Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Institut für Simulation und Graphik

G-Strokes

zur Verbesserung von Liniengrafik-Rendering

Diplomarbeit

Angela Brennecke

Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik

Institut für Simulation und Graphik

G-Strokes

zur Verbesserung von Liniengrafik-Rendering

Diplomarbeit

von:	Angela Brennecke
geb. am:	23. Januar 1978
in:	Braunschweig
Matrikelnummer:	15 53 93

1. Gutachter:	Dr.-Ing. Tobias Isenberg
2. Gutachter:	Prof. Dr.-Ing. Bernhard Preim
Betreuer:	Dr.-Ing. Tobias Isenberg

Zeit der Diplomarbeit:	01.04.2004 – 31.08.2004

Zusammenfassung

In dieser Diplomarbeit wird ein neues Konzept zur Verbesserung der analytischen Verfahren beim Liniengrafik-Rendering vorgestellt. Das entwickelte Konzept der G-Strokes basiert auf der Trennung von Geometrie und geometrischen Eigenschaften. Die Geometrie entspricht dabei den zu stilisierenden Kantenzügen, wohingegen die G-Strokes die jeweiligen Eigenschaften repräsentieren, z. B. die Sichtbarkeit der Kantensegmente. Durch diese Trennung ist eine neue Datenverwaltung möglich, die zu einer einfachen und einfach weiterzuentwickelnden Stilisierungs-Pipeline führt. Das hier vorgestellte G-Stroke-Konzept bietet dabei bereits eine Reihe von grundlegenden, die Geometrie beschreibenden Eigenschaften an, die für die Stilgebung des Strokes verwendet werden. Die Verwaltung der G-Strokes erfolgt dabei unabhängig vom restlichen Pipeline-System, was die Anwendung der Pipeline-Knoten und der einzelnen G-Strokes vereinfacht. Damit wird die Weiterentwicklung neuer Elemente und Stile unterstützt. Mit den hier entwickelten und eingesetzten Eigenschaften wurden bereits eine Reihe von Liniengrafiken generiert, die bisher nur unter hohem Aufwand hätten entstehen können.

Abstract

This thesis introduces a new concept for the improvement of object space algorithms concerning line drawings rendering. The developed concept of G-Strokes is based on the separation of geometry and geometric features. Edges to be stylized form the geometry objects whereas G-Strokes represent their respective features such as the visibility of edge segments. With this separation a new data management is possible, leading to a stylization pipeline that is simple and easily extendable. The G-Stroke concept introduced here already offers a range of basic, the geometry describing features that can be used for the stylization of the strokes. G-Strokes are managed independently of the rest of the pipeline system, simplifying the application of pipeline nodes and of single G-Strokes. By this, the further development of new elements and styles is supported. With the features developed and applied in this thesis, a number of line drawings can be generated. The effort to produce these drawings would have been much greater with common stylization pipelines.

Danksagung

Vielen Dank an alle, die mich während der letzten fünf Monate unterstützt haben.

Selbstständigkeitserklärung

Hiermit versichere ich, Angela Brennecke (Matrikel-Nr. 155 393), die vorliegende Arbeit allein und nur unter Verwendung der angegebenen Quellen angefertigt zu haben.

Magdeburg, Angela Brennecke

Inhaltsverzeichnis

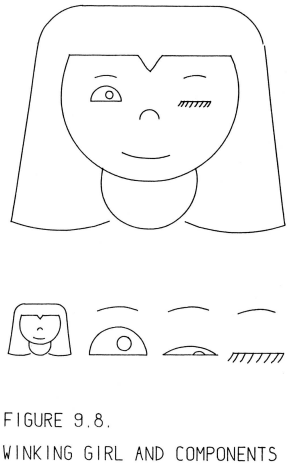
Kapitel 1

Einleitung

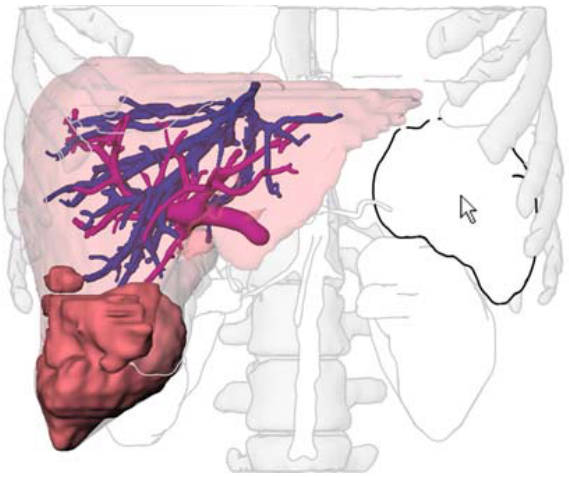
Auf dem Gebiet der Computergrafik hat sich insbesondere in den letzten fünfzehn Jahren ein neuer Forschungszweig entwickelt: Das *nicht-fotorealistische Rendering* (*Non-Photorealistic Rendering, NPR*) mit dem Ziel, gezeichnete oder künstlerisch attraktive Bilder am Computer zu erzeugen. Ganz im Gegensatz zu dem noch immer größeren Teilgebiet der Computergrafik, dem *fotorealistischen Rendering*, bei dem das computergenerierte Bild eine korrekte und fotorealistische Darstellung der Modellszene wiedergeben soll, wird beim nicht-fotorealistischen Rendering auf eine veränderte Darstellung der Objekte Wert gelegt. Das Ziel ist dabei einerseits die Simulation künstlerischer Techniken und Stile und andererseits die Erhöhung der Ausdruckskraft in computergenerierten Bildern. Dies spiegelt sich in einem sehr breit gefächerten Entwicklungsfeld wider, dessen gänzliche Beschreibung den Rahmen dieser Diplomarbeit sprengen würde. Da die Generierung nicht-fotorealistischer Liniengrafiken im Vordergrund dieser Arbeit steht, soll vor allem dieses Teilgebiet des NPR näher betrachtet werden.

Erste Versuche, computergenerierte Liniengrafiken zu erzeugen, gab es bereits in den sechziger Jahren. Das von SUTHERLAND (1963) entwickelte System SKETCHPAD ermöglichte u. a. einfache wissenschaftliche Zeichnungen am PC. Obwohl die Möglichkeiten von SKETCHPAD im Vergleich zu modernen Zeichen-Programmen rudimentär waren, begründete SUTHERLAND damit nicht nur die Weiterentwicklung computergenerierter wissenschaftlicher Zeichnungen und Illustrationen, sondern legte auch den Grundstein für interaktive grafische Systeme sowie die computergenerierte Animation, wie er mit dem „zwinkernden Mädchen“ unter Beweis stellte (SUTHERLAND, 1963, Kap. 9). Hierbei handelte es sich um die Liniengrafik eines Mädchengesichts, bei der durch das Austauschen der Grafik des einen Auges, in geschlossener, halboffener und offener Stellung, der Eindruck eines Zwinkerns entstand (vgl. Abbildung 1.1(a)).

Steckte die computergenerierte Animation gezeichneter Figuren damals noch in den Kinderschuhen, so ist sie spätestens heute, nach maßgeblichen Erfolgen von z. B. PIXARs und WALT DISNEYS „FINDET NEMO“, erwachsen und der Animationsfilm zu einer eigenen künstlerischen Ausdrucksform geworden. Auch auf dem Gebiet der wissenschaftlichen Illustration kann mittlerweile auf automatisierte Darstellungen zurückgegriffen werden. Insbesondere bei der medizinischen Visualisierung und beim Industrie- und Produktdesign kommen animierte und nicht-animierte Liniengrafiken u. a. zur Unterstützung der Übersichtlichkeit, zum Hervorheben von Objektdetails, in Form von Entwurfsskizzen oder als technische



(a) (SUTHERLAND, 1963)



(b) (TIETJEN, 2004)

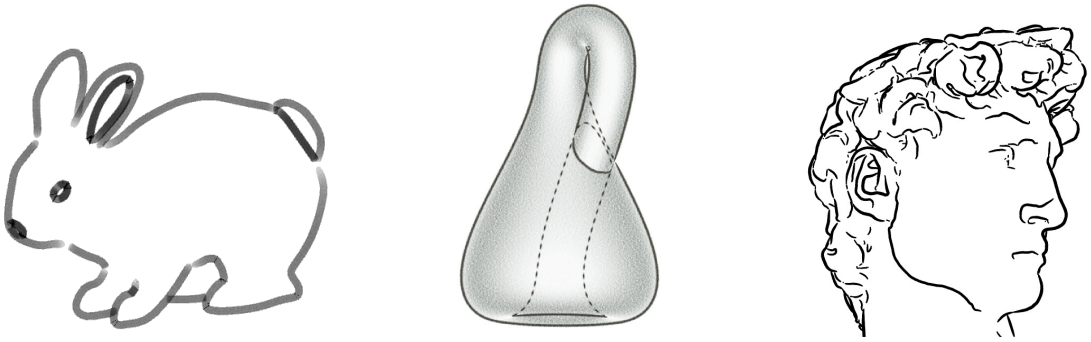
Abbildung 1.1: Zwischen diesen beiden computergenerierten Bildern liegt eine Zeitspanne von 41 Jahren: Abbildung (a) zeigt die Liniengrafik des zwinkernden Mädchens von SUTHERLAND. Die einzelnen Augenstellungen werden nacheinander gesetzt, sodass der Eindruck eines Zwinkerns entsteht. Abbildung (b) zeigt den unterstützenden Effekt von Silhouettenlinien bei der Visualisierung eines medizinischen Volumendatensatzes.

Illustrationen zum Einsatz (CSÉBFALVI et al., 2001; DONG et al., 2003; TIETJEN, 2004). Abbildung 1.1(b) zeigt ein derartiges Anwendungsbeispiel.

In dieser Diplomarbeit soll es um eine neue Entwicklung im Bereich der computergenerierten Liniengrafiken gehen, die sich auf die geometrischen Eigenschaften des Modells stützt, um nicht-fotorealistische Effekte zu erzielen.

1.1 Motivation und Ziele

Moderne Rendering-Systeme für animierte oder nicht-animierte nicht-fotorealistische Liniengrafiken von Polygonmodellen verfolgen in der Regel ein gemeinsames Konzept. Zunächst werden dabei die sichtbaren Silhouetten- und Merkmalskanten des Modells bestimmt. Sie sind von grundlegender Bedeutung für eine computergenerierte Liniengrafik, da mit ihnen bereits einfache Illustrationen erzeugt werden können (vgl. KALNINS et al., 2003; GOOCH et al., 1999; RASKAR und COHEN, 1999; ELBER, 1995). Um den Eindruck langer zusammenhängender Linien zu erhalten, werden die Kanten zu Linienzügen verbunden, den sogenannten *Strokes*. Diese Strokes werden dann von einer *Stilisierungs-Pipeline* verändert und so zum computergenerierten *Pendant* gezeichneter Striche (HALPER et al., 2003a; ISENBERG et al., 2002). Die Stilisierungs-Pipeline besteht dabei aus unterschiedlichen Elementen, welche z. B. den Kantenzug verändern, Segmente unterteilen oder mit Texturen belegen, um bestimmte stilistische Effekte zu erzielen. Abbildung 1.2 zeigt drei derartig erzeugte Liniengrafiken.



(a) (ISENBERG et al., 2002). (b) (KALNINS et al., 2003). (c) (DECARLO et al., 2003).

Abbildung 1.2: Verschiedene computergenerierte Liniengrafiken: (a) zeigt die Silhouettenkanten eines 3D-Hasen-Modells, (b) stellt die unterschiedliche Visualisierung sichtbarer und verdeckter Silhouettenkanten dar und (c) ist ein Beispiel für so genannte suggestive Konturen.

Die bisher genannten Ansätze besitzen jedoch den Nachteil, dass sie neben der Stilisierung gleichzeitig die Verwaltung aller Daten übernehmen müssen. Wird der Kantenzüg verändert, müssen alle von ihm abhängigen Eigenschaften mit angepasst werden, was die Entwicklung neuer Stile hemmt. Diese Arbeit schlägt daher vor, dass geometrische Eigenschaften zu einem eigenen Objekt erhoben werden – dem G-Stroke. Das Konzept der G-Strokes unterstützt die automatische Generierung computergenerierter Liniengrafiken, indem es die zu verarbeitenden Daten zunächst in die Primitive *Stroke* und G-*Stroke* trennt. Der G-Stroke repräsentiert dabei die in erster Linie geometrische Eigenschaft des Kantenzugs, anhand derer eine Stilgebung erfolgen kann. Desweiteren bietet das Konzept eine verbesserte Datenverwaltung und vereinfacht dadurch die Anwendung und Weiterentwicklung neuer Stile erheblich. Derzeit ist die Datenverwaltung der Stilisierungs-Pipeline sehr umständlich und damit ein entscheidender Faktor für die Behinderung neuer Entwicklungen im Bereich der computergenerierten Liniengrafiken. So erfordert das Hinzufügen neuer Datenstrukturen die Anpassung aller Pipeline-Elemente und das Darstellen unterschiedlicher Stile benötigt i. A. mehrere Rendering-Vorgänge (KALNINS et al., 2003; CURTIS, 1999; ROSSIGNAC und VAN EMMERIK, 1992).

Die G-Strokes bieten eine Lösung für diese Problematiken an. Sie werden hier in Anlehnung an das Konzept der G-*Buffer* von SAITO und TAKAHASHI (1990) entwickelt, sind jedoch im Gegensatz zu den G-Buffern als ein dynamischer Datentyp zu verstehen, die sich an die Geometrie des Strokes anpasst. G-Buffer werden i. A. dazu verwendet, dreidimensionale geometrische Informationen in einem zweidimensionalen Speicher zu sichern. Ein Beispiel für einen derartigen Buffer ist der z-Buffer, in dem der zu jedem Pixel gehörende z-Wert (Tiefenwert) der Szene gespeichert wird. Diese Information kann dann genutzt werden, nicht-fotorealistische Bilder zu erzeugen. Zum Beispiel werden Diskontinuitäten im G-Buffer zur Kantenbestimmung verwendet.

Im Unterschied zu den G-Buffern basiert das entwickelte Konzept der G-Strokes nicht auf Pixelspeichern. Zwar wird auch hier in erster Linie die geometrische Information des 3D-Modells genutzt, um nicht-fotorealistische Stile zu generieren. Ziel ist jedoch vor allem durch das Prinzip der G-Strokes eine Entkopplung von Stil-Pipeline und Stroke-Verwaltung zu ermöglichen. Ein G-Stroke repräsentiert dabei einerseits eine bestimmte Stroke-Eigenschaft, verwaltet diese aber andererseits selbstständig und unabhängig von der Stilisierungs-Pipeline. Das bedeutet, dass die Elemente der Stil-Pipeline die benötigten Eigenschaften abfragen, sogar neue Eigenschaften hinzufügen und Änderungen veranlassen können, die Durchführung aber vom G-Stroke-Konzept intern erfolgt.

Ein kurzes Beispiel soll an dieser Stelle motivierend die Ziele dieser Arbeit veranschaulichen: Die Strokes der Silhouettenkanten liegen in Koordinatenform vor. Um eine gleichmäßige Texturierung durchführen zu können, sollen sie parameterisiert werden. Dafür wird eine Parameter-Datenstruktur von einem Element der Stil-Pipeline erzeugt. Koordinaten- und Parameter-Daten (*K*- bzw. *P-Stroke*) werden nun an die Stil-Pipeline übergeben. Zunächst müssen sichtbare von nicht-sichtbaren Kanten unterschieden werden. Das heißt, der dafür zuständige Knoten muss K-Strokes und P-Strokes dementsprechend anpassen, also evtl. Kanten löschen oder kürzen. Wenn es nun einen „Sichtbar“-Stroke und einen „Verdeckt“-Stroke (*S*- und *V*-Stroke) geben soll, würden diese hier erzeugt werden. Im einfachsten Fall könnten die Kanten nun anhand der Stroke-Daten (Koordinaten und Parameter) texturiert und das Bild ausgegeben werden. Was passiert aber, wenn sie z. B. in Wellenform dargestellt werden sollen? Ein weiteres Stil-Element, dass die Kantenkoordinaten dazu leicht versetzen würde, müsste nun nicht nur die K- und P-Strokes anpassen, sondern auch S- und V-Strokes berücksichtigen. Da alle vier Stroke-Daten unterschiedliche Datentypen sein können, in diesem Beispiel Integer, Float und Boolean, müssen auch alle Stil-Elemente dementsprechend angepasst werden. Der Aufwand hierfür wächst mit jeder neuen Eigenschaft bei jedem weiteren Stil-Element.

Das Ziel des G-Stroke-Konzepts ist daher die interne Selbstverwaltung der Strokes, um die oben genannten Probleme zu verhindern. Die für die Strokes notwendigen Datenstrukturen sind begrenzt, ebenso die Operationen, die z. B. für das Einfügen, Entfernen oder Kürzen von Kanten notwendig sind. Es ist daher möglich, die expliziten Veränderungen, die die Stil-Pipeline z. B. beim Bestimmen sichtbarer und verdeckter Kanten durchzuführen hat, intern von den Strokes durchführen zu lassen. Eine derartige Entkopplung von Aufgaben löst nicht nur das Problem der aufwendigen Stroke-Verarbeitung, sondern fördert neben der Entwicklung neuer Stroke-Eigenschaften auch die Entwicklung neuer nicht-fotorealistischer Stile. Außerdem stellt sich an dieser Stelle die Frage, inwieweit geometrische Kanten-Eigenschaften verwendet und kombiniert werden können, um neue Stile zu erzeugen. Desweiteren wird es interessant sein herauszufinden, ob das G-Stroke-Konzept den Einsatz gezielter Vernachlässigungen und Hervorhebungen von Modelldetails bei computergenerierten Linienstil-Bildern ermöglicht.

1.2 Gliederung der Diplomarbeit

Die vorliegende Diplomarbeit gliedert sich neben diesem einleitenden in sechs weitere Kapitel. Das folgende Kapitel 2 führt zunächst in das Themengebiet der klassischen Darstellung von Objekten und Szenen in Form von Liniengrafiken ein. Es wird ein Überblick über die Zeichentechniken und -methoden zur Hervorhebung bestimmter Objekteigenschaften gegeben und diskutiert, wie Informationen zeichnerisch in einem Bild festgehalten werden können. Der zweite Teil des Kapitels beschäftigt sich im Anschluss daran mit der automatischen Generierung von Liniengrafiken auf dem Gebiet des nicht-fotorealistischen Renderings. Hier werden grundlegende Algorithmen und Methoden zur Kantenextraktion und Stilisierung vorgestellt und die Vor- und Nachteile der verschiedenen Verfahren beleuchtet.

Aufbauend auf den in Kapitel 2 vorgestellten Grundlagen beschäftigt sich das Kapitel 3 zunächst mit Problemen, die bei der automatischen Generierung von Liniengrafiken auftreten und noch nicht zufriedenstellend gelöst werden konnten. Hierbei geht es vor allem um die Datenverwaltung und -verarbeitung während des Rendering-Vorgangs. Der zweite Teil des Kapitels beschreibt den Entwurf für die in dieser Diplomarbeit entwickelte Lösung der Datenverarbeitungsprobleme. Dabei werden zuerst die verwendeten Datenprimitive beschrieben und definiert. Darauf aufbauend wird ein neues Verwaltungskonzept entworfen und vorgestellt. Beides wird im Anschluss für die Entwicklung einer ebenfalls neuen Stilgebung der dreidimensionalen Kantendaten eingesetzt.

An den Entwurf schließt sich die in Kapitel 4 beschriebene Umsetzung des neuen G-Stroke-Konzepts an. Zunächst werden die verwendeten Implementierungswerkzeuge und Entwurfsmuster beschrieben, die für eine enge Verbindung zwischen Entwurf und Realisierung elementar waren. Im weiteren Verlauf des vierten Kapitels werden dann die neu entwickelten Klassen vorgestellt. Unter Ausnutzung der Implementierungswerkzeuge wird eine sehr eng miteinander verbundene Beziehung der Klassen und ihrer Verwaltung ermöglicht, die sich an den Entwurf anpasst. Die enge Verwandtschaft zwischen entworfenem Konzept und entwickelten Klassen wird dabei mit Hilfe von Quelltext-Auszügen veranschaulicht.

Das Kapitel 5 illustriert schließlich die erfolgreiche Umsetzung des Konzepts anhand einer Reihe von Fallbeispielen. Dabei werden die visuell darstellbaren G-Strokes zunächst grafisch vorgestellt und ihre Anwendung in einem zweiten Schritt unter Ausnutzung der realisierten Methoden durch eine Reihe von automatisch erzeugten Liniengrafiken belegt.

Abschließend fasst Kapitel 6 die Ergebnisse dieser Diplomarbeit zusammen und hebt die wichtigsten Aspekte in einem kurzen Überblick hervor. Danach wird ein Ausblick auf mögliche zukünftige Erweiterungen und Anwendungen des hier entwickelten G-Stroke-Konzepts gegeben.

Kapitel 2

Grundlagen

Wie bereits in Kapitel 1 erwähnt, gehören Liniengrafiken zu den zentralen Zeichentechniken in der Illustration (HALPER et al., 2003b; STROTHOTTE und SCHLECHTWEG, 2002; GOOCH et al., 1999; ISENBERG, 1999; SCHÖNWÄLDER, 1997; HODGES, 1989). Insbesondere die Möglichkeit, Szenen und Objekte simplifiziert darzustellen, Details gezielt hervorzuheben und dabei trotzdem Umriss, Schattierung und Form wiedergeben zu können, machen diese Zeichentechnik nach wie vor zu einem mächtigen Werkzeug. Anwendungsgebiete finden sich fast überall, sei dies die technische und wissenschaftliche Illustration (HODGES, 1989; SCHUMPELICK et al., 2003), eine beliebige Bedienungsanleitung, Comics und Cartoons, die Verwendung von Skizzen z. B. bei Architekturentwürfen (STROTHOTTE et al., 1994), in der Lehre (DODSON, 1993; GORDON, 1989) oder Kunst (KOSCHATZKY, 1993), die Bandbreite ist groß.

Das vorliegenden Kapitel gliedert sich in zwei Teile. Abschnitt 2.1 führt grundlegend in das Gebiet der Zeichnung und Liniengrafik ein. Dazu werden u. a. verwendete Darstellungsstile bei Liniengrafiken vorgestellt und die Vorteile dieser Zeichentechnik gegenüber anderen Verfahren herausgearbeitet.

In Abschnitt 2.2 wird die automatische Erzeugung von Liniengrafiken als ein Teilgebiet des nicht-fotorealistischen Renderings vorgestellt. Desweiteren werden verwandte Verfahren und aktuelle Forschungsergebnisse erläutert. In erster Linie handelt es sich dabei um die G-Buffer-Technik, sowie um Kantenextraktions- und -stilisierungsverfahren.

2.1 Liniengrafiken

In diesem Abschnitt wird grundlegend in das Gebiet der Zeichnung und Liniengrafik eingeführt und seine Wichtigkeit und Wirkung bezogen auf unterschiedliche Anwendungsmöglichkeiten beleuchtet. Liniengrafiken verfügen über eine Reihe von Vorteilen gegenüber anderen Zeichenstilen, weshalb sie für die Computergrafik so interessant sind. Diese Vorteile sollen hier unter den folgenden zwei Fragestellungen näher herausgearbeitet und die Ergebnisse abschließend zusammengefasst werden: Was für Vorteile bieten Liniengrafiken im Gegensatz zu anderen Zeichentechniken und in welchem Zusammenhang sind diese Vorteile von Bedeutung?

2.1.1 Begriffsklärung

Um sich der Beantwortung ersterer Fragestellung anzunähern, ist eine Definition des Begriffs *Liniengrafik* ein erster Ansatzpunkt. Laut WORDNET (2000–2003) ist eine Liniengrafik definiert als:

Definition 2.1 *Eine Zeichnung der Kontur einer Form oder eines Objekts.*

Obwohl sehr kurz gehalten, enthält der Satz zwei entscheidende Informationen: Eine Liniengrafik gehört in den Bereich der Zeichnung und enthält als zentrales Zeichnungsprimitiv die Kontur. Anders formuliert heißt das, dass eine Liniengrafik durch gezeichnete Linien definiert ist, die den Umriss eines Objekt grafisch einfangen.

Im Gegensatz zur Malerei ist die Zeichnung laut KOSCHATZKY durch *„[. . .] klar gezogene und umgrenzte Formen, die unkörperlich jeder Illusion entgegengesetzt sind [. . .]“* charakterisiert, deren grundlegendes Gestaltungselement die Linie ist (KOSCHATZKY, 1993, S. 246). An anderer Stelle verdeutlicht er die Wichtigkeit der Linie, indem er feststellt, *„[. . .] dass nicht jeder Strich [. . .] als Zeichnung anzusprechen ist. Die Striche müssen [. . .] als lineare Gebilde begriffen werden können, Linien also sein, die befähigt sind, Bildmitteilungen auszusagen, kurzum: Ausdruck zu tragen*“ (KOSCHATZKY, 1993, S. 11). Auch DODSON bestätigt die Wichtigkeit der Linie, indem er feststellt, dass der oder die Zeichnende i. A. keine Dinge, sondern nur Linien zeichnen könne (DODSON, 1993, S. 16).

Hieraus folgt, dass Einsatz und Platzierung der Linien möglichst präzise und akzentuiert gewählt werden sollten, um einerseits eine klare *Umreißung* der Form erzielen und andererseits die gewünschte Aussage vermitteln zu können. Die Aufgabe des Zeichners ist, die wichtigsten geometrischen Merkmale des Objekts zu erkennen und diese so präzise wie möglich umzusetzen (DODSON, 1993). Das Gestaltungselement *Linie* ist dabei in Art und Stil abhängig von der gewünschten Bildaussage. Diese Festlegung ist entscheidend für alle weiteren Klassifiezierungen von Liniengrafiken.

Die Frage, warum Menschen überhaupt in der Lage sind, anhand von Strichen auf einem Blatt Papier auf einen real existierenden Gegenstand zu schließen, soll an dieser Stelle nur kurz angerissen werden. In der Realität können nur Flächen und keine Linien gesehen werden (KOSCHATZKY, 1993). Laut DODSON können *nur* Linien und keine Objekte gezeichnet werden. Dies scheint eine praktische Erklärung dafür zu sein, warum Menschen Linien zeichnen bzw. mit Linien Flächen einzufangen versuchen. Aus psychologischer Sicht bietet sich der algorithmische Ansatz von MARR zur Erklärung an. Er beschreibt den Wahrnehmungsprozess als einen stufenförmigen Vorgang. Ausgang ist das auf der Netzhaut entstehende Abbild der Realität. Laut MARR analysiert das visuelle System des Menschen zunächst die Intensitätswerte und identifiziert Kanten und Elementmerkmale. Das Ergebnis dieser ersten Stufe ist die *primäre Rohskizze*, die noch nicht bewusst wahrgenommen wird. Darauf folgt die *zweieinhalbdimensionale Stufe*, deren Ergebnis die Repräsentation und An-

ordnung der Flächen des Objekts ist. Ein letzter Verarbeitungsschritt führt dann schließlich zu der bewußten Wahrnehmung des dreidimensionalen Objekts (MARR, 1982).

Die psychologische Forschung brachte neben dieser weitere Theorien hervor, welche den menschlichen Wahrnehmungsprozess zu erklären versuchen. An dieser Stelle soll noch die *Gestaltpsychologie* erwähnt werden, bei der die Wahrnehmungsorganisation des Menschen im Vordergrund steht (GOLDSTEIN, 1997, Kap. 5). Dieser Theorie zufolge führt erst die Kombination und Organisation bestimmter Reize zur Wahrnehmung einer Szene oder eines Objekts. Die *Gestaltgesetze* repräsentieren hierbei die Regeln, nach denen der Mensch seine Wahrnehmung organisiert. Dabei werden z. B. ähnliche Elemente zu einheitlichen Gruppen zusammengefasst. Eine eindeutige Erklärung für den Vorgang der Wahrnehmung gibt es jedoch nicht (GOLDSTEIN, 1997). MARRs Theorie ist insofern interessant, als dass die primäre Rohskizze mit einer Liniengrafik verglichen werden könnte und es dem menschlichen Betrachter deshalb so leicht fällt, aus einer solchen Grafik einen Gegenstand zu erkennen. Nicht vergessen werden sollte jedoch auch, dass es dem Zeichner einer Liniengrafik nicht anders möglich ist, Flächen auf einem Blatt Papier einzufangen, als sie mit einer Linie zu begrenzen.

Linientypen

Laut Definition 2.1 sind die Hauptprimitive von Liniengrafiken Linien, die die Kontur des Objekts erfassen. Unter einer Kontur wird i. A. der Umriss des Objekts verstanden, der es von anderen abgrenzt. Um jedoch den plastischen Eindruck des zu zeichnenden Gegenstands zu verstärken, gibt es neben den *Konturlinien* weitere Linientypen, die bei Zeichnungen verwendet werden. KOSCHATZKY unterscheidet zwischen *Konturzeichnung* und *Binnenzeichnung*, wobei letztere nicht nur den äußeren Umriss wiedergibt, sondern auch die Oberflächenstruktur darstellt (KOSCHATZKY, 1993, S. 255 ff.). Er formuliert verschiedene Typen von *Binnenlinien*. Hier sollen jedoch synonym die geläufigeren Begriffe *Strukturlinien* und *Strukturzeichnung* eingeführt werden. Dies wird mit folgender Aussage gerechtfertigt: *„Die Binnenlinien haben die Struktur der sichtbaren Oberfläche des dargestellten Körpers oder Gegenstandes wiederzugeben“* (KOSCHATZKY, 1993, S. 262). Da auch diese Art der Zeichnung als Liniengrafik verstanden werden muss, soll die oben genannte Definition 2.1 wie folgt korrigiert werden:

Definition 2.2 *Eine Liniengrafik ist eine Zeichnung, die mit Hilfe von Kontur- und Strukturlinien Umriss und Oberflächenform eines Objekts wiedergeben kann.*

Diese zweite Definition wird von HODGES untermauert:

> *Taking advantage of what the human brain is used to seeing in nature, the artist can create perceptual illusions with line replacing continous tone. The quality of the line, used as outline or for modeling (shading), conveys information to the viewer about light, shade, and structure* (HODGES, 1989, S. 99).

Stil, Einsatz und Platzierung der Linie sind demnach neben oben genannten Gründen ausschlaggebend für die Vermittlung von Beleuchtung, Helligkeit und Struktur des Objekts. Zwischen den grundlegenden Linientypen wird in Anlehnung an die genannten Autoren wie folgt unterschieden:

Konturlinien werden Linien genannt, die die äußere Form des Objekts eindeutig identifizieren und es so vom Hintergrund und anderen Objekten trennen. Synonym wird hier auch der Begriff *Silhouette* verwendet (KOSCHATZKY, 1993, S. 262).
Laut den Gestaltgesetzen wird die Kontur immer als „zum Objekt gehörend“ betrachtet (WEBER, 1990; KOSCHATZKY, 1993). Die klare Abgrenzung des Objekts von seiner Umwelt ermöglicht außerdem, es in Relation zu anderen Objekten setzen und eine Beziehung zu ihnen erzeugen zu können (SCHÖNWÄLDER, 1997).

Innere Linien werden Linien genannt, die markante Oberflächenmerkmale darstellen. Sie gehören zu den oben formulierten *Strukturlinien* und markieren prägnante Strukturänderungen. Hierzu zählen z. B. Falten und scharfe Kanten (KOSCHATZKY, 1993). Ohne innere Linien wäre es schwer möglich, einen auf Papier gezeichneten Würfel von einem Sechseck zu unterscheiden.

Schraffurlinien werden Linien genannt, die die Krümmung des Objekts und die Beleuchtung der Szene wiedergeben sollen. Sie gehören ebenfalls zu den *Strukturlinien* und sind im Gegensatz zu den *inneren Linien* für die Darstellung „weicher“ Änderungen auf der Oberfläche zuständig (DODSON, 1993).

Die Differenzierung in Umrisslinien und volumengebende Schraffurlinien ist sehr wichtig für das Verständnis der im folgenden beschriebenen Zeichenstile. Abbildung 2.1 illustriert die verschiedenen Linientypen an einem Beispielobjekt.

(a) Konturlinien. (b) Innere Linien. (c) Schraffurlinien.

Abbildung 2.1: Linientypen am Beispiel einer Vase, in Anlehnung an (DODSON, 1993, S. 53).

2.1.2 Techniken und Stile

In Abschnitt 2.1.1 ging es um die Klarstellung, *was* eine Liniengrafik charakterisiert. In diesem Abschnitt liegt der Fokus darauf, *wie* die Primitive einer Liniengrafik dargestellt werden können. Die Art der Darstellung kann im Bild z. B. eine bestimmte Aussage oder Stimmung vermitteln. Geometrische Informationen wie die Beleuchtung oder der Malstil eines Künstlers können ebenfalls simuliert werden. Insbesondere der letzte Punkt bietet eine enorme Vielfalt an Stilen und Stilvariationen, auf die hier nicht im Detail eingegangen werden kann. Vielmehr sollen die grundlegenden Stile vorgestellt werden, deren Variationen für neue Ausdrucksmöglichkeiten genutzt werden können.

Soll die äußere Form eines Objekts möglichst genau und akurat dargestellt werden, ist eine Zeichnung bestehend aus Konturlinien und inneren Linien in den meisten Fällen eine einfache und ausreichende Darstellungsform. Die Stärke der gezeichneten Linien kann dabei so variiert werden, dass das Bild über zusätzliche Informationen bzgl. Szene und Objektrelationen verfügt. Laut HODGES können u. a. folgende Wirkungen erzielt werden (HODGES, 1989, S. 99):

Lichteinfall: Hellere Bereiche können durch dünnere, dunklere Bereiche durch dickere Linien vermittelt werden.

Tiefe: Eine mit zunehmender Entfernung zum Betrachter abnehmende Linienstärke erweckt den Eindruck von Tiefe. Dickere Linien erscheinen näher als dünnere.

Kontextabhängigkeit: Leichtere Details können durch dünnere, schwerere Details durch dickere Linien dargestellt werden. HODGES nennt hier als Beispiel die Zeichnung eines Insekts, dessen Flügel dünner als dessen Beine gezeichnet werden sollten.

Relation zu anderen Objekten: Dickere Linien können Kontakt zur Umgebung, z. B. zum Boden, auf dem das Objekt liegt, herstellen, wohingegen dünnere Linien freiliegende Konturen beschreiben.

Diese Techniken unterstützen vor allem den *globalen* räumlichen Eindruck der Szene (vgl. Abbildung 2.2). Soll neben der Kontur auch noch die Oberflächenbeschaffenheit dargestellt werden, so muss verstärkt mit Schraffurlinien gearbeitet werden. Es gibt hierfür andere Stile, die in erster Linie die geometrische Form des Objekts *lokal* hervorheben. HODGES nennt u. a. folgende[1] (HODGES, 1989, S. 100 f, S. 109 ff.):

Snodgrassing: Eine Technik nach ROBERT E. SNODGRASS, bei der Linienknotenpunkte ausgeprägt gezeichnet werden, um Form und Kontrast zu erhöhen.

Schraffur (*Hatching*): Hierbei werden parallele Striche nebeneinander gezeichnet, sodass Krümmung oder Lichteinfall beschrieben werden. Weitere Schraffur-Techniken sind die

1 Für einige Fachbegriffe fehlt die passende deutsche Übersetzung, weshalb der Originalbegriff belassen wurde.

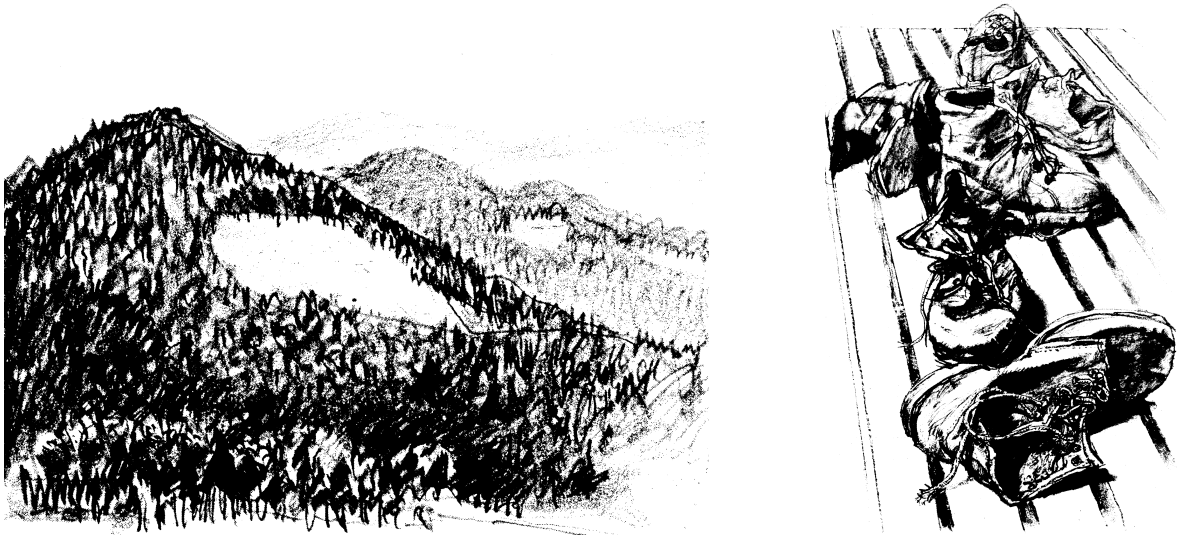


Abbildung 2.2: Die Linienstärke beeinflusst Tiefenwirkung und Lichteinfall sowie Kontrast, Kontext und Objektrelationen, Abbildungen aus (DODSON, 1993, S. 131,133).

Kreuzschraffur (*Crosshatching*), bei der die Striche parallel und gekreuzt verlaufen und das

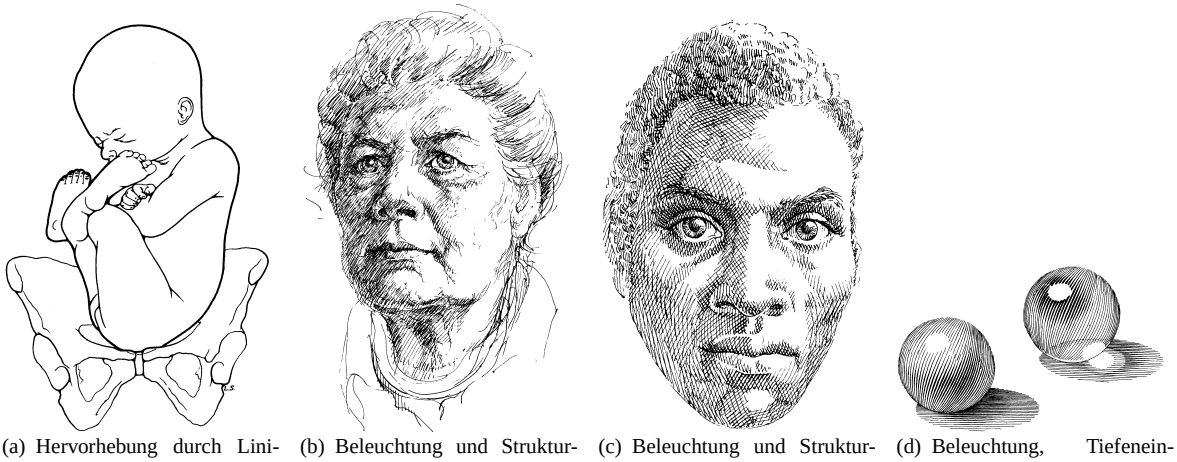
Eyelashing, bei dem die Linienstärke in Abhängigkeit von Krümmung oder Lichteinfall variiert wird.

Davon abgesehen existieren weitere Stilvariationen, die i. d. R. Kombinationen der bereits genannten Stile sind. In Abbildung 2.3 sind die genannten Stile und die durch sie hervorgerufenen Effekte dargestellt. Tabelle 2.1 fasst die Zeichenstile der Liniengrafiken abschließend zusammen.

Zeichenmedien und Zeichenwerkzeuge

Neben den Zeichentechniken können auch die Zeichenmedien und -werkzeuge zur Wirkung des Bildes beitragen. Die Umsetzung bestimmter Aufgaben erfordert die Anwendung dazu passender Zeichenwerkzeuge. Die Linienzüge einer technischen Illustration werden z. B. nicht verschwommen, sondern exakt und gerade dargestellt. Kreide oder ein weicher Pinsel könnten daher das falsche Werkzeug sein, um eine klar umrissene Zeichnung anzufertigen.

Die Palette der verwendeten Zeichenmedien reicht von Bleistift- bis zu Pinselzeichnungen. Federhalter und Tusche (*Pen and Ink*) gehören z. B. zu den wichtigsten Utensilien eines Illustrators, da sie eine große Vielfalt an Linientypen und -stilen ermöglichen. So kann z. B. die Linienbreite sehr leicht durch Druck variiert und somit „*feine Texturen und scharfe Details*“ erzeugt werden (DODSON, 1993, S. 63). HODGES zählt die Federzeichnung zu den herausfordernsten Techniken, die ein Illustrator beherrschen sollte. Diese könne bei Umriss-



(a) Hervorhebung durch Linienstärke, Formgebung durch Snodgrassing, aus (HODGES, 1989).

(b) Beleuchtung und Strukturwirkung durch Schraffur, aus (GORDON, 1989).

(c) Beleuchtung und Strukturwirkung durch Kreuzschraffur, aus (GORDON, 1989).

(d) Beleuchtung, Tiefeneindruck und Formgebung durch Eyelashing, aus (HODGES, 1989).

Abbildung 2.3: Verschiedene Zeichenstile und ihre Wirkung.

zeichnungen durchaus einfach sein, sollten jedoch Form und Tönung berücksichtigt werden, erhöhen sich die Anforderungen an den Zeichner erheblich (HODGES, 1989, S. 89).

Oft ist die Aussage und Stimmung der Liniengrafik abhängig vom verwendeten Zeichenmedium und dem verwendeten Zeichenstil. So erscheinen die meisten Kreide-Zeichnungen weicher und unpräziser als Feder-und-Tusche-Zeichnungen (DODSON, 1993), weshalb letztere in der technische Illustration bevorzugt werden. Bei Skizzen erwecken sie jedoch den Eindruck einer frei interpretierbaren Grafik und beflügeln die Fantasie des Betrachters (vgl. Abbildung 2.4).

Der Betrachter kann Form und Ausmaß sowie die fehlenden Flächen des gezeichneten Objekts abhängig von seiner subjektiven Vorstellung in die Skizze interpretieren (STROTHOTTE et al., 1994). HALPER et al. untersuchten den Zusammenhang von Zeichnung und Stimmung unter psychologischen Aspekten. Sie bestätigen, dass insbesondere der Linienstil bestimmte Reaktionen beim Betrachter auslöst, weniger das dargestellte Objekt (HALPER et al., 2003b). Abbildung 2.5 zeigt hierfür ein Beispiel.

2.1.3 Anwendungsgebiete

Liniengrafiken kommen meistens dort zum Einsatz, wo ein Sachverhalt oder eine Geschichte durch wenige, dafür aber akzentuiert gesetzte Linien, eindeutig visualisiert werden muss. Der Betrachter soll ohne Umschweife auf die Aussage des Bildes hingewiesen werden. Dies beinhaltet jedoch nicht, dass das Gezeichnete nicht interpretiert werden darf oder einem real existierenden *Etwas* entsprechen muss (vgl. hierzu Abschnitt 2.1.1). DURAND (2002) unterscheidet zwischen dem *Abbild* (*image*), das durch optische Akkuratesse bzgl. des darzustel-

Stilmittel	Fördert Wirkung von					Beispiel
	Umriss	Oberfläche	Tiefe	Beleuchtung	Kontrast	
Linienstärke	✓	–	✓	✓	✓	
Snodgrassing	✓	–	–	–	✓	
Schraffur	–	✓	✓	✓	✓	
Kreuzschraffur	–	✓	✓	✓	✓	
Eyelashing	–	✓	✓	✓	✓	

Tabelle 2.1: Zeichenstile im Überblick.

lenden Objekts charakterisiert ist, und der *Darstellung* (*picture*), die vielmehr eine freie und subjektive Repräsentation des darzustellenden Objekts ist. Die Liniengrafik fällt folgerichtig in den Bereich der Darstellung (DURAND, 2002, Abschnitt 2).

Exemplarisch eignet sich die wissenschaftliche Illustration, die Vorteile von Liniengrafiken zu erläutern. Dort geht es in erster Linie um eine präzise Darstellung eines Objekts und eines Sachverhalts. Dieser wird von einem Autor beschrieben und mit Hilfe einer Zeichnung illustrativ vermittelt (HODGES, 1989). Abhängig vom gegebenen Kontext werden dabei Details hervorgehoben oder vernachlässigt, um die Aufmerksamkeit auf die gewünschten Stellen lenken zu können. Die Grafiken müssen dabei nicht realitätsgetreu sein, sie sollten jedoch eindeutig interpretierbar sein. Der Illustrator macht sich die Freiheit der Zeichnung zu Nutze, um eine bestimmte Information zu vermitteln (vgl. Abbildung 2.6).

Diese Eigenschaften sind übertragbar auf jegliche Liniengrafiken. Einzig die festgelegte Interpretierbarkeit ist für die Illustration bindend. Andere Anwendungsgebiete machen sich Liniengrafiken zu Nutze, um dem Betrachter verstärkt eine freiere Interpretierbarkeit der Objekte oder Flächen bezogen auf seine subjektive Vorstellung zu ermöglichen (CURTIS, 1998). Die Freiheit der Zeichnung ermöglicht es desweiteren, auch nicht existierende Dinge zeichnerisch darzustellen, wie Fabelwesen, Comic-Figuren oder Karikaturen. Letztere heben Merkmale des Karikierten in einem Übermaß hervor, welches in der Form nicht real ist.

Ein weiteres Beispiel für die Darstellung nicht realer Dinge sind Zustände. So wird z. B. die Bewegung von Objekten durch sogenannte *Geschwindigkeitslinien* visualisiert, Dampf wird durch senkrecht geschlängelte Linie dargestellt (MCCLOUD, 1993; MASUCH et al.,

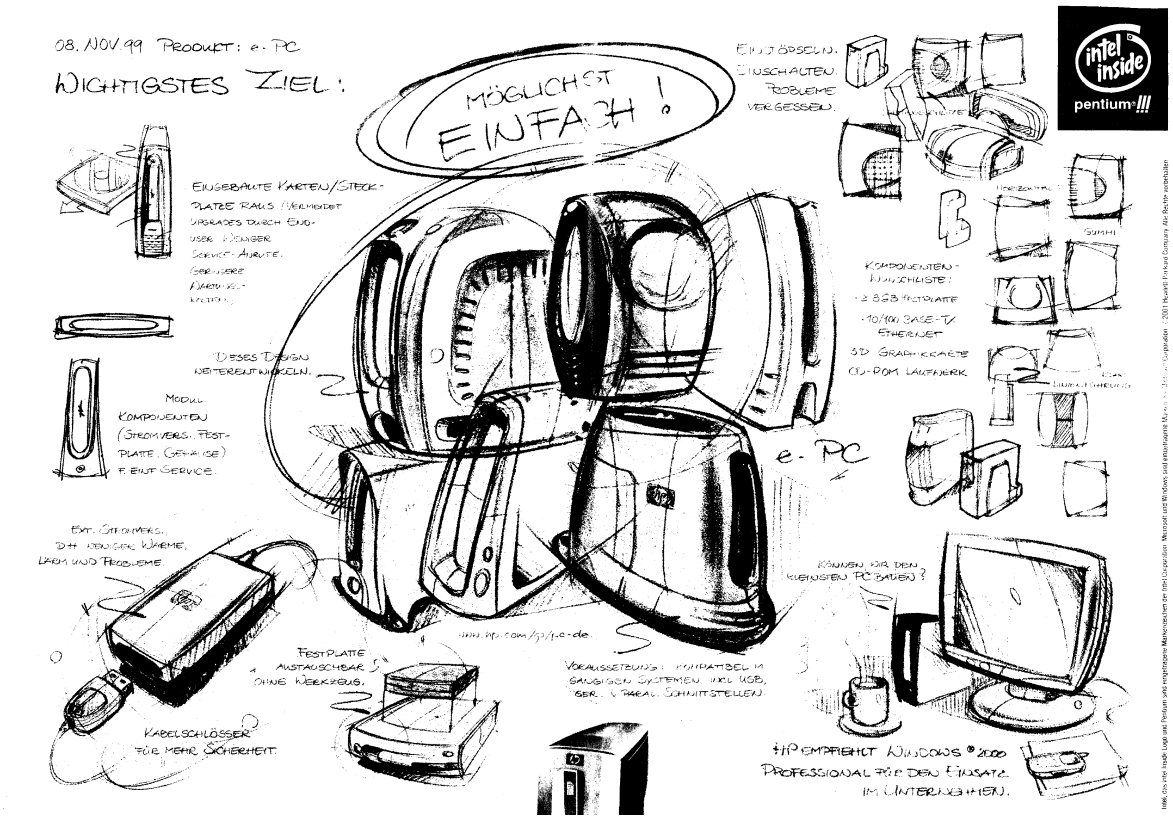


Abbildung 2.4: Diese Werbeanzeige von HEWLETT PACKARD© macht sich die freie Interpretierbarkeit der Skizze zu Nutze, um möglicherweise die persönliche Note ihres Produkts zu unterstreichen.

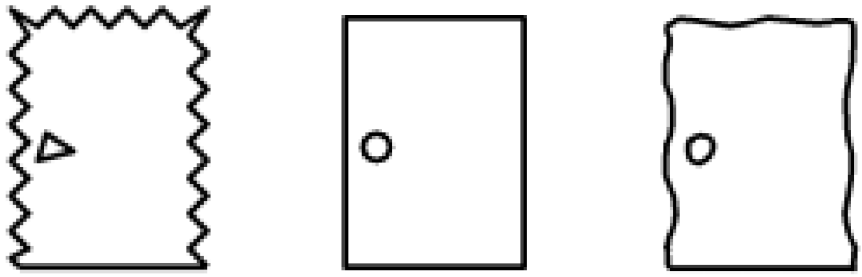


Abbildung 2.5: Linienstile beeinflussen die wahrgenommene Stimmung des Bildes. HALPER et al. stellten u. a. fest, dass die meisten Probanden auf die Frage hinter welcher Tür sich Gefahr verberge, die linke wählten. Abbildungen aus (HALPER et al., 2003b).

1999). Begünstigt wird die Verwendung von Liniengrafiken zusätzlich von der leichten und hochwertigen Reproduzierbarkeit im Gegensatz zu „aufwändigeren“ Bildern, u. a. (SCHÖNWÄLDER, 1997, S. 7). Abbildung 2.7 zeigt drei weitere liniengrafische Beispiele.

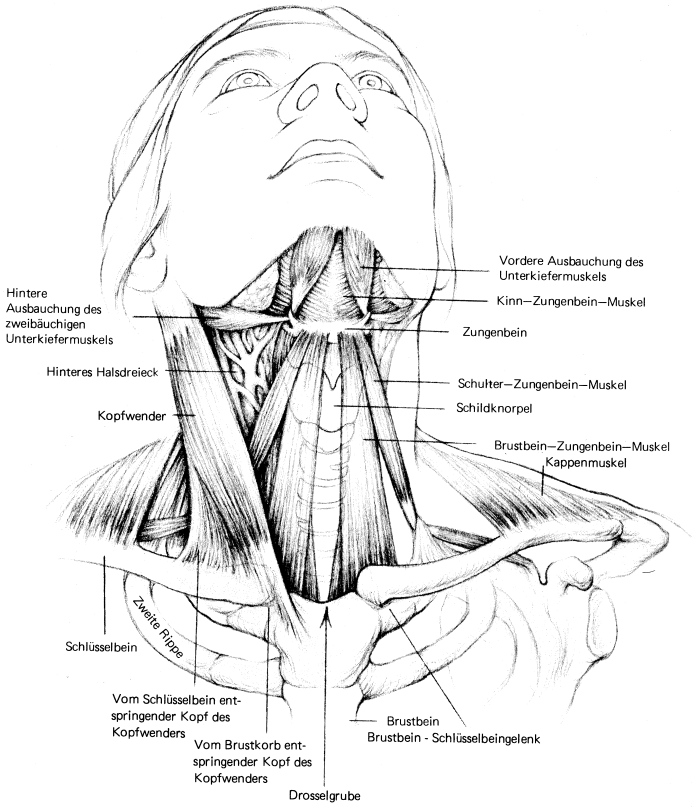


Abbildung 2.6: Bei dieser Illustration hat sich der Künstler die Freiheit der Zeichnung zu Nutze gemacht, indem die Halsmuskulatur innerlich, das Gesicht jedoch äußerlich dargestellt wurde.

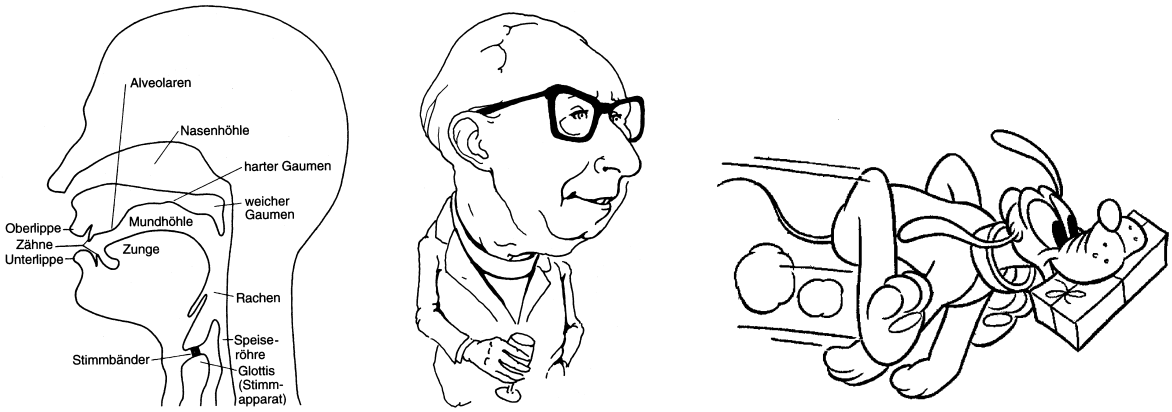


Abbildung 2.7: Drei weitere exemplarische Liniengrafiken. Die linke Abbildung zeigt eine medizinische Illustration des Nasen-Rachen-Raums (GOLDSTEIN, 1997). Die mittlere Abbildung ist ein Beispiel für eine Karikaturzeichnung, bei der die *Snodgrassing*-Technik verwendet wurde (GORDON, 1989) und die rechte Abbildung zeigt die Comic-Figur PLUTO©, deren Bewegung durch Geschwindigkeitslinien visualisiert wird (© WALT DISNEY).

2.1.4 Zusammenfassung

In den letzten Abschnitten wurde der Begriff *Liniengrafik* definiert und es wurden verschiedene Anwendungs- und Ausdrucksmöglichkeiten unter unterschiedlichen Gesichtspunkten vorgestellt und beleuchtet. Hier werden nun abschließend die Vorteile der Liniengrafiken unter Bezug auf die Fragestellungen zu Beginn des Abschnitts 2.1 zusammengefasst.

Die Liniengrafik ist eine spezielle Technik im Bereich der Zeichnung. Sie ist dadurch charakterisiert, dass sie die Formen von Objekten klar und eindeutig mit wenigen präzise gesetzten Linienzügen wiedergibt. Der Stil dieser Linien ist dabei frei wählbar und soll die zu vermittelnde Aussage des Bildes einerseits und die subjektive Interpretierbarkeit des Betrachters andererseits unterstützen. Durch die Stilisierung der Linie ist es möglich, der Zeichnung zusätzliche Informationen hinzuzufügen, wie z. B. den Eindruck von Tiefe oder eine bestimmte Stimmung zu erwecken (vgl. Abschnitt 2.1.2 und Abschnitt 2.1.2).

Diese Vorteile sind genau dann von Interesse, wenn in einem bestimmten Kontext zur Erleichterung des Verständnisses Kontextdetails visualisiert werden sollen. Diese können im Gegensatz zum restlichen Kontext hervorgehoben oder vernachlässigt werden. Die Unmittelbarkeit dieser Zeichentechnik, also die einfache Möglichkeit der Umsetzung, eröffnet desweiteren einen sehr leichten Zugang zur Zeichnung und gleichzeitig zur Beschreibung. Ein Comic ist z. B. nichts weiter als eine illustrierte Erzählung. Liniengrafiken sind folglich ein ideales Mittel, Textinformation in einer einfach umsetzbaren, einfach reproduzierbaren und künstlerisch attraktiven Art grafisch zu unterstützen. Keine andere Zeichen- oder Maltechnik ist in der Lage, mit einem einzigen Gestaltelement Kontext- und Objektinformationen zu visualisieren und dem Bild dabei Ausdruck und Wirkung zu verleihen.

2.2 Computergenerierte Liniengrafiken

In diesem Abschnitt wird dem Leser ein Überblick über die vergangene und aktuelle Forschung auf dem Gebiet der computergenerierten Liniengrafiken gegeben. Es wird dabei mit einer kurzen Einleitung über Liniengrafiken als nicht-fotorealistische Technik begonnen, an die sich die Definitionen einiger grundlegender Begriffe aus diesem Bereich anschließt. Danach werden aktuelle Verfahren zur Erzeugung von Liniengrafiken am Computer vorgestellt und untersucht. Abschließend werden die genannten Arbeiten dann mit ihren Vor- und Nachteilen bewertend zusammengefasst, um so eine Ausgangslage für neue Entwicklungen zu schaffen.

2.2.1 Liniengrafiken als nicht-fotorealistische Technik

Die Entwicklungen auf dem Gebiet des Nicht-Fotorealismus schlagen sich bisher hauptsächlich in der Veröffentlichung von Arbeiten und Berichten nieder. Bedenkt man, dass es erst zwei Bücher zu diesem Themenkomplex gibt, *Non-Photorealistic Rendering* von GOOCH und GOOCH (2001) und *Non-Photorealistic Computer Graphics. Modeling, Rendering, and Animation* von STROTHOTTE und SCHLECHTWEG (2002), so wird deutlich, dass es sich trotz der bereits in der Einleitung genannten Erfolge und zahlreicher Veröffentlichungen um ein sehr junges Forschungsgebiet handelt.

Neben der automatischen Generierung von Liniengrafiken beschäftigt sich die nicht-fotorealistische Forschung insbesondere mit der Bereitstellung von Werkzeugen zur Umsetzung künstlerischer Techniken. So können z. B. interaktiv oder automatisch bestimmte Kunststile nachgeahmt, Video-Mosaike erzeugt oder Verzerrungstechniken zur Informationshervorhebung angewendet werden (vgl. STROTHOTTE und SCHLECHTWEG, 2002). In der Regel wird dabei entweder mit bildbasierten Daten gearbeitet, z. B. wenn eine Fotografie derart verändert werden soll, dass sie einem Gemälde gleicht (MEIER, 1996; DECARLO und SANTELLA, 2002) oder nicht-fotorealistische Stile werden unter Ausnutzung der geometrischen Eigenschaften des 3D-Modells erzeugt (z. B. KOWALSKI et al., 1999). In dieser Diplomarbeit geht es um Verbesserungen bei der automatischen Generierung von Liniengrafiken, weshalb im folgenden vor allem jene Verfahren vorgestellt werden, die Liniengrafiken auf Grund von geometrischen Eigenschaften des Modells generieren können.

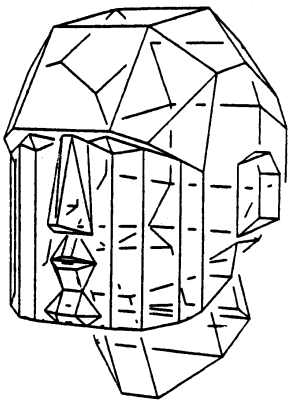
Der große Vorteil nicht-fotorealistischer Darstellungen liegt in erster Linie in der Möglichkeit begründet, Informationen in einem Bild kodieren zu können. Diese sind z. B. durch Polygonmodelle leicht zugänglich und müssen lediglich algorithmisch umgesetzt bzw. herausgefiltert werden. Eine Möglichkeit hierzu stellten STROTHOTTE et al. (1994) mit dem SKETCHRENDERER unter Beweis. Das Programm dient dazu, skizzenhafte Architekturzeichnungen anzufertigen. Die Skizzenartigkeit ist besonders beim Entstehungsprozess hilfreich, da der Betrachter den Entwurf weiterhin subjektiv interpretieren kann (vgl. Abschnitt 2.1.3). Fotorealistische Darstellungen eignen sich nur bedingt zu derartiger Informationsmitteilung. LANSDOWN und SCHOFIELD (1995) verweisen in diesem Zusammenhang auf das in der Literatur gängige Beispiel einer Bedienungsanleitung. Sie ist in den meisten Fällen nur in Form einer Illustration, nicht aber in Form einer Fotografie nützlich.

Rückblickend war es dabei auf dem Gebiet der Computergrafik nicht nur SUTHERLAND, der die Entwicklung computergenerierter Liniengrafiken vorantrieb. Hier ist vor allem auch APPEL zu nennen, der sich mit dem Entfernen verdeckter *Kanten* in 3D-Modellen beschäftigte, um Linienzüge hervorheben zu können (APPEL, 1967). Sein Verfahren wurde durch den Wunsch beflügelt, sichtbare und verdeckte Kanten unterschiedlich darzustellen und somit einfache Liniengrafiken zu erzeugen. Für jede Polygonkante wurde dafür die quantitative Unsichtbarkeit (***Q**uantitative **I**nvisibility*) bestimmt, also die Anzahl der sichtbaren Polygo-

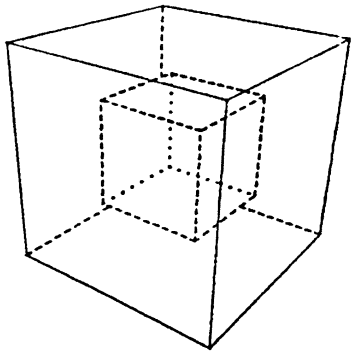
ne zwischen einem zu zeichnenden Kantenpunkt und dem Betrachter. Kantensegmente mit einem *QI* von Null sind dabei sichtbar. Da der Algorithmus zunächst die sichtbaren Silhouettenkanten extrahiert, ist er auch heute noch von Bedeutung und wurde hierfür u. a. von MARKOSIAN et al. (1997) in leicht veränderter Form verwendet.

Diese frühen Arbeiten beschäftigten sich vornehmlich mit der Liniengrafik-Erzeugung an sich. APPELs Veröffentlichung über den Lichthofeffekt (*Haloed Line Effect*) mehr als zehn Jahre später, zeigte dann eine erste algorithmische Umsetzung eines Zeichenstils (APPEL et al., 1979). Dabei werden verdeckte sowie sichtbare Kanten gezeichnet. Allerdings werden überall dort Lichthöfe (*Halos*) eingefügt, wo sichtbare von verdeckten Kanten geschnitten werden. Eigentliche Ursache für die Entwicklung dieses Verfahrens war der hohe Rechenaufwand, der beim Entfernen verdeckter Kanten entstand und geschickt eingespart werden sollte.

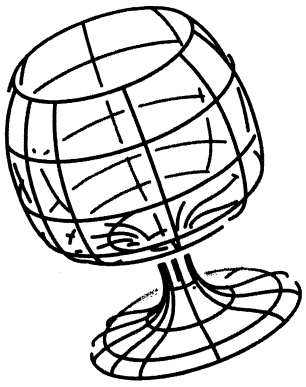
In diesem Zusammenhang sind auch KAMADA und KAWAI (1987) sowie ELBER (1995) zu nennen, die ebenfalls an der unterschiedlichen Darstellung verdeckter und sichtbarer Kanten arbeiteten. Erstere verfeinerten APPELs Arbeit über den Lichthofeffekt, indem sie den verdeckten Linien Attribute, u. a. in Abhängigkeit von der Anzahl der verdeckenden Flächen, zuwiesen. ELBER beschäftigte sich mit verschiedenen computergenerierten Linienstilen allgemein, wobei er auch den Lichthofeffekt umsetzte. Abbildung 2.8 veranschaulicht Ergebnisse aus diesen drei Arbeiten.



(a) (APPEL et al., 1979).



(b) KAMADA und KAWAI (1987).



(c) ELBER (1995).

Abbildung 2.8: Unterschiedliche Darstellung sichtbarer und verdeckter Kanten: Abbildung (a) verdeutlicht den Lichthofeffekt von APPEL. Abbildung (b) zeigt die unterschiedliche Darstellung von Kanten abhängig von der Anzahl der verdeckenden Flächen. Kanten, die von einer Fläche verdeckt werden, sind gestrichelt gezeichnet, Kanten, die von zwei Flächen verdeckt sind, gepunktet und Kanten, die von mehr als zwei Flächen verdeckt sind, werden nicht gezeichnet. Abbildung (c) stellt den Lichthofeffekt von ELBER dar.

Erst zum Ende der achtziger Jahre änderte sich das Interesse an der nicht-fotorealistischen Forschung. Auf dem Gebiet der Liniengrafik-Generierung wurde dies durch eine Reihe entscheidender Veröffentlichungen, u. a. von DOOLEY und COHEN (1990a,b), ELBER und COHEN (1990) sowie SAITO und TAKAHASHI (1990), ausgelöst. Von zentraler Bedeutung auch

für diese Diplomarbeit war die Entwicklung der (pixelorientierten) G-Buffer-Technik von SAITO und TAKAHASHI, mit denen auf einfache Art und Weise qualitativ hochwertige Liniengrafiken erzeugt werden können. Abschnitt 2.2.3 wird sich daher eingehend mit dieser Technik beschäftigen.

2.2.2 Erzeugung der Liniengrafiken: Von Kanten zu Strokes

In Abschnitt 2.1.1 wurde der Begriff der Liniengrafik definiert. Von grundlegender Bedeutung für die Darstellung eines Objekts sind dabei die Linien, die markante geometrische Merkmale beschreiben. Derartige geometrische Merkmale sind u. a. Kanten, Falten und Oberflächenkrümmungen, die zusammen mit *Kontur-* bzw. *Silhouettenlinien*, *inneren Linien* und *Schraffurlinien* gezeichnet werden.

Das Forschungsgebiet der computergenerierten Liniengrafiken unterteilt sich hier in die Entwicklung von Kantenstilisierungsverfahren und Schraffurvisualisierungen. Da sich die Verarbeitung von Kanten und Linienzügen exemplarisch für das G-Strokes-Konzept eignet, wird die Visualisierung von Schraffurlinien nicht weiter behandelt. Dennoch soll auch letzteres Verfahren durch die G-Strokes unterstützt werden können.

Allgemeine Vorgehensweise

Um Liniengrafiken am Computer erzeugen zu können, müssen zunächst die den Linien zugrunde liegenden Kanten vom gegebenen Datensatz extrahiert und bestenfalls mittels einer geeigneten Stilisierungs-Pipeline verändert werden (ISENBERG et al., 2002; NORTHRUP und MARKOSIAN, 2000). Hierzu gibt es verschiedene Techniken, die sich in pixelorientierte, hybride und objektraumorientierte bzw. analytische Verfahren unterteilen. Als Berechnungsgrundlage verwenden die Verfahren dabei im Allgemeinen ein in Form eines trianguliertes Polygonmodells vorliegendes 3D-Objekt.

Pixelorientierte Verfahren machen sich zur Erzeugung der Liniengrafiken die Bild-Puffer zu Nutze, um Informationen über die Geometrie bzw. die Lage der Kanten zu erhalten. Da die gefundenen Linien meistens in Form einer Pixelmatrix vorliegen, ist eine Stilisierung nicht möglich. Analytische Verfahren arbeiten im Gegensatz dazu direkt im Objektraum mit dem Polygonmodell. Die extrahierten Kanten liegen dabei in Koordinatenform vor und können von einer Stilisierungs-Pipeline weiterverarbeitet und stilisiert werden. Die hybriden Verfahren kombinieren beide Techniken. Hier werden nun zunächst wichtige Begriffe erläutert und definiert, die für das Verständnis der in den folgenden Abschnitten vorgestellten Verfahren zur Erzeugung von Liniengrafiken grundlegend sind.

Silhouettenkanten

Die *Silhouettenkanten* sind für die Erzeugung von Liniengrafiken von grundlegender Bedeutung, da sich mit ihnen bereits erste Konturzeichnungen generieren lassen (DURAND, 2002; FINKELSTEIN und MARKOSIAN, 2003). Da die Lage der Objektsilhouette immer von der jeweiligen Blickrichtung des Betrachters abhängt, ist ihre Berechnung jedoch insbesondere für animierte Bilder nicht trivial und hängt von der Art der Modelloberfläche ab (ISENBERG et al., 2003).

Definition 2.3 *Für stetig differenzierbare Oberflächen ist die Silhouette S als die Menge von Objektoberflächenpunkten x_i definiert, deren Oberflächennormalen n_i senkrecht zum Blickrichtungsvektor v stehen. Das Skalarprodukt zwischen n_i und v ist dabei gleich Null. Bei perspektivischer Projektion ergibt sich der Blickrichtungsvektor aus der Differenz zwischen x_i und dem Projektionszentrum c. Damit der Punkt auf der Silhouettenkante liegt gilt $\forall\, x \in S$:*

$$\frac{\mathbf{n_i}}{\|\mathbf{n_i}\|} * \frac{\mathbf{v}}{\|\mathbf{v}\|} = \mathbf{0} \qquad \frac{\mathbf{n_i}}{\|\mathbf{n_i}\|} * \frac{(\mathbf{x_i} - \mathbf{c})}{\|(\mathbf{x_i} - \mathbf{c})\|} = \mathbf{0}$$

Parallelprojektion *Perspektivische Projektion*

Definition 2.4 *Für nicht-stetig differenzierbare Teile von Oberflächen gilt, dass all jene Oberflächenpunkte x_i zur Menge S der Silhouette gehören, für die es ein $\varepsilon > 0$ gibt, sodass die ε-Umgebung von x_i außer S nur noch zwei jeweils zusammenhängende aber voneinander disjunkte Teiloberflächen A und B enthält. Für A und B gilt dabei, daß o.B.d.A. die Teilfläche A vollständig dem Betrachter zugewand und die Teilfläche B vollständig vom Betrachter abgewand ist. Somit bildet der Teil der Silhouette S innerhalb der ε-Umgebung von x_i eine Unstetigkeit der Oberfläche. Es gilt $\forall\ a \in A$ und $\forall\ b \in B$ mit entsprechender Oberflächennormale n_i:*

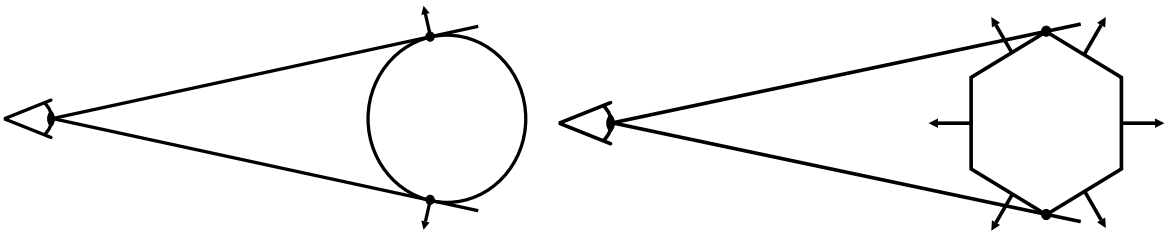
$$\frac{\mathbf{n_i}}{\|\mathbf{n_i}\|} * \frac{\mathbf{v}}{\|\mathbf{v}\|} < \mathbf{0} \qquad \frac{\mathbf{n_i}}{\|\mathbf{n_i}\|} * \frac{(\mathbf{a_i} - \mathbf{c})}{\|(\mathbf{a_i} - \mathbf{c})\|} < \mathbf{0} \qquad \forall\ a, n \in A$$

$$\frac{\mathbf{n_i}}{\|\mathbf{n_i}\|} * \frac{\mathbf{v}}{\|\mathbf{v}\|} > \mathbf{0} \qquad \frac{\mathbf{n_i}}{\|\mathbf{n_i}\|} * \frac{(\mathbf{b_i} - \mathbf{c})}{\|(\mathbf{b_i} - \mathbf{c})\|} > \mathbf{0} \qquad \forall\ b, n \in B$$

Parallelprojektion *Perspektivische Projektion*

mit $\mathbf{A} \cap \mathbf{B} = \emptyset$ *und* $\mathbf{A} \cup \mathbf{B} \subseteq \varepsilon$

In Abbildung 2.9(a) wird das Berechnungsverfahren für stetig differenzierbare, in Abbildung 2.9(b) das Verfahren für nicht-stetig differenzierbare Oberflächen illustriert. Die Definitionen für nicht-stetig differenzierbare Oberflächen können sehr einfach auf die Polygonmodelle angewendet werden, da dort nur die einzelnen Polygonflächen über Normalen

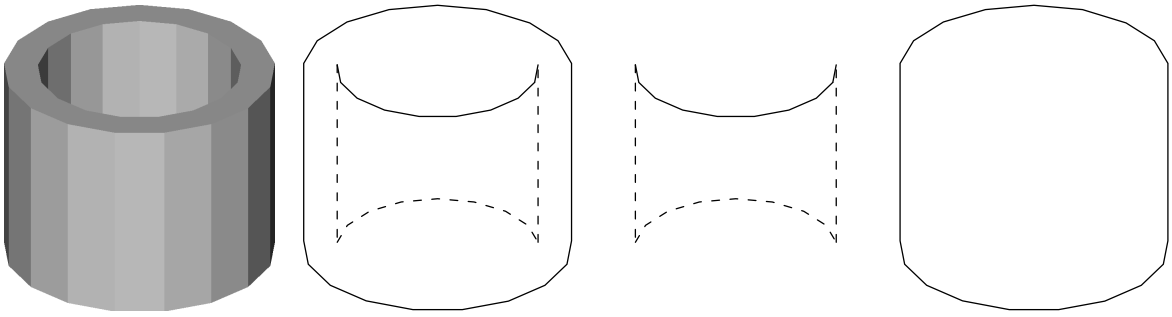


(a) Stetig differenzierbare Oberfläche. (b) Nicht-stetig differenzierbare Oberfläche.

Abbildung 2.9: Berechnungsverfahren von Silhouettenkanten. *Abbildungen mit freundlicher Genehmigung von Tobias Isenberg.*

verfügen. Zu den Silhouettenkanten gehören dann all jene Kanten des Modells, bei denen die Normale der einen anliegenden Polygonfläche entgegen der Blickrichtung des Betrachters und die Normale der anderen anliegenden Polygonfläche in Blickrichtung des Betrachters zeigt, der Winkel zwischen Normalen und Blickrichtung also einmal größer und einmal kleiner als 90° ist (APPEL, 1967).

Ein sich hieraus ergebender Nachteil bzgl. der Definition von Silhouettenkanten aus Abschnitt 2.1.1 ist, dass somit auch solche Kanten zur Silhouette gehören können, die im Inneren des Objekts liegen. ISENBERG et al. (2003) definieren daher neben der Silhouettenkante die beiden Untermengen *innere Silhouettenkante* und *Konturkante*. Zu den Konturkanten gehören nur Kanten, die das Objekt eindeutig vom Hintergrund trennen (vgl. Abbildung 2.10).



(a) Das Polygonmodell. (b) Die Silhouette. (c) Die innere Silhouette. (d) Die Kontur.

Abbildung 2.10: Silhouettenkanten im Überblick. *Abbildungen mit freundlicher Genehmigung von Tobias Isenberg.*

Bei pixelorientierten Verfahren wird die Silhouette wie auch die im kommenden Abschnitt beschriebenen Merkmalskanten des Objekts i. d. R. durch Unstetigkeiten in den Bild-Puffern extrahiert. Dies kann z. B. durch einen auf den z-Buffer angewendeten Kantenfilter erreicht werden (SAITO und TAKAHASHI, 1990).

Merkmalskanten

Neben den Silhouettenkanten werden in der Literatur weitere *Merkmalskanten* (*feature lines*) als Stilisierungsgrundlage genannt.[2] Grundlegend wird hier zwischen den folgenden Kantentypen unterschieden (DOOLEY und COHEN, 1990a; GOOCH et al., 1999; HERTZMANN, 1999; ISENBERG et al., 2003):

Falten (*creases* oder *folds*): Sie liegen im inneren des Objekts und bezeichnen Unstetigkeiten auf einer sonst glatten Oberfläche. Es wird dabei zwischen *Hügelfalten* (*ridge creases*) und *Talfalten* (*valley creases*) unterschieden. Erstere beschreiben stark nach außen gewölbte, zweitere stark nach innen gewölbte Oberflächenkrümmungen.

Begrenzungskanten (*boundary edges* oder einfach *borders*): Sie treten in ungeschlossenen Polygonmodellen auf und bezeichnen Kanten, die nur über eine anliegende Fläche verfügen.

Selbstüberschneidungen (*self-intersections*): Sie entstehen bei Überschneidungen der Polygonflächen selbst.

Die beschriebenen Merkmalskanten sind durch die Objektoberfläche eindeutig festgelegt. Daneben gibt es weitere Variationen. Hier sind u. a. die *suggestiven Konturen* (*suggestive contours*) von DECARLO et al. (2003, 2004) zu nennen. Suggestive Konturen „verlängern“ die Silhouettenkante in den sichtbaren Bereich des Objekts und unterstreichen somit weitere optische Merkmale des Modells. Dies ist im Bereich der Zeichnung eine sehr verbreitete Technik. Im Gegensatz zu den inneren Merkmalskanten sind die suggestiven Konturen blickrichtungsabhängig. Sie beschreiben solche Kanten, die, von einem leicht veränderten Blickpunkt aus betrachtet, zur Silhouettenkante gehören würden. Abbildung 2.11 veranschaulicht den Unterschied zwischen Silhouette und suggestiver Kontur.

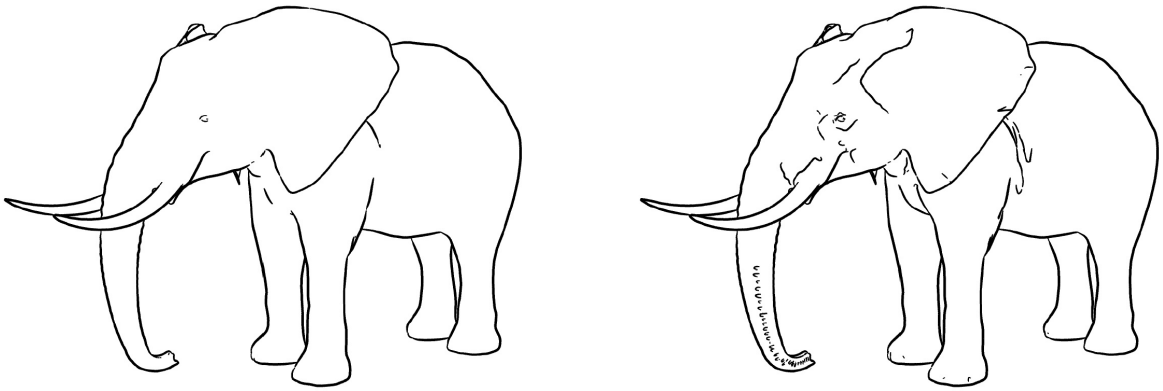


Abbildung 2.11: Ergebnisbilder von DECARLO et al. (2003). Links nur Silhouettenlinien, rechts Silhouette und suggestive Kontur.

2 Zwar gehören die Silhouettenkanten ebenfalls zu den Merkmalskanten, da sie jedoch eine objektbegrenzende und somit besondere Funktion haben, werden sie im weiteren Verlauf getrennt behandelt.

Strokes

In den letzten beiden Abschnitten wurden die für die Erzeugung von Liniengrafiken elementaren geometrischen Merkmale eines Modells, die Kanten, vorgestellt und definiert. Die dadurch gewonnenen Informationen reichen jedoch noch nicht aus, um eine künstlerisch zufriedenstellende Liniengrafik erzeugen zu können. Daher soll an dieser Stelle das Liniemodell von SCHUMANN (1997, Kap. 3) eingeführt werden.

Eine *Linie* wird demnach als eine Kombination aus *Pfad* und *Stil* definiert, wobei der Pfad die geometrische Lage angibt und somit den gefundenen Kanten entspricht. Pfad und Stil verfügen über bestimmte Attribute, z. B. Druck bzw. Linienbreite, Sättigung oder Farbe. Aufgabe des Stils ist es, den Pfad zu „stören“, also zu verändern. Dabei unterscheidet SCHUMANN zwischen der geometrischen Störung und der Störung der Attribute des Stils. Der Autor definiert an dieser Stelle die Störkurve und die Stillinie. Erstere bezeichnet den Verlauf und die Art des Stils, weshalb SCHUMANN den synonymen Gebrauch von Stil und Störkurve vorschlägt (SCHUMANN, 1997, Kap. 3, S. 33). Die Stillinie ist an den Pfad angepasst und dient als Referenz für die Störkurve. Sie verfügt über einen Anfangs- und einen Endpunkt, entlang derer sich der Stil anpasst (vgl. Abbildung 2.12).

Abbildung 2.12: Störkurve und Stillinie, aus SCHUMANN (1997).

Die geometrische Störung bedeutet eine geometrische Veränderung der Störkurve. Ein Beispiel hierfür ist die Kurvendarstellung aus Abbildung 2.12. Die Störung der Attribute bedeutet eine mögliche Veränderung in Abhängigkeit der Geometrie, also z. B. eine Variation der Linienbreite in Abhängigkeit von der Tiefeninformation.

Der *Stroke* soll nun an dieser Stelle als Kombination aus Störkurve und Stillinie definiert werden. Somit ist die Darstellung nicht an ein Kantensegment gebunden, sondern es ist eindeutig festgelegt, dass der Stil abhängig von der Stillinie über mehrere Kantensegmente verlaufen kann. Dies ermöglicht die Darstellung langer und geschlossener Linienzüge bzw. Strokes. Es ist dabei die Aufgabe der Stilisierungs-Pipeline, geschlossene Kantenzüge zu erzeugen und den jeweiligen Stil der Strokes festzulegen.

2.2.3 Pixelorientierte Verfahren: G-Buffer–Technik

Für die Kantenextraktion gibt es pixelorientierte, hybride und analytische Verfahren. Die im folgenden beschriebenen pixelorientierten Verfahren arbeiten ausschließlich im Bildbereich und dementsprechend mit zweidimensionalen Daten. Das Modell wird dabei in verschiedene Bild-Speicher gerendert. Die Pixeldaten werden dann weiterverarbeitet, z. B. mit be-

stimmten Filteralgorithmen. Zu den in diesem Zusammenhang am häufigsten verwendeten Bild-Speichern gehören u. a. die folgenden[3] (vgl. FOLEY et al. (1995) und Abbildung 2.13):

Der Color-Buffer speichert für jeden Pixel eine eindeutige Farbe und wird synonym mit dem Begriff *Frame-Buffer* benutzt. Im Color-Buffer steht nach jedem Rendering-Prozess das fertige Bild.

Der ID-Buffer speichert in Abhängigkeit von der Objekt-*ID* für jedes Szenenobjekt eine eindeutig identifizierende Farbe.

Der n-Buffer speichert für jeden Pixel die Normalenrichtung des entsprechenden Szenenpunktes. Die x-, y-, und z-Richtungen werden dabei durch RGB-Werte ersetzt.

Der z-Buffer speichert für jeden Pixel einen Graustufenwert, der proportional zu dem entsprechenden Tiefenwert (z-Wert) der Szene ist. Er kann auch genutzt werden, um eine korrekte Darstellung bzw. nicht-Darstellung der sichtbaren und verdeckten Kanten zu erreichen.

(a) Der Color-Buffer. (b) Der z-Buffer. (c) Der ID-Buffer.

Abbildung 2.13: Drei Bild-Buffer im Überblick (STROTHOTTE und SCHLECHTWEG, 2002).

SAITO und TAKAHASHI (1990) entwickelten unter Verwendung dieser Bildraumspeicher die G-Buffer-Technik. Ein G-Buffer ist dabei ein Bild-Buffer, der jeweils eine geometrische Information oder Eigenschaft des Modells enthält. Der am einfachsten zu erhaltende geometrische Speicher ist der z-Buffer, weil er meistens automatisch während des Rendering-Vorgangs berechnet wird. Da es sich bei den G-Buffern um zweidimensionale Pixelmatrizen handelt, die dreidimensionale Informationen enthalten, spricht man auch von $2\frac{1}{2}$D-Speichern (STROTHOTTE und SCHLECHTWEG, 2002).

Ziel der Arbeit von SAITO und TAKAHASHI war, eine fotorealistische Darstellung mit einer Liniengrafik zu überlagern. Eine Kernaussage der Autoren lautete dabei:

> „*Comprehensibility is mainly created through suitable enhancment rather than by accurate simulating optical phenomena*“ (SAITO und TAKAHASHI, 1990).

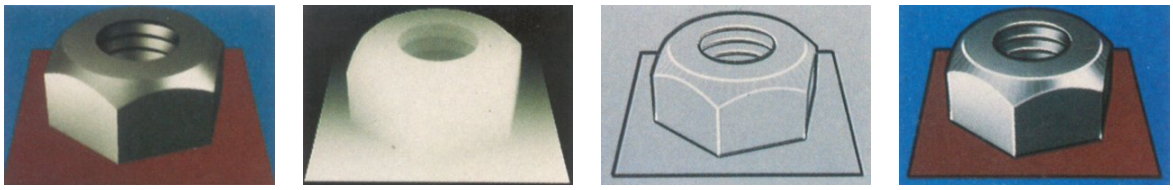
Die Verständlichkeit des dargestellten Objekts würde demzufolge nicht durch eine perfekte fotorealistische Darstellung gefördert werden können, sondern vielmehr durch eine kontext-

3 Der englische Begriff *Buffer* (*Speicher*) soll ebenfalls beibehalten werden, da er sich in der deutschsprachigen Literatur durchgesetzt hat.

abhängige Hervorhebung der Objektmerkmale. Dies entspricht den Feststellungen über die Wirkung von Liniengrafiken aus Abschnitt 2.1.

SAITO und TAKAHASHI gliedern den Rendering-Prozess in geometrische (Projektion, Entfernen verdeckter Kanten, etc.), physikalische (u. a. Beleuchtung und Texturierung) und künstlerische (z. B. Hinzufügen von Informationen zur Hervorhebung) Prozesse, die nacheinander abgearbeitet werden. Die G-Buffer werden dabei im ersten Schritt erzeugt und im letzten Schritt verwendet. Sobald die dreidimensionale geometrische Information in einer Pixelmatrix kodiert vorliegt, können beliebige Kombinationen von Hervorhebungen durchgeführt werden. Bei einer Veränderung der physikalischen oder künstlerischen Prozesse ist keine neue Berechnung der G-Buffer notwendig. Dies ist für die Verringerung des Rechenzeitaufwands von Vorteil.

Um die bessere Verständlichkeit der dargestellten Szene erreichen zu können, werden die geometrischen Eigenschaften in den G-Buffern von zweidimensionalen Bildbearbeitungsalgorithmen verändert. Am Beispiel der von SAITO und TAKAHASHI benutzten Gewindemutter aus Abbildung 2.14(a) soll ihre Technik nun verdeutlicht werden. Während des Rendering-Vorgangs wird der z-Buffer des Objekts erstellt (vgl. Abbildung 2.14(b)). Auf diesen G-Buffer werden im Anschluss Kantenfilter bzw. Ableitungen erster und zweiter Ordnung angewendet. Damit können dann die inneren und äußeren Silhouettenkanten gefunden und dargestellt werden (siehe Abbildung 2.14(c)). Die Überlagerung des Ausgangsbildes mit dem Kantenbild unterstützt den dreidimensionalen Eindruck der Gewindemutter deutlich, wie in Abbildung 2.14(d) zu sehen ist.



(a) Color-Buffer der Gewindemutter.

(b) Der z-Buffer.

(c) Innere und äußere Silhouette.

(d) Ergebnis der Überlagerung.

Abbildung 2.14: Die G-Buffer-Technik am Beispiel einer Gewindemutter (SAITO und TAKAHASHI, 1990).

Nachteilig an der G-Buffer-Technik ist jedoch, dass die Pixeldarstellung der Objektkanten einerseits auf Grund der begrenzten Bildauflösung zu Aliasing-Effekten führt und andererseits keine weitere Stilisierung zulässt (NORTHRUP und MARKOSIAN, 2000; ISENBERG et al., 2003). Desweiteren werden durch die Anwendung des Kantenfilters auf den z-Buffer nicht alle Kanten gefunden. Dies kann z. B. dann vorkommen, wenn Objekte oder Objektteile dieselben Tiefenwerte besitzen. DECAUDIN (1996) schlägt daher vor, zusätzlich zum z-Buffer den n-Buffer zu nutzen. Wird dieser Buffer abgeleitet, geben die gefundenen Kanten Auskunft über die Richtungsänderungen der Oberfläche. Eine Kombination der Kanten aus beiden Speichern liefert dann Merkmals- und Silhouettenkanten.

Trotz dieser Schwächen zählt die Veröffentlichung von SAITO und TAKAHASHI zu den elementarsten Verfahren bei der Erzeugung von Liniengrafiken. Dies liegt in der von den Au-

toren klar vollzogenen Trennung zwischen der Geometrie, der geometrischen Eigenschaften und der daraus resultierenden Stilisierung begründet und hat auch bei aktuellen Forschungsarbeiten nicht an seiner Wichtigkeit verloren.

Weitere Verfahren

Die pixelorientierten Verfahren verfügen im Gegensatz zu den analytischen Verfahren über den Vorteil, dass sie einen Großteil der Berechnung zur Erzeugung der Bild-Speicher an die ***G**raphics **P**rocessing **U**nit* (GPU) der Grafikkarte übergeben können. Das beschleunigt die Erzeugung von Liniengrafiken. Die Komplexität der Berechnung ist somit nicht mehr von der Anzahl der Polygonflächen des Modells, sondern von der Anzahl der Pixel des Bildes abhängig und demzufolge konstant (ISENBERG et al., 2003).

EISSELE et al. (2004) entwickelten in diesem Zusammenhang das sogenannte $\mathbf{G}^2$-Buffer Framework. Dabei können jegliche $\mathbf{G}$-Buffer-Operationen auf die GPU der Grafikkarte übertragen und dort ausgeführt werden. Dies ermöglicht eine einfache Umsetzung und Anwendung der $\mathbf{G}$-Buffer-Algorithmen und macht sie durch die Geschwindigkeit moderner Grafikkarten noch effektiver.

RÖSSL und KOBBELT (2000) nutzen zur Erzeugung der Silhouettenkanten und Schraffurlinien ebenfalls die $\mathbf{G}$-Buffer-Technik. Allerdings bevorzugen die Autoren den Begriff *enhanced edge buffer* im Gegensatz zu $\mathbf{G}$-Buffer, da sie die Diskretheit der Pixelsample unterstreichen wollen. Ziel ihrer Arbeit ist es, homogene Felder gleicher Orientierung zu erhalten und diese dann mit einer Schraffur belegen zu können. Hierfür interpretieren sie die Silhouettenkanten als Bereichsgrenzen und bieten zusätzlich Nutzerinteraktion an. Der Vorteil des Verfahrens liegt in der relativ variablen Stroke-Generierung der Schraffurlinien.

CURTIS (1998) entwickelte den *Loose and Sketchy* Filter, um den Ausdruck des Zeichners in der Linie darstellen und animieren zu können. Der Filter nutzt dabei Bildraumoperationen und ein stochastisch-physikalisches Partikelsystem zur Silhouettengenerierung. Als Eingabe wird lediglich der z-Buffer des 3D-Modells benötigt. Aus diesem Speicher wird dann der *Template-Buffer* erzeugt, der die Silhouettenkanten zur Bestimmung der jeweiligen Liniendichte hält. Desweiteren wird ein *Kraftfeld-Buffer* erzeugt, mit Hilfe dessen die Partikel entlang den Silhouettenkanten „geschoben“ werden sollen. In Abhängigkeit dieser beiden Buffer werden im Anschluss Partikel auf und um die Silhouettenkanten positioniert. Abbildung 2.15 zeigt Ergebnisbilder dieses Verfahrens.

2.2.4 Hybride Verfahren

Die hybriden Verfahren werden gekennzeichnet durch die Manipulation der 3D-Modelle im Objektraum, z. B. Translation und Skalierung der Polygone, und der sich anschließenden

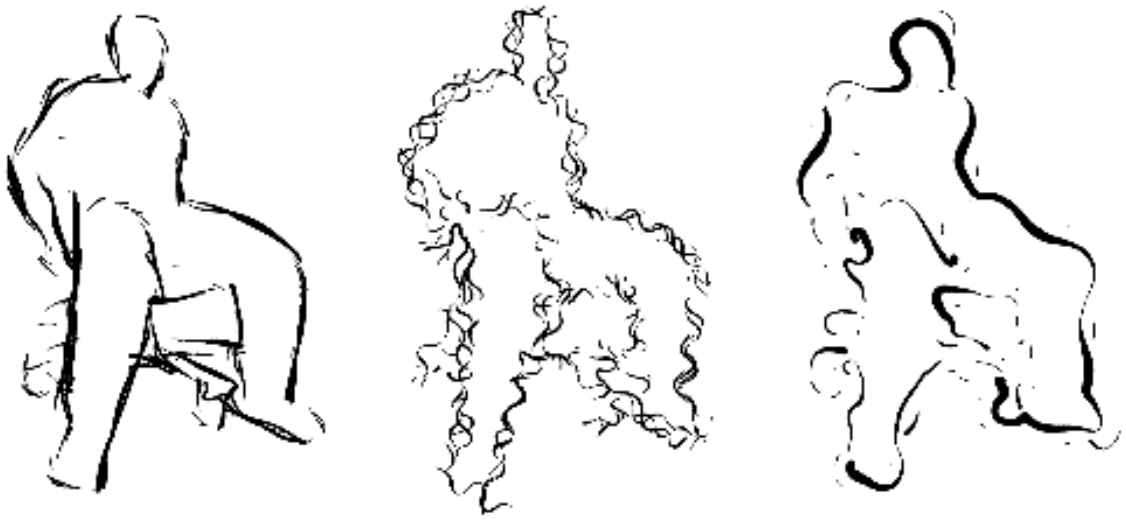


Abbildung 2.15: Bilder, die mit dem *Loose and Sketchy* Filter erzeugt wurden, aus (CURTIS, 1998).

Bearbeitung der Bild-Speicher, wie im letzten Abschnitt 2.2.3 beschrieben. Dies ermöglicht im Gegensatz zu den reinen pixelorientierten Verfahren in erster Linie eine variablere Darstellung der Silhouettenkanten. Jedoch liegen die gefundenen Kanten auch bei den hybriden Verfahren in Form einer Pixelmatrix vor (ISENBERG et al., 2003).

Ein frühes hybrides Verfahren zur Generierung von Liniengrafiken insbesondere für die technische Illustration wurde von ROSSIGNAC und VAN EMMERIK (1992) entwickelt. Die Autoren stellen dabei vier Algorithmen mit folgenden Aufgaben vor:

- Die sichtbaren Kanten (oder die sichtbaren Silhouettenkanten) werden immer gezeichnet.
- Die verdeckten Kanten (oder die verdeckten Silhouettenkanten) werden entweder nicht gezeichnet oder sie werden in einem anderen Stil als die sichtbaren Kanten dargestellt.

Die gerenderten Bilder ähneln denen von KAMADA und KAWAI (1987) aus Abbildung 2.8. Allerdings fehlt den Bildern von ROSSIGNAC und VAN EMMERIK der räumliche Eindruck, da sie bei der Darstellung der verdeckten Kanten nicht zwischen der Objektzugehörigkeit unterscheiden (vgl. hierzu Abbildung 2.16(a)).

Um die Silhouettenkanten extrahieren zu können wird zunächst der z-Buffer des 3D-Modells erstellt. Danach wird das Polygonmodell leicht in Richtung der negativen z-Achse verschoben und als Gittermodell mit großer Linienbreite erneut in den z-Buffer gerendert. Hieraus folgt, dass nur die Silhouettenkanten im z-Buffer vorliegen, da alle anderen Kanten hinter dem bereits gerenderten Modell liegen und somit nicht gezeichnet werden. Abschließend wird das Modell in den Vordergrund vor die ursprüngliche Position verschoben und mit normaler Linienbreite erneut gerendert. Somit liegen zusätzlich alle inneren Linien vor.

RASKAR und COHEN (1999) verallgemeinerten das Verfahren von ROSSIGNAC und VAN EMMERIK. Sie beziehen bei jeder neuen Erweiterung der z-Buffer-Daten *Front-* bzw. *Back-Face-Culling* und somit Hardwarebeschleunigung direkt in den Rendering-Vorgang ein. Zunächst lassen die Autoren dabei nur die sichtbaren und dem Betrachter zugewandten Flächen (durch *Back-Face-Culling*) in den z-Buffer zeichnen, um dann im Anschluss die verdeckten und dem Betrachter abgewandten Flächen in der gewünschten Silhouettenfarbe erneut zu rendern (durch *Front-Face-Culling*). Die Erstellung und Zeichnung des Modells in Gitterform kann ebenfalls zur Visualisierung der inneren Merkmalskanten genutzt werden (vgl. Abbildung 2.16(b)).

Ein weiteres hybrides Verfahren, das die Silhouettenkanten mit Hilfe von *Environment Mapping* und dem *Stencil Buffer* in Echtzeit stilisiert, stammt von GOOCH et al. (1999). Das Environment Mapping wird hier dazu genutzt, den Kantenpunkten, deren Normale senkrecht zur Blickrichtung steht, dunklere Farbwerte zuzuweisen. Der Stencil Buffer ist ein weiterer Grafikspeicher, dessen Werte zusammen mit den Tiefenwerten gespeichert werden. GOOCH et al. nutzen ihn u. a. dazu, den inneren Merkmalskanten andere Farben zuweisen zu können. Da das Modell zusätzlich mit klassischen Beleuchtungsverfahren gerendert wird, erinnert das Ergebnis an die Arbeit von SAITO und TAKAHASHI (1990). Abbildung 2.16(c) stellt ein derartig erzeugtes Bild dar.

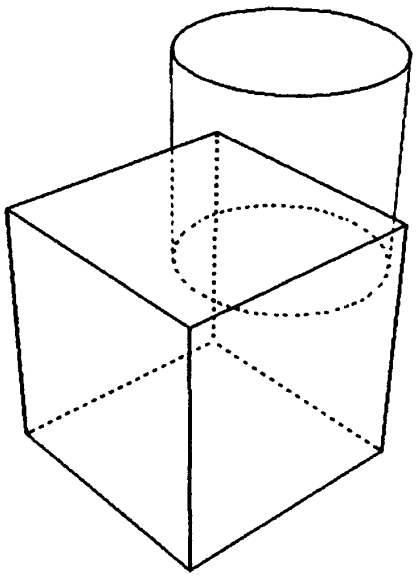
2.2.5 Analytische Verfahren

In den letzten beiden Abschnitten 2.2.3 und 2.2.4 wurden Verfahren vorgestellt, die die Silhouettenkanten unabhängig von einer vorherigen Polygonmodellmanipulation im Bildraum extrahieren. Vorteilhaft ist hierbei, dass die Bildraumverfahren viele Operationen an die *GPU* der Grafikkarte abgeben können. Demzufolge können die Liniengrafiken sehr schnell erzeugt werden. Da die Silhouettenkanten jedoch als Pixelmatrix und nicht in analytischer Form vorliegen, können sie nicht oder nur schwer weiterverarbeitet werden.

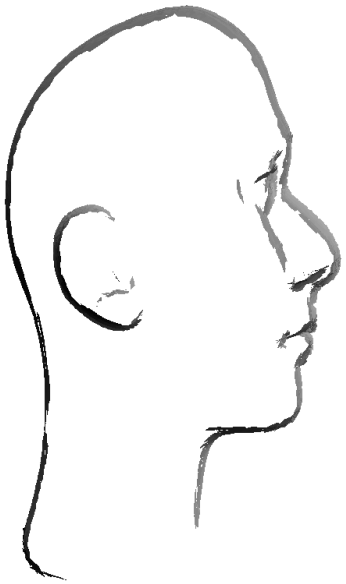
Hier setzen die analytischen Verfahren an. Sie extrahieren die Silhouetten- und Merkmalskanten im Objektraum, sodass die Kantenkoordinaten für eine weitere Stilisierung zugänglich sind. Dies ermöglicht dann z. B. die Anwendung verschiedener Stile auf ein und dasselbe Polygonmodell.

Kantenextraktion

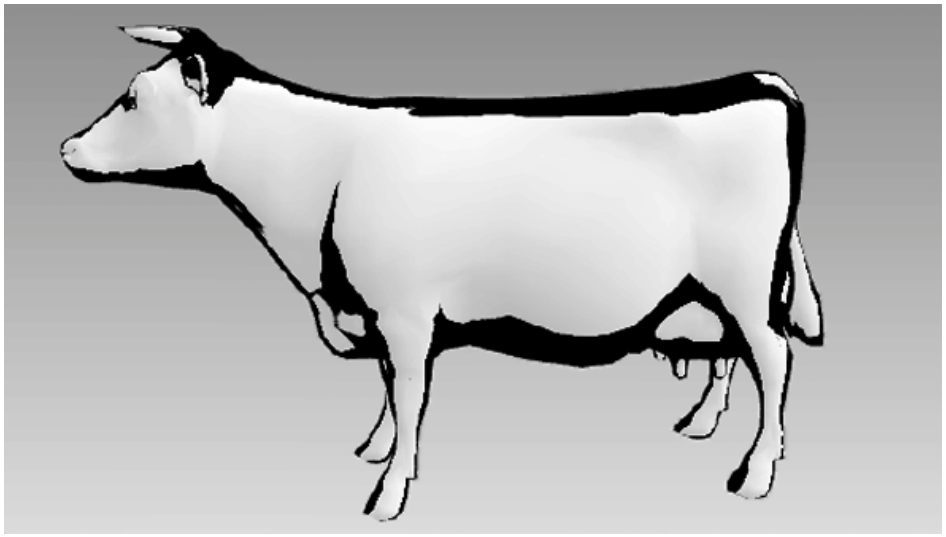
Die trivialste Methode zum Finden der Silhouettenkanten eines Polygonmodells im Objektraum wurde bereits in Abschnitt 2.2.2 für die nicht-stetig differenzierbaren Oberflächen beschrieben. Dabei werden alle Flächen bzgl. ihrer Oberflächennormale als *dem Betrachter zugewandt* (*front facing*) oder als *vom Betrachter abgewandt* (*back facing*), entsprechend



(a) (ROSSIGNAC und VAN EMMERIK, 1992)



(b) (RASKAR und COHEN, 1999)



(c) (GOOCH et al., 1999)

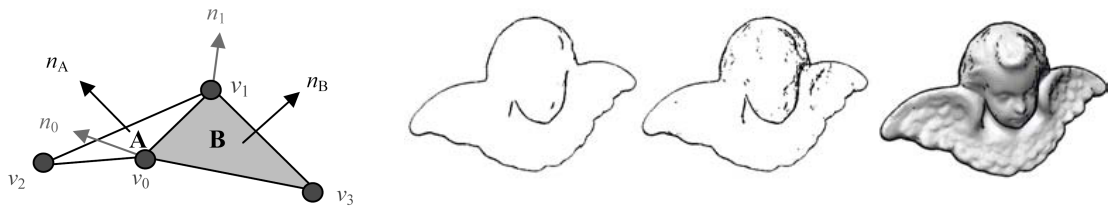
Abbildung 2.16: Drei Beispiele für die Silhouettengenerierung mit Hilfe von hybriden Verfahren.

Definition 2.4, deklariert und die Kanten als Silhouettenkanten gespeichert, deren anliegende Polygone beide Eigenschaften aufweisen (ISENBERG et al., 2003).

BUCHANAN und SOUSA (2000) verwenden als Prä-Prozess des eigentlichen Rendering-Vorgangs den sogenannten *Edge Buffer*. Hierbei handelt es sich nicht um einen Bild-Buffer, sondern um eine Datenstruktur. Diese kann bei der Extraktion von Silhouetten- und Merkmalskanten eines Polygonmodells genutzt werden und dient in erster Linie der Optimierung der Speicheroperationen. Im Edge Buffer werden dazu bzgl. der Normalenausrichtung der

Flächennormalen für jede Kante die Bitwerte *F* (*front facing*) und *B* (*back facing*) gespeichert. Durch eine XOR-Operation der Bitwerte können dann diejenigen Kanten herausgefiltert werden, deren benachbarte Polygone zum Betrachter hin bzw. vom Betrachter abgewandt sind.

MCGUIRE und HUGHES (2004) berechnen die Silhouetten- und Merkmalskanten mit Hilfe von Hardwarebeschleunigung durch die GPU. Problematisch bei diesem Ansatz ist die Tatsache, dass die GPU nur auf der Ebene von Eckpunkten und Pixeln arbeiten kann. Da für die Silhoutten- und Merkmalskantenextraktion jedoch Nachbarschaftsinformationen der Kanten und Flächen des Polygonnetzes notwendig sind, schlagen die Autoren die Nutzung einer neuen Datenstruktur vor: Das sogenannte Kantennetz (*Edge Mesh*) speichert die Kanteninformationen in einem *Edge-Vertex*, der einer Kante des Originalnetzes entspricht. Diese Datenstruktur enthält die Anfangs- und Endkoordinaten der zu speichernden Kante, die beiden Eckpunkte der angrenzenden Fläche sowie ein Skalar *r* für die Parameterisierung der Texturkoordinaten und einem Zähler *i* (vgl. Abbildung 2.17(a)). Da der Vertex-Prozessor



(a) Edge-Vertex-Daten. (b) Ergebnisbilder.

Abbildung 2.17: Die linke Abbildung zeigt die Daten eines Edge-Vertices. Die rechte Abbildung stellt drei Ergebnisse des Verfahrens von MCGUIRE und HUGHES (2004) dar.

der GPU nur einzelne Eckpunkte verarbeiten und keine Koordinaten löschen kann, werden Eckpunkte, die nicht zu einer Merkmalskante gehören, vor den sichtbaren Bildbereich (die *Near-Clipping-Plane*) geschoben. Desweiteren wird jeder Edge-Vertex viermal gespeichert, da nur durch das leicht verschobene Rendern der Edge-Vertices ein dicker rechteckiger Stroke erzeugt werden kann. Obwohl die Informationskodierung der Daten in einem Kantennetz wegweisend für die Verwendung von Hardwarebeschleunigung bei der Erzeugung nichtfotorealistischer Anwendungen sein kann, ist der Umfang des Kantennetzes gleichzeitig der Nachteil des Verfahrens. Dieses ist neunmal größer als das originale Polygonnetz und führt somit zu einer langsamen Datenverarbeitung.

Artefakte und Subpolygonsilhouetten

Polygonmodelle liegen i. A. in Form von triangulierten Gitternetzen vor und gehören somit zu den nicht-stetig differenzierbaren Oberflächen. Der Verlauf der Silhouettenkanten ist damit abhängig von den Dreieckskanten des Gitternetzes. Aus diesem Grund kommt es bei der Silhouettenextraktion i. d. R. zu Artefakten im Linienzug. Dies tritt insbesondere dann auf, wenn die Blickrichtung linear abhängig von einer Modellfläche ist, Blickrichtungsvek-

tor und Fläche also ungefähr parallel verlaufen. ISENBERG et al. (2002) und NORTHRUP und MARKOSIAN (2000) wenden daher Algorithmen zur Artefakt-Entfernung auf die extrahierten Silhouettenkanten an (vgl. Abbildung 2.18).

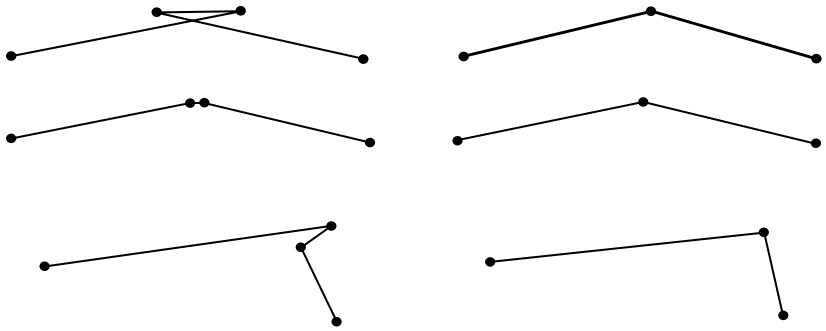


Abbildung 2.18: Diese Abbildung zeigt drei exemplarische Artefakt-Typen vor (jeweils links) und nach (jeweils rechts) ihrer Reduktion. Der obere Kantenzug enthält ein Zick-Zack-Artefakt, das auf Grund von numerischen Ungenauigkeiten entsteht. Der mittlere Kantenzug enthält eine zu kurze Kante, die mittels eines Schwellenwerts ermittelt und entfernt wird und der untere Kantenzug ist durch einen zu scharfen Winkel benachteiligt (ISENBERG et al., 2002).

HERTZMANN und ZORIN (2000) entwickelten eine andere Möglichkeit um diese Ungenauigkeiten zu vermeiden und mathematisch exakte Linienzüge zu erhalten. Sie berechnen die Subpolygonsilhouette des Modells. Dabei wird für jeden Punkt des triangulierten Polygonmodells das Skalarprodukt zwischen der Normale und dem Blickrichtungsvektor berechnet. Da die Polygonmodelle zu den nicht-stetig differenzierbaren Oberflächen gehören, sind die Ergebnisse der Skalarproduktberechnung an den Knoten i. d. R. größer oder kleiner Null (vgl. Abschnitt 2.2.2). Die Subpolygonsilhouette verläuft genau an der Grenze zwischen diesen beiden Extrema. Das heißt, dass all jene Kanten des Modells die Silhouette schneiden, bei denen das Ergebnis der Skalarproduktberechnung der Kantenendpunkte größer und kleiner Null ist. Dementsprechend muss zwischen diesen Endpunkten interpoliert werden, sodass der berechnete Nulldurchgang den Verlauf der Silhouette bzw. einen Silhouettenpunkt angibt. Abbildung 2.19 veranschaulicht diese Vorgehensweise bei der Berechnung von Subpolygonsilhouetten.

Vorverarbeitung

Die Berechnung der Silhouettenkanten ist blickrichtungsabhängig und muss demzufolge bei Animationen für jede Veränderung der Szene (jeden *Frame*) neu durchgeführt werden. Dies kann bei interaktiven Liniengrafiken zu langsamen Animationen führen, denn es müssen bei jedem Schritt alle Kanten und alle Flächen betrachtet werden. Die Komplexität beträgt folglich $O(N_{Kanten} + N_{Flaechen}) = O(N)$.

BENICHOU und ELBER (1999) beschleunigen diesen Prozess für orthografische Projektionen, indem sie vor der Kantenextraktion die Orientierungen der einzelnen Flächennormalen in einer Datenstruktur festhalten. Alle Flächennormalen werden dazu auf die Gaussku-

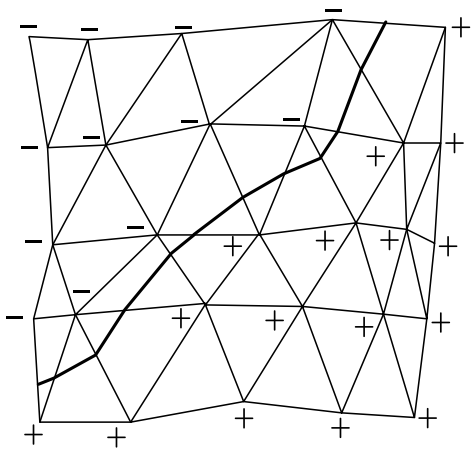


Abbildung 2.19: Darstellung der Subpolygonsilhouette, die über die Polygonflächen verläuft (HERTZMANN und ZORIN, 2000).

gel projiziert und die Normalen benachbarter Flächen zu Großkreissegmenten verbunden. Diese Segmente entsprechen damit den Orientierungen der Modellkanten. Danach wird die Gausskugel in ihrem Ursprung mit einer zur Blickrichtung orthogonal stehenden Ebene, der Bildebene, geschnitten. Eine der entstehenden Halbkugeln entspricht all den Flächen und Kanten, die dem Betrachter zugewandt sind, die andere Halbkugel enthält alle Flächen und Kanten, die vom Betrachter abgewandt sind. Um die Silhouettenkanten zu finden, muss nun lediglich überprüft werden, welche Großkreissegmente die Bildebene schneiden. Dies sind dann genau die Großkreissegmente, die vom sichtbaren in den nicht-sichtbaren Bereich verlaufen und dementsprechend mit den Silhouettenkanten korrespondieren. Die Laufzeitkomplexität der Berechnung wird somit auf die Schnittpunktberechnung der Kanten reduziert: $O(N_{Kanten})$.

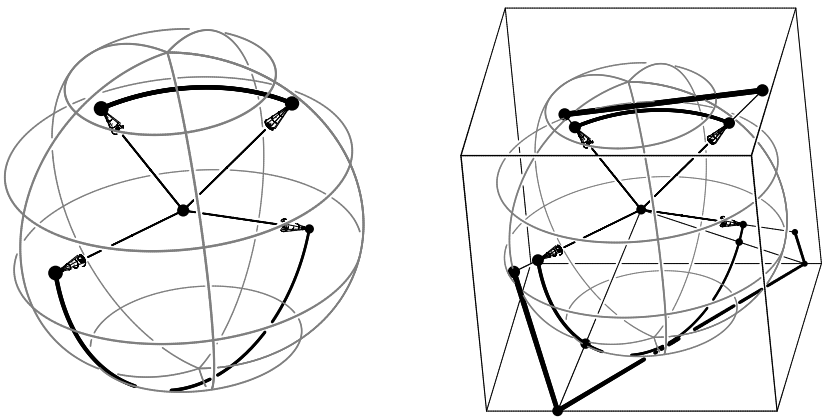


Abbildung 2.20: Links ist die Projektion der Flächennormalen auf die Gausskugel zu sehen. Die rechte Darstellung zeigt die Projektion der Großkreissegmente auf den umschließenden Würfel (BENICHOU und ELBER, 1999).

Um die Berechnungen weiterhin zu vereinfachen, projizieren BENICHOU und ELBER die Großkreissegmente auf einen die Gausskugel umschließenden Würfel. So wird die Schnittpunktberechnung einerseits vereinfacht und ermöglicht andererseits die Reduktion der er-

warteten Laufzeitkomplexität zu $O(N_{Silhouettenkanten})$, da die Autoren die Würfelflächen in ein Gitternetz aufteilen. In jeder Zelle dieses Gitternetzes speichern sie die darauf projizierten Grosskreissegmente. Wird der Würfel nun mit der aktuellen Bildebene geschnitten, müssen nur noch diejenigen Kanten überprüft werden, die in den entsprechenden Zellen, die die Bildebene geschnitten hat, liegen. BENICHOU und ELBER reduzieren das Problem der Silhouettenkantenextraktion durch die Bestimmung der Orientierungen somit auf das Problem der Schnittpunktberechnung. Abbildung 2.20 veranschaulicht die Projektionen auf Kugel und Würfel.

Sichtbarkeitsbestimmung

Nach der Kantenextraktion liegen die Silhouettenkanten in analytischer Koordinatenform vor. Bevor sie stilisiert werden können, muss zunächst ihre Sichtbarkeit bestimmt werden. Dies gehört in der Computergrafik zu den klassischen Problemen des Entfernens verdeckter Kanten (FOLEY et al., 1995).

Die von APPEL entwickelte Methode der quantitativen Unsichtbarkeit wird in leicht angepasster Form von MARKOSIAN et al. (1997) und HERTZMANN und ZORIN (2000) verwendet (vgl. hierzu Abschnitt 2.2.1). Da dieses Verfahren analytisch arbeitet, zeichnen sich die Ergebnisse durch eine hohe Präzision aus. Von Nachteil ist allerdings der durch diese Berechnung verursachte hohe Zeitaufwand (ISENBERG et al., 2003). Um diesen zu verringern nutzen NORTHRUP und MARKOSIAN (2000) den **ID**-Buffer zur eindeutigen Bestimmung der einzelnen Silhouettenkanten. ISENBERG et al. (2002) schlagen unter Verwendung des z-Buffers ebenfalls einen bildbasierten Ansatz vor. Die gefundenen Silhouettenkanten werden dabei zunächst in den z-Buffer gerendert. Die Bestimmung der Sichtbarkeit der 3D-Kante erfolgt mit Hilfe der korrespondierenden Pixelwerte und deren Achter-Nachbarschaft im z-Buffer. Die Verwendung der Achter-Nachbarschaft reduziert Unstetigkeiten in den Tiefenwerten, die insbesondere dann auftreten, wenn es mehrere ähnliche Tiefenwerte gibt.

Stilisierung

Nachdem die sichtbaren Silhouettenkanten bestimmt sind, können sie weiterverarbeitet bzw. stilisiert werden. Hierzu werden die Kanten zunächst zu Strokes verbunden und dann durch eine Stilisierungs-Pipeline geschickt, in welcher sie entsprechend der dort implementierten Stile verändert werden.

Eine erste Möglichkeit für die Erzeugung stilisierter Liniengrafiken besteht darin, einen einzigen Stil zu implementieren und auf beliebige Objekte anzuwenden. In Abschnitt 2.2.3 wurde der *Loose and Sketchy* Filter von CURTIS (1998) vorgestellt. Einen ähnlichen Stil entwickelten SOUSA und PRUSINKIEWICZ (2003), allerdings mit dem Ziel, die 3D-Form des Modell mit wenigen Strokes darzustellen. Der von ihnen implementierte *Loose*-Stil soll in erster Linie eine skizzenhafte Wirkung der Liniengrafik erzeugen. Dabei untersuchen die

Autoren insbesondere die *Shape Feature Selection* zur Auswahl der objektdefinierenden Linien, sowie die *Line Economy Control*, um die Anzahl dieser Linien bestimmen zu können. Die gefundenen Merkmalskanten werden dabei abhängig von der Oberflächenkrümmung gefunden und zu Strokes verbunden. Die Ergebnisse ihres Verfahrens sind in Abbildung 2.21 zu sehen. Es wird deutlich, dass nur einzelne Linien ausreichend sind, um ein Objekt zu definieren und visuellen Freiraum zu schaffen.

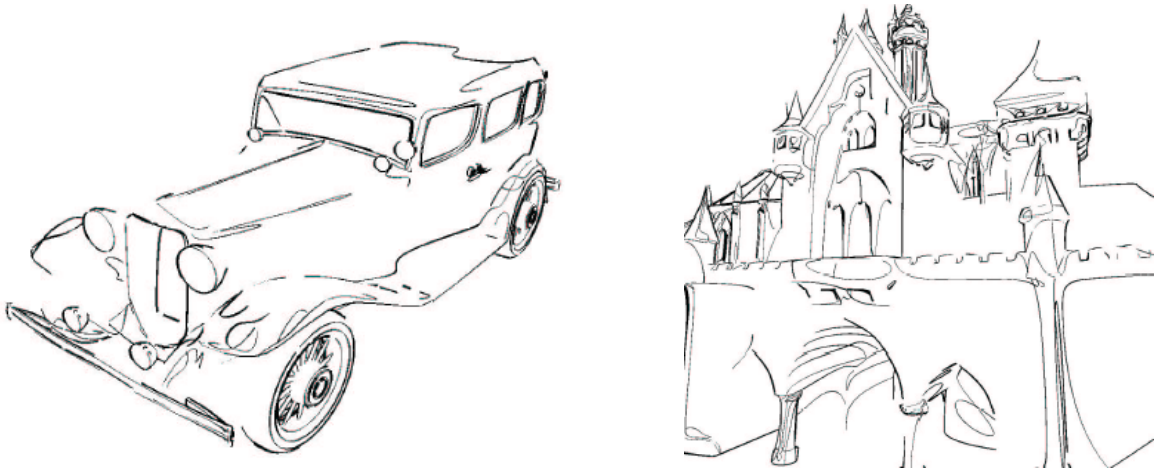


Abbildung 2.21: Ergebnisbilder des skizzenhaften *Loose*-Stils von SOUSA und PRUSINKIEWICZ (2003).

Obwohl die Ergebnisse von SOUSA und PRUSINKIEWICZ (2003) beeindruckend sind, handelt es sich um nur einen einzigen Stil zur Liniendarstellung. KALNINS et al. (2002) entwickelten ein interaktives Echtzeit-Rendering-System, WYSIWYG NPR, bei dem der Nutzer Stile, Hintergründe und Texturen auswählen und auf das 3D-Modell „zeichnen“ kann. Das System bietet eine gewaltige Vielzahl an möglichen Stilen und Stil-Kombinationen. Da WYSIWYG NPR in erster Linie jedoch als interaktiv nutzbares Programm für Designer entwickelt wurde, gehört es nicht direkt in den Bereich der automatischen Liniengrafik-Generierung.

KALNINS et al. (2003) bauten auf dem WYSIWYG NPR-System auf und entwickelten ein Echtzeit-Rendering-System, das die zeitkohärente Stilisierung der Silhouettenkanten bei animierten 3D-Modellen ermöglicht. Desweiteren ist die Darstellung unterschiedlicher Stile z. B. für sichtbare und verdeckte Kanten möglich. Das Polygonnetz wird für die unterschiedliche Stilisierung in Segmente aufgeteilt, denen jeweils ein Stil zugewiesen wird. Allerdings sind hierfür mehrere Rendering-Vorgänge nötig. Als zukünftige Arbeit nennen die Autoren den Wunsch, verschiedene Stile bei animierten Szenen ineinander übergehen zu lassen und somit die Trennung der Segmente auflösen zu können (vgl. Abbildung 2.22).

DURAND (2002) empfiehlt in diesem Zusammenhang eine Darstellungssoftware zu entwickeln, die aus möglichst kleinen Modulen bestehen und die Silhouette als Hauptprimitiv nutzen soll. Dies ermögliche laut Autor eine einfache Kombination verschiedenster Stile (vgl. (DURAND, 2002, Abschnitt 5.3)). Er hält desweiteren fest, dass die Projektion der 3D-

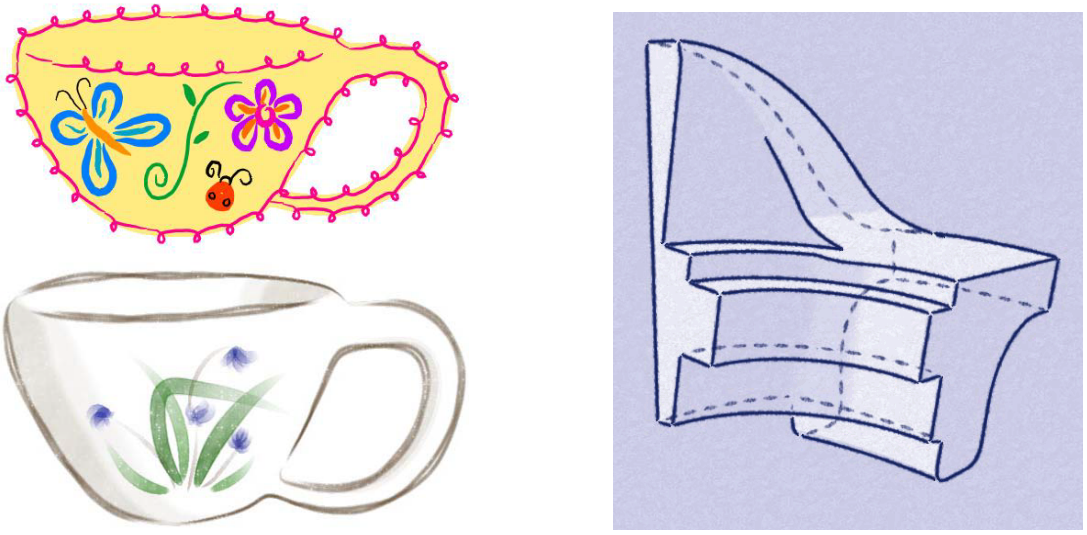


Abbildung 2.22: Die beiden linken Bilder sind Ergebnisse des Verfahrens von KALNINS et al. (2002), die rechte Abbildung gehört zu den Ergebnissen von KALNINS et al. (2003).

Szene auf ein 2D-Bild eine Abbildung der dreidimensionalen Eigenschaften in den zweidimensionalen Raum ist. Diese Feststellung scheint insbesondere im Zusammenhang mit der Entwicklung der G-Strokes von Interesse zu sein, da dort ebenfalls die dreidimensionalen Eigenschaften der Strokes für die nicht-fotorealistische Darstellung des Bildes genutzt werden sollen.

HALPER et al. (2003a) griffen DURANDs Empfehlung nach einem modularen Kombinationssystem als erste auf und entwickelten zur Erzeugung und Stilisierung von Liniengrafiken das modulare Echtzeit Rendering-System OPENNPAR.[4] Dieses verarbeitet die geometrischen Informationen des Polygonmodells und verfügt über eine Reihe von Stil-Elementen. So können z. B. Kanten in Wellenform gebracht, mit Texturen belegt oder in Abhängigkeit ihrer Tiefenposition mit unterschiedlicher Linienbreite dargestellt werden (*Depth Cueing*) (vgl. Abbildung 2.23). Die grundlegende Philosophie der Autoren ist dabei, alle Module frei miteinander kombinieren zu können. Die Stilisierungs-Pipeline hat hierbei die Aufgabe, entweder die Geometrie der Kanten zu verändern oder der Kante Eigenschaften hinzuzufügen, u. a. Parameterwerte für die Texturbelegung. Die Pipeline-Elemente müssen dementsprechend angepasst werden. Der Aufwand der Pipeline-Anpassung und der Modul-Kombination ist jedoch nicht unerheblich und beschränkt dementsprechend die Nutzung der implementierten Stile (vgl. TIETJEN, 2004).

GRABLI et al. (2004; 2003) entwickelten in Anlehnung an fotorealistische Rendering-Software ebenfalls ein modulares NPR-Rendering-System für nicht-animierte Liniengrafiken. Die Autoren legen dabei vor allem auf die freie und variable Kombination mehrerer Stile in einem Bild wert. Um dies zu erreichen, erzeugen sie eine Stroke-Datenstruktur, die den

4 Weitere Informationen unter http://www.opennpar.org/.

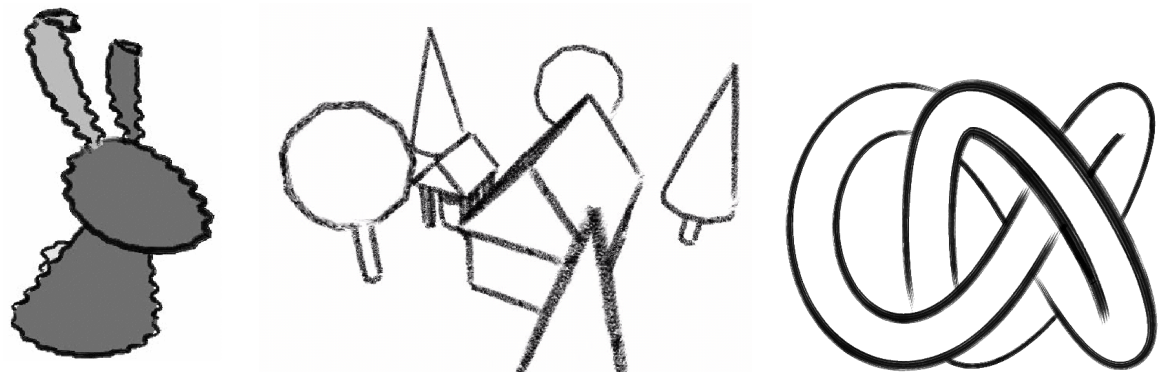


Abbildung 2.23: Diese Abbildungen wurden mit dem modularen NPR-System OPENNPAR erzeugt (ISENBERG et al., 2002).

Kantenverlauf des Strokes und die prozedural entwickelbaren Attribute (Linienbreite, etc.) enthält. Das System arbeitet dazu mit einzelnen Stil-Modulen, die jeweils aus den Schritten *Kantenextraktion*, *Kantenverknüpfung*, *Stroke-Generierung* und *Stilisierung* bestehen. Die Verkettung der einzelnen Stil-Module führt dann zu einem Bild, dass über mehrere Stile verfügt. Abbildung 2.24 zeigt drei Ergebnisse des GRABLI-Systems.

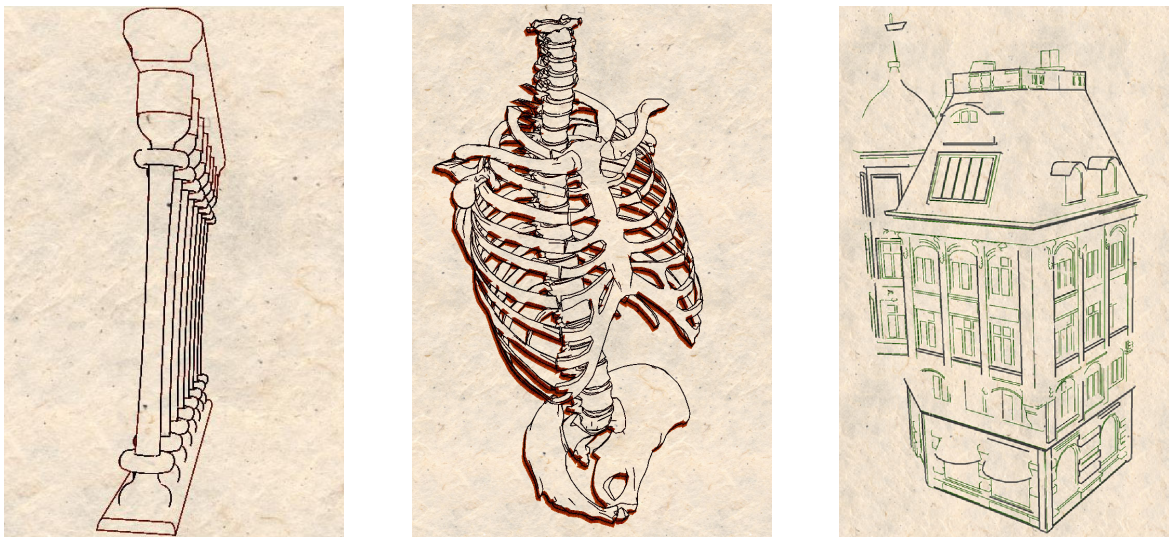


Abbildung 2.24: Diese drei Abbildungen wurden mit dem GRABLI-System erzeugt (GRABLI et al., 2003).

2.2.6 Zusammenfassung

Die hier vorgestellten Verfahren beschäftigten sich alle mit der Erzeugung von Liniengrafiken am Computer. Dazu verwenden sie triangulierte Polygonmodelle als Rendering-Grund-

lage. Die reinen pixelorientierten sowie die hybriden Verfahren können in den meisten Fällen die Hardwarebeschleunigung der Grafikkarte nutzen. Allerdings sind die erzeugten Liniengrafiken auf Grund der Bildraumberechnungen nur pixelgenau und neigen in der Regel zu Aliasing-Effekten. Hinzu kommt, dass durch die reine bildraumorientierte Silhouettenkantenextraktion nur ein geringfügiger Teil der geometrischen Modellinformationen in die Liniengrafik-Generierung eingeht. Der entscheidende Vorteil der G-Buffer-basierten Verfahren ist jedoch die klare Trennung zwischen Geometrie, geometrischen Eigenschaften (den G-Buffern) und der davon abhängigen Stilisierung. Für variable Stilisierungsverfahren und Stilkombinationen sind die pixelorientierten und hybriden Verfahren jedoch nicht geeignet.

Die analytische Kantenextraktion der analytischen Verfahren ist im Gegensatz dazu zeitaufwändiger, allerdings können die gefundenen Kanten stilisiert werden. Die Stile können zusätzlich durch die geometrischen Objekteigenschaften oder Nutzerinteraktion entweder einzigartig oder kombiniert auf ein Objekt angewendet werden. Analytische Verfahren entsprechen daher vielmehr dem zeichnerischen Ansatz aus Abschnitt 2.1.1, da sie die Geometrie als Grundlage für die Stilisierung nutzen können.

Die beschriebenen Verfahren sind i. d. R. auf die Entwicklung und Umsetzung eines einzigen Stils beschränkt. Die beiden einzigen modularen Systeme zur automatischen Liniengrafik-Generierung, OPENNPAR und das System von GRABLI et al., bieten zwar eine Fülle von Stilkombinationen, jedoch ist keine klare Trennung im Sinne der G-Buffer-Technik vorhanden. Da alle Pipeline-Elemente an jede Veränderung angepasst werden müssen, wird die Entwicklung und Kombination neuer Eigenschaften und dementsprechend neuer Stile behindert. Obwohl das GRABLI-System zur besseren Einteilung eine Stroke-Datenstruktur einführt, vereint diese Kantenzug und Attribute, wodurch beide Primitive wiederum voneinander abhängig sind. Die hierfür notwendige Implementierung verhindert desweiteren die Erzeugung animierter Liniengrafiken.

Die Entwicklung eines unabhängigen Konzepts, das Geometrie, geometrische Eigenschaften und Stilisierungen trennen kann, würde die Entwicklung neuer Stile demgegenüber fördern. Dies hätte neben einer freien und variablen Kombination von Modell-Eigenschaften zur Erzeugung bestimmter Stile auch den Vorteil, dass der Berechnungsaufwand durch einmaliges Festlegen der Eigenschaften verringert werden würde. Desweiteren könnten die Aufgaben der Stilisierungs-Pipeline klar in die Filterung von Eigenschaften und die Kombination zu Stilen getrennt werden, was zu einer erheblich intuitiveren Entwicklung führen würde. Hierzu soll im folgenden Abschnitt das Konzept der G-Strokes vorgestellt werden.

Kapitel 3

G-Strokes – Ein Stilkonzept

In Kapitel 2 wurden die grundlegenden Begriffe, Definitionen und Techniken auf dem Gebiet der zeichnerischen und algorithmischen Generierung von Liniengrafiken beschrieben und untersucht. Im Bereich der computergenerierten Liniengrafiken bieten dabei nur die analytischen Verfahren die Möglichkeit, die Kantenzüge für animierte oder nicht-animierte Grafiken zu stilisieren. Der hohe Berechnungs- und Kombinationsaufwand dieser Verfahren behindert jedoch die schnelle Entwicklung neuer bzw. eine variable Nutzung mehrerer Stile. Das vorliegende Kapitel gliedert sich zunächst in einen ersten Abschnitt über die Analyse des Stilisierungsproblems. Im Anschluss daran wird das G-Strokes-Konzept vorgestellt, das die Entwicklung neuer Stile fördert und eine einfache Kombination dieser Stile ermöglicht.

3.1 Problemanalyse

Die Objekte einer dreidimensionalen Modellszene am Computer liegen i. A. in Form von triangulierten Polygonnetzen vor, da sich glatte Objektoberflächen mit ihnen sehr einfach und flexibel annähern lassen (vgl. HERTZMANN und ZORIN, 2000). Die Eckpunkte des Polygonnetzes, die Vertizes, werden dabei einmalig als 3D-Koordinaten in einer *Vertex-Liste* gespeichert und über ihre Indizes angesprochen. Die Sequenz der Indizierung liegt entsprechend in einer *Index-Liste* vor und baut das Polygonnetz auf. So können dieselben Vertizes mehr als einmal referenziert werden, ohne dass ihre Koordinaten mehrfach gespeichert werden müssen. Abbildung 3.1 illustriert dieses Zusammenspiel von Vertex- und Index-Liste.

3.1.1 Stilisierungs-Pipeline

Bevor eine dreidimensionale Modellszene als zweidimensionales Bild auf dem Monitor dargestellt werden kann, muss sie eine sogenannte *Rendering-Pipeline* durchlaufen (vgl. z. B. FOLEY et al., 1995, Kap. 16). Eine solche Pipeline besteht aus mehreren Elementen, die in Form eines Graphen nacheinander angeordnet sind und in einer vorgegebenen Reihenfolge, i. d. R. in *preorder*, also bei der Wurzel beginnend von oben nach unten und von links nach rechts, traversiert werden. Jedes Element arbeitet dabei mit den angepassten Daten seines

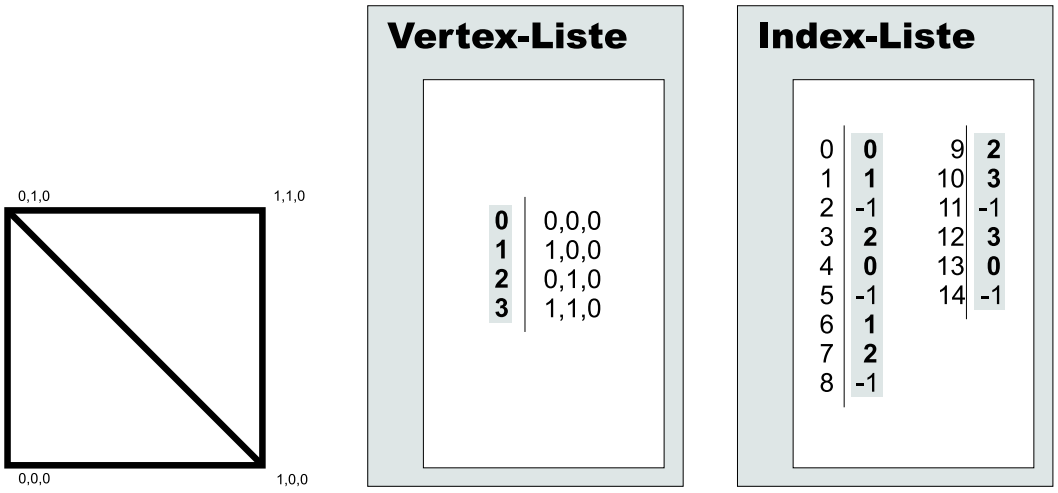


Abbildung 3.1: Funktionalität der Vertex- und der Index-Liste am Beispiel eines triangulierten Quadrats. Die Positionen der Koordinaten in der Vertex-Liste (0,1,2,3) werden in der Index-Liste als Sequenz gespeichert. Die Zahl -1 dient als Trennung zwischen den einzelnen Kanten und soll an dieser Stelle als Konvention für die Kanten- und Stroke-Trennung eingeführt werden.

Vorgängers. Um eine solche Pipeline zu implementieren, kann das Konzept des Szenengraphs eingesetzt werden. Diese Konzept dient i. A. der Beschreibung eines Geometriemodells, wobei einzelne Modell- und Szeneneigenschaften in einem Graph angeordnet werden und durch schrittweise Traversierung zum Modellbild führen. An Stelle der Modelleigenschaften können ebenso Pipeline-Elemente verarbeitet werden. Diese werden dabei als Graph-Knoten bezeichnet und sind über die Graph-Kanten miteinander verbunden. Diese Implementierung hat u. a. den Vorteil, dass die einzelnen Einstellungen der Knoten interaktiv verändert werden können.

Die Pipeline-Elemente (oder Szenengraph-Knoten) führen eine Reihe von Transformationen und Veränderungen der Szene durch, die schließlich zum gewünschten Bild führen. Für diese Diplomarbeit ist die Erzeugung nicht-fotorealistischer Liniengrafiken und dementsprechend eine dafür notwendige Rendering-Pipeline von besonderem Interesse. Neben dem Ein- und Ausgabe-Vorgang besteht sie hauptsächlich aus zwei Abschnitten: Pipeline-Elementen, die für die Kantenextraktion zuständig sind, sowie Pipeline-Elementen, die diese Kanten stilisieren (vgl. Abbildung 3.2). Die extrahierten Kanten werden in Vertex- und Index-Listen gespeichert und durch den Stilisierungsabschnitt, die *Stilisierungs-Pipeline*, geschickt. Die Elemente des Stilisierungsabschnittes können die Kanten dann entsprechend ihrer Funktionionalität verändern, also *stilisieren*. Hierzu gehört z. B. die Verbindung der einzelnen Kanten zu zusammenhängenden Kantenzügen, den Strokes, sowie das Entfernen verdeckter Stroke-Segmente. Ebenso könnten aus den geraden Segmenten der Strokes Kurven angenähert und eine Parameterisierung festgelegt werden. Anschließend werden die restlichen Strokes an das Ausgabe-Element übergeben. Um den Eindruck einer gezeichneten Liniengrafik zu unterstützen, repräsentieren verschiedene Texturen dabei bestimmte Strich-Typen

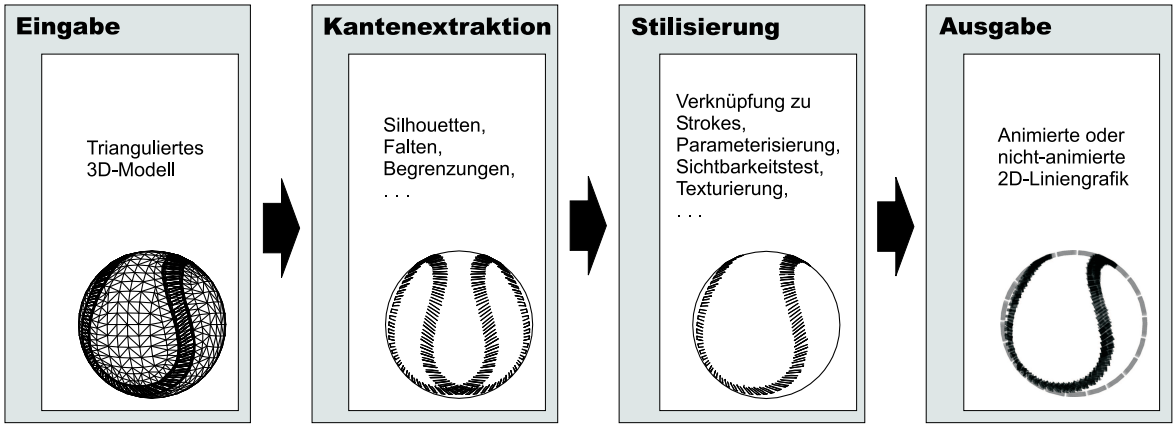


Abbildung 3.2: Allgemeiner Aufbau der Rendering-Pipeline bei nicht-fotorealistischen Grafiken.

bzw. Zeichenwerkzeuge. Die in Abbildung 3.2 verwendete Textur stellt z. B. den Strich eines mit Wasserfarbe getränkten Pinsels dar.

3.1.2 Problematik der Stilisierungs-Pipeline

Die beschriebene sequenzielle Abarbeitung der Pipeline-Elemente einschließlich der Kantenextraktion ist notwendig und von Vorteil für die Verarbeitung von 3D-Modellen. Für einfache Liniengrafiken oder die Verwendung eines einzigen Stils ist auch die Struktur der Stilisierungs-Pipeline gut geeignet, die hierzu einmal traversiert werden muss. Sollen jedoch anspruchsvollere Bilder erzeugt werden, z. B. im Zusammenhang mit der Entwicklung modularer Rendering-Systeme aus Abschnitt 2.2.5, birgt dieser Pipeline-Abschnitt erhebliche Probleme. Zur Veranschaulichung soll daher das Beispiel aus Abschnitt 1.1 erneut aufgegriffen werden.

Für den Fall, dass die gefundenen Strokes gleichmäßig texturiert werden sollen, muss die Stilisierungs-Pipeline über ein Element verfügen, welches die Strokes parameterisiert. Möglicherweise müssen dazu neue Koordinaten berechnet werden, sodass Vertex- und Index-Liste verändert werden. Das Ergebnis dieser Parameterisierung muss dem Ausgabe-Element der Stilisierungs-Pipeline zur Verfügung stehen. Folglich muss eine Parameter-Liste erzeugt werden, die gemeinsam mit Vertex- und Index-Liste an alle folgenden Elemente der Pipeline übergeben wird. Bereits an dieser Stelle treten drei Probleme auf:

1. Die folgenden Pipeline-Elemente, die den Stroke verändern, müssen die Anpassung der Parameter-Liste in ihre Methoden einbeziehen.
2. Die Reihenfolge der Pipeline-Elemente wird entweder unflexibel, da alle Elemente, die nicht über die Verarbeitung der Parameter-Liste verfügen, vorher in der Pipeline platziert werden müssen, oder alle Pipeline-Elemente müssen soweit angepasst wer-

den, dass sie eine Verarbeitung der Parameter-Liste vorsehen, sobald diese existiert.

3. Die Parameterisierung kann erst am Ende der Pipeline durchgeführt werden. Die Parameter würden sich dann jedoch nicht mehr auf die ursprünglich gefundenen Strokes, sondern nur noch auf die von der Stilisierung übrig gebliebenen Stroke-Teile beziehen.

Der Aufwand wächst dabei proportional mit jeder neuen Datenstruktur und ist unabhängig von der Art der Datentypen. So können zwar dieselben Datentypen vorliegen, dennoch müssen sie ggf. unterschiedlich behandelt werden.

Weitere Probleme treten auf, wenn mehrere Stile in einem Bild vereint werden sollen, z. B. bei der unterschiedlichen Darstellung sichtbarer und verdeckter Stroke-Segmente. Hierbei erhöht sich die Komplexität der Szenengraphen-Struktur (*Problem 1*), die ein Anwachsen des Berechnungsaufwands zur Folge hat (*Problem 2*). Bedingt sind diese Schwierigkeiten durch die Tatsache, dass vor dem Entfernen der verdeckten Segmente keine Informationen über ihre Sichtbar- bzw. Nicht-Sichtbarkeit vorliegen. Erst das für das Entfernen zuständige Pipeline-Element[1] führt eine derartige Einteilung durch, indem es die verdeckten Segmente entfernt und die übrigen für eine weitere Stilisierung bestehen lässt. Um beide Eigenschaften, Sichtbarkeit und Nicht-Sichtbarkeit, visualisieren zu können, müsste das HLR-Element dahingehend angepasst werden, dass es entweder die sichtbaren oder die verdeckten Segmente entfernt. Die übrigen Segmente können dann nacheinander stilisiert und gerendert werden. An dieser Stelle tritt Problem 1 auf: Da es nur unter großem Aufwand möglich wäre, beide Kantentypen getrennt zu verwalten und zu stilisieren, wird der Aufbau zweier Subgraphen notwendig (vgl. Abbildung 3.3). Diese Struktur bedingt wiederum Problem 2: Das HLR-Element braucht in beiden Fällen den gesamten Satz an Kanten, für die es jeweils die Sichtbarkeitsberechnungen durchführen muss.

Aufbau und Abarbeitung der Stilisierungs-Pipeline sind bereits bei diesem einfachen Beispiel umständlich und erfordern bei komplexeren Anforderungen an die Grafik noch größeren Aufwand. Zusätzlich müssen die für den jeweiligen Stil erforderlichen Berechnungen in allen Subgraphen wiederholt durchgeführt werden. Bei animierten Liniengrafiken kommt erschwerend hinzu, dass die Silhouettenkanten bei jedem neuen Frame berechnet werden müssen, die Pipeline somit wiederholt traversiert werden muss. Bei großen und verzweigten Pipelines wird die Animation dabei zu langsam. Ein Beispiel hierfür ist das System von GRABLI et al. (2004) (vgl. Abschnitt 2.2.2).

Dennoch gibt es neben dem Pipeline-System keine adequate Lösung zur Verarbeitung der dreidimensionalen Daten, insbesondere durch den Vorteil der interaktiv möglichen Veränderung der einzelnen Elemente. Das Pipeline-System ist für die Stilisierung der Strokes des 3D-Modells notwendig. Vertex- und Index-Daten der Stroke-Segmente müssen dabei jedem Pipeline-Element zur Verfügung stehen können, da sonst keine Veränderung der Strokes im Sinne einer Stilisierung ermöglicht werden kann. Außerdem müssen die Pipeline-Elemente

1 Dieses Element wird i. d. R. als *Hidden-Line-Remover* (*HLR*) bezeichnet und soll dementsprechend an dieser Stelle als *HLR-Element* eingeführt werden.

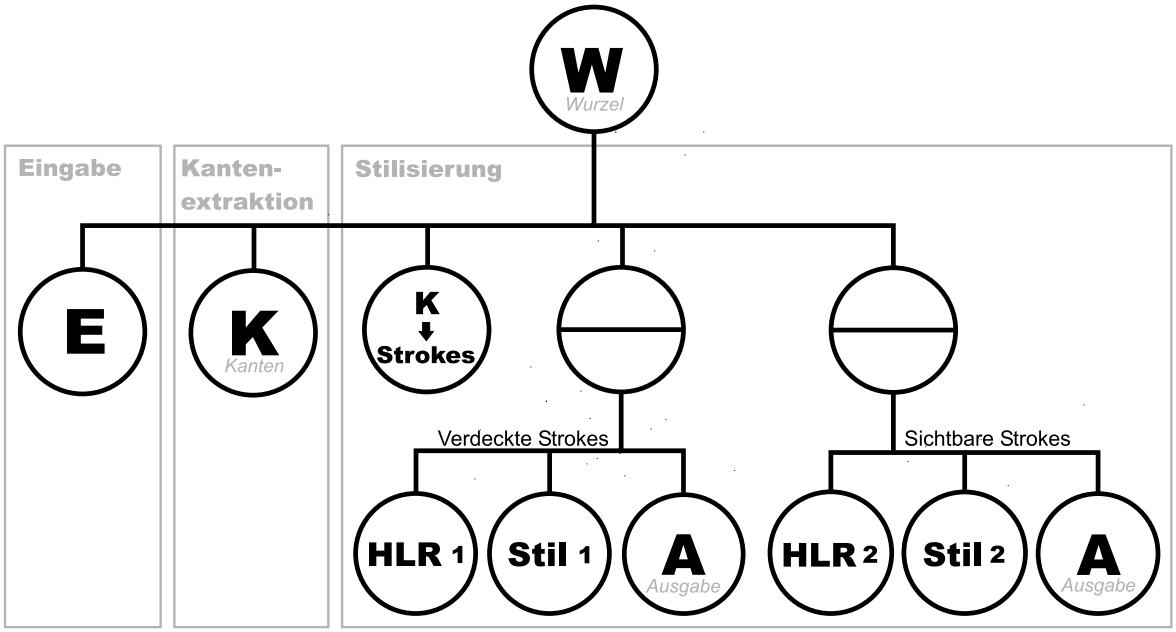


Abbildung 3.3: Möglicher Aufbau der Rendering-Pipeline bei der unterschiedlichen Darstellung sichtbarer und verdeckter Strokes.

nach wie vor neue Eigenschaften erzeugen und die Strokes entsprechend angleichen können. Um zu verhindern, dass alle Pipeline-Elemente bei jeder neuen Eigenschaft angepasst werden müssen oder große und verzweigte Szenengraphen entstehen, muss die Verarbeitung der Stroke-Daten neu konzipiert werden. Dies schließt vor allem auch die Aufgaben der einzelnen Pipeline-Elemente ein.

3.2 Stroke und G-Stroke

Im letzten Abschnitt wurde detailliert auf die Problematik bei der Stilisierung der dreidimensionalen Modelle eingegangen. Um die genannten Probleme zu beheben und eine flexible Stilisierungs-Pipeline zu ermöglichen, soll nun an dieser Stelle das G-Strokes-Konzept vorgestellt werden. Dieses Konzept ermöglicht eine einfache Stilisierung sowie die unterschiedliche Darstellung verschiedener Linienzüge in einem Bild. Grundlegend basiert es dabei auf der Trennung von *Stroke-Geometrie* und *Stroke-Eigenschaften*, woraus sich u. a. die in der Motivation aus Abschnitt 1.1 geforderte getrennte Verwaltung beider Primitive ergibt.

Der Stroke eines Polygonmodells ist ein geschlossener Kantenzug, der stilisiert werden kann und den Strichen einer Liniengrafik entspricht. Einführend soll hier an die Anforderungen an den Zeichner einer Liniengrafik im klassischen Sinne erinnert werden (vgl. SCHUMANNs Definition aus Abschnitt 2.1.1). Dieser muss die wichtigsten geometrischen Merkmale des Objekts erkennen und durch die Platzierung der Linien die Form des Objekts wiedergeben

können. Die Art bzw. der Stil der Linie sollte desweiteren dazu genutzt werden, der Zeichnung Ausdruck zu verleihen und bestimmte Wirkungen (z. B. Lichteinfall oder Tiefe) zu erzielen (vgl. Abschnitt 2.1.2).

In Anlehnung an diesen Prozess extrahieren die in Abschnitt 2.2.5 vorgestellten aktuellen analytischen Verfahren zur Generierung von Liniengrafiken die Kanten des 3D-Modells und stilisieren sie. Dieser Vorgang wird für die Visualisierung eines einzigen Stils sowie für die Darstellung mehrerer Stile durchgeführt. Soll nur ein Stil dargestellt werden, müssen die Elemente der Stilisierungs-Pipeline einmal traversiert werden. Bei mehreren Stilen wird der Aufbau mehrerer Subgraphen erforderlich, damit verschiedene Kanten extrahiert und nacheinander unterschiedlich stilisiert werden können.[2] Die analytischen Verfahren untermauern somit die Problemanalyse aus Abschnitt 3.1.

Unabhängig von diesen Schwierigkeiten scheint insbesondere die *wiederholte* Vorgehensweise der Pipeline-Traversierung (*Kanten extrahieren, Kanten stilisieren, Kanten rendern*) bei der automatischen Erzeugung von Liniengrafiken im Vergleich zur zeichnerischen Herangehensweise an die Darstellung von Objekten sehr unintuitiv zu sein. Es stellt sich daher die Frage, warum die automatische Generierung von Liniengrafiken nicht in die Schritte

a) Extrahieren *aller* Kanten und Verbinden zu Strokes (*Erkennen der wichtigen Objektmerkmale sowie Platzieren der Linie*) und

b) Stilisieren der Strokes bzgl. ihrer geometrischen Eigenschaften (*Erzielen einer Wirkung durch die Linienart*)

eingeteilt wird.

Lägen die Koordinaten der Strokes in einer Liste und die Eigenschaft der Sichtbarkeit in einer parallel indizierten Liste vor, so könnte die Stilisierung einfach und intuitiv anhand dieser Eigenschaft durchgeführt werden (siehe Abbildung 3.4).

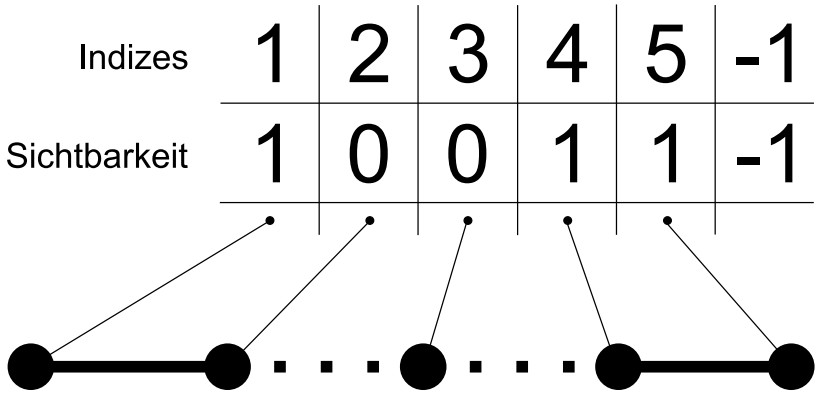


Abbildung 3.4: Ein erstes Beispiel für die Trennung von Stroke-Geometrie und Stroke-Eigenschaft.

2 Die Stil-Module des Systems von GRABLI et al. (2004) sind in diesem Zusammenhang nichts anderes als solche Subgraphen.

Der dargestellte Stroke ist ein Kantenzug bestehend aus fünf Koordinaten, die über die Indizes 1, 2, 3, 4, 5 angesprochen werden. Zur Information über die geometrische Position des Strokes wird die davon abhängige bzw. daraus folgende geometrische Eigenschaft seiner Sichtbarkeit hinzugefügt. Diese bezieht sich auf die Stroke-*Segmente*, weshalb die Zahlen immer für das folgende Segment zu verstehen sind. Die letzte 1 bezieht sich auf kein Segment mehr, ist daher ohne Bedeutung und erhält einfachheitshalber den zuletzt vergebenen Sichtbarkeitswert. Da die Sichtbarkeit eine *logische* Eigenschaft ist, die entweder erfüllt (*wahr* bzw. *1*) oder nicht erfüllt (*falsch* bzw. *0*) werden kann, muss sie durch den Datentyp *Boolean*[3] repräsentiert werden. Das Ausgabe-Pipeline-Element kann sich nun einfach an der Sichtbarkeits-Eigenschaft orientieren und für die sichtbaren und verdeckten Kanten unterschiedliche Stile anwenden.

Diese Vorgehensweise beinhaltet zusätzlich eine Veränderung des Stilisierungsbegriffs. Dieser basierte bisher nur auf der Stroke-Veränderung durch die Stilisierungs-Pipeline. Durch das G-Stroke-Konzept werden dem (durch die Stilisierungs-Pipeline stilisierten) Stroke jedoch weitere Stilisierungsmöglichkeiten angeboten, z. B. unterschiedliche Stile für sichtbare und verdeckte Segmente auf Grund der Stroke-Eigenschaften. Um Mißverständnissen vorzubeugen soll daher an dieser Stelle klar zwischen

- *Stilisierung* der Geometrie mittels Stilisierungs-Pipeline und
- *Stilgebung* durch die G-Strokes

unterschieden werden.

3.2.1 Begriff und Definition: Stroke

Bisher beinhaltete der Begriff *Stroke* immer die Kombination aus Stillinie und Störkurve, also dem geschlossenen Kantenzug und dem erzeugten Stil. Im Zusammenhang mit dem G-Stroke, der im folgenden begrifflich genau definiert werden wird, bietet sich jedoch eine Variation der Definition des Begriffes *Stroke* an.

Der *Stroke* wird an dieser Stelle als das geometrische Primitiv der Liniengrafik eingeführt. Da der Stil des Strokes erst durch die Stilisierungs-Pipeline erzeugt wird, soll bereits der geschlossene Kantenzug als Stroke gelten. Dieser definiert sich dann über die korrekte Adressierung der vorliegenden Koordinaten. Das heißt, dass der Stroke grundsätzlich aus einer Vertex- und einer Index-Liste besteht. In diesen Listen werden die Koordinaten und Indizes aller extrahierten und zu Strokes verknüpften Kanten nacheinander gespeichert. Die Koordinaten werden dabei durch den Datentyp *3D-Vector*,[4] die Indizes durch den Datentyp *Integer*

3 Für die verschiedenen Datentypen sollen hier die englischen Bezeichnungen verwendet werden, da sie auch in der deutschen Literatur als Fachbegriffe verwendet werden.

4 Hierbei handelt es sich um ein Feld, das z. B. drei Integer- oder drei Float-Werte speichern kann. Diese entsprechen dann den Positionen x, y und z. Gleiches gilt für den *2D-Vector*, der allerdings nur die Positionen x und y speichert.

repräsentiert. Jeder Vertex kommt somit einmal vor und kann mehrfach adressiert werden. Diese sequenzielle Art der Datenspeicherung beinhaltet, dass die gesamte Anzahl an Strokes verallgemeinert als *der Stroke* der Liniengrafik bezeichnet werden darf. Wenn im folgenden Verlauf der Arbeit von dem *Stroke* gesprochen wird, ist i. d. R. das Datenobjekt gemeint, das die einzelnen durch -1 getrennten Kantenzüge (die Strokes) in Vertex- und Index-Liste speichert. Es gilt außerdem, dass der Stroke einzigartig ist. Ein 3D-Modell kann nur über *eine* Stroke-Geometrie verfügen, eine Pipeline darf deshalb nur mit einem Stroke arbeiten. Desweiteren gilt, dass der Stroke über beliebig viele Eigenschaften, den G-Strokes, verfügen kann. Um die enge Relation zwischen Stroke und G-Stroke zu verdeutlichen, „sammelt“ der Stroke seine Eigenschaften in einer G-Stroke-Liste.

3.2.2 Begriff und Definition: G-Stroke

Der G-*Stroke* wird als eine eindeutige, in erster Linie geometrische Eigenschaft des Strokes definiert. Die Namensgebung wurde in Anlehnung an die G-Buffer von SAITO und TAKAHASHI (1990) gewählt. Im Unterschied zu den G-Buffern müssen sich die G-Strokes jedoch immer der Geometrie des Strokes anpassen, die durch die Stilisierungs-Pipeline geändert werden kann. Ein G-Buffer ist als ein bildabhängiger Pixelspeicher zu betrachten. Dieser wird einmal gesetzt und dann ggf. für Hervorhebungen im Bild genutzt. Seine Daten sind statisch. Sie werden in erster Linie als Informationsquelle für die Veränderung der zweidimensionalen Bilddaten genutzt, ändern sich selbst oder die 2D-Daten jedoch nicht. Im Gegensatz dazu stehen die G-Strokes. Sie beschreiben Eigenschaften dreidimensionaler Daten (hier Kantenzüge) und können neben den geometrischen auch andere Informationen tragen. Auf Grund der analytischen Berechnung der dreidimensionalen Daten (der Kantenzüge) ist eine Veränderung der Geometrie dieser Daten durch die Stilisierungs-Pipeline möglich. Die G-Strokes müssen sich dementsprechend an die neue Geometrie anpassen. Zusätzlich können sie auch selbst für eine Veränderung der 3D-Daten sorgen, z. B. im Zusammenhang mit der Stilgebung. Diese Tatsache wird als *doppelte Abhängigkeit* zwischen Stroke und G-Stroke bezeichnet und in Abschnitt 3.4 näher erläutert. Die G-Stroke-Daten sind dementsprechend dynamisch und ihre Anwendungsmöglichkeiten somit komplexer als die der G-Buffer.

Jeder G-Stroke besteht dabei aus einer Datenliste, die parallel zur Index-Liste des Strokes indiziert ist. Dies bedeutet, dass jeder Kantenzug des *gesamten* Stroke eine oder mehrere parallele Eigenschaften hat. Innerhalb der einzelnen Eigenschaften wird ebenfalls jeder einzelne G-Stroke durch eine -1 getrennt. Die parallele Kantenspeicherung ermöglicht für die jeweilige Position in der Index-Liste den direkten Zugriff auf den Wert der Eigenschaft. So gilt z. B. für das Segment, welches dem Vertex an der Position 3 aus Abbildung 3.4 folgt, dass es nicht sichtbar ist. Neben der Eigenschaft der Sichtbarkeit kann jede beliebige weitere Eigenschaft angelegt werden. Die erzeugten Eigenschaften können dann u. a. für die Generierung eines Stroke-Stils verwendet werden. Genausogut sind allerdings auch beliebige andere Anwendungen möglich. Es gilt desweiteren, dass eine Eigenschaft nur einmal vorkommen darf und

somit *einzigartig* ist. Es darf z. B. nicht mehrere Sichtbarkeits-Eigenschaften eines Strokes geben.

3.2.3 G-Strokes im Überblick

Die Art der Eigenschaft bzw. des G-Strokes entscheidet in diesem Zusammenhang über den Datentyp der Datenliste. Die Sichtbarkeit wird z. B. durch den Datentyp Boolean[5] repräsentiert und bezieht sich auf die Kantensegmente. Die Datenliste darf dementsprechend nur aus den Werten 0 und 1 bestehen, sowie aus -1 zur Trennung der einzelnen G-Strokes. Eine andere Eigenschaft könnte z. B. für die Festlegung der Linienbreite des Strokes genutzt werden. Sie muss für jeden Vertex einen Dicke-Parameter berechnen. Daher bietet sich die Verwendung von Gleitkommazahlen, also der Einsatz des Datentyps *Floating Point* oder kurz *Float*, an.

Hieraus ergeben sich zwei wesentliche Punkte, die den Einsatz und Gebrauch der G-Strokes festlegen:

1. Die G-Strokes werden durch verschiedene Datentypen repräsentiert. Die Anzahl der zur Verfügung stehenden Datentypen ist dabei begrenzt. Folglich bietet sich der Entwurf einer G-Strokes-Hierarchie an, bei dem die einzelnen Eigenschaften durch die Datentypen kategorisiert werden können. Dies beinhaltet z. B. auch die gemeinsame Verwendung von Operationen zur Anpassung des G-Strokes an eine mögliche Veränderung der Stroke-Geometrie.

2. Die Werte der Datenliste des G-Strokes beziehen sich entweder auf das (folgende) Stroke-Segment, wobei der zuletzt gespeicherte Wert nicht näher betrachtet werden muss, oder sie beziehen sich auf den Vertex.

Neben den Datentypen Boolean und Float können desweiteren Integer sowie 2D- und 3D-Vector genutzt werden. Tabelle 3.1 gibt einen Überblick über grundlegende G-Strokes.

Die Hierarchie dieser Eigenschaften ist in Abbildung 3.5 zu sehen. Hierbei handelt es sich um drei Ebenen. Der G-Stroke entspricht der obersten Ebene und stellt das (abstrakte) Primitiv für jegliche Stroke-Eigenschaften dar. Die zweite Hierarchie-Ebene wird von den Datentypen dieser Eigenschaften gebildet. Sie dienen der Kategorisierung der dritten Ebene, auf der sich die eigentlichen Eigenschaften befinden, die als G-Strokes verwendet werden können.

5 Da die -1 als Stroke/G-Stroke-Trenner zum Einsatz kommen muss, wird anstelle des Boolean-Datentyps der Datentyp Integer benutzt. Dieser lässt jedoch nur die genannten Werte 0, 1 und -1 zu und wird im weiteren Verlauf auf Grund seiner Funktionalität weiterhin als Boolean-Datentyp bezeichnet werden.

G-Stroke	Bezogen auf	Datentyp	Beschreibung
Visibility (Sichtbarkeit)	Segment	Boolean	Zur Beschreibung sichtbarer und verdeckter Segmente.
Edgetype (Kantentyp)	Segment	Integer	Zur Kategorisierung des Kantentyps.
ObjectID (ObjektID)	Segment	Integer	Zur Kategorisierung des Objekts.
Parameter (Parameter)	Vertex	Float	Zur Festlegung der Textur-Parameter.
Thickness (Breite)	Vertex	Float	Zur Festlegung der Linienbreite.
Orientation (Orientie-rung)	Vertex	2D-Vector	Zur Festlegung der Textur-Orientierung.
Normal (Nor-male)	Vertex	3D-Vector	Zur Festlegung der Normale jedes Vertizes.
Color (Farbe)	Segment	3D-Vector	Zur Festlegung der Farbe pro Kante.

Tabelle 3.1: Mögliche G-Strokes im Überblick. Ohne Details des folgenden Kapitels 4 vorwegnehmen zu wollen, werden hier bereits die bei der Implementierung verwendeten englischen Namen der einzelnen G-Strokes verwendet.

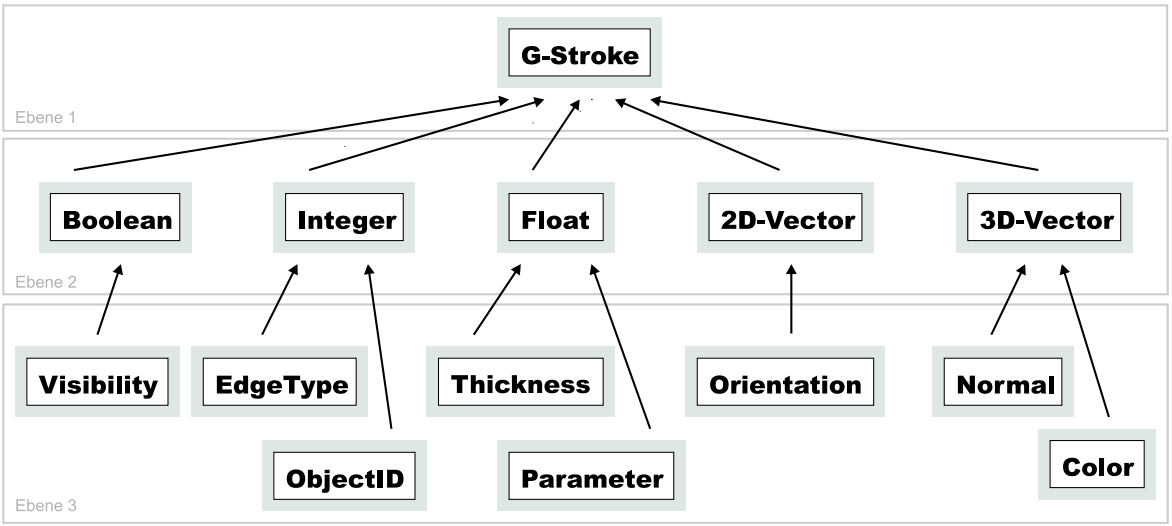


Abbildung 3.5: Die Hierarchie der G-Strokes.

3.3 Getrennte Datenverwaltung

Die logische Trennung von Geometrie und geometrischer Eigenschaft ermöglicht eine neue Verarbeitung der Stroke-Daten. In diesem Zusammenhang sind die folgenden Feststellungen grundlegend:

1. Die Geometrie besitzt bestimmte Eigenschaften. Das beinhaltet, dass die Eigenschaften von der Geometrie abhängig sind. Sobald sich die Geometrie verändert, müssen sich die Eigenschaften anpassen.

2. Der Stroke ist als geometrisches Primitiv die Grundlage für die G-Strokes. Sobald sich der Aufbau des Strokes ändert, müssen sich die G-Strokes der neuen Struktur anpassen.

3. Die Koordinaten des Strokes werden über eine Index-Liste adressiert und der Verlauf des Strokes somit eindeutig festgelegt. Die G-Strokes werden parallel zur Index-Liste angelegt, sodass ein direkter Zusammenhang zwischen Stroke und G-Stroke entsteht (vgl. Abbildung 3.6). Sobald sich diese Adressierung ändert, müssen sich die G-Strokes entsprechend ändern.

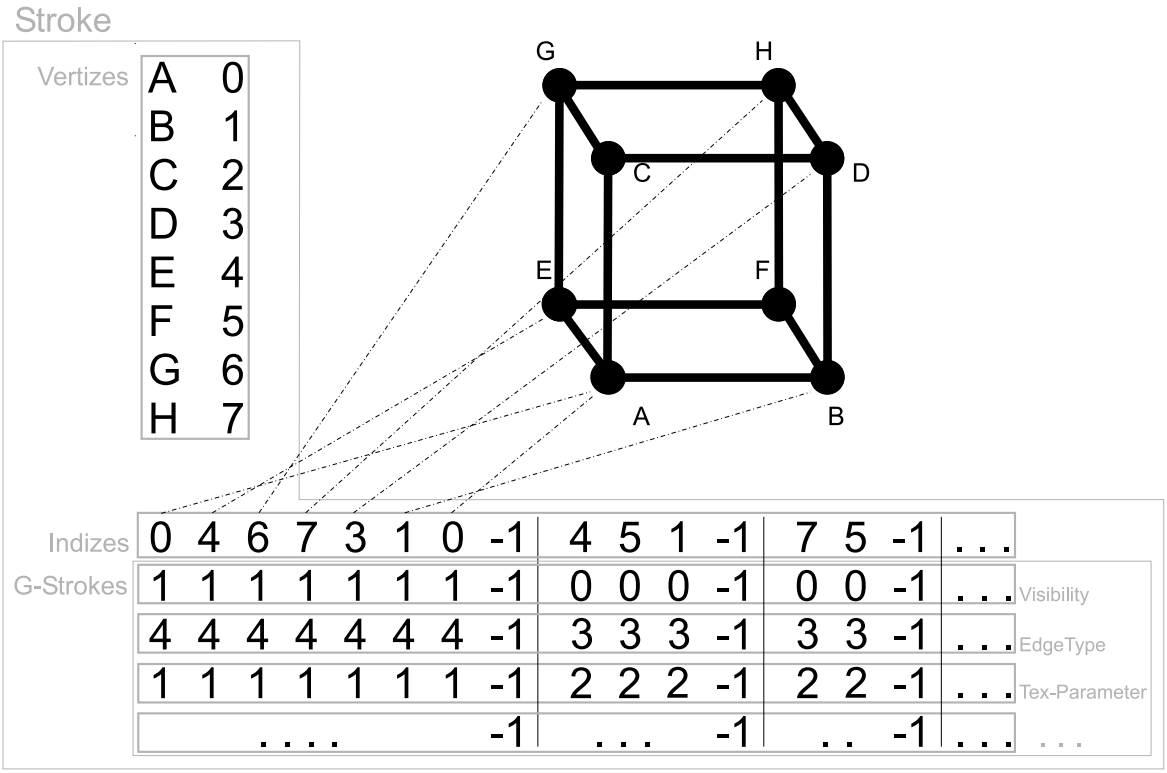


Abbildung 3.6: Der Stroke mit Vertex- und Index-Liste und seinen G-Strokes.

Für die Datenverwaltung bedeutet dies folgendes: Da der Stroke seine G-Strokes in einer Liste hält, kann er diese bei einer Veränderung traversieren und für jeden G-Stroke eine Anpassungs-Routine aufrufen. Diese muss vom jeweiligen G-Stroke durchgeführt werden. Eine derartige Vorgehensweise entspricht dem *Observer-Pattern* (GAMMA et al., 1995) und wird im folgenden Kapitel 4.2.2 detailliert beschrieben. In diesem Zusammenhang stellt sich die Frage, wodurch und wie sich der Stroke verändern kann und welche Operationen die Anpassungs-Routinen der G-Strokes diesbezüglich berücksichtigen müssen.

Auf Grund der Trennung der Daten in Stroke und G-Stroke sowie der Selbstverwaltung der G-Strokes müssen die Elemente der Stilisierungs-Pipeline nur noch den Stroke als einziges Datenpaket untereinander austauschen. Ausgehend von der Stroke-Geometrie können die Pipeline-Elemente dann bestimmte Eigenschaften, die G-Strokes, extrahieren und diese im Anschluss daran für die Erzeugung eines Stils nutzen.

Zunächst muss ein initiales Element für den Aufbau *einer* Stroke-Instanz auf Basis der extrahierten Kanten sorgen. Diese Instanz muss jedem Element der Stilisierungs-Pipeline zur Verfügung stehen können, da sonst keine Stilisierung der Kantenzüge möglich ist. Die einzelnen Pipeline-Elemente dürfen dabei die Geometrie des Strokes verändern. Dies beinhaltet das Hinzufügen, Löschen und „Spalten“ von Vertizes, womit die Trennung des Kantenzugs an einem Vertex gemeint ist. Eine Sequenz 1, 2, 3, 4, 5, -1 kann z. B. in die Sequenzen 1, 2, 3, -1 und 3, 4, 5, -1 gespalten werden. Die Anpassungs-Routinen der G-Strokes müssen diese Operationen umsetzen können. Durch das Hierarchie-Prinzip der G-Strokes sollte es möglich sein, einen Teil der Routinen auf eine höhere Ebene zu verlagern, sodass sie den Elementen der tieferen Ebene gemeinsam zur Verfügung stehen.

Desweiteren müssen die Elemente der Stilisierungs-Pipeline anhand der Stroke-Geometrie bestimmte Eigenschaften extrahieren und die G-Strokes erzeugen. Ein G-Stroke muss dabei einzigartig sein und dem Stroke als neue Eigenschaft hinzugefügt werden. Bei der Erzeugung eines G-Strokes soll es erlaubt sein, die Stroke-Geometrie durch das Hinzufügen von Eckpunkten entlang der Segmente an die Eigenschaft anzupassen, da sich der Kantenzug hierdurch visuell nicht ändert. Solche Fälle können z. B. bei der Generierung des Sichtbarkeits-G-Strokes auftreten. Wird ein Kantenzug von einer weiter vorne liegenden Fläche unterbrochen, müssen an den Schnittpunkten der Kanten neue Koordinaten berechnet werden, die das sichtbare vom verdeckten Kantensegment trennen (vgl. Abbildung 3.7).

Eine solche Hinzufüge-Operation verändert den Kantenzug nicht visuell, weshalb sie durchgeführt werden darf. Andere Auswirkungen können dagegen die Lösch- oder Spalt-Operationen auf den Verlauf des Strokes haben. Da eine durch sie hervorgerufene Veränderung nicht der ursprünglichen Abhängigkeit der Eigenschaft von der Geometrie entsprechen würde, sind sie bei der Erzeugung von G-Strokes verboten.

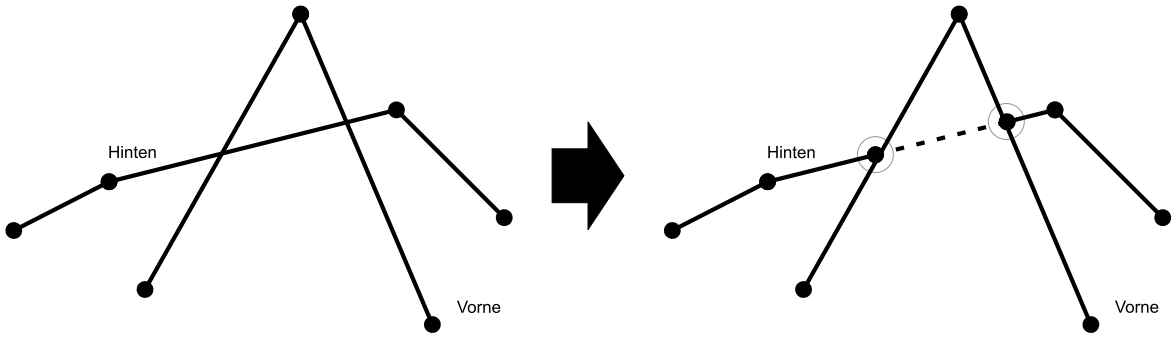


Abbildung 3.7: Notwendige Berechnung neuer Eckpunkte am Beispiel der Sichtbarkeits-Eigenschaft.

3.4 Stilgebung

Die G-Strokes sollen neben der Repräsentation von bestimmten Eigenschaften in erster Linie für die Erzeugung eines bestimmten Stils genutzt werden. Dies beinhaltet die mögliche Veränderung der Geometrie des Strokes. Bei der Erzeugung der G-Strokes war es auf Grund der Abhängigkeit der Eigenschaft von der Geometrie nicht erlaubt, den Kantenzug zu verändern. Da der Stil des Strokes jedoch von der oder den Eigenschaften abhängig ist, darf der Kantenzug bei der Stil-Erzeugung in Abhängigkeit von den G-Strokes angepasst werden. Diese Beziehung soll als *doppelte Abhängigkeit* von Stroke und G-Stroke bezeichnet werden und bezieht sich auf die G-Stroke-Erzeugung und Stroke-Stilgebung.

Da die G-Strokes von der Stroke-Instanz in einer Liste gehalten werden, haben die Elemente direkten Zugriff auf die Stroke-Eigenschaften und können sie auslesen. Zu beachten ist an dieser Stelle, dass die ausgelesenen G-Strokes ggf. angepasst werden müssen, sollte die Stroke-Geometrie verändert werden. Auch der Stroke selbst darf seine Eigenschaften auslesen und verändern.

Für die Generierung eines bestimmten Stils auf Basis der G-Strokes gibt es nun drei Möglichkeiten, wobei jeweils der Einsatz eines neuen Pipeline-Elements notwendig ist. Die erste Variante erfordert dabei den sogenannten *Filter-Knoten*. Dieser Knoten filtert bestimmte Kantensegmente bzgl. einer oder mehrerer G-Informationen und leitet sie an einen Subgraphen zur Stilisierung weiter. Der Nutzer kann die gewünschten G-Strokes interaktiv auswählen und ein Kriterium zur Filterung angeben. Zum Beispiel kann der Filter-Knoten dazu verwendet werden, sichtbare und verdeckte Kanten unterschiedlich darzustellen. Hierzu muss der Sichtbarkeits-G-Stroke ausgewählt und als Filter-Kriterium entweder „verdeckt“ oder „sichtbar“ angegeben werden. Dementsprechend filtert der Knoten entweder die verdeckten oder die sichtbaren Segmente. Diese können danach in einem Subgraph weiterverarbeitet und gerendert werden (vgl. Abbildung 3.8). Der Filter-Knoten ermöglicht damit eine modulare Stil-Kombination. Neben der Sichtbarkeits-Eigenschaft können beliebige G-Strokes nacheinander ausgewertet und stilisiert werden. Zwar wird wieder auf eine Subgrah-Struktur zurückgegriffen, doch brauchen die Berechnungsfunktionen nicht mehrfach durchgeführt

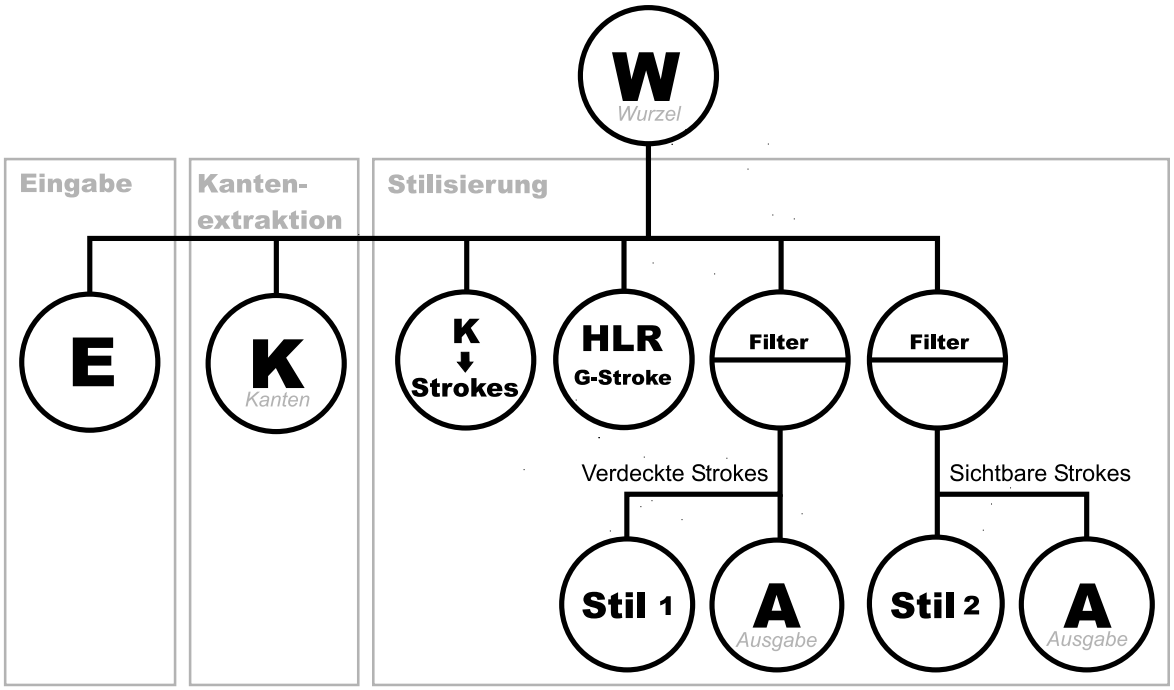


Abbildung 3.8: Möglicher Aufbau der Stilisierungs-Pipeline mit dem G-Stroke-Konzept und dem Filter-Knoten.

werden. Das HLR-Element muss z. B. nur einmal in die Stilisierungs-Pipeline gesetzt werden und den Visibility-Stroke erzeugen. Damit liegen alle für die Stilgebung notwendigen Informationen vor und müssen lediglich ausgewertet werden.

Die zweite Möglichkeit dient der Umsetzung eines spezifischen Stils. Dies erfordert den Einsatz des *Stil-Knotens*, der die vorliegenden G-Informationen nutzt und möglicherweise neue G-Strokes erzeugt. Der Stil-Knoten entspräche z. B. einem *Haloed-Lines*- oder einem *Loose*-Stil-Knoten, sehr speziellen Stilarten also. Bezogen auf das Beispiel der unterschiedlichen Darstellung sichtbarer und verdeckter Kanten muss zunächst durch das HLR-Element die Sichtbarkeits-Eigenschaft erzeugt werden, damit sie vom Stil-Knoten ausgelesen werden kann. Dieser könnte im Anschluss daran z. B. einen Color-Stroke erzeugen und die verdeckten bzw. sichtbaren Kanten mit verschiedenen Farben belegen. Ebenso könnten die einzelnen Kantensegmente unterschiedliche koloriert werden. Die G-Informationen würden somit in einem Schritt ausgewertet und um eine neue Information erweitert. Diese Variante ist im Gegensatz zu dem Szenengraphen aus Abbildung 3.8 nicht variabel und insbesondere dann von Interesse, wenn nur eine geringe Anzahl an Stilen verwendet oder der Szenengraph möglichst einfach gehalten werden soll (vgl. Abbildung 3.9).

Die dritte Möglichkeit ist eine Kombination aus Variante 1 und Variante 2. Hierbei wird ein *FilterStil-Knoten* als Pipeline-Element vorgeschlagen, wobei der Nutzer bestimmte G-Strokes zur Filterung und zur Stilgebung auswählen kann. Sind letztere noch nicht erzeugt, übernimmt der FilterStil-Knoten diese Aufgabe. Das Filter-Kriterium muss hierbei mit den

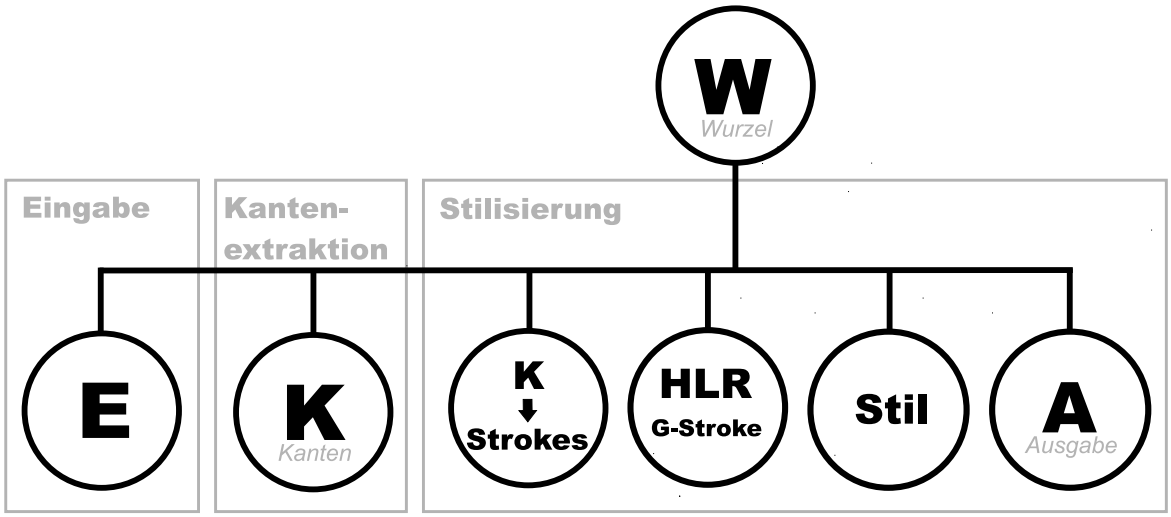


Abbildung 3.9: Möglicher Aufbau der Stilisierungs-Pipeline mit dem G-Stroke-Konzept und dem Stil-Knoten.

Stil-G-Strokes kombiniert werden. Bezüglich des verwendeten Sichtbarkeits-Beispiels könnte der Nutzer hier den Visibility-Stroke als die zu filternde Eigenschaft wählen und den Color-Stroke als die zu erzeugende Stil-Eigenschaft bestimmen. Durch das Filter-Kriterium muss dann festgelegt werden, mit welcher Farbe verdeckte und sichtbare Kanten gerendert werden sollen. Der FilterStil-Knoten stellt somit die variabelste Nutzung der G-Strokes dar. Er vereint dabei die Auswertung vorhandener sowie die Erzeugung neuer G-Strokes und die Stilisierung gefilterter Strokes. Damit ist eine modulare Stil-Kombination gegeben, die auf den Aufbau einer Subgraphen-Struktur verzichtet. Diese letzte Szenengraph-Variante ist in Abbildung 3.10 dargestellt.

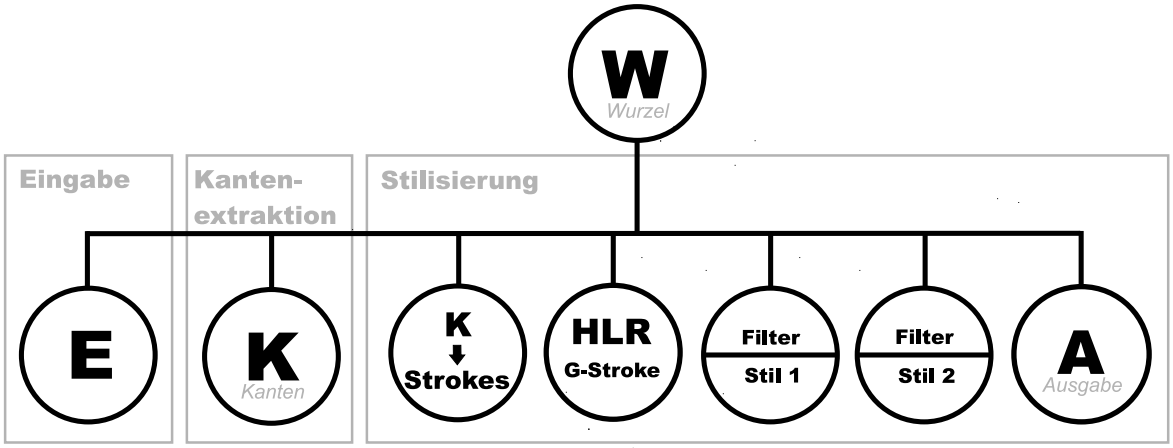


Abbildung 3.10: Möglicher Aufbau der Stilisierungs-Pipeline mit dem G-Stroke-Konzept und dem FilterStil-Knoten.

Das Konzept ist bereits an diesen einfachen Beispiel zu erkennen (vgl. im Gegensatz dazu Abbildung 3.3). Einerseits wird der Berechnungsaufwand verringert, indem Berechnungs-

Knoten (hier das HLR-Element), nur einmal aufgerufen werden müssen. Andererseits wird die Pipeline-Struktur erheblich klarer und einfacher. Dies hat nicht nur Vorteile bzgl. der Laufzeit, sondern erleichtert auch die Arbeit des Entwicklers. Das G-Stroke-Konzept hat somit zur Folge, dass die Stilisierungs-Pipeline in erster Linie für die Extraktion von geometrischen und anderen Eigenschaften zuständig ist. Erst abschließend werden diese Eigenschaften dazu genutzt, einen bestimmten Stil umzusetzen.

3.5 Zusammenfassung

Das G-Stroke-Konzept ermöglicht mehrere Dinge gleichzeitig. Zunächst wird eine logische Einteilung der Daten in Stroke-Geometrie und Stroke-Eigenschaft durch die Entwicklung einer Stroke-Instanz und einer G-Stroke-Hierarchie erreicht. Daraus ergibt sich

1. eine enge Verbindung der zeichnerischen und algorithmischen Generierung von Liniengrafiken,
2. die Umsetzung der Relation *Geometrie – Eigenschaft* durch die Speicherung der G-Strokes in einer zum Stroke gehörenden Liste und
3. die daraus folgende Selbstverwaltung der G-Strokes, welche die genannten Probleme der Stilisierungs-Pipeline löst.

Die doppelte Abhängigkeit von Stroke und G-Stroke schafft zusätzlich eine neue Aufgabenverteilung für die Pipeline-Elemente. Diese gliedert sich hauptsächlich in die Erzeugung von G-Strokes und die sich anschließende Stil-Generierung auf Basis der erzeugten Eigenschaften. Die Pipeline-Elemente können dabei folgende Aufgaben durchführen:

- Aufbau *einer* Stroke-Instanz auf Basis der extrahierten Kanten, die allen Pipeline-Elementen zugänglich sein muss,
- Operationen zur Veränderung der Stroke-Geometrie, dazu gehören:
  - das Hinzufügen,
  - das Löschen und
  - das Spalten von Vertizes,
- Extraktion von Eigenschaften und Generierung von G-Strokes auf Basis der Stroke-Geometrie,
- Auslesen und Verwenden von G-Strokes für die Erzeugung eines Stils mittels eines
  - Filter-Knotens,
  - Stil-Knotens oder
  - FilterStil-Knotens.

Das vorgestellte Konzept ermöglicht somit eine intuitive und logische Handhabung der Datenprimitive. Zusätzlich schafft es eine einfache Verwaltung der Daten und bietet durch das gleichzeitige Nutzen der Stroke-Eigenschaften die Umsetzung mehrerer Stile in einem Bild. Durch mögliche Interaktion ist es desweiteren nicht nur für neue Stil-Entwicklungen, sondern auch für eine modulare Anwendung geeignet. Das folgende Kapitel 4 setzt das G-Stroke-Konzept genau um.

Kapitel 4

Implementierung

Nachdem im letzten Kapitel das Konzept der G-Strokes entworfen und vorgestellt wurde, geht dieses Kapitel auf dessen praktische Realisierung ein. Dabei wird zunächst ein Überblick über die verwendeten Entwicklungswerkzeuge gegeben, woran sich eine kurze Beschreibung objektorientierter Entwurfskonzepte anschließt. Letztere beschreiben wiederkehrende Problematiken der Softwareentwicklung und bieten Lösungen an, die auch im Zusammenhang mit dem G-Stroke-Konzept ihre Anwendung finden. Der zweite Teil des Kapitels befasst sich schließlich mit der eingehenden Beschreibung der eigentlichen Implementierung des Stilkonzepts.

4.1 Entwicklungswerkzeuge

Das Konzept der G-Strokes ist in das Software-Projekt OPENNPAR des INSTITUTS FÜR SIMULATION UND GRAFIK der UNIVERSITÄT MAGDEBURG eingebunden. OPENNPAR ist eine auf OPEN INVENTOR basierende NPR-Bibliothek, die in der Programmiersprache C++ entwickelt wird. Da die Entwicklungswerkzeuge für die enge Verbindung zwischen Entwurf und Realisierung grundlegend sind, sollen sie in diesem Abschnitt kurz vorgestellt werden. Begonnen wird dabei mit einer Einführung in objektorientierte Prinzipien, die sich für die Umsetzung des G-Stroke-Konzepts sehr gut eignen.

4.1.1 Objektorientierte Prinzipien am Beispiel von C++

Objektorientierung ist ein Entwicklungsparadigma, bei dem es einerseits um die Kapselung von Strukturen und Eigenschaften in Objekten und andererseits um die Interaktion und Kommunikation der Objekte untereinander geht (u. a. BOOCH, 1995). Objektorientierte Prinzipien bieten insbesondere für die Umsetzung abstrakter und konkreter Konzepte eine Reihe von Realisierungstechniken, die für die Umsetzung des G-Stroke-Konzepts wesentlich waren. An dieser Stelle sollen daher die Möglichkeiten der Datenabstraktion, der Vererbung und des Polymorphismus beschrieben werden.

C++ ist eine in erster Linie objektorientierte Programmiersprache, die zu Beginn der achtziger Jahre von BJARNE STROUSTRUP entwickelt wurde und auf C basiert (STROUSTRUP, 2000). Da sie durch OPENNPAR als Entwicklungswerkzeug vorgegeben war, werden die objektorientierten Prinzipien am Beispiel dieser Sprache erläutert.

Die *Datenabstraktion* wird bei C++ durch den Einsatz von *Klassen* erreicht, mit welchen sich konkrete und abstrakte Datentypen implementieren lassen (STROUSTRUP, 2000, Teil 2). Eine Klasse repräsentiert dabei ein bestimmtes Konzept mit seinen Eigenschaften, auf welche über die Schnittstelle[1] der Klasse zugegriffen werden kann. Ein konkreter Datentyp soll dabei einem konkreten Konzept entsprechen, wohingegen ein abstrakter Datentyp der Umsetzung eines abstrakten Konzepts dient. Eine konkrete Klasse kann im Programm instanziiert und ihre Funktionalität direkt benutzt werden. Eine abstrakte Klasse repräsentiert die Schnittstelle zu mehreren Implementierungen eines gemeinsamen Konzepts. Sie kann nicht instanziiert, ihre Funktionalität jedoch durch Vererbung realisiert werden (vgl. STROUSTRUP, 2000, Teil 4).

Mittels öffentlicher *Vererbung* können in C++ Beziehungen zwischen Klassen hergestellt werden. Die öffentliche Vererbung[2] stellt dabei die „ist-ein“-Beziehung dar (MEYERS, 1998, S. 193 ff.): Eine Klasse `A` kann von einer Klasse `B` abgeleitet werden und erbt damit alle Eigenschaften von `B`, `A` *ist-ein* `B`. Gemeinsame Basisklassen bedeuten gemeinsame Eigenschaften, `A` *funktioniert wie* `B` (SUTTER, 2000, Kap. 4). Im Gegensatz dazu steht die private Vererbung,[3] welche die „ist-implementiert-mit“-Beziehung repräsentiert. Das bedeutet, dass nur die Funktionalität der Basisklasse geerbt werden soll und keine Beziehung zwischen den Klassen besteht (MEYERS, 1998, S. 233 ff.). Laut SUTTER sollte nur dann öffentlich vererbt werden, wenn die abgeleiteten Klassen selbst wiederverwendet werden sollen und nicht, um in der abgeleiteten Klasse Funktionalitäten der Basisklasse zu verwenden (SUTTER, 2000, S. 97). Die Wiederverwendung der abgeleiteten Klassen findet z. B. dann statt, wenn die Basisklasse polymorph genutzt wird.

Polymorphismus beschreibt die Möglichkeit, *eine* Basisformulierung einer Methode anzugeben und auf verschiedene Typen anzuwenden. Über den Aufruf der Basisformulierung kann dann jeweils die Abarbeitung der typbedingten Formulierung angestoßen werden. *Laufzeit-Polymorphismus* tritt während der Laufzeit auf und wird durch virtuelle Funktionen ermöglicht. Die Deklaration einer Basis-Funktion als *rein-virtuell* verfolgt den Zweck, der abgeleiteten Klasse nur die Schnittstelle der Basisfunktion zu vererben. Die abgeleitete Klasse muss die Funktion selbst definieren bzw. implementieren. Eine abstrakte Klasse besitzt daher mindestens eine rein-virtuelle Funktion. Demgegenüber verfolgt die Deklaration einer Basis-Funktion als *virtuell* den Zweck, der abgeleiteten Klasse sowohl die Schnittstelle als auch eine Standard-Definition zu vererben, die sie selbst neu definieren *kann*. Die Deklaration einer Basis-Funktion als *nicht-virtuell* verfolgt schließlich den Zweck, der abgelei-

1 Die Schnittstelle einer Klasse ist i. A. der öffentliche Deklarationsbereich, über welchen alle (anderen) Objekte auf die Funktionalität der Klasse zugreifen können.

2 Öffentliche Vererbung wird im Quelltext durch `class A : public B {...}` dargestellt.

3 Private Vererbung wird im Quelltext durch `class A : private B {...}` dargestellt.

teten Klasse die Schnittstelle sowie eine verbindliche Definition der Funktion zu vererben (MEYERS, 1998, S. 200 ff.).

C++ erlaubt es, Zeiger `Z` vom Typ der Basisklasse `B` auf Instanzen abgeleiteter Klassen A_i verweisen zu lassen. Wenn z. B. über `Z` auf eine A_i-Instanz gezeigt und eine virtuelle Funktion aus `B` aufgerufen wird, so wird die Implementierung der Funktion von A_i ausgeführt und nicht die Implementierung aus `B`.[4] Hierdurch kann die Funktionalität *mehrerer* Typen mit *einer* Variable verwaltet werden. Mit Hilfe von Laufzeit-Typinformationen (*run-time type information (RTTI)*) kann desweiteren festgestellt werden, auf welchen abgeleiteten Typ der Zeiger vom Typ der Basisklasse zeigt.

Übersetzungs-Polymorphismus tritt während der Übersetzung, also vor der Laufzeit des Programms auf und wird bei C++ durch den Einsatz von *Templates* ermöglicht. Ein Template `T` formuliert das Verhalten einer Funktion oder einer Klasse für einen Parameter `P`, einen variablen Datentyp. Um `T` instanziieren zu können, muss `P` durch einen allgemeinen oder benutzerdefinierten Datentyp `D` ersetzt werden. Damit entsteht ein neuer Datentyp `TD`, der die Eigenschaften des Templates bezogen auf den Datentyp `D` besitzt. `TD` kann wie ein konkreter Datentyp verwendet werden und wird vom Compiler während der Übersetzungszeit erzeugt. Dementsprechend wird auch hier ein grundlegendes Konzept einmalig formuliert und typbedingt ausgeführt (STROUSTRUP, 2000, Kap. 13).

Datenabstraktion, Vererbung und Polymorphismus sind für die Umsetzung des G-Stroke-Konzepts entscheidend und ermöglichen eine sehr enge Verbindung zwischen Konzept und Implementierung. Die G-Strokes sind hierarchisch angeordnete Datentypen, die in abstrakten und konkreten Klassen realisiert werden. Die objektorientierte Vererbung ermöglicht die Umsetzung der G-Stroke-Hierarchie und dient desweiteren als Grundlage für die polymorphe Nutzung der Klassen, wodurch sie einheitlich verwaltet werden können. Über diese grundlegenden Konzepte hinaus basiert die Implementierung der G-Strokes vor allem auf den Bibliotheken OPEN INVENTOR und OPENNPAR, die in den folgenden Abschnitten beschrieben werden.

4.1.2 Open Inventor als Grafikbibliothek

OPEN INVENTOR ist eine objektorientierte 3D-Grafikbibliothek, mit der interaktive 3D-Grafikapplikationen auf Basis von OPENGL entwickelt werden können (WERNECKE, 1994a,b). Hierzu werden jegliche Informationen, die die darzustellende Szene betreffen, in einer Szenendatenbank gespeichert. Hauptprimitive dieser Datenbank sind die sogenannten *Szenenobjekte* oder *Knoten* (`SoNode` und Derivate). Durch die Knoten können alle Objektinformationen repräsentiert werden. Neben Form- (`SoShape`), Material- (`SoMaterial`) oder Tex-

4 Virtuelle Funktionen werden im Gegensatz zu nicht-virtuellen Funktionen dynamisch an den Typ, auf den verwiesen wird, und nicht statisch an den Typ, von dem verwiesen wird (dem Typ des Zeigers), gebunden. Das ist die Ursache für Laufzeit-Polymorphismus.

tureigenschaften (`SoTexture2`) gehören auch Transformationen (`SoTransform`) für die Positionierung der Modelle und Gruppenknoten (`SoGroup`) für die Strukturierung mehrerer Knoten zu den möglichen Szenenobjekten.

Die Anordnung der Knoten in einer bestimmten Reihenfolge führt zu einer Graphstruktur, dem sogenannten *Szenengraph*. Dieser wird in der Szenendatenbank gespeichert und beginnt mit einem Wurzelknoten, „unter“ dem die restlichen Szenenobjekte platziert sind. Die *Traversierung* des Szenengraphen wird durch den Wurzelknoten ausgelöst und dient der Ausführung bestimmter Funktionen in den einzelnen Knoten. So wird die Szene z. B. während einer Traversierung gerendert und auf dem Monitor angezeigt. Die Anordnung der Knoten ist dabei variabel, ihre Abarbeitung wird bei der Wurzel beginnend als *preorder*-Traversierung, also von oben nach unten und von links nach rechts, durchgeführt.

Von grundlegender Bedeutung für diese Diplomarbeit sind folgende von OPEN INVENTOR angebotenen Konzepte:

Knoten (*Nodes*) repräsentieren die einzelnen Informationen über das geometrische Objekt, z. B. Material-, Oberflächen- oder Beleuchtungseigenschaften. Besondere Bedeutung haben dabei der `SoGroup`- und der davon abgeleitete `SoSeparator`-Knoten, die die Zusammenfassung mehrerer Szenenobjekte zu einem Graph ermöglichen. Mit ihnen kann der eigentliche Szenengraph wie auch ein oder mehrere Subgraphen zusammengestellt werden. Zusätzlich ermöglicht der `SoSeparator`, dass alle an ihm hängenden Knoten unabhängig verwaltet werden. Das heißt, dass jede Änderungen bzgl. Objektmaterial, -beleuchtung oder -position nur lokale Auswirkungen auf die Knoten des (Sub)-Graphen haben. Um die Einstellungen der Szenenobjekte anzupassen, ist es desweiteren möglich, bestimmte Parameter in sogenannten *Feldern* der Knoten zu setzen.

Felder (*Fields*) sind ein Mechanismus zum Kapseln allgemeiner Datentypen wie Integer oder Float sowie INVENTOR-eigener *Szenenbasis*-Typen wie *SbBool* etc. Sie ermöglichen die interaktive Anpassung der Knoten-Parameter. Die Veränderung letzterer kann dabei ständig überwacht werden und eine entsprechende *Aktion* auslösen, die zu einer erneueten Traversierung und somit einer Verarbeitung der Änderungen führt.

Aktionen (*Actions*) werden i. d. R. an der Wurzel des Szenengraph ausgelöst und traversieren diesen. Dabei rufen sie für jeden Knoten eine `update()`-Routine auf, in der sie den Knoten Zugriff auf den aktuellen *Traversierungszustand* ermöglichen. Der Traversierungszustand ist wiederum für die Verwaltung der *Elemente* des Szenegraphen zuständig.

Elemente (*Elements*) ermöglichen den Datenaustausch zwischen den einzelnen Szenenobjekten. Wenn ein Knoten traversiert bzw. von einer Aktion erreicht wird, kann er sich über den aktuellen Traversierungszustand Zugriff auf die einzelnen Elemente verschaffen. Aus diesen Elementen können dann Daten geholt, angepasst und wieder gesetzt werden. Trifft die Aktion auf den nächsten Knoten, kann dieser mit den angepassten Daten weiterarbeiten. Durch den `SoSeparator` wird eine Datenhierarchie geschaf-

fen. Das bedeutet, dass die Elemente eines `SoSeparator`-Subgraphs immer mit einer Kopie der höheren Hierarchie arbeiten, um die ursprünglichen Daten beim Verlassen des Subgraphs erhalten zu können.

Diese Konzepte bieten somit eine ideale Grundlage für die Umsetzung einer Stilisierungs-Pipeline. Die Beschreibung der dreidimensionalen Daten kann sehr einfach mit dem Szenengraphen-Prinzip von OPEN INVENTOR geschehen. Diese Funktionalität der Szenenobjekte kann dabei durch Vererbung in neue Knoten-Klassen übernommen und so erweitert werden – zum Beispiel um nicht-fotorealistische Methoden zur Kantenextraktion und Stilisierung, wie in der Bibliothek OPENNPAR erfolgt. Diese wird daher im folgenden vorgestellt.

4.1.3 OpenNPAR als NPR-Bibliothek

OPENNPAR ist eine 3D-Grafikbibliothek, die Algorithmen und Methoden zur Erzeugung nicht-fotorealistischer Bilder anbietet.[5] Zurzeit wird OPENNPAR in erster Linie für die Generierung von Liniengrafiken weiterentwickelt, die u. a. im Bereich der Visualisierung medizinischer Datensätze Anwendung finden (TIETJEN, 2004). Demzufolge gibt es bereits eine Reihe von Klassen zur Silhouetten- und Merkmalskantenextraktion sowie zur Stilisierung der Strokes. Die wesentlichen Komponenten sowie die Abarbeitung der Stilisierungs-Pipeline sollen im folgenden kurz vorgestellt werden.

OPENNPAR verfügt über eine Reihe von Modifikationstechniken zur Stilisierung von triangulierten Polygonmodellen. Das jeweils geladene Modell muss dabei zunächst in eine `Winged Edge`-Datenstruktur konvertiert werden, damit die für die Berechnung der Silhouetten- und Merkmalskanten notwendigen Konnektivitätsbeziehungen der Primitive des 3D-Modells vorliegen (siehe BAUMGART, 1972). Anschließend können die Silhouetten- und Merkmalskanten berechnet werden. Dazu bietet OPENNPAR neben der trivialen Berechnungsmethode auch das schnellere Verfahren von BENICHOU und ELBER (1999) (vgl. Abschnitt 2.2.5). Mittlerweile ist zusätzlich die Berechnung der Subpolygonsilhouette nach HERTZMANN und ZORIN (2000) implementiert worden, deren Ergebnisse jedoch in dieser Diplomarbeit nicht mehr verwendet werden konnten.

Die für die Kantenextraktion zuständigen Szenenobjekte `SoGenerateEdgesSilhouette` (für die Berechnung der Silhouettenkanten) und `SoGenerateEdgesFeatureLines` (für die Berechnung von scharfen Kanten und Begrenzungskanten) sind zwei abgeleitete Klassen der abstrakten Basisklasse `SoGenerateEdges`. Letztere sorgt für eine einheitliche Speicherung der einzelnen Kantensegmente (bestehend aus zwei Koordinaten) in eine Vertex- und eine Index-Liste und führt keine Berechnungen durch. Damit diese Listen für alle weiteren Knoten zur Verfügung stehen, wird jeweils ein Zeiger auf die Vertex-Liste in das `SoCoordinateElement` und ein Zeiger auf die Index-Liste in das `SoLineCoordinate-`

5 OPENNPAR steht für *Open Non-Photorealistic Animation and Rendering*. Weitere Informationen finden sich unter http://www.opennpar.org/.

`IndexElement` gesetzt. Alle Knoten können somit auf die zuletzt aktualisierten Daten zugreifen. Soll die Stroke-Geometrie verändert werden, verhalten sich die Pipeline-Knoten folgendermaßen: Der jeweilige Knoten holt sich die beiden Zeiger auf die Datenlisten als Referenz für den aktuellen Stroke-Verlauf. Die Anpassung des Kantenzuges bzgl. der jeweiligen Knoten-Funktionalität wird in einer neuen lokalen Vertex- und Index-Liste gespeichert. Abschließend setzt der Knoten zwei auf die lokalen Listen verweisenden Zeiger in das Element, so dass der nächste Pipeline-Knoten auf die aktuelle Stroke-Geometrie zugreifen kann (vgl. Abbildung 4.1). Die lokale Kopie ist erforderlich, weil die Elemente nicht

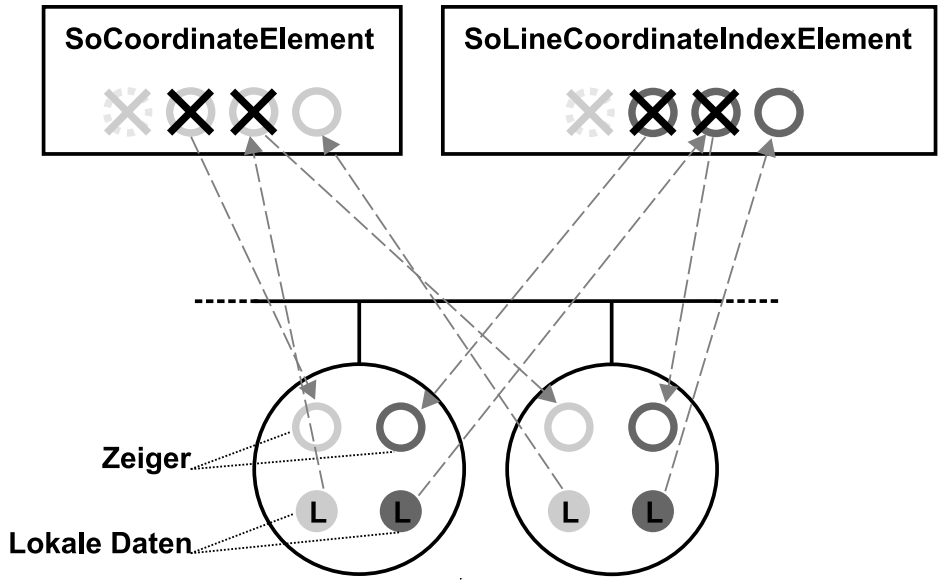


Abbildung 4.1: Die Verarbeitung der Stroke-Geometrie durch Ausnutzung von OPEN-INVENTOR-Elementen bei OPENNPAR. Die Elemente halten dabei immer nur Zeiger auf Datenstrukturen, wobei die Datenstrukturen in den einzelnen Pipeline-Knoten liegen und dort verwaltet werden. Daher löscht das Überschreiben des Zeigers nicht die Datenstruktur, sondern nur den Zeiger selbst.

für eine Datenspeicherung bzw. das Kopieren von Daten entworfen wurden. Soll nun eine Geometrie-Veränderung in einem von `SoSeparator` begrenzten Subgraph erfolgen, könnte der ursprüngliche Datensatz ohne lokale Kopie in einem vorherigen Knoten nicht wiederhergestellt werden.[6]

Nachdem die Kanten in Vertex- und Index-Liste vorliegen, können sie stilisiert werden. Die einzelnen Verfahren, die zu einem bestimmten Stil führen, sind dabei als Szenenobjekte implementiert worden und von der Basisklasse `SoLineModifier` abgeleitet. Letztere bietet z. B. Funktionen für die Projektion oder die Pixelmanipulation der Strokes an. OPENNPAR bietet ein großes Angebot an Kantenmodifikationen. Dazu gehören die Knoten

6 Da die Vertex- und Index-Elemente nur einen Zeiger auf die jeweiligen Listen speichern, würde eine Veränderung der Geometrie in einem durch `SoSeparator` getrennten Subgraph ohne Kopie zu einer konstanten Änderung der Daten führen. Dies widerspräche jedoch dem Sinn des `SoSeparator` und wird daher durch die Kopie verhindert.

- `SoLineConcatenator` für die Verknüpfung der Kanten,
- `SoLineParameterization` für die Stroke-Parameterisierung,
- `SoLineHiddenLineRemover` für die Bestimmung der sichtbaren Stroke-Segmente,
- `SoLineThickener` für die Festlegung der Linienbreite,
- `SoLineCoordinateOrientation` für die Festlegung der Textur-Orientierung,
- `SoLineSubdivider` für die Unterteilung der Segmente in Teilstücke,
- `SoLinePerturbator` für die Störung der Kantenzüge sowie
- `SoLinePurifier` für die Entfernung von Kantenartefakten.

Die Kantenzüge können schließlich von dem Knoten `SoLineShape` mit einer Textur belegt und ausgegeben werden. Neben der Vertex- und der Index-Liste verwaltet OPENNPAR eine Parameter-Liste, die während der Stroke-Parameterisierung erzeugt wird und die Anpassung aller betroffenen Knoten erfordert. Der Zugriff auf die Daten erfolgt dabei wieder als Zeiger über das `SoParameterizationElement`. Daneben gibt es weitere Elemente, u. a. das `SoGLNormalElement`, das `SoLineThicknessElement` oder das `SoOrientedCoordinateElement`, das die Textur-Orientierungsparameter hält. Die Geometrie des Strokes wird dabei hauptsächlich durch das Hinzufügen von Eckpunkten verändert. Einzig der `SoLinePurifier`-Knoten entfernt bestimmte Eckpunkte zur Begradigung des Kantenzugs. Ein möglicher Szenengraph-Aufbau ist in Abbildung 4.2 zu sehen.

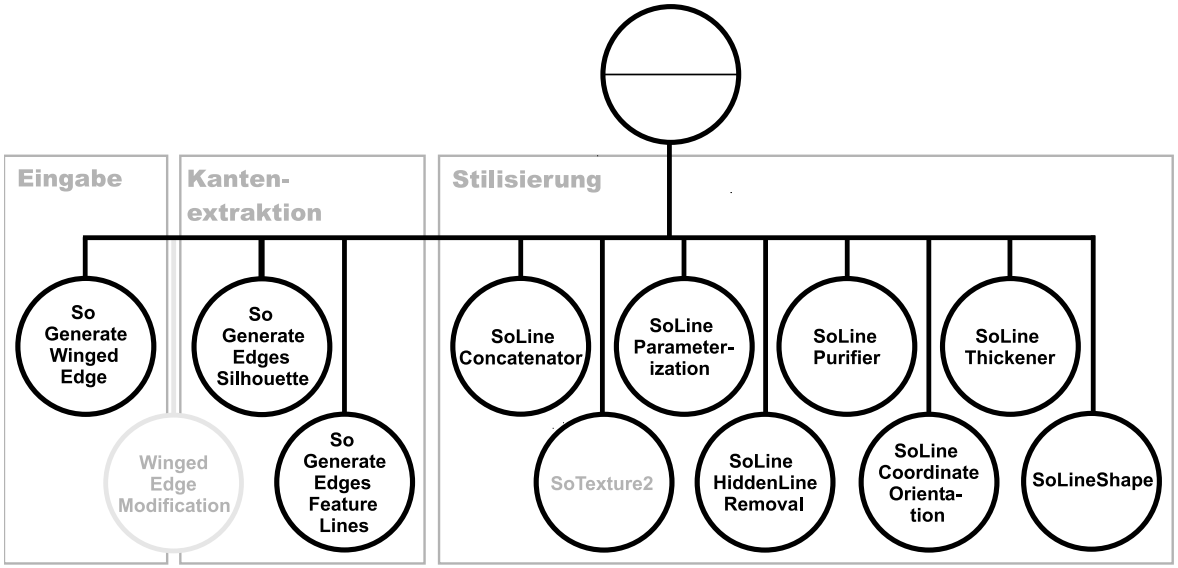


Abbildung 4.2: Der Aufbau des Szenengraphs aus der Beispiel-Applikation `DemoWE`, die die aktuellen Stilmöglichkeiten von OPENNPAR demonstriert. Nach dem Aufbau der Winged Edge-Datenstruktur gibt es noch mehrere Modifikationsknoten, welche z. B. für eine höhere Tesselierungsauflösung sorgen. Diese sind jedoch nicht für die Stilisierungs-Pipeline von Interesse, weshalb sie in dieser Grafik vernachlässigt wurden. Der `SoTexture2`-Knoten ist ein OPEN INVENTOR-Knoten, der die Textur festlegt und für die Parameterisierung notwendig ist.

Für den Fall, dass die extrahierten Kanten nur angezeigt und nicht texturiert oder weiter stilisiert werden sollen, bietet OPENNPAR zusätzlich die Klasse `SoDisplayStrokes`. Dieser Knoten stellt die extrahierten Kanten lediglich mit OPENGL-Linien dar und erzielt somit eine sehr einfache Darstellung.

Die in Abschnitt 3.1.2 diskutierten Probleme bzgl. der Datenverarbeitung der Stilisierungs-Pipeline sind charakteristisch für OPENNPAR und treten z. B. in Form der Parameter-Liste auf. Dies hemmt die Weiterentwicklung sowie die Anwendung (vgl. TIETJEN, 2004). Da die Bibliothek nach dem Aufbau von Vertex- und Index-Liste nicht mehr zwischen den verschiedenen Kantentypen unterscheiden kann, sind alle Kanten gleichmäßig von jeglicher Modifikation betroffen. Desweiteren ist der HLR-Knoten nur für das Entfernen von verdeckten Kanten konzipiert. Der Aufbau verzweigter Szenengraphen könnte daher nur in einer festgelegten Reihenfolge (sichtbare und verdeckte Kanten mit Textur 1 zuerst, nur sichtbare Kanten mit Textur 2 zuletzt) erfolgen, wobei desweiteren darauf geachtet werden müsste, dass Textur 2 mindestens so breit wie Textur 1 ist. Eine Umstrukturierung der Stilisierungskomponenten der Bibliothek ist daher notwendig.

Die in den letzten drei Abschnitten beschriebenen Werkzeuge bieten somit eine Reihe von Realisierungsmöglichkeiten, die für die Umsetzung des G-Stroke-Konzepts vorteilhaft und hilfreich sind. Dennoch beinhaltet das entworfenene Verwaltungssystem mit seinen Primitiven Stroke und G-Stroke grundlegende Eigenschaften, die nicht auf der rein technischen Ebene umgesetzt werden können. Hierzu gehört einerseits die Einzigartigkeit der beiden Primitive, sowie die Anpassung aller G-Strokes and den Kantenzug. Diese beiden Eigenschaften können jedoch mit entsprechenden Entwurfsmustern realisiert werden, die im folgenden Abschnitt erläutert werden.

4.2 Entwurfsmuster

Ein Entwurfsmuster beschreibt ein in der Softwareentwicklung wiederholt auftretendes Problem und gibt dafür eine Lösung an (GAMMA et al., 1995). Auch das in Kapitel 3 vorgestellte Konzept der G-Strokes beinhaltet gewisse Einschränkungen, die eine genaue Umsetzung erfordern. So wurde in Abschnitt 3.2 festgelegt, dass Stroke und G-Stroke einzigartig sein sollen. Die softwaretechnische Umsetzung dieser Festlegung bzw. dieses „Problems“ wird durch das *Singleton-Pattern* beschrieben. Desweiteren beinhaltet das Konzept eine logische Beziehung zwischen Stroke und G-Stroke. Eigenschaften und Geometrie müssen dabei immer übereinstimmen und aneinander angepasst werden, um die Beziehung aufrecht erhalten zu können (vgl. Abschnitt 3.3). Die Realisierung dieses Problems wird durch das *Observer-Pattern* gelöst.

4.2.1 Singleton-Pattern

Das hier beschriebene Singleton-Pattern gehört zu den Pattern, die Lösungen für Probleme bei der Objekterzeugung anbieten. Dieses Pattern beschreibt, wie die Anzahl der Instanzen eines Datentyps bzw. einer Klasse auf eine einzige beschränkt werden und diese dennoch überall zur Verfügung gestellt werden kann (GAMMA et al., 1995, Kap. 3). Damit bietet es eine Lösung für die Eigenschaft der Einzigartigkeit von Stroke und G-Stroke. Beide wurden als eindeutige und einzigartige Primitive entworfen und sollen als solche implementiert werden. Das Singleton-Pattern ermöglicht diese Umsetzung nun durch die Beschränkung der Instanziierung der jeweiligen Objekte. Das Muster wird hier am Beispiel der exemplarischen Klasse `Singleton` beschrieben.

Damit nur eine einzige Instanz des Typs `Singleton` erstellt werden kann, muss zunächst die Konstruktionsmöglichkeit der Klasse eingeschränkt werden: Der Konstruktor wird hierfür in den *privaten* Deklarationsbereich der Klasse platziert. So können weder fremde noch abgeleitete Klassen eine Instanz von `Singleton` erzeugen, nur `Singleton` selbst hat Zugriff auf den Konstruktor. Wie kann nun eine Instanz erzeugt werden, ohne den Konstruktor aufrufen zu müssen? Die Lösung liefert der Einsatz von `static`-Funktions- und Variablen-Deklarationen. Als *statisch* deklarierte Variablen werden nur einmal erzeugt und initialisiert. Sie sind Teil einer Klasse, jedoch nicht Teil einer Instanz der Klasse. Entsprechend bewirken statische Funktionsdeklarationen, dass die Funktionen einmal und unabhängig von einem Objekt existieren. Auch sie sind Teil der Klasse und haben somit Zugriff auf die Klassenelemente, sie werden jedoch nicht für eine bestimmte Instanz aufgerufen (STROUSTRUP, 2000, S. 242 f.). Um eine Instanz der Klasse `Singleton` zu erhalten, ohne den Konstruktor aufrufen zu müssen, bedient sich die Klasse des beschriebenen `static`-Prinzips. `Singleton` erhält eine öffentliche statische Funktion „`Singleton& getInstance()`“, die beim ersten Aufruf eine lokale statische Variable `S` vom Typ `Singleton` erzeugt und als Referenz an das aufrufende Objekt zurückgibt. `S` steht bei jedem weiteren Aufruf auf Grund seiner `static`-Eigenschaft als Rückgabewert zur Verfügung.

Hierdurch ist die Objektbeschränkung auf eine festgelegte Anzahl von Instanzen, in diesem Fall eine einzige, möglich. Die beschriebene Umsetzung entspricht dabei dem *Meyers-Singleton-Pattern* und ermöglicht durch Einsatz der lokalen statischen Variable eine elegantere Lösung als das klassische Verfahren, bei dem eine statische Zeigervariable verwendet wird (MEYERS, 1999, Abschnitt 5.2).

4.2.2 Observer-Pattern

Das Observer-Pattern gehört zu den Verhaltens-Pattern, die Kommunikations- und Verhaltensbeziehungen zwischen Objekten beschreiben. Es beschreibt die Abhängigkeits-Beziehung zwischen einer Reihe von Objekten (den *Observern*) zu einem einzigen Objekt (dem

Subject). Die Beziehung zeichnet sich dadurch aus, dass alle Observer vom Subject abhängig sind. Sobald sich der Zustand des Subjects ändert, müssen sich die Observer entsprechend anpassen (GAMMA et al., 1995, Kap. 5). Abbildung 4.3 veranschaulicht die Kommunikation der beiden Komponenten. Damit bietet dieses Muster eine Lösung für die entworfene Um-

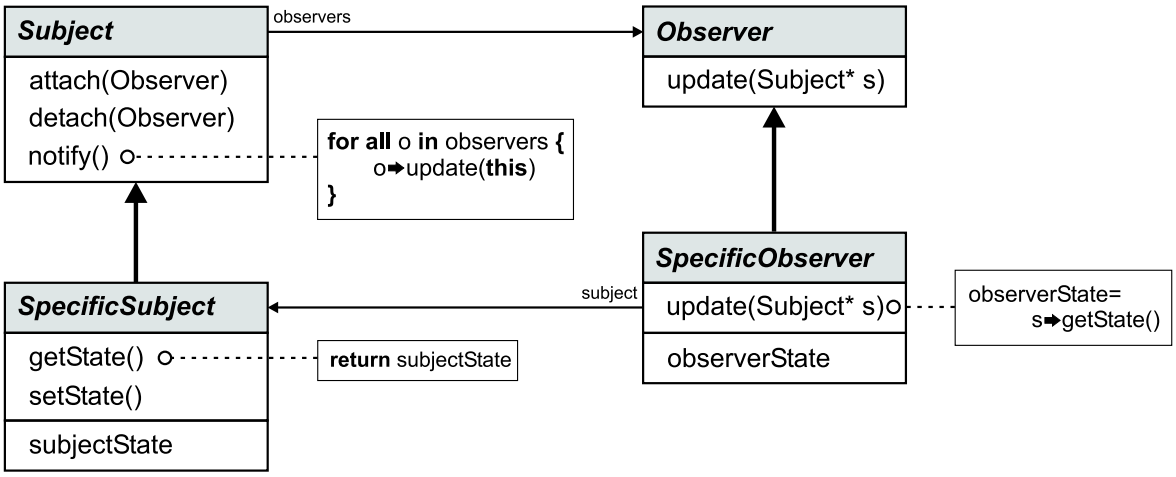


Abbildung 4.3: Die Beziehung zwischen Subject und Observer.

setzung der G-Stroke-Verwaltung, die unabhängig von den einzelnen Stilisierungs-Pipeline-Knoten erfolgen soll. Da sich die G-Strokes automatisch an jegliche Veränderung des Strokes anpassen müssen, entsprechen sie den hier vorgestellten Observern, welche sich immer auf den aktuellen Zustand des Subjects beziehen.

Das Observer-Pattern basiert auf vier exemplarischen Klassen. Dabei repräsentieren die beiden abstrakten Klassen `Subject` und `Observer` die Schnittstelle des umzusetzenden Entwurfsmusters. Die konkreten Klassen `SpecificSubject` und `SpecificObserver` entsprechen einem konkreten Problem und implementieren das Entwurfsmuster diesbezüglich. Um die Abhängigkeit zwischen beiden herstellen zu können, muss zunächst die `Subject`-Klasse über eine Liste `observers` von `Observer`-Objekten verfügen. Diese werden ihr von der Funktion `attach(P)`[7] hinzugefügt und von der Funktion `detach(P)` entfernt. Die Funktion `notify()` wird aufgerufen, sobald sich der Zustand von `Subject` ändert. Hier entsteht der Zusammenhang zwischen `Subject` und `Observer`. `notify()` ruft für jedes Listenelement die `Observer`-Funktion `update(P)` auf und übergibt sich dabei als Argument. Hiermit haben die Observer Zugriff auf den aktuellen Zustand des Subjects. Die `Subject`-Funktionen `getState()` und `setState()` sind diesbezüglich als exemplarisch zu verstehen.

7 Die Formatierung von Quelltext-Begriffen soll an dieser Stelle konvetionell festgelegt werden, um eine leichte Lesbarkeit zu ermöglichen. Damit im weiteren Verlauf des Texts zwischen Funktions- und Variablennamen einfach unterschieden werden kann, werden Funktionsnamen immer von zwei geschweiften Klammen „()“ gefolgt sein. Benötigt die Funktion weitere Parameter, beinhalten die Klammern einfachheitshalber ein „P“, welches für „Parameter“ steht. Somit müssen die Parameter nur bei Bedarf aufgeführt werden, was den Lesefluß erleichtern und eine korrekte Wiedergabe der Begriffe ermöglichen soll.

Die vorgestellte Wechselwirkung charakterisiert das Observer-Pattern und ermöglicht eine Abhängigkeits-Beziehung zwischen zwei verschiedenen Klassentypen, von denen sich der observierende an den aktuellen Zustand des zu observierenden anpasst.

4.3 Umsetzung des Konzepts

Nachdem die verwendeten Implementierungswerkzeuge und Hilfsmittel in den beiden letzten Abschnitten vorgestellt wurden, wird hier die Realisierung des G-Stroke-Konzepts beschrieben. Die exakte Umsetzung der einzelnen konzeptuellen Komponenten steht dabei im Vordergrund. Hierzu gehört

- die Umsetzung der Primitive Stroke und G-Stroke in den Klassen `SbStroke` und `SbGStroke`,
- die Umsetzung der Einzigartigkeit von Stroke und G-Stroke durch Implementierung des Singleton-Patterns,
- die Umsetzung der G-Stroke-Hierarchie mit Hilfe von Vererbung,
- Umsetzung der Operationen zur Veränderung der Stroke-Geometrie und damit einhergehend
- die Verwaltung der G-Strokes durch Einsatz des Observer-Patterns,
- die Anpassung der vorhandenen Pipeline-Knoten im Hinblick auf die Nutzung eines einzigen Datenpakets (des Strokes) sowie die Generierung und Handhabung der G-Strokes und
- die Implementierung neuer Pipeline-Knoten zur Filterung der G-Stroke-Daten für die Stilisierung.

Die zu entwickelnden Klassen sollen sich durch eine leichte Handhabung auszeichnen sowie für eine einfache Weiterentwicklung förderlich sein und somit einem klar strukturierten Muster entsprechen. Dieses soll bewirken, dass auch zukünftige Entwicklungen der konzeptuellen Umsetzung entsprechen. Im weiteren Verlauf dieses Abschnitts wird nun zunächst auf die Entwicklung der Klassen `SbStroke` und `SbGStroke` eingegangen, um im Anschluss daran ihre Anwendung und die damit einhergehende Anpassung der Pipeline-Knoten sowie der Stilisierung zu schildern. Abschließend werden die wesentlichen Punkte der Implementierung zusammengefasst.

4.3.1 Die Klasse SbStroke

Die Umsetzung des Strokes erfolgt in der Klasse `SbStroke`, deren Namens-Präfix *Sb* in Anlehnung an die Szenenbasis-Objekte von OPEN INVENTOR gewählt wurde. Laut Definition aus Abschnitt 3.2 muss der Stroke als *geometrisches Primitiv*

- einzigartig sein,
- über die Vertex- und die Index-Liste verfügen,
- eine Liste von G-Strokes besitzen und
- Operationen für die Veränderung der Stroke-Geometrie anbieten.

Diese Eigenschaften setzt die Klasse `SbStroke` um. Der letzte Punkt sowie die Tatsache, dass der Stroke das geometrische Primitiv des Konzepts repräsentiert, an das sich alle G-Strokes anpassen müssen, beinhaltet desweiteren, dass `SbStroke` dem *Subject* des Observer-Patterns entspricht (vgl. Abschnitt 4.2.2). Außerdem muss ermöglicht werden, dass der Stroke allen Stilisierungs-Pipeline-Knoten zur Verfügung stehen kann. Diese Funktionalitäten werden daher ebenfalls in den Klassenaufbau eingebunden. Einführend legt das Quelltext-Listing 4.1 einen Auszug mit den entscheidenden Elementen der Klassenschnittstelle dar.

Einzigartigkeit

Die Einzigartigkeit des Strokes wird durch die Implementierung eines angepassten Meyers-Singleton-Patterns erreicht (vgl. hierzu Abschnitt 4.2.1). Konstruktor, Kopierkonstruktor und Zuweisungsoperator (Zeilen 21–23) sind dazu im privaten Bereich der Klasse deklariert. Das bedeutet, dass keine Klassen-fremde Funktion eine Instanz vom Typ `SbStroke` erzeugen oder kopieren kann. Nur `SbStroke` selbst kann diese Methoden aufrufen. Desweiteren verbietet die private Konstruktor-Deklaration ein Ableiten von `SbStroke`, was die Einzigartigkeit der Klasse zusätzlich unterstützt. In diesem Zusammenhang ist die Deklaration des Kopierkonstruktors und des Zuweisungsoperators eigentlich nicht notwendig, da das Entwurfsmuster nur eine Instanz erlaubt. Um jedoch einem möglichen Mißbrauch des Zuweisungsoperators vorzubeugen und weil der Kopierkonstruktor bei der Stilgebung verwendet werden muss, berücksichtigt die Klassenschnittstelle ihre Bereitstellung.

Eine Instanz der Klasse kann nun über die statische Funktion `getInstance(P)` aus Zeile 6 erhalten werden. Diese Funktion legt dem Entwurfsmuster entsprechend eine statische lokale Variable vom Typ `SbStroke` an. Anstelle einer Referenz wird jedoch ein Zeiger auf die Variable zurückgegeben, da dieser während des Rendering-Vorgangs in dem `SoStrokeElement` gehalten und somit jedem Pipeline-Knoten zur Verfügung stehen kann. Zu Beginn der Pipeline-Traversierung wird desweiteren eine Stroke-Instanz benötigt, deren Daten der Erstinstanziierung entsprechen bzw. leer sind. Dies kann über den Boolean-

```
1  class SbStroke
2  {
3        friend class SoLineStyleFilter;
4
5     public:
6        static SbStroke* getInstance(const bool& freshInstance= false);
7
8        void createNewStroke();
9        void makeNewToCurrentStroke();
10       int addCoord(const SbVec3f& value);
11       int addIndex(const int& value, const int& oldIndex,
12                                      const int& priority= -2);
13
14       void updateGStrokes();
15       void attachGStroke(SbGStroke& gstroke);
16       template<class T>
17       void getGStroke(T** gstroke, SoState* state, const bool& fresh)
18       { /*...*/ }
19
20    private:
21       SbStroke();
22       SbStroke(const SbStroke& other);
23       SbStroke& operator=(const SbStroke& other);
24
25       SoMFVec3f               coords;
26       SoMFInt32               indices1, indices2, oldIndexIndices;
27       std::list<SbGStroke*>   gstrokes;
28       SbGSPriority            priorityGStroke;
29 };
```

Listing 4.1: Ein Auszug aus der Klasse SbStroke.

Parameter `freshInstance` der Funktion erreicht werden. Ist dieser Parameter *wahr*, so werden alle Stroke-Daten gelöscht. Die Instanz entspricht dann ihrem initialen Zustand.

Durch die an die Anforderungen der Klasse angepasste Realisierung des Meyers-Singleton-Patterns wird die Einzigartigkeit von `SbStroke` erreicht. Diese muss jedoch für die Stilgebung durch den in Abschnitt 2.2.5 entworfenen Filter-Knoten aufgehoben werden. Dieser Knoten basiert auf dem Einsatz von Subgraphen und wird demzufolge als eine von `SoSeparator` abgeleitete Klasse implementiert (vgl. Abschnitt 4.3.3). Folglich wird eine Kopie des Strokes notwendig, damit die ursprünglichen Daten nach einer Subgraph-Traversierung weiterhin vorliegen. Die Erzeugung einer Kopie widerspricht der Einzigartigkeit des Strokes, kann jedoch auf Grund des Szenengraph-Konzepts nicht umgangen werden. Um dieses Realisierungsproblem möglichst gering zu halten, wird es *nur* dem als `SoLineStyleFilter` implementierten Filter-Knoten erlaubt, eine Kopie des Strokes anzulegen (Abschnitt 4.3.3). Um den nötigen Zugriff auf die private Kopierkonstruktor-Funktion zu erhalten, wird `SoLineStyleFilter` als `friend`-Klasse von `SbStroke` deklariert (siehe Zeile 3 des Lis-

tings 4.1). Eine als `friend` deklarierte Klasse hat Zugriff auf alle, auch die privaten, Deklarationsbereiche einer Klasse und sollte nur dann benutzt werden, um eng gekoppelte Konzepte miteinander zu verbinden (STROUSTRUP, 2000, Abschnitt 11.5). Da der Einsatz des Filter-Knotens eng mit dem G-Stroke-Konzept gekoppelt ist, entspricht der Einsatz der `friend`-Deklaration einer angemessenen Lösung.

Datenlisten

Listing 4.1 zeigt, dass die Klasse `SbStroke` sowohl über die Vertex-Liste `coords`, drei Index-Listen `indices1`, `indices2` und `oldIndexIndices` als auch über die G-Stroke-Liste `gstrokes` verfügt. Erstere ist vom Typ `SoMFVec3f`, einem OPEN-INVENTOR-Feld zum Speichern von 3D-Vektoren des Typs `SbVec3f`. Sie wird mit der Funktion `addCoord(P)` (Zeile 10) gefüllt. Diese Funktion sorgt dafür, dass jede neue Koordinate an das Ende der Liste gefügt und die Speicherposition (der Index) des Eintrags zurückgegeben wird. Die drei Index-Listen sind vom Typ `SoMFInt32`, einem OPEN-INVENTOR-Feld zum Speichern von Integer-Werten. Hierbei sind nur `indices1` und `indices2` für den Aufbau und die Veränderung der Stroke-Geometrie zuständig. Sie werden über die Funktion `addIndex(P)` (Zeile 11 und 12) gefüllt und geben ebenfalls die Speicherposition zurück. Diese wird für den parallelen Aufbau der G-Stroke-Daten benötigt. Hierbei sind zwei Index-Listen notwendig, damit eine jeweils als Referenz- und die anderen als aktuelle Speicherliste benutzt werden kann. Die Datenliste `oldIndexIndices` ist für die Anpassungsroutine der G-Strokes gedacht und gibt über die zuletzt gespeicherten Index-Positionen Auskunft. In Abschnitt 4.3.2 wird der Nutzen dieser drei Listen eingehend erläutert.

Die Liste `gstrokes` ist vom Typ `std::list`, einem Templatecontainer der C++-eigenen Standardbibliothek, der auf Grund seiner optimierten Elementeinfüge- und -löschoperationen eingesetzt wird (STROUSTRUP, 2000, Teil 3). Hierin werden die G-Strokes polymorph als Zeiger vom Typ `SbGStroke*` und dem Observer-Pattern entsprechend mittels der Funktion `attachGStroke(P)` (Zeile 15) gespeichert. Die Funktion `updateGStrokes(P)` aus Zeile 14 implementiert desweiteren die vom Observer-Pattern geforderte `update(P)`-Funktion der Observer, die durch die G-Strokes repräsentiert werden. Der Zugriff auf einzelne G-Strokes erfolgt über die Template-Funktion `getGStroke(P)`. Bevor jedoch auf die genaue Umsetzung der G-Stroke-Verwaltung und des Observer-Patterns in Abschnitt 4.3.2 eingegangen wird, geht es an dieser Stelle zunächst um die neue Verarbeitung der Stroke-Geometrie.

Aufbau und Veränderung des Kantenzugs

Wie in Abschnitt 4.1.3 beschrieben, nutzen die Pipeline-Knoten von OPENNPAR die Elemente `SoCoordinateElement` und `SoLineCoordinateIndexElement` zum Austausch der Stroke-Daten und legen ggf. eine lokale Kopie an. Sie arbeiten somit immer mit einer

Referenz-Liste, die zur Berechnung neuer Werte, und einer aktuellen Liste, die zur Speicherung dieser benutzt wird. Die in Abschnitt 3.3 geforderten Operationen zur Geometrie-Veränderungen (Hinzufügen, Löschen, Spalten) setzt die OPENNPAR-Bibliothek somit implizit um. Da desweiteren alle Methoden der Pipeline-Knoten auf dem Referenz-Listen-Prinzip basieren, würde eine Anpassung dieser an explizite Hinzufüge-, Lösch- und Spalt-Operationen der neuen Stroke-Klasse zu einer völligen Umstrukturierung sämtlicher Algorithmen führen. Dies hätte eine Überschreitung des gegebenen Zeitrahmens zur Folge und würde eine vom Thema abweichende Implementierung notwendig machen. Deshalb wird das Referenz-Listen-Prinzip in Form von zwei Index-Listen übernommen. Damit ist eine leichte Anpassung aller Pipeline-Knoten möglich. Alle neu berechneten Eckpunkte werden an die bereits existierenden Einträge angehängt, was eine zweite Vertex-Liste überflüssig macht. Über den zurückgegebenen Index kann der Vertex dann direkt in die neue Index-Liste eingefügt werden.

Die Verwaltung des Strokes geschieht folgendermaßen: Nach der Kantenextraktion wird die `SbStroke`-Instanz initialisiert und die Vertex- sowie eine Index-Liste mit Daten gefüllt. Anschließend wird ein Zeiger auf `SbStroke` in das `SoStrokeElement` gesetzt. Über diesen Zeiger steht die Klasse jedem Pipeline-Knoten zur Verfügung. Ihre Daten können über `getValue()`-Methoden gelesen werden. Eine Veränderung des Kantenzuges wird mit der Funktion `createNewStroke()` (Zeile 8) ausgelöst. Der Funktionsaufruf bewirkt, dass die zweite Index-Liste zur aktuellen Speicher-Liste wird, wobei die erste Liste weiterhin als Referenz-Liste dient. Der Knoten kann nun mit seiner Veränderung beginnen. Hierbei treten im Zusammenhang mit der automatischen Anpassung der G-Strokes bzgl. des Löschens von Eckpunkten einige Einschränkungen auf, die jedoch erst im Zuge der Beschreibung der G-Strokes selbst verständlich werden. Daher wird an dieser Stelle auf eine genaue Erklärung des hierfür eingesetzten `priorityGStrokes` der Klasse `SbStroke` verzichtet und auf Abschnitt 4.3.2 verwiesen. Hat der Knoten das Ende seiner Anpassungsoperationen erreicht, macht der Aufruf von `makeNewToCurrentStroke()` (Zeile 9) die Speicher-Liste zur aktuellen Referenz-Liste. Abschließend muss der `SbStroke`-Zeiger wieder in das Element gesetzt werden (vgl. das exemplarische Quelltext-Listing 4.2).[8]

Da `SbStroke` über seine statische Funktion erreicht wird, könnten alle Pipeline-Knoten auch ohne das Element direkt auf die Instanz zugreifen. Um jedoch sicherzustellen, dass die Instanz zu Beginn eines jeden Rendering-Vorgangs „leer“ ist, kommt das `SoStrokeElement` zum Einsatz. Ein zweiter Grund hierfür ist, dass das Element sofort die `updateGStrokes(P)`-Funktion (siehe Listing 4.1, Zeile 14) anstoßen kann, welche die `gstrokes`-Liste traversiert und die jeweilige Anpassungsroutine aufruft.

8 Die dargestellte OPEN-INVENTOR-Funktion `doAction(P)` wird i. A. bei der Traversierung des Szenegraphen in jedem Knoten ausgelöst.

```
1  void PipelineNode::doAction(SoAction* action)
2  {
3     // Die *action* liefert den aktuellen Traversierungszustand *state*.
4     SoState* state= action->getState();
5     // Der *state* liefert das Element, welches den Stroke holt.
6     SbStroke* theStroke= (SoStrokeElement::getInstance(state))->getStroke();
7
8     theStroke->createNewStroke();
9
10    /* Stilisierungsoperationen. */
11
12    theStroke->makeNewToCurrentStroke();
13    SoStrokeElement::set(state, this, theStroke);
14 }
```

Listing 4.2: Dies ist ein Beispiel für die Veränderung der Stroke-Geometrie in einem beliebigen Pipeline-Knoten.

4.3.2 Die Klasse SbGStroke und Derivate

Die in Abschnitt 3.2.2 entworfenen G-Strokes verfügen über bestimmte Eigenschaften: Sie sollen

- einzigartig sein,
- Eigenschaften des Strokes bzgl. der Index-Liste in Datenstrukturen speichern, wobei sich die Werte entweder auf die Eckpunkte oder die Segmente beziehen müssen,
- sich an eine Geometrie-Veränderung des Strokes während der Stilisierung anpassen,
- den Stroke bei der Stilgebung verändern können sowie
- hierarchisch umgesetzt werden.

Zur Realisierung dieser Anforderungen wird zunächst eine Klassenhierarchie entwickelt, die der konzeptuellen Hierarchie aus Abbildung 3.5 auf Seite 48 entspricht. Abbildung 4.4 zeigt einen Überblick über die entwickelten abstrakten und konkreten G-Stroke-Klassen.

Die Basisklasse

Die Klasse `SbGStroke` ist als Basisklasse abstrakt, kann also nicht instanziiert werden, und bietet die grundlegende Schnittstelle der G-Strokes. Das Quelltext-Listing 4.3 zeigt einen Auszug aus der Klasse `SbGStroke`.

Entscheidend bei dieser Klasse sind drei Punkte: Erstens handelt es sich bei den G-Strokes um Observer, entsprechend dem gleichnamigen Entwurfsmuster. Diese Tatsache wird durch die nicht-virtuelle Funktion `update(P)` festgelegt, die allen G-Strokes gehören muss und

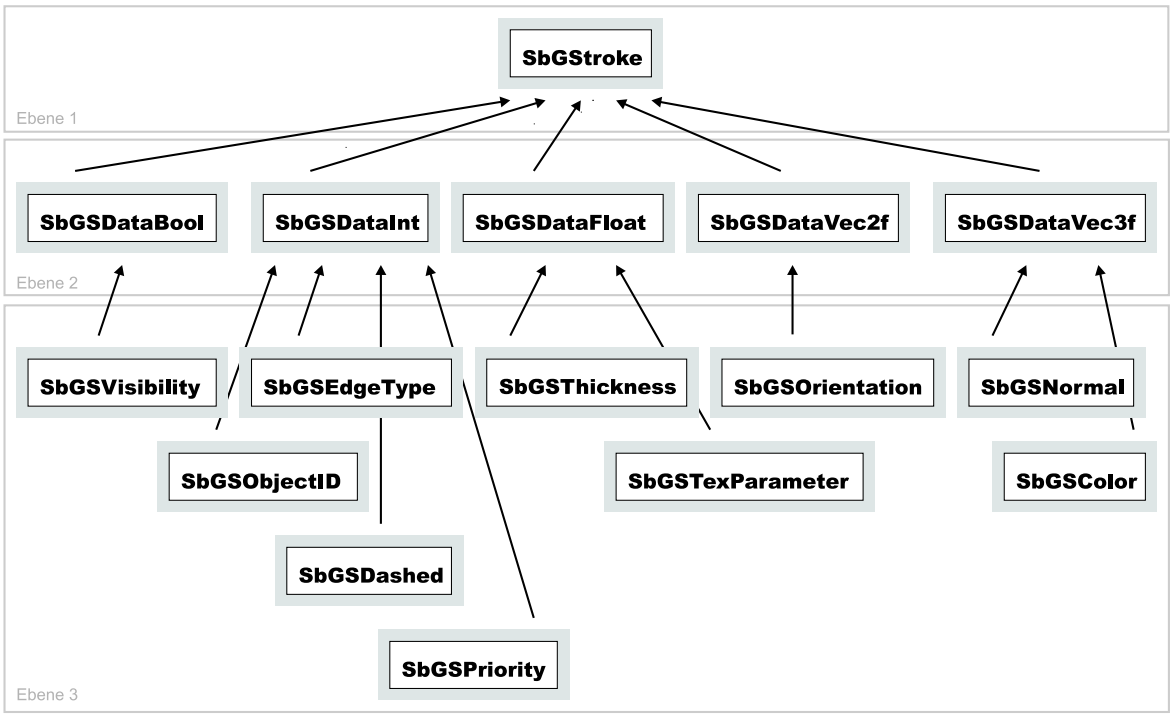


Abbildung 4.4: Die Abbildung veranschaulicht die implementierten G-Strokes und ihre Position in der umgesetzten Hierarchie.

```
1  class SbGStroke : private SoLineModifier
2  {
3     friend class SbStroke;
4
5     public:
6        void update(SbStroke* stroke) { updateGStroke(stroke); }
7        void setUpdate(bool update) { updateMe= update; }
8
9     protected:
10       bool        updateMe;
11
12    private:
13       virtual void updateGStroke(SbStroke* stroke)= 0;
14       virtual SbGStroke* getCopy() const= 0;
15 };
```

Listing 4.3: Ein Auszug aus der Klasse SbGStroke.

in der die Observer das Subject in Form des Parameters `SbStroke* stroke` erhalten. Die Nicht-Virtualität legt somit programmiersprachlich fest, dass alle abgeleiteten Klassen über diese Funktionalität verfügen und die rein-virtuelle Funktion `updateGStroke(P)` (Zeile 13) aufrufen müssen. Letztere muss wiederum von jedem G-Stroke selbst implementiert werden. Die Einteilung in eine nicht-virtuelle öffentliche und eine private rein-virtuelle

Funktion soll die Stabilität der Schnittstelle unterstützen (vgl. STROUSTRUP, 2000, Abschnitt 24.4.2). Die jeweilige *private* Implementierung der Anpassungsroutine ist somit vor dem Anwender versteckt. In diesem Zusammenhang ist auch die Variable `updateMe` vom Typ Boolean zu nennen, die angibt, ob der G-Stroke angepasst werden soll oder nicht. Dies ist z. B. dann notwendig, wenn ein Pipeline-Knoten den G-Stroke gemeinsam mit dem Stroke verändert und die Anpassung daher nicht automatisch durchgeführt werden braucht.

Der zweite entscheidende Punkt ist die private Ableitung von dem OPENNPAR-Knoten `SoLineModifier`. Wie bereits in Abschnitt 4.1.3 beschrieben, bietet dieser Knoten eine Reihe von Pixelmanipulations- und Geometrieoperationen, die auch im Zusammenhang mit der Anpassung der G-Stroke-Werte gebraucht werden. Da jedoch zwischen einem G-Stroke und dem Pipeline-Knoten keine konzeputellen Gemeinsamkeiten bestehen, soll hier nur die Implementierung des Knotens und somit seiner Funktionalität geerbt werden. Dies ist durch die private Vererbung möglich, wie in Abschnitt 4.1.1 beschrieben.

Der dritte entscheidende Punkt ist die „Freundschaft“ des G-Strokes zur Klasse `SbStroke` (Zeile 3) und die damit verbundene Funktion `getCopy()` (Zeile 14). In Abschnitt 4.3.1 wurde erläutert, dass bei der Stilgebung durch den Filter-Knoten eine Kopie des Strokes angelegt werden muss. Dies erfordert gleichzeitig eine Kopie der Elemente aus der G-Stroke-Liste, die mittels der genannten Funktion angefordert werden kann. Da die abgeleiteten konkreten G-Stroke-Klassen jedoch wie auch der Stroke als Singletons implementiert sind, darf die Kopie nur von einer bestimmten Klasse, `SbStroke`, angefordert werden. Ähnlich der Beziehung zwischen Stroke und Filter-Knoten besteht auch hier ein enger Zusammenhang zwischen Stroke und G-Stroke, der den Einsatz der `friend`-Deklaration rechtfertigt.

Die Basisklasse bietet somit die grundlegende Funktionalität der G-Strokes an. Sie kann jedoch keine Schnittstelle für allgemeine Funktionen bzgl. des Aufbaus der G-Strokes anbieten, da diese vom jeweiligen Datentyp abhängen. So kann z. B. keine eindeutige `addValue(P)`-Funktion deklariert werden, da der zu speichernde Wert abhängig vom Datentyp ist. Diese Funktionalitäten müssen daher auf der zweiten Hierarchie-Ebene angesiedelt werden.

Die Datenklassen

Die G-Strokes beschreiben unterschiedliche Eigenschaften, die nur unter Einsatz verschiedener Datenstrukturen umgesetzt werden können (vgl. Abschnitt 3.2.3). Die Aufbau- oder Anpassungsfunktionen hängen dabei vom jeweiligen Datentyp ab. Da derartige Funktionen auf Grund der Abhängigkeit vom jeweiligen Datentyp nicht in der Basisklasse `SbGStroke` gekapselt werden können, müssen sie in allen Datenklassen gleich deklariert werden.

Die Datenklassen der zweiten Hierarchie-Ebene sind wie die Basisklasse abstrakt, da auch sie keine reale Eigenschaft im Sinne eines G-Strokes aus Abschnitt 3.2 repräsentieren. Vielmehr kategorisieren sie die G-Strokes in Datenstrukturen vom Typ Boolean (`SbGSData-`

Bool), Integer (SbGSDataInt), Floating Point (SbGSDataFloat), 2D-Vector (SbGSDataVec2f) und 3D-Vector (SbGSDataVec3f). Da die angebotenen Funktionen für alle fünf Datenklassen gleich sind, ist der Quelltext-Auszug aus Listing 4.4 repräsentativ zu verstehen.

```
1  class SbGSDataInt : public SbGStroke
2  {
3     public:
4        int getValue(const int& pos) const;
5        bool addValue(const int& pos, const int& value);
6        void filter(SbStroke* stroke, const int& operation,
7                    int operValue) const
8        {
9           filterStroke(stroke, operation, operValue);
10       }
11
12    protected:
13       SoMFInt32   theData;
14
15    private:
16       virtual void updateGStroke(SbStroke* subject)
17       {
18          /* ... */
19          updateRealGStroke( ... );
20          /* ... */
21       }
22
23       virtual void updateRealGStroke(SoStroke* subject,
24                                      const int& i,
25                                      const int* indices,
26                                      const int* oldIndexIndices,
27                                      SoMFInt32& newIntStroke)= 0;
28       virtual void filterStroke(SbStroke* stroke, const int& operation,
29                                 int operValue) const= 0;
30 };
```

Listing 4.4: Ein Auszug aus der repräsentativen Klasse SbGSDataInt.

Die dargestellte Klasse repräsentiert den Integer als Datentyp-Kategorie. Dies spiegelt sich z. B. in der Datenstruktur theData vom Typ SoMFInt32 wider. Sie ist im protected-Bereich der Klasse deklariert und steht somit (nur) den abgeleiteten Klassen zur Verfügung. Mittels Positionsangabe kann über getValue(P) (Zeile 4) der gewünschte Wert aus theData angefordert werden. Interessanter erscheint in diesem Zusammenhang jedoch die Funktion addValue(P) aus Zeile 5 zu sein. Ihr Parameter pos gibt dabei die Index-Position an, an welcher der Wert value gespeichert werden soll. Die Position muss dabei mit der aktuellen Position aus Index-Liste des Strokes übereinstimmen, da sich der G-Stroke-Wert immer auf den jeweiligen Stroke-Eckpunkt bzw. das folgende Kantensegment beziehen und

somit parallel zum Stroke verlaufen muss (vgl. Abschnitt 3.2). Diese Forderung kann über den Rückgabewert der `addIndex(P)`-Funktion des Strokes erreicht werden. Der Rückgabewert der Funktion vom Typ Boolean gibt schließlich an, ob die Hinzufüge-Aktion erfolgreich war (`true`) oder nicht (`false`). In diesem Zusammenhang ist die Klasse `SbGSDataBool` zu nennen, bei der es sich ebenfalls um eine Integer-Klasse handelt. Diese Tatsache resultiert aus der Trennung der einzelnen G-Strokes durch -1. Da es sich hierbei um keinen Boolean-Wert handelt, repräsentiert die Klasse einen pseudo-Boolean Datentyp, der neben 0 und 1 auch die -1 speichert. Um neben diesen keine weiteren Integer-Werte zuzulassen, verfügt die Klasse desweiteren über eine Reihe an Wert-Überprüfungsfunktionen.

Desweiteren deklariert die Datenklasse eine weitere rein-virtuelle Anpassungsroutine: `updateRealGStroke(P)` aus Zeile 23. Der Grund hierfür ist, dass es eine Reihe an allgemeinen Aufgaben vor der eigentlichen Anpassung des G-Strokes an den Stroke gibt, die zusammengefasst werden können. Dazu gehört z. B. das Anfordern der benötigten Datenlisten des Strokes, also die Index-Listen `indices` und `oldIndexIndices`, oder das Erzeugen einer neuen Datenstruktur `newIntStroke`, die mit den angepassten Werten gefüllt wird und anschließend zur neuen `theData`-Struktur wird. Die Umsetzung dieser für alle abgeleiteten Klassen gleichen Aufgaben wird von der Funktion `updateGStroke(P)` übernommen, die somit auf der zweiten Hierarchie-Ebene implementiert wird. Die Funktion `updateRealGStroke(P)` dient der Implementierung der jeweiligen konkreten Eigenschaft auf der dritten Hierarchie-Ebene und wird im folgenden Abschnitt näher erläutert.

Abschließend soll hier die ebenfalls rein-virtuelle Funktion `filterStroke(P)` (Zeile 28) erwähnt werden. Sie stellt die Schnittstelle für eine mögliche Filterung des Strokes anhand der G-Stroke-Daten dar und realisiert somit die doppelte Abhängigkeit zwischen Stroke und G-Stroke. Die Funktion muss ebenfalls von jeder konkreten G-Stroke-Klasse implementiert werden und wird in Abschnitt 4.3.3 eingehend beleuchtet.

Die konkreten G-Strokes

Die bisher beschriebenen G-Stroke-Klassen sind abstrakt und können nicht instanziiert werden. Sie dienen der Funktionskapselung und dem Aufbau einer Klassenhierarchie, wie in Abschnitt 3.3. Die eigentlichen Stroke-Eigenschaften sind als konkrete Klassen implementiert und setzen die beschriebenen rein-virtuellen Funktionen um. Durch das Prinzip der Vererbung und der polymorphen Objekt-Nutzung sowie den damit verbundenen rein-virtuellen Funktionen ist die Struktur einer jeden G-Stroke-Klasse bereits vorgegeben. Hierzu gehören die Anpassungsfunktion `updateRealGStroke(P)`, die Stroke-Filterfunktion `filterStroke(P)` und die Kopierfunktion `getCopy()`. Die Einzigartigkeit der konkreten G-Strokes wird dabei wieder mit dem Meyers-Singleton-Pattern erzielt. Die geforderte Implementierung des Entwurfsmusters kann jedoch nicht automatisch durch Vererbung festgelegt werden. Das Singleton-Pattern beruht auf der statischen `getInstance(P)`-Funktion, die

jedoch nicht virtuell eingesetzt werden kann: Virtuelle Deklarationen nehmen im Gegensatz zu statischen Deklarationen immer auf ein bestimmtes Objekt der Klasse Bezug.

Da jeder G-Stroke eine bestimmte Eigenschaft repräsentiert, bieten die Klassen neben der reinen Datenstrukturverwaltung zusätzliche Merkmale und Funktionen an. So hält z. B. der Edgetype-Stroke eine Liste der verschiedenen Kantenidentifikationsnummern und der Parameter-Stroke eine Variable über die Texturlänge. Desweiteren muss für jede `addValue(P)`-Funktion überprüft werden, ob der hinzuzufügende Wert der Eigenschaft entspricht. So speichert der Edgetype-Stroke nur Werte, die in seine Kantenidentifikationsliste eingetragen sind. Tabelle 4.1 gibt einen Überblick über die umgesetzten G-Strokes und ihre jeweilige Funktionalität.

Am Beispiel des Edgetype-Strokes soll im Quelltext-Listing 4.5 der allgemeine Aufbau eines konkreten G-Strokes erläutert werden. Die dargestellte Klasse legt zunächst fest, von

```
1  class SbGSEdgeType : public SbGSDataInt
2  {
3     friend class SbStroke;
4     static SbGSEdgeType* getInstance();
5
6     public:
7        virtual ~SbGSEdgeType();
8
9     private:
10       SbGSEdgeType();
11       SbGSEdgeType(const SbGSEdgeID& );
12       SbGSEdgeType& operator=(const SbGSEdgeID& );
13
14       virtual void updateRealGStroke(SbStroke* subject,
15                                      const int& i,
16                                      const int* indices,
17                                      const int* oldIndexIndices,
18                                      SoMFInt32& newIntStroke);
19       virtual void filterStroke(SbStroke* stroke,
20                                 const int& operation,
21                                 int operValue) const;
22       virtual SbGStroke* getCopy() const;
23
24 };
```

Listing 4.5: Ein Auszug aus der konkreten G-Stroke-Klasse SbGSEdgeType. Er umfasst die Funktionen, die allen G-Stroke gemein sind.

welcher Datentyp-Kategorie der G-Stroke öffentlich abgeleitet wird. In Listing 4.5 handelt es sich um die Klasse `SbGSDataInt` (Zeile 1), also einem Integertyp. Die Einzigartigkeit der konkreten G-Stroke-Klassen wird dabei so umgesetzt, wie es in Abschnitt 4.3.1 für die Klasse `SbStroke` beschrieben wurde. Der einzige Unterschied dazu ist die private Deklaration der `getInstance(P)`-Funktion. Diese steht nur dem G-Stroke und dem befreundeten

G-Stroke	Beschreibung
SbGSVisibility	Bezieht sich auf das Kantensegment und gibt dessen Sichtbarkeit bzw. Nicht-Sichtbarkeit an.
SbGSEdgeType	Bezieht sich auf das Kantensegment und gibt den Kantentyp an (Silhouettenkante, scharfe Kante, ...).
SbGSObjectID	Bezieht sich auf das Kantensegment und gibt an, zu welchem Objekt das Segment gehört.
SbGSDashed	Bezieht sich auf das Kantensegment und gibt an, zu welchem *dash* („Strichelchen“) das Segment gehört. Dieser G-Stroke ist im Zusammanhang mit dem Unterteilen des Segments in viele Teilstücke sinnvoll und ermöglicht die Darstellung unterschiedlich gestrichelter Linien.
SbGSPriority	Bezieht sich auf den Vertex und gibt seine Wichtigkeit an. Diese bezieht sich auf die Vertex-Lösch-Operation. Es dürfen i. d. R. nur Eckpunkte zwischen Anfangs- und Endpunkt eines Kantenzuges gelöscht werden.
SbGSThickness	Bezieht sich auf den Vertex und gibt den Parameter für die Linienbreite am jeweiligen Vertex an.
SbGSTexParameter	Bezieht sich auf den Vertex und gibt den Parameterwert an, der in erster Linie für die Texturierung gebraucht wird.
SbGSOrientation	Bezieht sich auf den Vertex und gibt an, welche Orientierung des Linienzugs am Eckpunkt vorliegt. Diese wird anhand des Vorgängers und des Nachfolgers berechnet und ist für den Verlauf der Textur wichtig.
SbGSNormal	Bezieht sich auf den Vertex und gibt die jeweilige Normale im Eckpunkt an.
SbGSColor	Bezieht sich auf das Kantensegment und gibt den jeweiligen Farbwert an.

Tabelle 4.1: Umgesetzte G-Strokes und ihre Aufgabe im Überblick.

Stroke `SbStroke` (Zeile 3) zur Verfügung, was die enge Verbindung zwischen Stroke und G-Stroke unterstützt. Die jeweiligen Eigenschaften sind folglich nur über den Stroke zugänglich und können über die `SbStroke`-Funktion `getGStroke(P)` angefordert werden. Damit ist die Verwaltung und Nutzung der G-Strokes eindeutig von den Pipeline-Knoten getrennt.

In Abschnitt 3.3 wurden die Aufgaben der Pipeline-Knoten beschrieben. Hierzu gehörte das Erzeugen neuer G-Strokes und die Veränderung der Stroke-Geometrie. Hierbei ist darauf zu achten, dass die Anwendung möglichst einfach und intuitiv ist. Das Quelltext-Listing 4.6 er-

weitert an dieser Stelle das Listing 4.2 um die beispielhafte Erzeugung und Anwendung von G-Strokes. Das Beispiel zeigt zunächst, wie der G-Stroke vom Typ `SbGSEdgeType*` über

```
1  void PipelineNode::doAction(SoAction* action)
2  {
3     SoState* state= action->getState();
4     SbStroke* theStroke= (SoStrokeElement::getInstance(state))->getStroke();
5
6     SbGSEdgeType* edgeStroke= 0;
7     theStroke->getGStroke(edgeStroke, state, true);
8     edgeStroke->setUpdate(false);
9
10    theStroke->createNewStroke();
11
12    /* ... */
13
14    // Speichere neue Werte.
15    int lastCoordIdx= theStroke->addCoord(newCoord);
16    int lastIndexIdx= theStroke->addIndex(lastCoordIndex, -1, 0);
17    edgeStroke->addValue(lastIndexIdx, EDGE_ID);
18
19    /* ... */
20
21    theStroke->makeNewToCurrentStroke();
22    theStroke->attach(edgeStroke);
23    SoStrokeElement::set(state, this, theStroke);
24
25 }
```

Listing 4.6: Beispielhafte Erzeugung und Anwendung von G-Strokes.

die Funktion `getGStroke(P)` angefordert werden kann (Zeile 7). Hierzu muss lediglich ein mit Null initialisierter Zeiger vom Typ des jeweiligen Strokes angelegt werden. Der erste Parameter der Funktion ist der Template-Parameter, der es ermöglicht, dass *jeder* G-Stroke mit *einer* Funktion behandelt werden kann. Die Funktion sucht zunächst in der G-Stroke-Liste des Strokes, ob der jeweilige G-Stroke bereits gespeichert wurde. Auf Grund der Laufzeit-Typinformation von C++ kann dabei überprüft werden, auf welche abgeleitete Klasse das polymorphe Element der Liste zeigt. Führt die Suche zu keinem Ergebnis, wird eine neue Instanz des G-Strokes erzeugt. Handelt es sich um eine Kopie des Strokes, wird ein neuer G-Stroke per Konstruktor erzeugt, da der Aufruf der Instanz-Funktion dieselbe Instanz des Original-Strokes zur Verfügung stellen würde. Für den Fall, dass der dritte Parameter der `getGStroke(P)`-Funktion `wahr` ist, soll eine neue Instanz erzeugt werden. Das bedeutet, dass die möglicherweise gefüllten Datenstrukturen und gesetzten Variablen des G-Strokes in ihren initialen Zustand versetzt werden sollen. Für den Fall, dass der G-Stroke nur für Lesezwecke genutzt werden soll, muss der dritte Parameter dementsprechend `falsch` sein. Parameter zwei wird für die per privater Vererbung abgeleiteten und vom G-Stroke genutzten Funktionen des `SoLineModifier` benötigt (vgl. Abschnitt 4.3.2 und Listing 4.3).

Nachdem der neue G-Stroke zur Verfügung steht, kann er mit Werten gefüllt werden. Wird der Stroke, wie in diesem Beispiel, ebenfalls verändert, so muss die Anpassungsroutine des G-Strokes ausgeschaltet werden. Dies wird mit dem Funktionsaufruf `setUpdate(P)` und dem übergebenen Parameterwert `false` erreicht (Zeile 8). Da der G-Stroke hier automatisch an den Kantezug des Strokes angepasst wird, ist eine weitere automatische Anpassung überflüssig.

In den Zeilen 15–17 ist exemplarisch das Füllen der Datenstrukturen dargestellt. Zunächst wird ein neuer Eckpunkt in der Vertex-Liste gespeichert und die Index-Position zurückgegeben. Diese wird wiederum in die Index-Liste gespeichert. Der Parameter -1 der `addIndex(P)`-Funktion gibt an, dass es sich um einen neuen Wert an dieser Stelle handelt. Wird ein alter Wert übernommen, so muss seine Position in der Index-Liste übergeben werden, z. B. `addIndex(indexListe(x), x, 0)`. Der zweite Parameter wird für die automatische Anpassung der G-Strokes benötigt und in der Stroke-eigenen Datenstruktur `oldIndexIndices` gespeichert. Parameter drei gibt die Priorität des Eckpunktes an. Da sie in diesem Beispiel 0 ist, kann der Vertex problemlos gelöscht werden. Im Anschluss daran wird der G-Stroke mit Werten gefüllt. Hier wird zunächst die Speicherposition durch den zurückgegebenen Wert der `addIndex(P)` festgelegt und im Anschluss der gewünschte G-Wert gespeichert.

Nachdem die jeweilige Aufgabe des Pipeline-Knotens erledigt ist, kann der neue G-Stroke zu den Elementen der G-Stroke-Liste des Strokes hinzugefügt werden (Zeile 22). Abschließend wird das Element mit dem neuen Stroke aktualisiert. Da sich der Kantenzug des Strokes verändert hat, müssen die G-Strokes der Liste angepasst werden. Die Anpassung wird nach dem Setzen des Stroke-Zeigers vom Element ausgelöst und im folgenden beschrieben.

Automatische Anpassung

Die automatische Anpassung der G-Strokes wird ausgelöst, sobald sich der Kantenzug des Strokes verändert hat und entspricht damit dem Observer-Pattern aus Abschnitt 4.2.2. Eine Veränderung wird von den Pipeline-Knoten der Stilisierungs-Pipeline durchgeführt, nachdem die zu stilisierenden Kanten erkannt worden sind. Die Linienzüge stehen demnach fest. Das bedeutet, dass die Veränderung letztendlich nur die Kantensegmente innerhalb eines Kantenzuges, also zwischen Anfangs- und Endvertex, betreffen kann. Einzig das Kürzen eines Strokes kann dazu führen, dass sich seine Anfangs- oder Endkoordinaten verändern. Derartige Operationen werden jedoch nur von dem `SoLinePurifier`-Knoten bei zu kurzen Kantensegmenten durchgeführt. Zu den Operationen der einzelnen Pipeline-Knoten gehört in erster Linie das Hinzufügen neuer Eckpunkte.

Zum Zeitpunkt der Auslösung der Anpassungsroutine beziehen sich die Datenstrukturen der G-Strokes auf den Kantenzug, der durch die Referenz-Liste des Strokes, also der zweiten Index-Liste, indiziert wurde. Da bei jeder Veränderungsoperation eine völlig neue Index-Liste aufgebaut wird, stellt sich die Frage, woher die G-Strokes wissen sollen, was sich

an dem alten Kantenzug geändert hat. Wie können sich die G-Strokes einer Veränderung des Kantenzuges bzgl. den genannten Operationen nun automatisch und effizient anpassen? Anhand des folgenden Beispiels soll das entwickelte Anpassungs-Schema erläutert werden.

Der beispielhafte Kantenzug `A` besteht aus den Indizes 1, 2, 3, 4, 5, -1. Ein exemplarischer Edgetype-Stroke besteht aus den parallel indizierten Werten 1, 1, 2, 2, 2, -1. Die Werte der zweiten Index-Liste und der `oldIndexIndices` werden erst im nächsten Schritt mit in die Erläuterung einbezogen. Die Datenstrukturen entsprechen damit den Daten aus Tabelle 4.2. In einem nächsten Pipeline-Knoten wird ein neuer Eckpunkt berechnet und an

Position A	0	1	2	3	4	5
Speicher-Liste A	1	2	3	4	5	-1
Kantentyp	1	1	2	2	2	-1

Tabelle 4.2: Speicher-Liste und parallel indizierter Edgetype-Stroke. Die Positions-Liste gibt die Indizes der beiden Datenlisten an und bestätigt ihre Parallelität.

die nächste Stelle der Vertex-Liste gehängt, in diesem Fall an die Index-Position 6. Die Speicher-Liste wird dementsprechend zur neuen Referenz-Liste und umgekehrt. Um dem G-Stroke mitteilen zu können, an welcher Stelle sich eine Veränderung des Kantezuges ergeben hat und wo nicht, wird die Datenstruktur `oldIndexIndices` mit den Positionen der Referenz-Liste oder einer -1 für den neu berechneten Wert gefüllt (vgl. Tabelle 4.3). Mittels

Position A	0	1	2	3	4	5	
Referenz-Liste A	1	2	3	4	5	-1	
Kantentyp	1	1	2	2	2	-1	
Position B	0	1	2	3	4	5	6
Speicher-Liste B	1	2	6	3	4	5	-1
oldIndexIndices	0	1	-1	2	3	4	5

Tabelle 4.3: Speicher- und Referenz-Liste stimmen nicht überein. Die Liste `oldIndexIndices` wird an den veränderten Positionen mit einer -1 gefüllt und ansonsten mit den alten Positionen der Referenz-Liste, die einfach in die Speicher-Liste übernommen wurden.

der `oldIndexIndices`-Liste kann sich der G-Stroke nun anpassen. Da diese Liste die Positionen und keine Werte speichert, muss die G-Stroke-Anpassungsroutine lediglich überprüfen, wann eine -1 auftritt. Tritt keine -1 auf, so können die alten Werte über die Postion ermittelt und übernommen werden. Tritt eine -1 auf, so muss zwischen Vorgänger und Nachfolger interpoliert werden. Da sich die Veränderungen der Pipeline-Knoten immer nur auf die einzelnen Segmente beziehen, kann es nicht vorkommen, dass zwei neue Koordinaten nacheinander eingefügt werden, dementsprechend tritt die -1 nicht zweimal nacheinander auf und die Interpolation kann durchgeführt werden.

Da sich der beispielhafte Edgetype-Stroke auf die einzelnen Segmente bezieht und durch Hinzufügen eines neuen Eckpunktes ein Segment in zwei Teilstücke getrennt wird, kann nun einfach der Wert des Vorgängers für die neue Koordinate übernommen werden. Dieser Wert hatte sich auf das gesamte Segment bezogen und ist deshalb auch als neuer G-Wert gültig. Nach der Anpassungsroutine entsprechen die Datenstrukturen den Listen aus Tabelle 4.4.

`Position A`	0	1	2	3	4	5	
`Referenz-Liste A`	1	2	3	4	5	-1	
`Kantentyp`	1	1	2	2	2	-1	
`Position B`	0	1	2	3	4	5	6
`Speicher-Liste B`	1	2	6	3	4	5	-1
`Kantentyp`	1	1	1	2	2	2	-1
`oldIndexIndices`	0	1	-1	2	3	4	5

Tabelle 4.4: Alte und aktuelle Listen im Überblick. Der G-Stroke konnte mittels der alten Positionen erfolgreich an den neuen Kantenzug angepasst werden.

Beziehen sich die G-Strokes nicht auf die Kantensegmente, sondern auf die Eckpunkte, müssen ggf. Interpolationsberechnungen durchgeführt werden. Durch die private Ableitung vom `SoLineModifier`-Knoten stehen ihnen die notwendigen Operationen jedoch ohne jeglichen Aufwand zur Verfügung. Durch die Einführung der `oldIndexIndices`-Datenstruktur ist somit eine sehr einfache und effiziente Anpassung der G-Strokes möglich. Ohne diese Datenstruktur hätte nur unter großem Aufwand eine Relation zwischen neuem und altem Kantenzug hergestellt werden können. Die beschriebenen Anpassungen beziehen sich jedoch ausschließlich auf das Hinzufügen neuer Werte. Das Löschen eines Eckpunktes hätte zur Folge, dass die `oldIndexIndices`-Liste die Veränderungsposition nicht mehr eindeutig wiedergeben könnte. Vielmehr müsste nun jeder G-Stroke überprüfen, an welcher Stelle die Differenz zwischen den Positionen (Aufgrund des Entfernens eines Eckpunkts) größer als Eins ist. Eine mögliche `oldIndexIndices`-Liste 1, 2, 3, 4, 5 würde durch das Löschen eines Eckpunktes z. B. zu der Liste 1, 2, 4, 5 werden. Nun ist es möglich, dass sich hierbei gewisse Fehler in den Aufbau der G-Stroke-Daten einschleichen. Zum Beispiel würde folgender Visibility-Stroke 0, 0, 1, 1, 1 durch das Entfernen des Indizes 3 folgendermaßen aussehen: 0, 0, 1, 1. Das hätte zur Folge, dass das ehemals sichtbare Kantensegment von Position 3 plötzlich unsichtbar werden würde, da sich der zweite Wert des neuen Visibility-Strokes nun auch auf diesen Kantenteil beziehen würde. Um solche Fehler zu verhindern, konnte das entwickelte G-Stroke-Konzept in Form des Priority-Stroke `SbGSPriority` direkt eingesetzt werden. Dieser G-Stroke wird von der Stroke-Klasse `SbStroke` verwaltet und hält für jeden Vertex einen Prioritätswert. Dieser Wert entscheidet darüber, ob der Vertex entfernt werden darf oder nicht. So ist es z. B. verboten, den Anfangs- und den Endwert eines Kantenzuges zu löschen, damit die Interpolationen korrekt durchgeführt werden können. Das entwickelte Konzept konnte somit direkt weiterentwickelt und eingesetzt werden.

4.3.3 Stilgebung

Nachdem die automatische Anpassung der G-Strokes nun selbstständig und unabhängig von den Pipeline-Knoten geschieht, können die erzeugten G-Strokes für die Stilgebung verwendet werden. Hierfür wurden in Abschnitt 3.4 drei Möglichkeiten entworfen, von denen die erste umgesetzt und im folgenden beschrieben wird.

Der entworfene Filter-Knoten ist in der Klasse `SoLineStyleFilter` implementiert. Diese Klasse ist von einem `SoSeparator` abgeleitet und mit der Stroke-Klasse `SbStroke` befreundet, da sie eine Kopie der Stroke-Daten anlegen muss. Der `SoLineStyleFilter` ermöglicht es nun, einen oder mehrere zu filternde G-Strokes und ein Filter-Kriterium auszuwählen. Letzteres besteht dabei aus einer Operation `O` und einem Vergleichswert `V`. Die Operation ist eine Vergleichsoperation, die den angegebenen Vergleichswert mit den Werten der G-Strokes vergleicht. Wird z. B. der Visibility-Stroke als G-Stroke mit dem Vergleichswert 0 gewählt, so werden alle verdeckten Kanten für die Ausgabe gefiltert. Desweiteren kann noch ein Kantentyp angegeben werden, so dass dann alle verdeckten Kanten des gewählten Kantentyps gefiltert und ausgegeben werden.

Die Filter-Knoten können nacheinander in der Pipeline platziert werden, so dass verschiedene Kanten gefiltert und mit einem Stil versehen ausgegeben werden können. Abbildung 4.5 stellt einen möglichen Szenengraph mit dem Filter-Knoten dar. Hierzu ist das Anlegen einer Stroke-Kopie notwendig. Da der `SoLineStyleFilter` mit dem Stroke befreundet ist, darf er eine Kopie des eigentlichen Strokes mit allen G-Strokes anlegen. Die Kopie wird dazu verwendet, die gewählten G-Strokes zu erhalten. Die Filterung geschieht mittels der G-Stroke-Funktion `filter(P)`, welcher neben dem Kantenzug auch die Vergleichsoperation und der Vergleichswert gegeben werden. Nachdem die jeweiligen Kanten gefiltert worden sind, wird die Stroke-Kopie in das `SoStrokeElement` gesetzt und steht nun weiteren Stilisierungen im Subgraphen zur Verfügung. Nach der Traversierung und Ausgabe der gefilterten Segmente löscht der `SoLineStyleFilter` die Kopie und der alte Stroke wird wieder zum aktuellen Zeiger des Elements.

Der Filter-Knoten dient somit nicht nur der Filterung selbst, sondern bietet vielmehr auch die interaktiven Möglichkeiten, um die G-Strokes zu nutzen. Dem Entwurf entsprechend verfügt der Filter-Knoten über keine Möglichkeiten zur Veränderung oder Erzeugung neuer G-Strokes. Die Filterung geschieht in der G-Stroke-Klasse, um die doppelte Abhängigkeit zwischen Stroke und G-Stroke im Quelltext umzusetzen (vgl. Abschnitt 3.4).

4.4 Zusammenfassung

OPENNPAR ist als Erweiterung von OPEN INVENTOR konzipiert und wird in C++ entwickelt. Dies ermöglicht eine elegante Kombination von konzeptuellem Klassendesign und

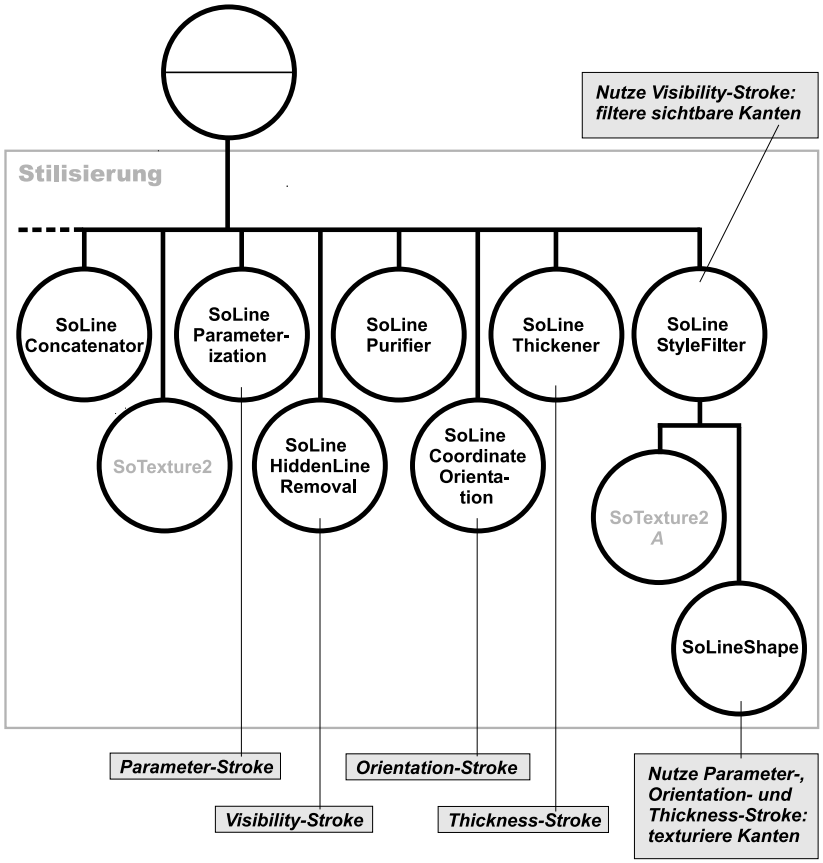


Abbildung 4.5: Die Abbildung zeigt den möglichen Aufbau des Stilisierungsteils eines Szenengraphs mit dem neuen Konzept. Dabei werden zunächst die G-Strokes Parameter-, Visibility-, Orientation- und Thickness-Strokes erzeugt und dann für die Stilgebung verwendet.

dem Szenengraph-Ansatz. Das Konzept der G-Strokes konnte unter Ausnutzung der damit angebotenen Werkzeuge und bestimmter Entwurfsmuster sehr gut umgesetzt werden. Genaue Kenntnisse der Programmiersprache förderten die enge Verbindung zwischen Entwurf und Implementierung. So konnte die getrennte Verwaltung von Stroke und G-Strokes erfolgreich realisiert werden. Das beinhaltet zum einen, dass die Pipeline-Knoten nur noch ein Datenpaket über ein dafür entwickeltes Element untereinander austauschen müssen. Alle weiteren Elemente der Stilisierungs-Pipeline von OPENNPAR konnten mit den G-Strokes umgesetzt werden. Die Knoten dienen dementsprechend in erster Linie der Extraktion von Eigenschaften und der Erzeugung von G-Strokes, die dann für die Stilgebung genutzt werden können. Dies führt zu erheblich weniger Quelltext in den einzelnen Klassen und vereinfacht ihre Anwendung erheblich. Ohne großen Aufwand können mittlerweile neue Pipeline-Knoten implementiert werden. So entstanden im Zuge der Realisierung des Konzepts der `SoLineColor`-Knoten zur Bestimmung der Linienfarbe und der `SoLineMerger`-Knoten, der als ein Stil-Knoten gedacht war, jedoch aus Zeitgründen nicht mehr fertiggestellt werden konnte. Der `SoLineColor`-Knoten kann gemeinsam mit dem `SoDisplayStrokes`-Knoten zur Farbgebung der OPENGL-Linien verwendet werden.

Durch die Umsetzung der `SbStroke`-Klasse und seiner Datenlisten ist es desweiteren nicht mehr nötig, in den einzelnen Pipeline-Knoten lokale Kopien der Daten anzulegen. Einzig der

`SoLineStyleFilter` wird hierfür verwendet. Allerdings ist dieser Knoten extra zu diesem Zweck konzipiert worden und insofern nicht mit einem *normalen* Stilisierungsknoten zu vergleichen. Hinzu kommt, dass der Stroke nur noch eine Vertex-Liste verwaltet und diese nicht nach jeder Änderung neu anlegen muss.

Durch den Einsatz des Priority-Strokes konnte das Konzept erfolgreich „an sich selbst“ getestet werden. Trotz der verschiedenen Datenstrukturen wurde erreicht, dass viele Gemeinsamkeiten auf einer höheren Hierarchie-Ebene gekapselt werden konnten. Durch den Einsatz des Polymorphismus und der virtuellen Funktionen wurde die Struktur der G-Stroke-Klassen streng vorgegeben. Dies hat zur Folge, dass sich auch neue G-Stroke-Klassen an das Konzept anpassen müssen und einfach anpassen lassen.

Die im folgenden Kapitel vorgestellten Fallbeispiele geben über den Erfolg der Implementierung Auskunft. Innerhalb kürzester Zeit konnten eine Reihe von Liniengrafiken erzeugt werden, die bisher nicht ansatzweise oder nur unter großem Aufwand mit OPENNPAR hätten hergestellt werden können.

Kapitel 5

Fallbeispiele

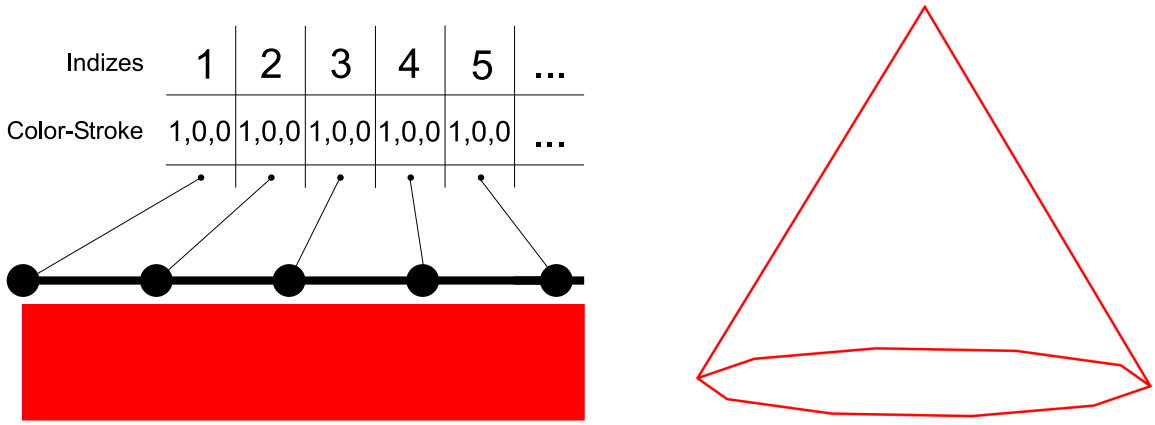
In den vergangenen Kapiteln wurde das Konzept der G-Strokes entworfen und umgesetzt. An dieser Stelle werden nun die Ergebnisse der Implementierung präsentiert. Die vorgestellten Bilder wurden dabei mit dem im Rahmen dieser Diplomarbeit weiterentwickelten Programm DEMOWE erzeugt und demonstrieren die Realisierung des Konzepts. Zunächst werden hierfür die entwickelten G-Strokes einzeln vorgestellt und im Anschluss daran in Kombination mit anderen präsentiert. Desweiteren bietet das Kapitel einen Überblick über Stil-Kombinationen und mögliche Anwendungsgebiete der G-Strokes.

Um die einzelnen Eigenschaften visuell darstellen und hervorheben zu können, ist es oftmals erforderlich, mehrere G-Strokes miteinander zu kombinieren. So bietet es sich bei der Erläuterung eines bestimmten Strokes in den meisten Fällen an, auch den Visibility-Stroke auszunutzen, um nur die sichtbaren Kanten zu zeigen. Dies ermöglicht eine klare Aussage über den G-Stroke im Fokus. In den folgenden Beispielen wird jeweils auf die verwendeten G-Strokes hingewiesen.

5.1 Color-Stroke

Der Color-Stroke repräsentiert die Farb-Eigenschaft der Kanten. Er wird immer dann verwendet, wenn keine Textur benutzt werden soll und gehört zu den am einfachsten umsetzbaren G-Strokes. Dabei wird pro Vertex ein RGB-Wert gespeichert, der sich auf das folgende Kantensegment bezieht. In Abbildung 5.1(a) ist der Zusammenhang zwischen den Datenstrukturen dargestellt, Abbildung 5.1(b) zeigt ein einfaches Beispiel für die Anwendung des G-Strokes. Da sich die Farb-Eigenschaft auf die bereits existierende Geometrie des Kantenzuges bezieht, ist es bei der Erzeugung des Color-Strokes nicht notwendig, neue Eckpunkte zu berechnen.

Farbige Kantenzüge sind bei OPENNPAR bereits zur Anwendung gekommen. Allerdings konnten die Farben nur durch den Ausgabe-Knoten `SoDisplayStrokes` direkt festgelegt werden. Der neu entwickelte `SoLineColor`-Knoten erzeugt den G-Stroke jetzt bereits im Vorfeld und kann variabel eingesetzt, die Farbe also an einer beliebigen Stelle im Szenengraph festgelegt werden. Das Speichern der Farb-Eigenschaft ist außerdem für zukünftige Implementierungen eines Stil- oder FilterStil-Knotens von Interesse. So könnten diese Kno-



(a) Datenlisten und korrespondierende Farbe.

(b) Farbliche Darstellung der Objektkanten.

Abbildung 5.1: Die Abbildungen veranschaulichen den Aufbau und die Anwendung des Color-Strokes. Abbildung (a) stellt dabei den Zusammenhang zwischen Datenstrukturen und Farb-Eigenschaft dar. Abbildung (b) zeigt als Beispiel eine farbige Liniengrafik. Bei dem Bild kam lediglich der Color-Stroke zum Einsatz.

ten z. B. anhand einer anderen G-Information die Farbe für bestimmte Kantensegmente festlegen und verändern. Mit dem Filter-Knoten ist dies noch nicht möglich. Hierbei können die G-Informationen lediglich ausgewertet, jedoch nicht verändert werden.

Für die farbige Darstellung des Objekts aus Abbildung 5.1(b) besteht der Stilisierungsteil des Szenengraphs lediglich aus drei Knoten (vgl. Abbildung 5.2). Hierzu gehört zunächst die Verknüpfung der Kanten zu Strokes durch den `SoLineConcatenator`. Danach wird der Color-Stroke erzeugt und das Bild schließlich vom `SoDisplayStrokes`-Knoten ausgegeben.

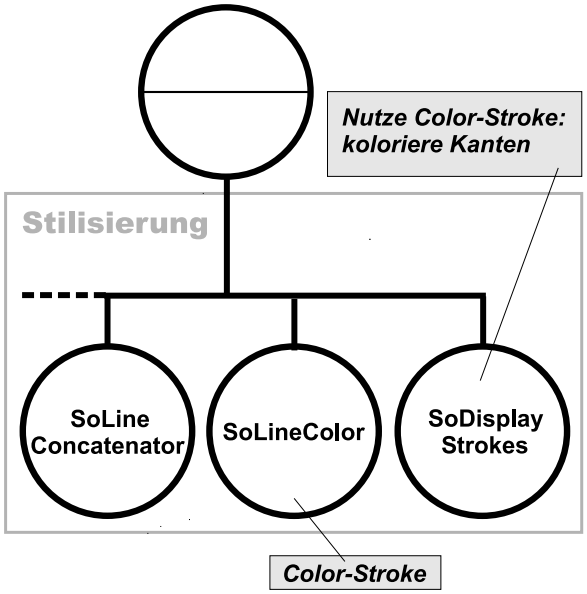
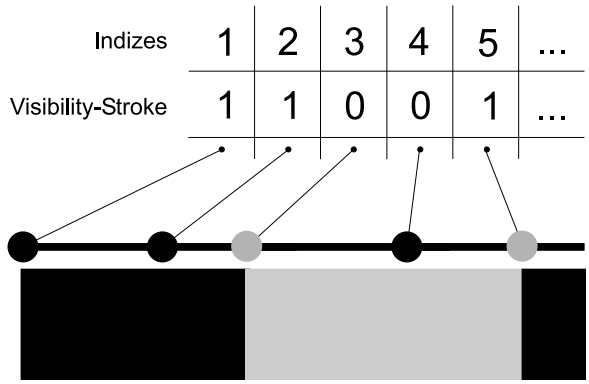


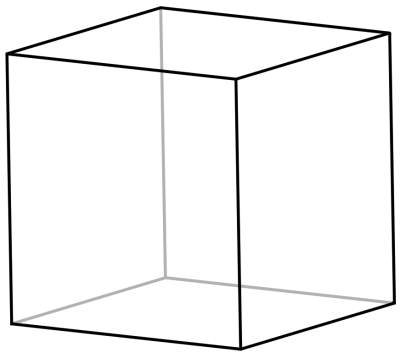
Abbildung 5.2: Dieser Szenengraph wird für die farbige Darstellung der gerenderten Objekte verwendet. Hierbei werden alle Kanten mit derselben Farbe belegt.

5.2 Visibility-Stroke

Der Visibility-Stroke repräsentiert die Eigenschaft der Sichtbarkeit der Kantensegmente. Dabei wird in Abhängigkeit vom Betrachterstandpunkt für jedes Kantensegment die Sichtbarkeit bzw. Nicht-Sichtbarkeit getestet und in Form von *true* und *false* im Visibility-Stroke gespeichert. Verändert sich die Sichtbarkeit im Laufe eines Segments, so wird ein neuer Vertex an der Stelle der Veränderung zum existierenden Kantenzug hinzugefügt. Abbildung 5.3(a) veranschaulicht den Zusammenhang der Datenstrukturen. Bei den grau gezeichneten Eckpunkten handelt es sich dabei um neu eingefügte Koordinaten. Der Color-Stroke kann dabei geschickt zur unterschiedlichen Kolorierung der sichtbaren und verdeckten Kanten eingesetzt werden. Abbildung 5.3(b) visualisiert diese G-Stroke-Kombination am Beispiel eines 3D-Würfelmodells. Hier wurden mit den Daten des Visibility-Strokes einmal die sichtbaren und im Anschluss daran die verdeckten Kanten gefiltert und dann mit Hilfe des vom Farb-Knoten `SoLineColor` erzeugten Color-Strokes koloriert.



(a) Datenlisten und korrespondierende Sichtbarkeit.

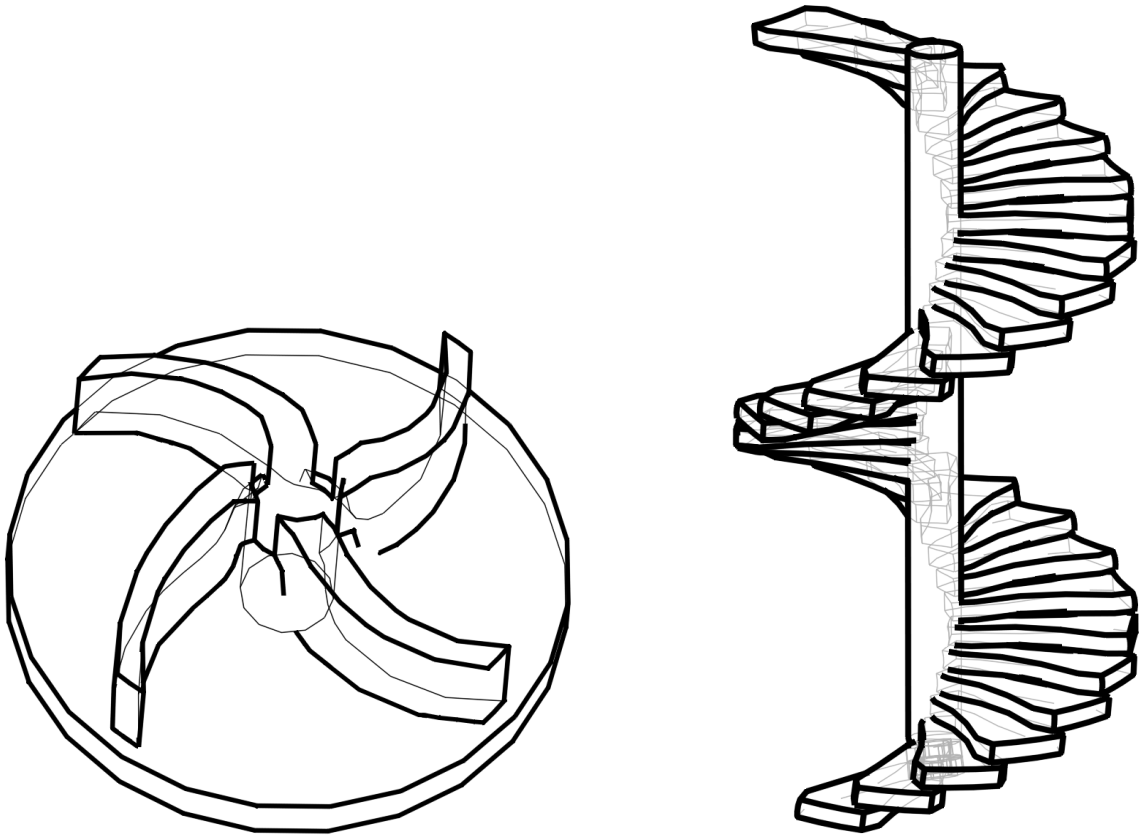


(b) Unterschiedliche Darstellung sichtbarer und verdeckter Kanten.

Abbildung 5.3: Die Abbildungen veranschaulichen den Aufbau und die Anwendung des Visibility-Strokes. Abbildung (a) stellt dabei den Zusammenhang zwischen Datenstrukturen und Sichtbarkeits-Eigenschaft dar. Die grauen Eckpunkte wurden auf Grund der sich veränderten Sichtbarkeit neu eingefügt. Abbildung (b) zeigt eine Liniengrafik, bei der sichtbare und verdeckte Kantensegmente unter Verwendung des Farb-Knotens unterschiedlich koloriert wurden.

Die unterschiedliche Darstellung sichtbarer und verdeckter Kanten in einem Bild war bisher mit OPENNPAR nicht möglich. Entweder mussten alle oder nur die sichtbaren Kanten in derselben Farbe oder mit derselben Textur dargestellt werden. Durch die Extraktion und Speicherung der Sichtbarkeits-Eigenschaft in einem G-Stroke ist die gleichzeitige und unterschiedliche Darstellung sichtbarer und verdeckter Kanten bei animierten Liniengrafiken möglich. Abbildung 5.4 zeigt zwei Beispiele für Liniengrafiken, die i. d. R. bei technischen Illustrationen eingesetzt werden (vgl. Abschnitt 2.1.3).

Die hierfür eingesetzten Szenengraphen sind sehr übersichtlich, da sie nur aus einer geringen Anzahl an Knoten bestehen. Abbildung 5.5 stellt den Szenengraphen für die in Abbil-



(a) Unterschiedliche Linienbreite.

(b) Unterschiedliche Linienbreite und Farbe.

Abbildung 5.4: Abbildung (a) stellt ein Scheibenrad dar, bei dem die verdeckten Kanten mit einer schmaleren Linienbreite gerendert wurden, als die sichtbaren Kanten. Abbildung (b) stellt eine Treppe dar, bei der die sichtbaren Kanten mit einer dicken schwarzen und die verdeckten Kanten mit einer dünneren grauen Linie gerendert wurden.

dung 5.4(b) gezeigte Liniengrafik dar. Dabei wird lediglich die Sichtbarkeits-Eigenschaft extrahiert und vom Filter-Knoten zur Filterung der verdeckten und sichtbaren Kanten benutzt. Der Subgraph besteht jeweils aus einem Farb- (`SoLineColor`) und einem Ausgabe-Knoten (`SoDisplayStrokes`). Letzterer kommt bei Liniengrafiken ohne Texturierung zum Einsatz. Da der entwickelte Filter-Knoten ausschließlich der Filterung und nicht der Veränderung der G-Strokes dient, ist die weitere Stilisierung der gefilterten Kantensegmente notwendig. In diesem Fall handelt es sich dabei um die Kolorierung und die damit einhergehende Erzeugung des Color-Strokes. Die Erzeugung des Color-Strokes *vor* der Filterung hätte zur Folge, dass für jeden Vertex derselbe Farbwert vorliegen würde. Da der Filter-Knoten den G-Stroke nicht verändern kann, wäre dieser somit unbrauchbar für eine unterschiedliche Kolorierung. Eine direkte Veränderung der G-Stroke-Werte wäre demgegenüber durch den Einsatz der noch nicht implementierten Stil- oder FilterStil-Knoten möglich (vgl. Abschnitt 3.4).

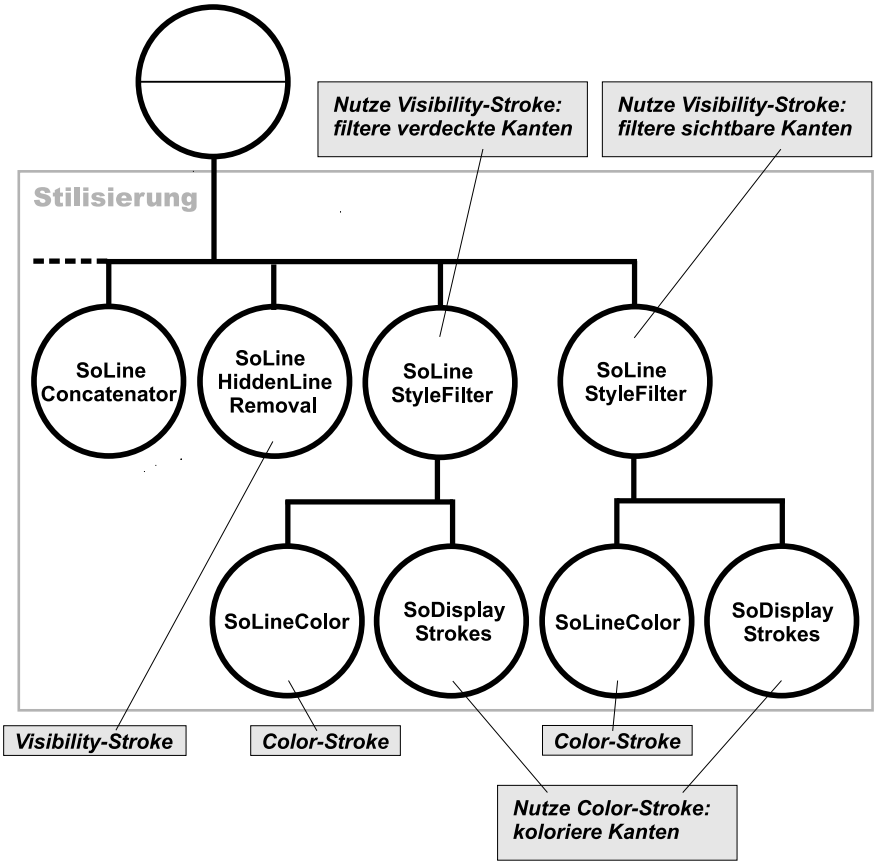
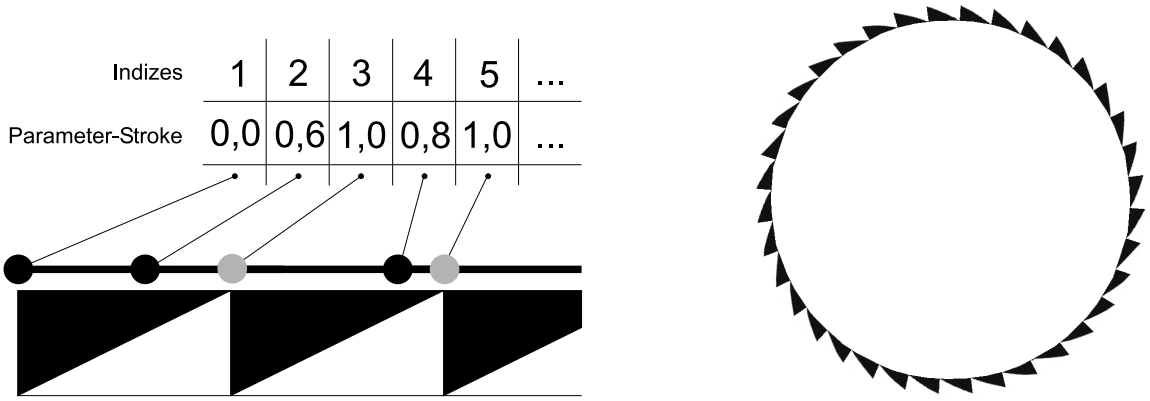


Abbildung 5.5: Dieser Szenengraph wird für die unterschiedliche Darstellung sichtbarer und verdeckter Kanten verwendet. Dabei werden zunächst die verdeckten Kanten mit einer grauen Farbe und im Anschluss die sichtbaren Kanten mit einem schwarzen Linienstrich gerendert.

5.3 Parameter-Stroke

Der Parameter-Stroke repräsentiert die Eigenschaft der Parameterisierung des Kantenzuges. Hierbei wird für jeden Eckpunkt ein Parameter im Bereich von 0,0 bis 1,0 berechnet, wobei eine Texturlänge der Parametereinheit 1,0 entspricht. Die Parameterisierung wird für eine gleichmäßige Texturierung des Kantenzuges benötigt, wobei die Texturkoordinaten immer einem Eckpunkt des Strokes zugeordnet werden. Die Textur wird sozusagen nacheinander auf den Stroke *geklebt*. Dabei kann es vorkommen, dass Kantensegmente länger als die Textur sind. Eine einfache Zuordnung von Parametern und Eckpunkten würde in so einem Fall zu einer verzerrten Darstellung der Textur führen, weshalb das Parameterisierungs-Element der Pipeline ggf. einen neuen Eckpunkt für das Ende der Textur berechnen und zum Segment hinzufügen muss. Abbildung 5.6(a) veranschaulicht diesen Zusammenhang. Um eine gleichmäßige Texturierung zu erzielen, muss für jeden Parameterwert 1,0 ein Eckpunkt vorliegen. Per Konvention entspricht die 1,0 innerhalb eines Strokes Texturende und -anfang, wird also im nächsten Schritt wie eine 0,0 behandelt. Abbildung 5.6(b) zeigt die durch die Parameterisierung hervorgerufene gleichmäßige Texturierung mit einer Dreiecks-Textur, welche den Anfang und das Ende eindeutig visualisiert.



(a) Datenlisten und korrespondierende Textur.

(b) Gleichmäßige Texturierung.

Abbildung 5.6: Die Abbildungen veranschaulichen den Aufbau und die Anwendung des Parameter-Strokes. Abbildung (a) stellt dabei den Zusammenhang zwischen Datenstrukturen und Texturierung dar. Abbildung (b) zeigt eine Liniengrafik, die nach der Vorgehensweise aus Abbildung (a) entstanden ist. Alle Kantensegmente wurden mit einer gleichmäßigen Textur belegt, es wurde ausschließlich der Parameter-Stroke verwendet.

Durch die gleichmäßige Texturierung ist es nun möglich, einfache Liniengrafiken zu erzeugen. Die Textur kann dabei frei nach den Vorstellungen des Anwenders entworfen werden. Bei OPENNPAR werden i. d. R. Texturen verwendet, die ein bestimmtes Zeichenwerkzeug repräsentieren sollen. Hierzu gehören z. B. Kohle-, Bleistift oder Wasserfarben-Texturen. Abbildung 5.7(a) zeigt das Modell einer Blüte, die mit einer Kohle-Textur gerendert wurde. Hierbei wurde nur der Parameter-Stroke verwendet, was zu einer relativ unübersichtlichen Grafik führt. Abbildung 5.7(b) zeigt dasselbe Modell, allerdings wurde hier neben dem Parameter- auch der Visibility-Stroke eingesetzt. Diese zweite Liniengrafik ist dementsprechend erheblich übersichtlicher. Hier wird deutlich, dass nun die Ansprüche an das Texturbild wachsen. Die verwendete Kohle-Textur bietet keinen fließenden Übergang, weshalb die Grafiken aus Abbildung 5.7 einen künstlichen Eindruck erwecken.

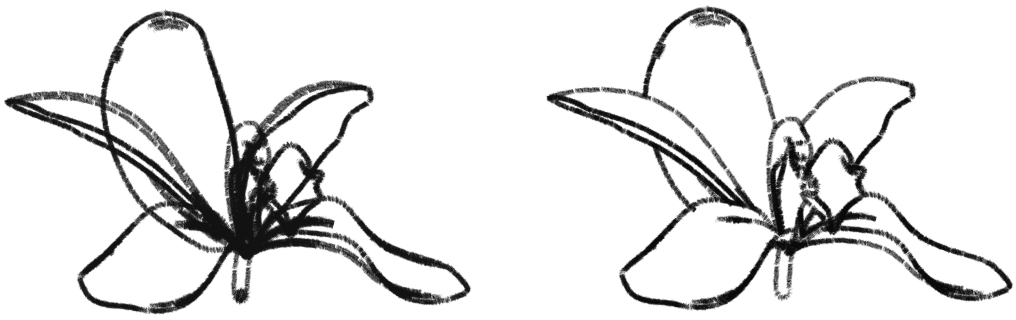
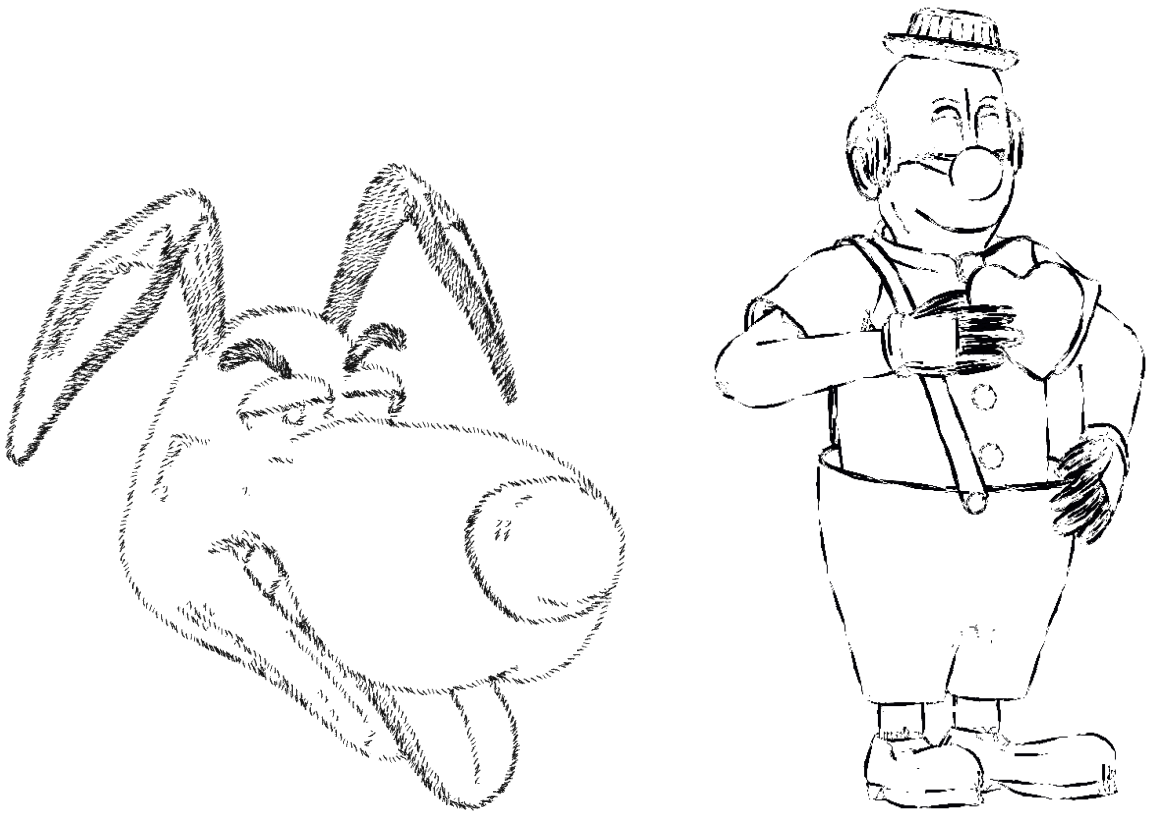


Abbildung 5.7: Die linke Abbildung zeigt die Liniengrafik einer Blüte, bei der nur der Parameter-Stroke und kein weiterer **G**-Stroke verwendet wurde. Alle Kanten sind mit einer gleichmäßig verteilten Kohle-Textur belegt. Die rechte Abbildung nutzt neben dem Parameter-Stroke auch den Visibility-Stroke, um nur die sichtbaren Kanten zu texturieren.

Durch den Einsatz von Texturen, die bei einer sequentiellen Anordnung einen fließenden Übergang ermöglichen, können im Gegensatz dazu realistischer wirkende Liniengrafiken erzeugt werden. Die Textur darf dabei über keine *abgehackten* Anfangs- und Endstellen verfügen. Sie sollten vielmehr einen nahtlosen Übergang simulieren, sodass z. B. bestimmte Zeichentechniken und -stile umgesetzt werden können. Abbildung 5.8(a) stellt das Modell eines Hundes dar, das mit einer Schraffur-Textur gerendert wurde. Da die Textur im Gegensatz zu der Kohle-Textur aus Abbildung 5.7 durchgängig bemalt ist, erzielt die gleichmäßige Anordnung den gewünschten Eindruck einer Schraffur-Zeichnung. Die in Abbildung 5.8(b) dargestellte Grafik eines Clowns erweckt im Gegensatz dazu *auf Grund* der unterteilten Textur den Eindruck einer skizzenhfaten Federzeichnung. Es kommt somit immer auf den zu erzielenden Effekt der Darstellung an.



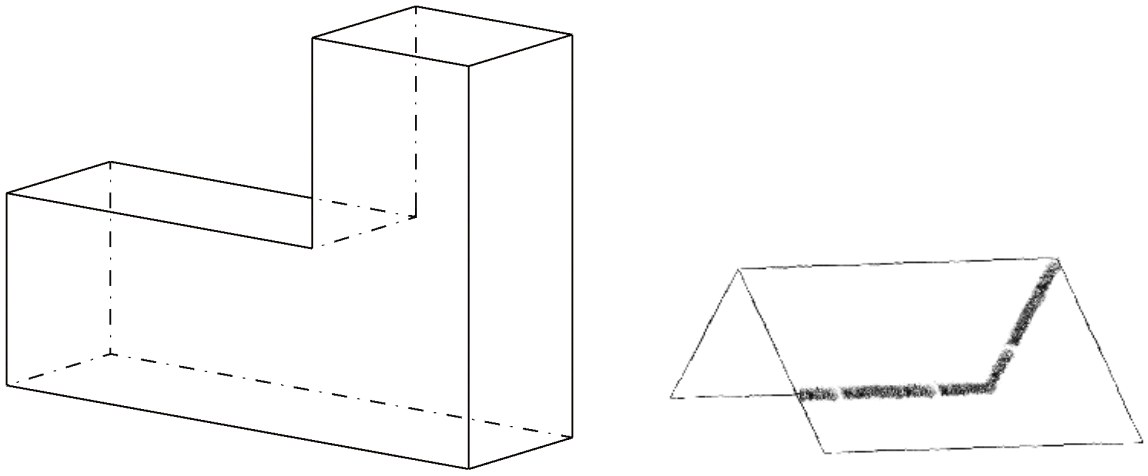
(a) Schraffierter Comic-Hund.

(b) Skizzenhafte Federzeichnung.

Abbildung 5.8: In der linken Abbildung wird durch die gleichmäßige Texturierung der sichtbaren Kanten der Eindruck einer schraffierten Zeichnung erweckt. Die rechte Abbildung erweckt hingegen gerade durch die eindeutigen Enden der Textur den Eindruck einer skizzenhaften Federzeichnung.

Neben diesen Techniken bietet der Einsatz von Parameter- und Sichtbarkeits-Stroke sowie des Filter-Knotens ebenfalls die Möglichkeit, technische Illustrationen wie in Abbildung 5.9(a) umzusetzen. Hierbei wurden die sichtbaren und die verdeckten Kanten mit zwei

verschiedenen Texturen belegt, wobei die für die verdeckten Kanten verwendete Textur aus einem Punkt und einem Strich besteht. Die gleichmäßige Parameterisierung führt nun dazu, dass der Eindruck einer gestrichelten Linie erweckt werden kann. Desweiteren sind erste Effekte möglich, wie der in Abbildung 5.9(b) durch das Patch angedeutete „Cut-Away-View“-Effekt, den TIETJEN (2004) in seiner Arbeit als zukünftige Technik anführt. Hierbei wird durch die unterschiedliche Dicke der Texturen der Fokus auf die verdeckten Kanten des Modells gelenkt. Dies erweckt den Eindruck, als sähe der Betrachter ins innere bzw. hinter die äußere Hülle des Modells.



(a) Technische Illustration.

(b) „Cut-Away-View“-Effekt.

Abbildung 5.9: In der linken Abbildung wird eine technische Illustration gezeigt. Diese wird durch die gleichmäßige Parameterisierung ermöglicht, da nun auch der Einsatz einer gestrichelten Textur zur Darstellung der verdeckten Linien möglich ist. Die rechte Abbildung lenkt den Fokus auf die verdeckten Kanten durch die Anwendung einer dicken Textur.

Eine solche technische Illustration wäre mit dem bisherigen Entwicklungsstand von OPENNPAR nicht möglich gewesen. Der hierfür verwendete Szenengraph ähnelt dem Szenengraph aus Abbildung 5.5. Statt der einfachen Kolorierung wird jedoch die Texturierung verwendet, was den Einsatz der dafür notwendigen Knoten erfordert und den Szenengraph umfangreicher werden lässt (vgl. Abbildung 5.10). Der erzeugte Visibility-Stroke wird für die Filterung der sichtbaren und verdeckten Kanten verwendet. Parameter-, Orientation- und Thickness-Stroke dienen der Texturierung und kommen erst beim Ausgabe-Knoten `SoLineShape` zum Einsatz. Der an zweiter Stelle platzierte Textur-Knoten dient lediglich als temporäre Variable für die Berechnung der Parameter, Orientierungen und Linienbreite-Werte. Visuell werden nur Textur A und Textur B eingesetzt.

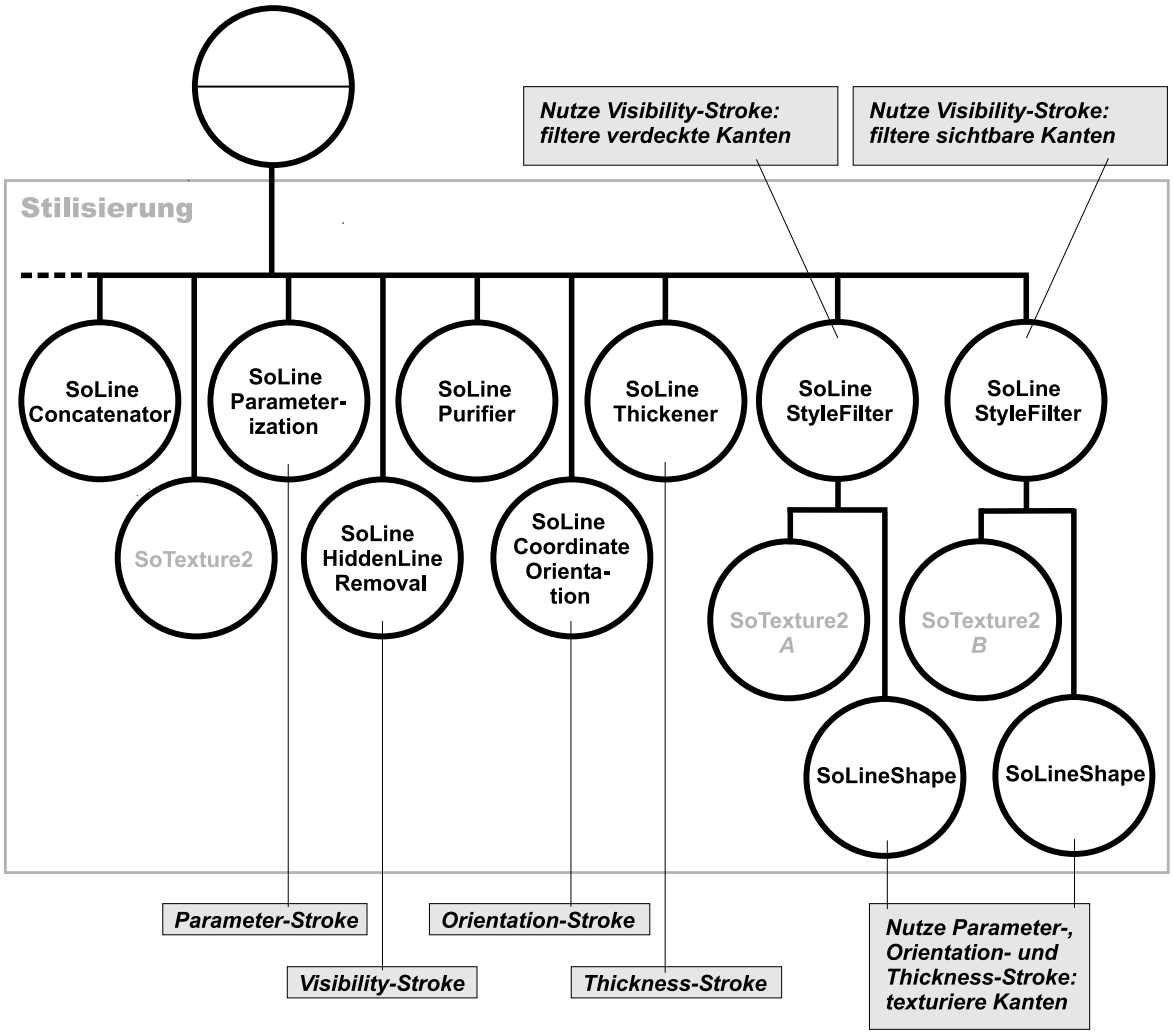
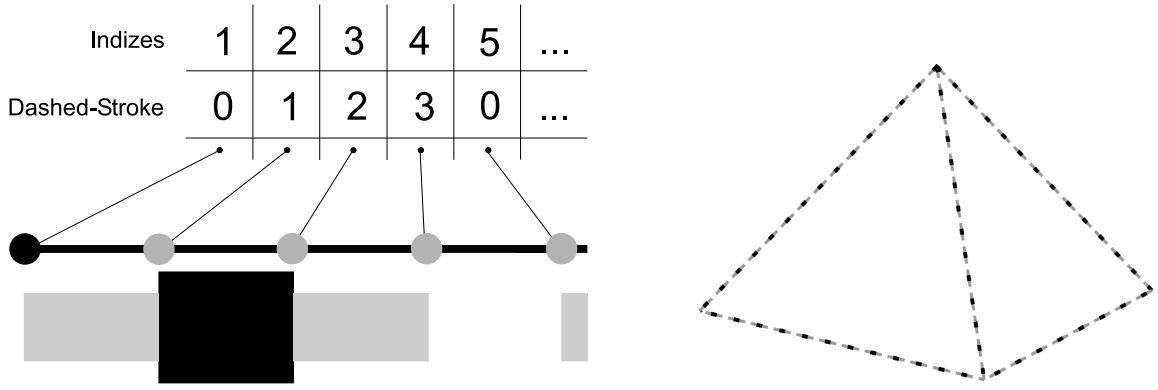


Abbildung 5.10: Dieser Szenengraph wird für die unterschiedliche Texturierung sichtbarer und verdeckter Kanten verwendet. Dabei werden zunächst die verdeckten Kanten mit einer Textur *A* dargestellt und im Anschluss die sichtbaren Kanten mit einer Textur *B*. Der Filter-Knoten nutzt hierfür jeweils den Visibility-Stroke, der Ausgabe-Knoten Parameter-, Orientation- und Thickness-Stroke.

5.4 Dashed-Stroke

Der Dashed-Stroke repräsentiert die Eigenschaft der Strichelung des Kantensegments in analytischer Form. Wurde im letzten Abschnitt 5.3 der Eindruck eines gestrichelten Kantenzuges mit Hilfe einer Textur hergestellt, so ermöglicht der Dashed-Stroke die unterschiedliche Darstellung einzelner Segmentteile. Dieser G-Stroke wird von dem Pipeline-Knoten `SoLineSubdivider` erzeugt, der jedes Kantensegment in Untersegmente teilt. An den betreffenden Stellen werden dabei lediglich neue Eckpunkte zum existierenden Kantenzug hinzugefügt. Beliebig viele Untersegmente können dann mit Hilfe des Dashed-Strokes unter-

schiedlich dargestellt werden. Sollen z. B. vier verschiedene Striche entstehen, so speichert der Dashed-Stroke pro Vertex nacheinander die Sequenz 0, 1, 2, 3 und bezieht sich wie auch der Visibility-Stroke auf die folgenden Kantensegmente. Der Filter-Knoten kann dann jeweils die Segmente 0, 1, 2 und 3 filtern und mit verschiedenen Farben oder Texturen belegen. Abbildung 5.11 verdeutlicht hierzu den Zusammenhang der Datenstrukturen und gibt ein Beispiel für die unterschiedliche Kolorierung der sichtbaren Kanten. Die grau gezeichneten Eckpunkte repräsentieren dabei neu eingefügte Koordinaten.



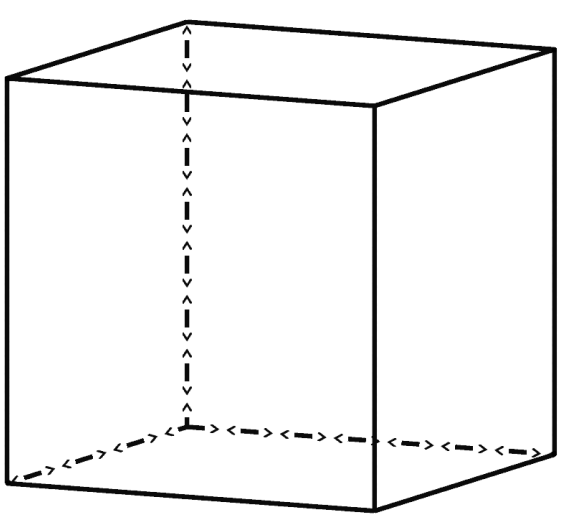
(a) Datenlisten und korrespondierende Strichelung.

(b) Gestrichelte Linien.

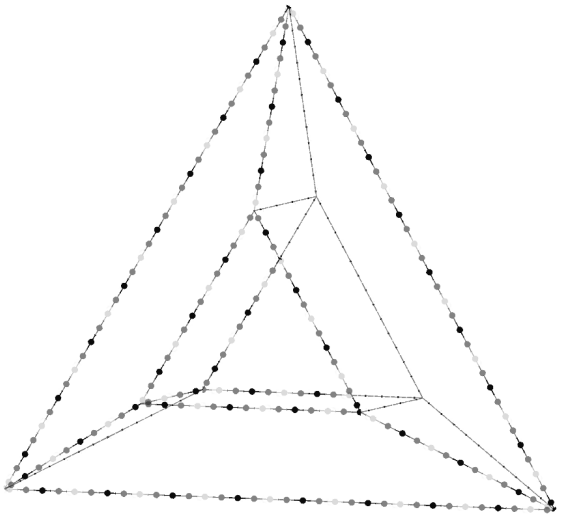
Abbildung 5.11: Die Abbildungen veranschaulichen den Aufbau und die Anwendung des Dashed-Strokes. Abbildung (a) stellt dabei den Zusammenhang zwischen Datenstrukturen und Strichelung dar. Abbildung (b) zeigt eine Liniengrafik, die nach der Vorgehensweise aus Abbildung (a) entstanden ist. Hier wurden die sichtbaren Kanten viermal nacheinander für die unterschiedlichen Striche gefiltert und jeweils mit einer anderen Farbe und Breite bzw. gar nicht gerendert.

Ein intuitives Anwendungsgebiet für den Dashed-Stroke ist natürlich die technische Illustration, bei der die verdeckten Kanten oft in gestrichelter Form dargestellt werden. Hierfür bietet der Dashed-Stroke eine Fülle an Möglichkeiten und Stil-Kombinationen. Abbildung 5.12 stellt drei Beispiele für technische Illustrationen mit dem Dashed-Strokes vor. Neben dem Dashed-Stroke kamen zusätzlich der Visibility-Stroke und der Parameter- bzw. der Color-Stroke zum Einsatz.

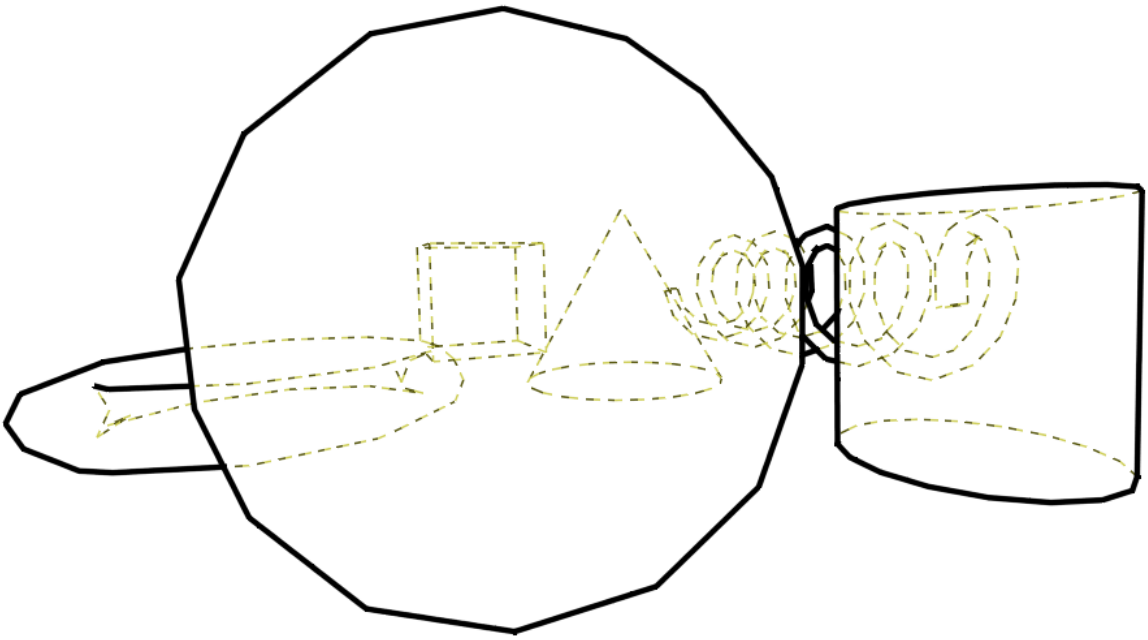
Neben dieser Zeichentechnik ermöglicht der Dashed-Stroke auch Effekte wie die Welligkeit der Kantenzüge ohne dafür die Geometrie verändern zu müssen. Die Darstellung von Kurvenförmigen Kanten war bei OPENNPAR bereits durch den Einsatz des Pipeline-Knotens `SoLinePerturbator` gegeben. Dieser Knoten nutzt dabei ebenfalls den `SoLineSubdivider`-Knoten, um die Untersegmente zu verschieben und dadurch Kurven zu erzeugen. Durch die gleichmäßige Parameterisierung und den Dashed-Stroke ist es nun jedoch möglich, einfach bestimmte Texturen zu verwenden, die den Eindruck einer Kurve wiedergeben (vgl. Abbildung 5.13(a)). Daneben können auch andere Effekte erzielt werden, wie z. B. die tuscheartige Fiederung des Vogels aus Abbildung 5.13(b) oder die unterschiedlich haarige Oberflächenstruktur des Affen aus Abbildung 5.13(c). Durch einfachen Austausch der jeweiligen Textur oder Untersegment-Textur können nun ohne aufwendige Berechnungen



(a)



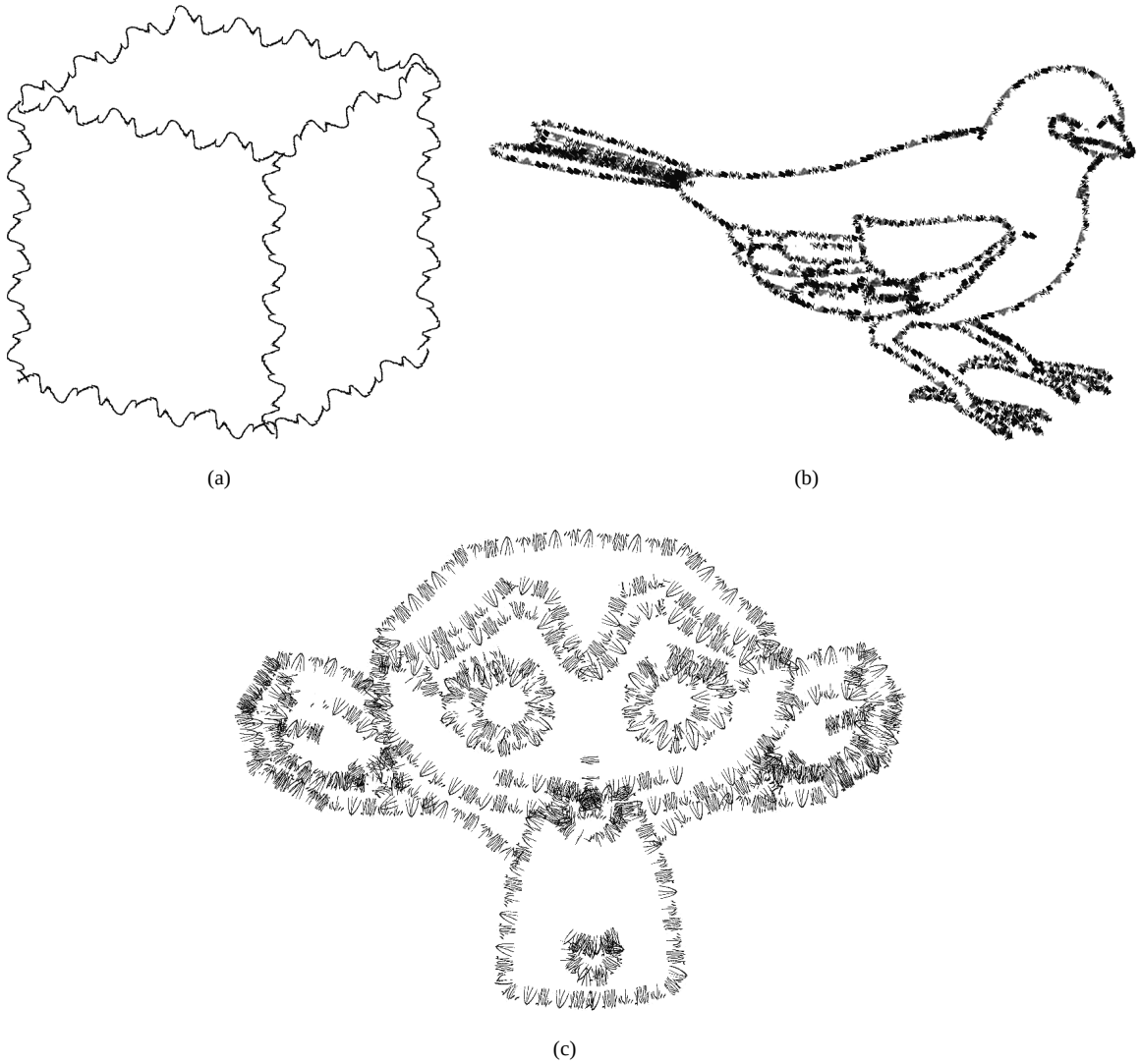
(b)



(c)

Abbildung 5.12: Abbildung (a) zeigt einen Würfel, dessen verdeckte Linien mit drei verschiedenen Texturen gezeichnet wurden. Hierbei handelt es sich um einen Pfeil nach links, einen Pfeil nach rechts und einen Strich. Der Dashed-Stroke ist hierbei für drei Striche erzeugt worden. Im Gegensatz dazu wurde für Abbildung (b) des Torus ein Dashed-Stroke mit vier Strichen angelegt. So konnten die sichtbaren Linien gleichmäßig von der hellen Punkte-Textur zur dunklen Punkte-Textur wechseln. Die verdeckten Kanten sind hierbei mit derselben Textur belegt. Die untere Grafik (c) zeigt mehrere Objekte, bei denen die verdeckten Linien schmaler als die sichtbaren und zusätzlich mit unterschiedlich gefärbten (gelb, grau und weiß) Strichen gerendert wurden.

verschiedenste Stile für dasselbe Modell erzeugt werden. Somit erhält der Programmierer eine wesentliche bessere Kontrolle über die Gestaltung der Grafik.



(a) (b)

(c)

Abbildung 5.13: Die linke Abbildung zeigt einen Würfel mit zwei Texturen eines unterschiedlich gekrümmten Strichs. Die mittlere Abbildung nutzt den Dashed-Stroke dazu aus, die „Haarigkeit“ des Affen-Modells zu variieren.

Abbildung 5.14 zeigt einen Szenengraph, der bei dem Würfel aus Abbildung 5.12 eingesetzt wurde. Dabei werden Visibility- und Dashed-Stroke für die Filterung der Kanten vom Filter-Knoten genutzt. Der Ausgabe-Knoten arbeitet wie auch im letzten Abschnitt 5.3 beschrieben mit dem Parameter-, Orientation- und Thickness-Stroke. Die gefilterten Kantensegmente bzw. Segmentteile werden dabei mit derselben Linienbreite aber unterschiedlichen Texturen dargestellt.

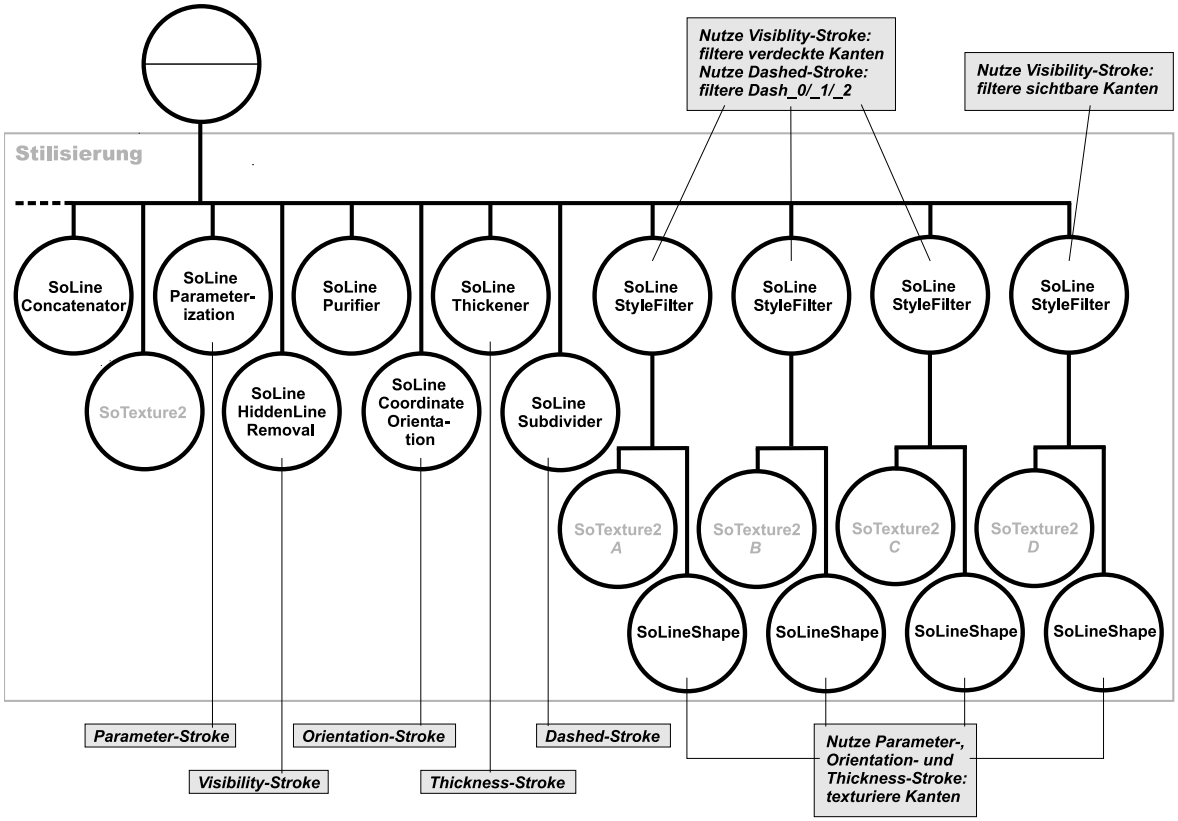
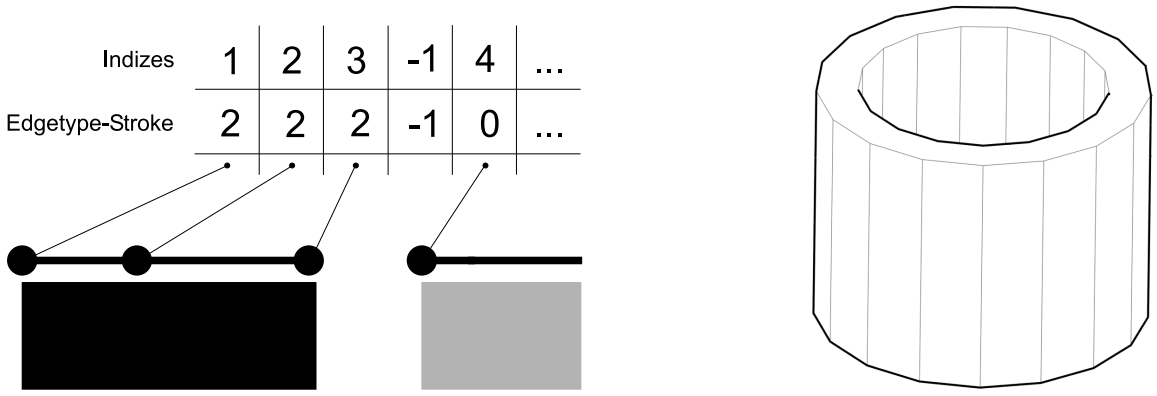


Abbildung 5.14: Dieser Szenengraph wird für den Einsatz des Dashed-Strokes verwendet. Dabei werden die Kanten zunächst in Untersegmente geteilt. Die verdeckten Kanten werden anhand des Dashed-Stroke gefiltert und pro Strich mit einer anderen Textur (*A, B, C*) belegt. Im Anschluss daran werden die sichtbaren Kanten gefiltert und mit einer Textur *D* gerendert. Der Filter-Knoten nutzt hierfür jeweils den Visibility- und den Dashed-Stroke, wohingegen der Ausgabe-Knoten die weiteren G-Strokes für die Texturierung verwendet.

5.5 Edgetype-Stroke

Der Edgetype-Stroke repräsentiert die Eigenschaft der Kantentyp-Zugehörigkeit. Bei der Kantenextraktion werden nacheinander Silhouetten-, scharfe und weitere Merkmalskanten berechnet. Jedem Typ wird eine bestimmte Identifikationsnummer (ID) zugeordnet und im Edgetype-Stroke gespeichert. Abbildung 5.15 veranschaulicht den Zusammenhang zwischen den Datenstrukturen. Der Edgetype-Stroke verändert die Geometrie nicht, sondern er beschreibt die extrahierten Kantenzüge. Der Kantentyp gilt dabei immer für alle Segmente eines gesamten Kantenzuges. Daher kommen verschiedene Kantentypen nur bei durch -1 getrennte Listen vor, siehe Abbildung 5.15(a).

Bisher konnte OPENNPAR nicht die Möglichkeit bieten, einzelne Linientypen unterschiedlich darzustellen. Durch die Einführung des Edgetype-Strokes und beflügelt durch die in den vorangegangenen Abschnitten vorgestellten G-Strokes können jetzt noch filigranere Liniengrafiken erzeugt werden. In Abbildung 5.16 werden zwei Liniengrafiken vorgestellt, bei



(a) Datenlisten und korrespondierende Kantentypen.

(b) Verschieden dargestellte Kanten.

Abbildung 5.15: Die Abbildungen veranschaulichen den Aufbau und die Anwendung des Edgetype-Strokes. Abbildung (a) stellt dabei den Zusammenhang zwischen Datenstrukturen und Typisierung dar. Abbildung (b) zeigt eine Liniengrafik, die nach der Vorgehensweise aus Abbildung (a) entstanden ist und bei welcher neben dem Edgetype-Stroke zusätzlich der Visibility- und der Color-Stroke verwendet wurden. Die Silhouettenkanten sind dabei durch dicke schwarze Linien, die inneren Merkmalskanten durch dünne graue Linien dargestellt.

denen die inneren Linien jeweils mit einer dünneren Textur als die Silhouettenlinien belegt wurden, entsprechend den klassischen Zeichentechniken aus Abschnitt 2.1.1. Um den Ein-

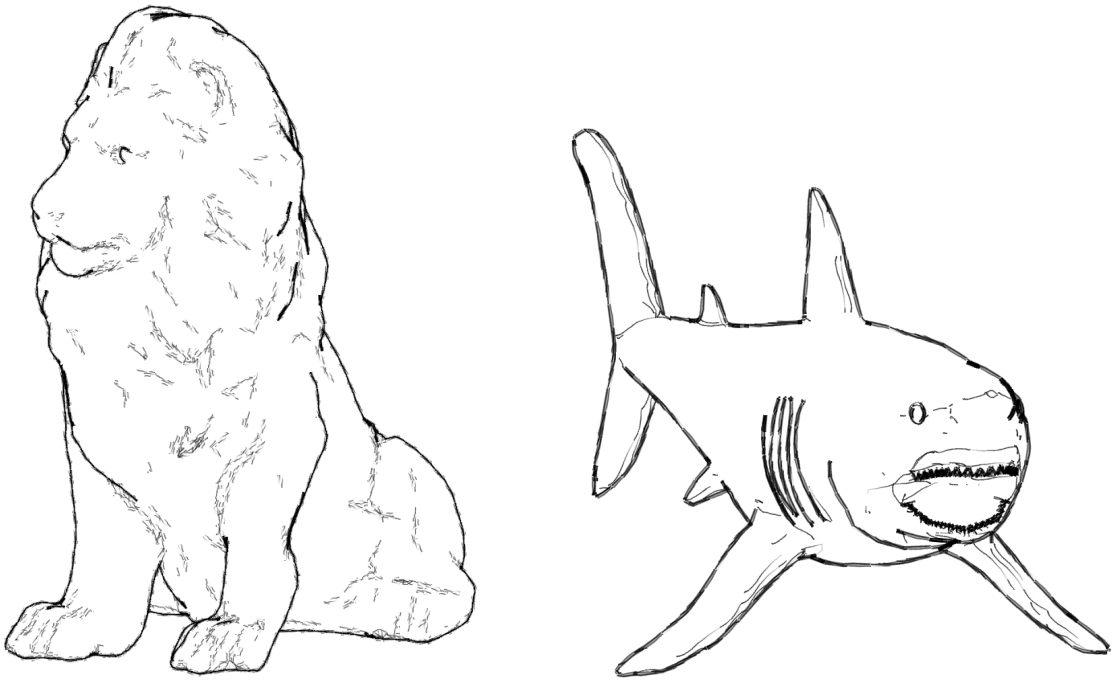


Abbildung 5.16: Bei diesen Abbildungen wurden die inneren Merkmalskanten mit anderen Texturen belegt als die Silhouettenkanten. Dieses Vorgehen fördert den Eindruck einer gezeichneten Grafik. Neben dem Edgetype-Stroke wurde der Visibility- und der Parameter-Stroke verwendet.

druck eines Fells zu verstärken, wurde die Textur der inneren Linien für Abbildung 5.16(a) mit einer leichten Schraffur versehen. Im Gegensatz dazu sollten die Texturen aus Abbildung 5.16(b) einen eher fließenden Eindruck erwecken und wurden dementsprechend in Anlehnung an Pinselstriche entworfen.

Durch die Möglichkeit der Filterung von G-Stroke-Daten im Filter-Knoten können nun auch Differenzierungen zwischen äußeren, inneren und verdeckten Linien gemacht werden, wie Abbildung 5.17 beweist. Für das dargestellte Schloss wurden zunächst alle verdeckten Kanten gefiltert und mit Hilfe des Dashed-Strokes texturiert. Im Anschluss daran wurden innere Kanten mit einer Bleistift- und alles Silhouettenkanten mit einer Wasserfarben-Textur belegt.

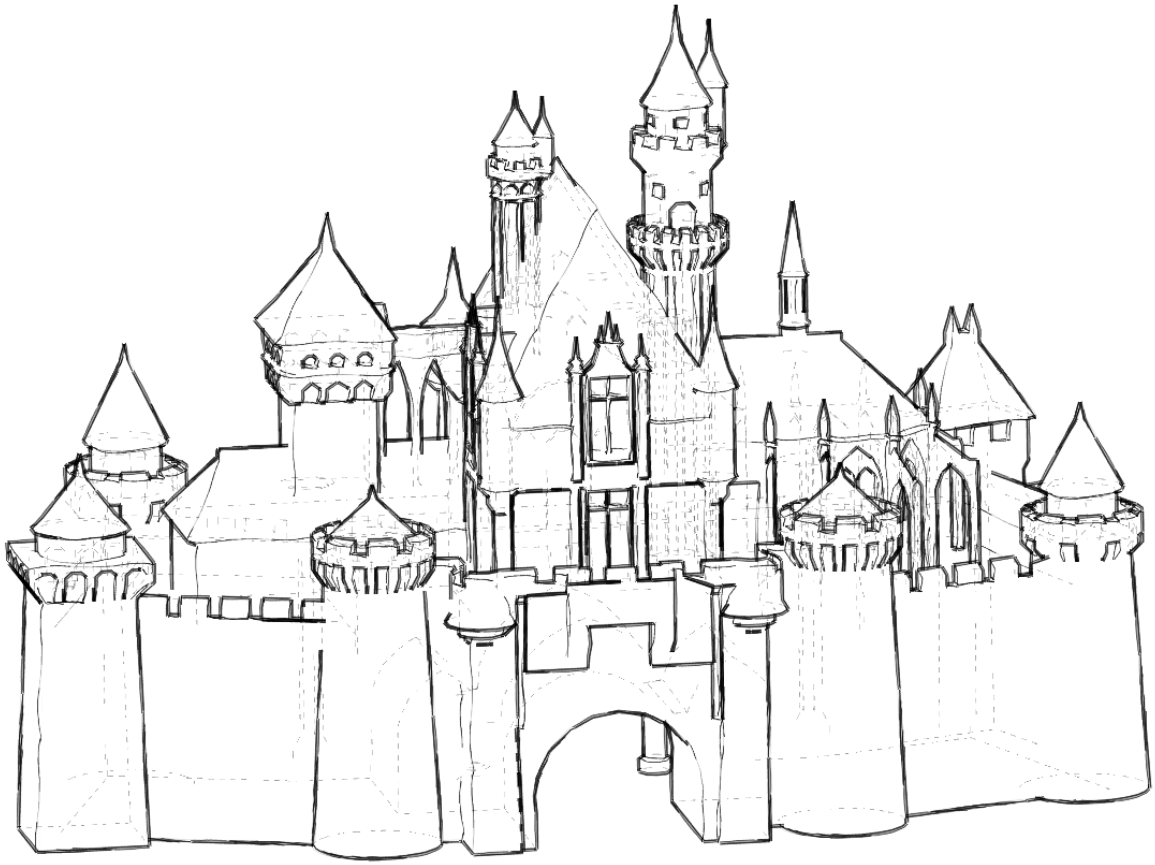


Abbildung 5.17: Diese Abbildung zeigt auf eindrucksvolle Weise die Möglichkeiten der Kombination mehrerer G-Strokes. So wurden für die unterschiedliche Darstellung der verdeckten, inneren und äußeren Kanten neben dem Edgetype-Stroke auch der Visibility-, Parameter- und der Dashed-Stroke eingesetzt.

Abschließend zeigt Abbildung 5.18 zwei Beispiele für die unterschiedliche Darstellung der scharfen und der Silhouettenkanten. Beide dort vorgestellten 3D-Modelle wurden mit Bleistift-Texturen belegt. Der in Abbildung 5.19 dargestellte Szenengraph veranschaulicht in diesem Zusammenhang Teile der Rendering-Pipeline, die für die in Abbildung 5.18 dargestell-

Abbildung 5.18: Die beiden Liniengrafiken veranschaulichen den Einsatz des Edgetype-Strokes sowie des Visibility- und des Parameter-Strokes. Hier wurden für die inneren Linien dünnere Bleistift-Texturen als für die äußeren Kanten verwendet.

ten Liniengrafiken verwendet wurde. Zunächst wurden hierbei die Kanten bzgl. ihres Typs kategorisiert und im Edgetype-Stroke gespeichert. Letzterer wurde neben dem Visibility-Stroke vom Filter-Knoten zur Filterung der gewünschten Kanten verwendet. Der Subgraph beinhaltet neben dem Textur- und dem Ausgabe-Knoten hier zusätzlich den Linienbreite-Knoten `SoLineThickener`, um die gefilterten Kanten mit unterschiedlichen Texturen und Linienbreiten darstellen zu können. Dafür sind zunächst die Silhouetten- und im Anschluss daran die scharfen Kanten gefiltert und jeweils entsprechend der Subgraph-Knoten gerendert worden.

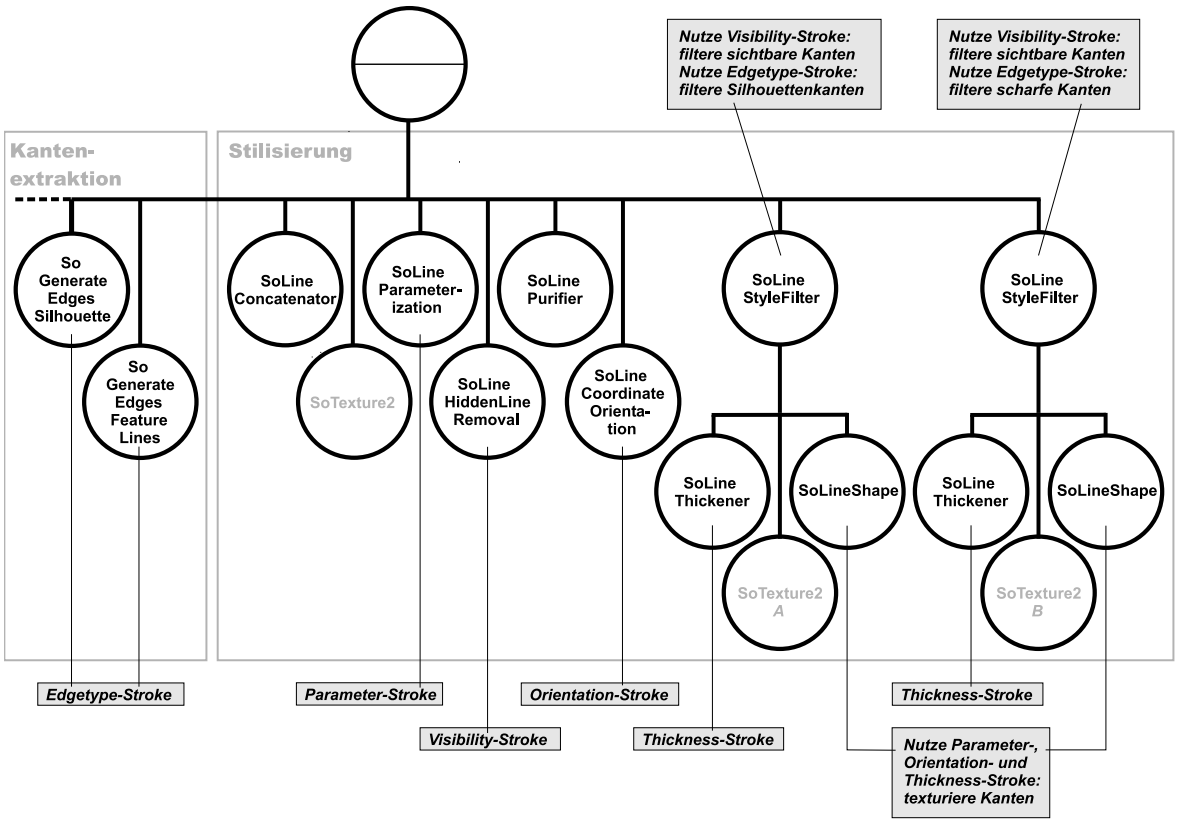
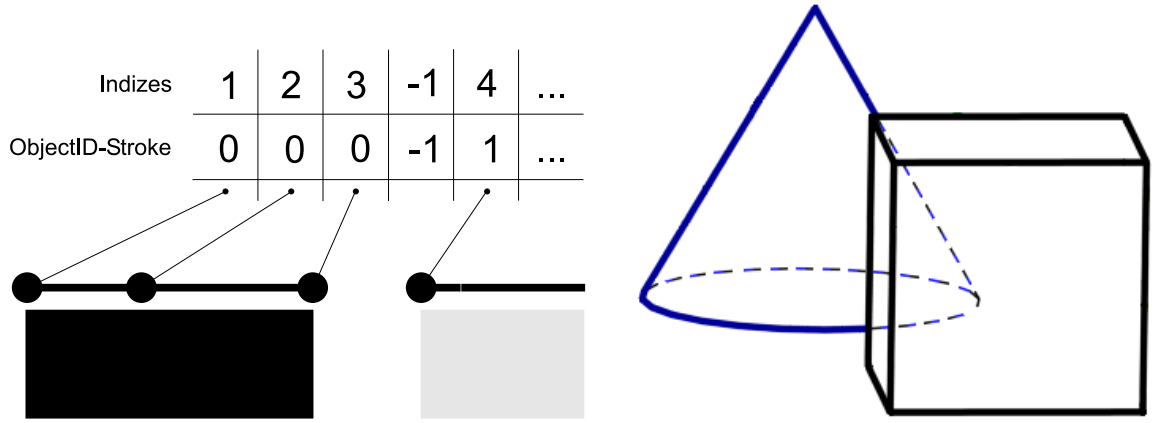


Abbildung 5.19: Dieser Szenengraph wird für den Einsatz des Edgetype-Strokes verwendet. Dabei werden die Kanten zunächst in bzgl. ihres Typs im **G**-Stroke gespeichert. Die sichtbaren Kanten werden dann anhand des Edgetype-Strokes gefiltert und mit unterschiedlichen Texturen (*A, B*) und Linienbreiten gerendert.

5.6 ObjectID-Stroke

Der ObjectID-Stroke repräsentiert die Eigenschaft der Objekt-Zugehörigkeit der Kanten. Die von OPENNPAR verwendeten 3D-Modelle oder 3D-Szenen bestehen ggf. aus mehreren Objekten, welche mit Hilfe dieses **G**-Strokes unterschiedlich gerendert werden können. Die Erzeugung des ObjectID-Strokes geschieht wie auch beim Edgetype-Stroke während der Kantenextraktion. Da sich die einzelnen Strokes immer auf ein Objekt beziehen, gilt auch hier, dass immer der gesamte Kantenzug zu einem Objekt gehört. Die ObjektID wird pro Vertex gespeichert und bezieht sich jeweils auf das folgende Kantensegment. Abbildung 5.20 veranschaulicht auch hier den Zusammenhang zwischen den Datenstrukturen und stellt ein einfaches Beispiel für die unterschiedliche Darstellung mehrerer Objekten vor.

Die unterschiedliche Darstellung der 3D-Modelle konnte bisher ebenfalls nicht mit OPENNPAR umgesetzt werden. Zwar gelang es TIETJEN (2004) verschiedene Objekte mit unterschiedlichen Farben darzustellen, doch war der Aufwand für den Aufbau der Szenengraphen nicht unerheblich. Durch den Einsatz des ObjectID-Strokes und des Filter-Knotens können



(a) Datenlisten und korrespondierende ObjektIDs.

(b) Gestrichelte Linien.

Abbildung 5.20: Die Abbildungen veranschaulichen den Aufbau und die Anwendung des ObjectID-Strokes. Abbildung (a) stellt dabei den Zusammenhang zwischen Datenstrukturen und Objekt-Zugehörigkeit dar. Abbildung (b) zeigt eine Liniengrafik, die nach der Vorgehensweise aus Abbildung (a) entstanden ist.

die verschiedenen Objekte nun relativ einfach mit unterschiedlichen Farben und Texturen belegt werden. Abbildung 5.21 zeigt in diesem Zusammenhang zwei Beispiele für die medizinische Visualisierung eines Leberdatensatzes und der umgebenden Struktur durch die Verwendung von unterschiedlich gefärbten Silhouettenlinien. Hierzu wurden pro ObjektID die

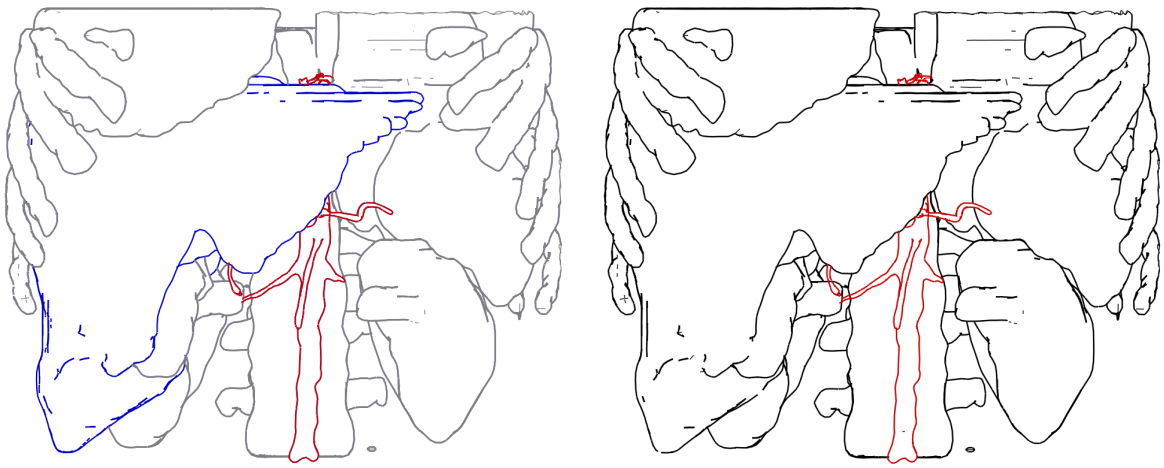


Abbildung 5.21: Die beiden oberen Bilder sind zwei Beispiele für die medizinische Visualisierung eines Leberdatensatzes. Links wurden dafür alle Objekte mit unterschiedlich kolorierten Silhouettenkanten gerendert, rechts wurde nur die Kontur der Aorta als Hervorhebung von den anderen Objekten mit einer unterschiedlichen Farbe gezeichnet. Für die Darstellung kamen neben dem ObjectID-Stroke der Visibility- und der Color-Stroke zum Einsatz.

sichtbaren Silhouettenkanten gefiltert und mit einer Farbe belegt. Abbildung 5.21(b) filtert und koloriert zunächst alle Silhouettenkanten in derselben Farbe, um dann in einerm zweiten Filterschritt nur die Aorta mit einer roten Kontur hervorzuheben. Die Abbildung 5.6 nutzt

zusätzlich noch den Dashed-Stroke aus, um die verdeckten Kanten der einzelnen Objekte gestrichelt und mit unterschiedlichen Farben darzustellen. Die unterschiedliche Darstellung der

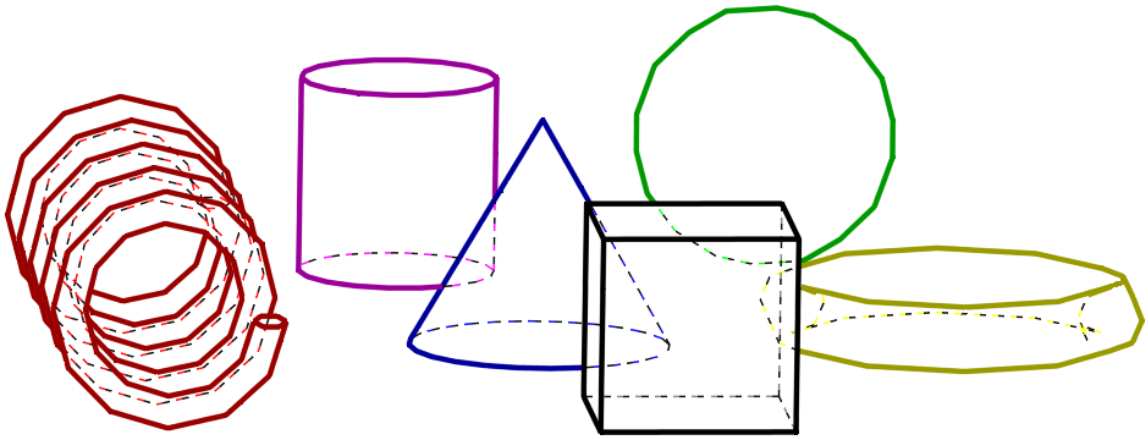


Abbildung 5.22: Die Abbildung stellt mehrere Objekte unterschiedlich dar und koloriert zusätzlich die verdeckten Kanten in verschiedenen jedoch zum Objekt gehörenden Farben. Neben dem ObjectID-Stroke wurden der Visibility-, Color- und der Dashed-Stroke eingesetzt.

einzelnen Objektteile kann natürlich auch mit Texturen erfolgen. So zeigt Abbildung 5.23 die Liniengrafik eines Stiers, bei der die einzelnen Objekte mit verschiedenen Texturen belegt wurden. Es wurde dabei darauf geachtet, den Texturstil an das jeweilige Objekt anzupassen. Bei der Grafik kamen lediglich der ObjectID-Stroke sowie der Visibility- und der Parameter-Stroke zum Einsatz.

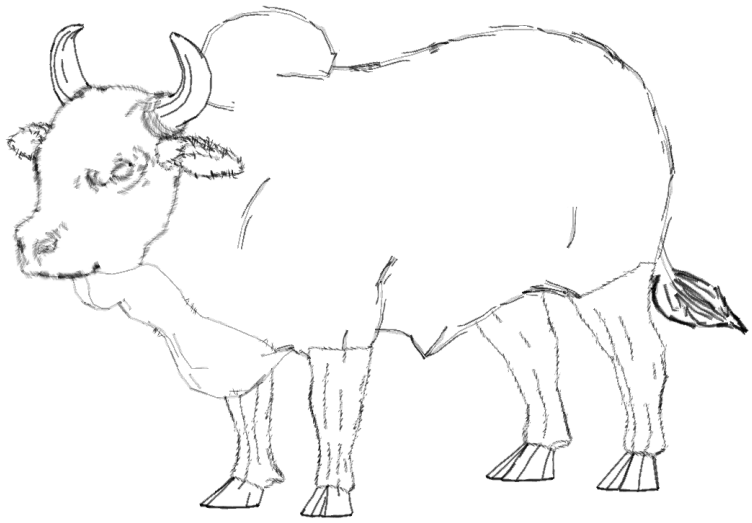


Abbildung 5.23: Diese Abbildung zeigt die Liniengrafik eines Stiers, bei der die einzelnen Objektteile mit unterschiedlichen Texturen belegt wurden.

Abbildung 5.24 zeigt den Szenengraph, der für die in Abbildung 5.20(b) dargestellte Liniengrafik aufgebaut wurde. Hier wird zunächst die Objekt-Zugehörigkeit der Kanten im ObjectID-Stroke festgehalten. Durch Filterung der sichtbaren Kanten des Objekts *A* und der sichtbaren und verdeckten Kanten des Objekts *B* können hier eine Reihe von geometrischen Eigenschaften ausgenutzt und auf unterschiedliche Art und Weise dargestellt werden. Die Schlichtheit des Szenengraphs belegt abermals die Effektiviät des G-Stroke-Konzepts.

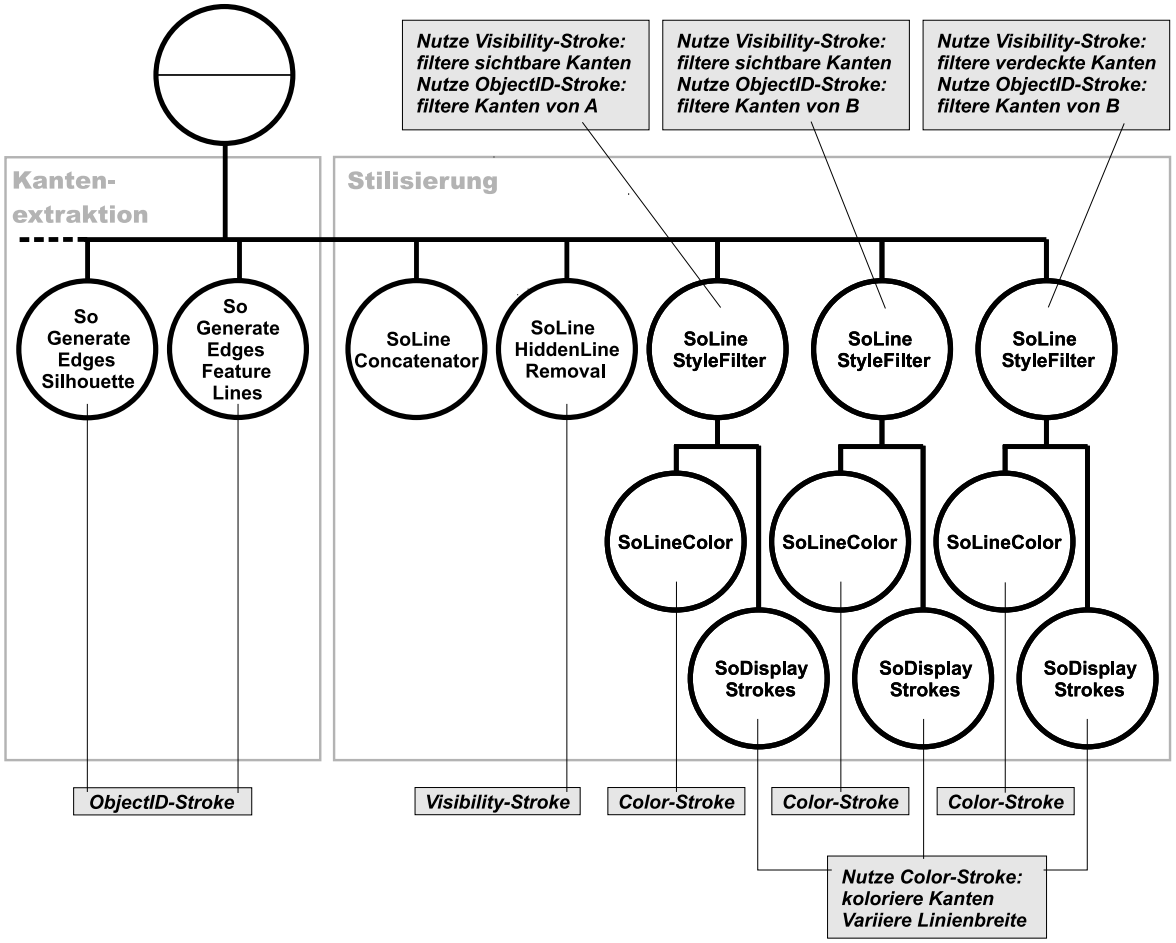


Abbildung 5.24: Dieser Szenengraph wird für den Einsatz des ObjectID-Strokes verwendet. Dabei werden die Kanten zunächst bzgl. ihrer Objekt-Zugehörigkeit im ObjectID-Stroke gespeichert. Danach werden die sichtbaren Kanten von Objekt *A* mit einer Farbe und einer bestimmten Dicke gezeichnet. Im Anschluss daran werden erst die sichtbaren und dann die verdeckten Kanten von Objekt *B* mit unterschiedlichen Linienbreiten in derselben Farbe gezeichnet.

5.7 Zusammenfassung

Dieses Kapitel hat an einer Reihe von Beispielgrafiken gezeigt, dass das Konzept der G-Strokes funktioniert und die Realisierung erfolgreich eingesetzt werden konnte. Obwohl die Elemente Stil- und FilterStil-Knoten aus Zeitgründen nicht implementiert werden konnten, hat bereits der Einsatz des Filter-Knotens und der damit verbundenen Filterung der Kantensegmente durch die G-Informationen zu Effekten und Stilkombinationen geführt, die bisher nicht oder nur unter hohem Aufwand hätten erzielt werden können.

Die vorgestellten G-Strokes sind hier in erster Linie als Basis zu verstehen. Sie repräsentieren die grundlegenden Eigenschaften der Kantenzüge und bestätigen, dass die Extraktion und Kombination bzw. Anwendung gewisser (geometrischer) Merkmale für die Stilgebung eingesetzt werden kann. Durch das implementierte System kann nun auf dieser Basis aufgebaut werden. Das Hinzufügen neuer Eigenschaften ist dabei in einem vorgefassten Rahmen leicht realisierbar und eröffnet dementsprechend eine Fülle von weiteren möglichen G-Strokes, auf welche im folgenden Kapitel 6 eingegangen wird.

Die Animation der einzelnen Liniengrafiken kann hier nicht gezeigt werden und ist auch nicht für alle gezeigten Abbildungen möglich. Die Ursache liegt hierbei jedoch in erster Linie an der Größe der Polygonnetze und weniger an den G-Strokes selbst. Abschließend bleibt zu sagen, dass mit diesem Kapitel der realisierte Entwurf die gewünschten Ergebnisse in Form der vorgestellten Grafiken erzielen konnte.

Kapitel 6

Fazit

In dieser Diplomarbeit wurde ein neues Konzept zur Verbesserung der automatischen Generierung von Liniengrafiken entwickelt. In diesem abschließenden Kapitel werden im ersten Abschnitt die Ergebnisse der Arbeit zusammengefasst und bewertet. Der zweite Teil listet einige weiterführende Entwicklungsmöglichkeiten auf, mit denen das Konzept der G-Strokes zukünftig ausgebaut und erweitert werden kann.

6.1 Zusammenfassung

Das in dieser Diplomarbeit entwickelte Konzept der G-Strokes hat auf eindrucksvolle Weise gezeigt, dass die logische Datentrennung in Geometrie und Eigenschaften zu einer neuen Datenverwaltung und somit zu einer erheblich einfacheren Erzeugung von Liniengrafiken führt. Hierdurch konnten erstens eine Reihe von Problematiken alter Stilisierungs-Pipeline-Anwendungen gelöst und zweitens die Entwicklung neuer Methoden und Stile gefördert werden. Zentral bei dieser Arbeit ist dabei der gelungene Versuch, die Vorgehensweise des klassischen Zeichnens softwaretechnisch umzusetzen, also bestimmte Objekteigenschaften zu extrahieren und für die Stilgebung der Linien zu nutzen. Die von KOSCHATZKY in Abschnitt 2.1.1 formulierte Forderung, dass eine Linie Ausdruck tragen müsse, wurde in Form von Stroke (*Linie*) und G-Stroke (*Ausdruck*) realisiert (KOSCHATZKY, 1993).

Der G-Stroke ist als ein dynamischer Datentyp zur Repräsentation der Eigenschaften des Strokes entwickelt worden. Die verwendeten Datenstrukturen können unterschiedliche allgemeine Datentypen wie Integer oder Float aber auch 2D- und 3D-Vektoren speichern und unabhängig von der Stilisierungs-Pipeline verwalten. Da sich der G-Stroke dynamisch an den Stroke anpasst, handelt es sich um einen weiterentwickelteren und umfangreicheren Datentyp als es bei dem G-Buffer der Fall ist. Letzterer ist ein statischer Datentyp, der lediglich der Datenspeicherung zweidimensionaler Werte dient.

Die Beziehung zwischen Stroke und G-Stroke ist in Form einer doppelten Abhängigkeit entworfen und realisiert worden. Dabei ermöglicht nur der Stroke den Elementen der Stilisierungs-Pipeline Zugriff auf die G-Strokes, also die ihn beschreibenden Eigenschaften. Mit diesen Eigenschaften ist es dann im Gegenzug einem bestimmten Pipeline-Element erlaubt, dem Stroke einen Stil zuzuweisen.

Die Datentrennung in Geometrie und Eigenschaften führte desweiteren zu einer neuen Aufgabenverteilung der Stilisierungs-Pipeline-Elemente. Diese müssen nur noch die Kanten-Geometrie des Objekts in Form des Strokes, jedoch keine weiteren Datenstrukturen verwalten. Ihre Aufgaben lassen sich klar in die Extraktion und Speicherung bestimmter Eigenschaften in den G-Strokes sowie die Veränderung der Kantenzüge und die abschließende Stilgebung durch Ausnutzung der G-Strokes einteilen. Da sich die extrahierten G-Strokes automatisch und unabhängig von den Pipeline-Elementen an jegliche Veränderung des Kantenzuges anpassen, ist eine einfache Weiterentwicklung neuer G-Strokes und neuer Pipeline-Elemente gegeben. Das Hinzufügen neuer Eigenschaften führt ggf. nur zur Entwicklung eines neuen Pipeline-Knotens, der die Eigenschaft extrahiert und den G-Stroke erzeugt. Alle weiteren Pipeline-Knoten bleiben davon unberührt.

Die anfängliche Frage, ob und inwieweit geometrische Eigenschaften für die Stilgebung genutzt werden können, ist mit den im letzten Kapitel vorgestellen Fallbeispielen eindeutig belegt worden. Zeichentechniken wie die unterschiedliche Darstellung innerer und äußerer Linien können jetzt auf einfache Art und Weise durch den Edgetype-Stroke automatisch umgesetzt werden. Auch die Kontextabhängigkeit kann z. B. durch den Einsatz geeigneter Texturen erzielt werden, welche mit Hilfe des Parameter-Strokes gleichmäßig auf dem Kantenzug angebracht werden können. Ebenso können bestimmte Objektdetails mit Hilfe des ObjectID-Strokes hervorgehoben werden. Das Konzept ermöglicht desweiteren durch den Dashed-Stroke eine sehr breite Palette an technischen Illustrationen.

Die Umsetzung des Konzepts ist programmiersprachlich sehr eng an den Entwurf angelehnt. Neben den entworfenen G-Strokes konnten weitere Eigenschaften realisiert werden, die zur Stabilität der Implementierung selbst beigetragen haben (u. a. der Prioritäts-Stroke). Die implementierten Klassen nutzen hierfür eingehend die Paradigmen objektorientierter Softwareentwicklung. Desweiteren wurde mit der Implementierung eine einfache Anwendung der G-Strokes erreicht, die insbesondere für neue Entwickler gewünscht worden war. Durch den Aufbau einer streng formulierten Klassenhierarchie und den Klassenbeziehungen untereinander ist die Weiterentwicklung neuer G-Strokes an das entwickelte Konzept gebunden, das dadurch stabil bleibt.

Abschließend bleibt zu sagen, dass das Konzept der G-Strokes dem von DURAND (2002) in Abschnitt 2.2.5 geforderten modularen System zur Generierung von Liniengrafiken entspricht bzw. das modulare System OPENNPAR diesbezüglich noch weiter verbessert hat. Da die Eigenschaften zur Stilgebung genutzt werden können, ist die Anordnung der Pipeline-Elemente sehr variabel geworden. Es muss lediglich darauf geachtet werden, dass die Kanten zuerst extrahiert, dann verändert und schließlich mit einem Stil versehen werden. Durch die Anlehnung an die klassische Erzeugung einer Zeichnung wurde hier ein stabiles System zur automatischen Generierung von Liniengrafiken entwickelt und angewendet. Die in der Motivation genannten Anforderung konnten somit erfolgreich umgestzt werden.

6.2 Ausblick

Das in dieser Arbeit entwickelte G-Stroke-Konzept ist als eine Basis für das Entwickeln und Hinzufügen weiterer Eigenschaften zu verstehen. Da das Verwaltungskonzept streng durchstrukturiert ist, ist das Hinzufügen neuer an das Konzept angepasster Eigenschaften leicht durchführbar. Hier sollen Ideen für zukünftige Weiterentwicklungsmöglichkeiten vorgestellt werden.

Zunächst ist dabei die Umsetzung von Stil- und StilFilter-Knoten zu nennen, da insbesondere der zweite den Aufbau von Subgraphen überflüssig macht. Mögliche Stil-Knoten könnten z. B. den Haloed-Line-Effekt oder einen Loose-and-Sketchy-Stil erzeugen. Hierzu gehört ebenfalls der bereits begonnende `SoLineMerger`-Knoten, der verschiedene Stile miteinander verbinden können soll. Für die Förderung technischer Illustrationen wäre die Entwicklung eines G-Stroke empfehlenswert, der pro Kantensegment die Anzahl der verdeckenden Kanten speichert und dadurch z. B. Liniengrafiken wie die von KAMADA und KAWAI (1987), bei denen die Kantenzüge unterschiedlich in Abhängigkeit von ihrem Verdeckungsgrad dargestellt wurden, ermöglicht.

Die Auswahl unterschiedlicher Texturen könnte außerdem durch den Einsatz eines Texture-Strokes vereinfacht werden, der z. B. von einem StilFilter-Knoten gesetzt und somit abhängig von einem anderen G-Stroke pro Kantensegment auf die gewünschten Texturen verweist. In diesem Zusammenhang bietet sich auch die Entwicklung eines Kantenende-Strokes an, der die in Abschnitt 5.3 beschriebene Problematik der Texturart beheben würde. Texturen, die nicht durchgängig bemalt sind, sondern einen kurzen Strich repräsentieren, erzeugen bei einer gleichmäßigen Parameterisierung den Eindruck von abgehackten Linienstrichen. Der Kantenende-Stroke würde diese Problematik umgehen, indem er die Kantenenden markieren würde. Die Textur müsste dann in drei Teile geteilt werden, wobei der mittlere Teil durchgängig bemalt ist und innerhalb des Kantenzuges auf die Kanten gelegt werden müsste. Anfang und Ende der Textur könnten dann entsprechend auf den Kantenanfang und das Kantenende gelegt werden, sodass auch mit diesen Texturen der Eindruck eines langen Striches entsteht.

Neben diesen G-Strokes könnten auch weitere geometrische Informationen wie die Beziehung zwischen Objekt und Untergrund dargestellt werden. Zum Beispiel könnten die Normalen für die Erzeugung eines Lage-Strokes genutzt werden, der die Lage des Objekts repräsentieren bzw. einen Schwerpunkt simulieren würde. Anhand des Schwerpunkts könnte dann der Linienstil beeinflusst werden, wie in Abbildung 6.1 zu sehen ist. Eine weitere Anwendung für den G-Stroke ist die Interaktion von Eigenschaften. So könnte der Interaction-Stroke z. B. die Verknüpfungspunkte zweier unterschiedlicher Kantentypen markieren und eine fließenden Übergang dieser ermöglichen. Verschiedene Stile könnten somit ineinander übergehen, was ein zukünftiges Ziel von KALNINS et al. (2003) ist. Ebenso könnte ein Frame-Coherence-Stroke mit Hilfe von Pixeldaten die Texturpositionen des vorherigen Frames speichern, um fließende Animationen zu ermöglichen.

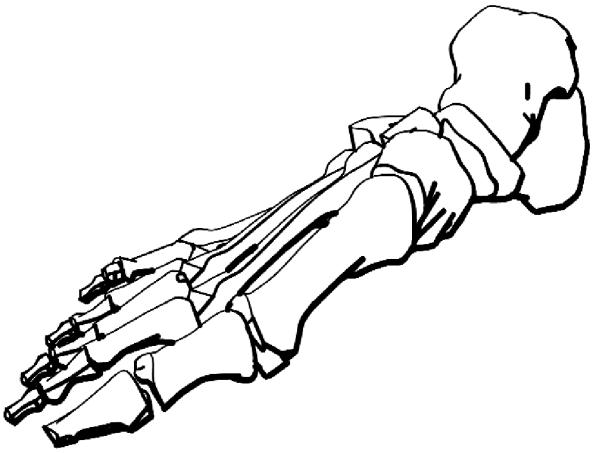


Abbildung 6.1: Die Abbildung zeigt ein Beispiel für die Hervorhebung der Lagebeziehung von Objekt und Untergrund (SCHUMANN, 1997).

Das G-Stroke-Konzept liefert somit die Basis für eine Fülle an Ideen und Möglichkeiten zur Informationskodierung. Natürlich können auch die bestehenden G-Strokes mit anderen Daten gefüllt oder für andere Stile genutzt werden. Das System ist somit sehr flexibel und breit gefächert in seinen Anwendungsmöglichkeiten.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

New York, NY, USA, 2000. ACM Press. ISBN 1-58113-277-8.

Band 22, New York, NY, USA, July 2003. ACM Press. Special Issue: Proceedings of ACM SIGGRAPH 2003.

ARTHUR APPEL. The Notion of Quantitive Invisibility and the Machine Rendering of Solids. In SOLOMON ROSENTHAL, Hrsg., *Proceedings of the 22nd ACM National Conference 1967 (Washington, D.C., USA)*, S. 387–393, New York, NY, USA, 1967. ACM Press. Library of Congress Catalog Number: 64-25615.

ARTHUR APPEL, F. JAMES ROHLF und ARTHUS J. STEIN. The Haloed Line Effect for Hidden Line Elimination. In TOM DEFANTI, BRUCE H. MCCORMICK, BARY W. POLLACK, NORMAN BADLER und S. H. CHASEN, Hrsg., *Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '79 (Chicago, IL, USA, August 8–10)*, Band 13(2), S. 151–157, New York, NY, USA, 1979. ACM SIGGRAPH, ACM Press. ISBN 0-89791-004-4.

TOM APPOLLONI, Hrsg. New York, NY, USA, 2002. ACM SIGGRAPH, ACM Press. ISBN 1-58113-521-1.

FOREST BASKETT, Hrsg. Band 24, New York, NY, USA, 1990. ACM SIGGRAPH, ACM Press. ISBN 0-201-50933-4.

BRUCE G. BAUMGART. Winged-Edge Polyhedron Representation. Technical report, Stanford University, Stanford, CA, USA, 1972.

F. BENICHOU und GERSHON ELBER. Output Sensitive Extraction of Silhouettes from Polygonal Geometry. In *Proceedings of the 7th Pacific Conference on Computer Graphics and Applications 1999 (Seoul, Korea, October 5–7, 1999)*, S. 60–69, Los Alamitos, CA, USA, 1999. IEEE Computer Society, IEEE Computer Society Press.

GRADY BOOCH. *Objektorientierte Analyse und Design. Mit praktischen Anwendungsbeispielen.* Addison Wesley Verlag, München, Germany, 1995. ISBN 3893196730.

JOHN W. BUCHANAN und MARIO C. SOUSA. The Edge Buffer: A Data Structure for easily Silhouette Rendering. In *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 5–7 2000)* NPA (2000), S. 39–42. ISBN 1-58113-277-8.

BALÁZS CSÉBFALVI, LUKAS MROZ, HELWIG HAUSER, ANDREAS KÖNIG und M. EDUARD GRÖLLER. Fast Visualization of Object Contours by Non-Photorealistic Volume Rendering. *Computer Graphics Forum*, 20(3), 2001. ISSN 1067-7055.

CASSIDY CURTIS. Loose and Sketchy Animation. In SCOTT CRISSON und JANET MCANDLESS, Hrsg., *ACM SIGGRAPH '98 Conference Abstracts and Applications*, S. 317, New York, NY, USA, 1998. ACM SIGGRAPH, ACM Press. ISBN 1-58113-046-5.

CASSIDY CURTIS. Non-Photorealistic Animation. In WAGGENSPACK (1999). ISBN 0-201-48560-5. Course 17: Non-Photorealistic Rendering.

DOUG DECARLO, ADAM FINKELSTEIN und SZYMON RUSINKIEWICZ. Interactive Rendering of Suggestive Contours with Temporal Coherence. In SPENCER (2004). ISBN 1-58113-887-3.

DOUG DECARLO, ADAM FINKELSTEIN, SZYMON RUSINKIEWICZ und ANTHONY SANTELLA. Suggestive Contours for Conveying Shape. In *ACM Transactions on Graphics* (ACM, 2003), S. 848–855. ISSN 0730-0301. Special Issue: Proceedings of ACM SIGGRAPH 2003.

DOUG DECARLO und ANTHONY SANTELLA. Stylization and Abstraction of Photographs. In APPOLLONI (2002), S. 769–776. ISBN 1-58113-521-1.

PHILIPPE DECAUDIN. Cartoon-Looking Rendering of 3D-Scenes. Research Report 2919, Institut National de Recherche en Informatique et en Automatique (INRIA), Le Chesnay Cedex, France, Juni 1996.

BERT DODSON. *Meisterschule Zeichnen: 48 Lektionen zum Selbststudium*. Weltbild Verlag GmbH Augsburg, 1993.

FENG DONG, GORDON J. CLAPWORTHY, HAI LIN und MELEAGROS A. KROKOS. Nonphotorealistic Rendering of Medical Volume Data. In *IEEE Computer Graphics and Applications* (IEE, 2003), S. 44–52.

DEBRA DOOLEY und MICHAEL F. COHEN. Automatic Illustration of 3D Geometric Models: Lines. In MICHAEL J. ZYDA, Hrsg., *Proceedings of the 1990 Symposium on Interactive 3D Graphics (Snowbird, UT, USA, March 1990)*, S. 77–82, New York, NY, USA, 1990a. ACM Press. ISBN 0-89791-351-5.

DEBRA DOOLEY und MICHAEL F. COHEN. Automatic Illustration of 3D Geometric Models: Surfaces. In ARIE E. KAUFMAN, Hrsg., *Proceedings of the IEEE Visualization 1990*, S. 307–314, Los Alamitos, CA, USA, 1990b. IEEE Computer Society, IEEE Computer Society Press.

FRÉDO DURAND. An Invitation to Discuss Computer Depiction. In *Proceedings of the Second International Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 3–5 2002)*, S. 111–124, New York, NY, USA, 2002. ACM Press. ISBN 1-58113-494-0.

MIKE EISSELE, DANIEL WEISKOPF und THOMAS ERTL. The G^2-Buffer Framework. In THOMAS SCHULZE, STEFAN SCHLECHTWEG und VOLKMAR HINTZ, Hrsg., *Simulation und Visualisierung 2004*, S. 287–298, Erlangen, San Diego, März 2004. SCS – Society for Computer Simulation Int., SCS European Publishing House.

GERSHON ELBER. Line Illustrations in Computer Graphics. *The Visual Computer*, 11: 290–296, 1995.

GERSHON ELBER und ELAINE COHEN. Hidden Curve Removal for Free Form Surfaces. In BASKETT (1990), S. 95–104. ISBN 0-201-50933-4.

ADAM FINKELSTEIN und LEE MARKOSIAN. Nonphotorealistic Rendering. In *IEEE Computer Graphics and Applications* (IEE, 2003), S. 26–27.

JAMES D. FOLEY, ANDRIES VAN DAM, STEVEN K. FEINER und JOHN F. HUGHES. *Computer Graphics – Principles and Practice*. Addison Wesley Publishing Company, München – Boston – San Francisco – Harlow, England – Don Mills, Ontario – Sydney – Mexico City – Madrid – Amsterdam, 2. Auflage, 1995. ISBN 0201848406.

ERICH GAMMA, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley Publishing Company, Reading, Massachusetts, 1995.

BRUCE E. GOLDSTEIN. *Wahrnehmungspsychologie: Eine Einführung*. Spektrum Akademischer Verlag, Heidelberg – Berlin – Oxford, 4. Auflage, 1997.

AMY GOOCH und BRUCE GOOCH. *Non-Photorealistic Rendering*. A. K. Peters Ltd., Nattick, MA, USA, 2001.

BRUCE GOOCH, PETER-PIKE J. SLOAN, AMY GOOCH, PETER SHIRLEY und RICHARD RIESENFELD. Interactive Technical Illustration. In ROSSIGNAC et al. (1999), S. 31–38. ISBN 1-58113-082-1.

LOUISE GORDON. *Portraitzeichnen anatomisch richtig*. Weltbild Verlag GmbH, Augsburg, 1989.

STÉPHANE GRABLI, FRÉDO DURAND, EMMANUEL TURQUIN und FRANÇOIS SILLION. A Procedural Approach to Style for NPR Line Drawing from 3D Models. Research Report 4724, Institut National de Recherche en Informatique et en Automatique (INRIA), Le Chesnay Cedex, France, February 2003.

STÉPHANE GRABLI, EMMANUEL TURQUIN, FRÉDO DURAND und FRANÇOIS X. SILLION. Programmable Style for NPR Line Drawing. In ALEXANDER KELLER und HENRIK WANN JENSEN, Hrsg., *15th EuroGraphics Symposium on Rendering (Norrköping, Sweden, June 21–23, 2004)*, S. 33–44, Oxford, UK, 2004. The Eurographics Association, Blackwell Publishers.

NICK HALPER, TOBIAS ISENBERG, FELIX RITTER, BERT FREUDENBERG, OSCAR MERUVIA, STEFAN SCHLECHTWEG und THOMAS STROTHOTTE. OPENNPAR: A System for Developing, Programming, and Designing Non-Photorealistic Animation and Rendering. In JON ROLNE, REINHARD KLEIN und WENPING WANG, Hrsg., *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications 2003 (Canmore, Alberta, Canada, October 8–10, 2003)*, S. 424–428, Los Alamitos, CA, USA, 2003a. IEEE Computer Society, IEEE Computer Society Press.

NICK HALPER, MARA MELLIN, CHRISTOPH S. HERRMANN, VOLKER LINNEWEBER und THOMAS STROTHOTTE. Psychology and Non-Photorealistic Rendering: The Beginning of a Beatiful Relationship. In JÜRGEN ZIEGLER und GERD SZWILLUS, Hrsg., *Mensch & Computer 2003: Interaktion in Bewegung (8.-10. September 2003, Stuttgart)*, S. 277–286, Stuttgart – Leipzig – Wiesbaden, 2003b. Teubner Verlag.

AARON HERTZMANN. Introduction to 3D Non-Photorealistic Rendering: Silhouettes and Outlines. In WAGGENSPACK (1999). ISBN 0-201-48560-5. Course 17: Non-Photorealistic Rendering.

AARON HERTZMANN und DENIS ZORIN. Illustrating Smooth Surfaces. In JUDITH R. BROWN und KURT AKELEY, Hrsg., *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '00 (New Orleans, LA, USA, July 23–28)*, S. 517–526, New York, NY, USA, 2000. ACM SIGGRAPH, ACM Press. ISBN 1-58113-208-5.

ELAINE R. S. HODGES, Hrsg. *The Guild Handbook of Scientific Illustration*. Van Nostrand Reinhold, New York, 1989.

Band 23, Los Alamitos, CA, USA, July/August 2003. IEEE Computer Society, IEEE Computer Society Press.

TOBIAS ISENBERG. Visualisierungseffekte in liniengraphischen Animationen und Stillbildern. Diplomarbeit, Institut für Simulation und Graphik, Otto-von-Guericke Universität Magdeburg, Magdeburg, Deutschland, 1999.

TOBIAS ISENBERG, BERT FREUDENBERG, NICK HALPER, STEFAN SCHLECHTWEG und THOMAS STROTHOTTE. A Developer's Guide to Silhouette Algorithms for Polygonal Models. In *IEEE Computer Graphics and Applications* (IEE, 2003), S. 28–37.

TOBIAS ISENBERG, NICK HALPER und THOMAS STROTHOTTE. Stylizing Silhouettes at Interactive Rates: From Silhouette Edges to Silhouette Strokes. In GEORGE DRETTAKIS und HANS-PETER SEIDEL, Hrsg., *Proceedings of EuroGraphics 2002 (Saarbrücken, Deutschland, September 2002)*, Band 21, S. 249–258, Oxford, UK, 2002. The Eurographics Association, Blackwell Publishers.

ROBERT D. KALNINS, PHILIP DAVIDSON, LEE MARKOSIAN und ADAM FINKELSTEIN. Coherent Stylized Silhouettes. In *ACM Transactions on Graphics* (ACM, 2003), S. 856–861. ISSN 0730-0301. Special Issue: Proceedings of ACM SIGGRAPH 2003.

ROBERT D. KALNINS, LEE MARKOSIAN, BARBARA J. MEIER, MICHAEL A. KOWALSKI, JOSEPH C. LEE, PHILIP L. DAVIDSON, MATHEW WEBB, JOHN HUGHES und ADAM FINKELSTEIN. WYSIWYG NPR: Drawing Strokes Directly on 3D Models. In APPOLLONI (2002), S. 755–762. ISBN 1-58113-521-1.

TOMIHISA KAMADA und SATORU KAWAI. An Enhanced Treatment of Hidden Lines. *ACM Transactions on Graphics*, 6(4):308–323, October 1987. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/35039.35042.

WALTER KOSCHATZKY. *Die Kunst der Zeichnung*. Residenz Verlag, Salzburg – Wien, 1993. ISBN 3-86146-118-8.

MICHAEL A. KOWALSKI, LEE MARKOSIAN, J. D. NORTHRUP, LUBOMIR BOURDEV, RONEN BARZEL, LORING S. HOLDEN und JOHN F. HUGHES. Art-Based Rendering of Fur, Grass, and Trees. In WAGGENSPACK (1999), S. 433–438. ISBN 0-201-48560-5.

JOHN LANSDOWN und SIMON SCHOFIELD. Expressive Rendering: A Review of Nonphotorealistic Rendering. *IEEE Computer Graphics and Applications*, 15(3):29–37, May 1995.

LEE MARKOSIAN, MICHAEL A. KOWALSKI, SAMUAL J. TRYCHIN, LUBOMIR D. BOURDEV, DANIEL GOLDSTEIN und JOHN F. HUGHES. Real-Time Nonphotorealsitc Rendering. In G. SCOTT OWEN, TURNER WHITTED und BARBARA MONES-HATTAL, Hrsg., *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '97 (Los Angeles, CA, USA, August 3–8)*, S. 415–420, New York, NY, USA, 1997. ACM SIGGRAPH, ACM Press. ISBN 0-89791-896-7.

DAVID MARR. *Vision: A Computational Investigation into the Human Representation and Processing of Visual Information*. W. H. Freeman and Company, New York, 1982.

MAIC MASUCH, STEFAN SCHLECHTWEG und RONNY SCHULZ. Speedlines: Depicting Motion in Motionless Pictures. In JODI GIROUX, ANNE RICHARDSON und JILL SMOLIN, Hrsg., *ACM SIGGRAPH '99 Conference Abstracts and Applications*, S. 277, New York, NY, USA, 1999. ACM SIGGRAPH, ACM Press. ISBN 1-58113-103-8.

SCOTT MCCLOUD. *Understanding Comics*. HarperCollins Publishers, Inc., New York, 1993.

MORGAN MCGUIRE und JOHN F. HUGHES. Hardware-Determined Feature Edges. In SPENCER (2004), S. 35–45. ISBN 1-58113-887-3.

BARBARA J. MEIER. Painterly Rendering. In JOHN FUJII, Hrsg., *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '96 (New Orleans, LA, USA, August 4–9)*, S. 477–484, New York, NY, USA, 1996. ACM SIGGRAPH, ACM Press. ISBN 0-89791-746-4.

SCOTT MEYERS. *Effektiv C++ Programmieren*. Addison Wesley Verlag, München, Germany, 3. Auflage, 1998. ISBN 3-8279-1305-8.

SCOTT MEYERS. *Mehr Effektiv C++ Programmieren*. Addison Wesley Verlag, München, Germany, 1. Auflage, 1999. ISBN 3-8273-1275-2.

J. D. NORTHRUP und LEE MARKOSIAN. Artistic Silhouettes: A Hybrid Approach. In *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (Annecy, France, June 5–7 2000)* NPA (2000), S. 31–37. ISBN 1-58113-277-8.

RAMESH RASKAR und MICHAEL COHEN. Image Precision Silhouette Edges. In ROSSIGNAC et al. (1999), S. 135–140. ISBN 1-58113-082-1.

JAREK ROSSIGNAC, JESSICA HODGINS und JAMES D. FOLEY, Hrsg. New York, NY, USA, 1999. ACM Press. ISBN 1-58113-082-1.

JAREK R. ROSSIGNAC und MAARTEN VAN EMMERIK. Hidden Contours on a Frame-Buffer. In *Proceedings of the 7th Workshop on Computer Graphics Hardware, EuroGraphics 1992 (Cambridge, UK, September 7–11, 1992)*, Oxford, UK, 1992. The Eurographics Association, Blackwell Publishers.

CHRISTIAN RÖSSL und LEIF KOBBELT. Line-Art Rendering of 3D-Models. In BRIAN A. BARSKY, YOSHIHISA SHINAGAWA und WENPING WANG, Hrsg., *Proceedings of the 8th Pacific Conference on Computer Graphics and Applications 2000 (Hong Kong, October 3–5, 2000)*, S. 87–96, Los Alamitos, CA, USA, 2000. IEEE Computer Society, IEEE Computer Society Press.

TAKAFUMI SAITO und TOKIICHIRO TAKAHASHI. Comprehensible Rendering of 3-D Shapes. In BASKETT (1990), S. 197–206. ISBN 0-201-50933-4.

BERT SCHÖNWÄLDER. Generierung charakteristischer Linienzüge aus 3D-Modellen. Diplomarbeit, Institut für Simulation und Graphik, Otto-von-Guericke Universität Magdeburg, Magdeburg, Deutschland, 1997.

LARS SCHUMANN. Ein parametrisierbares Modell zur Darstellung von Linien. Diplomarbeit, Institut für Simulation und Graphik, Otto-von-Guericke Universität Magdeburg, Magdeburg, Deutschland, 1997.

VOLKER SCHUMPELICK, NIELS M. BLEESE und ULRICH MOMMSEN. *Kurzlehrbuch Chirurgie*. Georg Thieme Verlag, 6. Auflage, September 2003. ISBN 3131271264.

MARIO COSTA SOUSA und PRZEMYSLAW PRUSINKIEWICZ. A Few Good Lines: Suggestive Drawing of 3D Models. In PERE BRUNET und DIETER W. FELLNER, Hrsg., *Proceedings of EuroGraphics 2003 (Granada, Spain, September 2003)*, Band 22, S. 381–390, Oxford, UK, 2003. The Eurographics Association, Blackwell Publishers.

STEPHEN N. SPENCER, Hrsg. New York, NY, USA, 2004. ACM Press. ISBN 1-58113-887-3.

THOMAS STROTHOTTE, BERNHARD PREIM, ANDREAS RAAB, JUTTA SCHUMANN und DAVID R. FORSEY. How to Render Frames and Influence People. In M. DAEHLEN und

L. KJELLDAHL, Hrsg., *Proceedings of EuroGraphics 1994 (Oslo, Norwegen, September 1994)*, Band 13, S. 455–466, Oxford, UK, 1994. The Eurographics Association, Blackwell Publishers.

THOMAS STROTHOTTE und STEFAN SCHLECHTWEG. *Non-Photorealistic Computer Graphics. Modeling, Rendering, and Animation*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 2002. ISBN 1-55860-787-0.

BJARNE STROUSTRUP. *Die C++ Programmiersprache*. Addison Wesley Verlag, München, Germany, 4. Auflage, 2000. ISBN 3-827-31660-X.

IVAN EDWARD SUTHERLAND. Sketchpad: A Man-Machine Graphical Communication System. In *AFIPS Conference Proceedings 23*, S. 323–328, 1963.

HERB SUTTER. *Exceptional C++*. Addison Wesley Verlag, München, Germany, 2000.

CHRISTIAN TIETJEN. Evaluation und Modifikation von Methoden zur Generierung von Liniengraphiken in der medizinischen Visualisierung. Diplomarbeit, Institut für Simulation und Graphik, Otto-von-Guericke Universität Magdeburg, Magdeburg, Deutschland, 2004.

WARREN WAGGENSPACK, Hrsg. New York, NY, USA, 1999. ACM SIGGRAPH, ACM Press. ISBN 0-201-48560-5.

ERNST A. WEBER. *Sehen, Gestalten und Fotografieren*. Birkhäuser, Basel – Boston – Berlin, 1990.

JOSIE WERNECKE. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*, Band 1. Addison Wesley Publishing Company, Amsterdam, 1994a.

JOSIE WERNECKE. *The Inventor Toolmaker: Extending Open Inventor, Release 2*, Band 1. Addison Wesley Publishing Company, München – Boston – San Francisco – Harlow, England – Don Mills, Ontario – Sydney – Mexico City – Madrid – Amsterdam, 1994b.

HYPERDICTIONARY WORDNET. Webnox Corporation. Online-Veröffentlichung, 2000–2003. Zugriff: Mittwoch, 28.04.2004, 13:03 Uhr, `http://www.hyperdictionary.com`.

Anhang

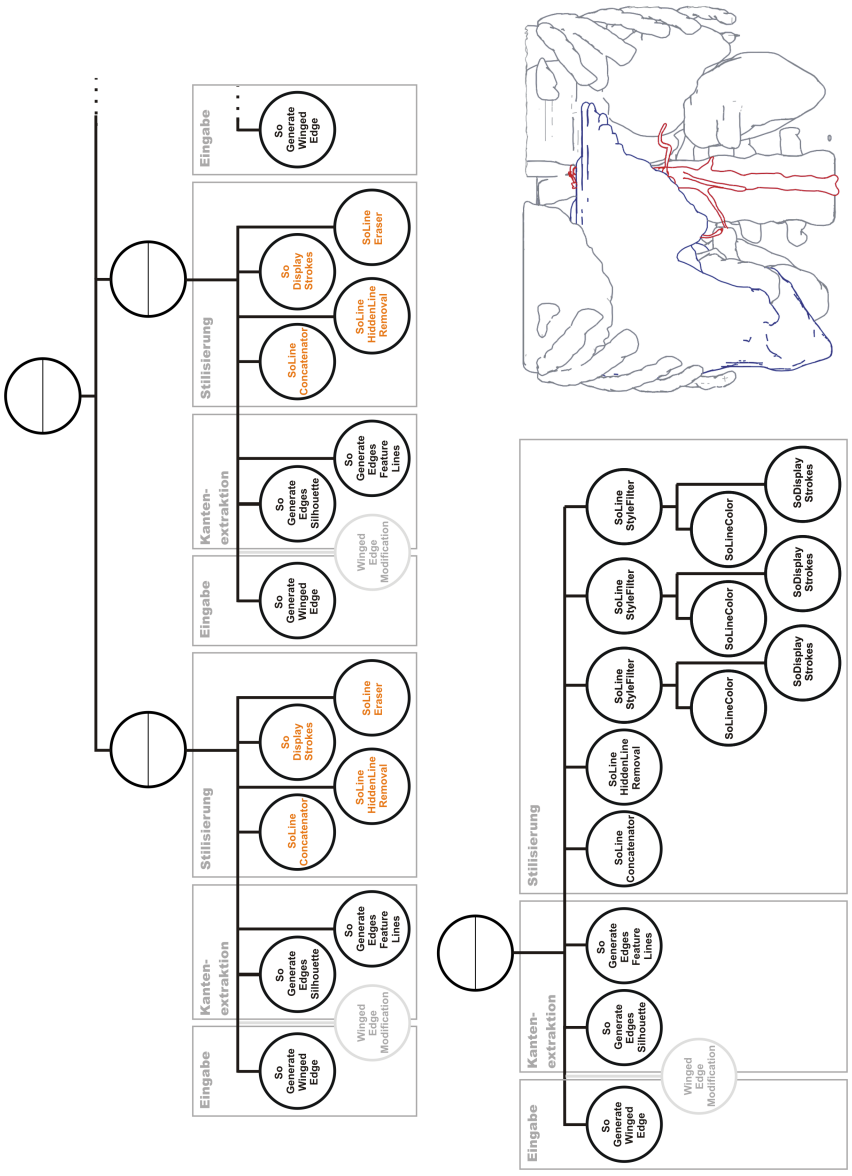


Abbildung 6.2: Szenengraph-Vergleich eines bisher notwendigen Aufbaus (oben) und eines aktuell möglichen Aufbaus (unten), um die unterschiedliche Darstellung verschiedener Objekte (unten rechts) zu erreichen.