Funktionales Programmieren mit Polygonen [1]

(mit 6 Bildern)

Von Karl Neumann, Braunschweig

ZUSAMMENFASSUNG: Die funktionale Programmiersprache Scheme eignet sich gut, um Operationen auf Polygonen zu realisieren. Wir zeigen dazu, wie Polygone als Listen von Koordinaten gehandhabt werden können und wie direkt sich typische Funktionen, etwa das Douglas/Peucker-Verfahren, funktional implementieren lassen. Diese grundlegenden Operationen werden dann auch in realistischen Anwendungen eingesetzt.

ABSTRACT: The functional programming language Scheme is well suited for implementing polygonal algorithms. For that purpose we demonstrate how polygons can be handled as lists of coordinates. Also we present the straight and elegant functional implementation of typical polygonal operations such as the well-known Douglas/Peucker algorithm. Finally, these operations are used within some realistic applications.

1 Einleitung

Obwohl imperative Programmiersprachen wie Fortran, Pascal, C oder Java sehr viel häufiger eingesetzt werden als Programmiersprachen mit anderen Grundkonzepten, kann die Verwendung von deduktiven, prädikativen oder funktionalen Sprachen auch vorteilhaft sein. So ist z.B. bei den funktionalen Programmiersprachen die Trennung zwischen einem ausführbaren Programm und der präzisen Spezifikation, welches Problem das jeweilige Programm eigentlich lösen soll, annähernd aufgehoben. Im Hinblick auf eine Anwendung kann daher hier die Spezifikation und nicht deren Auffassung als Programm im Vordergrund stehen. Da bei den funktionalen Programmiersprachen zudem oft statt eines Übersetzers, wie bei den imperativen Sprachen fast durchweg üblich, ein Interpreter eingesetzt wird, können komplexere Funktionen sehr komfortabel und annähernd interaktiv durch Kombination von einfacheren Funktionen konstruiert werden.

Im vorliegenden Text soll gezeigt werden, dass sich funktionales Programmieren auch bei der Implementierung von Funktionen für Anwendungen mit Raumbezug gut einsetzen lässt. Dazu gehen wir im nächsten Abschnitt kurz auf die Prinzipien der funktionalen Programmiersprachen im Allgemeinen ein und stellen dann einige Besonderheiten der im Weiteren benutzten Sprache Scheme vor. Anschließend wird gezeigt, wie Polygone in Scheme als Listen von Koordinaten dargestellt und gehandhabt werden können und wie einfach einige typische grundlegende Polygon-Funktionen mit Scheme funktional zu implementieren sind, etwa das Filtern von Polygonen oder die Polygon-Vereinfachung

[1] Erscheint in *Mitteilungen des Bundesamtes für Kartographie und Geodäsie, Frankfurt M., 2002.*

durch das Verfahren nach Douglas/Peucker. Im vorletzten Abschnitt werden diese einfacheren Funktionen in realistischen Anwendungen eingesetzt, und zum Schluss fassen wir unsere Erfahrungen beim funktionalen Programmieren mit Polygonen nochmals kurz zusammen.

2 Funktionale Programmiersprachen

Den verschiedenen Klassen der Programmiersprachen liegen auch jeweils unterschiedliche Denkmuster zu Grunde (vgl. etwa *Watt* 1990). So spiegeln die überaus verbreiteten imperativen Sprachen die Struktur des Von-Neumann-Rechners wider: Ein Programm besteht aus einer Folge von Befehlen, die vom Rechner abzuarbeiten sind. Wesentlich bei diesen Sprachen ist das Konzept der Variablen. Eingabewerte werden in Variablen (benannten Speicherzellen) gespeichert, durch Befehle verarbeitet und wieder in Variablen abgelegt.

Dagegen steht bei den funktionalen Sprachen der Begriff der Funktion im Vordergrund. Funktionen werden durch mathematische Ausdrücke beschrieben und bilden Eingabedaten in Ausgabedaten ab. Dabei gibt es elementare Ausdrücke für einfache Funktionen und Operationen, die auf Funktionen definiert sind, mit denen komplexere Funktionen aus einfachen konstruiert werden können. Ein Programm besteht dann aus einer Menge von Ausdrücken, die Funktionen definieren, und eine Berechnung ist die Anwendung (Applikation) einer Funktion auf eine Liste von Werten oder auch Ausdrücken. Funktionale Programmiersprachen werden daher auch applikativ genannt. Als Beispiel einer schon etwas komplizierteren Funktion hier die bekannte Definition der Berechnung des Abstandes d eines Punktes (x_p,y_p) von einer Geraden, die durch die Punkte (x_a,y_a) und (x_b,y_b) gegeben ist:

$$d = \sqrt{\frac{(((y_p - y_a)\cdot(x_b - x_a)) - ((x_p - x_a)\cdot(y_b - y_a)))^2}{(x_b - x_a)^2 + (y_b - y_a)^2}}$$

In diesem Beispiel werden als elementare Ausdrücke die Funktionen Addition, Subtraktion, Multiplikation, Division, Quadrat sowie Quadratwurzel benutzt, und es wird außerdem die Schachtelung von Funktionen angewandt. Die auftretenden Variablen sind hier keine Behälter, denen Werte zugewiesen werden, sondern Platzhalter für Werte, mit denen symbolisch gerechnet wird. Formale Grundlage der funktionalen Programmiersprachen ist das Lambda-Kalkül, ein Formalismus, der zur präzisen Definition des Begriffs der Berechenbarkeit entwickelt wurde. Dagegen liegen den imperativen Sprachen Automatenmodelle zu Grunde. Beide Formalismen oder Modelle sind in ihrer Berechnungsfähigkeit aber gleichmächtig. Weiteres zur Fundierung der funktionalen Sprachen kann *Reade* 1989 und *Huet* 1990 entnommen werden; zu Grundlagen von Programmiersprachen allgemein vgl. *Wilhelm/Maurer* 1997. Funktionale Programmiersprachen werden meist nicht durch Übersetzer sondern durch Interpreter realisiert. Die einzelnen elementaren Ausdrücke und Definitionen von Funktionen können also sofort nach dem

Eintippen jeweils syntaktisch überprüft und auch ausgewertet werden. Das erleichtert das Testen von komplexeren Funktionen und führt so relativ schnell zu ersten lauffähigen Lösungen.

3 Scheme

Die im Weiteren verwendete Sprache Scheme ist eine Variante von LISP (list processing language), einem Standardbeispiel für funktionale Programmiersprachen. LISP wurde bereits in den 50er Jahren entwickelt, und Ende der 70er Jahre kamen sogar auf diese Sprache spezialisierte Rechner auf den Markt, so genannte LISP-Maschinen. Mittlerweile hat aber besonders im Ausbildungsbereich Scheme eine wohl weitere Verbreitung gefunden (vgl. z.B. die Lehrbücher *Eisenberg/Abelson* 1990, *Ferguson et al.* 1995, *Dybvig* 1996, *Pearce* 1998, *Grillmeyer* 1999, *Harvey/Wright* 1999), und es werden für zahlreiche Betriebssysteme freie Scheme-Interpreter angeboten (*Kelsey et al.* 2001).

Das Grundelement in Scheme sind Ausdrücke der Form (X_0 X_1 X_2 ... X_n), wobei X_0 ein Operator ist und X_1 bis X_n die Operanden sind, d.h. alle Ausdrücke werden in Präfixschreibweise notiert. Die Operanden können ihrerseits wieder Ausdrücke sein. Wenn man z.B. folgenden Ausdruck der Dialogschnittstelle eines Scheme-Interpreters eingibt

```
(/ 40030 (* 2 6371))
```

antwortet das System mit der Zahl `3.1415790299796`. Der Teilausdruck X_0 bezeichnet hier also den Divisionsoperator `/`, und X_1 ist die Zahl `40030`. Der Operand X_2 ist selbst wieder ein Ausdruck mit dem Multiplikationsoperator `*` als Komponente $X_{2.0}$ und den Zahlen `2` sowie `6371` als weitere Operanden $X_{2.1}$ und $X_{2.2}$. Durch weitere Schachtelung können beliebig komplexe baumartige Ausdrücke konstruiert werden, die immer dann, wenn sie vollständig eingegeben worden sind, vom System ausgewertet werden. Um nicht nur arithmetische Berechnungen mit Konstanten durchführen zu können, werden auch Variablen und Parameter benötigt. Mit dem Operator `define` können dazu Ausdrücke benannt werden. So wird etwa mit

```
(define pi 3.1415927)
```

die Zahl `3.1415927` mit `pi` benannt, und der Bezeichner `pi` könnte nun als elementarer Ausdruck in anderen Ausdrücken benutzt werden. Ebenso ist es möglich, Ausdrücke als Funktionen zu definieren. Dazu muss jedoch deutlich gemacht werden, dass der 2. Operand der Define-Funktion – im obigen Beispiel ist das eine Zahlenkonstante – eine Funktion ist, die eventuell Parameter hat. Für diesen Zweck ist in Scheme der Lambda-Operator vorgesehen. Die beiden folgenden Beispiele für Funktionsdefinitionen zeigen den Einsatz des Lambda-Operators.

```
(define quadrat (lambda (x) (* x x)))

(define punkt-abstand (lambda (p1 p2)
  (sqrt (+ (quadrat (- (x-koord p2) (x-koord p1)))
           (quadrat (- (y-koord p2) (y-koord p1)))))))
```

Es wird zunächst die sehr einfache Funktion `quadrat` definiert, die einen Parameter hat und diesen mit sich selbst multipliziert als Ergebnis zurückgibt. Die Funktion besteht nur darin, dass bei einem Aufruf ein aktueller Wert an den Parameter `x` gebunden wird und mit dieser Bindung der Ausdruck `(* x x)` ausgewertet wird. Wie gewohnt ist die Benennung der Parameter ohne Bedeutung für die jeweilige Funktion, d.h. im obigen Fall könnte der Parameter `x` einen beliebigen anderen Namen haben, der dann allerdings an allen drei Positionen identisch sein müsste. Man kann an diesem einfachen Beispiel auch erkennen, dass es in Scheme keine Typangabe bei Parametern gibt: Wird die Funktion `quadrat` nicht mit numerischen Argumenten aufgerufen, so ergibt die Anwendung des Multiplikationsoperators einen Laufzeit-Typfehler.

Das zweite Beispiel, die Funktion `punkt-abstand`, benutzt die vom System zur Verfügung gestellten Operationen Quadratwurzel (`sqrt`), Addition, Subtraktion sowie die oben implementierte Quadratfunktion. Außerdem wird mit Hilfe der Funktionen `x-koord` und `y-koord`, die weiter unten noch vorgestellt werden, auf die X- bzw. Y-Koordinaten zweier Punkte zugegriffen. Als Ergebnis liefert die Funktion den Abstand dieser als Parameter angegebenen Punkte `p1` und `p2`. Punkte sind keine elementaren Objekte, wir haben hier bereits die Grunddatenstruktur von Scheme eingesetzt, die Paare.

Ein Paar besteht aus einem linken und einem rechten Element, wobei diese Elemente auch wieder Paare sein können. Konstruiert wird ein Paar mit den Elementen `a` und `b` durch den Ausdruck `(cons a b)`, und mit den Funktionen `car` sowie `cdr` kann das linke bzw. rechte Element eines Paares wieder selektiert werden. Die etwas ungewöhnlichen Namen dieser Selektionsfunktionen wurden seit den ersten LISP-Implementierungen beibehalten: Damals wurden Listenelemente durch spezielle Registerbefehle selektiert, und `car` bzw. `cdr` waren Kurzformen für "contents of address register" bzw. "contents of decrement register" (*Abelson et al.* 1996).

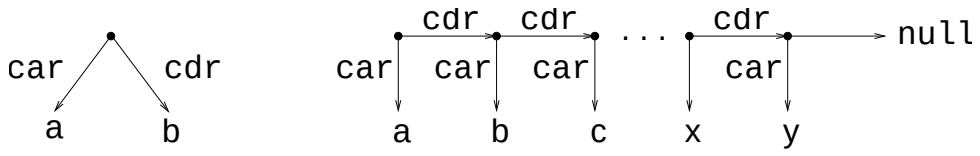

Bild 1 - Konstruktion von Paaren und Listen

In Bild 1 ist ein einzelnes Paar dargestellt und auch die Struktur einer Liste, die dadurch gebildet wird, indem als rechtes Element eines Paares immer wieder ein Paar auftritt, wogegen das linke Element stets ein elementares Objekt ist. Als letztes Element der

Liste – gleichzeitig als rechtes Element des innersten Paares – tritt ein spezielles leeres Element auf (`null`), das so das Listenende anzeigt. Durch andere Schachtelungen bei der Paarbildung können beliebige binäre Bäume konstruiert werden, nicht nur Listen. Für uns ist im Folgenden die Listenstruktur jedoch ausreichend.

4 Polygone in Scheme

Mit den Paaren und Listen lassen sich nun leicht Punkte und Polygone darstellen. Die Funktion zum Konstruieren eines Punktes (`konstr-punkt`) ist einfach eine Umbenennung der Funktion zum Paarbilden, und die weiter oben bereits benutzten Zugriffe auf die einzelnen Koordinaten (`x-koord`, `y-koord`) sind Umbenennungen der Funktionen `car` und `cdr`:

```
(define konstr-punkt (lambda (x y) (cons x y)))

(define x-koord (lambda (xy) (car xy)))
(define y-koord (lambda (xy) (cdr xy)))
```

Polygone realisieren wir durch Listen von Koordinaten, wobei der erste Punkt eines Polygons sowohl als erster Punkt in der entsprechenden Liste vorkommt als auch noch einmal als letzter Punkt. Das in Bild 2 dargestellte Quadrat wird somit durch folgenden geschachtelten Ausdruck als Liste konstruiert, wobei in diesem Beispiel ein vereinfachtes ganzzahliges Koordinatensystem zu Grunde gelegt wird:

```
(define quad (cons 4 (cons 3 (cons 10 (cons 3 (cons 10
             (cons 9 (cons 4 (cons 9 (cons 4  (cons 3 ()))))))))))
```

Weil in Scheme recht häufig längere Listen benötigt werden, gibt es eine abkürzende Schreibweise, die es erlaubt, alle Listenelemente ohne geschachtelte Operatoren und Klammern zu notieren. Der Ausdruck zur Konstruktion des quadratischen Polygons `quad` vereinfacht sich damit wie folgt: `(define quad '(4 3 10 3 10 9 4 9 4 3))`.

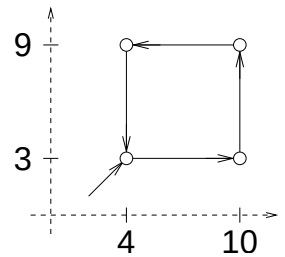

Bild 2 - Polygon mit Orientierung der Punkte in einer Liste

Polygone können nicht immer direkt durch Auflisten ihrer Koordinatenwerte angegeben werden, denn oft ergeben sich Polygone als Ergebnis einer Funktionsanwendung auf ein anderes Polygon, etwa bei der Formvereinfachung durch Anwendung von Filterfunktionen. In diesen Fällen werden Operationen zum Aufbau von Polygonen aus Punkten benötigt und andere Funktionen, mit denen man auf einzelne Punkte zugreifen kann. Beim Aufbau von Polygonen werden einfach nacheinander die einzelnen Punkte jeweils hinten an eine entsprechende Liste angefügt, es wird also zunächst nur eine Funktion `ad-punkt` benötigt, die die beiden Koordinaten eines gegebenen Punktes an eine Liste anhängt:

```
(define ad-punkt (lambda (poly punkt)
  (if (null? poly) (list (x-koord punkt) (y-koord punkt))
                   (cons (car poly) (ad-punkt (cdr poly) punkt)))))
```

Die Funktion `ad-punkt` erwartet als Parameter ein Polygon (eine Liste von Koordinaten) und einen Punkt (ein Paar von Koordinaten). Falls das Polygon leer ist, wird einfach eine Liste zurückgegeben, die aus der X- und der Y-Koordinate des Eingabepunktes besteht. Hier wird die in Scheme vorhandene If-Anweisung eingesetzt, die abhängig vom Wert eines booleschen Arguments einen von zwei folgenden Ausdrücken auswertet und dieses Ergebnis zurückliefert. Das boolesche Argument wird in diesem Fall von der Funktion `null?` berechnet, die ermittelt, ob eine gegebene List leer ist oder nicht. Falls die Liste (das Polygon) nicht leer ist, wird aus dem ersten Element und der Liste, die dadurch ensteht, dass die Funktion `ad-punkt` rekursiv auf die um das erste Element gekürzte Liste angewendet wird, ein Paar gebildet und dieses als Ergebnis geliefert. Damit wird der Aufruf von `ad-punkt` so lange im Polygon nach rechts geschoben, bis am dann leeren Ende die Koordinaten des gegebenen Punktes angehängt werden können. Diese Art der rekursiven Lösung ist typisch für Scheme und für funktionale Programmiersprachen allgemein.

Der Zugriff auf den ersten Punkt eines Polygons (`punkt1`) ist recht einfach zu realisieren: Hier braucht nur ein Punkt aus den ersten beiden Elementen der entsprechenden Liste konstruiert zu werden, also aus dem ersten Element und dem ersten Element der um das erste Element gekürzten Liste. Oft wird auch eine Funktion benötigt, die ein Polygon um den ersten Punkt kürzt (`trunc-poly`). Dazu wird muss lediglich vom Rest einer Liste der Rest geliefert werden, denn damit fehlen die ersten beiden Koordinaten. Schließlich soll als Grundoperation für Polygone noch der Zugriff auf den zweiten Punkt eines Polygons bereitgestellt werden (`punkt2`), was sich besonders elegant verwirklichen lässt, da durch die bereits implementierten Funktionen `punkt1` und `trunc-poly` einfach auf den ersten Punkt des um einen Punkt gekürzten Polygons zugegriffen werden kann.

```
(define punkt1 (lambda (poly)
  (konstr-punkt (car poly) (car (cdr poly)))))

(define trunc-poly (lambda (poly) (cdr (cdr poly))))
```

```
(define punkt2 (lambda (poly) (punkt1 (trunc-poly poly))))
```

Damit haben wir skizziert, wie Polygone in Scheme als Listen realisiert werden können, und es wurden vier recht einfache Grundfunktionen für solche "Polygonlisten" diskutiert. Schon bei diesen Grundfunktionen konnten bereits implementierte Funktionen als Bausteine wieder verwendet werden, einer der Vorteile des funktionalen Programmierens.

5 Funktionen auf Polygonen

In diesem Abschnitt stellen wir die Implementierung anspruchsvollerer Funktionen vor, nämlich das Berechnen des Umfangs, eine einfache Tiefpassfilterung und die Formvereinfachung nach Douglas/Peucker. Die Umfangsermittlung (`umfang`) kann durch Wiederverwendung der Funktionen `trunc-poly`, `punkt1`, `punkt2` (aus dem vorigen Abschnitt) und `punkt-abstand` (aus Abschnitt 3) sehr kompakt formuliert werden. Der Umfang eines Polygons berechnet sich durch Addition des Abstands der ersten beiden Punkte und des Umfangs des um den ersten Punkt gekürzten Polygons. Hier wird die Umfang-Funktion solange rekursiv aufgerufen, bis der letzte Punkt erreicht wird. Da wir den ersten Punkt eines Polygons als ersten und auch als letzten Punkt der entsprechenden Liste führen, wird die Umfang-Funktion bei n Polygonpunkten n-mal den Abstand zweier benachbarter Punkte zurückliefern und das letzte Mal eine Null. Alle diese Werte werden dann aufsummiert und liefern so den Umfang.

```
(define umfang (lambda (poly)
  (if (not (null? (trunc-poly poly)))
      (+ (punkt-abstand (punkt1 poly) (punkt2 poly))
         (umfang (trunc-poly poly)))
      0)))
```

Um Linienzüge zu glätten, können so genannte digitale Tiefpassfilter eingesetzt werden, und eine einfache Methode der Tiefpassfilterung ist die fortlaufende Mittelwertbildung über eine feste Anzahl von Originalstützpunkten. Im eindimensionalen Fall kann z.B. immer das Mittel von mindestens zwei benachbarten Werten einen neuen gefilterten Wert ergeben. Dieses Verfahren lässt sich leicht auf zweidimensionale Polygone erweitern, indem die Mittelwertbildung getrennt auf die X- und Y-Koordinaten angewendet wird. Bei einer solchen Filterung verändert sich die Anzahl der Stützpunkte also nicht, da immer zwei benachbarte Punkte durch ihren Mittelpunkt ersetzt werden. In Bild 3 ist ein Polygon mit 25 Stützpunkten zu sehen (gestrichelte Linie) und dessen gefilterte Version (durchgezogene Linie).

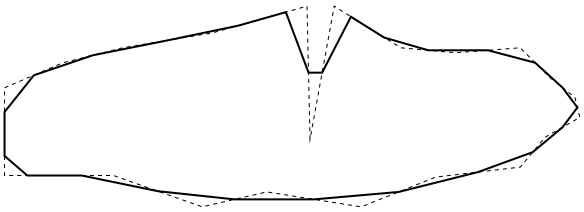

Bild 3 - mit einfachem Tiefpass gefiltertes Polygon

Die Funktion `tiefpass` implementiert das skizzierte Filterverfahren. Die beiden dabei benutzten Operationen `mittelwert`, zur Berechnung des Mittelwerts zweier Zahlen, und `mittelpunkt`, zur Ermittlung des Mittelpunktes zweier Punkte, sind hier nur der Vollständigkeit halber mit angegeben, denn sie bieten keine neuen Aspekte und sollten selbsterklärend sein; sie zeigen aber wieder den modularen Charakter des funktionalen Programmierens.

```
(define mittelwert (lambda (x y) (/ (+ x y) 2)))

(define mittelpunkt (lambda (p1 p2)
  (konstr-punkt (mittelwert (x-koord p1) (x-koord p2))
                (mittelwert (y-koord p1) (y-koord p2)))))

(define tiefpass (lambda (poly)
  (define lok-tiefpass (lambda (poly)
    (if (not (null? (trunc-poly poly)))
        (ad-punkt (lok-tiefpass (trunc-poly poly))
                  (mittelpunkt (punkt1 poly) (punkt2 poly)))
        ())))
  (ad-punkt (lok-tiefpass poly) (punkt1 (lok-tiefpass poly)))))
```

Die eigentliche Tiefpassberechnung wird von der lokalen Funktion `lok-tiefpass` vorgenommen, deren rekursiver Aufbau sehr stark der Umfangfunktion ähnelt. Jeweils vom ersten und zweiten Punkt des Eingabepolygons wird der Mittelpunkt berechnet, und dieser Punkt wird an die Polygonliste angehängt, die sich daraus ergibt, dass die lokale Tiefpassfunktion auf die um den ersten Punkt gekürzte Liste rekursiv angewendet wird. Wenn das Originalpolygon abgearbeitet ist, wird daher an eine leere Liste zuerst der inzwischen ausgerechnete letzte Mittelpunkt angehängt, an diese Liste mit einem Punkt dann der vorletzte Mittelpunkt usw. Das aus der Filterung resultierende Polygon hat deshalb eine dem Originalpolygon entgegengesetzte "Drehrichtung". Die äußere Funktion `tiefpass` ruft die lokale Funktion `lok-tiefpass` zunächst auf, dadurch wird wie beschrieben eine Liste mit den Mittelpunkten zurückgeliefert. Diese Liste muss nun noch durch den letzten Punkt, der ja identisch zum ersten ist, vervollständigt werden. Deshalb wird der erste Punkt der ermittelten Liste an diese Liste angehängt und das jetzt vollständige Resultatspolygon kann als Gesamtergebnis der Funktion `tiefpass` geliefert werden.

Als letzte komplexere Funktion soll das Verfahren von *Douglas/Peucker* 1973 als Scheme-Implementierung skizziert werden. Der Douglas/Peucker-Algorithmus ist ein bekanntes Verfahren (*Hake/Grünreich* 1994), das die charakteristische Form offener, unregelmäßiger Polygone trotz Reduzierung der Stützpunktanzahl recht gut bewahrt. In *Neumann/Selke* 2001 haben wir den Original-Algorithmus und auch eine Modifikation detailliert erläutert, deshalb soll hier eine Kurzbeschreibung genügen. Das Verfahren geht als gröbste Näherung eines Linienzugs von der Gerade zwischen seinem Anfangs- und Endpunkt aus. Dann werden die Abstände aller Stützpunkte zu dieser Geraden betrachtet. Wenn der maximale dieser Abstände größer ist als ein vorgegebener Grenzwert d, bleibt der zugehörige Punkt als signifikant erhalten, und das Verfahren wird rekursiv für die beiden sich ergebenden Teillinienzüge angewendet. Falls der maximale Abstand kleiner als der Grenzwert d ist, werden alle Punkte zwischen Anfangs- und Endpunkt entfernt.

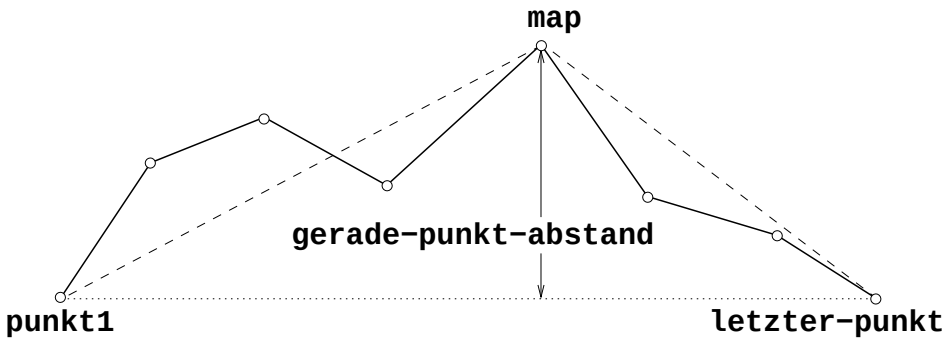

Bild 4 - Ermittlung des Punktes mit dem größten Abstand

Die Funktion `douglas-peucker` implementiert diesen Original-Algorithmus, sie erwartet daher als ersten Parameter kein geschlossenes Polygon sondern einen Linienzug (offenes Polygon), d.h. bei der entsprechenden Polygonliste ist jetzt der erste Punkt nicht mit dem letzten identisch. Der zweite Parameter der Funktion ist der vorgegebene Grenzwert (oder Abstand) d. Falls der Linienzug weniger als 3 Punkte hat, wird er nicht modifiziert sondern unverändert als Ergebnis ausgegeben. Andernfalls, das offene Polygon enthält also mindestens 3 Punkte, wird zunächst der Punkt des Polygons mit dem größten Abstand zur Basislinie ermittelt (vgl. Bild 4). Dieser Punkt wird `map` genannt, und die Funktion `max-abstand-punkt` führt diese Berechnung aus. Wir verzichten aus Platzgründen darauf, die Scheme-Implementierung dieser Funktion zu erläutern. Das gilt auch für die weiteren Hilfsfunktionen `anzahl-punkte`, `gerade-punkt-abstand`, `letzter-punkt`, `teil1` und `teil2`. Alle diese Funktionen sind nicht komplexer als die Hauptfunktion `douglas-peucker`, z.B. ist die Funktion `gerade-punkt-abstand` eine direkte Umsetzung der Formel aus Abschnitt 2 in die Scheme-Präfixschreibweise.

```
(define douglas-peucker (lambda (poly d)
  (if (> (anzahl-punkte poly) 2)
      (sequence
        (define map (max-abstand-punkt poly))
```

```
      (if (> (gerade-punkt-abstand
               (punkt1 poly) (letzter-punkt poly) map)
             d)
          (concat-poly (douglas-peucker (teil1 poly map) d)
                       (douglas-peucker (teil2 poly map) d))
          (ad-punkt (ad-punkt () (punkt1 poly))
                    (letzter-punkt poly))))
    poly)))
```

Nachdem der Punkt mit maximalem Abstand bestimmt ist, wird überprüft, ob der Abstand dieses Punktes zur Basislinie größer ist als der vorgegebene Grenzwert `d`. Die Basislinie ist dabei die Gerade durch den ersten Punkt (`punkt1`) und den letzten Punkt (`letzter-punkt`) des Ausgangspolygons (vgl. auch Bild 4). Falls der Abstand größer ist als `d`, wird das Polygon am Punkt `map` in zwei Teilpolygone aufgespalten, und die Funktion `douglas-peucker` wird rekursiv für diese Teilpolygone mit unverändertem Parameter `d` aufgerufen. Falls der Abstand kleiner ist, wird die Liste aus erstem Punkt und letztem Punkt als Resultatspolygon zurückgegeben, da ja alle anderen Punkte, die dazwischen liegen, nicht signifikant sind. Die Funktion `concat-poly` sorgt dafür, dass aus den beiden formvereinfachten Teilpolygonen, die durch die rekursiven Aufrufe von `douglas-peucker` zurückgegeben werden, wieder ein Resultatspolygon wird: An die erste Polygonliste wird einfach die zweite Polygonliste ohne ihren ersten Punkt angehängt.

```
(define concat-poly (lambda (poly1 poly2)
  (define lok-concat-poly (lambda (poly1 poly2)
    (if (not (null? poly2))
        (lok-concat-poly
          (ad-punkt poly1 (punkt1 poly2))
          (trunc-poly poly2))
        poly1)))
    (lok-concat-poly poly1 (trunc-poly poly2))))
```

Die Funktion `douglas-peucker` zeigt, wie direkt sich ein ohnehin rekursiv formulierter Algorithmus in ein funktionales Programm umsetzen lässt: Im Prinzip wurde die rekursive Definition einfach in Scheme kodiert, wobei die benötigten Hilfsfunktionen dann im nächsten Schritt implementiert wurden. Mit relativ wenig Aufwand lässt sich die angegebene Funktion auch auf geschlossene Polygone erweitern: Man spaltet das gegebene Polygon an den zwei Punkten auf, die den maximalen Abstand voneinander haben, und wendet die Funktion `douglas-peucker` auf die beiden sich ergebenden Teillinienzüge an. Die erhaltenen zwei Ergebnislinienzüge müssen dann zum Schluss nur wieder zusammengesetzt werden, wozu erneut die Funktion `concat-poly` benutzt wird.

6 Beispiel-Anwendungen

Neben den bislang skizzierten Funktionen, inklusive des Douglas/Peucker-Verfahrens für geschlossene Polygone (`poly-douglas-peucker`) haben wir noch weitere Operationen implementiert, mit denen u.a. Polygone verschoben, gedreht und skaliert werden können. Außerdem entstand – ebenfalls mit vergleichsweise geringem Aufwand – eine Sammlung von Funktionen zur Ein- und Ausgabe von Polygonen. Als Dateiformat wurde dabei dasjenige gewählt, das dem im Unix-Bereich weit verbreiteten vektorbasierten Zeichenprogramm XFig (*Sato/Smith* 2001) zu Grunde liegt. Mit diesen Funktionen können Polygone im XFig-Format gelesen, dann als Polygonlisten bearbeitet und schließlich wieder als Linienzüge oder auch in Form einzelner Stützpunkte in Dateien zurückgeschrieben werden. So war es möglich, die implementierten Scheme-Funktionen an Hand realistischer Polygondaten, die aus einem früheren Projekt (*Neumann et al.* 1992) stammen, umfassend zu testen.

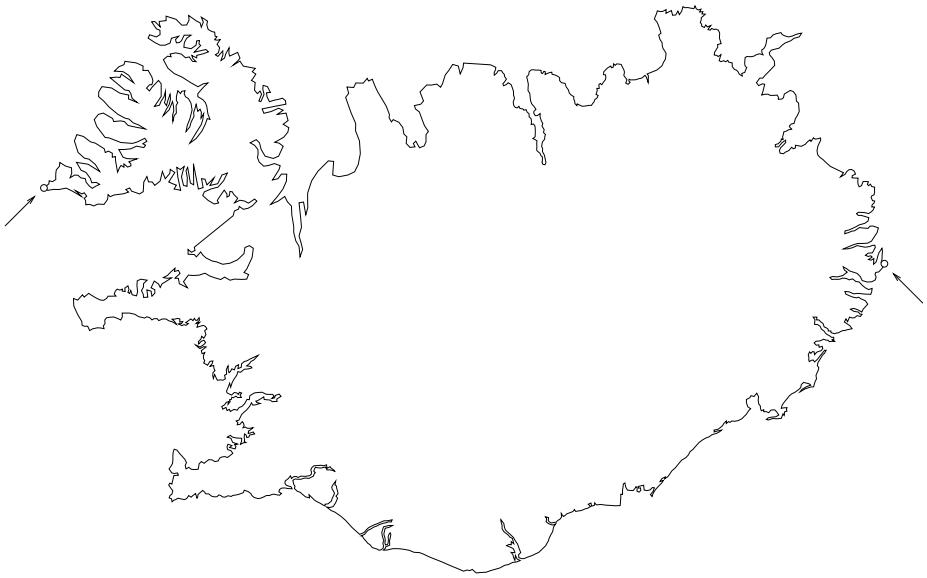

Bild 5 - Polygonpunkte mit größtem Abstand

Bild 5 zeigt das Ergebnis der Ermittlung der am weitesten von einander entfernten Punkte des Umrisses von Island. Diese Punkte könnten, wie erwähnt, als Ausgangspunkte für das Douglas/Peucker-Verfahren benutzt werden.

```
(define pga (punkte-groessterabstand island))
(zeichne-poly island xfigausdatei)
(zeichne-punkt (punkt1 pga) xfigausdatei)
(zeichne-punkt (punkt2 pga) xfigausdatei)
```

Zunächst werden diese beiden Punkte mit der Funktion `punkte-groessterabstand` berechnet, wobei der Umriss von Island bereits in die Liste `island` aus einer Datei eingelesen wurde. Danach wird der Umriss unverändert mit der Funktion `zeichne-poly` als

Linienzug in eine Ausgabedatei geschrieben, und es werden die beiden zuvor berechneten Punkte einzeln ebenfalls in diese Datei geschrieben. In Bild 5 wurden abschließend lediglich die beiden Markierungspfeile per Hand eingefügt, indem passende Pfeilobjekte mit der interaktiven Schnittstelle von XFig an den entsprechenden Stellen platziert wurden. Die restlichen graphischen Objekte entstanden direkt durch das Schreiben der Linienkoordinaten (`zeichne-poly`) und der Punktkoordinaten (`zeichne-punkt`). Das Ermitteln der beiden Punkte mit maximalem Abstand dauert in diesem Fall allerdings mehrere Minuten, denn das Ausgangspolygon hat 2094 Stützpunkte, und das Berechnungsverfahren ist von quadratischer Laufzeit.

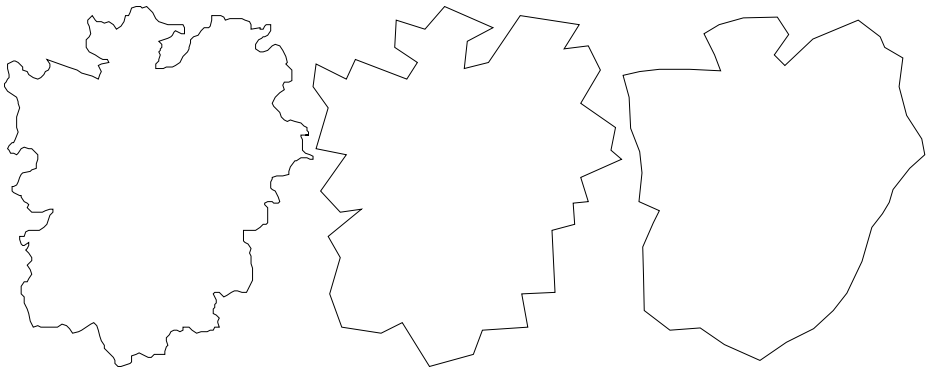

Bild 6 - vereinfachtes und gefiltertes Polygon

Als zweite und letzte Anwendung soll der Umriss des Stadtgebietes von Braunschweig original, mit Douglas/Peucker vereinfacht und diese Vereinfachung einmal gefiltert als drei Polygone nebeneinander dargestellt werden. Der Umriss von Braunschweig, hier mit 437 Stützpunkten, liegt schon in der Liste `bs` vor. Wir berechnen zunächst das umschließende Rechteck des Ausgangspolygons mit der Funktion `bbox` und vereinfachen dann das gegebene geschlossene Polygon `bs` mit der Funktion `poly-douglas-peucker`, das Ergebnis wird `bsdp` genannt. Der Parameter d = 250 wurde so gewählt, dass nur noch 47 Stützpunkten erhalten bleiben.

```
(define bsbox (bbox bs))
(define bsdp (poly-douglas-peucker bs 250))

(zeichne-poly bs xfigausdatei)
(zeichne-poly
  (verschiebe-poly bsdp
    (konstr-punkt
      (- (x-koord (punkt2 bsbox)) (x-koord (punkt1 bsbox)))
      0))
  xfigausdatei)
(zeichne-poly
  (verschiebe-poly (tiefpass bsdp)
    (konstr-punkt
```

```
        (* 2 (- (x-koord (punkt2 bsbox)) (x-koord (punkt1 bsbox))))
        0))
    xfigausdatei)
```

Nun wird der unveränderte Umriss als Polygon in die Ausgabedatei geschrieben. Als zweites Polygon wird der vereinfachte Umriss `bsdp` ausgegeben, allerdings muss dieses Polygon nach rechts verschoben werden, damit sich die Polygone nicht überschneiden. Die Y-Koordinate des Verschiebevektors ist Null, da nur in X-Richtung verschoben werden muss. Die X-Koordinate ist die Differenz der beiden X-Koordinaten des linken unteren und des rechten oberen Punktes des umschließenden Rechtecks `bsbox`. Das dritte Polygon entsteht durch Tiefpassfilterung des vereinfachten Polygons `(tiefpass bsdp)`. Zur Ausgabe wird es doppelt so weit nach rechts geschoben wie das zweite Polygon. Das Ergebnis der oben aufgeführten Aufrufe der Scheme-Funktionen zeigt Bild 6.

7 Erfahrungen

Bei der Implementierung der hier vorgestellten Funktionen, aber auch bei den nicht im Detail diskutierten Operationen zur graphischen Ausgabe von Punkten und Polygonen, zeigte sich, dass der Einsatz einer funktionalen Programmiersprache tatsächlich sehr rasch zu lauffähigen Anwendungen führt: Zunächst wurden die elementaren Operationen wie `punkt-abstand` oder `umfang` realisiert, die so wenig Programmcode enthalten, dass sie fast von vornherein zu überschauen sind und daher auch kaum getestet werden mussten. Der kompliziertere Douglas/Peucker-Algorithmus ließ sich direkt in eine einzige entsprechende rekursive Funktion umsetzen, wobei die nötigen Hilfsfunktionen in einem zweiten Schritt implementiert wurden. Bei den Laufzeiten von realistischen Anwendungen, etwa beim Ermitteln der am weitesten von einander entfernten Punkte des Umrisses von Island, traten allerdings die Nachteile des Interpreteransatzes der benutzten Sprache in den Vordergrund: Während eine ähnliche Anwendung auf einem vergleichbaren Rechner in Java implementiert nur wenige Sekunden benötigte, war die Scheme-Lösung erst nach einigen Minuten verfügbar.

Um rasch Algorithmen für Polygone und auch Varianten davon zu realisieren ist Scheme jedoch gut einsetzbar, denn seine wenigen Sprachkonstrukte sind leicht erlernbar, und durch die frei verfügbaren Interpreter können aufeinander aufbauende Funktionen schnell ausgetestet werden. Außerdem lassen sich Polygone sehr direkt durch die Listenstruktur von Scheme darstellen.

Literatur

Abelson, H.; Sussman, G.J.; Sussman, J.: Structure and interpretation of computer programs. 2. Auflage, MIT Press, Cambridge, 1996.

Douglas, D.H.; Peucker, T.K.: Algorithms for the Reduction of the Number of Points required to represent a digitized Line or its Caricature. The Canadian Cartographer, Jhrg. 10, Heft 2, 1973, 112–122.

Dybvig, R.K.: The Scheme Programming Language. 2. Auflage, MIT Press, Cambridge, 1996.

Eisenberg, M.; Abelson, H. (Hrsg.): Programming in Scheme. MIT Press, Cambridge, 1990.

Ferguson, I.; Martin, E.; Kaufmann, B.: The Schemer's Guide. 2. Auflage, Schemers Inc., Ford Lauderdale, 1995.

Grillmeyer, O.: Exploring Computer Science with Scheme. Springer, Berlin, 1999

Hake, G.; Grünreich, D.: Kartographie. 7. Auflage, Walter de Gruyter, Berlin, 1994.

Harvey, B.; Wright, M.: Simply Scheme. 2. Auflage, MIT Press, Cambridge, 1999.

Huet, G. (Hrsg.): Logical Foundations of Functional Programming. Addison-Wesley, Reading, 1990.

Kelsey, R.; Clinger, W.; Rees, J. (Hrsg.): Scheme – Revised(5) Report on the Algorithmic Language Scheme. http://www.swiss.ai.mit.edu/∼jaffer/r5rs_toc.html, 2001.

Neumann, K.; Lohmann, F.; Ehrich, H.-D.: An Experimental Geoscientific Database System. Geologisches Jahrbuch A122, Hannover, 1992, 91–100.

Neumann, K.; Selke, M.: Elemente der Programmiersprache Java vorgestellt an einer Modifikation des Douglas/Peucker-Algorithmus zur Erhaltung rechter Winkel. Mitteilungen des Bundesamtes für Kartographie und Geodäsie, Band 20, Frankfurt M., 2001, 87–97.

Pearce, J.: Programming and Meta-Programming in Scheme. Springer, Berlin, 1998.

Reade, C.: Elements of Functional Programming. Addison-Wesley, Reading, 1989.

T.Sato, T.; Smith, B.V.: XFIG Users Manual. http://www-epb.lbl.gov/xfig/, 2001.

Watt, D.A.: Programming Language Concepts and Paradigms. Prentice Hall, New York, 1990.

Wilhelm, R.; Maurer, D.: Übersetzerbau: Theorie, Konstruktion, Generierung. 2. Auflage, Springer, Berlin, 1997.