Großer Beleg

Thema:
Nichtrealistische Darstellung von Gebirgen
mit OpenGL

Torsten Keil
TU Dresden, Fakultät Informatik
September 2001

Verantwortlicher Hochschullehrer und Betreuer:
Prof. Dr. Oliver Deussen

0 Inhaltsverzeichnis

0 Inhaltsverzeichnis 2
1 Einleitung und Aufgabenstellung: 3
2 Einlesen und Darstellen: 4
3 Vorüberlegungen 5
4 Das Programm und seine Bedienung 6
5 Der Algorithmus – I 7
5.1 ReadZBuffer() – Am Anfang 7
5.2 ReadZBuffer() – Die Dreiecke 7
5.3 ReadZBuffer() – Die Kanten 9
5.3.1 Farbkodierung der Kanten – I 9
5.3.2 Farbkodierung der Kanten – II 11
5.3.3 Farbkodierung der Kanten – III 11
5.4 ReadZBuffer() – Feinarbeit 12
6 Der Algorithmus – II 13
6.0 Das Programm 13
6.1 Erster Schritt – Kantenerkennung auf Pixelbasis 13
6.2 Zweiter Schritt – Kantenselektion 14
6.2.1 Kantenselektion in 12 Teilschritten 15
6.2.2 Kantenselektion - Feinarbeit 15
6.3 Dritter Schritt – Intelligente Nachbereitung 17
6.4 Wertung 18
7 Zusammenfassung und Ausblick 19
8 Beispiele 20
8.1 Welle – 40 x 40 Pixel 20
8.2 Tropfen – 40 x 40 Pixel 21
8.3 Tropfen – 100 x 100 Pixel 22
8.4 Gebirge – 100 x 100 Pixel 23
9 Abbildungsverzeichnis 24

1 Einleitung und Aufgabenstellung:

Zielstellung für diesen Beleg war die Entwicklung eines Algorithmus, der aus einer gegebenen Geometrie die Kanten selektiert, die ausreichen, um die Geometrie eindeutig visuell darzustellen. Die Geometriedaten bestehen aus einer n*m-Matrix, die für jeden Punkt $P_{ij}(x_i,z_j)$ den zugehörigen Höhenwert y_{ij} enthält (Abb. 1-1). Dieses einfache Raster diente als Grundlage einen Algorithmus zu finden (Abb. 1-2).

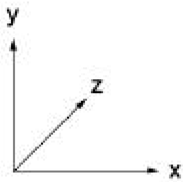


Abb. 1-1: Koordinatensystem

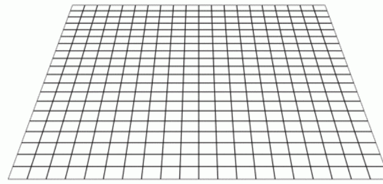


Abb. 1-2: Das dem zu entwickelnden Algorithmus zugrundeliegende Raster

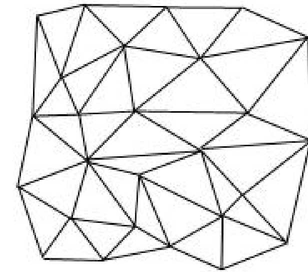


Abb. 1-3: Raster einer beliebigen Geometrie

Eine spätere Aufgabe wird sein, den Algorithmus so zu erweitern, das er auf beliebige Geometrien anwendbar ist (Abb. 1-3).

2 Einlesen und Darstellen:

Im ersten Schritt noch bevor die Darstellung erfolgt, müssen die Daten importiert werden. Dazu werden aus einer *.pgm-Datei (pixel greyscale map) die Geometriedaten gelesen. Dieses Dateiformat besteht aus ASCII-Text und dient als Austauschformat für Graustufenbilder. Die Graustufenwerte des Bildes werden fortlaufend zeilenweise in der Datei gespeichert.
In den ersten beiden Zeilen befinden sich Statusinformationen. In Zeile 3 sind die Dimensionen n und m der Geometrie bzw. der Matrix angegeben. Es folgen n*m Zeilen in denen jeweils der Graustufenwert/Höhenwert des jeweiligen Punktes auszulesen ist.
Zur Darstellung der Daten werden diese in Dreiecksdaten umgewandelt. Ein Raster aus 2*2 Punkten (Quadrat: $P_{i,j}$, $P_{i+1,j}$, $P_{i,j+1}$, $P_{i+1,j+1}$) wird dabei in 2 Dreiecke (Dreieck 1: $P_{i,j}$, $P_{i+1,j}$, $P_{i,j+1}$; Dreieck 2: $P_{i+1,j}$, $P_{i+1,j+1}$, $P_{i,j+1}$) zerlegt. Es entsteht ein Netz aus Dreiecken, welches der Landschaftsgeometrie entspricht.

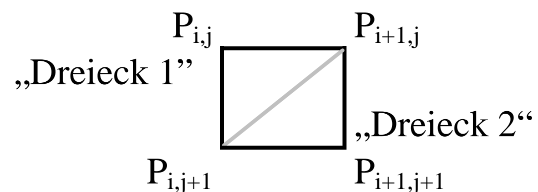


Abb. 2-1: Schema der Generierung der Dreiecksdaten aus dem quadratischen Raster

Diese Dreiecksgeometrie wird nun mittels OpenGL dargestellt. Die Punkte befinden sich in x- und z-Richtung auf dem vorgegebenen Raster, die variable y-Koordinate bestimmt das Aussehen.

3 Vorüberlegungen

Um die wenigen Kanten extrahieren zu können, müssen diese erst einmal gefunden werden. Einer ersten Überlegung nach sind alle Kanten visuell notwendig, die einen Beitrag zum Erfassen der Geometrie leisten. Also Kanten die am Rand liegen bzw. Höhen- oder Tiefenzüge kennzeichnen.
Da die Landschaft nun 3-dimensional mit Hilfe von OpenGL dargestellt ist, hat man Zugriff auf zusätzliche Informationen jedes einzelnen Bildpixels. Die wichtigste davon ist der Tiefenwert (z-Buffer-Wert) des Pixels. Die Tiefenwerte des ganzen Bildes als `unsigned Integer` erhält man mit der OpenGL-Anweisung:

```
glReadPixels(0, 0, WindowSize_Width, WindowSize_Height, GL_DEPTH_COMPONENT,
GL_UNSIGNED_INT, pixels_z);
```

Darauf hin kann man durch Vergleich der Tiefenwerte der Pixel für jedes Pixel entscheiden, ob es zur Geometrie gehört oder nicht (vordere Clipping-Ebene (Tiefenwert = 0) <= Tiefenwert des Pixels <= hintere Clipping-Ebene (Tiefenwert = maximaler Wert von `unsigned Integer`)).

Der Algorithmus muss dann entscheiden, welches Pixel zu einer notwendigen Kante gehört und die entsprechende Kante aus den Bildinformationen extrahieren.

4 Das Programm und seine Bedienung

Das Programm lädt beim Start standardmäßig die Datei „flaeche.pgm“. Sollte diese Datei nicht vorhanden sein, so wird es sofort wieder beendet, da keine Daten zur Verfügung stehen.
Während der Programmlaufzeit sind folgende Tasten zur Steuerung der Programmfunktionen vorgesehen:

Taste	Funktion
x, y, z	Rotation der Geometrie in math. positiver Drehrichtung um die jeweilige Achse
X, Y, Z (Shift + Taste)	Rotation der Geometrie in math. negativer Drehrichtung um die jeweilige Achse
+, -	Inkrementiert, Dekrementiert den Drehwinkel um jeweils 10°
k, l	Translation der Geometrie in x-Richtung (links/rechts)
,, .	Translation der Geometrie in y-Richtung (auf/ab)
i, o	Translation der Geometrie in z-Richtung (Zoom in/out)
s, S	Dekrementiert/Inkrementiert den Schwellwert
m	Schaltet zwischen den beiden Zeichen-Modi um
t	Zusätzlicher Schalter für die Darstellung
r	Startet den Algorithmus
Leertaste	Zurücksetzen aller Werte auf Standard-Werte
Ziffern 0..9	Auswahl eines Geometriemodells

Abb. 4-1: Programmfunktionen

Außerdem muss im System eine Farbtiefe von mind. 24 Bit eingestellt sein, da die genauen Farbinformationen die Grundlage für den Algorithmus darstellen.

5 Der Algorithmus – I

5.1 ReadZBuffer() – Am Anfang

Zu Beginn entstand die Idee, einfach den z-Buffer auszulesen und anhand dieser Informationen die entsprechenden Pixel der Kanten zu finden. Die erste Implementierung begann also damit den Tiefenwert eines Pixels mit dem des nachfolgenden Pixels zu vergleichen. Sollte der Betrag der Differenz der Tiefenwerte einen vorher bestimmten, statischen Schwellwert überschreiten, so merkt sich der Algorithmus dieses Pixel. Er speichert dazu für jedes Pixel einen Boolean-Wert.

5.2 ReadZBuffer() – Die Dreiecke

Da nun die entsprechenden Pixel im ersten Schritt gefunden waren, stand nun zur Aufgabe, zu jedem Pixel das dazugehörige Dreieck der Geometrie zu finden. Da die Dreiecke in einem definierten Raster liegen, ist es einfach, mit Hilfe von Indizes auf diese zuzugreifen. Um aber bestimmte Dreiecke zu selektieren, benötigt man einen Weg vom Pixel zum Dreieck. Dabei werden neben den Tiefeninformationen auch die Farbinformationen des Bildes ausgewertet.
Im ersten Algorithmus wurden die Dreiecke mittels Smooth-Shading je nach Wert der y-Koordinate gefärbt. Dies wird nun durch einen eindeutige Farbgebung eines jeden Dreiecks ersetzt. D.h. jedes „Dreieck 1“ erhält eine Farbkodierung nach folgendem Schema: `glColor3ub(255-i, 255-j, 0)`.
Analog dazu jedes „Dreieck 2“: `glColor3ub(255-i, 255-j, 255)`; i und j entsprechen den Indizes des jeweiligen Dreiecks in x- und y-Richtung.

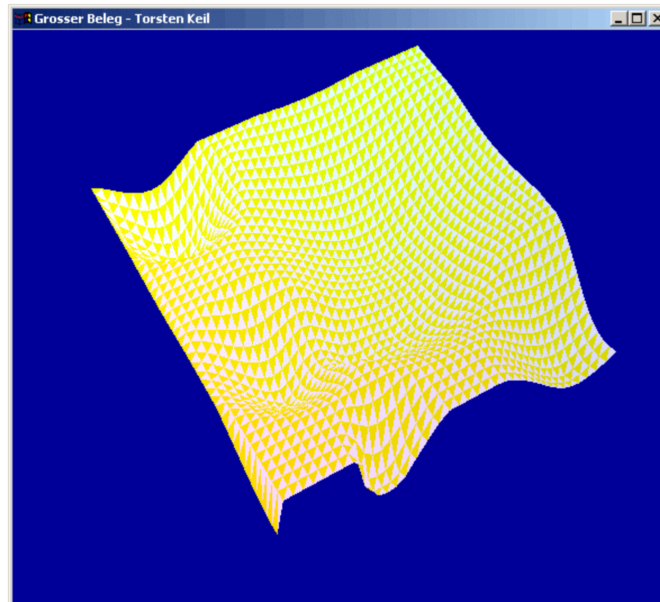


Abb. 5-1: Farbkodierung der Dreiecke durch Farbindizes

Auf diese Weise ist es möglich, mit Hilfe der Tiefen- und Farbinformationen auf den Index des Dreieckes zu schließen. Diese wurden wieder in einem Feld als Boolean-Werte gespeichert.
Nachfolgend wurde eine neue Displayliste nur mit den selektierten Dreiecken erstellt und angezeigt.

Dieses erste visuelle Zwischenergebnis offenbarte Fehler in der Auswahl der Pixel, so dass der erste Schritt überarbeitet werden musste. Der Fehler dabei war, das nur ein einziger Vergleich durchgeführt wurde (der mit dem horizontal nachfolgenden Pixel). So wurden zwar Kanten der Geometrie erkannt, aber nicht ausreichend genau. Das bedeutet:

- Beim horizontalen Übergang von Hintergrund zu Geometrie wird das letzte Pixel des Hintergrunds gemerkt (ungenau).
- Beim horizontalen Übergang von Geometrie zu Hintergrund wird das letzte Pixel der Geometrie gemerkt (korrekt).
- Andere Übergänge (z.B. vertikal) wurde nicht berücksichtigt (es entstehen Lücken).
- Für Pixel innerhalb der Geometrie treten diese Fehler natürlich ebenfalls in ähnlicher Form auf.

Als Konsequenz davon wurde dieser Schritt auf horizontale und vertikale Vergleiche, sowohl mit dem vorhergehenden, als auch mit dem nachfolgenden Pixel erweitert (4 Vergleiche). Nachdem alle Pixel verglichen wurden, kann in einem weiteren Schritt das Ergebnis zur Kontrolle visuell dargestellt werden

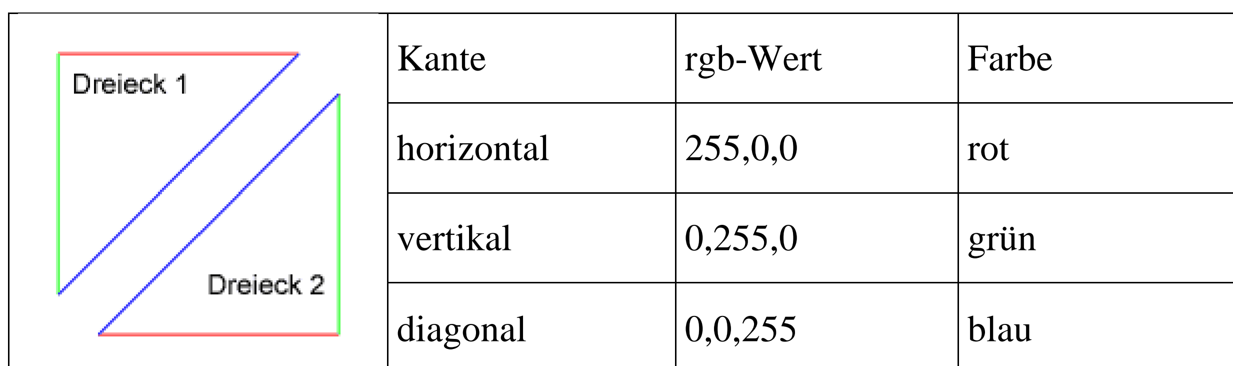
(wechseln des Zeichen-Modus; siehe Tabelle). Dazu wird direkt in den Bildspeicher geschrieben, wobei jedes gemerkte Pixel schwarz dargestellt wird, alle anderen weiß.
Dadurch das mit dem vorhergehenden und nachfolgenden Pixel verglichen wird, entsteht an einem Übergang wie oben beschrieben eine zwei Pixel starke Linie. Das erste Pixel wird selektiert, da beim Vergleich zwischen dem aktuellen Pixel und dem nachfolgenden der Schwellwert überschritten wird. Das nachfolgende Pixel wird auch selektiert, da der Vergleich mit dem vorhergehenden Pixel den selben Wert ergibt, der den Schwellwert überschreitet.

5.3 ReadZBuffer() – Die Kanten

Um die endgültige Geometrie, bestehend nur aus den wenigen notwendigen Kantenzügen, zu erhalten, muss die Auswahl von den Dreiecken auf die Kanten verfeinert werden.
Da nicht jeder Algorithmus zu Beginn optimale Ergebnisse liefert, seien die verschiedenen Ansätze im folgenden dargestellt.

5.3.1 Farbkodierung der Kanten – I

Nach den ersten beiden Schritten waren die entsprechend wichtigen Dreiecke selektiert. Diese werden nun nicht mehr als Dreiecksprimitive, sondern als einzelne Linien (Kanten) dargestellt. Dabei erhalten die verschiedenen Kanten verschiedene Farben:



Kante	rgb-Wert	Farbe
horizontal	255,0,0	rot
vertikal	0,255,0	grün
diagonal	0,0,255	blau

Abb. 5-2: Farbkodierung der Kanten

So ist es möglich, pro Pixel zu entscheiden, ob es eine Kante ist (1.Schritt), zu welchem Dreieck es gehört (2.Schritt) und schließlich welche Kante es ist.

Auch hier wurde ein Feld mit einem Boolean-Wert für jede Kante geführt. Der Wert einer Kante war nach dem 3. Schritt `true`, wenn mindestens ein Pixel der

Kante auch gleichzeitig einem Pixel der ausgewählten Pixelmenge aus dem ersten Schritt entsprach.
Alle Kanten mit Wert `true` sind notwendige Kanten.

Wertung: Der Algorithmus arbeitet recht zuverlässig für Kanten am Rand der Geometrie, aber zu ungenau, da angrenzende Kanten ebenfalls ausgewählt werden. Einen Erklärung dafür liegt darin, dass jede Kante zu 2 Dreiecken und jeder Eckpunkt zu 6 Dreiecken gehört. Die Pixel überlagern sich. Welches Pixel von welchem Dreieck (mit welcher Farbe) gezeichnet wird, ist im Endeffekt von der OpenGL-Implementierung, der Auflösung und der Genauigkeit (Zeichengenauigkeit, Genauigkeit des z-Buffers u.ä.) abhängig.

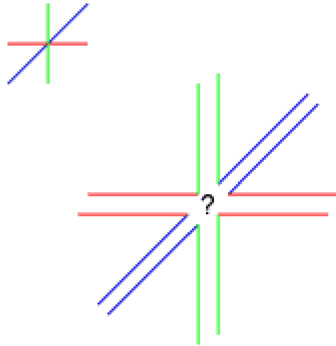


Abb. 5-3: Problemstellen der Farbkodierung – die Eckpunkte

5.3.2 Farbkodierung der Kanten – II

Um die Entscheidungsgenauigkeit zu erhöhen und den Algorithmus so zu verbessern, wurde das Feld der Boolean-Werte der Kanten durch ein Integer-Feld ersetzt, das die Pixel-Treffer pro Kante zählt. So konnte nun flexibler darüber entschieden werden, wann eine Kante gezeichnet wird. Eine Kante mit Zähler ≥ 5...8 wird mit Sicherheit ausgewählt. Andere Kanten mit Zähler ≤ 2 werden eventuell nur an den Eckpunkten erfasst und können als angrenzende Kanten identifiziert werden und dementsprechend nicht gezeichnet. Dieses Verfahren hat eine gute Genauigkeit für Kanten die parallel oder annähernd parallel zur Betrachterebene liegen. Durch räumliche Streckung oder Stauchung von Kanten wird es aber zunehmend ungenauer. Als Folge davon können stark gestauchte Kanten aussortiert werden und hinterlassen somit Lücken im Kantenzug. Andere, stark gestreckte Kanten, erhalten übermäßig viele Pixel-Treffer und werden gezeichnet.

Ein weitere Versuch die Genauigkeit zu erhöhen bestand darin, die Pixel bei denen die meisten Fehler auftreten – die Eckpunkte der Dreiecke – von der Zählung auszuschließen. Sie werden „gelöscht“, indem sie als Punkte in der Hintergrundfarbe überzeichnet werden. Dadurch werden die Probleme an den Ecken minimiert, aber gleichzeitig erhöht sich der Fehler bei stark gestauchten Kanten.

5.3.3 Farbkodierung der Kanten – III

Um die zu gewinnenden Kanten-Informationen gerade an den Ecken exakter dem richtigen Dreieck zuordnen zu können wurde der Algorithmus ein weiteres Mal modifiziert. In den vorangegangenen Verfahren wurden alle Kanten der zuvor selektierten Dreiecke in einem Durchgang gezeichnet, wodurch es zu Überlagerungen kam. Diese Überlagerungen können vermieden werden, wenn jede Kante einzeln gezeichnet wird. Dazu wird der Algorithmus in 6 Schleifen zerlegt.
In der ersten Schleife wird vom ersten selektierten Dreieck vom Typ „Dreieck 1“ nur die horizontale Kante gezeichnet. Nachdem diese gezeichnet wurde, wird jedes selektierte Pixel darauf getestet, ob es auf dieser Kante liegt. Falls ja wird der Zähler für diese Kante inkrementiert. Es wird mit der horizontalen Kante des nächsten selektierten Dreiecks vom Typ „Dreieck 1“ fortgefahren. Die Schleife läuft bis alle selektierten Dreiecke vom Typ „Dreieck 1“ getestet wurden.
Analog dazu wird in den Schleifen 2 und 3 mit den vertikalen und horizontalen Kanten der selektierten Dreiecke vom Typ „Dreieck 1“ verfahren.

Natürlich wird dies für die Dreiecke vom Typ „Dreieck 2“ wiederholt (ebenfalls je eine Schleife für die horizontale, vertikale und diagonale Kante).
Wie man leicht sieht, steigt bei diesem Verfahren der Aufwand um ein Vielfaches an. Die Kanten-Selektion wird etwas verbessert, ist aber noch nicht optimal.

5.4 ReadZBuffer() – Feinarbeit

Der Weg der Selektion vom einfachen Pixel bis zum Kantenzug ist mit der beschriebenen Vorgehensweise der Farbkodierung ausgereizt. Verbesserungen waren aber bei der Auswahl der Pixel, also im ersten Schritt, notwendig.
Das Problem liegt in dem statischen Schwellwert.

- Ist dieser zu groß gewählt, so werden keine Pixel ausgewählt.
- Ist er zu klein, werden fast alle Pixel ausgewählt und somit auch jede Kante.
- Nähert man sich von oben an den optimalen Wert an, so erreicht man einen Wertebereich, in dem die äußeren Kanten der Geometrie gut erkannt werden. Aber nur diese.
- Will man auch Kantenzüge innerhalb der Geometrie gut erkennen, muss man den Schwellwert weiter minimieren. Allerdings kommt man dann sehr schnell wieder in den Bereich, in dem zu viele Pixel gewählt werden.

Um nun das Problem des statischen Schwellwertes zu beheben, wird eine Art statistischer Erhebung über die Tiefenwerte durchgeführt. Dabei werden folgende Werte zusätzlich ermittelt: Der maximale und minimale Tiefenwert aller Pixel die zur Geometrie gehören und zu jedem Pixel die maximale und minimale Differenz der Tiefenwerte des aktuellen Pixels mit den benachbarten Pixel. Mit diesen Informationen sollte für jedes Pixel der Schwellwert so angepasst werden, dass eine bessere Selektion von Kantenzügen möglich ist.

Zwei dieser Anpassungsversuche seien hier erwähnt:

```
Schwelle = max((z_max - z_min) / 2, (absolute_max[j]-absolute_min[j]) / 2);

Schwelle = (((z_max - z_min) / 2) + ((absolute_max[j]-absolute_min[j]) /
2)) / (4*2);
```

Auch hier konnte wieder eine minimale Verbesserung hinsichtlich der visuellen Darstellung erreicht werden. Das Ergebnis zeigt aber deutlich, dass noch immer nicht alle Kanten korrekt erkannt werden.

6 Der Algorithmus – II

Zum Abschluss des Beleges sollte ein Programm stehen, das von den bisher gesammelten Erfahrungen profitiert und mit einem weiteren neuen Algorithmus diese effizient nutzt. Eine Bestandsaufnahme der schon vorhandenen Funktionalität und eine Analyse der noch nicht umgesetzten Anforderungen sollte die Entwicklung eines fast komplett neuen Algorithmus vereinfachen. Im folgenden sind die wichtigsten Punkte zusammengefasst.

6.0 Das Programm

Die vorhandenen Programmfunktionen zum Einlesen, Darstellen und Bedienen wurden beibehalten. Die Dreiecke der Geometrie werden weiterhin so farblich dargestellt, dass aus ihnen die Rasterindizes gewonnen werden können (siehe Abschnitt 5.2).

6.1 Erster Schritt – Kantenerkennung auf Pixelbasis

Im ersten Teilschritt wird für jedes Pixel entschieden, ob es zur Geometrie gehört, also einen z-Buffer-Wert kleiner als der maximale z-Buffer-Wert besitzt. Dies wird pro Pixel als Boolean-Wert erfasst.
Zusätzlich existiert zu jedem Pixel ein Integer-Zähler, der die Werte 0 ... 9 enthalten kann. Dieser Zähler ist nach dem ersten Teilschritt 1, wenn das Pixel zur Geometrie gehört und 0 sonst. Im zweiten Teilschritt wird dieser Zähler für jedes Nachbarpixel das zur Geometrie gehört um 1 inkrementiert. Mit diesen Informationen kann man für jedes Pixel entscheiden ob es ein Randpixel der Geometrie ist oder inmitten der Geometrie liegt (Abb. 6-1).

						0	0	0	
			0	0	0	0	1	2	
	0	0	0	1	2	3	4	5	
0	0	1	2	4	5	6	7	8	
0	1	3	5	7	8	9	9	9	
0	2	5	8	9	9	9			
0	3	6	9	9					

Abb. 6-1: Schema der Kantenerkennung auf Pixelbasis

In einem 3.Teilschritt werden alle Pixel, die zur Geometrie gehören anhand der Tiefeninformationen darauf getestet, ob sie zu einer potentiellen Kante gehören. Dabei wird jedes Pixel sowohl horizontal als auch vertikal auf ein lokales Maximum oder Minimum untersucht (Vergleich mit vorhergehendem und nachfolgendem Pixel). Nur an dieser Stelle im ersten Schritt des Algorithmus wird ein Schwellwert eingesetzt, um die Genauigkeit der Kantenerkennung steuern zu können. Auch dies wird pro Pixel als Boolean-Wert erfasst.

Das Zwischenergebnis der Kantenerkennung kann grafisch dargestellt werden (wechseln des Zeichen-Modus). Die nachfolgende Tabelle liefert einen Überblick über die Farbkodierung der Pixel:

Farbe des Pixels	Bedeutung
weiß	Pixel gehört nicht zur Geometrie
schwarz	Randpixel der Geometrie
grau	Pixel innerhalb der Geometrie
rot	Pixel innerhalb der Geometrie und potentielles Kantenpixel

Abb. 6-2: Farbdefinition und deren Bedeutung bei der grafischen Darstellung der Kantenerkennung

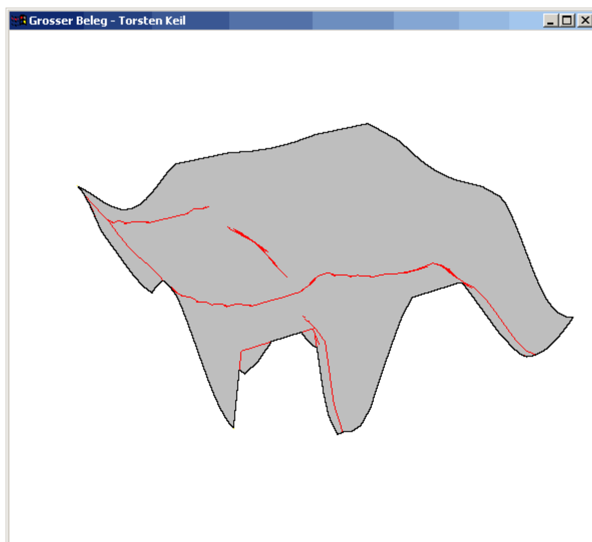


Abb. 6-3: Grafische Darstellung der Kantenerkennung auf Pixelbasis

Vorteile gegenüber den vorhergehenden Verfahren:

- Der Algorithmus erkennt Pixel der Geometrie genau.
- Der Algorithmus erkennt ein Randpixel genau, d.h. die Randpixelkante (schwarz) ist genau ein Pixel breit.
- Der Algorithmus benötigt theoretisch keinen Schwellwert, dieser wird hauptsächlich dazu genutzt, die Genauigkeit skalierbar zu halten.

6.2 Zweiter Schritt – Kantenselektion

Die Schritte 2 (Dreiecksselektion) und 3 (Kantenselektion) der vorhergehenden Implementierungen werden nun miteinander verknüpft. Dies lässt sich ohne Probleme realisieren, da die Farbinformationen je Pixel (Indizes der Dreiecke) schon von Beginn des Algorithmus an vorliegen.
Um die Fehler zu minimieren werden folgende Überlegungen berücksichtigt:

(1) benachbarte Kanten sollen sich nicht verdecken/überlagern
(2) das Problem der Eckpunkte soll gelöst werden
(3) die Kanten sollen eindeutig zugeordnet werden können (horizontale, vertikale, diagonale Kanten der Dreiecke vom Typ „Dreieck 1“ bzw. „Dreieck 2“

6.2.1 Kantenselektion in 12 Teilschritten

Dieser Schritt wird in 12 vom Ablauf her identische Teilschritte zerlegt. Um Punkt (3) gerecht zu werden wird für jeden Kantentyp eine eigene Schleife durchlaufen → 6 Teilschritte. Damit, wie unter Punkt (1) gefordert, sich keine benachbarten Kanten beeinflussen wird jeder dieser 6 Teilschritte noch einmal in 2 Teilschritte zerlegt (siehe Abb. 6-4) → 12 Teilschritte. Punkt (2) wird berücksichtigt, indem die Eckpunkte der Kanten wie in 5.3.2 beschrieben gelöscht werden.

				n
				n+2
				n+4

Vertikale Kanten – 1. Teilschritt

				n+1
				n+3

Vertikale Kanten – 2. Teilschritt

n		n+2		n+4

Horizontale Kanten – 1. Teilschritt

	n+1		n+3	

Horizontale Kanten – 2.Teilschritt

Abb. 6-4: Schema der Kantenselektion

Bei diagonale Kanten wird analog zu den Horizontalen Kanten vorgegangen.

In jedem Teilschritt werden die entsprechenden Kanten gezeichnet und als Farbinformationen gespeichert. Anschließend werden alle in Schritt 1 selektierten Pixel getestet, ob sie auf solch einer Kante liegen und wenn ja, ermittelt zu welchem Dreieck diese Kante gehört (jeweils über die Farbinformationen). Zu jeder Kante eines Dreiecks existiert wieder ein Zähler, der um 1 inkrementiert wird, wenn ein solches Pixel zu dieser Kante gehört.

6.2.2 Kantenselektion - Feinarbeit

An dieser Stelle tritt ein weiteres Problem zu Tage: Obwohl es durch diese Methodik möglich ist, die jeweilige Kante exakt zu bestimmen, werden einzelne Kanten nicht erfasst. Grund dafür ist eine abweichende Behandlung unterschiedlicher Primitive unter OpenGL: Bei horizontalen, vertikalen und bestimmten diagonalen Kanten existiert eine Abweichung der Position um

genau ein Pixel. Dies betrifft im speziellen Fall die Kanten von Dreiecken im Vergleich zu Linien.
Gelöst wird dieses Problem, indem zu jedem selektierten Pixel erfasst wird, ob ihm eine Kante zugeordnet werden konnte. Anschließend werden alle Pixel die keiner Kante zugeordnet werden konnten noch einmal gesondert betrachtet:

- Die Dreiecksindizes des Pixels werden ermittelt.
- Die benachbarten Pixel (davor, danach, darüber und darunter) werden getestet, ob sie auf einer Kante liegen. Ist dies der Fall, wird der Kantentyp dieser Kante ermittelt.

→ Der Zähler der Kante des jeweiligen Dreiecks wird inkrementiert.

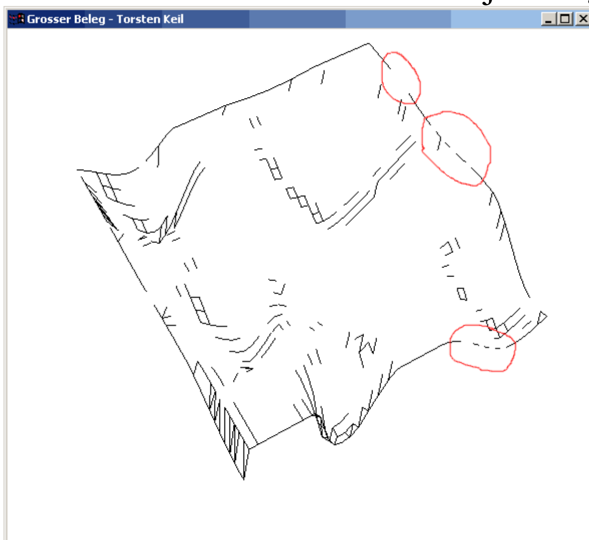


Abb. 6-5: Ergebnis nach „6.2.1 Kantenselektion in 12 Teilschritten“

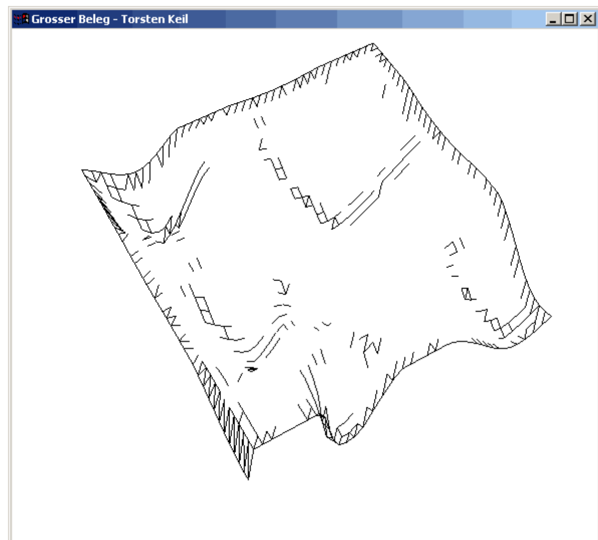


Abb. 6-6: Ergebnis nach „6.2.2 Kantenselektion - Feinarbeit“

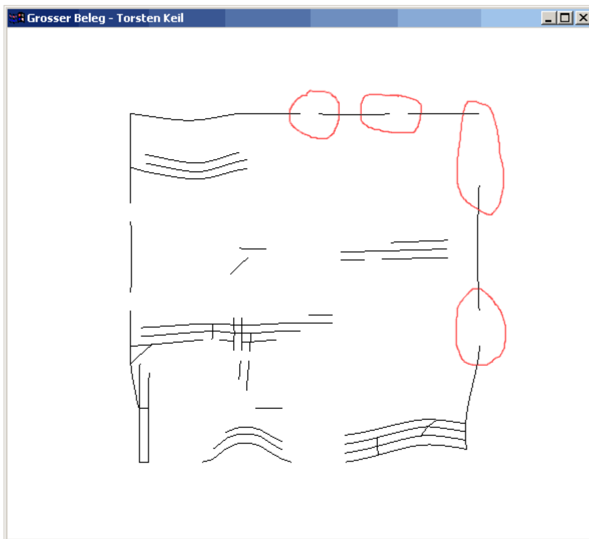


Abb. 6-7: Ergebnis ohne „6.2.2 Kantenselektion - Feinarbeit“ und mit „6.3 Dritter Schritt – Intelligente Nachbereitung“

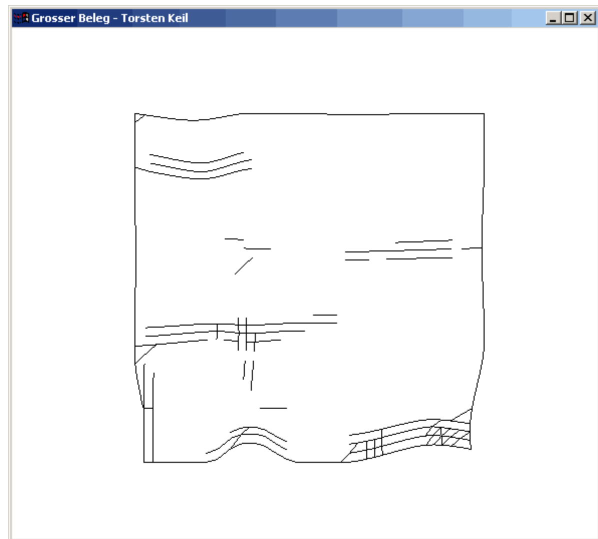


Abb. 6-8: Ergebnis des kompletten Algorithmus (6.1 bis 6.3)

6.3 Dritter Schritt – Intelligente Nachbereitung

Um zum Schluss ein visuell ansprechendes Ergebnis zu erhalten, wird noch eine „intelligente Nachbereitung“ der ermittelten Kanten angeschlossen. Diese Nachbereitung soll zum einen einzelne Lücken in Kantenzügen schließen und zum anderen einzelne selektierte Kanten eliminieren. Dazu werden zuerst die Kanten auf die des „Dreiecks 1“ normiert. Im speziellen Fall bedeutet dies, das die Zähler, von den beiden Kanten die sich jeweils überlagern, addiert werden und unter der Kante des „Dreiecks 1“ gespeichert werden. Durch diese „Normierung“ gehen keine Kanteninformationen verloren, da man allein durch die Kanten des „Dreiecks 1“ das komplette Raster erhält.

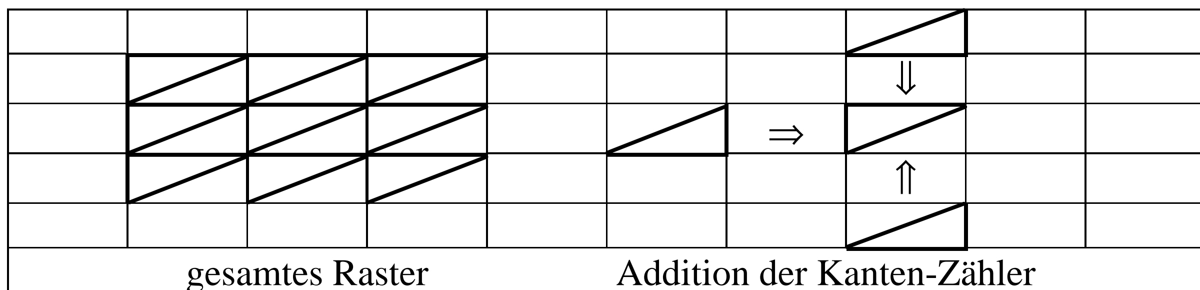


Abb. 6-9: „Normierung“ der Kanten auf die des „Dreiecks 1“

Danach kann einfacher auf Lücken bzw. einzelne Kanten getestet werden. Wie nachfolgendes Beispiel zeigt spielt die Reihenfolge dabei eine entscheidende Rolle:

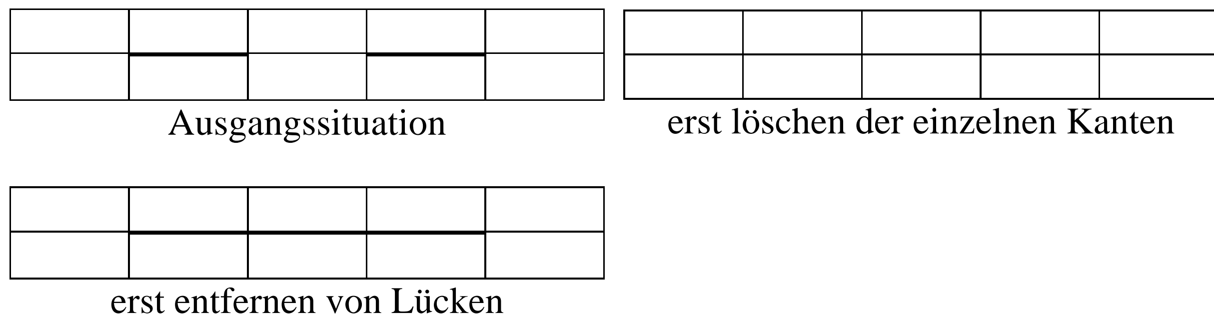


Abb. 6-10: Entfernen von Lücken und löschen einzelner Kanten

Werden einzelne Kanten zuerst gelöscht, gehen Informationen (Kanten) verloren bzw. verschlechtern das visuelle Ergebnis definitiv.

Wichtig: An den äußeren Kanten der Geometrie treten Sonderfälle bei den beschriebenen Verfahren auf und müssen gesondert behandelt werden.

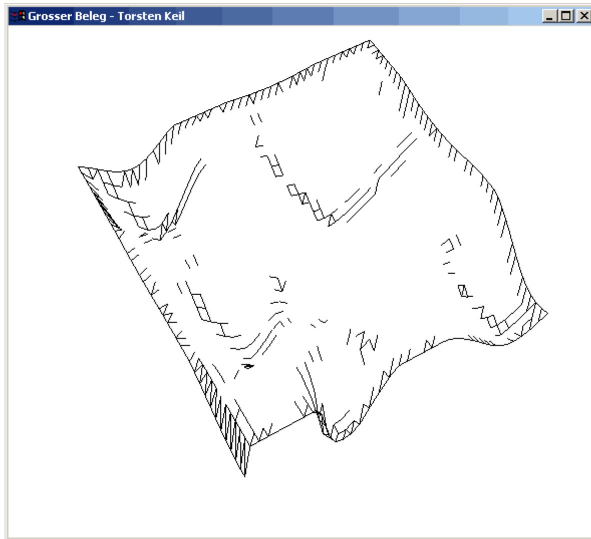


Abb. 6-11: Ergebnis nach „6.2.2 Kantenselektion - Feinarbeit“

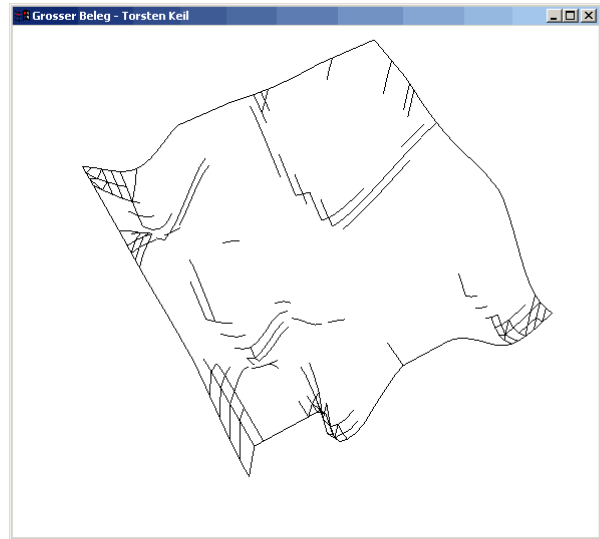


Abb. 6-12: Ergebnis des kompletten Algorithmus

6.4 Wertung

Der erste Schritt, die Kantenerkennung, arbeitet zuverlässig und liefert für die anschließenden, darauf aufbauenden Teilschritte die nötige Grundlage.

Der Zweite Schritte, die Kantenselektion, ist der aufwändigste Teil des Verfahrens. Bezüglich Zeitaufwand konnte eine deutliche Verbesserung gegenüber dem in 5.3.3 beschriebenen Verfahren erreicht werden. Der Zeitaufwand setzt sich aus der benötigten Zeit der beiden unter 6.2.1 und 6.2.2 genannten Teile zusammen. Den Hauptanteil daran nimmt das Auslesen des Bildspeichers in Anspruch. Dieser ist abhängig von der Auflösung (Fenstergröße) aber unabhängig von der Geometrie. Der Anteil der „Feinarbeit“ dagegen ist vernachlässigbar, da er mit bereites erfassten Daten arbeitet. Vervielfacht hat sich dagegen der Speicherbedarf. Ohne den Teil der „Feinarbeit“ konnte der Speicherbereich in den der Bildinhalt geschrieben wird in jedem der 12 Teilschritte wiederverwendet werden. Da die „Feinarbeit“ aber genau diese Daten benötigt, muss nun für jeden Teilschritt ein eigener Speicherbereich reserviert werden (12-facher Aufwand).
Der durch diese Maßnahmen gewonnene Vorteil liegt auf der Hand: Verbesserung von Qualität und Quantität der Kantenerkennung.

Der dritte Schritt, die Nachbereitung rundet das Gesamtbild der Kantenselektion ab.

7 Zusammenfassung und Ausblick

Der entwickelte Algorithmus liefert gute bis sehr gute Ergebnisse. Die Qualität des Algorithmus ist jedoch von einigen Faktoren abhängig, so das man sagen muss, dass er nicht vollautomatisch das beste Ergebnis für jede Geometrie liefert. Nachfolgend sind noch einmal einige Vor- und Nachteile des Verfahrens aufgeführt, sowie Ideen den Algorithmus zu verbessern:

- Ungenügende Genauigkeit hat negativen Einfluss auf das Verfahren (z.B. geringe Bildschirmauflösung, niedrige Genauigkeit des z-Buffers).
- Eine aus vielen kleinen Primitiven zusammengesetzte Geometrie verringert die Leistungsfähigkeit.
- Die Methode der Farbindizes begrenzt das Verfahren auf 8 388 608 Dreiecke, 16 777 216 Quadrate bzw. ein 4096*4096 Raster bei 24 Bit Farbtiefe.
- ✓ Eine Erhöhung der Bildschirmauflösung führt zu einer Steigerung der Qualität.
- ✓ Eine flexiblere Unterteilung der Quadrate in Dreiecke (bezüglich der Richtung der Diagonalen) kann zu einer besseren visuellen Darstellung führen.

Um das Verfahren auf ein beliebiges Dreiecksnetz zu erweitern, muss hauptsächlich das Problem der Lokalität der Kanten gelöst werden. Der erste Schritt dieses Algorithmus, die Selektion der Pixel, kann übernommen werden. Ist eine geeignete Datenstruktur für die Geometriedaten gefunden (gemeinsame Kanten und benachbarte Dreiecke) sollte auch eine Farbindizierung für die Dreiecke möglich sein. Eventuell kann solch eine Datenstruktur beim schon beim Import der Daten gegeben sein. Die Selektion der Kanten muss allerdings an die Datenstruktur angepasst werden. Durch die flexiblere Geometrie eines solchen variablen Dreiecksnetzes sollte die Selektion der Kanten visuell ansprechende Ergebnisse liefern.

8 Beispiele

8.1 Welle – 40 x 40 Pixel

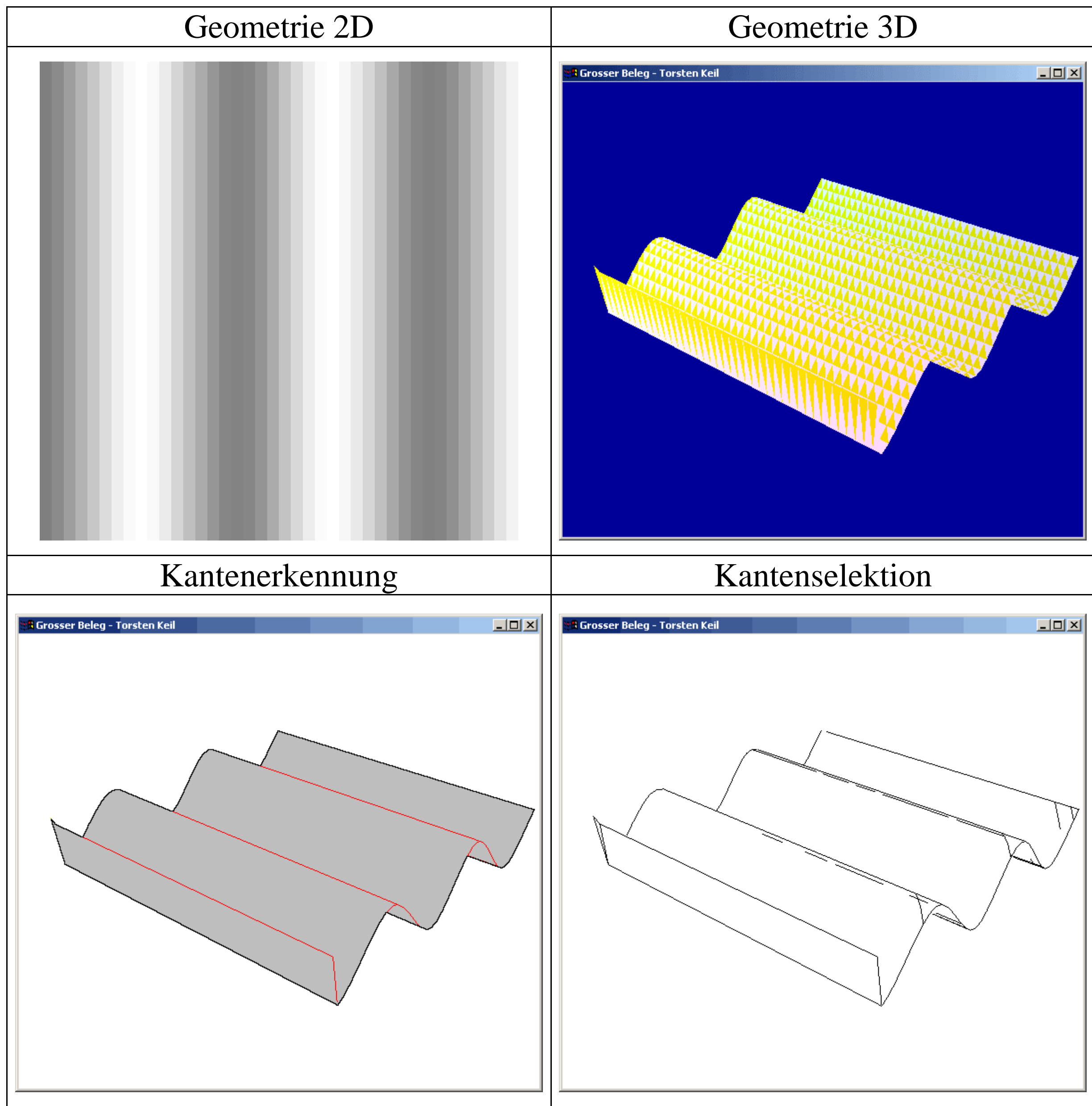


Abb. 8-1: Beispiel 1 – Welle (40 x 40)

8.2 Tropfen – 40 x 40 Pixel

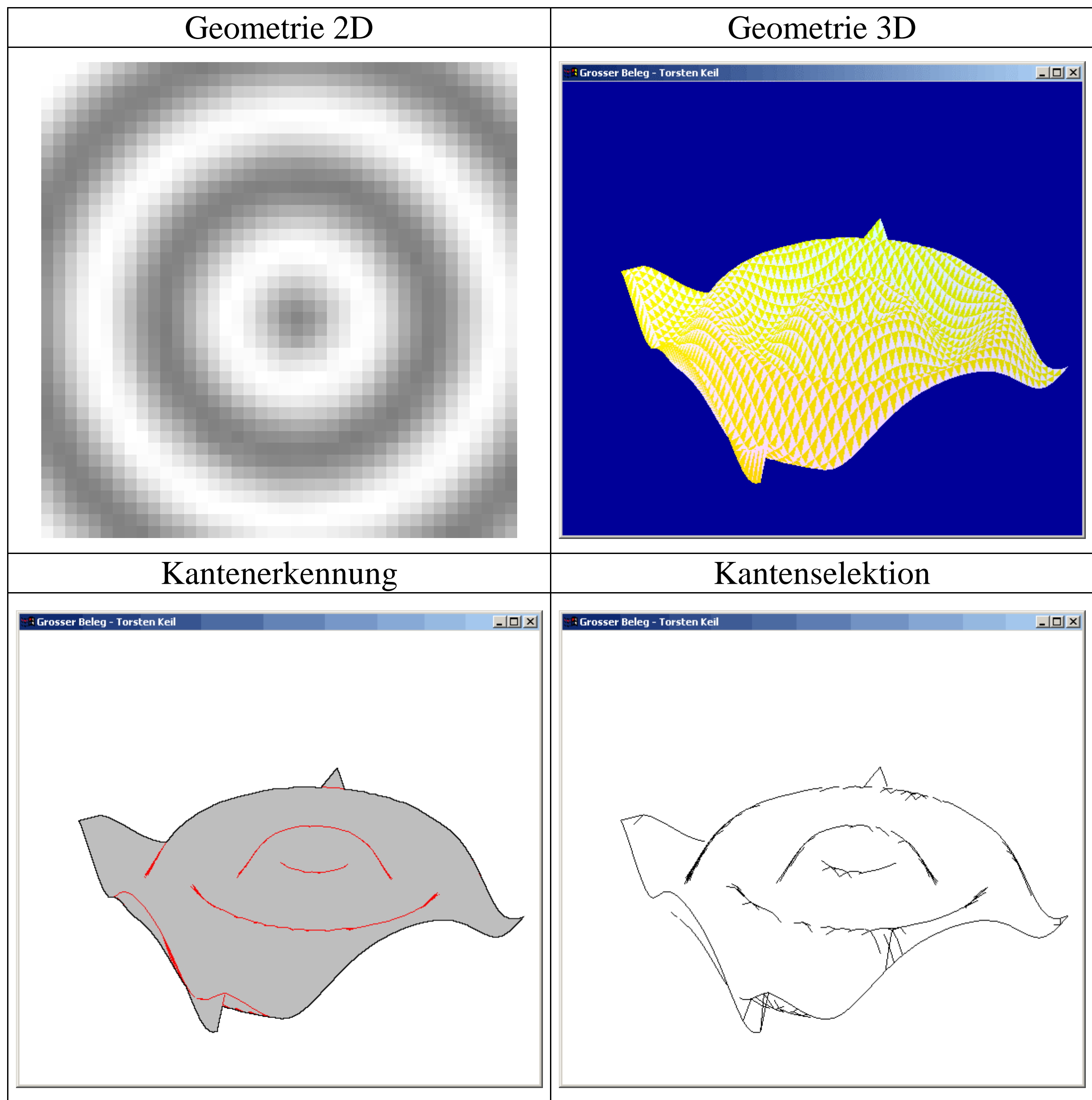


Abb. 8-2: Beispiel 2 – Tropfen (40 x 40)

8.3 Tropfen – 100 x 100 Pixel

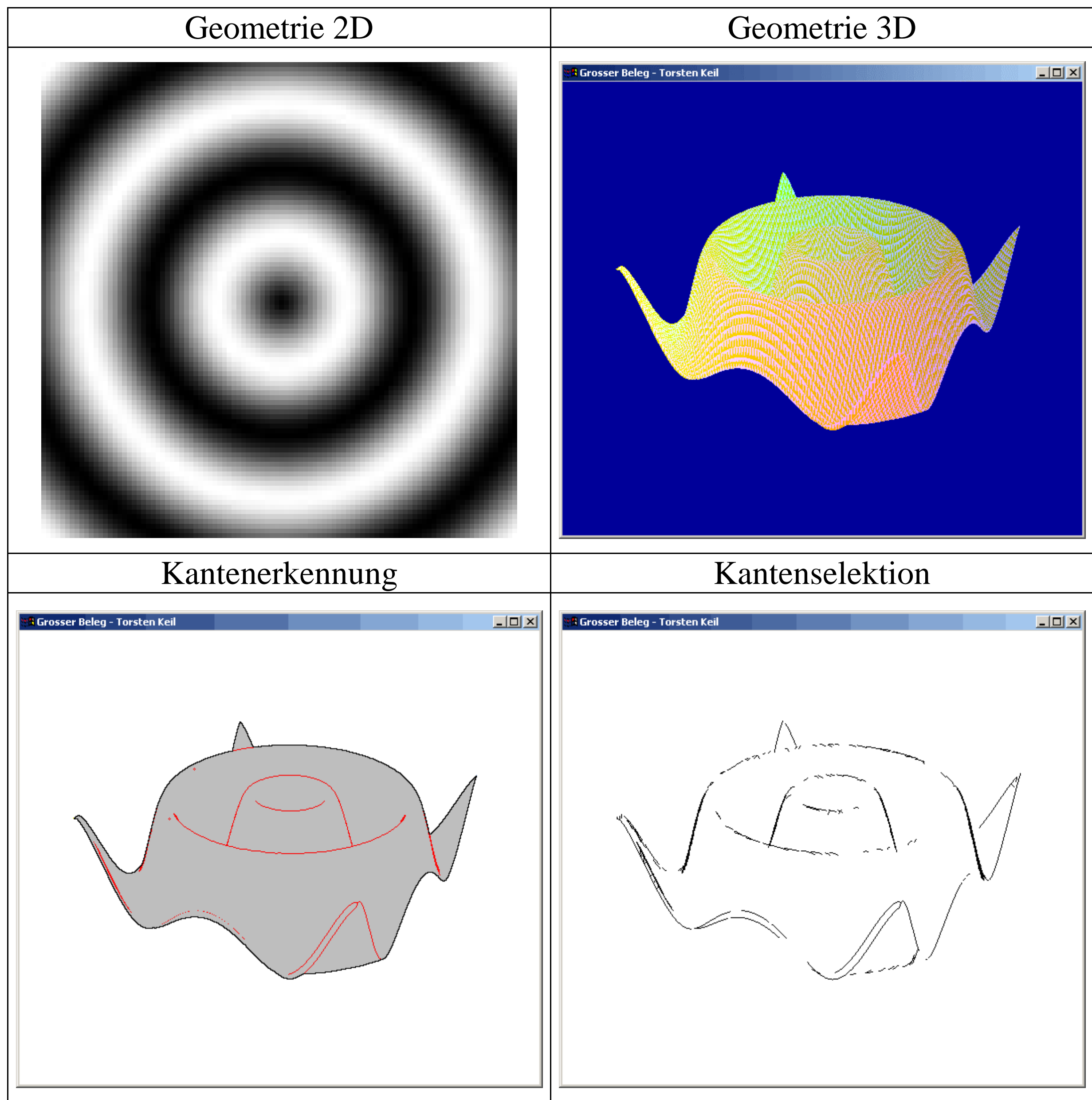


Abb. 8-3: Beispiel 3 – Tropfen (100 x 100)

8.4 Gebirge – 100 x 100 Pixel

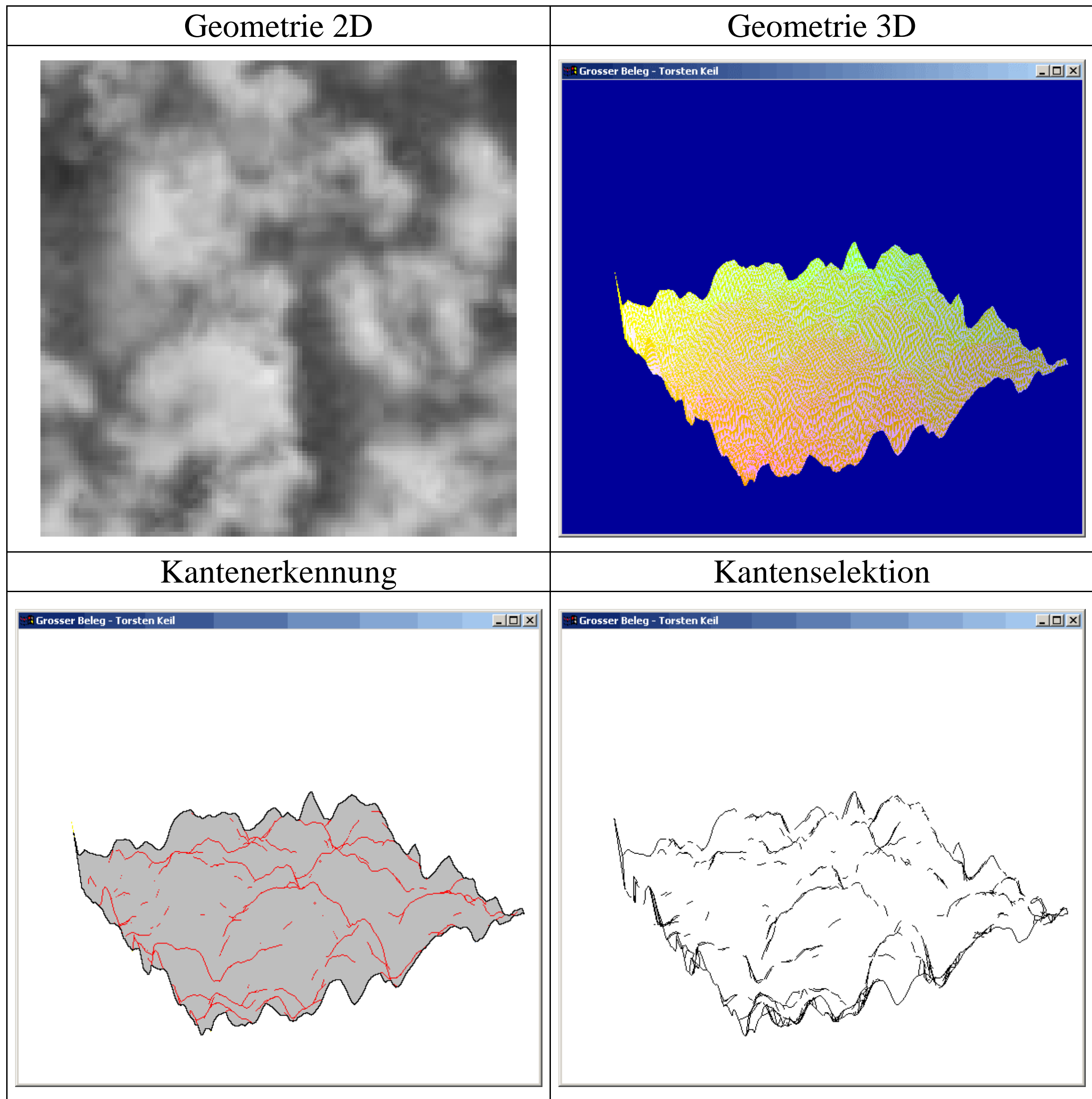


Abb. 8-4: Beispiel 4 – Gebirge (100 x 100)

9 Abbildungsverzeichnis

Abb. 1-1: Koordinatensystem 3
Abb. 1-2: Das dem zu entwickelnden Algorithmus zugrundeliegende Raster 3
Abb. 1-3: Raster einer beliebigen Geometrie 3
Abb. 2-1: Schema der Generierung der Dreiecksdaten aus dem quadratischen Raster 4
Abb. 4-1: Programmfunktionen 6
Abb. 5-1: Farbkodierung der Dreiecke durch Farbindizes 8
Abb. 5-2: Farbkodierung der Kanten 9
Abb. 5-3: Problemstellen der Farbkodierung – die Eckpunkte 10
Abb. 6-1: Schema der Kantenerkennung auf Pixelbasis 13
Abb. 6-2: Farbdefinition und deren Bedeutung bei der grafischen Darstellung der Kantenerkennung 14
Abb. 6-4: Schema der Kantenselektion 15
Abb. 6-5: Ergebnis nach „6.2.1 Kantenselektion in 12 Teilschritten" 16
Abb. 6-6: Ergebnis nach „6.2.2 Kantenselektion - Feinarbeit" 16
Abb. 6-7: Ergebnis ohne „6.2.2 Kantenselektion - Feinarbeit" und mit „6.3 Dritter Schritt – Intelligente Nachbereitung" 16
Abb. 6-8: Ergebnis des kompletten Algorithmus (6.1 bis 6.3) 16
Abb. 6-9: „Normierung" der Kanten auf die des „Dreiecks 1" 17
Abb. 6-10: Entfernen von Lücken und löschen einzelner Kanten 17
Abb. 6-11: Ergebnis nach „6.2.2 Kantenselektion - Feinarbeit" 18
Abb. 6-12: Ergebnis des kompletten Algorithmus 18
Abb. 8-1: Beispiel 1 – Welle (40 x 40) 20
Abb. 8-2: Beispiel 2 – Tropfen (40 x 40) 21
Abb. 8-3: Beispiel 3 – Tropfen (100 x 100) 22
Abb. 8-4: Beispiel 4 – Gebirge (100 x 100) 23