Dokumentation: Terraindigitalisierung

Andreas Reifenberger, Simon Rube

08. April 2008

Inhaltsverzeichnis

1 Einleitung

1.1 Ziel des Praktikums

Ziel unseres Softwarepraktikums 'Terraindigitalisierung' war es, ausgehend von einer (digitalen) topographischen Landkarte des Angkor-Tempelareals in Kambodscha ein Höhenfeld (engl. heightmap) zu erstellen und in einem geeigneten Format abzuspeichern. Dieses sollte von Jens Rannachers 'TerrainViewer' unterstützt werden, einer in C++ programmierten Konsolenanwendung zur dreidimensionalen Darstellung digitaler Geländedaten. Die eine 3D-Visualisierung des Angkor-Temperareals auf Grundlage unseres Höhenfeldes ermöglichen werden.

1.2 Motivation

Höhenfelder kommen vor allem als digitale Geländemodelle (DGM) in der 3D-Terrainvisualisierung zum Einsatz. Ein DGM approximert eine Geländeoberfläche ausgehend von einer bestimmte Menge an regelmäßig oder unregelmäßig angeordneten Messpunkten. Gewöhnlich wird die Oberfläche durch eine Triangulierung dieser Messpunkte modelliert, indem man die Geländehöhe innnerhalb der Dreiecke linear aus den Messwerten interpoliert. Die gemessenen und interpolierten Höhenwerte werden dann in Form eines Höhenfeldes abgespeichert. Solche digitalen Landschaftsmodelle werden in vielen Bereichen der Geowissenschaften oder 3D-Computergraphik verwendet: z.B. in der Geodäsie, im Bauwesen, in der Verkehrplanung, bei 3D-Computerspielen, Navigationssoftware, ...

2 Realisierung der Praktikumsziele / Darstellung unserer Vorgehensweise

Ausgangspunkt des Projekts war die topographische Karte des Angkor - Tempelareals in Kambodschia, die Höheninformationen in Form von mehreren Höhenlinien bereitstellte und uns als TIFF Bilddatei vorlag.

Um zunächst die vorhandenen Höheninformationen zu extrahieren, mussten wir ein Bildbearbeitungsprogramm verwenden. Wir haben uns für GIMP entschieden, da dies ein OpenSource-Projekt ist, welches für Linux, Windows und Mac OS X verfügbar ist. Mittels GIMP markierten wir ca. 2500 Pixel bekannter Höhe auf einer über der Karte gelegten Bildebene und erstellten auf diese Weise eine TIFF Bilddatei, die für die markierten Pixel die Höheninformation in Form eines 8 bit Grauwertes abspeichert. Pixel mit

unbekannter Höhe wurden nicht markiert und mit dem RGB-Wert 0xffffff (weiß) versehen. Auf unserer Homepage wird diese Prozedur im Bereich 'Documentation/Creating the input file' genauer erläutert.

Unser C Programm liest diese TIFF Bilddatei mit Hilfe der frei erhältlichen C Bibliothek 'LibTIFF' als Datenarray ein. Um für jeden verbleibenden Pixel einen Höhenwert auf Grundlage der vorhandenen Höheninformation zu interpolieren, werden die markierten Stützpunkte nach Kriterien der Delauny - Triangulation dreiecksvermascht, um so ein unregelmäßiges Dreiecksnetz (engl. Triangulated Irregular Network, TIN) zu erhalten. Zu diesem Zweck benutzten wir Jonatahan Shewchuk's frei erhältliche Konsolenanwendung 'Triangle', welche wir aus unserem eigenen Programm heraus aufrufen. Anschließend lokalisiert unser Programm für jeden Pixel der Eingangsbilddatei das umgebende Dreieck der Delaunay-Triangulation, um dessen Höhenwert aus den Dreickspunkten linaer zu interpolieren. Auf diese Weise erhalten wir ein vollständiges Höhenfeld, das für jeden Pixel der topographischen Karte eine 16 bit Ganzzahl als Höhenwert abspeichert.

Unser Programm unterstützt zwei Ausgabeformate: Das Höhenfeld kann als 16 bit Graustufen TIFF oder im ARC/INFO ASCII Grid Format ausgegeben werden. Das ARC/INFO ASCII Grid kann weiterhin auch auf quadratische Maße erweitert werden, um die Kompatibilität mit Jens Rannachers 'TerrainViewer' zu gewährleisten. TIFF ist ein weitverbreitetes Dateiformat zur Speicherung von Bilddaten. Mit LibTIFF konnten wir auf eine quelloffene einfach zu benutzende C Bibliothek zum Lesen und Schreiben von TIFF-Dateien zurückgreifen. Das ARC/INFO ASCII Grid Format besteht aus einem Header und Höhendaten im ASCII-Format. Der Header liefert Informationen über die geographische Position und Auflösung der Daten. Die Höhendaten selbst bestehen aus einem Gitter von Integerzahlen. Dieses Format wird von Jens Rannachers 'TerrainViewer' unterstützt, mit dessen Hilfe man eine 3D-Darstellung des von unserem Programm generierten Höhenfeldes erhalten kann.

3 Theoretische Grundlagen

3.1 Höhenfeld / digitales Geländemodell

Im Bereich der Computergraphik und Terrainvisualisierung versteht man unter einem Höhenfeld (engl. heightmap) ein zweidimensionales skalares Feld, das die Topologie eines bestimmten geographischen Ortes repräsentiert. Jedes Feldelement speichert die Höhe desjenigen Ortes, dessen geographischen Koordinaten durch die Indizes des Feldelements beschrieben werden. Ge-

wöhnlich werden Höhenfelder als Graustufenbilder visualisiert, wobei schwarz minimale Höhe und weiß maximale Höhe bedeutet.

3.2 Triangulierung

Für eine formale Beschreibung identifiziere man die euklidische Ebene mit dem $\mathbb{R}$-Vektorraum $\mathbb{R}^2$, versehen mit der euklidischen Norm und der hiervon induzierten Topologie. Ferner sei an folgende topologische Grundbegriffe erinnert:

Def. 1: Sei $M \subseteq \mathbb{R}^2$. Dann heißt M konvex, wenn mit je zwei beliebigen Punkten aus M auch deren Verbindungsstrecke ganz in M liegt.

Def. 2: Sei $M \subseteq \mathbb{R}^2$. Die konvexe Hülle conv(M) ist die kleinste konvexe Teilmenge in $\mathbb{R}^2$, die M enthält.

Def. 3: Eine 2D-Punktwolke ist eine endliche Teilmenge $P \subseteq \mathbb{R}^2$.

Gegeben sei nun eine endliche Menge P von Punkten in der Ebene. Dann versteht man unter einer Triangulierung T(P) dieser 2D-Punktwolke P eine Zerlegung der konvexen Hülle conv(P) in sich nicht überdeckende Dreiecke derart, dass deren Vereinigung wieder conv(P) ergibt und die Eckpunkte der Dreiecke genau die Punkte aus P sind. Typischer Einsatzbereich von Triangulierungen ist die approximative Geländedarstellung in 3D-Anwendungen ausgehend von einer gewissen Anzahl an Höhenmesspunkten. Es ist sofort ersichtlich, dass für $|P| > 3$ keine eindeutige Triangulierung der Punktwolke existiert, sofern man keine weiteren Forderungen stellt.

3.3 Delaunay-Triangulierung

Eine spezielle Triangulierung mit zusätzlichen Bedingungen ist die Delaunay - Triangulierung. Sie besitzt einige schöne Eigenschaften, die insbesondere in der Terrainvisualisierung erwünscht sind.

Sei T(P) eine Triangulierung einer 2D-Punktwolke $P \subseteq \mathbb{R}^2$

Def. 1: Eine Dreieckskante aus T(P) mit Endpunkten A,$B \in P$ heißt Delaunaykante, wenn ein Kreis $K \subset \mathbb{R}^2$ existiert, auf dessen Kreislinie A und B liegen, sich ansonsten jedoch kein weiterer Punkt aus P in seinem Inneren befindet.

Def. 2: Ein Dreieck $\triangle$ aus T(P) erfüllt die Umkreisbedingung, wenn sich im Inneren seines Umkreises keine weiteren Punkte aus P befinden.

T(P) heißt dann Delaunay-Triangulierung DT(P), wenn jede Dreieckskante eine Delaunaykante ist. Hierzu ist äquivalent, dass jedes Dreieck der Triangulierung die Umkreisbedingung erfüllt.

Man kann zeigen, dass durch diese Bedingung der minimale Dreieckswinkel unter allen legalen Triangulierungen T(P) maximiert wird. Auf diese Weise werden spitzwinklige Dreiecke vermieden, was in der Computergraphik sehr erwünscht ist, denn hierdurch werden numerische Rundungsfehler minimiert. Desweiteren minimert eine Delaunay-Triangulierung unter allen legalen Triangulierungen T(P) den durchschnittlichen Abstand aller Punkte einer Dreiecksfläche vom nächsten ihrer Eckpunkte. Diese Eigenschaft von DT(P) ist vor allem in der Terrainvisualisierung erwünscht, wenn man Geländeoberflächen durch Triangulierung der Höhenmesspunkte approximieren möchte. Die exakte Höhe ist dann nur für die Messpunkte bekannt, für Punkte innerhalb der Dreiecke wird sie linear aus den Höhen der Eckpunkte interpoliert. Dabei wird der potentielle Interpolationsfehler (also die Abweichung von der tatsächlichen Geländehöhe) umso größer, je größer der Abstand des Punktes vom nöchsten Eckpunkt seines umgebenden Dreiecks ist. Durch oben genannte Eigenschaft von DT(P) wird dieser potentielle Interpolationsfehler minimiert. Algorithmisch ist eine Delaunay-Triangulierung zu einer 2D-Punktwolke $P \subseteq \mathbb{R}^2$ mit $|P| = n$ mit einer Laufzeitkomplexität von $O(n \log(n))$ zu bewerkstelligen, es existieren also effiziente Verfahren. Zwei davon sollen hier kurz vorgestellt werden:

Edge flipping: Ausgangspunkt ist eine beliebige (!) Triangulierung der Punktmenge. Um hieraus eine Delaunay-Triangulierung zu erhalten, wird dann für jeweils zwei Dreiecke, die eine gemeinsame Kante besitzen, überprüft, ob diese Kante die (lokale) Delaunay-Bedingung erfüllt. Ist dies der Fall, so kann einfach das nächste Paar benachbarter Dreiecke überprüft werden. Anderenfalls wird die gemeinsame Kante geflippt (edge flipping), d.h. sie wird durch diejenige Kante ersetzt, die die beiden anderen Dreiecksenden verbindet. Diese neue Kante erfüllt dann die lokale Delaunay-Bedingung. Aufgrund der Lokalität des Flippens ist eine rekursive Nachbesserung der gesamten Netztstruktur nicht notwendig. Der Vorteil dieser Methode ist ihre einfache Implementierung, allerdings ist die Laufzeit von $O(n^2)$ nicht optimal.

Divide and conquer: Hier wird die gegebene Punktmenge P in mehrere kleine Teilmengen zerlegt, welche dann zunächst separat delaunay - trianguliert werden (z.B. mit edge flipping oder rekursiv). Die resultierenden Teiltriangulierungen werden dann jeweils paarweise zusammengefügt durch Triangulierung ihrer Zwischenräume, wodurch größere Teiltriangulierungen entstehen, welche wiederum so lange rekursiv zusammengefügt werden, bis man eine Delaunay-Triangulierung der gesamten Punktmenge P erhalten hat. Die Laufzeitkomplexität beträgt bei dieser Methode $O(nlog(n))$, allerdings

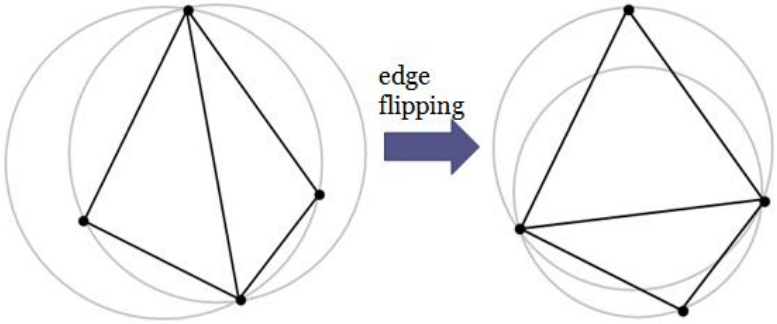

Abbildung 1: edge flipping

ist sie auch anfälliger für numerische Rundungsfehler.

3.4 Punktlokalisation / lineare Höheninterpolation im Dreieck

Gegeben sei ein Dreieck $\triangle(ABC)$ und ein Punkt P in der Ebene. Dann versteht man unter dem Punktlokalisationsproblem die Frage, ob P in der Dreicksfläche liegt oder nicht.

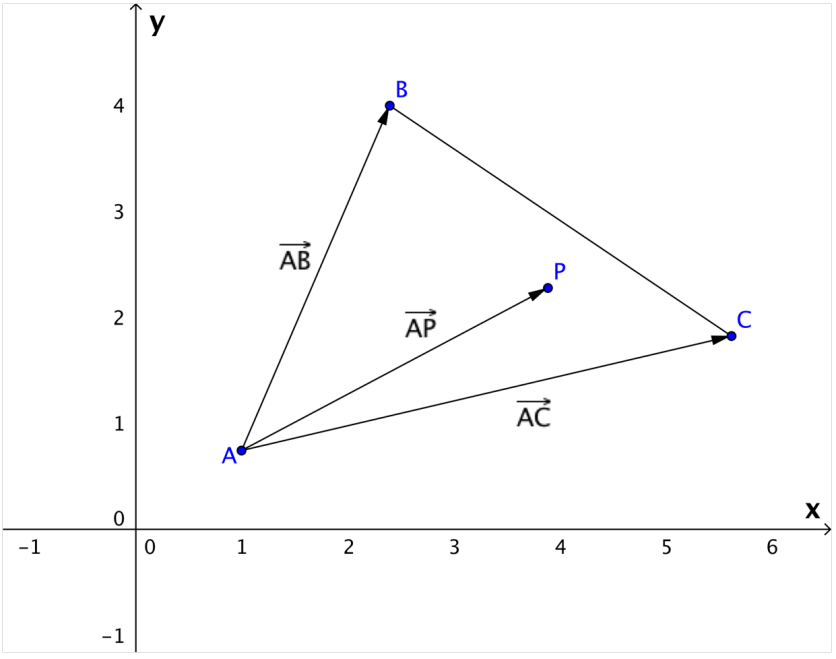

Abbildung 2: Punktlokalisierung

Die Vektoren $\overrightarrow{AB}$ und $\overrightarrow{AC}$ sind linear unabhängig und bilden aus Dimensionsgründen bereits eine Basis des $\mathbb{R}^2$. Es existieren also eindeutig bestimmte Koeffizienten $r, s \in \mathbb{R}$ mit $\overrightarrow{AP} = r\overrightarrow{AB} + s\overrightarrow{AC}$. Mit analytsicher Geometrie ergibt sich folgendes Kriterium: $P \in \triangle(ABC) \Leftrightarrow \overrightarrow{AP} = r\overrightarrow{AB} + s\overrightarrow{AC}$ mit $r, s \geq 0, r + s \leq 1$

Besitzen die Eckpunkte A,B,C des Dreiecks noch eine dritte Kooardinate, die man als Höhe interpretiert, so legen diese im dreidimensionalen Raum eindeutig eine Ebene E_{ABC} fest. Seien also $A = (a_1, a_2, a_3)$, $B = (b_1, b_2, b_3)$, $C = (c_1, c_2, c_3) \in \mathbb{R}^3$ und $pr_{12} : \mathbb{R}^3 \to \mathbb{R}^2, (x_1, x_2, x_3) \mapsto (x_1, x_2)$ die Projektion auf die 1. und 2. Komponente. Setze dann $A' = pr_{12}(A), B' = pr_{12}(B), C' = pr_{12}(C)$. Liegt der Punkt $P \in \mathbb{R}$ dann in $\triangle(A'B'C')$, so kann man in diesem Kontext die Höhe h von P linear im Dreieck interpolieren. Damit meint man, dass dem Punkt P die Höhe desjenigen Punktes aus der Ebene E_{ABC} zugeordnet wird, dessen Projektion auf die 1. und 2 Komponente gerade P ergibt. Mit denselben Koeffizienten wie oben ergibt sich: $h(P) = a_3 + r(b_3 - a_3) + s(c_3 - a_3)$

4 Neue Erkenntnisse

Das Projekt ist im Rahmen eines Anfängerpraktikums entstanden. Hauptziel musste es daher sein, die eigenen Fähigkeiten im Bereich der Programmierung zu erweitern und eigenständig neue Problemstellungen zu bewältigen. Dies wurde im Rahmen dieses Praktiums durchaus erreicht. Neben einigen Problemen, die sich uns im Alltag des Programmierens stellten (z.B. wie man dynamische und statische Bibliotheken linken kann, wie stellt man mehrere hundert Zeilen Code übersichtlich dar und behält den Überblick) wurde im Rahmen des Praktikums auch deutlich sichtbar, dass man sich schon vor Beginn der eigentlichen Programmierung überlegen muss, wie man eine gewisse Programmstelle implementieren möchte. So kann man zeitkritische Sequenzen optimieren und so maßgeblich zur Performance beitragen. Stellt man allerdings erst hinterher fest, dass ein gewisser Teil des Programms sehr viel mehr Zeit in Anspruch nimmt als urspünglich geplant, kann es viel Zeit kosten, das Programm entsprechend umzuschreiben.

Weiterhin mussten wir im Laufe des Praktikums auf einige externe Programme zurückgreifen. Neben der Einarbeitung in das Grafiktool GIMP war es manchmal von Nöten, für bestimmte Programmaufgaben externe Programme ausfindig zu machen. In unserem Fall betrifft das zum einen die Bibliothek LibTIFF, um komfortabel mit TIFFs arbeiten zu können, zum anderen das Programm 'Triangle', um die Delaunay-Triangulation zu erhalten. Beide Programme liegen als OpenSource-Projekte vor und so konnten

wir uns etwas damit auseinandersetzen, die Anleitungen anderer Programme zu verstehen und so die Programme korrekt zu implementieren. Dieser Lernerfolg ist durchaus nicht zu verachten, da man wohl bei allen größeren Projekten auf externe Quellen zurückgreifen muss. Allein das Programm 'Triangle' umfasst mehrere tausend Zeilen Code und wäre wohl von uns nicht zu stemmen gewesen. Diese Phase in unserem Praktikum hat uns auch ein wenig gezeigt, wie interessant und nützlich der OpenSource-Gedanke für Programmierer abseits der Bastion Microsoft sein kann. Es ist ein Geben und Nehmen und befruchtet sich so oft gegenseitig.

5 Erklärung der Funktionsweise der Software (Anleitung)

Dieser Abschnitt der Dokumentation befasst sich mit der Erklärung des Quellcodes. Eine Anleitung, wie die Software zu bedienen ist, findet man dann in englischer Sprache auf unserer Homepage. Der Quellcode selbst ist in 7 Sinnabschnitte (Parts) aufgeteilt, sodass es sinnvoll ist, jeden für sich zu erläutern. Beginn und Ende der Abschnitte sind auch im Quellcode makiert, sodass es leicht nachvollziehbar sein dürfte, über welche Stelle gerade gesprochen wird.

5.1 Part 1: Getting started

Der erste Teil sorgt dafür, dass die für das Programm nötigen Standard - Bibliotheken geladen werden (**Part 1: A**). Weiterhin wird hier eine betriebssystem - spezifische Variable (TRIANGLE_EXECUTE) definiert. Diese ist nötig, um das externe Program Triangle(.exe) korrekt aufzurufen. Das Macro muss beim Kompilieren definiert werden.

Part 1: B deklariert dann noch die Funktionen, welche später benötigt werden. Es handelt sich hier um folgende Funktionen:

```
int cmpxminfirst(const void*, const void*);
inline unsigned short seekIndex(unsigned short*,
     unsigned short*, unsigned long, uint32);
void computeBoundingBox(T_triangle*);
int checkParameter(const char*, int, char**);
```

Die Definitionen dieser Funktionen befinden sich in Part 7. Dort wird auch der Sinn und die Funktionsweise erläutert.

5.2 Part 2: Reading TIFF

Part 2: A umfasst den Teil des Quellcodes, welcher für das Einlesen des TIFF-Files zuständig ist, welches vom Nutzer bereitgestellt werden muss. Zunächst müssen in **Part 2: A** aber alle Variablen deklariert werden, die im Laufe der main()-Funktion benötigt werden. Zur Laufzeitmessung wird außerdem unmittelbar danach die Systemzeit gespeichert.

Part 2: B öffnet nun das zu lesende TIFF-File und prüft einige Bedingungen. Nur wenn diese Bedingungen erfüllt sind, kann das Programm fortfahren. Andernfalls wird eine Fehlermeldung generiert und das Programm bricht ab. Unter anderem wird überprüft, ob die Syntax-Eingabe beim Programmstart korrekt war, ob es sich um ein 8- oder 16-bit Graustufenbild handelt und ob einige TAGS gesetzt sind (z.B. Länge und Höhe des Bildes). Die Einzelheiten entnimmt man dem Quellcode in Verbindung mit der TIFF-Spezifikation.

Part 2: C allokiert nun genügend Speicherplatz, um das Bild in den Arbeitsspeicher zu laden. Dabei wird unterschieden, ob das Eingangs-TIFF mit 8 oder 16 Bit kodiert ist. Bei einer 16-Bit-Kodierung wird das entsprechen Bild einfach in den Speicher geladen. Bei 8 Bits pro Pixel wird das Bild zunächst auf 16 Bit erweitert, indem der entsprechende Byte mit $2^8 + 1$ multipliziert wird. Dies bewirkt, dass der entsprechende Byte kopiert wird und auf 2 Byte erweitert wird. Beispielsweise würde aus dem 8-Bit-Input 10100110 die 16-Bit-Zahl 1010011010100110 generiert werden. Die so erhaltene 16-Bit-Zahl wird nun in den Speicher kopiert, welcher normalerweise dafür vorgesehen ist, ein 16-Bit-Input-Bild zu speichern. Durch diesen Programmschritt ist sichergestellt, dass unser Programm ab jetzt immer mit einem 16-Bit-Input hantieren kann - unabhängig vom Eingangsbild.

Part 2: D ist für die Funktion unseres Programmes nicht von belangen. Es werden lediglich einige Daten über das TIFF-File ausgegeben (z.B. Länge, Breite, Speicherbedarf im RAM), welche den Nutzer interessieren könnten.

5.3 Part 3: Manipulating TIFF-File

Dieser Teil des Codes ist dafür zuständig, einige Korrekturen am Eingangs-TIFF vorzunehmen, um sicherzustellen, dass der spätere Algorithmus korrekt abläuft, ohne dass viele Sonderfälle berücksichtigt werden müssen. **Part 3: A** überprüft zunächst, ob für den Punkt in der oberen linken Ecke des Bildes ein Höhenwert gesetzt wurde. Dieser ist für den Algorithmus - so wie er von uns implementiert wurde - unbedingt notwendig. Daher bricht das Programm ab, wenn hier kein Höhenwert gesetzt wurde. Im Anschluss daran können in einem gewissen Abstand entlang der Bildränder Punkte gesetzt werden. Dadurch wird vermieden, dass am Rand ein einziges großes Dreieck entsteht,

bei dem zwei Ecken des Bildes auch zwei der drei Dreieckspunkte definieren. Ein solch großes Dreieck ist für die Triangulation meist nicht gut und ergibt keine realistisch wirkenden Höhenkarten im Endergebnis. Insbesondere für kleinere Eingangsbilder kann diese Option aber auch negativ sein, weshalb die Option über den Switch '-nopoints' beim Programmaufruf ausgeschaltet werden kann.

Genauso wie zu große Dreiecke zu keinen schönen Ergebnissen führen, verhält es sich auch mit deutlich zu kleinen Dreiecken, die nicht wesentlich größer als einige Pixel sind. Daher suchen wir in **Part 3: B** nach solchen zu eng gestrickten Mustern und löschen im Zweifelsfall zu nah gelegene Punkte. Meist entstehen in realistischen Szenarien solche engen Muster sowieso nicht und sind oft durch fehlerhafte Eingaben vom Benutzer ungewollt verschuldet, beispielsweise durch leichtes Verrutsches der Maus während des Setzens der Pixel mit GIMP. Während der Suche nach solch fehlerhaften Pixeln wird auch gleichzeitig die Gesamtzahl der gesetzten Höhenpunkte gezählt. Diese Zahl wird später benötigt. Außerdem werden in diesem Part auch noch die übrigen Eckpunkte des Bildes mit einem Höhenwert beschrieben, da unser Algorithmus später davon ausgeht, dass jeder Punkt definitiv in einem Dreieck liegt. Dadurch, dass alle 4 Eckpunkte gesetzt werden, ist sichergestellt, dass wirklich jeder Punkt im Innern eines Dreiecks liegt.

Das durch **Part 3: A** und **Part 3: B** korrigierte Eingabe-TIFF kann bei Bedarf in korrigierter Form ausgegeben werden. Ist dies gewünscht, muss das Programm mit dem Switch '-cortif' aufgerufen werden. **Part 3: C** sorgt dann dafür, dass das korrigierte Bild in die Datei 'corrected_input.tif' abgespeichert wird. Alte Versionen dieser Datei werden bei Bedarf ohne Rückfrage überschrieben.

5.4 Part 4: Delaunay-Triangulation

Der Sinn und Zweck der Delaunay-Triangulation wird im entsprechenden Kapitel in dieser Dokumentation erklärt. In diesem Teil wird daher nur die Implementation erläutert. Zur Triangulation selbst haben wird auf ein externes Programm zurückgegriffen, welches während der Programmlaufzeit aufgerufen wird. Dieses Programm 'Triangle' benötigt neben der Anzahl der Punkte, die vermascht werden sollen, auch dessen Koordinaten. In **Part 4: A** werden daher die Koordinaten unserer gesetzten Höhenpunkte einfach in die Textdatei 'vertex.node' geschrieben. Die Syntax innerhalb dieser Datei ist vom Programm 'Triangle' vorgegeben und kann bei Bedarf auf den entsprechenden Seiten näher eingesehen werden. Der Aufbau ist aber im Prinzip sehr simpel: In der ersten Zeile muss die Anzahl der Punkte sowie die Dimension, in der sich das ganze Abspielt, stehen. Im Anschluss folgt die Anzahl

der Attribute. Jeder Punkt kann Attribute enthalten, welche in die Textdatei geschrieben werden, aber vom Programm 'Triangle' nicht berücksichtigt werden. Wir benutzen diese Schnittstelle, um die Höhe der gesetzten Punkte abzuspeichern. Weiterhin muss eine weiter für uns unbedeutende Zahl gesetzt werden, hier 0. In den folgenden Zeilen folgt dann für jeden Punkt die Nummer, die x- und y-Koordinaten sowie die Höhe als Attribut.

Part 4: B ruft nun das externe Programm 'Triangle' auf. Das Programm ist quelloffen und kann bei Bedarf eingesehen werden. Für das weitere Verständnis ist dies aber nicht von Nöten. Man beachte, dass der Programmaufruf betriebssystem-spezifisch stattfindet. Das entsprechende Macro wird beim Kompilieren über einen entsprechenden Switch gesetzt. Das Programm 'Triangle' generiert nun eine Textdatei 'vertex.ele', in der alle Dreiecke stehen, die generiert wurden. Dabei folgt nach einer Header-Zeile für jedes Dreieck die Dreiecksnummer und anschließend die Nummern der drei Eckpunkte.

Diese Datei wird in **Part 4: C** eingelesen und zusammen mit der von uns erstellten Datei 'vertex.node' werden die Dreiecke mit den Koordinaten ihrer Punkte (nicht mit den Nummern der Punkte wie in der Textdatei gegeben!) in das Feld mytriangles geschrieben, wobei mytriangles ein Feld einer von uns bereitgestellten Struktur ist. Diese Stuktur wird in der Quelldatei 'terrain.h' definiert und sieht wie folgt aus:

```
unsigned short x1,x2,x3,y1,y2,y3,z1,z2,z3,
   maxx,maxy,minx,miny;
} T_triangle;
```

Diese Struktur enthält neben den Höhen für die drei Punkte, welche auch aus den Attributen in der Datei vertex.node geschrieben werden, noch Variablen für den größten und kleinsten x- und y-Wert. Diese werden später benötigt. Das setzen dieser vier Werte wird mit dem Funktionsaufruf

```
computeBoundingBox(&mytriangles[i]);
```

erledigt, sobald alle anderen Werte gesetzt wurden.

In **Part 4: D** wird nun noch die Liste der Dreiecke mit dem Sortieralgorithmus Quicksort nach aufsteigendem minx-Wert sortiert. Bei Dreiecken mit selbem xmin-Wert ist der maxx-Wert ausschlaggebend, wobei ein kleinerer maxx-Wert weiter vorne in dem Array steht. Der Sortieralgorithmus Quicksort wird schon von C aus in der Standard-Bibliothek 'stdlib' mitgeliefert. Lediglich die Funktion, welche bestimmt, welches von zwei Argumenten das kleinere bzw. größere ist, muss selbst implementiert werden. Dies erledigt bei uns die Funktion

```
int cmpxminfirst(const void* triangle1, const void* triangle2)
```

, welche in **Part 7: A** definiert wird. Die Sortierung mit dem Quicksort-Verfahren nimmt zwar etwas Zeit in Anspruch, durch die Sortierung kann aber viel Zeit bei der späteren Suche (in welchem Dreieck ist welcher Punkt) gespart werden, sodass sich das Verfahren mehr als lohnt. Laufzeitmessungen brachten einen Laufzeitbeschleunigung um den Faktor 3.

5.5 Part 5: Computing height for each pixel

Dieser Teil ist das Herz des Programms, denn hier wird nun für alle weiteren Punkte, deren Höhe noch nicht gesetzt wurde, eine Höhe berechnet. Zunächst einmal wird in **Part 5: A** eine Index-Datei erstellt. Diese Index-Datei beinhaltet für jeden möglichen minx-Wert den größt-möglichen maxx-Wert. Dieser ist der größte maxx-Wert aus allen Dreiecken mit einem minx-Wert kleiner-gleich dem aktuellen minx-Wert. Außerdem wird für jeden minx-Wert noch eine Sprungadresse gespeichert, ab welcher Dreiecksnummer (also Index in der Liste der Dreiecke) Dreiecke mit einem solchen minx-Wert auftauchen. Man beachte die Ausnutzung der vorherigen Quicksort-Sortierung. Die genaue programmtechnische Implementierung ist schwer in Worte zu fassen. Eine detaillierte Erklärung, wann bei der Index-Erstellung was geschieht, ist direkt im Quellcode einsehbar. Wer neben dem Wissen, was der Index speichert, auch verstehen möchte, wie dies programm-technisch implementiert wurde, kommt nicht umher, die entsprechende Stelle im Code selbst nachzuvollziehen. Dort befindet sich auch eine nähere Erläuterung zur Erstellung des Index.

Part 5: B befasst sich nun mit der Suche nach dem Dreieck für jeden Punkt und der Kalkulation der Höhe. Zunächst einmal zur Suchfunktion: Als erstes wird überprüft, ob der entsprechende Punkt im selben Dreieck liegt wie der vorherige Punkt. Zur Überprüfung, ob dies der Fall ist, wird die Lösung des folgenden Gleichungssystems herangezogen:

```
1: parameter1 * (x2-x1) + parameter2 * (x3-x1) + x1 = spalte
2: parameter1 * (y2-y1) + parameter2 * (y3-y1) + y1 = zeile
```

Dabei bezeichnen x1, x2, x3, y1, y2 und y3 die entsprechenden Koordinaten des Dreieckes und spalte und zeile die Koordinaten des zu untersuchenden Punktes. Aus diesem Gleichungssystem berechnen sich parameter1 und parameter2. Analytische Geometrie liefert dann, dass der Punkt nur im Dreieck liegt, wenn folgende drei Bedingungen erfüllt sind:

```
1: parameter1 >= 0
2: parameter2 >= 0
3: parameter1 + parameter2 <= 1
```

Wegen Rundungseffekten haben wir die dritte Bedingung etwas abgeschwächt und verwenden als Obergrenze der Summe die Zahl 1.000005 statt 1. In vielen Fällen (über 95% in unserem getesteten Fall) ist das zuletzt-verwendete Dreieck bereits das Richtige und die Suche kann beendet werden. Ist dies nicht der Fall, so beginnt ein etwas aufwändigerer Suchalgorithmus: Zunächst kann man durch die Spalte des zu untersuchenden Punktes (welche ja bekannt ist) sagen, dass nur Dreiecke mit maxx >= spalte in Frage kommen. Daher greifen wir auf unseren Index, der in **Part 5: A** generiert wurde, zurück und suchen mit einer binären Suche im Index den minx-Wert heraus, für den der zugehörige maxx-Wert erstmals größer gleich 'spalte' ist. Die binäre Suche wird von der Funktion

```
inline unsigned short seekIndex(unsigned short* indexcount,
   unsigned short* indexmaxx,unsigned long x,uint32 arraySize)
```

erledigt. Der zürckgegebene Index erlaubt uns einen sofortigen Zugriff auf die entsprechende Sprungaddresse in der Liste der Dreiecke, sodass durch diese Ausnutzung der Index-Datei einige Dreiecke am Anfang der Liste der Dreiecke von vornherein ausgeschlossen werden kann. Innerhalb der Dreiecke, die nun noch in Frage kommen, wird zunächst für jedes Dreieck überprüft, ob der Punkt in dem Viereck liegt, welches von den vier Punkten minx, maxx, miny und maxy aufgespannt wird. Diese zusätzliche Überprüfung bringt in der Realität gute Laufzeitvorteile. Für Dreiecke, die diese Bedingung erfüllt ist, wird nun das etwas aufwändigere Gleichungssystem wieder verwendet, welches schon einige Zeilen vorher erklärt wurde.

Ist das entsprechende Dreieck gefunden, so wird die Höhe sehr einfach mittels analytischer Geometrie aus folgender Formel berechnet:

```
Höhe = <Höhe #1> + parameter1 * ( <Höhe #2> - <Höhe #1> ) +
     + parameter2 * (  <Höhe #3> - <Höhe #1>)
```

Dabei bezeichnet <Höhe #i> die Höhe des entsprechenden Dreieck - Eckpunktes. Die so berechnete Höhe wird dann abgespeichert. Nachdem die Höhe für jedes Pixel im Bild berechnet wurde, hat man so eine vollständige Höhenkarte in Form eines Arrays. Dieses wird nun in **Part 7** exportiert.

5.6 Part 6: Generating output

In **Part 6: A** wird die berechnete Höhekarte in Form eines TIFF-Bildes exportiert. Vorherige Versionen der Datei 'heightmap.tif' werden dabei überschrieben. Zunächst werden einige TIFF-spezifische TAGS, anschließend das

Array in die Datei geschrieben. Über die Bibliothek LibTIFF ist dies recht einfach zu handhaben.

Part 6: B exportiert das berechnete Höhenprofil in eine 'ArcASCII Info Grid'-Datei. Neben einem Header über die Angabe von Breite und Weite entspricht dieses Format einfach einer Aufzählung aller Punkte in einer Textdatei, wobei für jeden Punkt eine Höhe angegeben wird. Eine offizielle Homepage zu diesem Format existiert leider nicht, bei Fragen zu diesem Format hilft aber Google ohne weiteres aus. Während in **Part 6: C** einfach unser berechnetes Höhenfeld in ein solches Format exportiert wird, wird in **Part 6: B** die Karte noch auf ein quadratisches Format erweitert. Dies war nötig geworden, weil Hilfsprogramme zur Visualisierung unseres Höhenfeldes nur quadratische ASC-Files lesen können. Die neu erstellten Punkte erhalten die Höhe 0.

5.7 Part 7: function definitions

Dieser Part dient dazu, die deklarierten Hilfsfunktionen zu definieren.

In **Part 7: A** wird zunächst die Funtion cmpxminfirst definiert, welche dazu dient, für die Sortierung unserer Dreiecke mit Hilfe von Quicksort eine entsprechende Funktion bereitzustellen, die entscheidet, welches von zwei Dreiecken als größer bzw. kleiner behandelt werden soll. Die Funktion ist zusammen mit der Erklärung der qsort()-Funktion weitestgehend selbsterklärend.

Part 7: B definiert schließlich eine Funktion seekIndex(). Sie hat folgenden Zweck: Während der Suche, in welchem Dreieck ein gewisser Punkt liegt, sollen mit Hilfe des erstellen Index einige Dreiecke von vornherein ausgeschlossen werden: solche, deren maxx-Wert kleiner ist als die x-Koordinate (Spalte) des zu untersuchenden Punktes. So wie unsere Index-Datei aufgebaut ist, muss lediglich der Index-Wert gesucht werden, bei dem der maxx-Wert erstmals größer oder gleich der x-Koordinate des Punktes ist. Dies geschieht mit einer binären Suche, die in dieser Funktion definiert wird.

Part 7: C definiert die Funktion computeBoundingBox(). Diese berechnet mit Hilfe der Koordinaten eines Dreieckes die Werte minx, maxx, miny und maxy.

Part 7: D ist eine Hilfsfunktion, um zu überprüfen, welche Schalter beim Programmstart gesetzt wurden. Durch diese Funktion erübrigt sich eine genaue Definition der Reihenfolge der Schalter. Sie trägt damit zur Benutzerfreundlichkeit bei.

6 Ausblick

Das Programm wurde im Rahmen eines Anfänger-Softwarepraktikums erstellt und ist sicherlich nicht bis ins letzte ausgereift. Primäres Ziel war es, die uns gestellte Aufgabe ohne größere Umschweife zu erfüllen. Das Programm ist daher recht spezifisch und bietet Möglichkeiten zur Ergänzung.

Eine mögliche Erweiterung besteht darin, weitere Ausgabeformate zu unterstützen. Bildformate wie JPEG für eine Darstellung im Web oder ähnliches könnten ohne weitere Probleme implementiert werden, waren aber für uns nicht von Relevanz. Weitere Ausgabeformate - wie z.B. die Kodierung in Farben statt in Graustufen - sind denkbar und könnten benutzerspezifisch implementiert werden.

Weiterhin besteht die Möglichkeit, den externen Programmaufruf 'Triangle' zu entfernen und den Quellcode dieses Programms direkt in den unseren zu implementieren. Dies könnte nötig sein, um die Kompatibiltät im Programmaufruf für weitere Betriebssysteme zu ermöglichen. Da der Quellcode sowie ein Beispiel zur Implementation des Programms 'Triangle' im Web verfügbar sind, kann dies bei Bedarf erledigt werden. Im Rahmen unseres Praktikums war dies aber nicht von Nöten.

Der größte Aufwand zur Erstellung eines Höhenfeldes mit unserem Programm ist es, das richtige Eingabebild manuell zu erstellen. Im Zuge der Fortschritte im Bereich der Bilderkennungs-Software könnte man unser Programm diesbezüglich erweitern, um dies den Computer erledigen zu lassen. Damit wäre es dann möglich, in kürzester Zeit für eine große Anzahl von Geländen ein Höhenfeld zu generieren. Die Implementation dieses Schrittes dürfte aber nicht ganz einfach sein und würde den Rahmen unseres Projekts sprengen.

Literatur

[1] Homepage zu diesem Projekt:

`http://pille.iwr.uni-heidelberg.de/~terrain03/`

[2] Homepage zu Jens Rannachers 'TerrainViewer':

`http://pille.iwr.uni-heidelberg.de/~terrain02/`

[3] Homepage zu Jonathan Shewchuks 'Triangle':

`http://www.cs.cmu.edu/~quake/triangle.html`

[4] Homepage zu 'LibTIFF':

http://www.remotesensing.org/libtiff/

[5] Homepage zur Delaunay-Triangulation (Wikipedia):
http://en.wikipedia.org/wiki/Delaunay_triangulation

[6] Homepage zu Heightmaps (Wikipedia):
http://en.wikipedia.org/wiki/Heightmap

[7] Script zu Michael Wincklers Proseminar Computergraphik:
http://www.iwr.uni-heidelberg.de/groups/ngg/Txt/Proseminar06.pdf.gz
(gefunden hier: http://www.iwr.uni-heidelberg.de/groups/ngg/services.php?L=E)