Vergleich verschiedener Datenstrukturen zur Verwaltung von Dreiecksnetzen

Sven Emme
Matrikelnummer: 20675649

Überarbeitete Fassung vom
29. März 2012

Inhaltsverzeichnis

1 Einleitung

In der Computergrafik werden komplexe Objekte häufig mithilfe von Polygonnetzen approximiert. Es gibt eine Reihe von Datenstrukturen um diese Polygonnetze zu speichern und zu verarbeiten. Diese unterscheiden sich in Bezug auf Speicherverbrauch und Performance voneinander. Der Speicherverbrauch der Datenstrukturen ist dabei der wichtigere Faktor, da der Speicher eine obere Schranke darstellt. Wenn ein Polygonnetz zuviel Speicher verbraucht, wäre die negative Konsequenz, dass das Modell nicht in den Speicher passt und damit nicht bearbeitet werden kann, oder dass durch häufiges Auslagern die Performance extrem darunter leidet [4].

Die Wahl der Datenstruktur ist bei der Bearbeitung von Polygonnetzen bedeutsam, da die Performance von Algorithmen, die auf Polygonnetzen arbeiten, davon abhängen kann [12]. Betrachtet man z.B. einen Algorithmus, der auf benachbarte Flächen zugreifen muss, läuft dieser schneller, wenn die verwendete Datenstruktur einen direkten Zugriff auf benachbarte Flächen gewährt. Durch den direkten Zugriff müssen die benachbarten Flächen nicht erst ermittelt werden.

In dieser Arbeit werden zwei dieser Datenstrukturen vorgestellt, die in der Computergrafik häufig verwendet werden. Zum einem die *Winged Edge*-Datenstruktur, die in der Computergrafik lange Zeit verwendet wurde, sowie die etwas jüngere *Half-Edge*-Datenstruktur, die etwas ausgereifter ist.

Dann wird eine neuere Datenstruktur, namens *Directed Edges* (gerichtete Kanten), vorgestellt, die in der Lage ist speichersparend Dreiecksnetze zu repräsentieren und dabei die Möglichkeit mit sich bringt Speicherplatz auf Kosten der Performance zu sparen.

2 Polygonnetze

2.1 Definitionen

Polygonnetze stellen im *Echtzeit Rendering* die Grundlage zur Modellierung und Darstellung von komplexen Objekten dar. Da die hier vorgestellten Datenstrukturen Polygonnetze verwalten können sollen, ist es nötig zu definieren was Polygonnetze sind. Die folgenden Definitionen basieren auf den Definitionen aus [10].

Definition 1. Ein Polygon P ist ein Tupel (V, E). Wobei $V := \{v_0, v_1, \ldots, v_{n-1}\}$ n Punkte, die in einer Ebene liegen, und $E := \{e_1 = v_0v_1, e_2 = v_1v_2, \ldots, e_{n-1} = v_{n-1}v_0\}$ n Liniensegmente sind, die diese Punkte verbinden. P ist ein Polygon, wenn

1. der Schnitt von jedem Paar aufeinanderfolgender Liniensegmente der Punkt, den sie sich teilen, ist. Wobei e_0 Nachfolger von e_{n-1} ist.
2. der Schnitt von Liniensegmenten, die nicht aufeinander folgen, stets leer ist.

Nach dieser Definition werden nur einfache, planare Polygone betrachtet.

Definition 2. Ein Polygonnetz M ist ein Tripel (V, E, F),

- Wobei F eine endliche Menge konvexer Polygone, der Flächen von M, ist, mit der Eigenschaft, dass je zwei paarweise verschiedene Flächen entweder
  1. disjunkt sind, oder
  2. genau einen Punkt gemeinsam haben, oder
  3. genau zwei Punkte und die Kante, die diese beiden Punkte verbindet, gemeinsam haben
- Weiter ist E die Menge aller Kanten aller Flächen aus F und V die Menge aller Endpunkte aller Kanten aus E.

Diese Definition vereinfacht die Beschreibung von Polygonnetzen, ohne sie einzuschränken. Konkave Polygone können in mehrere koplanare konvexe Polygone zerlegt werden [10]. Durch diese Definition fallen unschöne Konstrukte wie *T-junktions* sowie selbstschneidende Polygone und Polygonnetze weg. Ausserdem ist bei dieser Definition klar, wie mit Löchern in Polygonen zu verfahren ist ohne Bedingung 2 von Def. 1 zu verletzen (siehe Abb. 1 links).

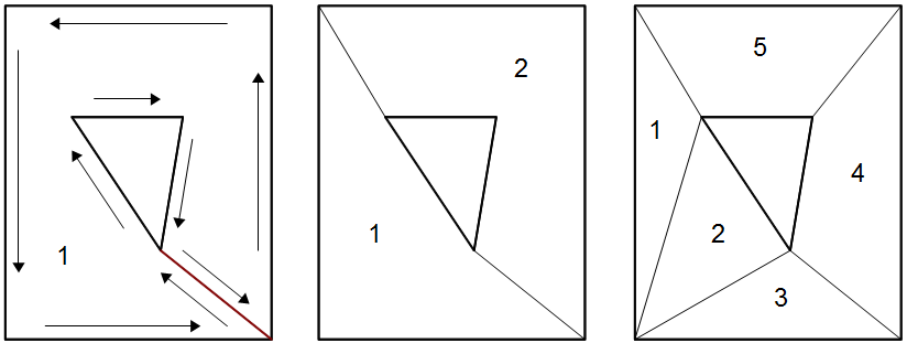

Abbildung 1: 2 Repräsentationen eines Rechtecks mit einem dreieckigen Loch. Die linke Repräsentation ist ungültig, da sie gegen Def. 1 und Def. 2 verstößt, denn das Polygon ist nicht konvex und die rot markierte Kante kommt zweimal vor und schneidet sich damit selbst. Die mittlere Zerlegung verstößt gegen Def. 2 und bringt die unschöne Eigenschaft mit sich, dass die beiden Polygone an zwei Kanten benachbart sind. Die rechte Repräsentation ist erlaubt. Hier wurde das Polygon in 5 konvexe Polygone zerlegt.

Das zentrale Thema dieser Arbeit stellen jedoch Dreiecksnetze dar. Dabei handelt es sich um Polygonnetze, deren Flächen nur aus Dreiecken bestehen. Dreiecke besitzen eine Reihe von Vorteilen in der Computergrafik, so sind sie die einzigen Polygone, die immer konvex sind [6] und von Grafikkarten direkt unterstüzt werden [9].

Des weiteren wird in dieser Arbeit der Begriff der Nicht-Mannigfaltigkeit von Ecken und Kanten auftauchen. Die folgenden Definitionen basieren auf Definitionen aus [2].

Definition 3. Eine nicht-mannigfaltige Kante eines Polygonnetzes M ist eine Kante, die in mehr als zwei Polygonen aus M enthalten ist (siehe Abb. 2 rechts).

Die Definition für nicht-mannigfaltige Ecken gestaltet sich schwieriger, deshalb stellt man sich die Grundflächen der Polygone trianguliert, also in Dreiecke zerlegt, vor. Dann kann folgende Definition angewendet werden.

Definition 4. Eine nicht-mannigfaltige Ecke e eines Dreiecksnetzes M ist eine Ecke, die mit mehr als einem Dreiecksfächer verbunden ist (siehe Abb. 2 links).

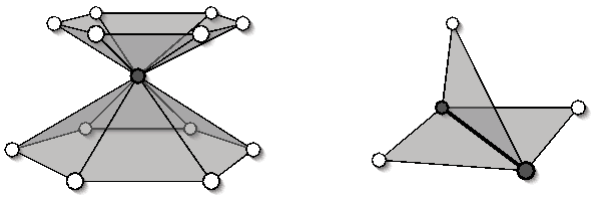

Abbildung 2: Links: Eine nicht-mannigfaltige Ecke. Rechts: Eine nicht-mannigfaltige Kante (Grafik aus [3]).

Definition 5. Ein Polygonnetz M, das eine nicht-mannigfaltige Ecke oder Kante enthält, ist ein nicht-mannigfaltiges Polygonnetz.

2.2 Nachbarschaftsbeziehungen von Polygonnetzen

T. C. Woo hat 1984 Datenstrukturen für Polygonnetze auf ihre Laufzeit und ihren Speicherplatzverbrauch untersucht. Zu diesem Zeitpunkt war die *Winged-Edge*-Datenstruktur seit zwölf Jahren der Standard für die Repräsentation von Polygonnetzen.

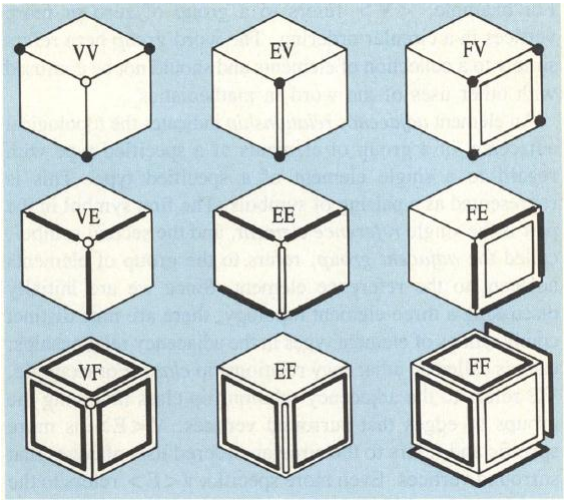

Abbildung 3: Die 9 Nachbarschaftsbeziehungen am Beispiel eines Würfels. Die hell ausgefüllte Instanz stellt die Eingabe dar, die dunkel ausgefüllten Instanzen die Ausgaben [11]

Nach Woo gibt es neun Nachbarschaftsbeziehungen der Form $X \rightarrow Y$ in Polygonnetzen, wobei X eine Kante, Ecke oder Fläche des Polygonnetzes und

Y alle benachbarten Kanten, Ecken oder Flächen von X darstellt. Relationen können dabei, wenn sie in der Datenstruktur gespeichert sind, miteinander verknüpft werden. Wenn z.B. $F \to V$ gesucht ist und $F \to E$ und $E \to V$ gespeichert sind, kann, durch Kombinieren der beiden Relationen, $F \to V$ berechnet werden. Bei Relationen, die nicht direkt gespeichert sind, bzw. nicht durch Kombinieren der gespeicherten Relationen berechnet werden können, ist es nötig die Relation umzukehren und alle Y zu suchen, die mit X benachbart sind. Diese Operation würde eine Laufzeit von $\mathcal{O}(n)$ haben, wobei n die Anzahl der gespeicherten Elemente der Instanz ist. Daraus folgt, dass, wenn man diese Relation für jedes Element des Polygonnetzes ermitteln möchte, mit quadratischer Laufzeit zu rechnen ist [12].

3 Datenstrukturen für Polygonnetze

Bei der Repräsentation von Objekten durch Polygonnetze kommen unterschiedliche Datenstrukturen in Frage. Je nach Anwendungsbereich eignen sich bestimmte Strukturen besser als andere, da jede Struktur ihre eigenen Vor- und Nachteile hat. Ein besonderes Augenmerk fällt dabei auf Speicherverbrauch und die Rechenzeit um Nachbarschaftsbeziehungen zu ermitteln. Des weiteren soll überprüft werden, ob die Datenstrukturen nicht-mannigfaltige Objekte darstellen können. Es ist zwar möglich, dass nicht-mannigfaltige Objekte nicht unterstützt werden und sie stattdessen in mannigfaltige umgewandelt werden, wie in [5]. Dies würde allerdings weitere Berechnungen auf dem Objekt erfordern und dessen Geometrie verfälschen. Deshalb ist es wünschenswert, nichtmannigfaltige Objekte in einer Datenstruktur nativ zu unterstützen.

3.1 Shared-Vertex

Eine weit verbreitete Technik in der Computergrafik ist es, bei Polygonnetzen die geometrischen von den topologischen Informationen zu trennen. Durch diese Trennung wird der Speicherverbrauch reduziert und Rechenzeit gespart.

Die hier vorgestellte Technik bildet die Grundlage für komplexere Datenstrukturen. Die *Shared-Vertex*-Datenstruktur speichert die Positionen der Ecken in einem Array jeweils einmal als drei 4 byte *float*-Werte. Flächen des Polygonnetzes müssen so nicht mehr für jede Ecke diese 3 Werte angeben, sondern können mit einem 4 byte *integer* auf die Position im Array verweisen.

Im Falle von Dreiecksnetzen (siehe Abb. 4) wird der Speicherverbrauch durch diese Technik von 36 byte pro Dreieck auf 18 byte pro Dreieck reduziert [4]. Neben einem reduzierten Speicherverbrauch bietet diese Technik noch zwei Geschwindigkeitsvorteile. Zum einen ist es möglich anstatt der geometrischen Positionen der Ecken nur deren Indizes zu vergleichen, z.B. wenn man benachbarte Dreiecke zu einem bestimmten Dreieck finden möchte. Zudem müssen bei Transformationen oder Modifikationen der Geometrie nur noch die geometrischen Informationen im Array transformiert werden. Würde man z.B. eine Ecke eines bestimten Dreiecks verschieben, müsste man alle Dreiecke finden, die sich diese Ecke teilen, und bei jedem dieser Dreiecke die Position anpassen. Genauso wird auch beim Transformieren eines kompletten Dreiecksnetzes verfahren. Anstatt die Transformation auf jedes Dreieck des Netzes anzuwenden, muss die Transformation nur einmal für jeden Vertex im Array angewandt werden. Bei

der *Shared-Vertex* hingegen referenzieren all diese Dreiecke auf eine Position in einer Eckenliste. So muss nur die Ecke in dieser Liste transformiert werden und alle Dreiecke verweisen auf die selbe Koordinate.

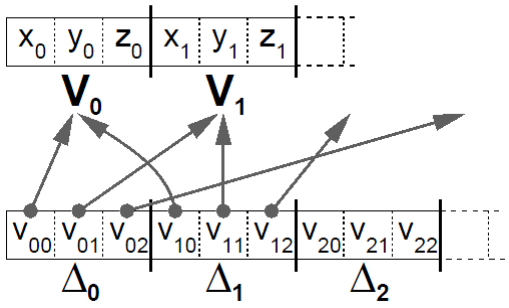

Abbildung 4: Verwaltung der *Shared-Vertex*-Datenstruktur mit Arrays. [4].

Die folgenden hier vorgestellten Datenstrukturen bauen auf der Technik der *Shared-Vertex*-Datenstruktur auf, indem auch sie die Geometrie von der Topologie trennen.

3.2 Winged-Edge

Um in konstanter Zeit auf alle Nachbarschaftsbeziehungen eines Polyeders zugreifen zu können, müssen mehr Informationen gespeichert werden. Dies ermöglicht die *Winged-Edge*-Datenstruktur, die 1972 von Baumgart entwickelt worden ist [1]. Dazu speichert sie zu jeder Ecke und Fläche zusätzlich eine anliegende Kante. Zu jeder Kante werden Referenzen zu den beiden Eckpunkten der beiden anliegenden Polygone, sowie der Vorgänger- und Nachfolgerkante, gegen den Uhrzeigersinn, der beiden angrenzenden Flächen gespeichert (siehe Abb.5).

Ein Nachteil dieser Repräsentation ist, dass die Kanten ungerichtet sind und dadurch nicht ersichtlich ist, in welchem Zusammenhang die aktuelle Kante genutzt wird. Um dies zu ermitteln muss immer eine zusätzliche Referenz zur aktuellen Kante angegeben werden [7]. Diese zusätzliche Referenz kann eine der beiden Flächen der Winged-Edge, einer der Eckpunkte oder die zuletzt besuchte Kante sein.

Der verwendete Speicherplatz der *Winged Edge* Datenstruktur ist mit etwa 60 bytes pro Dreieck [4] auch vergleichsweise hoch. Zudem bietet die *Winged-Edge*-Datenstruktur nicht die Möglichkeit, nicht-mannigfaltige Bereiche darzustellen.

3.3 Half-Edge

Die *Half-Edge*-Datenstruktur ähnelt der *Winged-Edge*-Datenstruktur und wurde 1988 von Mäntylä [8] entwickelt. Sie basiert allerdings nicht mehr auf ungerichteten Kanten, sondern auf gerichteten. Die ungerichteten Kanten werden durch zwei gerichtete Kanten namens *Half-Edges* (Halbkanten) ersetzt, wobei eine von der Startecke v^a zur Endecke v^b läuft und eine entgegengesetzt von v^b

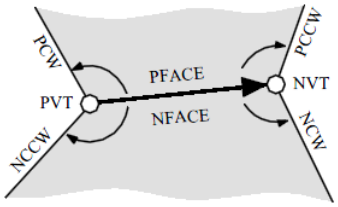

Abbildung 5: Die *Winged-Edge*-Datenstruktur [7].

nach v^a (siehe Abb. 6). Jede dieser Kanten enthält jeweils eine Referenz zu ihrer gegenüberliegenden Kante, die Zwilling genannt wird.

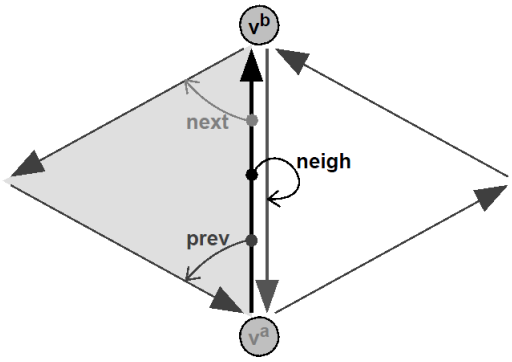

Abbildung 6: Die *Half-Edge*-Datenstruktur [4].

Jede *Half-Edge* repräsentiert mit ihrem Zwilling eine Kante im Polygonnetz. Jedoch ist eine *Half-Edge* immer nur derjenigen Fläche der ursprünglichen Kante zugeordnet, die sie gegen den Uhrzeigersinn umkreist. Dadurch löst die *Half-Edge*-Datenstruktur das Problem der *Winged-Edge*-Datenstruktur, dass eine Kante eine Referenz benötigt um ihre aktuelle Orientierung zu erhalten. Zu jeder *Half-Edge* gehören noch zwei Referenzen, damit ein Umkreisen der Flächen möglich ist, die Vorgänger *Half-Edge prev* und die Nachfolger *Half-Edge next*, die jeweils die nächste *Half-Edge* im Uhrzeigersinn und gegen den Uhrzeigersinn, die der selben Fläche zugeordnet sind, darstellen. Wie bei der *Winged-Edge*-Datenstruktur referenziert jede Ecke noch auf eine *Half-Edge*, die auf sie zeigt, und jede Fläche eine *Half Edge* die sie beschreibt. Im Vergleich mit der *Winged-Edge*-Datenstruktur bietet die *Half-Edge*-Datenstruktur einige Vorteile. Neben der schon erwähnten Möglichkeit, nur mit Referenzen auf die aktuelle *Half-Edge* das Polygonnetz abzuwandern, sind dabei auch keine Fallunterscheidungen zu machen, was sich in der Laufzeit bemerkbar macht [7].

4 Directed-Edges

In diesem Abschnitt wird die *Directed-Edges*-Datenstruktur aus dem Artikel *Directed Edges - A Scalable Representation for Triangle Meshes* [4] von Swen

Campagna et al. vorgestellt, die sich durch einen sehr geringen Speicherverbrauch und ihr Skalierbarkeit auszeichnet.

4.1 Aufbau

Die *Directed-Edges*-Datenstruktur, basiert auf der *Half-Edge*-Datenstruktur, beschränkt sich in ihrer Darstellung allerdings auf Dreiecksnetze. Die geometrischen Positionen der Ecken sind in einem Array zusammen mit einer Referenz zu einer benachbarten Kante gespeichert. Die einzelnen Dreiecke sind als drei gerichtete Kanten, hintereinander im Kantenarray, gespeichert (siehe Abb. 7 *full-sized*). Zu jeder gerichteten Kante werden der Start- und Endpunkt v^a, v^b als Referenz auf das Ecken-Array gespeichert. Dazu kommen die drei benachbarten Kanten e^{pv}, e^{nx} und e^{ng}. Die *Half-Edge* e^{pv} ist die Vorgängerkante der aktuellen Kante, deren Endpunkt bei v^a liegt, analog dazu gibt es die Nachfolger-*Half-Edge* e^{nx} deren Startpunkt bei v^b liegt. Die Kante e^{ng} ist das Komplement der aktuellen *Half-Edge*, die *Half-Edge*, die von v^b nach v^a läuft.

Die Dreicke, zu denen die einzelnen *Half-Edges* gehören, müssen nicht extra referenziert werden, sondern lassen sich dadurch, dass die drei *Half-Edges* eines Dreiecks hintereinander im Array liegen, berechnen, indem die Position der *Half-Edge* durch Drei geteilt wird.

4.2 Speicherplatz sparende Varianten

Um weiteren Speicherplatz zu sparen wird die *Directed-Edges* Datenstruktur um zwei weitere Modelle erweitert (siehe Abb. 7). In der *medium size*-Darstellung werden die Referenzen v^a und e^{nx} entfernt. v^a kann über v^b der Vorgängerkante angesprochen werden und e^{nx} durch zweimaliges Abwandern auf den Vorgängerkanten, da die Struktur auf Dreiecke beschränkt ist. Dadurch werden pro *Half-Edge* 8 byte an Speicher eingespart.

Um noch mehr Speicherplatz einzusparen gibt es noch die *small size* Konfiguration, in der zusätzlich die Referenz auf e^{pv} entfernt wird. Dies ist möglich, da die *Half-Edges* eines Dreiecks hintereinander im Array liegen und der Nachfolger sowie der Vorgänger berechnet werden können. Hier ein Beispiel für die Berechnung des Vorgängerknotens in Pseudocode.

Methode 1 Berechne Vorgänger

```
if e mod 3 = 0 then
  e ← e + 2
else
  e ← e − 1
end if
return e
```

4.3 Unterstützung für nicht-mannigfaltige Polygonnetze

Da es beim Designen von Objekten durch die Manipulation von Polygonnetzen passieren kann, dass das Objekt nicht-mannigfaltige Kanten oder Ecken enthält, ist es wünschenswert, dass eine Datenstruktur diese auch darstellen kann. Da davon auszugehen ist, dass ein Großteil des Polygonnetzes mannigfaltig ist,

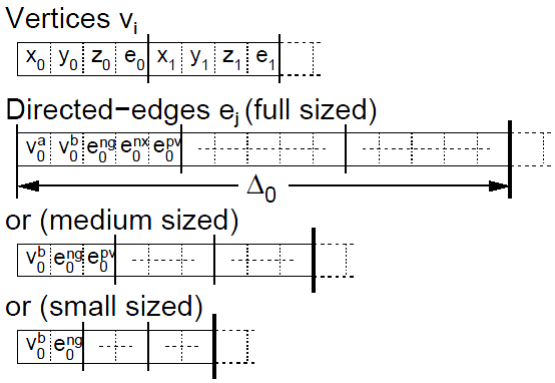

Abbildung 7: Die unterschiedlichen Varianten der *Directed-Edges*-Datenstruktur, mit den Positionen der Referenzen im Kanten-Array [4].

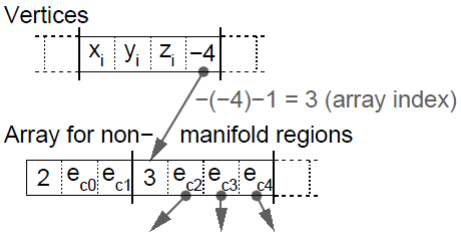

Abbildung 8: Verwaltung nicht mannigfaltiger Ecken [4].

werden die nicht-mannigfaltigen Bereiche durch einen negativen *integer*-Eintrag gekennzeichnet. Für nicht-mannigfaltige Kanten und Ecken wird jeweils ein eigenes Array angelegt. Die nicht-mannigfaltigen Regionen werden mit negativen Integer-Werten durchnummeriert, nach folgender Formel lässt sich dann die *Position* im Array berechnen.

$$-Eintrag - 1 = Position$$

Dies liegt daran, dass der Wert -1 bei Kanten reserviert ist um eine fehlende Komplementärkante zu beschrieben, wie es am Rand eines Objektes der Fall wäre. Bei nicht-mannigfaltigen Kanten wird die Referenz zur Komplementärkante e^{ng}, im Falle der nicht-mannigfaltigen Ecken die Referenz zu der ausgehenden Kante der Ecke als negative Position angegeben. In den Arrays selber folgen an der Position die verbundenen Bereiche, repräsentiert durch jeweils eine von der Ecke ausgehenden Kante, im Falle einer nicht-mannigfaltigen Ecke (vergl. Abb. 8), bzw. alle Kanten, die sich die selben Start- und Endpunkte teilen, im Falle einer nicht-mannigfaltigen Kante.

4.4 Performance

Als Performance-Test nutzten die Autoren einen Algorithmus zum Ausdünnen von Dreiecksnetzen. Dabei wurde gemessen, wieviele Dreiecke pro Sekunde ent-

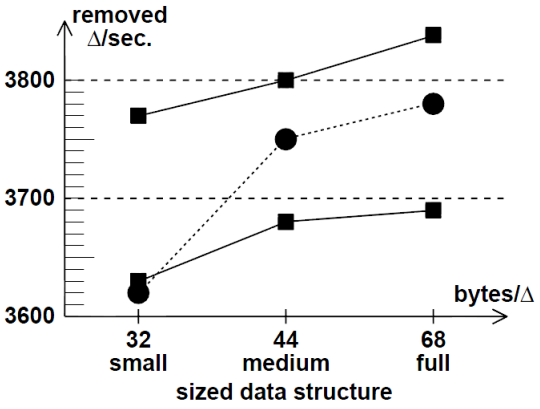

Abbildung 9: Die Messwerte des Performance-Test aus [4]. Die beiden durchgezogenen Linien sind die Ergebnise des Ausdünnens eines Polygonnetzes, das in zwei verschiedenen Ausgangsauflösungen bearbeitet wurde. Die vertikale Achse stellt dar, wieviele Dreiecke pro Sekunde entfernt wurden.

fernt wurden (Messwerte siehe Abb. 9). Der Anstieg in der Laufzeit liegt dabei beim Wechsel von der *small-size* zur *medium-size*-Konfiguration etwa zwischen $0,8\%$ und $3,5\%$ und von der *small-size* zur *full-size* zwischen $1,6\%$ und $4,14\%$. Wenn man dabei beachtet, dass der Speicherbedarf dabei um $37,5\%$ Prozent ansteigt, bzw. beim Wechsel von der *small-size* zur *full-size* sich mehr als verdoppelt, ist dieser Mehrwert in der Laufzeit zu vernachlässigen. Die Autoren erwähnen ausserdem, dass es Situationen geben kann, in denen die Messwerte genau andersherum aussehen können, d.h. dass die *small-size* eine bessere Laufzeit erzielen kann. Dies wäre dadurch möglich, dass die Zeit für die zusätzlichen Berechnungen geringer sein könnte, als die Zeit, die benötigt würde, um ausgelagerte Speicherzellen anzufordern.

5 Fazit

Die *Directed-Edges*-Datenstruktur ermöglicht es Dreiecksnetze sehr speichersparend zu verwalten, in der kleinsten Ausführung belegt sie zwar mehr Speicher als die *Shared-Vertex*-Datenstruktur, bietet verglichen mit ihr allerdings einen direkten Zugriff auf alle wichtigen Nachbarschaftsbeziehungen. Diese effiziente Darstellung ist dadurch möglich, dass diese Beziehungen berechnet werden und nicht explizit referenziert werden müssen. Dies ist in konstanter Zeit möglich, und nicht, wie im Falle der *Shared-Vertex* in der Zeit $\mathcal{O}(n^2)$, wenn man keine geeignete Beschleunigungsstruktur benutzt.

Im Vergleich der hier beschriebenen Datenstrukturen (siehe Tab.5) überzeugt die *Directed-Edges* durch ihren niedrigen Speicherverbrauch in der *small-size*-Konfiguration, sowie die Unterstützung für nicht-mannigfaltige Objekte. Im Gegensatz zur *Winged-Edge* ist ein effektiveres Navigieren auf dem Polygonnetz möglich, so wie bei der *Half-Edge*-Datenstruktur.

Daten Struktur	$^{\text{bytes}}/_{\Delta}$	nicht-mannigfaltige Objekte	benach-barte Flächen	Vor-gänger Kante	Nach-folger Kante
einzelne Dreiecke	36	Ja	$\mathcal{O}(n^2)$	mögl.	mögl.
Shared-Vertex	18	Ja	$\mathcal{O}(n^2)$	mögl.	mögl.
Winged-Edge	60	Nein	Ref.	Ref.	Ref.
Half-Edge	68	Nein	Ref.	Ref.	Ref.
Directed-Edges (gross)	68	Ja	Ref.	Ref.	Ref.
Directed-Edges (mittel)	44	Ja	Ref.	Ref.	$\mathcal{O}(1)$
Directed-Edges (klein)	32	Ja	Ref.	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Tabelle 1: Vergleich der unterschiedlichen Datenstrukturen zur Verwaltung von Polygonnetzen. In der Tabelle steht der Eintrag "Ref.ädafür, dass die Beziehung referenziert wird. Die Einträge "mögl.söllen andeuten, dass es möglich wäre die Datenstruktur so zu erweitern, dass die Beziehung in konstanter Zeit bestimmt werden kann (Tabelleneinträge sind aus [4] übernommen).

Die Implementation der *Directed-Edges*-Datenstruktur stellt sich zwar als sehr aufwendig heraus und sollte hinter einem objektorientierten Interface liegen, um bei der Einbindung komfortabler benutzt werden zu können, dieser Aufwand ist es allerdings wert, um speichersparend komplexe Operationen auf Dreiecksnetzen auszuführen.

Literatur

[1] Bruce G. Baumgart. Winged Edge Polyhedron Representation. Technical report, Stanford University, Stanford, CA, USA, 1972.

[2] Mario Botsch, Leif Kobbelt, Bruno Levy, Alliez, and Pierre. *Polygon Mesh Processing*. A K Peters, Natick, Mass, 2010.

[3] Mario Botsch, Mark Pauly, Christian Rossl, Stephan Bischoff, and Leif Kobbelt. Geometric Modeling Based on Triangle Meshes. In *ACM SIGGRAPH 2006 Courses*, SIGGRAPH '06, New York, NY, USA, 2006. ACM.

[4] Swen Campagna, Leif Kobbelt, and Hans-Peter Seidel. Directed Edges - A scalable Representation for Triangle Meshes. *Journal of Graphics Tools*, 3:1–11, 1998.

[5] André Guéziec, Gabriel Taubin, Francis Lazarus, and William Horn. Converting Sets of Polygons to Manifold Surfaces by Cutting and Stitching. In

Proceedings of the Conference on Visualization '98, VIS '98, pages 383–390, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[6] Philip M. Hubbard. Constructive Solid Geometry for Triangulated Polyhedra. Technical report, Brown University, Providence, RI, USA, 1990.

[7] Lutz Kettner. Using Generic Programming for Designing a Data Structure for Polyhedral Surfaces. *Comput. Geom. Theory Appl*, 13:65–90, 1999.

[8] Marti Mäntylä. *Introduction to Solid Modeling*. W. H. Freeman & Co, New York, NY, USA, 1988.

[9] Tomas Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering*. A.K. Peters, Wellesley, Mass, 3 edition, 2008.

[10] Joseph O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, NY, USA, 2 edition, 1998.

[11] Kevin Weiler. Edge-Based Data Structures for Solid Modeling in Curved-Surface Environments. *IEEE Comput. Graph. Appl*, 5:21–40, 1985.

[12] Tony Woo. A Combinatorial Analysis of Boundary Data Structure Schemata. *IEEE Comput. Graph. Appl*, 5:19–27, 1985.