellsimplifikation und Level Of Detail*, unveröffentlichte Diplomarbeit, Fachhochschule Wedel, 2001

[Kolb 01] A.Kolb; L.John: *Volumetric Model Repair for Virtual Reality Applications*; EUROGRAPHICS, 2001

[Lorensen 87] W.E.Lorensen; H.E.Cline: *Marching Cubes: A high resolution 3D surface construction algorithm*, ACM SIGGRAPH, Volume 21, Number 4, 1987

[Nooruddin 99] F.S.Nooruddin; G.Turk: *Simplification and Repair of Polygonal Models Using Volumetric Techniques*, Technical Report, Giorgia Institute of Technology, 1999

[Schumann 00] H.Schumann; W.Müller: *Visualisierung*, Springer Verlag, ISBN 3-540-64944-1, 2000

[SGI] Silicon Graphics Inc: *SGI - Silicon Graphics Onyx2: Tech Specs*, http://www.sgi.com/products/remanufactured/onxy2/tech_specs.html

[UoS] *Homepage der University of Surrey*, http://www.surrey.ac.uk/

[Wyvill 89] B.Wyvill; G.Wyvill: *Field functions for implicit surfaces*, The Visual Computer (1989)5, Springer-Verlag, 1989

Weitere Quellen

In diesem Abschnitt werden die nicht in das Literaturverzeichnis passenden Quellen aufgeführt, die aber mindestens genauso viel zu dieser Arbeit beigetragen haben wie die dort aufgeführten.

LaTeX

M.Goosens, F.Mittelbach, A.Samarin: *Der LaTeX Begleiter*, Addison-Wesley, ISBN 3-8273-1689-8, 2000

OpenGL

M.Woo, J.Neider, T.Davis, D.Shreiner: *Open GL Programming Guide, Third Edition*, Addison-Wesley, ISBN 0-201-60458-2, 2001

Antworten auf alle möglichen (mehr oder weniger) fachlichen Fragen gaben:

- Prof. Dr. Andreas Kolb, Fachhochschule Wedel
- Dipl.-Ing.(FH) Martin "Herbert" Dietze, University of Buckingham
- Prof. Dr. Uwe Schmidt, Fachhochschule Wedel

Rechtschreibung

Duden Band 1, Rechtschreibung der deutschen Sprache und der Fremdwörter, 18. Auflage, Bibliographisches Institut, Dudenverlag, ISBN 3-411-00901-2, 1980

Webster's New Encyclopedic Dictionary, Black Dog & Leventhal Publishers Inc., ISBN 1-884822-25-8, 1996

Korrekturlehser:

- Jakob Dittmar
- Anja Kleta

seid bedankt...

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe.

Hamburg, den 11.9.2002

(Henry G. Kleta)