Diplomarbeit
Fachhochschule Wedel

Tesselierung und Komprimierung von dreidimensionalen Geometriedaten

Martin Wendt
Fachrichtung: Technische Informatik
Referent: Dipl.-Ing. P. Pook-Haffmans

August 1992

[Konvertiert im November 2002; martin@wwWendt.de]

INHALTSVERZEICHNIS

I.Einleitung

Diese Diplomarbeit entstand im Zeitraum Dezember 1991 bis August 1992 bei der Firma WISE software GmbH in Lübeck.

Thema war die Erstellung eines Programms, das Daten eines 3D-Digitalisierers einliest, in ein Flächenmodell konvertiert und reduziert. Die Farbinformation sollte extrahiert und getrennt als Textur abgespeichert werden.

Betreuender Dozent war Dipl.Ing. Peter Pook-Haffmans von der Fachhochschule Wedel.
Für ihre Unterstützung danke ich außerdem besonders Michael Wise, Michael Krüger und Anke.

Ich versichere, daß ich die vorliegende Arbeit selbständig erstellt habe. Alle benutzten Quellen habe ich entsprechend gekennzeichnet.

Wedel, August 1992

Martin Wendt

II. Ausgangsposition

A.3D-Digitizersystem Spex3D

Spex3D dient zum Digitalisieren der Geometrie und Textur von dreidimensionalen Objekten. Mit seiner Hilfe ist es möglich, die Koordinaten eines Objektes in einem maschinenlesbaren Format zu gewinnen, ohne daß man es mit einem CAD-Programm nachmodellieren muß.

Der Digitizer besteht aus mehreren Hardwarekomponenten:
- Meßtisch mit schrittmotorgesteuerter Linear- und Rotationsachse
- SuperVHS Video Kamera
- Screen Machine™ Graphikkarte zum Mischen der Videosignale von Kamera und Computer
- Eine oder zwei Laserdioden mit Fächerlinsen
- PC-Computer mit Hardware-Interface zur Ansteuerung dieser Elemente

Die zugehörige Software läuft unter Windows und steuert den interaktiven Digitalisiervorgang.

1.Funktionsweise

Das aufzunehmende Objekt wird auf den Meßtisch gestellt. Eine Laserdiode projiziert einen vertikalen Fächer darauf, so daß auf dem Objekt eine schmale senkrechte Linie erscheint. In einem Winkel von ca. 30° neben dem Laser ist die Kamera aufgebaut, die das Bild vom Objekt mit der darauf abgebildeten Linie zum Rechner überträgt.

Das grundlegende Problem bei der Digitalisierung ist, daß man dreidimensionale Koordinaten erzeugen möchte, die Kamera aber nur ein zweidimensionales Bild liefert. Der Lösungsansatz heißt 'Lasertriangulation': Betrachtet man die Linie, die der Laser auf der Objektoberfläche hinterläßt schräg von der Seite, so verläuft diese Linie nicht mehr senkrecht, sondern gekrümmt.

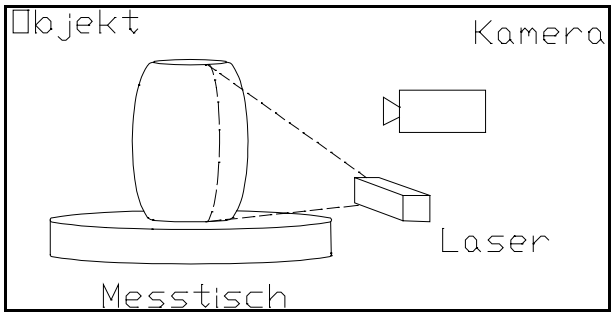


Abb. II.1 Spex3D, Funktionsprinzip

Wenn man z.B. einen Ball als Objekt wählt und sich der Laser links von der Kamera befindet, so liefert die Kamera ein Bild mit einer gekrümmten (nach links ausgebeulten) Laserlinie. Je geringer die Entfernung zwischen Objektoberfläche und Laser ist, desto weiter wird sich die Projektionslinie im Kamerabild nach links verschieben.

Das Kamerabild wird in den Rechner kopiert ('capture') und dort mittels Bildverarbeitung ausgewertet. Ziel ist es, die Spur die der Laser auf dem Objekt hinterläßt zu finden[1], und als Koordinatenvektor (Polygonzug) abzuspeichern. Da die Kamera eine begrenzte Auflösung hat (in unserem Fall 680 x 512 Pixel, Breite x Höhe), ist dadurch der Bereich der Bildkoordinaten vorgegeben. Der gefundene Polygonzug hat also zweidimensionale, diskrete Koordinaten.

Wenn man nun die Positionen und Ausrichtungen von Kamera, Laser und Meßtisch kennt, kann man den Zusammenhang zwischen Bildkoordinaten (X_B, Y_B) und Objektkoordinaten (X_O, Y_O, Z_O) durch eine Transformationsmatrix T ausdrüken. Diese Matrix wird durch Kalibrierung mit einem bekannten Objekt gewonnen[2]. Wenn man jeden Punkt des Polygonzuges mit T multipliziert, erhält man die Raumkoordinaten des Polygonzuges, den der Laser auf dem Objekt hinterlassen hat.

Ein Objekt wird digitalisiert, indem eine Anzahl von Polygonzügen auf die oben beschriebene Weise aufgenommen werden. Zwischen den einzelnen Aufnahmen wird das Objekt bewegt (z.B. durch Rotation des Meßtisches um 1°). Am Ende erhält man eine Menge von Polygonzügen, die nebeneinander auf dem Objekt liegen und so seine Oberfläche beschreiben.

Diese Polygonzüge werden zusammen mit Informationen über die Bewegungen des Meßtisches in einem komprimierten Binärformat[3] gespeichert.

---

[1] Man bezeichnet diesen Vorgang in der Bildverarbeitung auch als 'Edge Detection'

[2] Die Kalibriermatrix wird bei jeder Tischbewegung durch Konkatenation mit einer entsprechenden Transformationsmatrix aktuell gehalten

[3] Dieses '.CEF'-Format wird in Kapitel III näher beschrieben

Da normalerweise nicht nur die Form, sondern auch die Farbe eines Objektes von Interesse ist, kann diese in einem zweiten, automatisch ablaufenden Arbeitsgang ergänzt werden. Das geschieht auf folgende Weise:

1. Der Meßtisch samt Objekt wird wieder in die Ausgangsposition gefahren.
2. Der Laser wird abgeschaltet, das Objekt durch geeignete Lichtquellen gleichmäßig beleuchtet.
3. Die bereits im ersten Durchlauf erzeugte Datei wird benutzt, um noch einmal die einzelnen Tischpositionen anzufahren, an denen ein Polygonzug aufgenommen wurde.
4. An jeder Position wird das Kamerabild eingelesen. Da die Koordinaten des Polygonzuges in Bildkoordinaten gespeichert wurden, können jetzt an den entsprechenden Punkten die Farbwerte ausgelesen werden.

Nach Abschluß eines erfolgreichen Digitalisiervorgangs liegt eine Datei mit einer Folge von Polygonzügen vor, die die Oberfläche beschreiben. Jeder dieser Polygonzüge besteht aus einer Folge von Punkten ('Vertizes') mit Koordinaten und Farben:

$$V\ (X_B, Y_B, R, G, B)^4$$

Außerdem enthält die Datei Transformationsmatrizen, die die Bildkoordinaten in die Objektkoordinaten überführen. Diese Datei bildet den Ausgangspunkt für die Nachbearbeitung, die Thema meiner Diplomarbeit ist. Ich werde diese Ausgangsdateien im weiteren, nach ihrer Dateinamenserweiterung, als 'CEF-Dateien' bezeichnen.

2.Ziele der Datennachbearbeitung

Das Ausgabeformat von Spex3D ist auf geringen Speicherbedarf optimiert, was sicher sinnvoll ist, wenn man die Menge der erzeugten Daten bedenkt:

> Ein vollständig digitalisierter Körper (360 Schnitte mit Rotation um jeweils 1°, durchschnittliche Anzahl von Vertizes pro Schnitt: 450) wird durch ungefähr 360 x 450 ≈ 150.000 Vertizes beschrieben.

Sämtliche Informationen, die während der Digitalisierung anfallen, werden deshalb in einem komprimierten Format abgespeichert.
Bevor man sich die Resultate mit Hilfe eines CAD- oder Renderingprogramms auf dem Bildschirm betrachten kann, sind noch einige Zwischenschritte nötig:

1. Die Bildschirmkoordinaten (X_B, Y_B) müssen in Objektkoordinaten (X_O, Y_O, Z_O) transformiert werden.
2. Die Vertexfarben, die als Rot-, Grün- und Blauanteile mit einem Wertebereich von 0 bis

[4] Der Index 'B' steht für Bildschirmkoordinaten. R, G und B geben den Rot-, Grün- und Blauanteil des Punktes an.

255 vorliegen, müssen in die Farbdarstellung des nachfolgenden Programms überführt werden. ARENA erwartet z.B. die RGB-Anteile im Fließkommaformat auf 1,0 normalisiert (Wertebereich: 0,0 bis 1,0).

3. Ein Renderingprogramm benötigt für eine ansprechende dreidimensionale Darstellung[5] die Beschreibung eines Körpers oder seiner Oberfläche. Spex3D erzeugt aber nur eine Menge von **Punkten**, die auf dieser Oberfläche liegen. Eine wichtige Aufgabe der Datennachbearbeitung ist also die Generierung einer Fläche aus einer Punktmenge.
4. Schließlich muß das Ausgabeformat eingehalten werden. Ich habe mich auf zwei Formate beschränkt: Das ARENA Format (ZOF) und das AutoCad Format (DXF). Beides sind Ascii-Formate, mit allerdings sehr unterschiedlicher Struktur.

Als Repräsentationsform für die Oberfläche habe ich ein Netz mit dreiekigen Maschen gewählt (**Triset**).
Die oben genannte maximale Zahl von Vertizes (150.000) führt zu einem Triset mit ca. 300.000 Faces[6]. Um diese große Anzahl von Flächen zu rendern, benötigt auch ein schneller Rechner einige Minuten. Eine weitere Anforderung an die Datennachbearbeitung ist also

5. eine Reduktion der Datenmenge bei möglichst geringem Verlust von Genauigkeit und 'Echtheit'.

Dieses letzte Ziel bildet den Schwerpunkt meiner Diplomarbeit.

B.Arena

ARENA steht für '**A**dvanced 3D-Modelling and **Ren**dering **A**pplication'. Es handelt sich hierbei um ein 3D-Renderingprogramm zur wirklichkeitsnahen Darstellung von Geometriedaten. Es bietet folgende Funktionalität:

1. Einlesen von Geometriedaten verschiedener Formate (u.a. DXF und das firmeninterne ZOF-Format)
2. Interaktives Transformieren der Objekte (Zoomen, Skalieren, Rotieren, etc)
3. Zuweisen von Oberflächenattributen (Farbe, Reflektionsgrad, Materialeigenschaften, usw.), sowie Texturen[7]
4. Definitionen von verschiedenen Beleuchtungsarten (Spotlight, Parallel, Ambient, ...), sowie Kamerapositionen

---

[5] Dazu gehört das Unterdrücken von versteckten Linien und Flächen, das Beleuchten mit (evtl. farbigen) Lichtquellen, Schattieren, Aufsetzen von Glanzlichtern, etc.

[6] Ein Triset hat näherungsweise doppelt so viele Maschen wie Vertizes. Die Details beschreibe ich in Kapitel IV.

[7] Diese Fähigkeit (Texture Mapping) hat für die Datenreduktion noch eine große Bedeutung (vgl. Kapitel V.)

5. Rendern der 3D-Szenen mittels verschiedener Schattierungstechniken (Flat-, Gouraud- und Phong-Shading)

ARENA bildet die Senke für die nachbearbeiteten Spex3D-Daten, oder andersherum formuliert: Das von mir zu erstellende Programm hat eine Filterfunktion zwischen Spex3D und ARENA.

C.Graphikstandard Phigs

Phigs (**P**rogrammer's **H**ierarchical **I**nteractive **G**raphics **S**ystem) ist ein nach ANSI und ISO standardisiertes 3D-Grafiksystem, das ähnlich einer Grafikbibliothek benutzt werden kann.
WISE software hat eine Phigs-Implementation für DOS (Z-Phigs) und Windows (Z-Phigs für Windows) entwickelt, mit Bindings für C und Pascal. Das Programm ARENA basiert auf Z-Phigs für Windows.
Ich benötige keine Phigs-Routinen, verwende aber einige der im Standard definierten Typen, um einheitliche Datenformate zu erhalten.
Für eine ausführliche Beschreibung des Phigs-Standard verweise ich auf [Hopgood91] und [Hewitt91].

D.Entwicklungsumgebung, Systemanforderungen

1.Hardware

Die anfallenden Datenmengen und komplexen Fließkommaberechnungen stellen Mindestanforderungen an die Hardware. Da unter Microsoft Windows™ gearbeitet wird ist allein deshalb schon eine gewisse Rechenleistung sinnvoll.
Für ein vernünftiges Arbeiten sollte eine i386 CPU mit Coprozessor bzw. eine i486 CPU vorhanden sein. 4 M Bytes, besser noch 8 MB oder mehr sollte an Speicher verfügbar sein. Die Festplattenkapazität sollte mindestens 100 MB betragen. Farbgrafik mit mindestens 640x480 Punkten ist Voraussetzung.

2.Betriebssystem

Als Betriebssystem wurde MS-Dos 5.0 und MS-Windows 3.1 verwendet. Windows wird im 386er Enhanced Mode betrieben.
Mindestanforderungen sind MS-DOS ab Version 3.0 und MS-Windows ab Version 3.0.

3.Software

Für die Entwicklung wurde folgende Software benutzt:

1. Borland C++ 3.0 & Framework Applications
2. Microsoft Software Development Kit 3.0 (SDK)
3. Spex3D Software für Digitizer
4. ARENA Rendering Software

E.Vorgehensweise

Zunächst werde ich die Aufgabenstellung genauer definieren und in Teilaufgaben gliedern. Die Teilaufgaben werde ich dann analysieren, einen Lösungsansatz entwickeln und diesen schrittweise verfeinern, bis am Ende ein Algorithmus steht.
Ablaufpläne, Organisationspläne und die eigentliche Programmierung dieser Algorithmen soll getrennt im Anhang als Listing erscheinen. Die Dokumentation der Programmteile, die nicht primär der Problemlösung dienen (also z.B. Windows-Verwaltung, Speicherverwaltung, Benutzerführung, Debug-Optionen, etc.) erfolgt im Quellcode als Inline-Dokumentation.

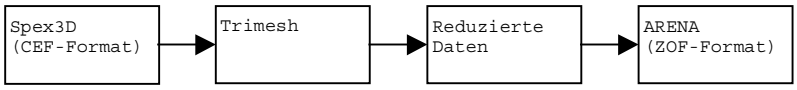
1.Aufgabenstellung

Es soll ein Programm geschrieben werden, das die Daten des 3D-Digitizers Spex3D einliest und für eine Weiterverarbeitung aufbereitet.

Der Arbeitstitel für dieses Programm ist **Comp3D**. Dieses Programm soll im einzelnen folgende Funktionalität haben:

1. Einlesen der Spex3D-Daten
2. Erzeugen eines Trimesh aus diesen Daten
3. Reduktion der Geometrieinformation
4. Schreiben der Daten (primäres Ziel: ARENA)

Parametereinstellungen, Auswahl von Dateinamen, etc. sollen interaktiv (z.B. über Dialogboxen) erfolgen.
Der grobe Datenfluß ist:



a.Einlesen der Daten

Die Eingabedaten werden von Spex3D binär kodiert abgespeichert. Nach dem Einlesen sollen die Daten in einem leicht weiterzuverarbeitenden Format vorliegen.
Als Sonderfunktion soll eine einfache Filterung möglich sein (Ändern der Farben, Verwerfen von Punkten, die außerhalb eines bestimmten Wertebereichs liegen).
Als zusätzliche Option wird das Einlesen von Trisets im ZOF-Format unterstützt. Dadurch wird auch die Reduktion von Datensätzen möglich, die nicht von Spex3D erzeugt wurden.
Diese Aufgabe wird in Kapitel III. behandelt.

b.Erzeugen eines Trimesh (Tesselierung)

Die eingelesenen 3D-Polygonzüge sollen in ein Trimesh konvertiert werden. Die Größe des Trimesh soll nur durch den verfügbaren Speicher begrenzt sein.
Ich behandele dieses Thema in Kapitel IV.

c.Datenreduktion

Das eingelesene Trimesh soll soweit wie möglich verkleinert werden, ohne daß sich sein Aussehen (wenn es gerendert wird) dabei ändert.
Der grundlegende Ansatz dazu soll das **Two-Part Mapping** sein:
In einem ersten Schritt wird die Farbe der Objektoberfläche (Textur, also die Farbinformation des eingelesenen Trimesh) in Form einer Bitmap abgespeichert. Anschließend können Punkte des Trisets entfernt werden (Reduktion der Geometrieinformationen). Auf das nun viel grobmaschiger gewordene Triset kann ARENA dann die feine Farbauflösung des Originalobjektes projizieren.
Für die Reduktion der Daten habe ich zwei Algorithmen entwickelt:
der erste, 'direkte' Weg löscht alle Punkte, sofern es die Restriktionen (maximaler Fehler zum Original, ...) nicht verletzt.
Die zweite Methode ist wesentlich flexibler und basiert auf **Simulated Annealing**.
Ich gehe auf diese Punkte in Kapitel V. und VI. ein.

d.Datenausgabe

Das reduzierte Triset soll in einem für ARENA lesbaren Format (ZOF) gespeichert werden. Auch die Ausgabe im AutoCad Format (DXF) soll unterstützt werden.
Dieses Thema wird in Kapitel VII. besprochen.

e.Datenflußplan

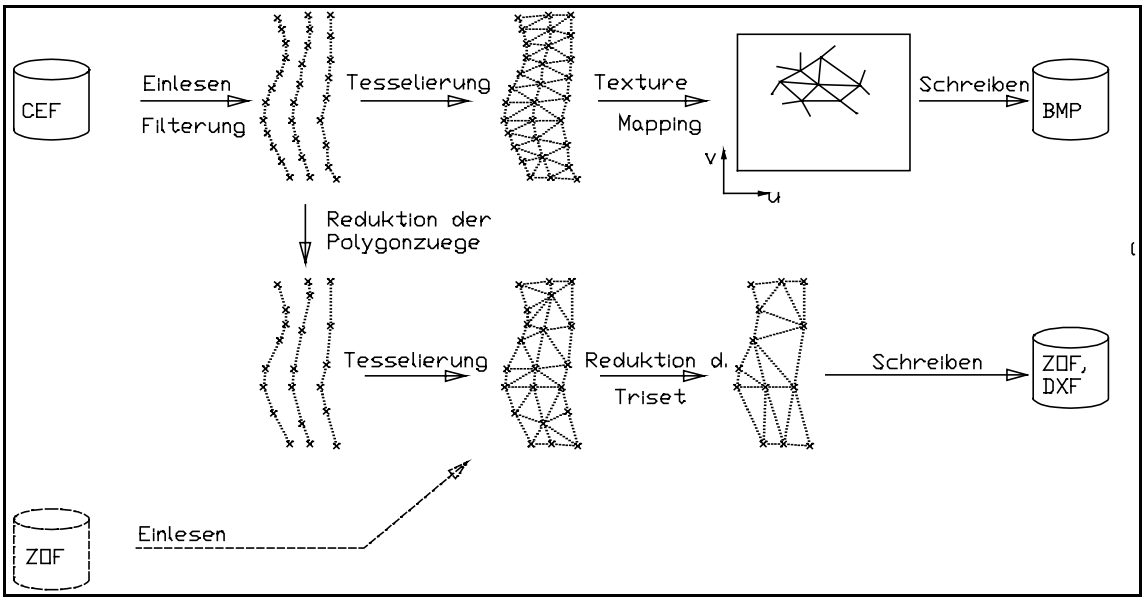


Abb. II.2 Comp3D: Datenflußplan

2 zeigt den Datenfluß des zu erstellenden Programms.

Wichtig ist, daß das Texture Mapping vor der Reduktion stattfindet, damit keine Farbinformationen verlorengehen. Weil die Farben zwischen den Punkten interpoliert werden sollen, werden immer alle Eckpunkte eines Face auf einmal in die Textur projiziert. Deshalb wird vorher tesseliert.

Die Funktion 'Reduktion der Polygonzüge' wurde als Option eingeführt, um große Datensätze durch eine Vorfilterung zu verkleinern.

III.Einlesen der Daten

A.Dateiformat CEF

CEF (**C**ompressed **E**dge **F**ormat) ist ein internes Datenformat der Firma WISE software. Es wurde entworfen um die Polygonzüge (Polylines), die der 3D-Digitalisierer Spex3D erzeugt, platzsparend zu archivieren. Der geringe Speicherbedarf wird durch zwei Konzepte erreicht:

1. Die Punkte der Polylines werden in Bildschirmkoordinaten, wie sie die Kamera liefert, abgespeichert. Das hat gegenüber der Speicherung von 3D-Modellkoordinaten mehrere Vorteile:
   a. Die Z-Koordinate ist immer 0 und braucht daher nicht gespeichert zu werden.
   b. X und Y sind vom Typ Integer (jeweils 2 Byte) und nicht vom Typ Fließkomma mit doppelter Genauigkeit (jeweils 8 Byte).
   c. Die RGB-Farben der Polylinepunkte (Vertizes) werden als Byte-Tripel gespeichert. (Phigs arbeitet mit Fließkommawerten doppelter Genauigkeit zwischen 0,0 und 1,0).
2. Die Daten werden komprimiert. Dabei wird ausgenutzt, daß die Polygonzüge stetig sind, das heißt, Koordinaten und Farben von zwei aufeinanderfolgenden Punkten unterscheiden sich in der Regel nur wenig. Es reicht also in den meisten Fällen aus, für einen Punkt nur die Differenz zum vorigen Punkt anzugeben.

Der reduzierte Speicherbedarf wird durch einen erhöhten Rechenaufwand beim Lesen der Daten erkauft:
Aus den zweidimensionalen Bildkoordinaten müssen die dreidimensionalen Modellkoordinaten erzeugt werden, die das digitalisierte Objekt beschreiben. Um das möglich zu machen, werden zusätzlich zu jedem Polygonzug Transformationsmatrizen gespeichert, mit deren Hilfe die Bildschirmkoordinaten in Modellkoordinaten transformiert werden können.

B.Aufbau der CEF-Dateien

1.Struktur

Es handelt sich hier um ein binäres Format, das aus einer Folge von Blöcken gleichen Aufbaus besteht:

Aufbau eines Blocks:

Key	Länge	Daten	Füll

Key: Eindeutiger Schlüssel der den Blocktyp bestimmt (Typ unsigned int, 2 Byte)
Länge: Länge des Blocks in Paragraphen (16 Byte Abschnitte) (Typ: long int, 4 Byte)
Daten: abhängig vom Blocktyp

Füll: Auffüllen des Blocks bis zur nächsten Paragraphengrenze mit Nullen.

Aufbau einer Datei:

Tag und Versionsnummer
Headerblock
Globaler Kommentar*
Globale Transformationen**
{ Geometriedaten }
End-of-file Block

* : kann entfallen

** : kann mehrfach wiederholt werden

{..} : besteht aus mehreren Blöcken

Aufbau eines Geometriedatenblocks:

Lokale Transformationen
Lokaler Kommentar*
Polygonzugdaten**

* : kann entfallen

** : kann mehrfach wiederholt werden

2.Die einzelnen Elemente

1. Der Tag dient zur Identifikation von CEF-Dateien.
2. Im Headerblock sind weitergehende Informationen gespeichert (z.B. die Auflösung der verwendeten Videokarte).
3. Globale und lokale Kommentarblöcke enthalten ASCII Zeichenketten.
4. Globale Transformation(en) werden einmalig am Anfang der Datei angegeben. Hier werden die Matrizen für die Kameratransformationen abgelegt, mit deren Hilfe die Bildkoordinaten einer Kamera in

dreidimensionale Objektkoordinaten umgerechnet werden können. Diese homogenen Matrizen ("**Camera Transformations**") vom Grad 4 werden von Spex-3D während der Kalibrierung erzeugt.

5. Lokale Transformationen geben die Bewegung des Objektes während des Digitalisiervorganges wieder und werden deswegen vor fast jedem Polygonzug abgelegt. Konkateniert man diese "**Edge Transformation**" mit der Camera Transformation, erhält man eine Matrix, welche die Bildkoordinaten der Polygonzüge in dreidimensionale Weltkoordinaten überführt.
   Die Edge Transformation kann auf verschiedene Arten angegeben werden:
   1. relativ zur letzten Bewegung des Objektes, oder
   2. absolut zur Ausgangsposition des Objektes.

   Wie die Matrix zu interpretieren ist, wird in einem zusätzlichen Feld angegeben.
6. Datenblöcke enthalten die Vertexdaten für jeweils einen Polygonzug in einem komprimierten Binärformat. Auf die Dekodierung der Daten gehe ich weiter unten ein.
7. Das Ende jeder CEF-Datei wird durch einen End-Of-File Block gekennzeichnet.

C.Format der Polygonzugdaten

An dieser Stelle soll das Format der eigentlichen Vertexdaten beschrieben werden, also der Inhalt des Geometriedatenblocks.
Wie bereits oben erwähnt, enthält ein Datenblock die Koordinaten und Farben aller zu **einem** Polygonzug gehörenden Punkte. Diese Vertizes werden sequentiell abgelegt, wie der Bildverarbeitungsalgorithmus von Spex-3D sie liefert. Zur Erinnerung noch einmal kurz der Ablauf[8]: Das Objekt samt der darauf befindlichen Laser-Linie wird von einer CCD-Kamera aufgenommen. Da die Kamera nur eine endliche Auflösung hat und außerdem eine zweidimensionale Bildebene, werden bei diesem Vorgang implizit zwei Transformationen durchgeführt:

1. Die dreidimensionale Laserlinie wird in einen **zweidimensionalen** Polygonzug überführt.
2. Die analogen Raumkoordinaten werden in **diskrete** Bildschirmkoordinaten umgewandelt.

In der Konsequenz brauchen nur noch zwei Koordinaten im unsigned int Format (je 16 Bit) gespeichert zu werden. Da die Vertexfarbe durch die Kamera in Rot-, Grün- und Blauanteile zerlegt und im Bereich von 0 bis 255 diskretisiert wird, reichen für die Farbe drei mal acht Bit, also 3 Byte, aus.
Ein Punkt kann also wie folgt abgespeichert werden:

[8] vgl. hierzu auch Kapitel II.A.

Absolute Darstellung:

Feld	X	Y	Rot	Grün	Blau
Typ	Integer	Integer	Byte	Byte	Byte

Die Bezeichnung 'absolute Darstellung' läßt es vermuten: es gibt noch eine zweite Art der Datenrepräsentation, die

Relative Darstellung:

Feld	ΔX	ΔR	ΔG	ΔB
Typ	Nibble	Nibble	Nibble	Nibble

Man macht sich hier eine besondere Eigenschaft der Daten zunutze:

> Im Allgemeinen beschreibt der Polygonzug eine zusammenhängende Linie wie sie der Laser auf der Oberfläche des abgetasteten Objektes hinterläßt.

Das heißt, daß sich zwei aufeinanderfolgende Punkte nur geringfügig in ihren Koordinaten und Farben unterscheiden[9]. Eine weitere Besonderheit der Daten liegt in ihrer Erzeugung begründet:

> In den meisten Fällen ist die Y-Koordinate eines Punktes um genau eins größer, als die des Vorgängers[10].

Hat man erst einmal einen Punkt absolut definiert, bietet es sich also an, für die folgenden nur noch den relativen Zuwachs (Offset) zum Vorgänger anzugeben. Der Y-Offset ΔY wird zu +1 angenommen. Als Datentyp für die Offsets wurde ein Halbbyte (Nibble, 4 Bit) gewählt. Damit ist folgende Zuordnung möglich:

Wert (Hex)	Wert (bin)	Offset
9	1001	-7
A	1010	-6
B	1011	-5

[9] Das gilt natürlich nur für weiche Farbverläufe und stetige Kurvenzüge. Die Praxis zeigt aber, daß dies sehr oft der Fall ist.

[10] Der Laser erzeugt auf dem Objekt eine tendentiell senkrechte Linie; das Bild der Kamera wird zeilenweise von oben nach unten abgetastet.

Wert (Hex)	Wert (bin)	Offset
C	1100	-4
D	1101	-3
E	1110	-2
F	1111	-1
0	0000	0
1	0001	+1
2	0010	+2
3	0011	+3
4	0100	+4
5	0101	+5
6	0110	+6
7	0111	+7
8	1000	ESCAPE

Wie man sieht, erlaubt die relative Darstellung Offsets von ±7. Wenn ein Punkt sich jedoch in seinem X-, Rot-, Grün- **oder** Blauwert um mehr als 7 von seinem Vorgänger unterscheidet, oder sein Y-Wert nicht um genau eins größer ist, versagt die relative Darstellung, und es wird wieder in den absoluten Modus geschaltet. Dies geschieht, indem ΔX auf einen reservierten 'Escape'-Wert (8_{16}) und ΔRot auf den Wert 0_{16} gesetzt werden. Danach wird **ein** Punkt absolut geschrieben und sofort wieder in den relativen Modus geschaltet.
Das Ende der Datenblockes wird durch ΔX=ESCAPE, ΔRot=F_{16} definiert.

1.Abschätzung der Dateigröße

Um einen Eindruck von den zu verarbeitenden Dateigrößen zu gewinnen, schätze ich den Speicherbedarf ab.
Im ungünstigsten Fall (stark unstetiger Kurvenzug) muß für jeden Punkt die absolute Darstellung gewählt werden, wobei jedesmal ein Escape-Byte (ΔX=Escape, ΔRot=0) vorangestellt wird. Der maximale Speicherbedarf für einen Polygonzug ergibt sich zu:

$$S_{max}(N) = N \bullet (2 + 2 + 1 + 1 + 1) + (N - 1) \bullet 1 + 1 = 8 \bullet N \qquad \textbf{(III.1)}$$

Für den günstigsten Fall reichen pro Punkt (mit Ausnahme des Ersten) zwei Byte, zuzüglich einem End-Of-Data Byte pro Polygonzug:

$$S_{\min}(N) = 1 \bullet 7 + (N - 1) \bullet 2 + 1 = 2 \bullet N + 6 \qquad \textbf{(III.2)}$$

Pro Polygonzug kommen noch 80 Bytes für die lokale Matrix hinzu. Außerdem benötigen Header und globale Transformation noch Platz.
In der Praxis kann man im Mittel mit ungefähr 5 Bytes pro Vertex rechnen. Die Dateigröße für einen Datensatz mit 360 Polygonen à 400 Punkten liegt somit bei etwa 720 kByte.

D.Transformation der Vertexkoordinaten

Wie wir oben gesehen haben, läßt sich die Datenmenge stark reduzieren, wenn man diskrete Bildschirmkoordinaten und Transformationsmatrizen getrennt abspeichert. Der Preis: um die eigentlich gewünschten 3D-Daten zu erhalten, muß jeder gelesene Punkt mit der gültigen Matrix multipliziert werden. Diese "gültige" Transformation T setzt sich aus der (am Dateianfang definierten) globalen Transformation T_G[11] und der aktuellen lokalen Transformation T_L[12] zusammen.
T_L wird normalerweise vor jedem Polygonzug neu definiert und bleibt gültig bis ein neuer Wert gelesen wird. Wie bereits oben beschrieben, kann T_L auf verschiedene Arten angegeben werden (vgl. Seite 15): absolut oder relativ. Abhängig davon ist natürlich die Auswertung:

T_L absolut:

$$T = T_L \bullet T_G \qquad \textbf{(III.3)}$$

T_L relativ:

$$T = T_L \bullet T_{L'} \bullet T_G = T_L \bullet T' \qquad \textbf{(III.4)}$$

wobei T_L' die vorige aktuelle lokale Transformation, und T' die vorige gültige Matrix ist.

Der eingelesene Polygonzug wird transformiert, indem alle seine Punkte mit T multipliziert werden. Jeder Bildpunkt P_B wird also in einen Modellpunkt P_M überführt.

[11] Andere Bezeichnung für T_G: Kamera Transformation

[12] Andere Bezeichnung für T_L: Edge Transformation

Allgemein:

$$P_M = T \bullet P_B \qquad \textbf{(III.5)}$$

Eine gute Darstellung der Themen *Transformationen* und *homogene Matrizen* geben M.A.Penn und R.R.Patterson in [Patterson86].

E.Algorithmen

1.Cef-Datei lesen

Wie oben gesehen, sind die CEF-Daten auf minimalen Speicherbedarf optimiert und können nicht direkt weiterverarbeitet werden. Es wäre sehr speicherintensiv, die ganze Datei im Stück in den Rechner einzulesen und erst dann zu konvertieren.
Effizienter ist es, jeweils nur wenige Polygonzüge in einen Puffer zu lesen, zu konvertieren, den Puffer mit den nächsten Polygonzügen zu überschreiben, usw.
Parallel dazu werden die konvertierten Polylines auf verschieden Arten weiterverarbeitet:

1. Es erfolgt wahlweise eine Datenreduktion (beschrieben in Kapitel VI.B)
2. Aus jeweils zwei Polylines wird ein Streifen des Triset aufgebaut (vgl. nächstes Kapitel)

```
Funktion readNextPolyline
PARAMETER:      ↔ Taktuell
                → TG
                ← Polygonzug
RÜCKGABE:       TRUE, wenn gelesen werden konnte

REPEAT
        Lies bis zum nächsten Blockbeginn
        IF Blocktyp=Transformation THEN
                Lies Transformation T
                IF relative Transformation THEN
                        Taktuell ← T · Taktuell
                ELSE
                        Taktuell ← T · TG
                ENDIF
        ENDIF
UNTIL (Blocktyp=Polygonzug) OR (Blocktyp=EoF) OR (Fehler)

IF Blocktyp=Polygonzug THEN
        RETURN readPlData (Taktuell, Polygonzug)
ELSE
        RETURN FALSE
ENDIF
```

Alg. I,1 Jeweils nächsten Polygonzug lesen

Benötigt wird deswegen eine Funktion zum Einlesen des jeweils nächsten Polygonzugs (**readNextPolyline**). Voraussetzung dafür ist allerdings, daß die Datei vorher geöffnet und die globale Transformation T_G gelesen wurde. Der Funktion müssen T_G und die gerade gültige Transformation $T_{aktuell}$ übergeben werden. Es wird dann solange in der Datei weitergelesen, bis ein Polygonzug-Datenblock gefunden wird. Alle bis dahin auftretenden Transformationen werden ausgewertet, und $T_{aktuell}$ entsprechend auf den neuesten Stand gebracht.
Der Polygonzug selbst wird durch Aufruf der Funktion **readPlData** gelesen. Damit dabei die Punkte in das Modellkoordinatensystem transformiert werden können, benötigt **readPlData** die gültige Matrix $T_{aktuell}$.
Wie **readNextPolyline** in die Hauptverarbeitungsschleife eingefügt wird, zeige ich im Anhang.

2.Polygonzug lesen

```
Prozedur readPlData
PARAMETER:      → Transformationsmatrix T
                ← Polygonzug

Absolut ← TRUE
REPEAT
        IF Absolut THEN
                Lies XB, YB, RB, GB, BB
                Absolut ← FALSE
        ELSE
                Lies △X, △R
                IF △X=8₁₆ THEN
                        IF △R=F₁₆ THEN
                                BREAK           (weiter nach UNTIL)
                        ELSE
                                Absolut ← TRUE
                                CONTINUE(zurück zum REPEAT)
                        ENDIF
                ELSE
                        Lies △G, △B
                        XB ← XB + △X
                        YB ← YB + 1
                        RB ← RB + △R
                        GB ← GB + △G
                        BB ← BB + △B
                ENDIF
        ENDIF
        (XM,YM,ZM) ← T · (XB,YB,0)
        RM ← RB / 255
        GM ← GB / 255
        BM ← BB / 255
        Polygonzug um Vertex (XM,YM,ZM,RM,GM,BM) erweitern
UNTIL FALSE
```

Alg. I,2 Polygonzug lesen

Bei der Dekodierung eines einzelnen Blocks mit Polygonzugdaten muß darauf geachtet werden, daß die einzelnen Vertexdaten nach dem oben beschriebenen Mechanismus als relativ oder absolut behandelt werden.

Des weiteren müssen die Vertexkoordinaten von der Bildebene in das Modellkoordinatensystem transformiert werden. Dies geschieht durch Multiplikation jedes gelesenen Punktes mit $T_{aktuell}$. Die Vertexfarben werden durch 255 geteilt. So wird sichergestellt, daß die RGB-Anteile im Intervall [0, 1] liegen , wie es dem Phigs-Standard entspricht.

IV.Erzeugen von Trimeshes

A.Vorüberlegungen, Begriffsdefinitionen

Wie bereits beschrieben erzeugt Spex3D eine Menge von Punkten, die auf der Oberfläche des digitalisierten Objektes liegen.
Sieht man sich diese 'Punktwolke' mit Hilfe eines Renderingprogrammes an, stellt man ernüchtert fest, daß das Ergebnis nur wenig mit dem Original gemein hat. Die einzelnen Punkte liegen zwar an den richtigen Positionen und haben auch die richtige Farbe. Die Tatsache, daß man zwischen den Punkten hindurch in das Innere des Objektes blicken kann, macht die Darstellung aber unrealistisch.
Aber selbst wenn die digitalisierten Punkte so dicht beieinander lägen, daß auf den ersten Blick keine Lücken erkennbar sind, wäre eine Visualisierung nicht zufriedenstellend:
Eine (foto-) realistische Darstellung zeichnet sich zum Beispiel dadurch aus, daß ein Objekt Schatten wirft, oder eine Lichtquelle sogenannte 'Highlights'[13] erzeugt. Renderingprogramme können solche Darstellungen produzieren, indem sie neben der Objektgeometrie und -farbe noch weitere Eigenschaften in die Bildberechnungen einbeziehen:

1. Oberflächeneigenschaften, wie Struktur (glatt, rauh), Transparenz etc.
2. Orientierung der Oberfläche (Flächennormale)
3. 'Hidden Line, Hidden Surface' Darstellung, d.h. für den Betrachter unsichtbare Teile werde ausgeblendet.

Da ein Punkt aber keine Orientierung hat (wo ist bei einem Punkt 'oben' ?), kann man nicht sagen, in welche Richtung er einen einfallenden Lichtstrahl reflektieren würde. Und weil ein Punkt keine Fläche hat (ein Punkt ist unendlich klein), kann er auch keine hinter ihm liegenden Objekte verdeken.
Es muß also ein Oberflächenmodell erzeugt werden, indem man jeweils N benachbarte Punkten als Ecken eines Polygons definiert[14]. Wenn man zwischen allen Punkten Polygone definiert hat, beschreiben diese eine zusammenhängende Oberfläche.
Ich habe die Polygone vom dritten Grad gewählt (also Dreieke). Dadurch ist sichergestellt, daß die Flächenstücke immer planar sind, weil drei Punkte immer auf einer Ebene liegen. Diese Dreiecke nennt man auch **Faces** oder **Facets**. Die Fläche, die aus diesen Facets besteht, heißt **Trimesh** oder **Triset**. Die Eckpunkte eines Faces werden als **Vertizes** (Einzahl: Vertex) bezeichnet. Das Geradenstück zwischen zwei Vertizes nennt man auch **Edge**.
Für den Vorgang der Erzeugung von Trisets aus Polygonen verwende ich den Begriff **Tesselierung** oder **Triangulation**.

---

[13] Ein Highlight, auch Glanzlicht, entsteht zum Beispiel, wenn man einen glänzenden Gegenstand betrachtet, der von einem Punktstrahler beleuchtet wird.

[14] Anstatt durch ebene Polygone kann man die Flächenstücke auch durch gekrümmte Flächen definieren, die Weiterverarbeitung solcher Splineflächen ist aber ungleich schwieriger. Es wäre auch denkbar statt einem Oberflächenmodell ein Solidmodell zu wählen, was aber ebenfalls eine Weiterverarbeitung verkomplizieren würde.

B.Lösungsansatz

Der Schlüssel zur Erzeugung eines Trimesh liegt in der Bestimmung der benachbarten Vertizes, die jeweils zu einem Face verbunden werden sollen. Man kann diesen Vorgang stark vereinfachen, wenn man bekannte Eigenschaften der Eingangsdaten ausnutzt:

1. Spex3D generiert Polygonzüge, in denen die Vertizes geordnet auftreten.

D.h. Vertizes, die innerhalb eines Polygonzugs nebeneinander liegen, sind auch im Triset Nachbarn und somit zwei Eckpunkte eines dreieckigen Facets. Zur vollständigen Bestimmung fehlt also nur noch ein weiterer Vertex. Aber auch der kann nicht weit sein:

2. Spex3D generiert eine Folge von benachbarten Polygonzügen.

D.h. benachbarte Vertizes liegen in benachbarten Polygonzügen oder im selben Polygonzug.
Ein einfacher Ansatz für die Triangulation ist folgender:
Man betrachtet jeweils zwei benachbarte Polygonzüge P_n und P_{n+1} (3) und erzeugt daraus einen Facet-Streifen (auch **Tristrip**). Ein weiterer Streifen wird aus P_{n+1} und P_{n+2} generiert, an den bestehenden angefügt usw[15].
Leider ist die Anzahl der Vertizes in den Polygonzügen nicht konstant. Auch die Abstände zwischen den Punkten können unregelmäßig sein. Wie man in der Abbildung sehen kann ist es keineswegs egal, welche der Punkte man zu Facets verbindet[16].

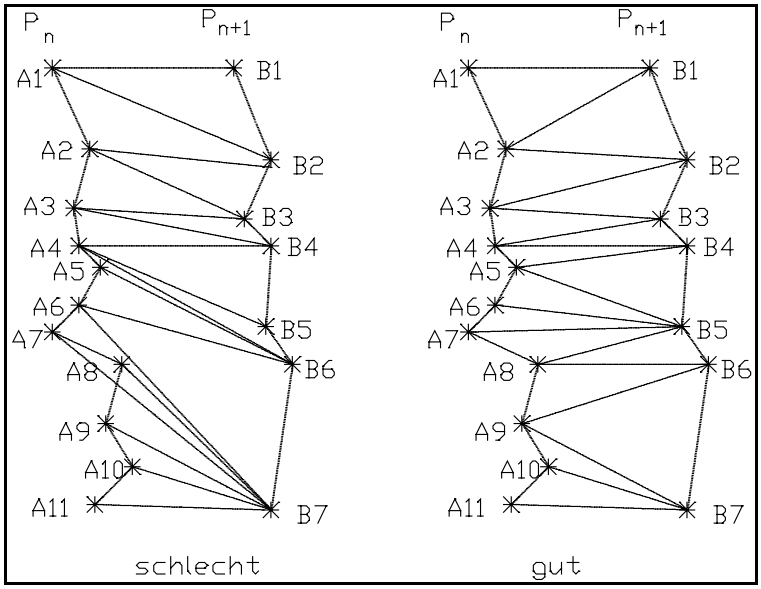


Abb. IV.3 Zwei Tesselierungsvarianten

Um die Frage zu beantworten, wie die beiden Polygonzüge zu einem guten Tristrip verbunden werden können, erörtere ich zunächst, was einen

[15] Aus einem Datensatz mit N Polygonzügen werden auf diese Weise N-1 Tristrips generiert. Wenn ein Objekt rundherum digitalisiert wurde (360°), dann ist es sinnvoll, zusätzlich den ersten und letzten Polygonzug durch eine Tesselierung zu verbinden.

[16] Im linken, schlechten, Fall treten wesentlich mehr schmale Dreiecke auf, als nötig.

'guten' Tristrip ausmacht:
Die Facets der einzelnen Tristrips ergeben zusammen die Oberfläche des Objektes. Sie sollen sich wie eine Haut um seine Konturen legen und dabei alle digitalisierten Punkte berühren. Wie im richtigen Leben ist Faltenbildung dabei unerwünscht: die Fläche der Haut soll möglichst klein sein. Außerdem haben sich sehr schmale Dreiecke in der Praxis als ungünstig erwiesen[17]. Eine weitere Forderung ist, daß sich die Faces nicht überschneiden dürfen.
In meinem Lösungsansatz versuche ich Punkte zu verbinden, die möglichst dicht beieinander liegen. Da die Fläche der Faces proportional zu den Seitenlängen ist, wird damit auch die Gesamtfläche klein bleiben.
Es wäre allerdings ungünstig, für die einzelnen Punkte A_1, A_2, ... der Reihe nach den nächstgelegenen Partner B_i in der Nachbar-Polyline zu suchen, weil dann evtl. für die letzten Punkte (.., A_{10}, A_{11}) nur noch 'Notlösungen' übrigbleiben. Ich habe deswegen einen rekursiven Algorithmus gewählt, der die einzelnen Vertizes gleichwertig behandelt:

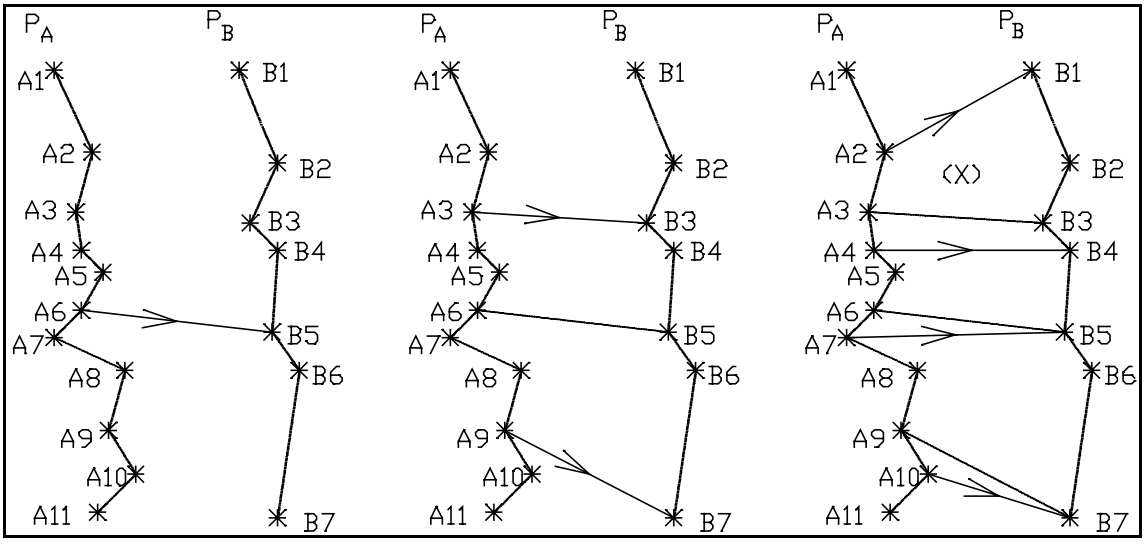


Abb. IV.4 Rekursive Triangulation von zwei Polygonen

Die beiden beteiligten Polygonzüge P_A und P_B werden solange in kleinere, gegenüberliegende Intervalle unterteilt, bis sich die Faces von selbst ergeben (4). Dazu wird der mittlere Punkt von P_A (A_6) mit dem nächstgelegenen Punkt aus P_B (B_5) verbunden. Für das Intervall $P_{A1\text{-}A6}$ und sein 'Gegenüber' $P_{B1\text{-}B5}$ wird dieselbe Funktion rekursiv aufgerufen. Der Mittelpunkt A_3 wird mit dem nächstgelegenen Punkt aus P_B (B_3) verbunden. Ebenso wird das Intervallpaar $P_{A6\text{-}A11}$, $P_{B5\text{-}B7}$ rekursiv behandelt.
Nach der dritten Rekursionsebene sind bereits acht Intervalle entstanden (im Beispiel von 4 sind damit gleichzeitig schon vier Faces eindeutig definiert[18]).
Das mit (X) gekennzeichnete Intervallpaar zeigt einen Sonderfall, der berücksichtigt werden muß: da hier [B_1, B_3] mehr Vertizes enthält als [A_2, A_3], wird jetzt das Intervall im Polygonzug P_B halbiert (Mittelpunkt: B_2) und der nächstgelegene Punkt in P_A gesucht. Das Intervall im Polygonzug B muß offensichtlich so behandelt werden, wie

[17] Zu schmale Faces zeigen beim Texture Mapping unschöne Effekte

[18] _(A_1,A_2,B_1), _(A_6,A_7,B_5), _(A_9,A_{10},B_7) und _(A_{10},A_{11},B_7)

normalerweise die Intervalle im Polygonzug A. Dies kann durch Vertauschen von Quell- und Zielinterval erreicht werden (vgl. Algorithmus).

1.Gleichmäßige Orientierung der Faces

Viele Renderingprogramme können eine schnellere Darstellung durchführen, indem sie Faces, die dem Betrachter ihre Rückseite zuwenden, ausblenden ('Backface Culling'). Man geht dabei davon aus, daß bei der Betrachtung eines Objektes mit **geschlossener** Oberfläche von jedem Blickwinkel aus nur die Vorderseite der Faces sichtbar ist[19]. Das gilt jedoch nur, wenn alle Faces des Triset gleichmäßig nach außen orientiert sind.
Leider ist es nicht immer möglich, zu sagen wo bei einer Oberfläche innen oder außen ist. Das ist aber auch nicht nötig, solange die Faces nur **gleichmäßig** ausgerichtet sind. Falls sich bei der Betrachtung herausstellt, daß **alle** Faces falsch orientiert sind, läßt sich das egalisieren, indem nicht Backface- sondern 'Frontface Culling' benutzt wird.
Eine gleichmäßige Orientierung wird erreicht, wenn alle Faces auf die gleiche Art definiert werden, d.h. indem z.B. die Eckpunkte aller Faces im Uhrzeigersinn angegeben werden. In 4 bedeutet das z.B., daß ein Dreieck als $\Delta(A_1,A_2,B_1)$ und nicht etwa als $\Delta(A_1,B_1,A_2)$ definiert wird.
Vorsicht ist geboten, wenn in dem oben erwähnten Sonderfall die Polygonzüge A und B vertauscht wurden: die Reihenfolge der Face-Ecken muß trotzdem gleich bleiben. Wenn eine Funktion rekursiv mit vertauschten Parametern aufgerufen wurde, muß ihr das aus diesem Grund über ein Flag mitgeteilt werden (vgl. Algorithmus).

2.Abschätzung der Anzahl der Faces

Ich leite nun eine Abschätzung für die Anzahl N_F der Faces her, die bei der Tesselierung eines Datensatzes entstehen. Dieser Datensatz habe N_V Vertizes in N_{PG} Polygonzügen.
Der Einfachheit halber nehme ich die Anzahl N_{VP} der Vertizes pro Polyline für alle Polygonzüge konstant zu $\tilde{N}_{VP} = N_V/N_{PG}$ an[20].
Alle benachbarten Punkte eines Polygonzugs sind jeweils durch eine Edge verbunden und bilden somit die Seite eines Faces (der fehlende Punkt des Dreiecks ist im jeweils anderen Polygonzug zu finden). Jeder Polygonzug steuert damit im Mittel $\tilde{N}_{VP}$-1 Faces zu einem Tristrip bei. Jeder Tristrip wird aus zwei Polygonzügen gebildet, und das gesamte Trimesh wiederum aus N_{PG} Tristrips:

$$\begin{aligned} N_F &= (\overline{N}_{VP} - 1) \bullet 2 \bullet N_{PG} \\ &\approx \overline{N}_{VP} \bullet 2 \bullet N_{PG} \\ &\approx \frac{N_V}{N_{PG}} \bullet 2 \bullet N_{PG} \\ &\approx 2 \bullet N_V \end{aligned} \qquad \textbf{(IV.6)}$$

[19] Ein gutes Beispiel ist ein Fußball, von dessen einzelnen Lederflicken man nur die Außenseiten sehen kann. Jeder Flicken, der dem Betrachter seine Rückseite zeigt wird durch einen oder mehrere Flicken verdeckt, die man von vorne sieht.

[20] Die Anzahl der Vertizes pro Polygonzug liegt im Bereich von 300..550.

Ein Triset hat etwa doppelt soviele Faces wie Vertizes.

C.Algorithmus

Für die programmtechnische Umsetzung wird offensichtlich eine Funktion benötigt, die als Parameter zwei Polygonzugintervalle erhält, die ggf. rekursiv weiter unterteilt werden.

```
Prozedur connectInterval
Parameter: → PA:      Polygonzug A
                  → Astart:   Intervallstart in PA
                  → Aend:     Intervallende in PA
                  → PB:       Polygonzug B
                  → Bstart:   Intervallstart in PB
                  → Bend:     Intervallende in PB

IF Bstart=Bend THEN
        FOR i := Astart TO Aend DO
                processFace (PB [Bstart], PA [i], PA [i+1])
        ENDFOR
        RETURN
ENDIF
IF Astart=Aend THEN
        FOR i := Bstart TO Bend DO
                processFace (PA [Astart], PB [i+1], PB [i])
        ENDFOR
        RETURN
ENDIF
IF (Aend-Astart)=1 AND (Bend-Bstart)>1 THEN
        Swapped ← NOT Swapped
        connectInterval (PB, Bstart, Bend, PA, Astart, Aend)
        Swapped ← NOT Swapped
        RETURN
ENDIF
Amitte ← (Astart + Aend) DIV 2
AKoord ← Koordinaten des Punktes Amitte
Bmitte ← nearestPlVertex (AKoord, PB, Bstart, Bend)
connectInterval (PA, 0, Amitte, PB, 0, Bmitte)
connectInterval (PA, Amitte, Aend, PB, Bmitte, Bend)
```

Alg. IV,3 Tesselierung eines Tristrip

Zunächst werden Sonderfälle abgeprüft: Wenn eines der beiden Intervalle nur aus einem Vertex besteht ($A_{start}=A_{end}$ oder $B_{start}=B_{end}$), ist eine weitere Unterteilung nicht mehr möglich. Der Vertex wird mit allen Vertizes des gegenüberliegenden Polygonzugs zu Faces verbunden, die Rekursion ist beendet.

Der zweite Sonderfall, der behandelt werden muß tritt auf, wenn das Intervall im Polygonzug B mehr Punkte enthält, als das in A. In diesem Fall ruft sich **connectInterval** selbst noch einmal mit vertauschten Parametern auf. Damit diese Vertauschung bei der Orientierung der Faces berücksichtigt werden kann, wird gleichzeitig ein statisches Flag (**Swapped**) invertiert.

Wenn keiner der genannten Sonderfälle vorliegt, dann wird das Intervall A halbiert (Mittelpunkt: A_{mitte}). Intervall B wird ebenfalls geteilt, und zwar an dem Punkt, der A_{mitte} am nächsten ist[21].
Es sind damit zwei Intervallpaare entstanden, die rekursiv bearbeitet werden.
Die eigentliche Implementierung wurde um einige Eigenschaften erweitert, die ich im Pseudocode aus Gründen der Übersichtlichkeit nicht berücksichtigt habe:

- Wenn **Swapped** gesetzt ist, wird die Reihenfolge zweier Face-Ecken vertauscht, bevor diese an **processFace** übergeben werden. So wird eine gleichmäßige Orientierung der Dreiecke gewährleistet.
- Für statistische Zwecke wird außerdem die maximale Rekursionstiefe ermittelt.

ProcessFace ist eine Funktion, die als Parameter die Eckpunkte eines Dreiecks erhält. Sie wird von **connectInterval** genau einmal für jedes Dreieck des generierten Tristrips aufgerufen. In dem hier besprochenen Fall wird **processFace** das übergebene Dreieck dem Triset hinzufügen (näheres dazu im folgenden Abschnitt 0).
Gesteuert über ein globales Flag, kann **processFace** aber auch eine zweite Funktion ausführen: die Dreiecke gelangen nicht in das Triset, sondern werden in die Textur-Bitmap projiziert. Auf diese Weise kann **connect-Interval** auch für das Texture Mapping aufgerufen werden (vgl. Kapitel V).

D.Datenstrukturen, Zugriffsmethoden

Die naheliegendste Methode, ein Triset im Speicher abzubilden ist wohl, die einzelnen Faces mit ihren Eckpunkten nacheinander in einer Liste abzulegen:

Face #	Vertex 1	Vertex 2	Vertex 3
1	X,Y,Z,R,G,B	X,Y,Z,R,G,B	X,Y,Z,R,G,B
2	X,Y,Z,R,G,B	X,Y,Z,R,G,B	X,Y,Z,R,G,B
...	...	...	...
N_F	X,Y,Z,R,G,B	X,Y,Z,R,G,B	X,Y,Z,R,G,B

Vertexkoordinaten und -farben liegen im Fließkommaformat vor und belegen bei einfacher Genauigkeit 6·4=24 bytes pro Vertex. Pro Face ergibt sich ein Speicherbedarf von 3·24=72 bytes.
Wie ich in 0 gezeigt habe, kommen in einem 'durchschnittlichen' Triset doppelt soviele Faces wie Vertizes vor. Im vorigen Beispiel werden aber dreimal soviele Punkte definiert wie Faces, also sechsmal mehr als nötig.
Es ist wesentlich effizienter, Vertexdaten in einer getrennten Liste abzulegen. Bei der Definition eines Dreiecks

[21] Die Routine **nearestPlVertex** ermittelt zu einem Punkt V (X,Y,Z) den nächstgelegenen Punkt eines Polygonzugintervalls.

genügt jetzt die Angabe der **Indizes** der Eckpunkte (typ long int, 4 bytes).
Legt man zugrunde, daß auf ein Face etwa 0,5 Vertizes kommen, so benötigt man jetzt 3·4 + 0,5·24 = 24 bytes pro Face.

Vertexarray:

Vertex #	Koordinate	Farbe
1	X,Y,Z	R,G,B
2	X,Y,Z	R,G,B
...	...	...
N_V	X,Y,Z	R,G,B

Facearray:

Face #	Index 1	Index 2	Index 3
1	I_1	I_2	I_3
2	I_1	I_2	I_3
...	...	...	...
N_F	I_1	I_2	I_3

Wenn ich hier von Listen spreche, meine ich damit keine über Pointer verketteten Listen. Ich habe diese Listen in Form von dynamischen Arrays realisiert, die eine ähnlich flexible Speicherausnutzung bieten[22]. Der Vorteil liegt darin, daß auf Elemente direkt über Indizes zugegriffen werden kann. Im Gegensatz zu einer verketteten Liste, ist es leider nicht so einfach möglich, Elemente aus der Mitte zu entfernen oder dort einzufügen. Eine häufige Änderung der Elementreihenfolge ist aber auch nicht erwünscht, weil dann jedesmal alle darauf verweisenden Indizes neu berechnet werden müßten. Ich behelfe mir, indem ich Elemente nicht lösche, sondern nur als gelöscht markiere. Die entstehenden Löcher werden durch gelegentliche Garbage Collections eliminiert. Bei dieser (relativ seltenen) Gelegenheit werden dann auch alle referenzierenden Indizes aktualisiert. Neue Elemente werden immer am Ende angefügt, was in Arrays einfach zu realisieren ist.
Da beliebig große Trisets verarbeitet werden sollen, Arrays aber normalerweise nicht größer als 64 kByte werden dürfen[23], zerlege ich das Vertex- und das Facearray ggf. in kleinere Blöcke, die wiederum in einer Liste verwaltet werden. Diese interne Verwaltung bleibt aber weitgehend transparent. Über einfache Routinen können die einzelnen

[22] Im Gegensatz zu einem statische Array liegt die Größe eines dynamisches Arrays zur Übersetzungszeit noch nicht fest. Vielmehr 'wächst' oder 'schrumpft' es mit der Anzahl der Elemente, die es aufnimmt.

[23] Diese Beschränkung fällt mit den kommenden Windows-Versionen und 32-Bit Compilern

Elemente wie in einem einzigen großen Array indexiert werden[24].
Da mehr als 2^{16}=65535 Elemente in einer Liste verwaltet werden sollen, reichen zwei Bytes für den Index nicht aus. Indizes sind deswegen vom Typ **long int** (4 bytes).

Ich habe die Struktur für Comp3D noch um einige Datenfelder erweitert, um

1. den Zugriff auf das Triset zu beschleunigen (besonders für die Suche nach bestimmten Daten) und
2. gelöschte Vertizes zu verwalten[25].

Aus diesen Gründen erhält jeder Eintrag des Facearray einen Zeiger auf eine Liste der zugehörigen gelöschten Vertizes. Die Vertexdaten wurden ebenfalls um ein Feld erweitert, in dem für gelöschte Vertizes das zugehörige Face eingetragen ist.

Um die Vertexliste möglichst klein zu halten, ist der Eintrag von doppelten Punkten zu vermeiden. Da keine Aussage über die Reihenfolge gemacht werden kann, mit der neue Faces dem Triset hinzugefügt werden, ist es nötig, daß die Eckpunkte jedes neuen Dreiecks mit allen bereits vorhandenen Vertizes verglichen werden. Um diese (bei großen Listen zeitaufwendige) lineare Suche zu beschleunigen, wird während des Trisetaufbaus über eine Hash-Tabelle auf das Vertexarray zugegriffen. Der Hashingalgorithmus ist u.A. in [Sedge89] S.231 ff. beschrieben.

[24] So liefert beispielsweise die Funktion **getFaceP (i)** einen Zeiger auf das i-te Element im Facearray.

[25] Warum Vertizes, die während der Reduktion für überflüssig befunden wurden, nicht völlig vergessen werden dürfen, erläutere ich in Kapitel VI.

V.Extraktion der Farbinformation

Ich möchte anhand eines Beispiels das Prinzip erläutern, nach dem ich die Datenreduktion durchführe.
Angenommen, das Ausgangsobjekt sei ein einfarbiger Würfel mit einer Kantenlänge von ca. 20 cm. Bei einer vollständigen Digitalisierung erhält man z.B. 360 Polygonzüge à 300 Punkte.
Diese gut 100.000 Vertizes werden nach dem Verfahren aus Kapitel IV zu ca. 200.000 Faces konvertiert.
Würde jemand einen Würfel mit Hilfe eines CAD-Programms konstruieren, so wären dagegen lediglich acht Vertizes und 12 Faces nötig (Ein Punkt an jeder Ecke und zwei Dreiecke pro Seite).
In diesem Beispiel liefert die Digitalisierung eine riesige Menge redundanter Information die in einem weiteren Schritt zu reduzieren wäre.
Anders sähe es aus, wenn der beschriebene Würfel nicht einfarbig, sondern bunt gemustert wäre. Der oben erwähnten Technische Zeichner hätte jetzt große Mühe, den Würfel zu modellieren.
Die gewaltige Punktmenge die der Digitalisierer erzeugt ist nicht mehr redundant, weil die Punkte nun nötig sind, um die Farben des Objektes abzubilden. Auch für einen Körper mit einfacher Geometrie müssen viele kleine Faces erzeugt werden, damit die Farbinformation erhalten bleibt. Das liegt daran, daß ein Face nicht beliebig bunt sein kann, sondern nur an den Eckpunkten Farbinformationen erhält[26].
Wäre es möglich, die Farbinformation von den Geometriedaten zu trennen und unabhängig zu speichern, dann könnte man allerdings wie im ersten Beispiel die Geometriedaten stark reduzieren. Das macht natürlich nur Sinn, wenn danach die alten Farben auf das reduzierte Objekt zurück übertragen werden können. Man könnte den Würfel so durch 12 große, bunt gemusterte Dreiecke darstellen.
Dieses Verfahren hätte einen zweiten Vorteil:
Feine Oberflächenstrukturen können nur durch sehr viele kleine Faces nachgebildet werden (z.B. ein schrumpliger Apfel). Dabei reicht es für eine realistische Darstellung oft aus, die Oberflächen**strukturen** zu ignorieren und nur die -**farben** zu erhalten (in diesem Fall würde eine glatte Kugel mit aufgemalten Runzeln sehr ähnlich aussehen). Wenn man während der Reduktion einen kleinen Fehler zuläßt, werden feine Strukturen durch ebene Flächen ersetzt, was mit wesentlich weniger Faces möglich ist. Der optische Eindruck bleibt trotzdem erhalten.

A.Two-Part-Mapping

Gesucht ist eine Methode mit der die Farbinformation eines 3D-Trimesh isoliert werden kann. Nachdem die Geometrie des Trimesh vereinfacht wurde, muß dieser Schritt wieder rückgängig gemacht, d.h. die 'Original-Farbe' auf das nun grobmaschigere Triset übertragen werden.
Die extrahierte Farbinformation (**Textur**) wird sinnvollerweise als Bitmap abgespeichert. Diese Bitmap habe die Ausdehnung U_{max} x V_{max} Pixel.

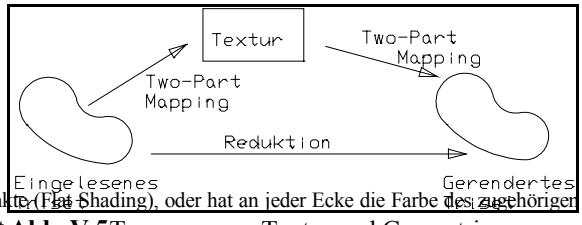


Abb. V.5 Trennung von Textur und Geometrie

[26] Ein Face erhält entweder die Farbe eines seiner Eckpunkte (Flat Shading), oder hat an jeder Ecke die Farbe des zugehörigen Vertex. Dann wird die Färbung der restlichen Fläche interpoliert.

Es ist eine Abbildungsfunktion nötig, die jeden Punkt P (X,Y,Z) des Triset auf einen Punkt P' (U,V) in der Textur abbildet. Weil der Vorgang nach der Reduktion rückgängig gemacht werden soll, sollte die Zuordnung möglichst ein- eindeutig sein.
Für ein einfaches Objekt, wie beispielsweise einen aufrecht stehenden Zylinder ohne Bodenflächen ist das einfach zu erreichen:
Schneidet man den Zylinder längs der Hochachse auf und 'biegt' die Oberfläche gerade, so erhält man bereits eine rechteckige Fläche, die man als Textur bezeichnen kann. Geschnitten und gebogen wird natürlich nur mathematisch, mittels einer Transformation.
Bei komplexeren Objekten, mit denen wir es im Allgemeinen zu tun haben, ist so eine einfache Transformation nicht möglich.
Die grundlegende Idee des Two-Part-Mapping ist nun, die Objektoberfläche in zwei Schritten zu transformieren: zunächst werden die Vertizes auf eine einfache Zwischenfläche projiziert (z.B. auf eine Zylinderoberfläche, die das Objekt umschließt). Diese Zwischenfläche wird in einem zweiten Schritt auf eine Texturebene abgebildet.
Während der zweite Schritt relativ einfach durchzuführen ist (aus diesem Grund wählt man ja eine **geeignete** Zwischenfläche), bleibt das Problem, wie man die Farbe des Objektes auf die Zwischenfläche projiziert.
Ich möchte dazu ein geeignetes Verfahren anschaulich machen:
Angenommen, ein farbiges Objekt steht in der Mitte eines oben und unten offenen Zylinders. Weiter soll das Objekt aus transparentem Material bestehen und von innen beleuchtet sein (z.B. durch eine punktförmige Lichtquelle). Auf der Innenfläche des Zylinders werden jetzt die Objektfarben erscheinen.
Das beschriebene Verfahren ist als Two-Part-Mapping ist seit längerem in der Computergrafik bekannt, wobei dort im Allgemeinen der umgekehrte Weg gegangen wird: man steht oft vor dem Problem, daß man die Oberfläche eines konstruierten 3D-Körpers durch Aufprojektion einer anderweitig gewonnenen Textur (z.B. ein gescanntes Bild) optisch aufpeppen möchte[27]. Man spricht hier vom S-Mapping (Surface Mapping: Projektion der Texturebene auf die dreidimensionale Zwischenfläche), und O-Mapping (Object Mapping: Projektion der Zwischenfläche auf die Objektoberfläche).
Die von mir durchgeführten Rückprojektionen bezeichne ich analog als S^{-1}-Mapping und O^{-1}-Mapping.

1.Wahl von Zwischengeometrie und Methode

In dem oben genannten Beispiel habe ich als Zwischenfläche eine Zylinder gewählt. Denkbar sind aber noch andere Geometrien wie z.B. eine Kugel, eine Ebene oder ein Würfel[28].

Auch die Methode, nach der das O^{-1}-Mapping im Beispiel durchgeführt wurde ist nur eine von mehreren möglichen. Hier wurden Strahlen aus dem Zentrum des Objektes nach Außen geschickt. Der Punkt der Zwischenoberfläche, wo sie auftrafen erhielt die Farbe des Punktes an dem die Strahlen das Objekt durchdrungen haben. Diese Methode

[27] Die Textur muß nicht unbedingt die Farbe beeinflussen. Es ist z.B. auch möglich, den Reflexionswinkel, die Transparenz oder andere Eigenschaften der Körperoberfläche in Abhängigkeit von den Texturwerten zu verändern.

[28] Möglich ist natürlich jede beliebige Fläche. Kugel, Würfel und Zylinder haben sich wegen ihrer Einfachheit jedoch durchgesetzt.

wird als Centroid-Mapping bezeichnet.

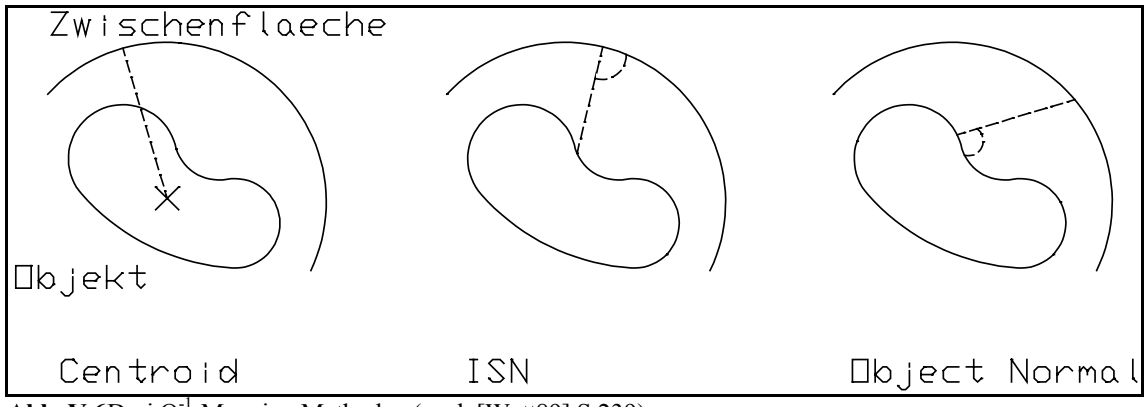


Abb. V.6 Drei O^{-1}-Mapping Methoden (nach [Watt89] S.239)

Weitere Mappingmethoden sind das ISN-Mapping und das Object-Normal-Mapping.
Beim ISN-Mapping (**I**ntermediate **S**urface **N**ormal) gehen die Strahlen nicht von einem zentralen Punkt im Objekt aus, sondern verlaufen senkrecht zur Zwischenoberfläche (vgl. 6).
Beim Object-Normal-Mapping werden von jedem Punkt der Objektoberfläche Strahlen losgeschickt, die in Richtung der Oberflächennormalen verlaufen.
Die genannten Zwischengeometrien und O^{-1}-Mappingmethoden lassen sich zu mehreren Mappingmethoden kombinieren, von denen einige im Vorwege ausgeschlossen werden können:

1. Beim Object-Normal-Mapping kann keine ein-eindeutige Abbildung garantiert werden, weil theoretisch sehr viele Oberflächennormalen auf den selben Punkt der Zwischenoberfläche zeigen können.
2. Eine Ebene ist als Zwischenfläche ungeeignet, weil es viele Strahlen gibt, die die Ebene überhaupt nicht berühren. Diese Farbinformationen wären verloren.
3. Auch bei der Kombination Zylinder/Centroid-Mapping gibt es Strahlen, die durch den Boden oder den Deckel verlaufen. Auch hier könnte Informationsverlust auftreten.
4. Da die Flächennormalen einer Kugel sich alle in einem Punkt treffen, sind für eine Kugel das Centroid-Mapping und ISN-Mapping identisch.
5. Eine würfelförmige Zwischenfläche, ist nur bedingt für das S^{-1} Mapping geeignet, weil die rechteckige Texturbitmap nicht voll ausgenutzt werden kann.

Ich habe mich bei der Implementierung für die Kombinationen Kugel/Centroid-Mapping und Zylinder/ISN-Mapping entschieden. Da die Kugel eine geschlossene Zwischenoberfläche bildet, können keine Strahlen verloren gehen.
Zylinder/ISN -Mapping liefert sehr gute Ergebnisse, weil hier die Verzerrungen relativ klein bleiben.

2.Zylinder, ISN

In 7 ist schematisch das O^{-1}-Mapping nach der Intermediate-Surface-Normal Methode zu sehen. Zwischengeometrie ist hier eine Zylinderfläche, die das digitalisierte Objekt umgibt.
Der Vertex P (X,Y,Z) ist Element des Triset und wird auf den Punkt P'(X',Y',Z') auf dem Zylindermantel abgebildet.
P' wird wiederum in P" (U,V) in der Texturebene transformiert.
Die Zuordnung Zylindermantel → Textur kann man sich anschaulich machen, indem man sich die Texturebene als Blatt vorstellt, das man um den Zylinder wickelt. Man erkennt, daß V lediglich von der Y-Koordinate abhängt, während U eine Funktion von X, Z und _ ist.
Man erhält folgende Beziehungen:

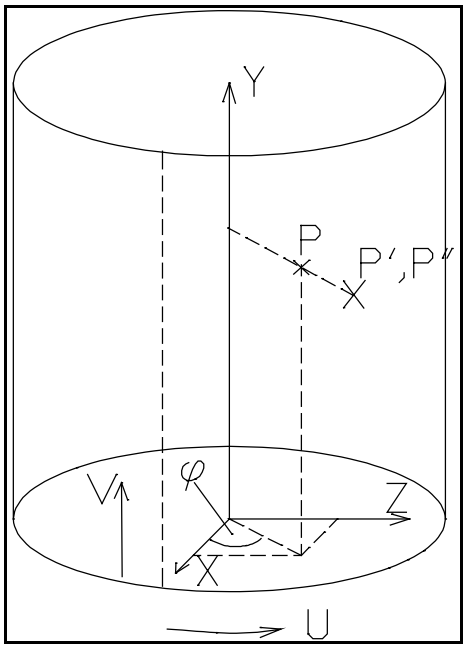


Abb. V.7 Two-Part-Mapping, Cylinder, ISN

$$v = \frac{v_{max}}{y_{max}} \bullet y$$

$$u = \frac{u_{max}}{2\pi} \bullet \varphi\,; \quad \text{mit } \varphi = \arctan\left(\frac{x}{z}\right) \qquad \textbf{(V.7)}$$

$$= \frac{u_{max}}{2\pi} \bullet \arctan\left(\frac{x}{z}\right)$$

Da die Koordinaten durch die jeweiligen Maximalwerte (2π, bzw. Y_{max}) geteilt werden, erhält man Werte im Intervall [0, 1].
Die beschriebene Form des Two-Part-Mapping ist in der umgekehrten Richtung auch unter dem Namen 'Shrinkwrap-Mapping' bekannt (vgl. [Bier86], [Watt89] S.238 ff).

3.Kugel, Centroid

In 8 ist O^{-1}- und S^{-1}-Mapping nach dem Centroidverfahren für eine Kugelfläche zu sehen. Die Berechnung der U-Koordinate von P" erfolgt wie beim Zylinder/ISN-Mapping:

$$u = \frac{u_{\max}}{2\pi} \bullet \arctan\left(\frac{x}{z}\right)$$

Für die Berechnung der V-Koordinate müssen Y und ' herangezogen werden:

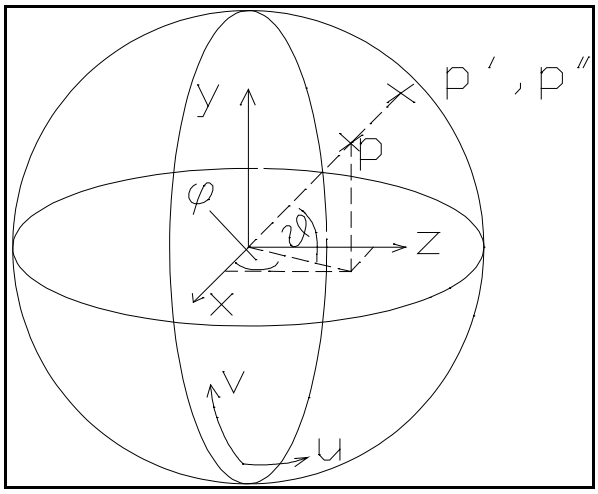


Abb. V.8 Two-Part-Mapping, Kugel, Centroid

$$\tan\vartheta = \frac{y^*}{\sqrt{x^2 + z^2}}; \quad mit\ y^* = y - \frac{y_{\max}}{2}$$

$$\_\vartheta = \arctan\left(\frac{y - \frac{y_{\max}}{2}}{\sqrt{x^2 + z^2}}\right)$$

wobei Y^* benötigt wird, um das Koordinatensystem in die Kugelmitte zu verschieben. V^* wird zunächst relativ zum 'Äquator' berechnet. Da V positiv sein soll, wird ein Offset dazu addiert:

$$\Delta v = \frac{v_{\max}}{\pi} \bullet \vartheta$$

$$v = \Delta v + \frac{v_{\max}}{2}$$

V ergibt sich damit zu

$$v = \frac{v_{\max}}{\pi} \arctan\left(\frac{y - \frac{y_{\max}}{2}}{\sqrt{x^2 + z^2}}\right) + \frac{v_{\max}}{2}$$

B.Interpolation der Farben

Das Two-Part Mapping wird für jeden Vertex des Triset durchgeführt. Damit sind alle Farbinformationen in der Textur abgespeichert. Um in der Textur gleichmäßige Farbverläufe zu erhalten, ist es notwendig, die Lüken zwischen den Punkten auszufüllen.
Aus diesem Grund transformiere ich jeweils drei Eckpunkte eines Face auf einmal in die Textur. Das entstandene Dreieck in der U,V-Ebene wird ausgefüllt, indem seine Eckfarben linear interpoliert werden[29].

C.Schreiben der Texturdaten

Die Textur wird als geräteunabhängige Bitmap im Windows-BMP Format abgespeichert. Dabei werden für jeden Punkt Rot-, Grün- und Blauanteil als Byte abgelegt. Es sind auf diese Weise 2^{24} Farben darstellbar, so daß keine Farbinformation des Digitizers verloren geht.
Damit bei der Weiterverarbeitung durch nachfolgende Programme eine Zuordnung zwischen Textur und Trimesh möglich ist, werden beim Schreiben der Geometriedaten zu jedem Vertex neben den Raumkoordinaten zusätzlich die Texturparameter (U, V) ausgegeben.

D.Reduktion der Texturdaten

Die Reduktion der Farbinformationen soll nicht Thema meiner Diplomarbeit sein. Die Größe der Textur-Bitmap läßt sich mit bestehenden Programmen effizient verringern.
Die Textur wird in sehr hoher Qualität gespeichert (True Color mit über $16 \cdot 10^6$ möglichen Farben). Oft reicht es aus eine Darstellung zu wählen, in der nur 256 aus $16 \cdot 10^6$ Farben benutzt werden[30]. In diesem Fall kann die Bilddatei beispielsweise über einen Median Cut Algorithmus auf ca. ein drittel komprimiert werden[31].

---

29 Ein schneller Algorithmus wird in [Gems90] S.361 ff. beschrieben.

30 Zumal die Hardware oft garnicht mehr Farben darstellen kann.

31 vgl. Diplomarbeit [Micha91].

VI.Reduktion der Geometrieinformation

A.Vorüberlegungen

Als wichtiges Ziel der Datennachbearbeitung wurde bereits die Datenreduktion genannt. Erst dadurch wird eine Weiterverarbeitung der Daten mit vertretbarem Speicher-, Hardware- und Zeitaufwand möglich.
Nun müssen Randbedingungen definiert werden, innerhalb derer die Reduktion verläuft. Die wichtigste Randbedingung ist offensichtlich:

> *Das reduzierte Triset soll dem Original ähnlich sein.*

Diese 'Ähnlichkeit' muß nur noch im Bezug auf die Geometrie gewährleistet sein. Die Farbe und bis zu einem gewissen Grad auch die Oberflächenstruktur wurde ja bereits in einem vorangegangenen Schritt als Textur in einer Bilddatei abgespeichert. Als Maß der Ähnlichkeit sind mehrere Kennzahlen denkbar:

1. Das Volumen zwischen der alten und der reduzierten Oberfläche.
2. Der Abstand zwischen den gelöschten (wegreduzierten) Vertizes und der neuen Oberfläche.

Weitere Randbedingungen möchte ich zunächst in einem allgemeinen Satz zusammenfassen:

> *Das reduzierte Triset soll eine hohe Qualität haben.*

An das reduzierte Triset werden bestimmte Anforderungen gestellt ('Restriktionen'). Gesucht ist dann das kleinste Triset (Anzahl der Vertizes N_V = minimal), das diesen Restriktionen genügt. Es handelt sich bei der Reduktion also um eine Optimierungsaufgabe.
Ich habe zwei Stufen der Reduktion vorgesehen, die unabhängig voneinander eingesetzt werden können.
Die erste Stufe (Reduktion der Polygonzüge) kann schon während des Einlesens der Daten angreifen (Filterfunktion). Vertizes, die bereits an dieser Stelle herausgefiltert werden, gelangen garnicht erst in das Triset. Diese Vorfilterung ist relativ einfach zu realisieren und kann große Eingangsdaten auf ein handhabbares Maß reduzieren.
Die zweite Stufe (Reduktion des Triset) versucht eine Minimierung des gesamten Triset.
Grundsätzlich gehe ich davon aus, daß die Eingabedaten korrekt sind. Das heißt, wenn im Datensatz einzelne Punkte besonders aus der Oberfläche hervortreten, werden diese als wichtige Bestandteile der Oberfläche auch im reduzierten Triset zu finden sein. Es ist deswegen die Aufgabe der vorhergehenden Programme dafür zu sorgen, daß der Datensatz keine ungewollten Ausreißer enthält[32].

[32] Dies kann durch eine geeignete Filterung erreicht werden.

Ich gehe von korrekten Eingangsdaten aus.

B.Reduktion der Polygonzüge

Das Auffinden eines (lokalen oder globalen) Optimums ist ein kombinatorisches Problem, dessen Lösung, wenn überhaupt, nur unter großem Zeitaufwand möglich ist. Als eine erste Reduktionsstufe habe ich deswegen einen Filter entwikelt, der die Datenmenge schon beim Einlesen verkleinert. Als Parameter sind zwei Vorgaben vorgesehen:

1. der maximale Fehler ε_{max}, um den der reduzierte Polygonzug von dem Original abweichen darf, und
2. der maximal zulässige Abstand l_{max} zwischen zwei Vertizes. Damit kann verhindert werden, daß die Maschen des eingelesenen Trisets zu groß werden, was den Spielraum für die nachfolgende globale Reduktion begrenzen würde.

Ausgehend von einem Polygonzug P, bestehend aus n Vertizes V_1 bis V_n sollen Teile von P durch Geradenstücke G ersetzt werden. Allgemein:
Das Teilpolygon $P_{a,b}$, bestehend aus den Vertizes V_a bis V_b soll durch ein Geradenstück $G_{a,b}$ ersetzt werden.
Um die oben genannten Restriktionen einzuhalten, muß gelten:

1. Keiner der Punkte V_a bis V_b darf weiter als ε_{max} von $G_{a,b}$ entfernt sein, und
2. der Abstand zwischen V_a und V_b muß kleiner gleich l_{max} sein

Beginnend mit einem Punkt V_i prüfe ich schrittweise, ob die folgenden Punkte durch ein Geradenstück ersetzt werden können, indem ich den Teilpolygonzug $P_{i,i+2}$ durch die Gerade $G_{i,i+2}$ ersetze und auf Einhaltung der Restriktionen prüfe. Ist diese Vereinfachung erlaubt, darf der Punkt V_{i+1} überbrückt werden, und ich merke mir den Index X=i+2 als gültigen Endpunkt der Ersatzgeraden G (vgl. 9 a.).

Als nächstes versuche ich zwei Punkte (V_{i+1}, V_{i+2}) durch eine Gerade $G_{i,i+3}$ zu ersetzen. Ist das möglich, erhält X den Wert i+3 (vgl. 9 b.).
Auf diese Weise kann fortgefahren werden, bis das Ende des Polygonzugs erreicht ist. Die längste erlaubte Ersatzgerade ist dann $G_{i,x}$, die Punkte V_{i+1} bis V_{x-1} werden gelöscht (Im Beispiel X=i+6).
Wenn die Ersatzgerade G nicht bis zum Ende des Polygonzugs reicht (X<n), dann wird X als Startpunkt einer weiteren Geraden gewählt und von vorne begonnen.
Durch die Wahl einer sinnvollen Abbruchbedingung kann die Suche nach einem gültigen Endpunkt V_x beschleunigt werden:

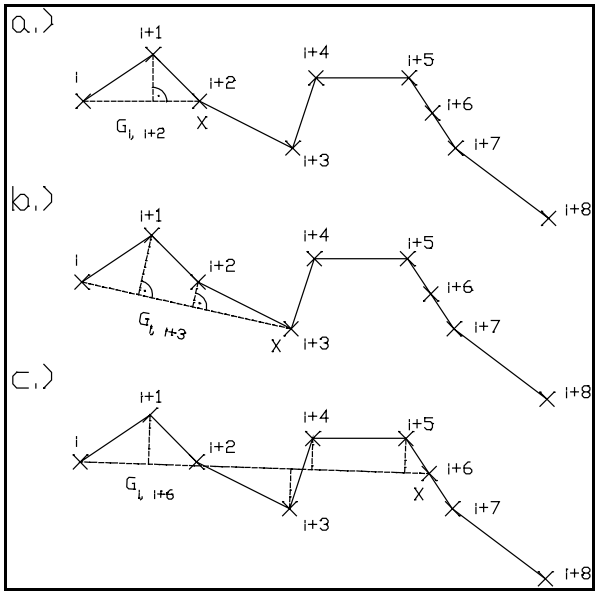


Abb. VI.9 Reduktion eines Polygonzugs

1. Wenn die Länge der betrachteten Ersatzgeraden $G_{i,j}$ größer als l_{max} ist, wird dies wahrscheinlich auch für $G_{i,j+1}$ gelten, d.h. die Suche kann abgebrochen werden.
2. Wenn der Fehler der durch $G_{i,j}$ entstehen würde größer ist $2 \cdot \varepsilon_{max}$, dann kann er für $G_{i,j+1}$ nicht mehr kleiner werden als ε_{max}. Die Suche wird ebenfalls abgebrochen. In 9 b. ist der Abstand zwischen der Geraden und V_{i+1} beispielsweise größer als ε_{max}. Trotzdem ist eine spätere gültige Lösung möglich, weil die Gerade wieder 'zurückgependelt' ist (Fall c.). So ein zurückpendeln ist allerdings unwahrscheinlich, wenn der Fehler bereits mehr als doppelt so groß ist wie ε_{max}.

C.Reduktion des Trisets (lokale Optimierung)

Nachdem die gefilterten Eingangsdaten eingelesen und in ein Trimesh konvertiert wurden, beginnt nun der Hauptteil der Arbeit. Dieses immer noch sehr feinmaschige Netz muß derart reduziert werden, daß ein wesentlich grobmaschigeres Netz entsteht, welches aber dennoch dem Original ähnlich ist.
Wie bereits in den Vorüberlegungen zu diesem Kapitel erwähnt, werden an das neue Triset Bedingungen gestellt, die der Benutzer in Form von Restriktionen definiert. diese Restriktionen sind:

1. Eine maximale Abweichung ε_{max} der neuen Fläche von der Originalfläche
2. Eine maximale Seitenlänge l_{max} der neuen Faces
3. Die neuen Faces sollen nicht zu schmal werden (optimal sind gleichseitige Dreiecke). D.h. die Proportion der Faces muß ausgewogen sein.
4. Die Konsistenz im Trimesh muß gewahrt bleiben

Die Variablen ε_{max} und l_{max} können hierbei andere Werte annehmen als bei der Reduktion der Polygonzüge. Dabei ist zu beachten, daß der resultierende maximale Gesamtfehler ε_{Σ} gleich der Summe der Einzelfehler ε_{max} der beiden Reduktionsstufen ist.
Die Restriktion 1. ist sofort einsichtig, denn mit ihr kann der Benutzer selber entscheiden, welche Abweichungen vom Original er in Kauf zu nehmen bereit ist um die Reduktionsrate zu erhöhen.
Die Restriktionen 2. und 3. machen Sinn, wenn man auf ein gleichmäßiges Aussehen des Triset Wert legt. Außerdem eignen sich extrem schmale Dreiecke nicht für das Texture-Mapping. Ich werde die Begriffe maximale Abweichung und Proportion später in diesem Kapitel definieren.
Restriktion 4. werde ich ebenfalls in diesem Kapitel erläutern, wenn ich auf die Topologie des Trimesh eingehe (Kapitel VI.C.4).
Hat man erst einmal die Randbedingungen festgelegt, kann damit begonnen werden, Vertizes aus dem Trimesh zu entfernen.
Die Vorgehensweise nach diesem Konzept ist in folgenden Algorithmus beschrieben.

```
DO
        Wähle einen Vertex des Trisets
        IF NOT (Löschung würde eine Restriktion verletzen)
        THEN
                Vertex löschen
        ENDIF
WHILE NOT Endekriterium
```

Alg. VI,4 Direkte Reduktion

Wählt man hierbei das Endekriterum so, daß erst abgebrochen wird, wenn keine weitere Löschung mehr möglich ist, so erhält man als Lösung ein Triset mit einer minimalen Anzahl von Punkten.
Allerdings ist es möglich, daß eine andere Reihenfolge bei der Auswahl der Testkandidaten günstiger gewesen wäre und zu einer höhere Reduktionsrate geführt hätte.
Ein prinzipieller Nachteil dieses Verfahrens kann also schon festgehalten werden:

*Es wird nur ein **lokales** Optimum gefunden !*

Dieser Algorithmus strebt direkt auf ein (lokales) Minimum zu. Ich möchte ihn deswegen als 'direkte Reduktion' bezeichnen. Trotz der Einschränkung können auch hiermit recht gute Ergebnisse erzielt werden. Ich werde zunächst anhand dieses Konzepts einige Probleme der Reduktion erklären.
Später wird dieses Verfahren dann erweitert, so daß auf Kosten einer längeren Laufzeit (zumindest theoretisch) ein globales Optimum gefunden wird.
Erst einmal aber zu den einzelnen Schritten der 'direkten Reduktion'.

1.Auswahl eines Kandidaten für eine Löschung

Die einfachste Möglichkeit wäre natürlich, alle Vertizes in der Reihenfolge wie sie im Triset abgelegt sind, als mögliche Kandidaten für eine Löschung zu wählen. Versuche haben aber gezeigt, daß die Qualität des Reduktionsergebnisses von der Reihenfolge der Löschungen abhängt. Um zu verhindern, daß das Programm für einen Datensatz immer das gleiche Ergebnis liefert, ist es deswegen sinnvoll, den Zufall ins Spiel zu bringen.
Andererseits kann ein Triset sehr groß werden, möglicherweise so groß, daß der Hauptspeicher nicht ausreicht. Unter Windows ist das kein Problem, weil nicht benötigte Speicherseiten auf Massenspeicher ausgelagert werden können ('Paging'). Ein völlig wahlfreier Zugriff auf ein teilweise ausgelagertes Triset würde aber zu intensiver Pagingtätigkeit führen und damit die Verarbeitungsgeschwindigkeit stark sinken lassen.
Aus diesen Gründen habe ich mich für eine Kombination von zufälliger und sequentieller Auswahl entschieden:
Ich wähle zunächst einen Kandidaten aus, indem ich eine Zufallszahl I_n im Bereich zwischen 0 und N_V-1 (N_V: Anzahl der Vertizes) generiere. Über diesen Index wird der Kandidat dereferenziert.
Der nächste Kandidat hat einen Index I_{n+1} der zwischen I_n-a und I_n+b liegt.
Durch dieses begrenzte Intervall wird ein häufiges Wechseln der Segmente vermieden. Um trotzdem eine gleichmäßige Auswahlwahrscheinlichkeit aller Vertizes zu gewährleisten, wähle ich b > a. Im Mittel wird I_{n+1} also größer sein als I_n.
Durch eine Modulo Division wird sichergestellt, daß I_{n+1} immer im gültigen Intervall [0, N_V-1] liegt:

$$I_{N+1} = [\ I_N + \text{random}(a + b) - a\ ] \bmod N_V$$

Die Funktion **random(x)** liefert hier eine Zufallszahl $Z \in N$ aus dem Intervall [0, x-1].

2.Prüfen, ob Löschung möglich ist

Um festzustellen, ob das Löschen eines Vertex V erlaubt ist, muß man die Löschung zunächst durchführen und dann das so veränderte Triset überprüfen. Wenn dabei festgestellt wird, daß eine der Restriktionen verletzt wurde, wird die Löschung rückgängig gemacht.
Zunächst werde ich betrachten, wie sich das Löschen eines Punktes auf das Triset auswirkt.

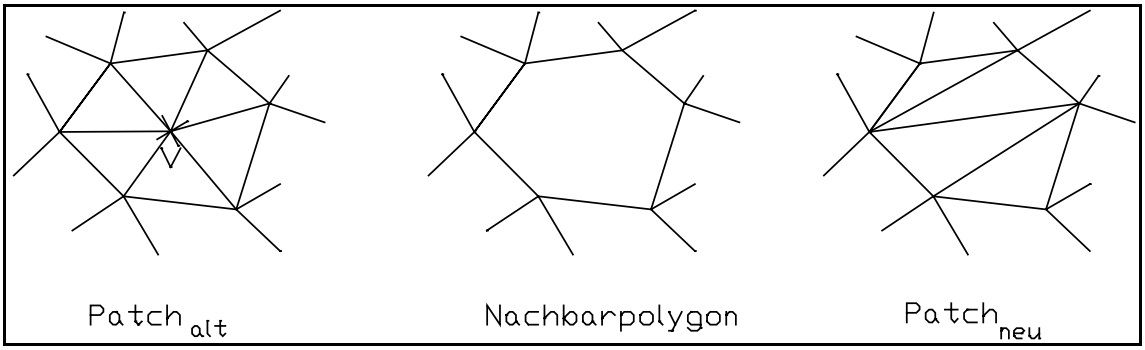


Abb. VI.10 Vertex im Triset löschen

In einem Triset kommt ein Vertex nur als Eckpunkt eines oder mehrerer Faces vor. Wird ein Punkt entfernt, dann sind auch die Faces, die diesen Punkt als Eckpunkt haben nicht mehr definiert. Es entsteht ein Loch im Triset (10). Dieses Loch muß durch neue Faces wieder geschlossen werden ('Tesselierung'). Es bleibt also festzuhalten:

1. Das Löschen eines Vertex wirkt sich im Triset nur auf seine unmittelbare Umgebung, die Nachbar-Faces, aus.
2. Beim Löschen eines Vertex werden im Triset $N_{F\,alt}$ Faces durch $N_{F\,neu}$ Faces ersetzt.

Sowohl die alten, als auch die neuen Faces grenzen aneinander, bilden also ein kleines Triset (im weiteren auch als '**Patches**' (Flicken) bezeichnet).
Während alle Faces im alten Patch den Vertex V als gemeinsamen Eckpunkt haben, kommt V im neuen Patch nicht mehr vor. Da aber das neue Patch das Loch schließt, welches das alte hinterlassen hat, sind beide Patches durch dasselbe Polygon begrenzt. Die Eckpunkte dieses Polygons sind die Nachbarvertizes von V.
Ein interessanter Aspekt ist die Anzahl der Faces im alten und neuen Patch. Im alten Patch gibt es genau so viele Faces, wie es Nachbarpunkte von V gibt (vgl. 10). Im Abschnitt über die Tesselierung werden wir sehen, daß ein Polygon mit N Eckpunkten durch N-2 Dreiecke trianguliert werden kann.

*Beim Löschen **eines** Vertex nimmt die Anzahl der Faces im Triset um **zwei** ab.*

Um zu überprüfen, ob ein Vertex V gelöscht werden darf, sind folgende Schritte nötig:

1. Nachbarfaces von V suchen ($\rightarrow Patch_{alt}$)
2. Randpolygon von $Patch_{alt}$ bestimmen ($\rightarrow$Nachbarpolygon)
3. Nachbarpolygon tesselieren ($\rightarrow Patch_{neu}$)
4. Prüfen, ob die Restriktionen verletzt werden.

Ich werde auf diese Punkte jetzt im einzelnen eingehen.

a.Finden der Nachbarn

In einem ersten Schritt müssen alle Nachbarfaces von V gefunden werden. Dazu wird jedes Face des Triset überprüft. Wenn es als einen Eckpunkt V enthält, dann gehört es zu $Patch_{alt}$ und wird in einer Liste gespeichert[33].

Das Nachbarpolygon P besteht aus den Eckpunkten der gefundenen Faces, die ungleich V sind (11). Jeder Punkt des Polygons kommt dabei in zwei Faces vor.

Da bei dem Polygon P die Reihenfolge seiner Eckpunkte P_1 bis P_n wichtig ist, müssen zunächst die Faces von $Patch_{alt}$ so sortiert werden, daß seine Faces F_1 bis F_m aneinandergrenzen:

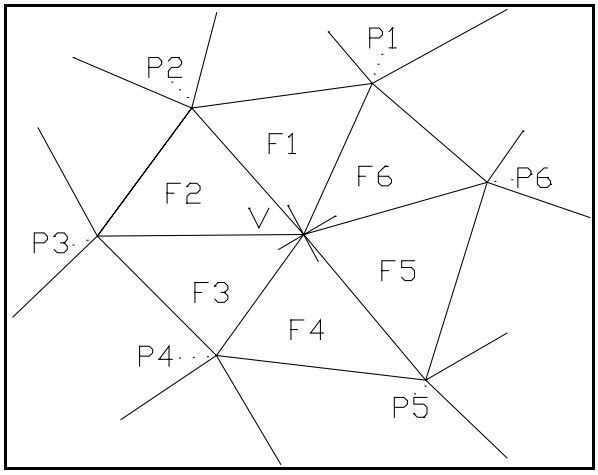


Abb. VI.11 Nachbarfaces, Nachbarpolygon

1. Das erste Face von $Patch_{alt}$ wird F_1.
2. Das nächste Face soll an F_1 grenzen, d.h. es wird ein Face gesucht, daß mit F_1 den Punkt V und einen weiteren gemeinsam hat. Dieses Face wird F_2.
3. F_2 hat mit F_1 also die Punkte V und P_2 gemeinsam. F_3 wird dann das Face, das mit F_2 die Punkte V und P_3 gemeinsam hat.
4. Dieser Vorgang wird fortgesetzt, bis ein Face F_m gefunden wurde, das an F_1 grenzt.

Beim Sortieren erhält man die Eckpunkte P_1 bis P_n des Nachbarpolygons P.

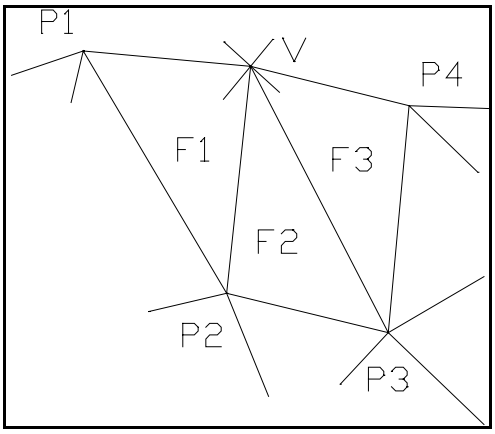


Abb. VI.12 Randpunkt

Einen Sonderfall tritt auf, wenn V ein Randpunkt im Triset ist: jetzt kann das Polygon nicht mehr geschlossen werden (12). Man kann auch sagen, daß V selbst ein Punkt des Polygons ist. In der Tat ist das die Stelle, an der Comp3D Randpunkte erkennt. Randpunkte werden durch ein Flag gekennzeichnet und erfahren eine Sonderbehandlung.

b.Tesselierung des Nachbarpolygons

Im nächsten Schritt wird das n-eckige Loch, das das alte Patch hinterlassen hat, durch eine Anzahl von dreieckigen Faces

[33] Obwohl diese sequentielle Suche trivial ist, wird hier ein großer Teil der gesamten Rechenzeit verbraucht, wenn das Triset groß ist.

geschlossen.
Die Aufgabe lautet also:

Triangulation eines Polygons P vom Grade N.

Dieses Polygon habe die Eckpunkte V_1 bis V_N.
Man kann den Grad des Polygons um eins verringern, indem man eine Ecke 'abschneidet': ein Vertex V_i wird abgetrennt, indem seine beiden Nachbarn V_{i-1} und V_{i+1} verbunden werden[34]. Es bleibt dann ein Polygon vom Grade N-1

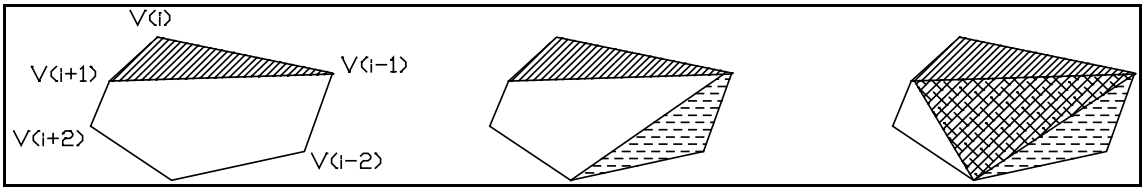


Abb. VI.13 Grad eines Polygons schrittweise verringern

und ein Dreieck (bestehend aus den Punkten V_{i-1},V_i und V_{i+1}) übrig (13).
Ein einfacher Weg ist es, solange eine Ecke von P abzutrennen, bis P nur noch vom Grad 3 ist, d.h., es ist nur noch ein Dreieck übriggeblieben. Um ein Polygon vom Grad N auf ein Dreieck zu vermindern, müssen demnach N-3 Vertizes abgetrennt werden, wobei N-3 Dreiecke entstehen. Zusammen mit dem dreieckigen Restpolygon sind also N-2 Faces entstanden:

Ein Polygon vom Grade N wird in N-2 Dreiecke trianguliert.

(1)Anforderungen an die Tesselierung

[34] Ich setze im weiteren voraus, daß der Nachfolger des letzten Vertex wieder V_1 ist. Der Vorgänger von V_1 sei V_N.

Nun gibt es allerdings mehrere Möglichkeiten ein Polygon in Dreiecke zu zerlegen.
Einige dieser Varianten könnten z.B. so schmale Dreiecke enthalten, daß die Restriktionen verletzt würden (14 a.).

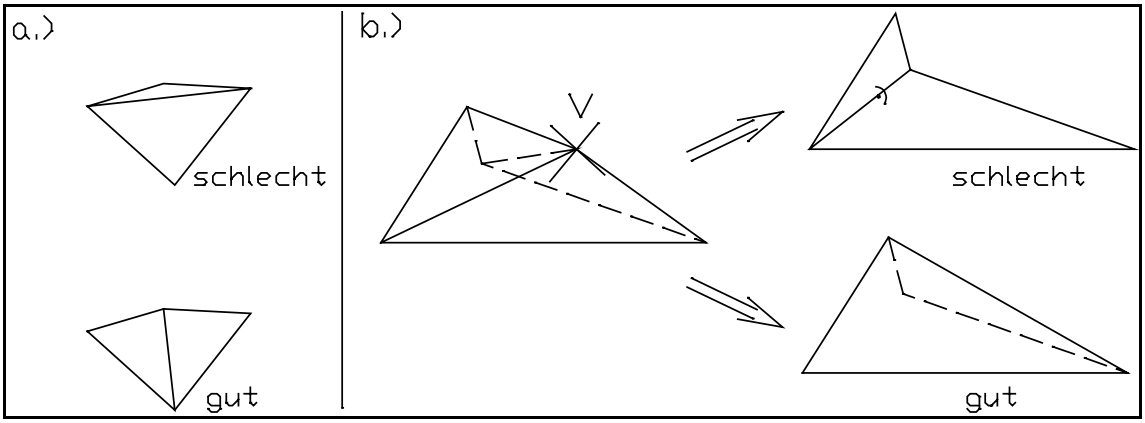


Abb. VI.14 Tesselierungsvarianten

Aber auch unter den erlaubten Varianten gibt es einige, die besser oder schlechter geeignet sind. In 14 b. ist ein Beispiel zu sehen[35]: Nachdem V gelöscht wurde, bleibt ein Randpolygon mit vier Ecken zurück, das auf zwei Arten tesseliert werden kann. Auch wenn beide Varianten die Restriktionen erfüllen, gibt eine davon die ursprüngliche Form wesentlich genauer wieder.
Ich definiere ein Auswahlkriterium:

> Das alte Patch soll durch diejenige Tesselierung seines Randpolygons ersetzt werden, die ihm am ähnlichsten ist, ohne dabei Restriktionen zu verletzen.

Als Maß für die Ähnlichkeit habe ich das Volumen gewählt, das von den beiden Patches eingeschlossen wird[36], weil sich das neue Patch dann am engsten an das alte Patch anschmiegt, wenn ΔV am kleinsten ist. Das genauere Auswahlkriterium lautet also:

> Unter allen gültigen Tesselierungen wird die gewählt, die kleinste Volumendifferenz ΔV zu $Patch_{alt}$ hat.

[35] Hierbei handelt es sich um eine räumliche Darstellung.

[36] Berechnung von ΔV im Anhang.

Bei späteren Versuchen hat sich gezeigt, daß spezielle Situationen zu einer unerwünschten Auswahl führen. Diese Sonderfälle treten bei planaren, konkaven Polygonen auf. Weil in diesem Fall $Patch_{alt}$ und $Patch_{neu}$ auf einer Ebene liegen, ist ΔV für alle Varianten Null und kann nicht als Auswahlkriterium dienen. Trotzdem gibt es bei konkaven Polygonen unerwünschte Tesselierungen, wie man in 15 sehen kann. Im ungünstigen Fall werden Teile des Polygons doppelt überdeckt[37], wenn sich einige Dreiecke überlappen. In einer optimalen Tesselierung überlappen sich die Faces nicht, die Fläche des Patches $A_\Sigma = \Sigma A_N$ ist minimal. Nun ist es theoretisch möglich, daß alle Lösungen in denen **keine** überlappenden Faces auftreten aus anderen Gründen nicht erlaubt sind (weil sie gegen Restriktionen verstoßen). Es wäre aber keine gute Idee in so einem Fall überlappende Faces in Kauf zu nehmen. Besser ist es, die Tesselierung in so einem Fall zurückzuweisen, d.h. V darf nicht gelöscht werden.

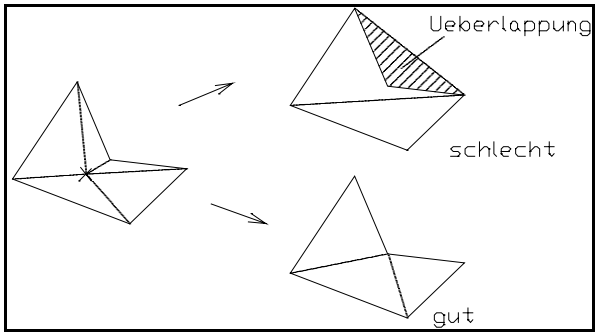


Abb. VI.15 Konkave, planare Polygone

Ich formuliere die endgültige Auswahlbedingung:

> *Unter allen Varianten für die gilt: ΔV=min, wird diejenige mit der kleinsten Fläche A_Σ gesucht. Wenn diese Variante gegen keine Restriktion verstößt, wird sie gewählt. Anderenfalls ist eine Tesselierung nicht möglich.*

Versuche haben gezeigt, daß dieses Kriterium gute Ergebnisse liefert.

(2)Generierung von Patchvarianten

Um für ein Polygon eine optimale Triangulation zu finden, müssen die Varianten zunächst generiert, und dann nach dem oben genannten Kriterium bewertet werden. Ich werde mich jetzt mit dem Erzeugen von Patchvarianten beschäftigen.

[37] In der Abbildung schraffiert gezeichnet.

Ein einfacher rekursiver Ansatz zur Triangulation ist im Algorithmus zu sehen. Bevor die Prozedur **Tesselate1** aufgerufen wird müssen zunächst die globalen Datenstrukturen zur Speicherung von Patches geleert werden. Das Polygon wird um einen Grad vermindert indem jeder seiner Eckpunkte nacheinander abgetrennt wird, ein global gespeichertes $Patch_{aktuell}$ wird um das abgetrennte Face vergrößert. Für das nun kleinere Polygon wird die Funktion rekursiv aufgerufen. Die Rekursion wird beendet, wenn das Polygon nicht mehr vermindert werden kann (Grad N=3). $Patch_{aktuell}$ enthält jetzt eine Triangulationsvariante, die bewertet wird.
Dieses Verfahren funktioniert zwar, stößt aber schnell an Rechenzeitgrenzen: bei einem Polygon vom Grad N gibt es N verschiedene Ecken die abgetrennt werden können. Das verbleibende Polygon bietet immer noch N-1 Ecken für den nächsten Schnitt, usw. Solange ein Polygon mehr als 3 Vertizes hat, wird einer davon abgetrennt. Es gibt also

$$N \bullet (N - 1) \bullet (N - 2) \bullet \ldots \bullet 4 = \frac{N!}{3 \bullet 2 \bullet 1} = \frac{N!}{6}$$

Möglichkeiten der Tesselierung.
Für ein Polygon vom Grad 10 ergeben sich damit z.B. schon mehr als 600.000 Varianten. Es wäre sinnlos, einen dermaßen großen Rechenaufwand zu betreiben, nur um zu prüfen ob ein Punkt gelöscht werden darf oder nicht. Andererseits kann man es nicht riskieren, eine Tesselierung zu wählen, die nicht optimal ist, weil dann unschöne, auffällige Stellen im Triset entstehen können.
Der Aufwand läßt sich auf zwei Arten begrenzen:

1. Wenn ein Punkt mehr als N_{max} Nachbarpunkte besitzt, wird er als 'nicht löschbar' bewertet.
2. Der oben gezeigte erste Ansatz liefert zwar alle möglichen Triangulationen, viele werden aber doppelt oder öfter generiert. Diese Mehrfachbewertungen sind auszuschließen.

Methode 1. wirkt recht unsensibel, hat aber keine so drastischen Auswirkungen auf die Reduktion wie man vielleicht meinen könnte. Zum einen nimmt die Wahrscheinlichkeit, daß ein Vertex gelöscht werden darf ohnehin mit der Zahl seiner Nachbarn ab, weil mehr Faces entstehen, die alle den Restriktionen genügen müssen. Zum anderen verläuft die Auswahl der Vertizes ja zufällig. Es ist also möglich, daß so ein Punkt zunächst stehengelassen wird, seine Nachbarn aber im weiteren Verlauf der Reduktion gelöscht werden. Zu einem späteren Zeitpunkt kann

```
PROZEDUR Tesselate1
PARAMETER:      Polygon

N ← Anz. der Punkte in Polygon
IF N=3 THEN
        Erweitere Patch_aktuell um Polygon
        IF (Patch_aktuell ist besser als Patch_optimal) THEN
                Patch_optimal ← Patch_aktuell
        entferne das letzte Dreieck aus Patch_aktuell
        ENDIF
ELSE
        FOR I=1 TO N
                Lösche Punkt I in Polygon
                Erweitere Patch_aktuell um das abgetrennte Dreieck
                CALL Tesselate1 (Polygon)
                entferne das letzte Dreieck aus Patch_aktuell
                Füge Punkt I wieder zum Polygon hinzu
        ENDFOR
ENDIF
```

sich die Anzahl seiner Nachbarn so verringert haben, daß er endlich auf eine mögliche Löschung hin untersucht werden darf[38].

Um das Generieren von doppelten Varianten zu verhindern, betrachte ich zunächst deren Entstehung.

Ein Polygon vom Grade N=4 läßt sich auf zwei verschieden Arten in zwei Dreieke zerlegen, der Algorithmus liefert aber vier Varianten, weil es vier Ecken gibt, die abgetrennt werden können[39].

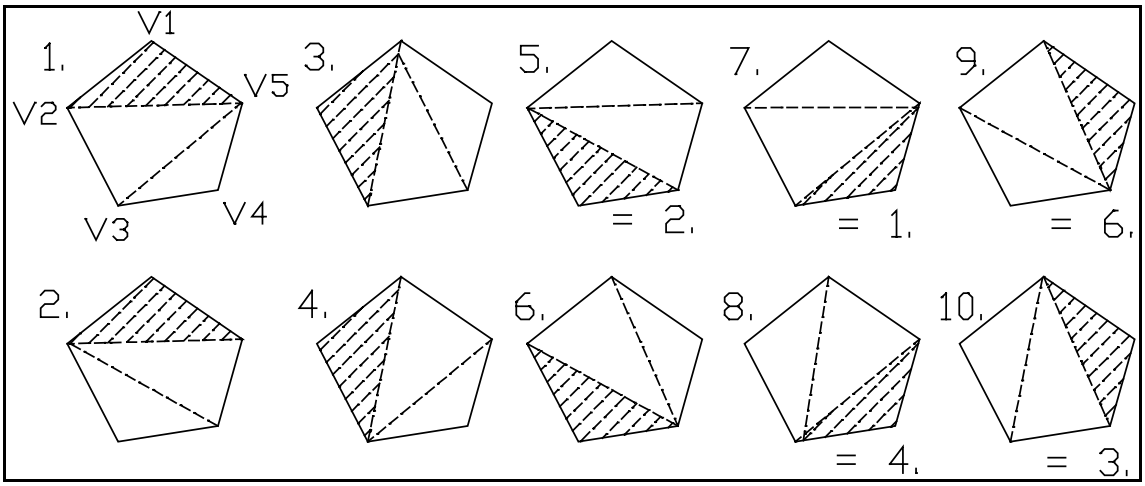


Abb. VI.16 Tesselierungsmöglichkeiten eines Fünfecks

Für ein Fünfeck werden 5!/6 = 20 Triangulationen generiert, unter denen aber nur fünf Verschiedene sind (16)[40]. Eine nähere Betrachtung des oben beschriebenen Algorithmus am Beispiel eines Fünfecks mit den Vertizes V_1 bis V_5 erklärt dieses Verhalten. Zunächst wird V_1 abgetrennt: es bleibt ein Viereck, für das rekursiv alle möglichen Tesselierungen generiert werden. Dann wird V_2 abgetrennt und wieder alle möglichen Variationen für das Restpolygon durchprobiert. Danach das gleiche Spiel mit V_3. Aber genau jetzt wird es kritisch: jetzt werden die ersten doppelten Varianten erzeugt.

Alle Varianten, die durch ein Abtrennen von V_1 eingeleitet wurden beinhalten das Dreieck _(V_1, V_2, V_5). Alle Tesselierungen, die mit dem Entfernen von V_2 begonnen wurden, enthalten _(V_1, V_2, V_3). Diese Dreiecke haben beide die identische Außenseite V_1,V_2 und können deswegen niemals gleichzeitig in einem Patch auftreten. Durch Abtrennen von V_1 und V_2 werden also Patches erzeugt, die sich immer in mindestens einem Face voneinander unterscheiden. Damit sind auch diese Patchvarianten alle verschieden.

Beginnt man die Tesselierung mit dem Entfernen von V_3 enthalten alle Folgetesselierungen das Dreieck _(V_2, V_3, V_4). Dieses kann aber auch in einer der Folgetesselierungen nach dem Abtrennen von V_1 enthalten sein.

Aus dem oben Gesagten folgt, daß für jede Triangulierung des Polygons V_1 bis V_N gilt:

---

38 In der Implementation von 'Comp3d' habe ich N_{max} auf 7 gesetzt.

39 Dieser doppelte Aufwand wirkt sich natürlich auch auf Polygone vom Grad 5 und höher aus, weil diese rekursiv bis zum Grad 4 reduziert werden.

40 In der Abbildung sind die doppelten Tesselierungen für ein Polygon vom Grad 4 nicht gezeichnet worden.

1. Jede Triangulierung enthält ein Dreieck, das die Streke V_1,V_2 als eine Seite hat.
2. In einer Triangulierung gibt es nur **genau** ein Dreieck daß die Strecke V_1,V_2 als eine Seite hat.

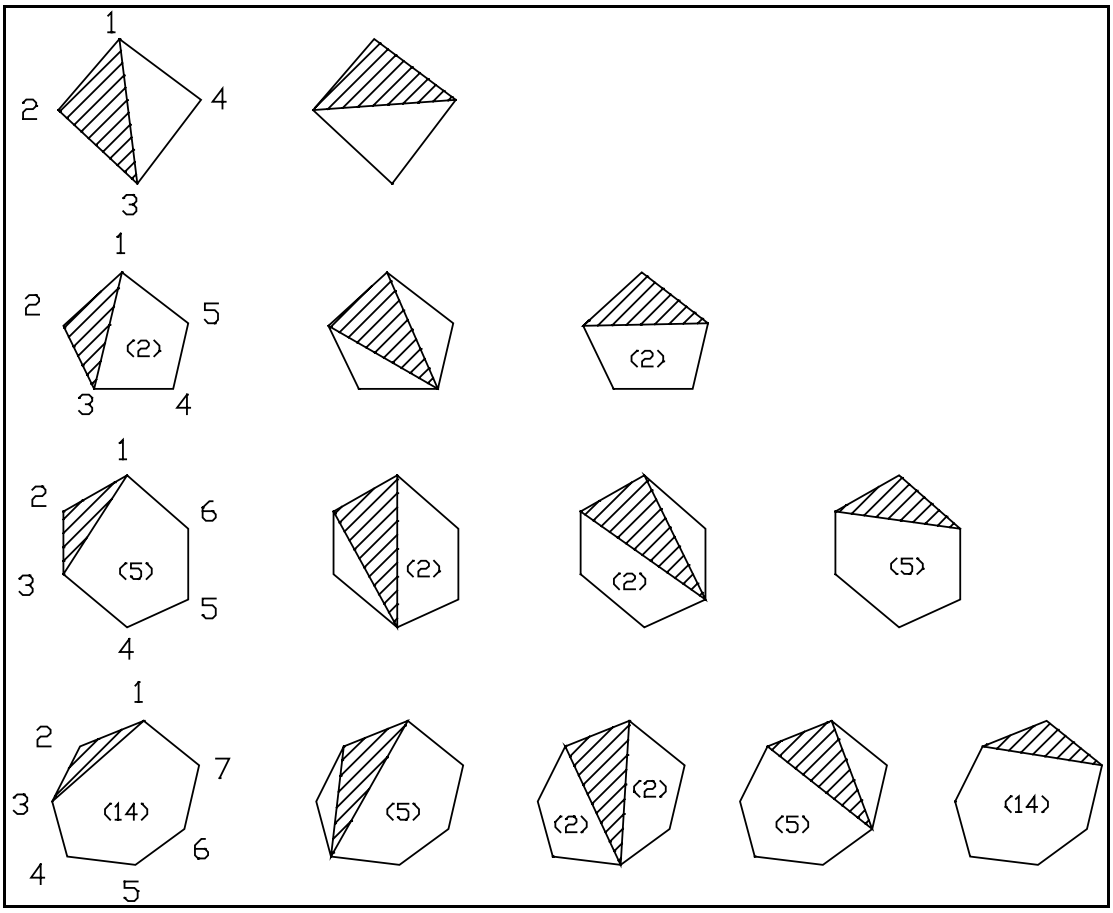


Abb. VI.17 Tesselierung durch eindeutiges Dreieck 1,2,i

Betrachtet man für das Polygon alle Varianten, die das Dreieck $\_(V_1,V_2,V_i)$ für $\{i \mid 3 \le i \le N\}$ beinhalten, enthält man alle Tesselierungen, ohne daß dabei doppelte auftreten können. Die Restpolygone, die dabei entstehen, müssen rekursiv auf die gleiche Weise behandelt werden.

In 17 sind die möglichen Dreiecke $\_(V_1,V_2,V_i)$ für Polygone vom Grad vier, fünf, sechs und sieben zu sehen. Die Zahlen in Klammern geben die Anzahl der möglichen Tesselierungsvarianten für die Restpolygone an. Für ein Polygon vom Grad sieben gibt es z.B.

$$T_7 = 14 + 5 + 2 \cdot 2 + 5 + 14 = 42$$

mögliche Varianten (nach dem ersten Verfahren waren es 7!/6 = 840).

Der optimierte Algorithmus ist im Folgenden beschrieben.

```
PROZEDUR Tesselate2
PARAMETER:        Polygon

N ← Anz. der Punkte in Polygon
IF (N>Nmax) OR () THEN
   EXIT
ENDIF
IF N=3 THEN
        Erweitere Patchaktuell um Polygon
        IF (Patchaktuell ist besser als Patchoptimal) THEN
                Patchoptimal ← Patchaktuell
        entferne das letzte Dreieck aus Patchaktuell
        ENDIF
ELSE
        FOR i=3 TO N
                Erweitere Patchaktuell um _(V1,V2,Vi)
                PolygonA ← V2,V3, .. ,Vi
                PolygonB ← V1,Vi,Vi+1 .. ,V1
                Für jede Tesselierung von PolygonA
                   CALL Tesselate2 (PolygonB)
                entferne das letzte Dreieck aus Patchaktuell
        ENDFOR
ENDIF
```

Alg. VI,6 Triangulation ohne Erzeugung redundanter Varianten

3.Restriktionen überprüfen

Wir sind jetzt an folgendem Punkt angelangt:

1. Es wurde ein Vertex ausgewählt, für den geprüft werden soll, ob seine Löschung erlaubt ist.
2. Es wurden seine Nachbarfaces und -vertizes gefunden.
3. Aus den Nachbarvertizes wurde ein Nachbarpolygon generiert.
4. Für das Polygon wurde eine, nach definierten Kriterien, optimale neue Tesselierung erzeugt ($\rightarrow Patch_{neu}$).

Nun muß entschieden werden, ob $Patch_{neu}$ die alte Tesselierung ersetzen darf, oder ob dabei Benutzervorgaben ('Restriktionen') verletzt würden.
Ich möchte jetzt noch einmal genauer auf die Restriktionen zu sprechen kommen.

a.Maschengröße, Proportion

Zwei Randbedingungen, nämlich 'Faces nicht zu schmal' und 'Seitenlänge der Faces nicht zu groß' betreffen die

Dreiecke in $Patch_{neu}$ und lassen sich relativ leicht überprüfen. Es gilt dabei: wenn auch nur eines der Faces in $Patch_{neu}$ zu schmal oder zu groß ist, dann ist damit die gesamte Tesselierungsvariante ungültig.
Die Bedingung für 'Seitenlänge kleiner gleich l_{max}' ist trivial. Um allerdings zu schmale Dreiecke auszusondern muß zunächst eine Bewertungsfunktion für die Proportion eines Dreiecks definiert werden. Ein Maß für die Proportion eines Dreiecks ist das Verhältnis von der längsten Seite s_{max} zur Höhe über der längsten Seite H_{smax}:

$$\text{proportion} = \frac{s_{max}}{H_{s_{max}}} \qquad \textbf{(VI.14)}$$

Dieser Wert wird für ein sehr schmales Dreieck sehr groß werden.
Der kleinste Wert wird für ein gleichseitiges Dreieck geliefert (18):

$$H^2 = s^2 - \frac{S^2}{4}$$

$$H = \frac{\sqrt{3}}{2} s \qquad \textbf{(VI.15)}$$

$$\text{proportion}_{min} = \frac{s_{max}}{H_{s_{max}}} = \frac{2}{\sqrt{3}} \approx 1{,}15$$

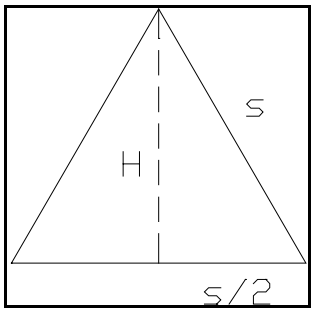


Abb. VI.18 Proportion

b.Maximaler Fehler

Etwas schwieriger wird es, wenn man die Einhaltung eines maximalen Fehlers ε_{max} garantieren möchte.
Zunächst definiere ich den Fehler ε[41]:

> *Der Fehler ε der entsteht, wenn ein Punkt V gelöscht wird, ist gleich dem Abstand zwischen V und dem Face, daß ihm am nächsten ist.*

Nachdem $Patch_{neu}$ erzeugt wurde müssen also die Abstände ε_i zwischen V und jedem Face der Tesselierung berechnet werden. V darf nur gelöscht werden, wenn gilt: $\min(\varepsilon_i) \leq \varepsilon_{max}$. In Worten ausgedrückt: Es muß wenigstens **ein** Face geben, das zu V einen Abstand hat, der kleiner oder gleich dem Maximalfehler ist.
Damit allein ist es aber noch nicht getan: zwar ist sichergestellt, daß beim Löschen **dieses** Punktes kein zu großer Fehler entsteht. Wenn ein maximaler Fehler ε_{max} definiert wird, erwartet man aber, daß am Ende der Reduktion **kein einziger** der gelöschten Punkte um mehr als diesen Betrag von der Oberfläche des Objektes entfernt ist. Um zu verhindern, daß sich die Einzelfehler, die beim Löschen jedes Punktes entstehen, zu einem (größeren) Gesamtfehler kumulieren, wird ein gelöschter Punkt intern nicht wirklich aus dem Speicher entfernt, sondern nur als gelöscht

[41] Die genaue Berechnung zeige ich im Anhang.

markiert. Ich bezeichne die gelöschten Vertizes deswegen auch als 'versteckte' oder 'unsichtbare' Punkte[42].
Jedesmal, wenn ein Vertex entfernt werden soll, wird vorher geprüft ob er selbst oder einer der unsichtbaren Punkte zu weit von der neuen Oberfläche entfernt ist.
Um den Rechenaufwand für diese Vergleichsoperationen möglichst klein zu halten, verwalte ich die gelöschten Punkte in sogenannten Hidden Vertex Listen. Dabei handelt es sich um dynamische Arrays, die für jedes Face des Triset deklariert sind. Wann immer ein Vertex gelöscht wird, wird er in die Hidden Vertex Liste des nächstgelegenen Face aufgenommen. Wann immer ein Patch durch ein Neues ersetzt wird, werden alle Hidden Vertices der alten Faces auf die jeweils nächstgelegenen neuen Faces verteilt.
Jedes Face des reduzierten Triset 'weiß' also immer, welche unsichtbare Punkte ihm am nächsten sind. Wenn ein weiterer Vertex gelöscht werden soll, dann wird nur er selbst und alle Versteckten Punkte, die seinen Nachbarfaces zugeordnet sind für eine Überprüfung der maximalen Abweichung herangezogen.

c.Randpunkte

Als es um das Auffinden der Nachbarpunkte eines Vertex ging, habe ich gezeigt, daß sich bereits hier Randpunkte identifizieren lassen.
In der Praxis zeigt sich, daß die Ränder eines Objektes durch die Reduktion oft 'ausfransen'. Optional kann das Löschen dieser Randpunkte deswegen unterbunden werden.

d.Algorithmus

Ich komme nun zur praktischen Umsetzung der bisher gewonnen Erkenntnisse.

```
PROZEDUR reduceDirect
PARAMETER:      -

REPEAT
        DO
                Sichtbaren Punkt auswählen → V
                Suche Nachbarfaces → Patch_alt
                Generiere Nachbarpolygon
        WHILE ( V ist kein Randpunkt )
        Tesseliere Nachbarpolygon → Patch_neu
        Verteile die Hidden Vertices von Patch_alt auf
                die Faces von Patch_neu
        Ordne V dem nächsten Face aus Patch_neu zu
        IF (Patch_neu erfüllt alle Restriktionen) THEN
                Ersetze Patch_alt durch Patch_neu
        ENDIF
UNTIL Endekriterium
```

Alg. VI,7 Direkte Reduktion

[42] Im Programm auch 'Hidden Vertices'

Das Endekriterium wird durch zwei Faktoren ausgelöst: entweder wenn der Benutzer eine Abbruchtaste betätigt oder wenn hintereinander eine bestimmte Anzahl der ausgewählten Punkte nicht gelöscht werden konnte.
Der Algorithmus VI.4 beschreibt nur den prinzipiellen Ablauf. Das tatsächliche Programm ist im Anhang beschrieben. Es wurde um einige Punkte erweitert:

1. Berechnung von statistischen Informationen.
2. Ausgabe von Statusinformationen auf Bildschirm und in Log-Datei.
3. Redundante Plausibilitätstests
4. Automatisches Speichern des Triset in regelmäßigen Abständen[43].

4.Erhaltung der Konsistenz im Trimesh

In 19 ist ein Triset zu sehen, wie man es sich als Ergebnis einer Reduktion wünscht. Die Faces sind gleichmäßig groß und nicht zu schmal. Um ein Triset von so hoher Qualität zu erzeugen, müssen zusätzlich zu den bis jetzt genannten Punkten noch weitere beachtet werden.

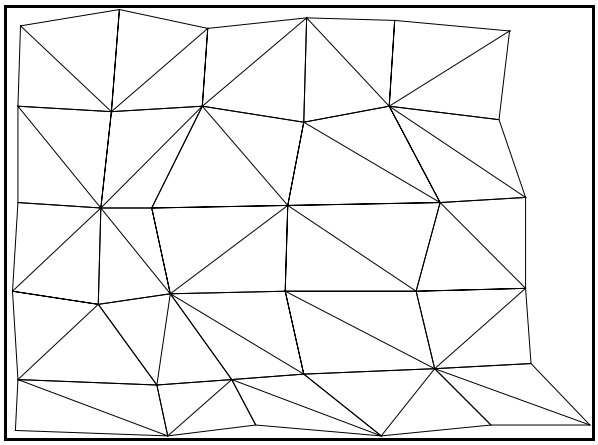


Abb. VI.19 Trimesh

Die Reduktion eines Triset ist ein dynamischer Vorgang. Die Oberfläche des Objektes verändert sich laufend, indem kleine Ausschnitte durch neue Patches ersetzt werden. Da das Ausgangs-Triset beliebig komplex sein kann und der Verlauf der Reduktion durch eine Zufallsfunktion gesteuert wird, treten während der Reduktion alle denkbaren und z.T. auch überraschenden Konstellationen auf.
Ich möchte jetzt einige dieser unerwünschten Sonderfälle zeigen, die während der ersten Testläufe aufgetreten sind.
Das linke Beispiel in 20 zeigt die typische 'Haifischflosse': ein Face (_ABE) ragt steil aus dem Triset auf und ist nur über ein Edge (AB) mit diesem verbunden[44].

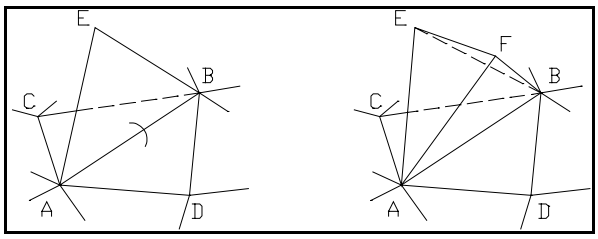


Abb. VI.20 Sonderfälle

Im rechten Beispiel sieht man eine einfache

[43] Da das Programm für lange Laufzeiten ausgelegt wurde, soll nach einem Stromausfall oder Systemabsturz zumindest ein Teilergebnis verfügbar sein.

[44] Bei genauer Betrachtung stellt man fest, daß diese 'Flosse' aus zwei aufeinanderliegenden Faces _ABE und _AEB besteht.

Variante einer 'Blase'. hierbei handelt es sich um ein kleines eigenständiges, geschlossenes Triset, daß mit dem eigentlichen Triset nur über eine Edge (AB) verbunden ist.
Beide Sonderfälle verstoßen nicht gegen die bisher aufgestellten Restriktionen und Kriterien. Trotzdem sind sie unerwünscht: Man kann davon ausgehen, daß die Eingabedaten wie sie der Digitizer liefert eine relativ einfache Oberfläche beschreiben. 'Blasen' oder 'Flossen' könnten schon aus technischen Gründen nicht erkannt werden. Das Ausgangstriset hat prinzipiell die Form wie in 19 mit einfacher 'Topologie'. Ich werde nun zeigen, wie die Sonderfälle die Topologie des Trimesh ändern.

Eine wichtige Eigenschaft eines Trimesh mit einfacher Topologie ist, daß jedes seiner Faces an genau drei Nachbarfaces grenzt (eine Ausnahme bilden Faces die am Rand liegen). Daraus folgt, daß jede Edge (Strecke zwischen zwei Vertizes) von genau zwei Faces berührt wird.

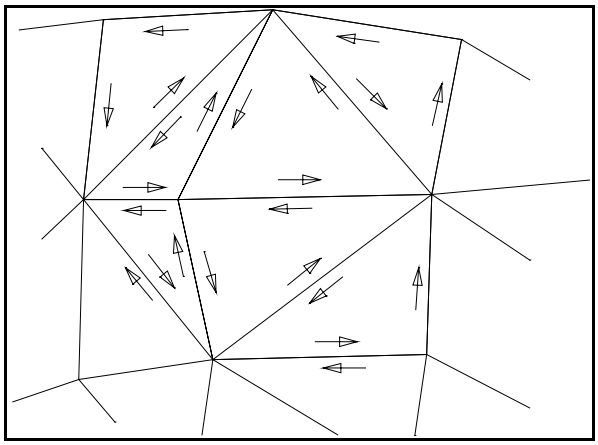


Abb. VI.21 Topologie im Triset

In Kapitel IV habe ich bereits die gleichmäßige Orientierung der Faces angesprochen. Die Faces des Ausgangs-Trimesh wurden so definiert, daß bei benachbarten Faces die gleiche Seite 'vorne' ist. Noch einmal zur Erinnerung: wenn man die Vorderseite eines Face betrachtet, dann sind seine Eckpunkte gegen den Uhrzeigersinn definiert. Die gleichmäßige Orientierung der Faces wird auch während der Reduktion nicht verändert.
Wie man in 21 sieht, folgt daraus:

> *In einem Triset mit einfacher Topologie kommt eine Edge niemals zweimal mit der gleichen Ausrichtung vor.*

Beide Sonderfälle aus 20 verstoßen gegen diese Regel, weil die Edge AB die Seite von mehr als zwei Faces definiert, d.h. Edge AB tritt auch mehrfach in der gleichen Ausrichtung auf.
Die oben beschriebenen Sonderfälle werden vermieden, indem vor jedem Reduktionsschritt auf das Entstehen von doppelten identischen Edges geprüft wird.

D.Simulated Annealing (globaler Ansatz)

Der oben beschriebene Weg der 'Direkten Reduktion' liefert zwar brauchbare Ergebnisse, hat aber prinzipielle Nachteile:

1. Es wird nur ein lokales Optimum gefunden.
2. Randbedingungen können nur als Restriktionen definiert werden. Es wird nur zwischen 'erlaubt' und 'nicht erlaubt' unterschieden.

Wünschenswert ist ein Ansatz, der zum einen ein globales Optimum sucht. Zum anderen sollen die Randbedingungen flexibler gehandhabt werden:

1. Unter den erlaubten Varianten sollen die 'besseren' bevorzugt werden.
2. Die Randbedingungen sollen gewichtet werden: ein etwas schmaleres Dreieck wird z.B. akzeptiert, wenn dadurch die neue Oberfläche näher an dem Original liegt.
3. Oberstes Ziel der Optimierung ist nicht mehr eine minimale Anzahl von Vertizes N_V. Es wird eine etwas größere Anzahl von Punkten in Kauf genommen, wenn dadurch die anderen Randbedingungen besser erfüllt werden können.

Der erste Schritt in diese Richtung ist:

Anstatt auf N_V=minimal wird jetzt auf Qualität=maximal optimiert.

Jede Randbedingung geht gewichtet in die Berechnung der Qualität mit ein. Gleichzeitig können weiterhin Restriktionen definiert werden, die eingehalten werden müssen.
Im Folgenden werde ich einen Ansatz untersuchen, der diese Eigenschaften auf Kosten einer längeren Programmlaufzeit aufweist.

1.Was ist Simulated Annealing ?

Simulated Annealing ('simulierte Abkühlung') ist eine Methode zur Lösung von Optimierungsaufgaben. Es basiert auf Beobachtungen aus der Thermodynamik, speziell der Kristallbildung.

In z.B. einer Metallschmelze hoher Temperatur können sich die einzelnen Moleküle frei bewegen. Bei langsamer Abkühlung verlieren sie aber ihre thermische Mobilität und reihen sich häufiger geordnet aneinander. Diese Raumgitterstrukturen können mehrere Millionen Atome in jede Richtung umfassen. Man spricht in so einem Fall von einer kristallinen Struktur. In einem idealen Kristall haben sich die Atome so angeordnet, daß die Summe der gegenseitigen Anziehungskräfte minimal ist.

Die Energie eines idealen Kristalls ist minimal.

Entscheidend dafür, ob sich ein Zustand minimaler Energie einstellt, ist die Geschwindigkeit, mit der die Abkühlung verläuft. Kühlt man eine Metallschmelze schlagartig ab, haben die Atome nicht genügend Zeit sich anzuordnen. Es entstehen polykristalline oder amorphe Strukturen höherer Energie.
Die Beweglichkeit der Moleküle ist eine Funktion von der Temperatur '. Für ' » 0 bewegt sich jedes Molekül frei gegenüber seinen Nachbarn. Die thermische Mobilität ist so groß, daß die Anziehungskräfte der Moleküle untereinander vernachlässigt werden können. Die Wahrscheinlichkeit, daß sich das System von einer idealen Kristallstruktur entfernt, und die Energie E damit zunimmt, ist etwa genau so groß, wie die Wahrscheinlichkeit für eine Abnahme der Energie. Bei kleineren Temperaturen wird die Bewegung der Moleküle zunehmend von den Anziehungskräften untereinander beeinflußt. E wird im Mittel kleiner werden.
Die Wahrscheinlichkeit, daß die Systemenergie zu einem bestimmten Zeitpunkt eine bestimmte Energie E hat, läßt sich durch die Boltzmann Verteilung ausdrücken[45]:

$$p(E) \sim e^{-E/k'}$$;k: Boltzmannkonstante; E>0

Selbst bei kleinen Temperaturen besteht eine (wenn auch geringe) Wahrscheinlichkeit, daß die Energie größer als minimal ist.
Würde man versuchen, die Systemenergie auf direktem Weg zu reduzieren (analog zu **reduceDirect**), dann würde man eine Zunahme von E nicht akzeptieren, sondern nur stetig 'bergab' auf ein Minimum zugehen. Der Preis wäre, wie wir gesehen haben, daß ein einmal erreichtes lokales Minimum nicht mehr verlassen werden kann.
Beim Annealing dagegen kann E mit einer bestimmten Wahrscheinlichkeit größer werden. Bildlich gesehen kann dadurch das System theoretisch aus einem lokalen Minimum wieder 'herausklettern', um dann in ein, hoffentlich globaleres, wieder abzusteigen[46].
Die Methode der langsamen Abkühlung ('annealing') macht man sich seit langem z.B. in der Glas- oder Stahlerzeugung zunutze.

*Durch Annealing wird (theoretisch) das **globale** Energieminimum gefunden.*

[45] Unter der Voraussetzung, daß sich das System im thermischen Gleichgewicht befindet.

[46] Ähnliche Verfahren werden angewandt, um beim Lernvorgang von Neuronalen Netzen eine lokale Sackgasse zu vermeiden → Boltzmann-Maschinen.

Bemerkenswert ist, wie die Natur es 'schafft' die Schmelze mit hohem Energiegegehalt in ein Kristall nahezu **minimaler** Energie umzuwandeln.

1953 haben Metropolis und Mitarbeiter zum erstenmal eine Analogie zwischen Annealing und kombinatorischen Optimierungsaufgaben hergestellt[47].
Annahme ist hierbei, daß die Wahrscheinlichkeit für eine Zunahme der Systemenergie um ΔE gleich

$$p(\Delta E) = e^{-\Delta E/k'}$$

ist[48].

Dieser Ausdruck wird für $\Delta E<0$ größer als eins. In diesem Fall wird $p(\Delta E<0) = 1$ definiert. Eine Abnahme der Energie wird also auf jeden Fall akzeptiert. Die Wahrscheinlichkeit für eine Energiezunahme ist dagegen sowohl von ΔE als auch von ' abhängig (22).
Eine Optimierung nach dem Verfahren des Simulated Annealing beginnt mit einer hohen Temperatur ('»0), die im weiteren Verlauf gegen Null geht.
In der Konsequenz wird zu Beginn relativ häufig eine Energiezunahme akzeptiert. Je weiter die Temperatur absinkt, desto seltener ist dies der Fall, bis zum Schluß ('=0) nur noch eine Abnahme der Energie zugelassen wird.

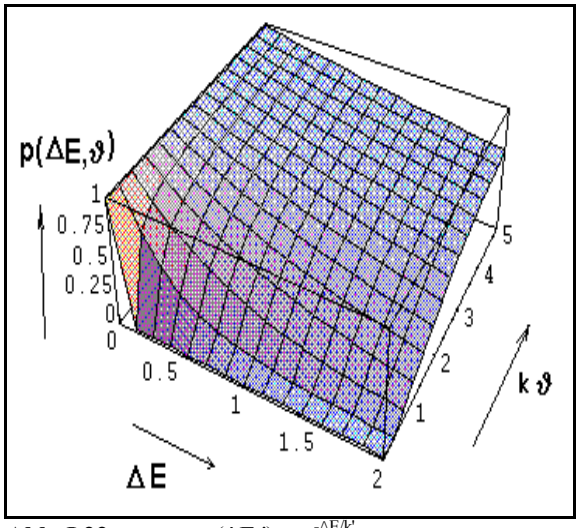


Abb. I.22 $p(\Delta E,') = e^{-\Delta E/k'}$

Ich zeige jetzt den prinzipiellen Ablauf des Simulated Annealing.

```
' ← 'Start
Ealt ← Energy (System)
DO
        Generiere neue Systemkonstellation K
        Eneu ← Energy (K)
        ΔE ← Eneu - Ealt
        IF Metropolis (ΔE, ') THEN
                Akzeptiere K
                Ealt ← Eneu
        ENDIF
        NeueTemperatur (')
WHILE NOT Endekriterium
```

Alg. I,8 Metropolis Algorithmus

Es sind Funktionen zu definieren, die

[47] Eine ausführliche Beschreibung findet man in 'The Annealing Algorithm' ([Otten89]). Die praktische Anwendung am Beispiel des 'Travelling Salesman' wird in [Recipes87, S.326 ff] gezeigt.

[48] Auch hier wieder unter der Vorraussetzung, daß sich das System im thermischen Gleichgewicht befindet.

1. eine neue Systemkonfiguration generieren,
2. die Energie des Systems berechnen,
3. das Metropoliskriterium gewährleisten,
4. die Temperaturänderung steuern (Annealing Schedule) und
5. ein Abbruchkriterium definieren.

Das Kriterium, welches unter Berücksichtigung von ' und ΔE eine Änderung des Systemzustandes akzeptiert oder verwirft heißt '**Metropoliskriterium**'. Hier wird nach der oben beschrieben Wahrscheinlichkeitsverteilung p(ΔE,') berechnet und mit einem Zufallswert verglichen.
Als Abbruchkriterium definiere ich das Erreichen einer Endtemperatur $'_{end}$.
Ich gehe jetzt auf die Punkte 1., 2. und 4. ein.

2.Generierung von Systemkonstellationen

Es gibt mehrere Möglichkeiten ein Triset zu manipulieren. Denkbar ist es z.B., neue Punkte hinzuzufügen oder die Position bestehender Vertizes zu verändern.
Ich beschränke mich auf zwei Aktionen:

1. Löschen von Vertizes.
2. Zurückholen bereits gelöschter Vertizes (entlöschen).

Wie Punkte in einem Triset gelöscht werden, habe ich bereits ausführlich beschrieben.
Beim Entlöschen wird das Face, welches dem gelöschten Vertex am nächsten liegt, durch eine neue Tesselierung ersetzt.Dabei wird das Triset um einen Vertex und zwei Faces erweitert[49].
Denkbar sind auch andere Entlöschmethoden, z.B. indem zusätzlich benachbarte Faces gelöscht und das entstandene Loch neu tesseliert wird.

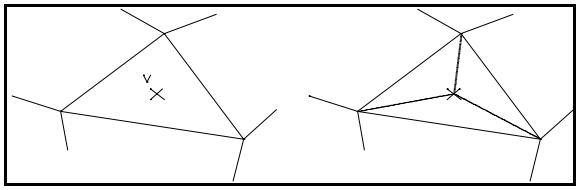


Abb. I.23 Vertex entlöschen

Wichtig ist dabei nur, daß alle Systemzustände, die durch das Löschen von Punkten erreicht werden, wieder rückgängig gemacht werden können.

Alle Systemänderungen müssen reversibel sein.

Anderenfalls könnten 'Sackgassen' entstehen, die das Erreichen eines Optimums verhindern.

[49] Als es um das Löschen von Punkten ging, habe ich den umgekehrten Fall hergeleitet: das Triset wird um einen Vertex und zwei Faces verkleinert.

3.Die Energiefunktion

Simulated Annealing sorgt für eine Minimierung der Triset-Energie.
Damit am Ende der Optimierung das gewünschte Triset entsteht, muß die Energie umgekehrt proportional zu den positiven Eigenschaften sein.
Ich berechne die Energie aus vier Anteilen:

1. Anzahl der Vertizes im Triset: N_V.
2. Oberfläche des Triset: A_Σ.
3. Summe der Fehlerquadrate zwischen der reduzierten Oberfläche und den gelöschten Punkten: ε_Σ.
4. Summe der Face-Proportionen: P_Σ[50].

Diese Anteile werden mit Faktoren gewichtet, die der Benutzer definiert.
Als Funktion für die Energie berechne ich:

$$E = F_V N_V + F_A A_\Sigma + F_\varepsilon \varepsilon_\Sigma + F_P P_\Sigma$$

mit

$$A_\Sigma = \sum_{i=1}^{N_F} A_i$$

$$\varepsilon_\Sigma = \sum_{i=1}^{N_F} \varepsilon_i^2$$

$$P_\Sigma = \sum_{i=1}^{N_F} P_i$$

Wenn alle Faktoren größer als Null sind, dann führt das Annealing zu einem Triset mit wenig Vertizes, kleiner Fläche, gleichmäßigen Dreiecken und geringer Abweichung zum Original.
Die Energie des Triset wird einmal zu Beginn berechnet. Bei jeder folgenden Änderung (Löschen oder Entlöschen eines Vertex) wird ΔE ermittelt und addiert.
Um Rundungsfehler zu begrenzen, wird E in regelmäßigen Abständen erneut absolut berechnet.

4.Annealing Schedule

Beim 'Annealing Schedule' handelt es sich um den 'Fahrplan',nach dem die Abkühlung verläuft. Dieser wird bestimmt durch die Anfangstemperatur $'_{start}$, die Endtemperatur $'_{end}$, sowie Anzahl und Größe der Temperaturänderungen.

[50] Berechnung der Proportion in Kapitel 0, S. 49. P ist um so größer, je schmaler das Dreieck ist. Ziel ist daher eine Minimierung von P_Σ.

Die Anfangstemperatur ist so zu wählen, daß die Wahrscheinlichkeit einer Energiezunahme ungefähr so groß ist, wie die Wahrscheinlichkeit einer Energieabnahme, also ungefähr 1. Ich definiere:

$$p(\overline{\Delta E}, \vartheta_{start}) = 0.9$$

Die verschiedenen Systemzustände werden durch löschen bzw. entlöschen von einzelnen Vertizes generiert. Den zu erwartenden Mittelwert von ΔE schätze ich daher grob anhand der Energiefunktion und der Trisetgröße ab:

$$\overline{\Delta E} \approx \frac{E_{total}}{N_V}$$

Die Anfangstemperatur[51] ist dann:

$$\vartheta_{start} = - \frac{\overline{\Delta E}}{\ln(0,9)}$$

Die Endtemperatur definiere ich zu:

$$\vartheta_{end} = 0,1$$

Eine Temperaturabsenkung wird durch Multiplikation der aktuellen Temperatur mit einem Faktor F realisiert:

$$\vartheta_{neu} = F \bullet \vartheta \;\; ; mit \; 0 < F \leq 1$$

Voraussetzung für die Anwendbarkeit der Boltzmann-Verteilung ist, daß sich das System im Gleichgewicht befindet[52]. In diesem Fall trifft das Metropoliskriterium in ca. 50% aller Fälle zu.
Die Absenkung der Temperatur sollte so erfolgen, daß das thermische Gleichgewicht erhalten bleibt. Eine Temperaturänderung ist also erst dann erlaubt, wenn der Anteil von angenommenen und zurückgewiesenen Systemänderugen in etwa gleich ist. Um diese Randbedingung einzuhalten ist es nötig, eine Laufzeitstatistik zu führen[53].

---

51 Da die Boltzmannkonstante k konstant ist, ziehe ich ihn in den Wert von ' mit hinein.

52 Übertragen auf das Beispiel der Metallschmelze bedeutet dies, daß sich nicht nur die Umgebungstemperatur ändert, sondern wirklich jedes Molekül die neue Temperatur angenommen hat.

53 Berechnung eines dynamischen Mittelwertes im Anhang.

VII.Speichern des Triset

A.Speichermöglichkeiten

Beim Schreiben der Trisetdaten habe ich verschiedene Möglichkeiten der Speicherung vorgesehen. Der Benutzer kann zwischen folgenden Arten wählen:

1. Es werden die Vertexkoordinaten und die zugehörigen Texturparameter gespeichert (X, Y, Z, U, V).
2. Es werden nur die Vertexkoordinaten abgelegt (X, Y, Z).
3. Es werden Vertexkoordinaten und -farben gespeichert (X, Y, Z, R, G, B).

Der erste (übliche) Fall ist natürlich nur sinnvoll, wenn gleichzeitig eine Texturdatei erzeugt wird.
Die zweite Methode kommt in Frage, wenn schon die Eingangsdaten keine Farbinformation enthalten haben oder die Farbe nicht relevant ist.
Methode drei wird z.B. gewählt wenn eine CEF-Datei lediglich konvertiert werden soll, ohne eine Reduktion oder Texture-Mapping durchzuführen.

B.Ausgabeformate

Comp3D unterstützt zwei Ausgabeformate: ZOF und DXF. Beides sind zeilenorientierte ASCII-Formate. Während das übersichtlichere ZOF-Format z.Zt. nur von ARENA gelesen werden kann, ist DXF (**D**ata e**X**-change **F**ormat) zu einem Quasi-Standard geworden, den viele Programme verarbeiten.

1.Einhalten maximaler Blockgrößen

Neben dem Speicherformat muß ein weiterer Punkt beim Datenaustausch berücksichtigt werden: Nicht alle Zielprogramme sind in der Lage, beliebig große Trisets zu verarbeiten.
Je nach Programm kann eine maximale Anzahl von Elementen (Faces, Vertizes) oder eine Speichergrenze (oft 64 kB pro Triset) vorgegeben sein.
Um möglichst flexibel auf alle Anforderungen reagieren zu können, gibt es in Comp3D die Option, Trisets zu blocken. Der Benutzer kann Gewichtsfaktoren für Faces und Vertizes, sowie einen Schwellwert definieren. Beim Schreiben des Triset wird darauf geachtet, daß sein Gesamt-'Gewicht' unterhalb des Schwellwerts bleibt. Ggf. werden mehrere Trisets angelegt.
Wenn keine Beschränkungen erforderlich sind, ist es sinnvoll alle Daten in ein Triset zu schreiben, weil dann die Redundanz am kleinsten ist (andernfalls müssen die Punkte an denen zwei Trisets aneinandergrenzen doppelt definiert werden).

2.'Small'-Option

Sowohl für das DXF- als auch für das ZOF-Format gibt es die Möglichkeit, Nullen am Ende einer Zahl zu entfernen und auf eine gleichmäßige, spaltenweise Anordnung zu verzichten.
Auf diese Weise kann die Datei auf Kosten der Lesbarkeit verkleinert werden.

C.Algorithmus

In beiden unterstützten Formaten wird ein Triset durch jeweils ein Vertex- und ein Facearray beschrieben. Da das dem Format entspricht, wie ich es intern verwende, ist die Implementierung einfach zu realisieren (vgl. Listing im Anhang).

VIII.Auswertung

Testläufe mit Comp3D haben gezeigt, daß die entwickelten Algorithmen gute Ergebnisse liefern.
Ich habe das Programm mit verschiedenen Typen von Eingabedaten getestet:

1. Künstlich erzeugte Trisets mit einfacher Geometrie, die geeignet sind spezielle Teilaspekte der Reduktion zu untersuchen.
2. Datensätze die vom Digitizer Spex3D erzeugt wurden.
3. Fremderzeugte Datensätze.

Die Simulationsdaten (farbige Kugel, Würfel, Zylinder, ...) waren aus mehreren Gründen nützlich:

- Aufgrund der bekannten Abmessungen, konnte ich nachweisen, daß die Randbedingungen (maximaler Fehler, maximale Schmalheit, etc.) eingehalten werden.
- Durch die kleinen Datensätze (100 bis 400 Vertizes) konnten relativ schnell Ergebnisse produziert werden.
- Viele der fremderzeugten Datensätze hatten 'doppelte Edges' (vgl. Kapitel 0, S.52), und waren daher nur bedingt für Simulated Annealing geeignet. Außerdem waren für diese Daten keine einzelnen Vertexfarben definiert, so daß nur die Geometrie reduziert werden konnte. Two-Part Mapping war nicht möglich.
- Da Spex3D noch in der Entwicklungsphase ist, verläuft die Digitalisierung sehr aufwendig und langsam. Aus diesem Grund standen mir nur wenige vollwertige Datensätze zur Verfügung.

A.Laufzeitverhalten

1.Direkte Reduktion

Comp3D legt auf Wunsch eine Laufzeitstatistik an. Diese Datei läßt sich mit Hilfe von geeigneten Programmen grafisch Aufbereiten.
Dabei erhält man regelmäßig typische Kurvenverläufe.
Bei der direkten Reduktion sind zwei Laufzeitvariablen von Bedeutung:
Zum einen die Anzahl der gelöschten Punkte (in Prozent). Wie zu erwarten war, strebt dieser Wert stetig gegen ein Maximum, wobei er immer langsamer konvergiert. Das liegt daran, daß immer weniger der ausgewählten Vertizes den Restriktionen für eine Löschung genügen.

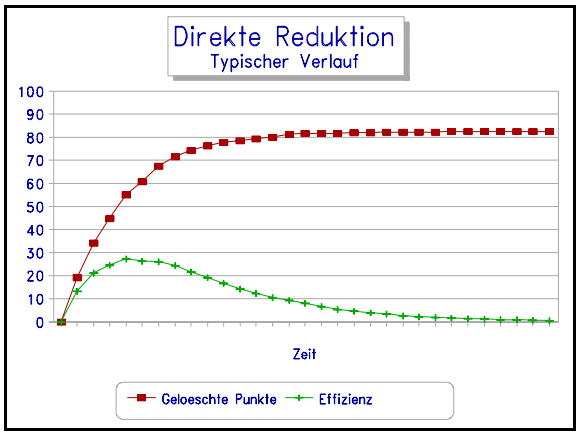


Abb. VIII.24 Typischer Verlauf einer direkten Reduktion

Der Endwert dieser Kurve ist abhängig von den eingestellten Parametern (besonders vom maximal erlaubten Fehler ε_{max}) und der Oberflächenstruktur des Objektes.
Die zweite Kurve zeigt die Effizienz des Algorithmus, d.h. das Verhältnis der betrachteten Kandidaten zu den tatsächlich gelöschten Vertizes (Angabe ebenfalls in Prozent).
Obwohl die Effizienz am Anfang groß ist, beginnt der Kurvenverlauf bei Null, weil der Mittelvert dynamisch berechnet wird.

2.Simulated Annealing

Die Laufzeitstatistik wurde für das Simulated Annealing um Kurven für Temperatur- und Energieverlauf erweitert (jeweils auf 100 normiert).
Beim Betrachten des Graphen fällt sofort der prinzipielle Unterschied zur direkten Reduktion auf: Die Anzahl der gelöschten Punkte steigt nicht stetig, sondern fällt zeitweise auch wieder. Dies ist besonders am Anfang des Reduktionslaufes der Fall, wenn die Temperatur noch hoch ist und demzufolge viele Vertizes **ent**löscht werden.
Die Energie verhält sich in etwa umgekehrt proportional zur Anzahl der gelöschten Punkte.

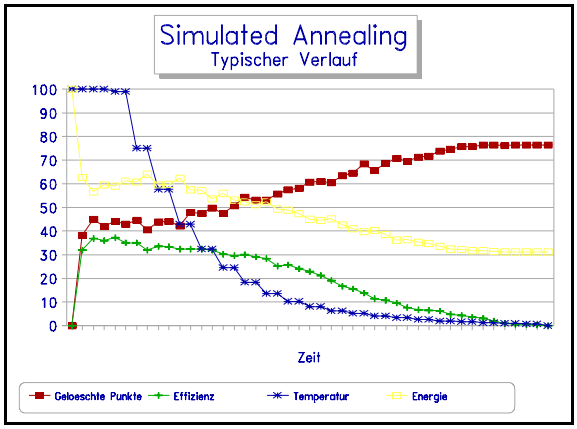


Abb. VIII.25 Annealing, Laufzeitverhalten

Gegen Ende des Laufes ist die Temperatur fast zu Null geworden. Energie und die Zahl der gelöschten Punkte konvergieren nahezu stetig gegen ihren Endwert.
Man erkennt am Anfang einen Zeitraum, in dem die Temperatur maximal bleibt. Der Grund dafür liegt darin, daß zunächst ein thermisches Gleichgewicht erreicht werden muß, bevor die Abkühlung beginnt.

Ich konnte durch mehrere Testläufe zeigen, daß die Ergebnisse der Reduktionen um so besser sind, je langsamer die Abkühlung erfolgt (26). Wichtig ist dabei, daß der Temperaturverlauf dem verfügbaren Zeitraum angepaßt wird, so daß $'_{end}$ genau mit Ablauf des Zeitlimits erreicht wird. Der Annealingalgorithmus kann nur die Energie des Systems minimieren. Es bleibt dem Benutzer überlassen, die Parameter zur Energieberechnung so zu wählen, daß 'minimale Energie' und 'gewünschte Trisetstruktur' zusammenfallen.

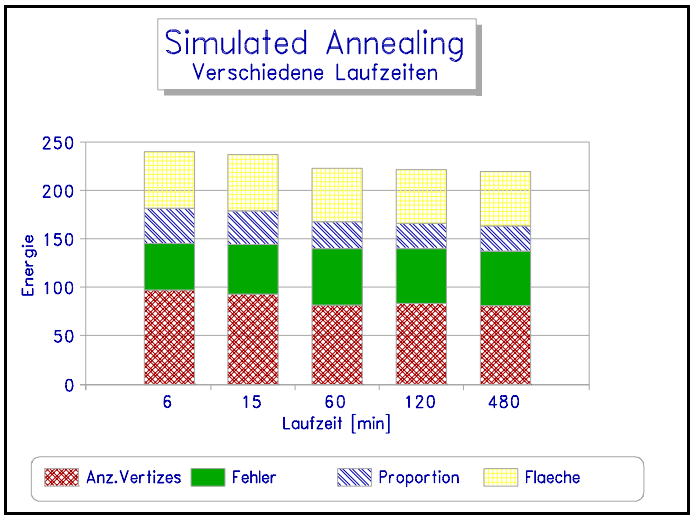


Abb. VIII.26 Typisches Laufzeitverhalten beim Simulated Annealing

Um die direkte Reduktion mit dem Annealing-Verfahren vergleichen zu können, müssen die Annealingparameter so gewählt werden, daß die Energie ausschließlich von der Anzahl der Vertizes abhängt (F_V=1, F_A=F_ε=F_P=0). Damit wird das Löschen von Punkten zum obersten Optimierungsziel. Unter diesen Bedingungen erreicht Simulated Annealing höhere Reduktionsraten als die direkte Reduktion (allerdings immer unter höherem Zeitaufwand).

3.Zusammenfassung

Bei geeigneter Parameterwahl konnten Datensätze ohne weiteres auf 20% bis 2% der Originalgröße reduziert werden[54]. Im gerenderten Zustand (mit Textur) waren dabei kaum Unterschiede zum Original feststellbar (vgl. folgende Beispiele).
Damit ist das Ziel der Diplomarbeit erreicht worden.
Verbesserungsmöglichkeiten sehe ich hauptsächlich im Bereich der Arbeitsgeschwindigkeit. Besonders bei großen Datensätzen sind mehrere Stunden Laufzeit nötig, um vernünftige Ergebnisse zu produzieren.
Ein Ansatz dazu könnten redundante Datenstrukturen für das Triset sein, um Suchvorgänge zu beschleunigen[55].

[54] Am Beispiel eines Würfels mit 400 Vertizes, der auf 8 Vertizes reduziert wurde, konnte das Erreichen eines globalen Optimums nachgewiesen werden.

[55] Z.B. 'Winged Edge'-Strukturen (vgl. [GemsII91]).

B.Beispiele

Hier folgen einige praktische Beispiele von durchgeführten Probeläufen.
Die Bilder dazu sind auf den folgenden Seiten zu finden.

1.Box

Bei 'Box' handelt es sich um einen künstlich erzeugten Datensatz. Das Triset beschreibt einen Würfel ohne Deckel und Boden. Es besteht aus 400 Vertizes und 738 Faces.
Parameterwahl: ε_{max}=1mm, F_V=1, F_ε=100, F_A=5, F_P=0.5
Simulated Annealing mit Laufzeit 10 min.

2.Sphere

Sphere ist eine an den Polen geöffnet Kugel mit dem Durchmesser 1 cm. Sie besteht aus 380 Vertizes und 720 Faces. Der Datensatz konnte auf 67 Vertizes (18%) reduziert werden.
Parameterwahl: ε_{max}=1mm, F_V=1, F_ε=100, F_A=5, F_P=0.5
Simulated Annealing mit Laufzeit 10 min.

3.Ronny

Ronny ist ein fremderzeugter Datensatz ohne Farbinformatinen, mit 751 Vertizes und 1494 Faces. Unter der Randbedingung ε_{max}=0,1cm wurde er auf ca. ein Viertel reduziert.
Simulated Annealing mit Laufzeit 2 Std.

4.Genscher

Bei 'Genscher' handelt es sich um eine Gummimaske (Höhe etwa 30 cm), die mit Spex3D teilweise digitalisiert wurde (90 Polygonzüge). Comp3D erzeugt daraus ein Triset mit ca.37500 Vertizes.
Für die erste Reduktionsstufe (Reduktion der Polygonzüge) habe ich einen maximalen Fehler von 2 mm zugelassen. Beim anschließenden Annealing wurde ε_{max} auf 4 mm gesetzt.
Das Ergebnis ist eine Reduktionsrate von 2,1% (800 Vertizes).

5.Box, negative Proportionen

In diesem Beispiel habe ich noch einmal den Würfel reduziert. Allerdings wurden hier bewußt unsinnige Parameter gewählt:
Maximaler Fehler und Fläche des Triset gehen nicht in die Energieberrechnung ein. Als Vertex Gewichtsfaktor wurde 1 gewählt. Die Proportion wurde mit -1 gewichtet.
Der negative Wert von F_P führt zu einem Triset mit vorzugsweise schmalen Faces!

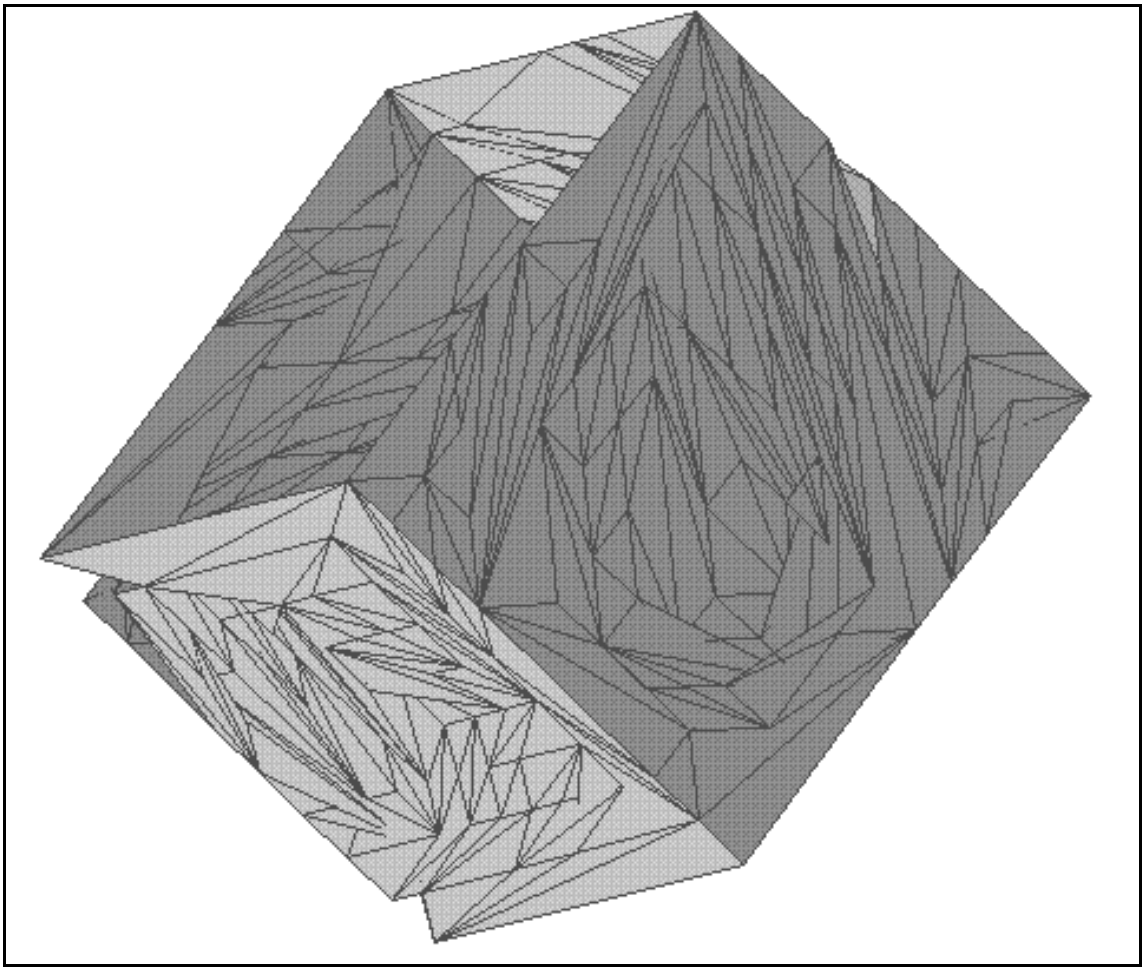


Abb. VIII.27 Box mit schmalen Faces

6.Bilder

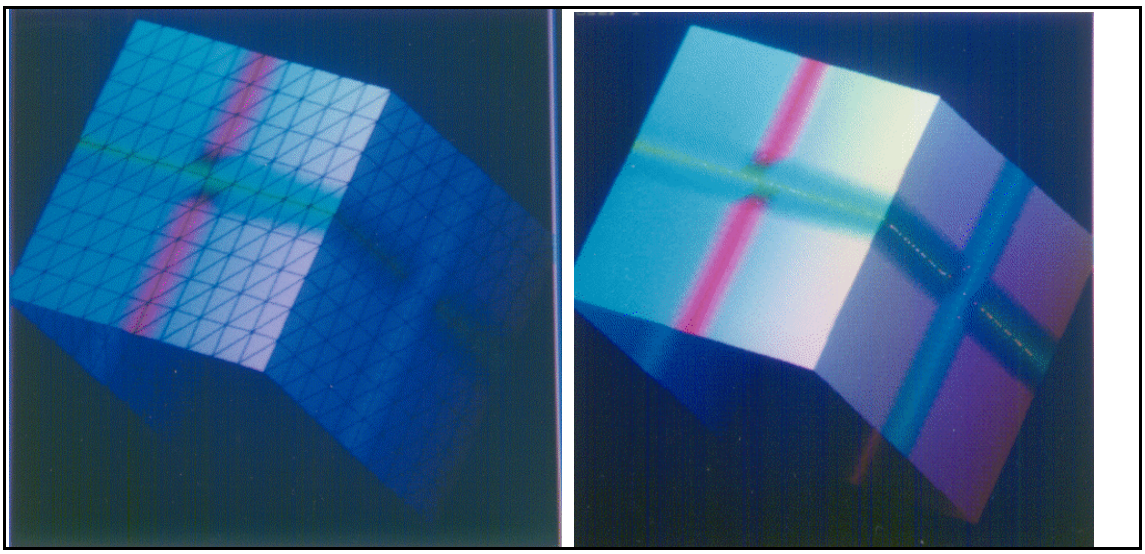


Abb. VIII.28 Original Box, mit und ohne sichtbaren Edges

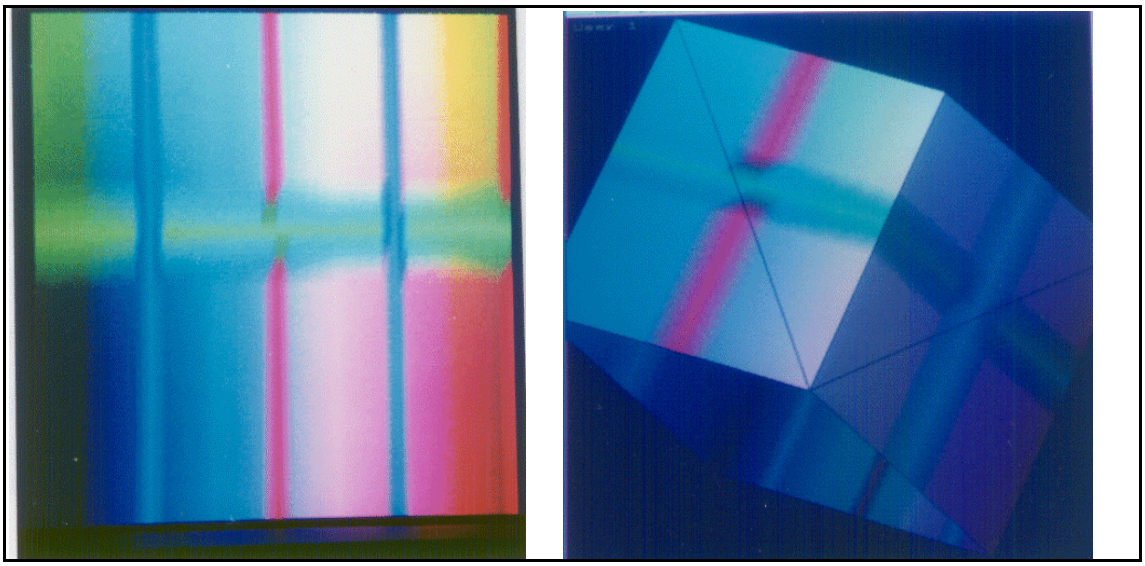


Abb. VIII.29 Textur, reduzierte Box

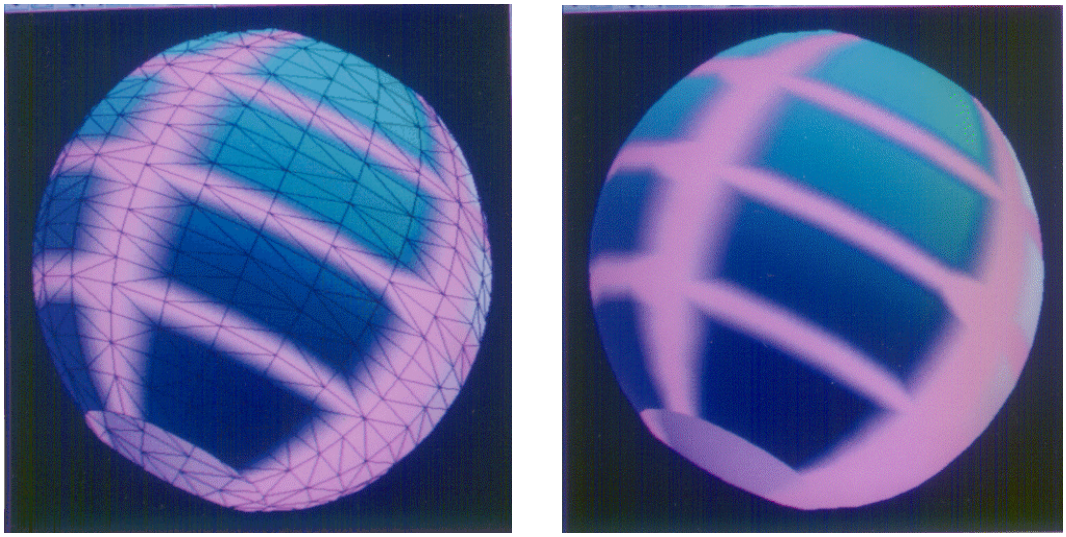


Abb. VIII.30 Original Sphere, mit und ohne Edges

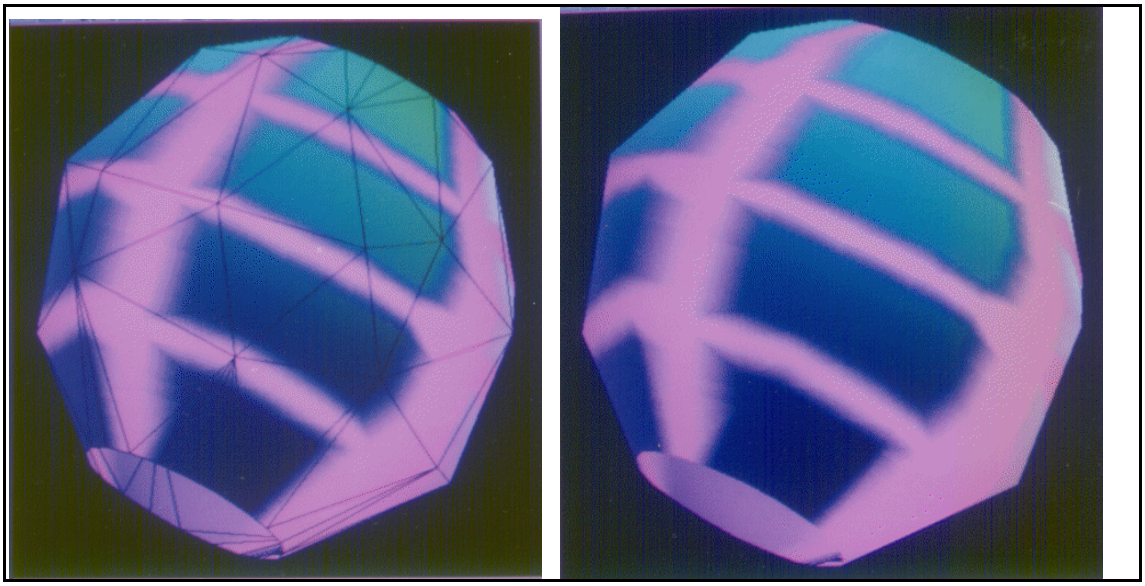


Abb. VIII.31 Reduzierte Sphere, mit und ohne Edges

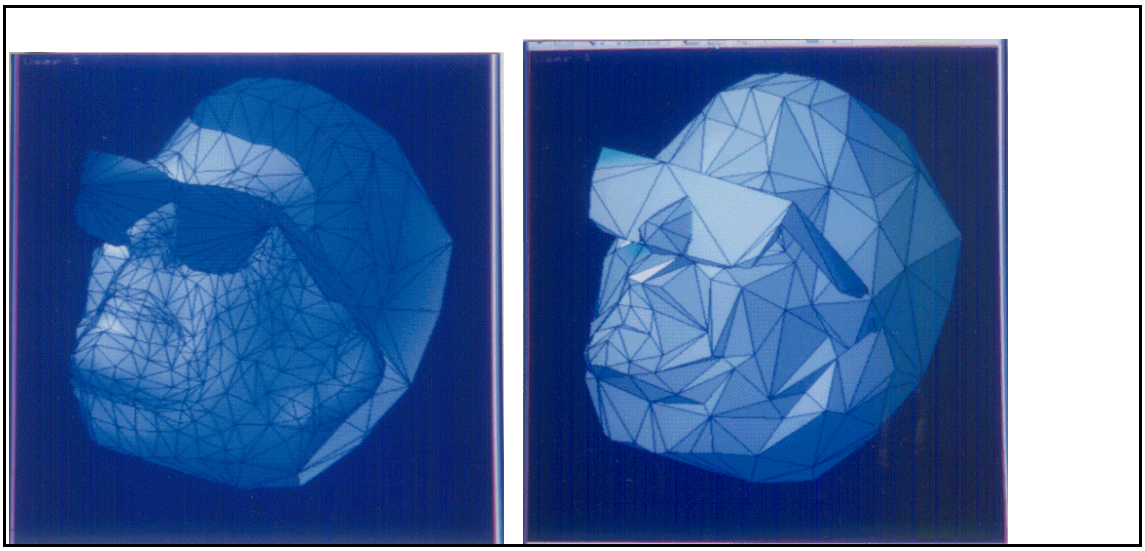
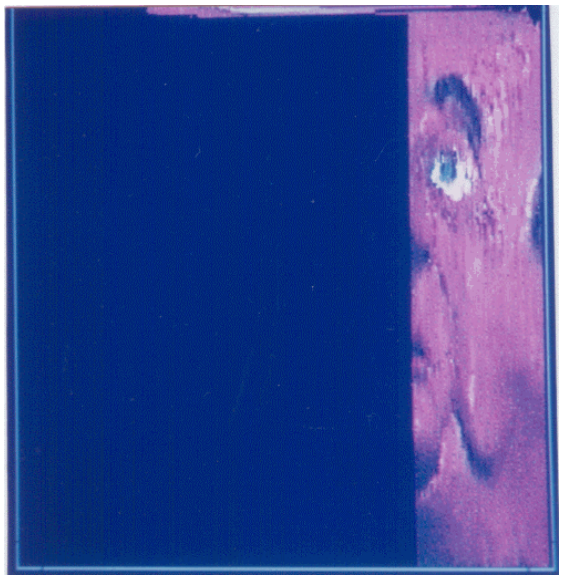


Abb. VIII.32 Ronny, original und reduziert



Genscher Textur

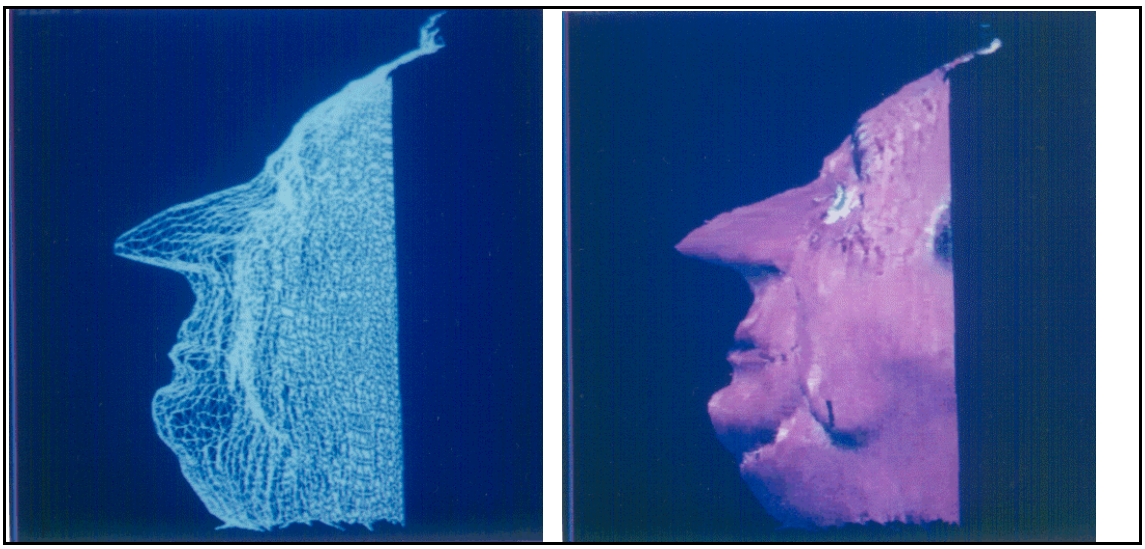


Abb. VIII.33 Original Genscher, Drahtmodell und Gouraud

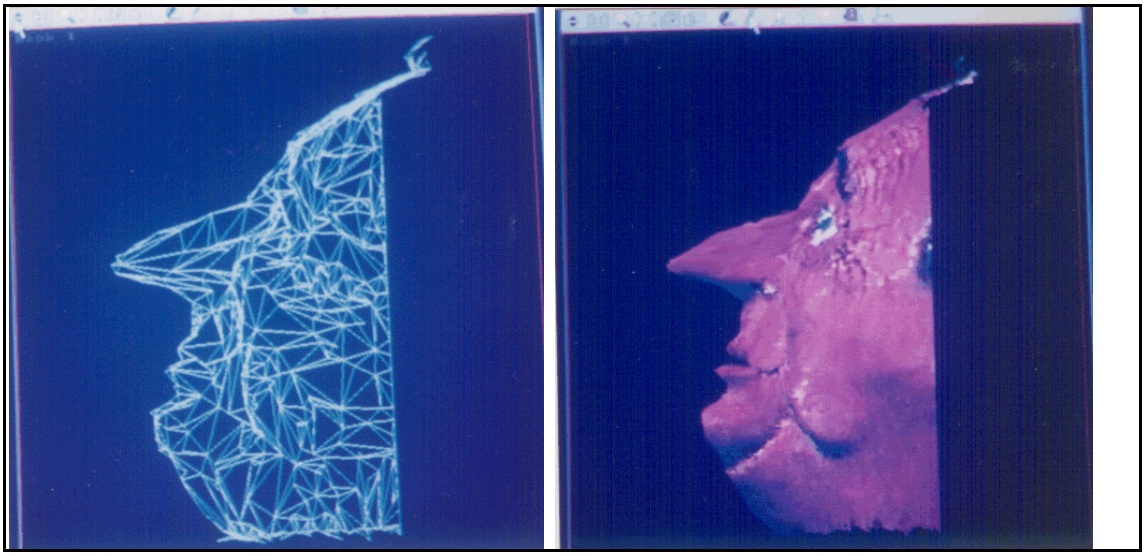


Abb. VIII.34 Reduzierter Genscher, Drahtmodell und Gouraud

7
Verwendete Formelzeichen und Abkürzungen

ε	(epsilon) Fehler,
E	Energie (des Trisets)
ΔE	Energiedifferenz zwischen zwei Systemkonstellationen
ΔV	Volumen zwischen zwei Patches
'	(theta) Temperatur beim Simulated Annealing
N_F	Anzahl der Faces im Triset
N_V	Anzahl der Vertizes im Triset

Prefixe bei der Angabe von Parametern:

$\leftarrow$ NAME:	reiner Rückgabewert, muß nicht initialisiert werden
$\rightarrow$ NAME:	Paramter wird nicht verändert (call by value)
$\leftrightarrow$ NAME:	call by reference

Glossar

ARENA:	**A**dvanced **Ren**dering and Modelling **A**pplication, Programm zur wirklichkeitsnahen Darstellung von 3D-Daten
Backface Culling:	$\rightarrow$Culling
CEF:	Ausgabedatenformat des Digitizers Spex3D.
Comp3D:	Im Rahmen der Diplomarbeit erstelltes Programm zum Lesen, Konvertieren, Reduzieren und Schreiben von dreidimensionalen Geometriedaten.
Culling:	Verfahren in der Computergrafik um versteckte Flächen zu unterdrücken.
DXF:	**D**ata **E**xchange **F**ormat, verbreitetes Dateiformat zum Speichern von CAD-Daten.
Edge:	('Kante') Geradenstück zwischen zwei $\rightarrow$Vertizes. Mehrere Edges begrenzen eine Polygon.
Face:	('Masche') Teil einer Fläche, wird Vertizes und Edges begrenzt.
Facet:	(Facette) $\rightarrow$Face
Frontface Culling:	$\rightarrow$Culling

Hidden	
-Line:	Linie, die in einer 3-dimensionalen Darstellung durch andere Flächen des Objekts verdeckt wird. Ihre Darstellung wird unterdrückt.
-Surface:	Fläche, die in einer 3-dimensionalen Darstellung durch andere Flächen des Objekts verdeckt wird. Ihre Darstellung wird unterdrückt.
-Vertex:	Durch Reduktion gelöschter Punkt des Trisets. Wird nur als gelöscht markiert, bleibt aber im Datensatz erhalten.
Patch:	('Flicken') kleines →Triset
Polyeder:	Durch →Polygone begrenzter Körper.
Polygon:	(Vieleck) Durch Geraden begrenzte Fläche.
Polygonzug:	
Polyline	→Polygonzug
Quadmesh:	Netz aus viereckigen Maschen
Rendern:	Wirklichkeitsnahe Darstellung von 3D-Grafikdaten
Spex3D:	3D-Digitalisierer, erzeugt die Ausgangsdaten, die ich in meiner Diplomarbeit verarbeite.
Tesselierung:	Erzeugen von Trimeshes, indem ein Polygon in Dreike zerlegt wird
Textur:	Farbinformation einer Objektoberfläche, meist als Bitmap abgelegt.
Texture Mapping:	Zuordnung zwischen Textur und Objektoberfläche
Triangulation:	→Tesselierung
Trimesh:	Netz aus dreieckigen Maschen (→Faces)
Triset:	→Trimesh
Two-Part Mapping:	→Texture Mapping unter Zuhilfenahme einer Zwischengeometrie.
Vertex:	Start- bzw. Endpunkt einer →Edge,
ZOF:	Datenformat von →ARENA

Quellenverzeichnis

[Bier86]Two-Part Texture Mapping (Artikel) E.A.Bier, K.R.Sloan Sept. 1986 in IEEE CG&A 33
[Gems90] Graphics Gems James Arvo (Editor) 1991 Academic Press Inc. 35
[GemsII91] Graphics Gems II James Arvo (Editor) 1991 Academic Press Inc. 64
[Hewitt91]A Practical Introduction to Phigs and Phigs Plus Howard, Hewitt, Hubbold, Wyrwas 1991 Addison-Wesley Publishing 9
[Hopgood91]A Primer for Phigs Hopgood, Duce 1991 John Wiley & Sons 9
[Micha91] Image Komprerssion (Diplomarbeit) Michael Krüger 1991 Fachhochschule Wedel 35
[Otten89] The Annealing Algorithm R.H.J.M. Otten, L.P.P.P. van Ginneken 1989 Kluwer Academic Publishers 56
[Patterson86]Projective Geometry and its Applications to Computer Graphics M.A.Penna, R.R.Patterson 1986 Prentice-Hall 19
[Recipes87]Numerical Recipes Press, Flannery, Saul, Vetterling 1987 Cambridge University Press 56
[Sedge89]Algorithms Second Edition Robert Sedgewick 1989 Addison-Wesley Publishing Company 29
[Watt89]Fundmentals of Three-Dimensional Computer Graphics Alan Watt 1989 Addison-Wesley Publishing Company 33

Literaturliste

Ich habe hier einige der Bücher zusammengestellt, die mir bei der Arbeit nützlich waren und einige der von mir behandelten Themen vertiefen.

Computergrafik allgemein:

Computer Graphics, Principles and Practice
Foley, van Dam, Feiner, Hughes
1990 Addison-Wesley Publishing Company

Fundamentals of Three-Dimensional Computer Graphics
Watt, Alan
1989 Addison-Wesley Publishing Company

Algorithmen:

Graphics Gems
Glassner, Andrew S. (Editor)
1990 Academic Press Inc.

Graphics Gems II
Arvo, James (Editor)
1991 Academic Press Inc.

Computer Graphics Handbook
Mortenson, Michael E.
1990 Industrial Press Inc.

Numerical Recipes
Press, Flannery, Saul, Vetterling
1987 Cambridge University Press

Projective Geometry and its Apllications to Computer Graphics
Penna, Patterson
1986 Prentice-Hall

Algorithms Second Edition
Robert Sedgewick
1989 Addison-Wesley Publishing Company

Phigs:

A Primer for Phigs
Hopgood, Duce
1991 John Wiley & Sons

A Practical Introduction to Phigs and Phigs Plus
Howard, Hewitt, Hubbold, Wyrwas
1991 Addison-Wesley Publishing Company

Spezielle Themen:

Two-Part Texture Mappings (Artikel)
Eric A. Bier, Kenneth R.Sloan Jr.
Sept. 1986 in IEEE CG&A

The Annealing Algorithm
Otten, van Ginneken
1989 Kluwer Academic Publishers

Image Compression (Diplomarbeit)
Michael Krüger
1991 Fachhochschule Wedel

Abbildungsverzeichnis

Algorithmusverzeichnis

Stichwortverzeichnis

Fehler! Keine Indexeinträge gefunden.