Elemente der Programmiersprache Java vorgestellt an einer Modifikation des Douglas/Peucker-Algorithmus zur Erhaltung rechter Winkel [1]

(mit 8 Bildern)

Von Karl Neumann und Martin Selke, Braunschweig

ZUSAMMENFASSUNG: Der bekannte Douglas/Peucker-Algorithmus liefert bei der Formvereinfachung von unregelmäßigen Polygonen gute Ergebnisse. Bei Polygonen mit einem hohen Anteil rechter Winkel, etwa bei Gebäudegrundrissen, sind die Ergebnisse weniger gut. Wir schlagen deshalb eine Modifikation des Douglas/Peucker-Verfahrens vor, das rechte Winkel möglichst erhält. Zur Realisierung benutzen wir die aktuelle objektorientierte Programmiersprache Java, deren wichtigste Elemente am Beispiel unserer Implementierung erläutert werden.

ABSTRACT: When applied to irregular polygons, the well-known Douglas/Peucker algorithm achieves good results. Unfortunately, the results are insufficient for polygons containing mainly right angles. Therefore, we propose a new version of the algorithm which is designed to modify only those segments of a given polygon which enclose acute or obtuse angles but not right angles. The algorithm is implemented by using the popular object oriented language Java.

1 Einleitung

Bekanntlich kann bei der Ableitung von kartographischen Objekten mit kleinerem Maßstab aus Objekten mit größerem Maßstab nicht nur einfach verkleinert werden, sondern die Objekte müssen als Teilaufgabe der kartographischen Generalisierung (vgl. dazu etwa *Hake/Grünreich* 1994) meist geometrisch vereinfacht werden. Ein sehr bekanntes Verfahren zur Formvereinfachung von Linienzügen, das auch auf flächenförmige Objekte angewendet werden kann, ist der Douglas/Peucker-Algorithmus (*Douglas/Peucker* 1973). Dieser Algorithmus reduziert abhängig von einem vorgegebenen Grenzwert die Anzahl der Stützpunkte von Polygonen, wobei deren charakteristische Form in gewissen Grenzen erhalten bleibt. Allerdings sind die Generalisierungsergebnisse bei Polygonen mit vorwiegend rechten Winkeln, wie etwa Gebäuden, häufig wenig zufriedenstellend. Es wurden daher verschiedene andere Ansätze vorgestellt, die besser zur Formvereinfachung von Gebäudegrundrissen geeignet sind (z.B. *Schmidt* 1992, *Sester* 2000).

[1] Erscheint in *Mitteilungen des Bundesamtes für Kartographie und Geodäsie, Frankfurt M., 2001.*

Der vorliegende Text beschäftigt sich ebenfalls mit der Vereinfachung von Polygonen, die hauptsächlich rechte Winkel aufweisen: Wir geben dazu eine Modifikation des Douglas/Peucker-Verfahrens an, und wir skizzieren die Implementierung dieser Modifikation an Hand der noch immer sehr aktuellen objektorientierten Programmiersprache Java, die seit ihrer Einführung zunehmend weitere Verbreitung findet (*Eckel* 1998, *Krüger* 1999). Im nächsten Abschnitt gehen wir zunächst kurz auf den ursprünglichen Douglas/Peucker-Algorithmus ein, um dann unsere Modifikationen zu diskutieren, die die Formvereinfachung von Gebäudegrundrissen optimieren sollen. Danach stellen wir zentrale Sprachelemente von Java vor, und benutzen diese anschließend bei der Skizzierung unserer Implementierung. Zum Schluss präsentieren wir Resultate einiger Formvereinfachungen unseres Programmsystems und diskutieren kurz, welche weitere Funktionlität wünschenswert wäre.

2 Douglas/Peucker-Algorithmus

Das Verfahren von *Douglas/Peucker* 1973 vereinfacht Linienzüge, indem von einem ursprünglichen Linienzug L mit n Stützpunkten und einem Grenzwert g ausgegangen wird. Der Grenzwert g bestimmt den Grad der Vereinfachung und ist im Kontext des jeweiligen Koordinatensystems zu sehen. Als gröbste Näherung des Linienzugs wird zunächst die Gerade zwischen seinem Anfangs- und Endpunkt bestimmt. Danach werden die Abstände aller n-2 Punkte zu dieser Geraden berechnet. Wenn der maximale dieser Abstände größer ist als der Grenzwert g, 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 g ist, werden alle Punkte zwischen Anfangs- und Endpunkt entfernt.

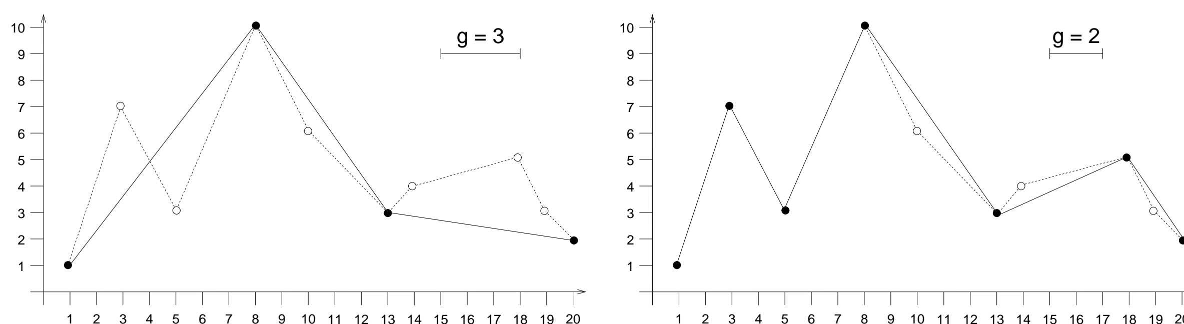

Bild 1 - Linienvereinfachung durch Douglas/Peucker-Verfahren

Als Beispiel ist in Bild 1 ein Linienzug mit 10 Stützpunkten angegeben (gestrichelte Linie), der links mit dem Grenzwert 3 und rechts mit dem Grenzwert 2 vereinfacht wird. Der relativ große Grenzwert 3 eliminiert 6 der 10 Stützpunkte, während durch

den Grenzwert 2 nur 3 Punkte entfernt werden. Man erkennt auch an diesem Beispiel recht gut, dass sich für kleiner werdendes g der Resultatslinienzug immer mehr dem Ausgangslinienzug annähert.

Das Douglas/Peucker-Verfahren lässt sich leicht von Linienzügen auf geschlossene Polygone übertragen: Als zusätzliche Parameter werden nur ein Start- und ein Endpunkt benötigt. Diese erhält man etwa durch Auswahl der beiden Punkte des betrachteten Polygons, die den größten Abstand zueinander haben. Es zeigt sich, dass so vereinfachte Polygone ihre charakteristische Form recht gut bewahren, vorausgesetzt der Grenzwert g wird nicht zu groß gewählt. Im oberen Teil von Bild 2 wird z.B. eine Stadtfläche, die durch ca. 400 Stützpunkte gegeben ist, durch geeignete Wahl des Grenzwertes in ein Polygon mit nur noch ca. 100 Stützpunkten vereinfacht.

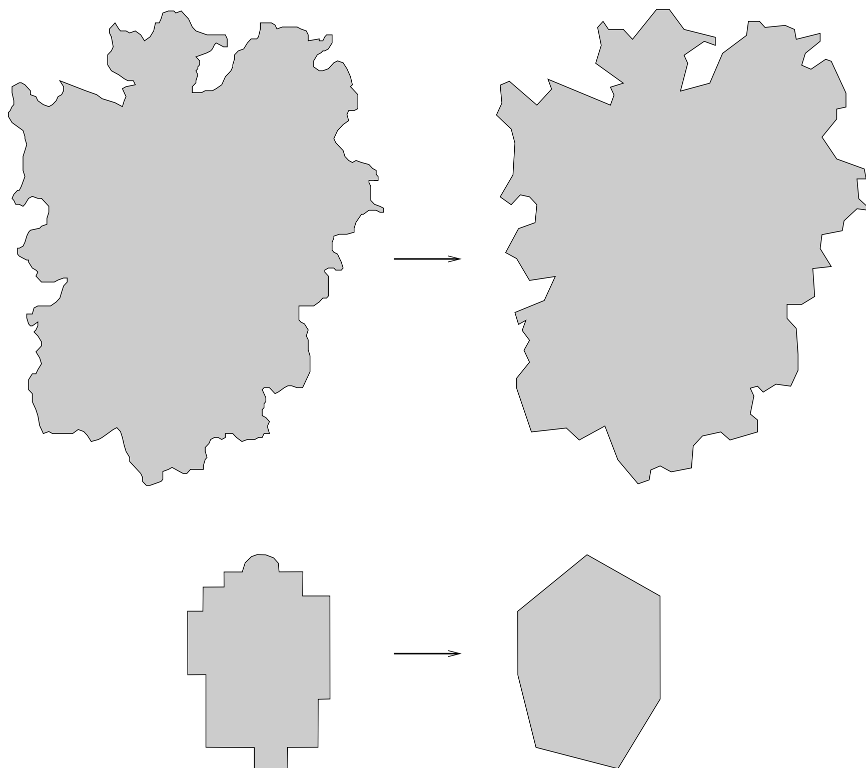

Bild 2 - Vereinfachung von geschlossenen Polygonen

Dieses Bewahren der charakteristischen Form gilt jedoch nur für unregelmäßige Polygone, wie größere Siedlungsflächen, Moore, Seen oder andere möglichst natürlich entstandene Gebietsbegrenzungen. Wendet man dagegen das Douglas/Peucker-Verfahren auf Polygone an, deren Form durch das Auftreten von rechten Winkeln bestimmt wird, wie das besonders für Gebäudegrundrisse gilt, so sind die Ergebnisse meist wenig ansprechend. Als Beispiel dazu zeigt Bild 2 (unten) die Formvereinfachung eines Grundrisses von ursprünglich 28 auf 7 Stützpunkte.

3 Modifikationen zur Erhaltung rechter Winkel

Um die charakteristische Form von Gebäudegrundrissen möglichst zu erhalten, gehen wir von der Heuristik aus, dass lange Kanten in den Ausgangspolygonen besonders wichtig sind. Wir ändern daher die Aufrufparameter des Douglas/Peucker-Verfahrens: Anstatt der beiden Punkte mit dem größten Abstand wählen wir die acht Endpunkte der vier längsten Kanten. Der eigentliche Vereinfachungsalgorithmus wird dann rekursiv auf jeweils benachbarte Punkte dieser Ausgangsmenge angewendet. Einfache rechteckige Polygone werden so überhaupt nicht verändert. Bild 3 zeigt die vier längsten Kanten des bereits bekannten Gebäudegrundrisses und die Menge der acht Startpunkte. Die Nummerierung der Punkte erfolgte willkürlich; die Punkte V und VI fallen zusammen.

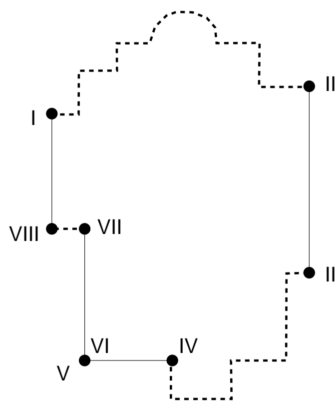

Bild 3 - längste Kanten mit Startpunkten

Anders als beim ursprünglichen Verfahren wird jetzt für jedes zu betrachtende Segment nicht einfach der Punkt mit dem größten Abstand zur jeweiligen Geraden ermittelt, sondern aus der Menge der Punkte, die Scheitelpunkte eines rechten Winkels sind, derjenige mit dem größten Abstand. Gibt es keine rechten Winkel, so wird nur der Punkt mit dem größten Abstand bestimmt. Falls der Abstand größer ist als der vorgegebene Grenzwert g, bleibt auch hier der zugehörige Punkt als signifikant erhalten. Handelt es sich bei dem als signifikant erkannten Punkt um einen Scheitelpunkt eines rechten Winkels, so werden zur Erhaltung des rechten Winkels auch die beiden Nachbarpunkte beibehalten. Dies geschieht allerdings nur, wenn mindestens eine der beiden Kanten zwischen dem Punkt und seinen Nachbarn länger ist als der Grenzwert g.

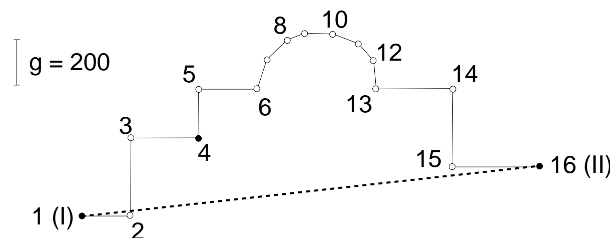

Bild 4 - Bestimmung eines signifikanten Punktes

In Bild 4 wird als Beispiel des Ablaufs das Segment I-II aus Bild 3 betrachtet. Aus Gründen der Übersichtlichkeit sind nicht alle Stützpunkte mit Nummern versehen worden; Punkt I ist der 1. Punkt des Ausgangspolygons, und Punkt II hat die Nummer 16. Punkt 9 hat den größten Abstand zur Geraden 1-16, ist aber kein Scheitelpunkt eines rechten Winkels. Auch die Punkte 3, 5 und 14 sind nicht "rechtwinklig", nur Punkt 4 ist Scheitelpunkt eines rechten Winkels und wird als signifikant angesehen, da sein Abstand von der Geraden 1-16 größer ist als der vorgegebene Grenzwert g = 200. Auch die Punkte 3 und 5 werden beibehalten, da sie zusammen mit Punkt 4 den rechten Winkel bilden und die Gerade 3-4 länger als g ist. Als nächstes werden daher die Segmente 1-3 und 5-16 betrachtet (Bild 5, oben links). Punkt 2 hat einen zu geringen Abstand zur Geraden 1-3, und Punkt 10 wird als wesentlich für Segment 5-16 erkannt. Dementsprechend werden noch die Segmente 5-10, 10-16 und schließlich 10-15 behandelt, und es bleiben insgesamt die Punkte 1, 3, 4, 5, 10, 14, 15, 16 erhalten (Bild 5, unten rechts).

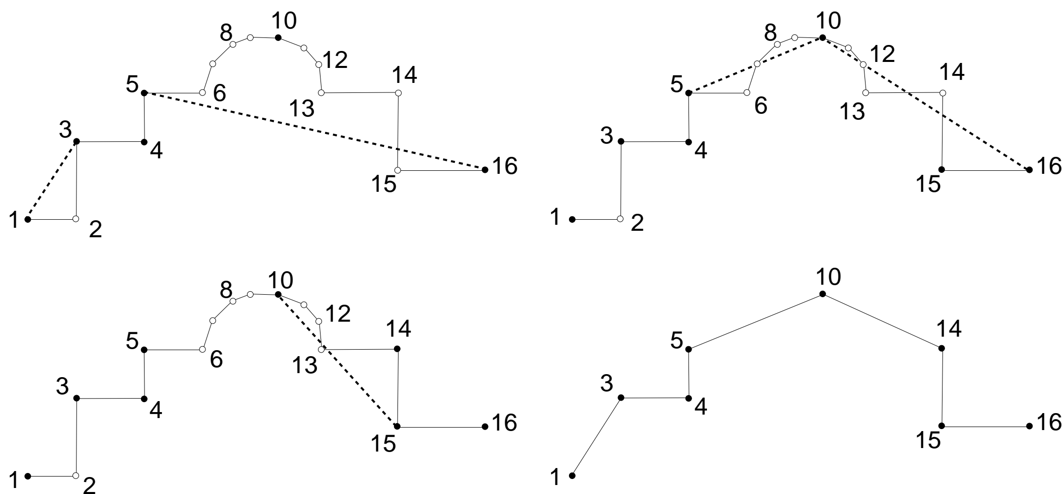

Bild 5 - Bestimmung weiterer Punkte

Durch die beiden relativ einfachen Modifikationen – die acht Endpunkte der vier längsten Kanten bilden die Startpunkte, und es werden im weiteren rekursiven Verfahren bevorzugt Scheitelpunkte von rechten Winkeln als signifikant ausgewählt – scheint die Charakteristik von Gebäudegrundrissen tatsächlich besser erhalten zu werden als vom ursprünglichen Douglas/Peucker-Algorithmus. Versuche mit einigen "gebäude-ähnlichen" Polygonen ergaben jedenfalls entsprechende Resultate (siehe Abschnitt 6).

4 Elemente der Programmiersprache Java

Die objektorientierte Programmiersprache Java wurde seit den frühen 90er Jahren entwickelt (vgl. *Arnold/Gosling* 1996) und ist seit ca. 1995 mit der Verbreitung des World Wide Webs recht populär geworden, da Java zunächst hauptsächlich als "Netzprogrammiersprache" eingesetzt wurde. Sämtliche Daten und Funktionen müssen in Java in Klas-

sen angegeben werden, die ihrerseits Typen von Objekten definieren. Es sind zwar starke Ähnlichkeiten zur Programmiersprache C++ vorhanden, zahlreiche fehleranfällige Konstrukte wurden jedoch vermieden: So gibt es in Java keine expliziten Zeiger, die Speicherverwaltung geschieht automatisch, Feldgrenzen lassen sich nicht überschreiten, es werden ein rigoroses Typkonzept und eine saubere Ausnahmebehandlung unterstützt. Javaprogramme werden in einen speziellen Bytecode übersetzt, der dann durch die so genannte Java Virtual Machine interpretiert wird. Da diese virtuelle Maschine inzwischen auf sehr vielen Plattformen implementiert wurde, sind einmal übersetzte Programme dort überall lauffähig. Javaprogramme, die als Methoden der speziellen Applet-Klasse realisiert wurden, können sogar durch jeden internetfähigen Browser ausgeführt werden.

Die konsequente Objektorientierung macht zur besseren Lesbarkeit von Programmen nur bei den einfachen Datentypen eine Ausnahme: Die Werte der acht vorgegebenen Datentypen boolean, char, byte, short, int, long, float, double werden – wie auch in anderen Programmiersprachen üblich – nicht als Objekte behandelt. Auf diesen Datentypen stehen zahlreiche Operationen zur Verfügung, z.B. die arithmetischen (+, -, *, /, %) und die logischen (==, !=, <, <=, >, >=), und es lassen sich Werte eines Datentyps durch explizite und implizite Typkonvertierung in Werte eines anderen Datentyps umwandeln. Da es manchmal hilfreich ist, auch Werte von Datentypen als Objekte ansprechen zu können, steht für jeden Datentyp eine so genannte Objekt-Hüllklasse (wrapper class) mit entsprechenden Umwandlungsmethoden zur Verfügung.

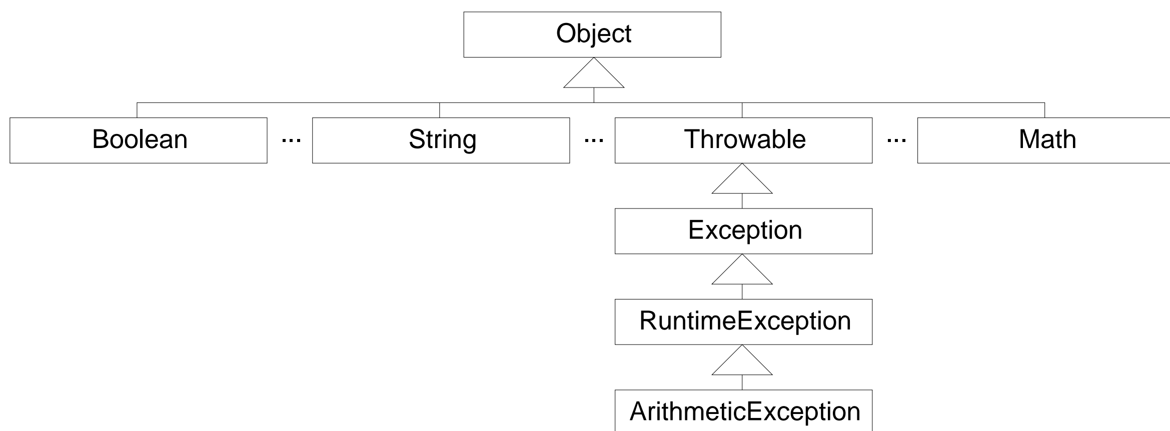

Bild 6 - einige Objektklassen aus dem Paket java.lang

Java bietet im Wesentlichen die aus anderen imperativen Sprachen gewohnten Anweisungen zur Steuerung des Programmablaufs, wie Wertzuweisung, Do-, While- und For-Anweisung, If-Else- und Case-Anweisung, Return-Anweisung. Neben diesem relativ kleinen Sprachkern gibt es eine große Sammlung von Paketen (packages), die Bibliotheken von für viele Anwendungen nützliche Klassen bereitstellen. So existieren u.a. Pakete mit mathematischen Funktionen, Datenbankschnittstellen und Klassen zur Realisierung von portablen graphischen Benutzerschnittstellen wie das so genannte Abstract Windowing

Toolkit (AWT). In Bild 6 ist ein sehr kleiner Ausschnitt aus dem Paket java.lang dargestellt, das u.a. die oben angesprochenen Hüllklassen enthält und auch die Klasse Object, die Oberklasse für alle anderen Objektklassen ist. Eine umfassende Übersicht über die meisten Java-Bibliotheken mit ihren Klassen, Variablen, Konstanten und Methoden bieten *Chan/Lee/Kramer* 1998 und *Chan/Lee* 1998.

Wegen der zahlreichen bereits vorhandenen Objektklassen und des Vererbungsprinzips werden Java-Anwendungen häufig nicht komplett neu entwickelt, sondern es werden eher existierende Klassen um neue Funktionalität erweitert, was meist durch das Anlegen von geeigneten Unterklassen geschieht. Allerdings ist es besonders in der Anfangsphase schwierig, sich einen Überblick über die vorhandenen Bibliotheken mit ihren Klassen zu verschaffen. Empfehlenswerte Einführungen in die Programmierung mit Java findet man z.B. in *Rauh* 1999 und *Echtle/Gödicke* 2000.

5 Skizzen der Implementierung

Die Implementierung unserer Variante des Douglas/Peucker-Algorithmus verteilt sich auf sieben eigene Objektklassen, von denen wir in diesen Abschnitt die vier wichtigsten kurz vorstellen werden. In Bild 7 ist dazu als Übersicht ein Klassendiagramm dargestellt, das unsere Klassen Polygon, Punkt, Vektor und Funktionen in UML-Notation zeigt. Diese Notationsform haben wir auch bereits in *Neumann/Eckstein* 2000 benutzt; zu UML vgl. z.B. auch *Fowler/Scott* 1997, *Oestereich* 1997, *Rumbaugh/Jacobson/Booch* 1999 und *Hitz/Kappel* 1999.

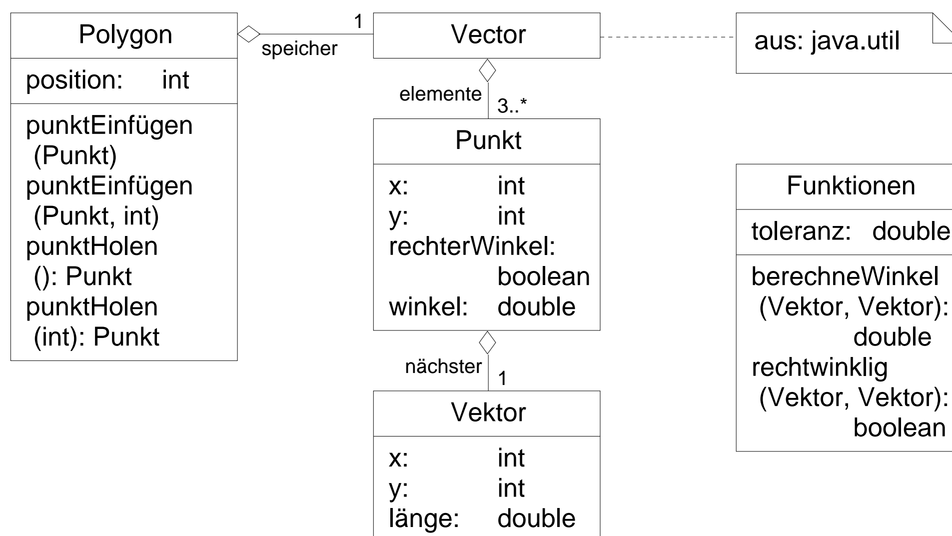

Bild 7 - wichtige Objektklassen der Implementierung

Einzelne Punkte von Polygonen werden durch Objekte der Klasse Punkt repräsentiert. Ein Punkt enthält die beiden Koordinaten x und y sowie die boolesche Information, ob es

sich um den Scheitelpunkt eines rechten Winkels handelt. Zusätzlich wird noch der Kosinus des Winkels gespeichert. Zur Optimierung der Ermittlung von Kantenlängen und Winkeln im Laufe des rekursiven Vereinfachungsverfahrens hat jeder Punkt einen zugehörigen Vektor, der auf den nächsten Punkt zeigt; außerdem ist die Länge des Vektors gespeichert. Für Objekte der Klassen Punkt und Vektor sind keine Methoden definiert, da sie nur zur Aufnahme von Daten dienen. Dies ist bei der zentralen Klasse Polygon anders, die zur Verwaltung der verschiedenen Polygone eingesetzt wird. Ein Polygon wird hier realisiert durch eine Liste von Objekten der Klasse Punkt. Dazu verwenden wir die Klasse Vector aus dem Paket java.util, die gerade solche Listen für Objekte der Klasse Object bereitstellt. Da unsere Klasse Punkt – wie jede Klasse – automatisch eine Unterklasse der Klasse Object ist, können wir solche Listen zu Verwaltung von Punkten benutzen. Die in Bild 7 angedeuteten Methoden zum Einfügen und Holen von Punkten sind jeweils überladen: Sie können mit verschiedenen Anzahlen von Argumenten aufgerufen werden. So fügt der Aufruf poly.punktEinfügen(p) den Punkt, der durch die Variable p gegeben ist, am Ende der durch poly angesprochenen Liste ein, während poly.punktEinfügen(p,pos) den Punkt p an der Stelle pos in die Liste einfügt. Das folgende Programmfragment ist ein kleiner Ausschnitt aus der Klasse Polygon. Wir folgen den Java-Konventionen, nach denen Objektklassen stets mit einem großen und Variablen sowie Methoden mit einem kleinen Buchstaben beginnen.

```
01  import Punkt;
02
03  public class Polygon {
04    int position;
05    private Vector speicher;
06
07    Polygon () {
08      position = 0;
09      speicher = new Vector(50,10);
10    }
11
12    void punktEinfuegen(Punkt p){
13      Punkt p1;
14      if (speicher.size() == 0)
15        speicher.addElement(p);
16      else {
17        p1 = (Punkt)speicher.elementAt(speicher.size()-1);
18        if (!(p.x==p1.x & p.y==p1.y))
19          speicher.addElement(p);
20      }
21    }
22  ...
23  }
```

Zunächst wird in Zeile 1 die Klasse Punkt als Import deklariert, und in Zeile 4 und 5 werden zwei Variable deklariert, die jedes Objekt vom Typ Polygon besitzt: Die ganze Zahl position dient zum zyklischen Durchlaufen von Polygonen, und das Vector-Objekt speicher nimmt wie angesprochen die Liste der Punkte auf. Dabei ist der direkte Zugriff auf diese Liste von Außen nicht erlaubt (Attribut: privat). Die Zeilen 7–10 enthalten den Konstruktor der Objektklasse Polygon: Wenn ein neues Objekt dieser Klasse angelegt wird, wird die Variable position mit Null belegt, und es wird ein neues Vector-Objekt generiert, das zunächst potenziell 50 Elemente aufnehmen kann. Werden im Programmlauf mehr Elemente benötigt, so werden vom Laufzeitsystem bei Bedarf jeweils 10 neue Plätze zur Verfügung gestellt. Diese Parameter hätten auch anders gewählt werden können, sie sind hier abgestimmt auf Polygone mit relativ wenig Stützpunkten.

Auf Zeile 12 beginnt die Methode punktEinfuegen zum Einfügen eines Punktes in ein Polygon-Objekt, wobei hier nur die Variante mit einem Parameter aufgeführt ist. Da die Methode kein Ergebnis zurückliefert, hat sie das Attribut void. Die in Zeile 13 deklarierte Variable p1 ist eine innerhalb der Methode lokale Hilfsvariable, die eine Referenz auf ein Objekt vom Typ Punkt aufnehmen kann. Falls es noch keine Elemente in der Liste speicher des betrachteten Polygons gibt, wird der als Parameter übergebene Punkt einfach in die Liste eingefügt und ist damit zugleich erstes und letztes Element der Liste (Zeilen 14, 15). Dabei benutzen wir die auf Objekten des Typs Vector definierten Methoden size() und addElement(...): size liefert die aktuelle Anzahl von Listenelementen zurück, und addElement fügt ein neues Element hinten an der Liste an, hier den neuen Punkt p. Falls bereits mindestens ein Punkt in der Liste existiert, wird der bislang letzte Punkt durch die Hilfsvariable p1 zugreifbar gemacht (Zeile 17). Hier wählt die Methode elementAt(n) das n-te Element eines Vector-Objekts aus. Durch die Angabe speicher.size()-1 greifen wir auf das letzte Objekt in der Liste zu, und durch die explizite Typangabe (Punkt) wird spezifiziert, dass das zugegriffene Objekt als Objekt der Klasse Punkt interpretiert werden soll und nicht als Objekt der Klasse Object. Schließlich enthält Zeile 18 den Test, ob der neue einzufügende Punkt auch tatsächlich ein neuer Punkt ist und nicht schon als bislang letzter Punkt gespeichert ist: Falls nicht beide Koordinaten der beiden betrachteten Punkte gleich sind, wird der neue Punkt hinten an die Liste angefügt (Zeile 19).

Damit sind bereits zahlreiche Java-Konstrukte durch Beispiele illustriert worden, und auch die zentrale Klasse Polygon ist inklusive einer zugehörigen Methode erläutert worden. Im folgenden Programmfragment soll daher nur noch sehr knapp auf die Klasse Funktionen eingegangen werden, in der einige zentrale Konstanten und Hilfsfunktionen bereitgestellt werden.

```
01  public class Funktionen {
02    private static final double winkel895 = 0.00872653;
03    static final double toleranz = winkel895;
04
05    static double berechneWinkel(Vektor v1, Vektor v2){
06      return (((v1.x*v2.x)+(v1.y*v2.y))
```

```
07               / ((v1.laenge)*(v2.laenge)));
08     }
09
10     static boolean rechtwinklig(Vektor v1, Vektor v2){
11       boolean ergebnis;
12       if (Math.abs(berechneWinkel(v1,v2)) <= toleranz)
13         ergebnis = true;
14       else ergebnis = false;
15       return ergebnis;
16     }
17   ...
18   }
```

In den Zeilen 2 und 3 werden zunächst zwei Konstanten definiert (Attribut: final), wobei die Konstante winkel895 nach Außen nicht sichtbar ist (private). Beide Konstanten sind auf Klassenebene und nicht auf Objektebene definiert (Attribut: static). Auch die beiden aufgeführten Methoden sind Klassenmethoden. Die lokale Konstante winkel895 enthält den Kosinus von 89.5 Grad, und die nach Außen sichtbare Konstante toleranz wird auf den gleichen Wert gesetzt. Wir betrachten hier also Winkel im Bereich von 89.5 bis 90.5 Grad als "rechtwinklig". Die Methode berechneWinkel berechnet den Kosinus des Winkels zwischen zwei Vektor-Objekten, indem das Skalarprodukt der Vektoren durch das Produkt ihrer Längen dividiert wird (Zeilen 6 und 7). Die boolesche Methode rechtwinklig benutzt diese Methode zur Winkelberechnung (Zeile 12): Zwei Vektoren sind dann rechtwinklig, wenn der Absolutbetrag des Kosinus ihres Schnittwinkels kleiner ist als der vorgegebene Toleranzwert. Dabei wird die Funktion abs der Klasse Math aus dem Paket java.lang benutzt.

6 Ergebnisse und Ausblick

Das modifizierte Douglas/Peucker-Verfahren zur Berücksichtigung rechter Winkel wurde, wie im letzten Abschnitt skizziert, von *Selke* 1999 implementiert. Die realisierte Java-Applikation kann Polygone einlesen und vereinfachen, die dem Format des Zeichenprogramms Xfig folgen, ein in der Unix-Welt oft angewendetes Werkzeug. Da die vereinfachten Polygone auch wieder in diesem Format ausgegeben werden, können die Ein- und Ausgaben des Programmsystems leicht graphisch dargestellt und so verglichen werden. In Bild 8 sind drei Gebäudegrundrisse – darunter der aus Abschnitt 3 bekannte – mit zwei verschiedenen Grenzwerten vereinfacht worden. Oben in der Mitte befinden sich die Originalgrundrisse, in der linken Hälfte die nach dem unveränderten Douglas/Peucker-Algorithmus vereinfachten und rechts die mit dem hier vorgestellten modifizierten Verfahren vereinfachten Grundrisse. Diese und mit weiteren Polygonen durchgeführte Experimente deuten daraufhin, dass unsere recht einfachen Modifikationen des Douglas/Peucker-Verfahrens die Charakteristik von "gebäude-ähnlichen" Polygonen tatsächlich relativ gut erhalten.

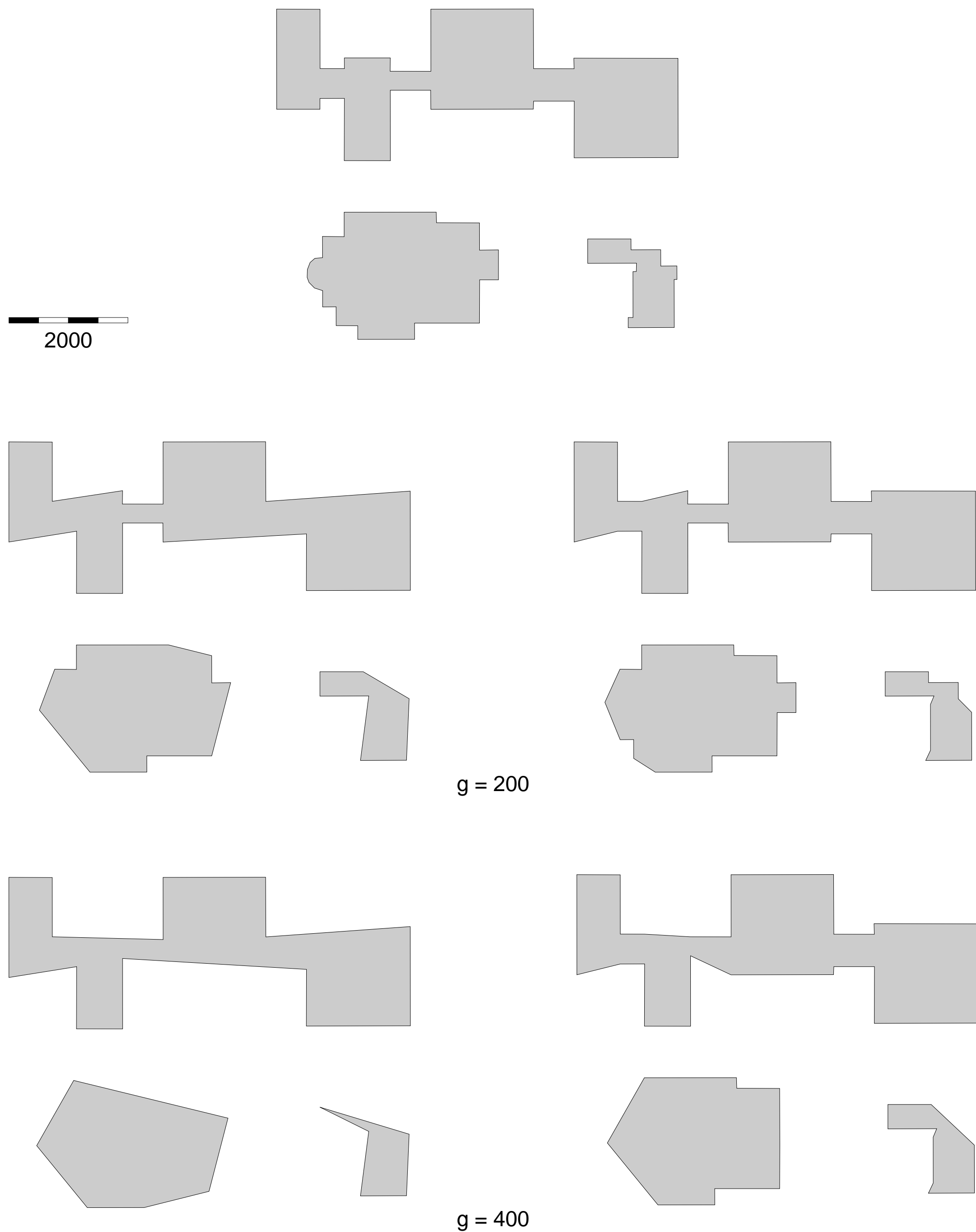

Bild 8 - Vereinfachungsexperimente

Um die Tauglichkeit des Verfahrens im Rahmen kartographischer Generalisierung zu testen, wären weitere Experimente nötig. Diese müssten sich auf realistische Massendaten beziehen, z.B. auf Gebäudedaten der Automatisierten Liegenschaftskarte (ALK-Daten). Dazu wäre eine komfortable graphische Benutzeroberfläche wünschenswert, die es etwa gestattet, den jeweiligen Wert des Grenzwertes g mit einem “Schieberegler” einzustellen, und die die dadurch ausgelösten Änderungen an den Gebäudegrundrissen möglichst umgehend visualisiert. Wegen der in Java vorhandenen Klassen zur Unterstützung von

ereignisgesteuerten graphischen Benutzerschnittstellen hielte sich der Implementierungsaufwand in Grenzen.

Literatur

Arnold, K.; Gosling, J.: The Java Programming Language. Addison-Wesley, Reading, 1996.

Chan, P.; Lee, R.; Kramer, D.: The Java Class Libraries, Volume 1. 2. Auflage, Addison-Wesley, Reading, 1998.

Chan, P.; Lee, R.: The Java Class Libraries, Volume 2. 2. Auflage, Addison-Wesley, Reading, 1998.

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.

Echtle, K.; Gödicke, M.: Lehrbuch der Programmierung mit Java. dpunkt, Heidelberg, 2000.

Eckel, B.: Thinking in Java. Prentice Hall, Upper Saddle River, 1998.

Fowler, M.; Scott, K.: UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley, Reading, 1997.

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

Hitz, M.; Kappel, G.: UML@Work. dpunkt, Heidelberg, 1999.

Krüger, G.: Go To Java 2. Addison-Wesley, München, 1999.

Neumann, K.; Eckstein S.: Einführung in die Unified Modeling Language am Beispiel von ATKIS. Mitteilungen des Bundesamtes für Kartographie und Geodäsie, Band 17, Frankfurt M., 2000, 81–88.

Oestereich, B.: Objektorientierte Softwareentwicklung mit der Unified Modeling Language. Oldenbourg, München, 1997.

Rauh, O.: Objektorientierte Programmierung in Java. Vieweg, Wiesbaden, 1999.

Rumbaugh, J; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Guide. Addison-Wesley, Reading, 1999.

Schmidt, C.: Stand der Entwicklung eines Programmsystems zur automatischen Generalisierung von Gebäuden und Verkehrswegen. Nachrichten aus dem Karten- und Vermessungswesen, Reihe I, Nr. 108, Frankfurt M., 1992, 159-177.

Selke, M.: Entwicklung einer Variante des Douglas/Peucker-Verfahrens und deren Implementierung in Java. Diplomarbeit, TU Braunschweig, 1999.

Sester, M.: Automatische Generalisierung mittels Ausgleichung. Mitteilungen des Bundesamtes für Kartographie und Geodäsie, Band 17, Frankfurt M., 2000, 105–113.