1. CAD - Computerintegrierte Projektbearbeitung im Bauwesen

1.1 Rechnergestützte Zeichnungserstellung

Aus vielen Bereichen des Ingenieurwesens ist heute das rechnergestützte Entwerfen und Konstruieren (*CAD=computer aided design*) nicht mehr wegzudenken.

Der Begriff *CAD* ist allerdings recht unscharf. Zunächst hat man unter dem Begriff CAD die *rechnergestützte Zeichnungserstellung* verstanden: Das traditionelle Zeichnen mit Stift und Papier ist dabei durch das Zeichnen mit Digitalisierstift, Digitalisierbrett und Bildschirm ersetzt worden. Produkt eines solchen Programmes ist eine elektronisch gespeicherte *Zeichnung*, also ein *zweidimensionales* Abbild der dreidimensionalen Welt. Zu den wichtigsten Funktionen eines Zeichnungserstellungssystems gehören:

- Unterstützung beim Zeichnen durch *Konstruktionshilfen*. Solche Hilfen sind zum Beispiel Ausrichtung an einem Raster, "Einfangen" bereits gezeichneter Punkte oder Objekte, automatische Erkennung von End- und Mittelpunkten oder geometrischen Zusammenhängen wie z.B. Lotfußpunkten, Tangenten an Kreise, rechten Winkeln, usw.
- Unterstützung beim Konstruieren gleichartiger Objekte durch *Kopier-*, *Einfüge-* und *Verzerrungsoperationen.*
- Unterstützung für die Konstruktion häufig wiederkehrender gleichartiger Konstruktionselemente durch Aufzeichnung von *Makros* (sprachliche Beschreibung von Konstruktionsvorgängen), de dann beliebig oft wieder abgespielt werden können. Das Aufzeichnen von Konstruktionsvorgängen ist auch Voraussetzung für *Undo*-Funktionalität.
- Makros allein helfen jedoch wenig, wenn sie nur als starres, unverändertes Programm abgespult werden können. Zum Beispiel will man ein Konstruktionselement mit Hilfe eines Makros an beliebiger Stelle oder in beliebiger Größe einfügen. Die Makros müssen also *parametrisierbar* sein. Man spricht dann von *Variantenkonstruktion*.
- Selbstverständlich sind heute auch *automatische, assoziative Vermaßung* und automatische Beschriftung.
- Zur Organisation einer Zeichnung ist es wichtig, die Inhalte strukturiert abzuspeichern und zu bearbeiten. Das heißt, Zeichnungsinhalte müssen thematisch gruppiert und zu Zeichnungen (Plänen) kombiniert werden können. Dies wird erreicht durch die sogenannte *Folientechnik.*
- Zu der Funktionalität eines zeichnungserstellenden Programmes gehört überdies das Anordnen von Teilzeichnungen auf einem Blatt, eventuell einschließlich automatischer Plazierung und Wahl geeigneter, normierter Maßstäbe, das Anbringen von Planköpfen, usw.

1.2 Rechnergestützte geometrische Modellierung

Heutige *CAD*-Systeme im engeren Sinn zielen über die reine *Zeichnungserstellung* hinaus auf die *geometrische Modellierung* von Objekten des Ingenieurwesens. Das bedeutet, daß in solchen Systemen zunächst nicht *Zeichnungen*, sondern *geometrische Modelle* bearbeitet werden. Objekte des Ingenieurwesens sind in der Regel dreidimensionale Körper. In einem *modellzentrierten* System arbeitet man also stets an einem *dreidimensionalen geometrischen Modell* eines Objektes, nicht an einem Plan.

1.3 Rechnergestützte Modellierung eines Produktes

1.4 Rechnergestützte Projektbearbeitung

Kommunikation, Informationsfluß, Dokumentation.

Heute wird vom rechnergestützten Konstruieren allerdings ein Mehrwert über die Erleichterung des Erstellens von Zeichnungen hinaus erwartet. Dafür stehen Begriffe wie *CAE* (*computer aided engineering*, rechnergestütztes Ingenieurwesen) und *CIM* (*computer integrated manufacturing*, rechnerintegrierte Fertigung).

Die Zeichnungserstellung wird nur noch als kleiner Teil einer *Prozeßkette* angesehen, und Ziel ist es, die gesamte Prozeßkette, die zur Planung, Konstruktion, Fertigung und Verwaltung eines technischen Produktes erforderlich ist, durchgängig auf dem Rechner zu erfassen und alle zugehörigen Daten einheitlich zu verwalten.

Damit wird CAD zu einer einzelnen Facette einer umfassenden Beschreibung und Simulation eines Produktes – im Bauwesen eines Bauwerks – auf dem Rechner: Nicht nur die geometrische Gestalt des Bauwerks, sondern auch sein Tragverhalten, die zu seiner Fertigung notwendigen Vorgänge und die Kosten des Bauwerks sowie die Nutzungsformen sollen im Rechner beschrieben werden. Das Bauwerk in allen seinen Facetten soll also umfassend auf dem Rechner *modelliert* werden (Bild 1). Eine wichtige Rolle spielt dabei die Beschreibung der dreidimensionalen Geometrie des Bauwerks. Dem CAD kommt hierbei die Aufgabe zu, eine graphisch-interaktive Benutzungsoberfläche bereitzustellen, mit Hilfe derer der Benutzer die geometrische Gestalt des Bauwerksmodells eingeben und modifizieren kann. Die Zeichnungserstellung tritt demgegenüber zurück; Ansichten und Schnitte können automatisch erstellt werden, wenn ein voll dreidimensionales Bauwerksmodell vorliegt.

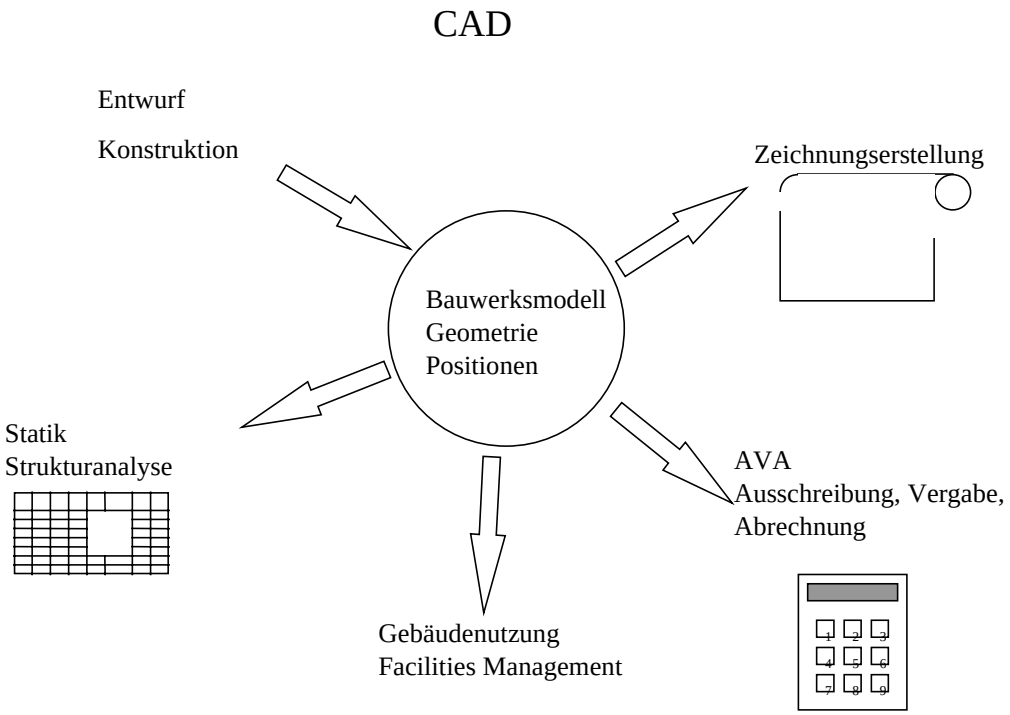

Bild 1: Im Zentrum der Arbeit unterschiedlichster Nutzer steht im modernen CAD ein Gebäudemodell. Die Zeichnungserstellung ist nur noch einer von vielen Nutzern.

Im Gegensatz zur stationären Industrie der Massenprodukte (z.B. Maschinenbau) fertigt das Bauwesen fast ausschließlich *Unikate*, also einmalige, in ihrer Gestalt völlig verschiedene Produkte. Auch für die einzelnen Bauteile, aus denen Bauwerke sich zusammensetzen, kann der Bearbeiter aus einer fast unüberschaubaren Formenvielfalt wählen, und unterschiedlichste Materialien, vorgefertigte Bauteile, Anschlußtypen und Verbindungen und Fertigungsarten können eingesetzt werden. Überdies sind im Bauwesen außerordentlich viele Beteiligte mit unterschiedlichen Interessen (Bauherr, Behörden, Baufirma, Konstruktionsbüro usw.) in das Projekt einbezogen. Nicht zuletzt ist das Bauwesen in der Praxis von der euphemistisch so genannten "baubegleitenden Planung" charakterisiert, das heißt, während der Laufzeit eines Projektes, sogar während der Ausführungsphase, werden grundlegende Änderungen am ursprünglichen Konzept vorgenommen, die nachträglich in die bestehende Planung und Ausführung integriert werden müssen. Alle diese Aspekte haben dazu beigetragen, im Bauwesen die durchgängig rechnerunterstützte Projektbearbeitung zu erschweren.

Auch bei optimistischer Einschätzung muß man wohl annehmen, daß es im Bauwesen sicherlich immer ein Traum bleiben wird, einen durchgängigen, integrierten "Datenfluß" vom Architektenplan bis zur Fertigung zu erreichen. Insbesondere ist die Vorstellung abwegig, den gesamten Prozeß *vollautomatisch* ablaufen zu lassen, etwa nach dem Motto: Sobald der Architekt seinen Plan fertiggestellt hat, reicht ein Knopfdruck, um statische Berichte, Bewehrungs- und Installationspläne und Bauzeitenplan automatisch zu erstellen.

Möglich und machbar ist aber eine weitgehende Unterstützung von Routinetätigkeiten durch rechnerbasierte Werkzeuge. Unterstützung bedeutet dabei eine automatische Generierung von *Lösungsvorschlägen*, die dann von einem fachkundigen Bearbeiter kontrolliert, modifiziert und ergänzt werden. Automatisch können aber immer nur kleine Teilschritte des Gesamtprozesses bearbeitet werden.

Die vorliegende Vorlesung beschäftigt sich mit der Auswahl rechnergestützter Werkzeuge bei der Bearbeitung von Projekten des konstruktiven Ingenieurbaus. Schon heute wird kaum ein Projekt dieses

Bereichs ohne den Einsatz von rechnergestützten Hilfsmitteln bearbeitet. Um für jedes Teilproblem bzw. jeden Teilschritt eines Projektes das adäquate Hilfsmittel zu finden und das betreffende Hilfsmittel optimal einzusetzen, sind Grundkenntnisse über theoretische Grundlagen und Funktionalität des Werkzeugs notwendig. Fast jeder einzelne Teilprozeß der Bauplanung und Fertigung läßt eine Teilautomation zu.

Computergestützte Werkzeuge sollten aber nicht nur individuell und unabhängig voneinander als Insellösungen eingesetzt werden, sondern auf miteinander verknüpft werden können. In der vorliegenden Vorlesung wird die "Integration verschiedener Werkzeuge" nicht auf das Problem der Übertragung von *Daten*, etwa in Form standardisierter *Austauschdateien*, reduziert. Ein solcher Ansatz wäre schon deswegen unzureichend, weil dabei notwendigerweise ein "Nacheinander" von Prozessen (*Preprocessing-Processing-Postprocessing*) gefordert wird und keine aktive "Interaktion" möglich ist. Standardformate und Datenaustausch durch Schnittstellen spielen in der vorliegenden Vorlesung eine untergeordnete Rolle.

Vielmehr wollen wir uns intensiv mit den Möglichkeiten zur aktiven *Kommunikation* zwischen verschiedenen Werkzeugen beschäftigen. Dabei geht es nicht nur um einen reinen Datenaustausch, sondern auch um die gegenseitige "Fernsteuerung" zwischen verschiedenen Programmen. Jedes Programm wird als "Dienstleister" angesehen, der einen eigenen Datenbestand verwaltet und *Operationen* mit diesem Datenbestand als Dienste für andere Programme bereitstellt.

Einer dieser Dienstleister hat die Aufgabe, das zentrale *Modell* unseres Projektes zu verwalten (Bild 1). Das zentrale Modell umfaßt nicht nur geometrische Informationen, sondern dient allen Phasen des Bearbeitungsprozesses als zentrale Informationsquelle. Für die Verwaltung großer Datenbestände haben sich als Werkzeug *Datenbankmanagementsysteme* durchgesetzt. Datenbanksysteme gestatten den wahlfreien Zugriff auf die Daten, den gleichzeitigen Zugriff verschiedener Benutzer, und die Festlegung von Zugriffsrechten. Wir werden in der vorliegenden Vorlesung Ideen zu einem "*Bauwerksmodell*" entwickeln, das von einem Datenbankmanagementsystem verwaltet wird und allen Teilprozessen des Bauplanungsprozesses zur Nutzung zur Verfügung steht. An dieses Datenbanksystem wollen wir von außen Abfragen und Befehle zur Modifikation und Ergänzung der Daten richten. Dabei benötigen wir Grundkenntnisse der Sprache SQL (*structured query language*). Dies ist eine standardisierte Sprache zur Kommunikation mit Datenbankverwaltungssystemen.

Mit den übrigen Werkzeugen, die wir im Rahmen der vorliegenden Vorlesung einsetzen werden, wollen wir mit Hilfe der Programmiersprache VBA (*Visual Basic for Applications*) kommunizieren. Diese microsoftspezifische Sprache erlaubt die Fernsteuerung anderer Programme durch "Makros", also kleine, vom Anwender selbst entwickelte Programme zur Erweiterung der Funktionalität von Standardprogrammen wie Textverarbeitung (Microsoft Word) oder Tabellenkalkulation (Microsoft Excel). Neben diesen Standard-Büroprogrammen wollen wir einige Spezialprogramme für das computergestützte Konstruieren einsetzen, die durch VBA ferngesteuert werden können.

Sehen wir uns den prinzipiellen Ablauf einer Projektbearbeitung im konstruktiven Ingenieurbau an, um einen Überblick über die verfügbaren Hilfsmittel und die notwendige Ergänzung durch eigene Makros zu gewinnen.

Am Anfang steht der *Entwurf*. Ziel des ersten Entwurfes ist es, einen ersten Vorschlag für die topologische Struktur des Bauwerks festzulegen und einen groben *Rahmen* für die geometrische Gestalt des Bauwerks festzulegen. Die wesentlichsten dabei zu berücksichtigenden Einflußfaktoren sind die Zweckbestimmung und Nutzung des Bauwerks und die Einpassung des Bauwerks in seine Umgebung, also *funktionale* Gesichtspunkte sowie *ästhetische* Kriterien. Am Ende des ersten Entwurfs

steht als Ergebnis ein erstes, sicher noch zu modifizierendes geometrisches Modell des Bauwerks. Als Werkzeug zur Bearbeitung des Entwurfsmodells steht uns CAD zur Verfügung. In der Entwurfsphase ist besonders wichtig, daß das eingesetzte System gute Möglichkeiten zur *Visualisierung* bietet, da das Entwurfsmodell auch für die Diskussion des Bauvorhabens mit Bauherr und Öffentlichkeit eingesetzt wird. Visualisierungsmethoden sind daher auch ein Gegenstand der vorliegenden Vorlesung.

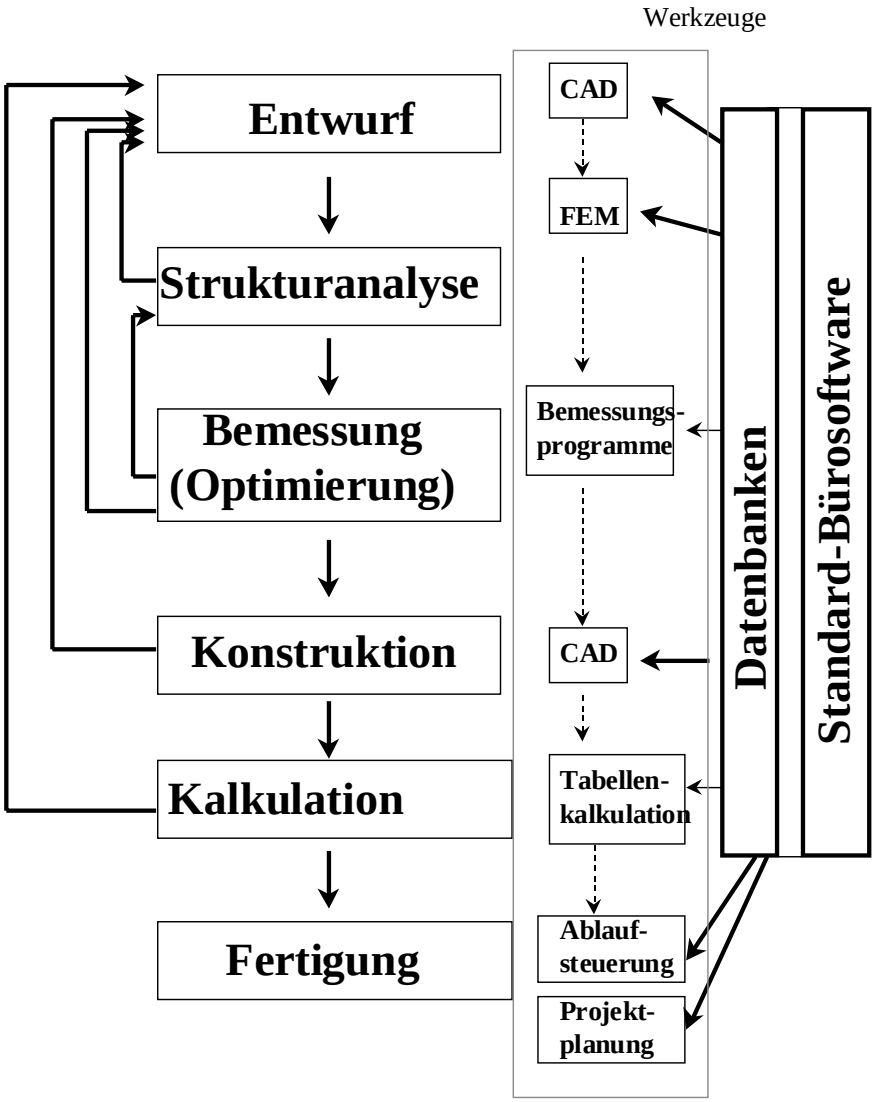

Bild 2: Teilprozesse und Werkzeuge der Bauplanung

Sobald der Entwurf vorliegt, setzt die *Tragwerksplanung* ein. Dazu muß ein *Tragwerksmodell* erzeugt werden, das den Vorgaben des Architekturentwurfs weitgehend gerecht wird. In der Regel sind dabei geometrische Vereinfachungen der Bauwerksgeometrie (z.B. Idealisierung dreidimensionaler Tragwerkselemente als eindimensionaler Balken oder zweidimensionale Platte/Scheibe) erforderlich. Die Konstruktion der *Systemgeometrie des Tragwerks* kann wie die Eingabe der Architekturgeometrie in einem CAD-System erfolgen. Liegt die Architekturgeometrie bereits vor, kann die Konstruktion von Systemachsen u. dgl. durch Automation innerhalb des CAD-Systems unterstützt werden. Da diese Funktionalität aber nicht zu den Standardfunktionen eines CAD-Systems gehört, benötigen wir ein *offenes System*, das dem Benutzer die Entwicklung und Einbindung selbst entwickelter Funktionen und den Zugriff auf das geometrische Modell des CAD-Systems gestattet.

Wir setzen in der vorliegenden Vorlesung AutoCAD R. 14 als CAD-System ein, da dieses System offen ist und mit der Sprache VBA programmiert und erweitert werden kann. Überdies bietet AutoCAD gute Visualisierungsmöglichkeiten.

Liegt die Systemgeometrie des Tragwerks vor, so kann noch lange nicht die statische Berechnung erfolgen, sondern es sind weitere Zusatzeingaben notwendig: Insbesondere wird das Bauwerk in den seltensten Fällen als ganzes gerechnet, sondern stets in einzelne *Positionen* zerlegt. Die gegenseitigen Auswirkungen der Tragwerkselemente aufeinander werden dabei idealisiert. Das zugehörige Modell des Bauwerks wird in einem sogenannten "*Lastabtragsplan*" beschrieben.

Für die Konstruktion des Lastabtragsplans liegen heute keine brauchbaren Werkzeuge vor; wir werden daher in der vorliegenden Vorlesung für diesen Teil der Projektbearbeitung im Bauwesen eigene "Notlösungen" entwickeln müssen. Der Lastabtragsplan schließt auch die Beschreibung der *äußeren Einwirkungen* auf ein Bauwerk sowie die Regeln zur Ermittlung und Überlagerung von *Lastfällen* mit ein.

Die Beschreibung unseres Bauwerks als Menge untereinander verbundener *Positionen* legen wir in einer *Datenbank* nieder. Wir verwenden hier exemplarisch das Datenbanksystem Microsoft Access, mit der wir über SQL kommunizieren. Da SQL eine standardisierte Datenbanksprache ist, könnte man jede beliebige andere relationale Datenbank anstelle von Access substituieren.

Die eigentliche *Berechnung* des Tragwerks (Schnittgrößenermittlung) erfolgt in aller Regel mit einem diskretisierenden *Berechnungsverfahren*. Beispiele sind FEM (*Finite-Elemente-Methode*) und BEM (*boundary element method*, Randintegralelementmethode). Als Eingangsdaten für die Berechnung benötigen wir Systemgeometrie und Lasteinwirkungen der jeweils zu untersuchenden Positionen sowie eine Beschreibung der Lagerungsbedingungen. Alle Eingangsdaten der Finite-Elemente-Berechnung wollen wir anhand einer graphischen Darstellung des Systems kontrollieren, ergänzen und modifizieren. Da wir auch die FE-Analyse durch VBA steuern wollen, kommen in diesem Schritt nur die von unserem Fachgebiet selbst entwickelten FE-Programme als Werkzeug in Betracht, da diese wie AutoCAD R. 14 durch VBA ferngelenkt werden können und eine weitgehend automatisierte FE-Analyse ermöglichen.

An die Strukturanalyse schließt sich die *Bemessung* an. Darunter ist die Dimensionierung der Tragwerkselemente unter Berücksichtigung der geltenden Normen zu verstehen. Ziel der Bemessung ist die Sicherung der Standsicherheit des Bauwerks auch in extremen Situationen und der Gebrauchsfähigkeit unter Gebrauchslast. Die Bemessung können wir in der vorliegenden Vorlesung nur am Rande behandeln. Wir verallgemeinern die Bemessungsaufgabe hier zu einer allgemeinen *Optimierungsaufgabe*, nämlich der, ein möglichst kostengünstiges Tragwerk zu finden, innerhalb dessen nirgends die zulässigen Höchstspannungen des Materials überschritten werden. Dieses Optimierungsproblem lösen wir mit einem kleinen VBA-Makro.

Sobald eine befriedigende Lösung der Bemessungsaufgabe gefunden ist, kann als nächste Aufgabe die *Konstruktion* des Tragwerks angegangen werden. Unter Konstruktion ist hierbei insbesondere die Durchbildung der erforderlichen *Anschlüsse* zwischen den einzelnen Tragwerkselementen zu verstehen. Auch hier können wir nur exemplarisch eine kleine Lösung für einen begrenzten Teilbereich untersuchen, nämlich die Auswahl *typisierter Verbindungen* für Bauelemente im Stahlbau. Hier liegt eine Auswahl vorgegebener Anschlüsse vor, aus der anhand der bemessungsrelevanten Schnittgrößen eine passende Lösung ausgewählt werden kann.

Schließlich wird ein Entwurf abschließend *beurteilt*. Kriterien zur abschließenden Beurteilung sind *ökonomische*, im öffentlichen Leben aber oft auch *ästhetische*. Um die abschließende Beurteilung eines Entwurfs zu ermöglichen, ist also eine *Kostenkalkulation* und wiederum eine graphische *Präsentation* erforderlich. Wir wollen uns hier auf eine grobe Ermittlung der *Angebotssumme* anhand der Massen beschränken. Als Werkzeug dafür verwenden wir Tabellenkalkulation. Zur graphischen Präsentation des endgültigen Bauwerks setzen wir AutoCAD ein.

Schließlich wollen wir noch einen groben *Bauablaufplan* als ersten Schritt zur Unterstützung der *Fertigung* erzeugen. Dabei greifen wir auf ein von unserer Arbeitsgruppe entwickeltes, durch VBA fernsteuerbares Programm zur *Graphentheorie* zurück.

Die vorliegende Vorlesung kann unmöglich alle genannten Bereiche erschöpfend behandeln; es ist auch nicht Ziel der Vorlesung, "Patentrezepte" für diese außerordentlich komplexe Thematik anzubieten. Vielmehr sollen mögliche Lösungsansätze aufgezeigt, diskutiert und exemplarisch realisiert werden. Die Teilnehmer der Vorlesung sollen in eigenen Studienarbeiten jeweils einen Teilaspekt des Gesamtprozesses vertieft untersuchen.

Wir haben in der Einführung CAD-Systeme als Werkzeuge zur Visualisierung und interaktiven Bearbeitung geometrischer Modelle charakterisiert. Im ersten Teil der vorliegenden Vorlesung steht der Einsatz des CAD-Systems AutoCAD R. 14 im Vordergrund. Parallel zu den praktischen Übungen mit diesem System wollen wir einen Überblick über die wichtigsten Grundlagen von CAD-Programmen gewinnen.

Dabei sind von Anfang an die drei Komponenten Interaktion (Mensch-Maschine-Interaktion), Visualisierung (Methoden der Computergraphik) und Modellierung (Beschreibung geometrischer Objekte durch Datenstrukturen im Rechner) zu unterscheiden (Bild 3).

CAD

Modellierung	Visualisierung	Interaktion
topologische Modellierung Graphentheorie computational geometry solid modeling	Computer-Graphik Projektion und Perspektive	MMI = Mensch - Maschine - Interaktion GUI = graphical user interface ereignisgesteuerte Programme fensterorientierte Systeme
Attribute		

Bild 3: Dreiteilung der Aufgaben eines modernen CAD-Systems: Modellierung, Visualisierung und Mensch-Maschine-Interaktion.

Der Benutzer eines CAD-Systems ist zunächst mit der *Bedienung* des Programms und somit mit der *Hardware* (Gerätetechnik) und dem *GUI* (*graphical user interface*, *Benutzungsoberfläche*) konfrontiert. Auch wir wollen uns zunächst diesen Themen zuwenden. Die zugehörigen Inhalte gelten

auch für andere graphisch-interaktive Programme, und im weiteren Verlauf der Vorlesung greifen wir auch bei der Entwicklung eigener Programme auf einige dieser Grundlagen zurück.

Der Themenkreis *Visualisierung* bezieht sich schon etwas stärker auf die Darstellung *geometrischer Objekte* und damit auf den eigentlichen Kern eines CAD-Systems; jedoch ist auch zur sinnvollen Anwendung von CAD-Programmen eine Grundkenntnis der Koordinatensysteme, Transformationen, Projektionen und Perspektiven sowie photorealistischer Darstellungstechniken hilfreich.

Im Themenbereich *Modellierung* schließlich dringen wir in den innersten Kern eines CAD-Systems ein, mit dem normalerweise nur Entwickler und Programmierer direkt konfrontiert werden. Da wir in der vorliegenden Lehrveranstaltung jedoch nicht nur mit fertigen CAD-Systemen umgehen wollen, sondern auch eigene Erweiterungen und Verklammerungsfunktionen mit anderen Systemen entwickeln wollen, benötigen wir auch auf diesem Gebiet einige elementare Grundkenntnisse.

Man muß sich bewußt sein, daß es *die einheitliche Geometrie* eines Bauwerks *nicht* gibt. Während für den Architekten die Wände, Stützen und Decken häufig nur als Begrenzung der *Räume* interessant sind, stehen für den Konstrukteur gerade die Tragelemente im Vordergrund der geometrischen Modellierung, und mit einem Raummodell kann er wenig anfangen. Der Statiker wiederum ist meist an einer *idealisierten*, meist dimensionsreduzierten Geometrie interessiert. Eine Stütze beispielsweise stellt sich für den Konstrukteur als dreidimensionaler, prismatischer Körper dar, während der Statiker die Stütze vorzugsweise zu einer linienförmigen Geometrie (Längsachse der Stütze) idealisiert.

Es werden also bei einer integrierten computerbasierten Projektbearbeitung verschiedene geometrische Modelle derselben Struktur nebeneinander existieren. Um die Integration der Projektbearbeitung erreichen zu können, müssen Techniken bereitstehen, um verschiedenartige Modelle ineinander zu überführen oder Abhängigkeiten zwischen verschiedenartigen Modellen formulieren zu können.

2. Modellierung

2.1. Geometrische Modellierung

Der Begriff CAD (computer aided design, rechnergestütztes Konstruieren) ist irreführend; CAD-Programme dienen in erster Linie der interaktiven Erstellung und Bearbeitung sowie der Visualisierung der *Bauwerksgeometrie*. Somit stellt eine rechnerorientierte Beschreibung *geometrischer Objekte* die Kernkomponente eines jeden CAD-Systems dar. Heute kann eine *dreidimensionale* Modellierung von Bauobjekten als Standard betrachtet werden.

Es ist alles andere als trivial, ein geometrisches Gebilde durch Datenstrukturen in einem digitalen Rechner abzubilden. Wir stellen im folgenden mehrere gängige Techniken zur Abbildung dreidimensionaler geometrischer Gebilde durch Datenstrukturen vor.

Die Entscheidung für eines dieser rechnergestützten Geometriemodelle wird entscheidend durch den intendierten Verwendungszweck des Modells mitbestimmt. Eine Datenstruktur, die geeignet ist, eine Visualisierung auf einem rasterorientierten Ausgabegerät zu erzeugen, wird nicht notwendigerweise auch die Anforderungen des Konstrukteurs erfüllen können, der mit einfachen geometrischen Grundobjekten wie Prismen und Quadern zu arbeiten gewöhnt ist. Nicht jede Datenstruktur wird gleich gut geeignet sein, den Volumeninhalt des dargestellten geometrischen Objektes zu ermitteln, und nicht jede Datenstruktur wird gleich guten Zugriff auf Oberflächeninformationen wie Flächennormalen oder Krümmungen der Oberfläche bieten.

Im folgenden wird ein Überblick über die wichtigsten Arten von Datenstrukturen zur geometrischen Modellierung gegeben:

- *Generative Modelle*: Dies sind Modelle, die auf die Beschreibung von "Sweep"-Operationen aufbauen, die also geometrische Gebilde höherer Dimensionalität durch Beschreibung der Verschiebung eines Objektes niedrigerer Dimensionalität längs einer erzeugenden Kurve definieren.
- *Zellzerlegungsmodelle*: In Zellzerlegungsmodellen werden beliebige geometrische Gebilde näherungsweise aus kleinen, einfachen Grundkörper, z.B. Würfeln, zusammengesetzt.
- *Modelle auf Basis parametrisierter Grundkörper*: In solchen Modellen stehen im Gegensatz zu den elementaren Grundkörpern der Zellzerlegungsmodelle komplexe, durch Parameter beschriebene Grundkörper zur Verfügung, aus denen wie aus Bausteinen das geometrische Modell zusammengesetzt werden kann. Die Parametrisierung beschreibt dabei Gestalt und räumliche Anordnung der Grundkörper.
- *CSG-Modelle*: CSG-Modelle sind keine grundständigen Modelle, sondern erweitern die Möglichkeiten zur Erzeugung komplexer geometrischer Gebilde aus einfacheren Grundkörpern. An die Stelle des bloßen Aneinanderfügens der Grundelemente tritt in CSG-Modellen die Anwendung von Mengenoperationen wie Vereinigung, Schnitt, Differenz.
- *Boundary Representation*: In Boundary-Representation-Modellen (B-Rep) werden geometrische Gebilde durch Beschreibung ihrer Berandung beschrieben.

Da ein CAD-Programm oft mehreren Anwendungszwecken gleichzeitig dient, ist oftmals die Überführung eines gewählten Modells in eine andere Darstellung erforderlich; auch diese Problematik wird angesprochen.

2.2 Generative Modelle

Eine naheliegende Möglichkeit, dreidimensionale Körper zu beschreiben, besteht darin, ihre "Entstehungsgeschichte" aufzuzeichnen. Durch Wiederholung des Konstruktionsvorganges läßt sich der einmal erzeugte Körper beliebig oft reproduzieren.

Modelle, die auf der Idee der Aufzeichnung der Entstehungsgeschichte beruhen, werden als *generative* Modelle bezeichnet. Im Extremfall geht man von der Vorstellung aus, daß ein dreidimensionaler Körper ausschließlich schrittweise durch "Aufblasen" geometrischer Objekte kleinerer Dimensionalität entsteht.

Im ersten Schritt erzeugt man beispielsweise einen *Punkt* im Raum. Verschiebt man diesen Punkt längs einer Erzeugenden im Raum, so bilden alle dabei überstrichenen Raumpunkte zusammen ein geometrisches Gebilde einer höheren Dimensionalität, nämlich eine Kurve, im einfachsten Fall eine geradlinige Strecke. Durch Verschieben dieser Kurve längs einer weiteren Erzeugenden kann man nun eine Fläche, also ein zweidimensionales geometrisches Gebilde, erzeugen. Die Fläche schließlich beschreibt durch die Menge aller überstrichenen Punkte beim Verschieben im Raum längs einer Erzeugenden einen räumlichen Körper.

Man kann versuchen, diese Idee streng von "0-dimensionalen" bis zu "3-dimensionalen" Objekten durchzuhalten und somit ein räumliches Objekt ausschließlich durch punktförmige "Samen" und eine Reihe von Erzeugungsvorschriften zu beschreiben. Der Konstruktionsvorgang, der dabei erforderlich wird, läuft vielleicht der Intuition zuweilen etwas zuwider, und man kann zunächst auch nur *konvexe* Flächen und Körper erzeugen. Allerdings kann man – was in anderen Modellen Schwierigkeiten bereitet – in einem solchen Modell beliebig ein-, zwei- und dreidimensionale Elemente mischen und auch sogenannte *Nichtmannigfaltigkeiten* (siehe unten bei B-Rep-Modellen) beschreiben.

Gängiger ist vielleicht die Idee, nur den Übergang von zweidimensionalen, ebenen Flächen zu dreidimensionalen Körpern durch Beschreibung einer "erzeugenden Kurve" zu schaffen. Man nähert sich damit der Vorstellung der Erzeugung eines dreidimensionalen Körpers mit Hilfe einer Strangpresse ("Extrusion"). Läßt man nur translatorische Sweeps orthogonal zur Ebene der Leitkurve zu, so kommt man zu einem sogenannten 2½-D-Modell. Solche Extrusionsmodelle sind im Bauwesen, wo man es meist mit orthogonalen, prismatischen Strukturen zu tun hat, beliebt (vgl. Bild 2.1). Man reduziert das Problem der Beschreibung dreidimensionaler Körper damit im wesentlichen auf das Problem der Beschreibung eines ebenen geometrischen Gebildes. Für die Beschreibung des ebenen Gebildes steht dann – im Sinne eines "hybriden" Modells – wieder die gesamte Bandbreite der Techniken der geometrischen Modellierung zur Verfügung.

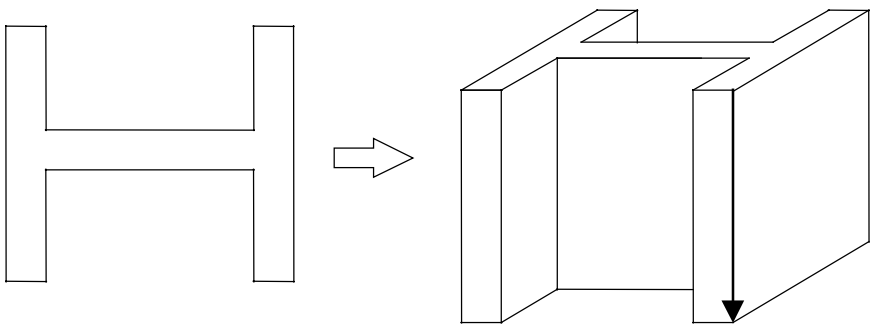

Bild 2.1: Darstellung eines 3D-Körpers in einem Extrusionsmodell. Gegeben ist eine zweidimensionale "Leitfläche" und ein Extrusionsvektor.

Es ist klar, daß ein solches Modell einen dreidimensionalen Körper nicht eindeutig beschreibt, weil man sich im allgemeinen verschiedene Konstruktionsvorgänge vorstellen kann, die dasselbe Objekt zum Ergebnis haben (z.B. kann das in der Abbildung dargestellte Objekt von oben nach unten oder von unten nach oben extrudiert werden). Auch für die Visualisierung ist ein solches generatives Modell nicht besonders gut geeignet. Generative Modelle werden wohl meist zusammen mit anderen Modellen benützt, aber selten eigenständig und ausschließlich.

2.3 Zellzerlegung (räumliche Enumeration)

Eine weitere Möglichkeit, geometrische Gebilde zu beschreiben, beruht auf der einfachen Idee, alle "Punkte" aufzuzählen, die innerhalb des zu beschreibenden Objektes gelegen sind.

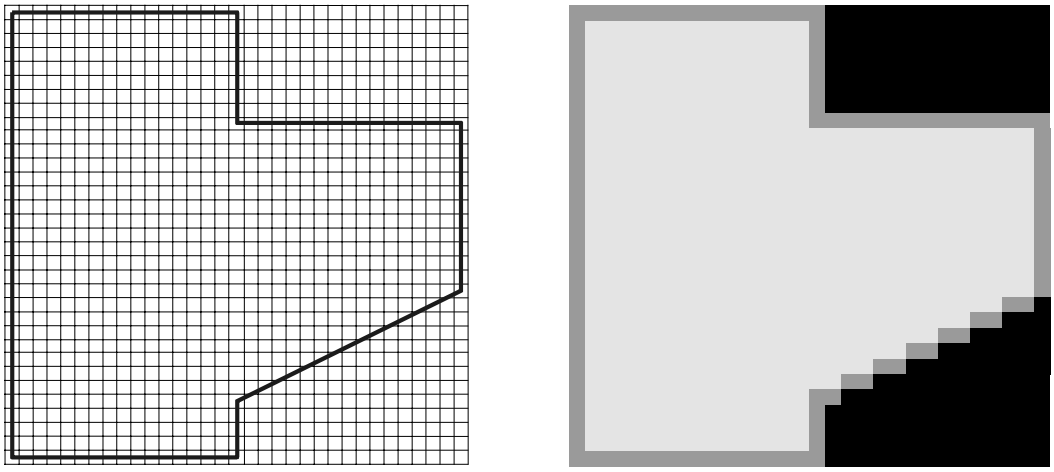

Bild 2.2: Darstellung eines zweidimensionalen Objektes durch eine Zellzerlegung

Dazu verwendet man eine Aufteilung des Raums in *Zellen* von "kleiner" Ausdehnung. Beispielsweise teilt man den Raum mit Hilfe eines *Maschengitters* gleichmäßig in Zellen auf. Man markiert nun alle Zellen, die vom Objekt "ausgefüllt" werden. Im zweidimensionalen Fall (siehe Abbildung 2.2) kann man kleine Quadrate als Zellen wählen, im dreidimensionalen würfelförmige Zellen einsetzen. Solche Einheitszellen werden auch als *Normzellen* bezeichnet.

Im einfachsten Fall kann man sich das Vorgehen im zweidimensionalen Fall so vorstellen, daß man ein Blatt karierten, transparenten Papiers über das darzustellende Objekt legt. Nun markiert man alle Karos, die ganz innerhalb des Objektes liegen, mit "innen", die leeren Karos mit "außen" und die teilweise gefüllten Karos mit "Rand".

Im Rechner kann man eine solche *Normzellenzerlegung* durch eine Matrix beschreiben, die für jede Zelle einen Eintrag vorsieht. Programmiertechnisch ist eine Matrix ein "Feld", d.h. ein zusammenhängender Speicherbereich, der über Indices adressiert werden kann. Im Dreidimensionalen wird die Zellzerlegung durch ein dreidimensionales Feld beschrieben.

Die Idee der Beschreibung eines geometrischen Objektes durch *Zellzerlegung* ist jedermann aus der Technik vieler heute üblicher *Ausgabegeräte* für Rechner bekannt: Die meisten der heute üblichen Ausgabegeräte bauen auf die *Rastergraphik* auf, setzen ein Bild also aus einzelnen *Bildpunkten* zusammen. Die Bildpunkte werden dabei als *Pixel* (*picture elements*) bezeichnet. Analog zu diesem Begriff spricht man bei dreidimensionalen Zellzerlegungen von *Voxels* (räumliche Elemente, *volume elements*).

Mit einer Zellzerlegung kann man beliebige räumliche Gebilde sehr einfach und vor allem – bei gegebener Maschenweite – *eindeutig* beschreiben.

Es entsteht nun die Frage, wie man mit dem Computer die *Zellzerlegung* eines geometrischen Objektes ermittelt. Die Methode mit dem Transparentpapier und dem Markieren der Kästchen ist ja nicht ohne weiteres auf den Rechner übertragbar.

Im folgenden seien kurz die Grundideen zu einem für diese Aufgabe geeigneten *Algorithmus* für *zweidimensionale geometrische Objekte*, hier speziell für ebene, stückweise geradlinig berandete Polygone, genannt.

Wir gehen dabei in drei Schritten vor. Der allererste Schritt ist einfach: Wir markieren *alle* Zellen unserer Zellzerlegung mit dem Wert "außen". Sodann ermitteln wir alle Zellen (Rasterpunkte), die auf dem *Rand* des Polygons liegen. Dies bedeutet, daß wir für jede Berandung des Polygons eine zusammenhängende Folge von *Pixeln* suchen, die mit minimaler Abweichung der gegebenen Originalstrecke folgen. Als Maß der Abweichung kann man z.B. die Summe der Abstandsquadrate verwenden.

Im letzten Schritt schließlich markieren wir alle Punkte, die im *Inneren* des Polygons liegen. Ausgehend von einem beliebigen Innenpunkt markieren wir alle Nachbarzellen ebenfalls als "innen", bis wir zum bereits markierten Rand der Struktur kommen.

Der Rand des Polygons setzt sich in unserer vereinfachten Problemstellung aus geradlinigen Strecken zusammen. Die Rasterung einer Strecke ist die Grundaufgabe der Computergraphik, und für unsere Aufgabe aus der geometrischen Modellierung wollen wir uns die Methoden der Computergraphik (sogenannter BRESENHAM-Algorithmus) zunutze machen:

Wir bewegen uns dabei in einem Koordinatensystem *(x,y)*, das die Lage der einzelnen Zellen (Bildpunkte) in der Zellenmatrix (hier Pixelmatrix) beschreibt (siehe Bild 6). Die Koordinaten *x* und *y* können nur ganzzahlige Werte (Zeile und Spalte des jeweiligen Bildpunktes) annehmen. Eine gerade Linie, die zwei Punkte (x_0, y_0) und (x_1, y_1) verbindet, soll durch eine Folge von markierten (eingefärbten) Zellen (Bildpunkten) angenähert werden. Es ist nun zu bestimmen, welche Zellen markiert werden müssen. Da die zu zeichnende Linie nicht in das Pixelraster fällt, müssen wir jeweils den der Linie am nächsten gelegenen Rasterpunkt markieren.

Gegeben seien die bereits in das ganzzahlige Rasterkoordinatensystem transformierten Endpunkte (x_0, y_0) und (x_1, y_1).Es sei $x_0 < x_1$ und $y_0 < y_1$, d.h. die Linie soll sich im 1. Quadranten der Ebene befinden. Für Linien, deren Richtung im 3. Quadranten liegt, vertauschen wir die beiden Endpunkte; Linien aus dem 2. und 4. Quadranten können wir symmetrisch zum ersten Fall erledigen.

Wir definieren die Steigung $m = (y_1 - y_0)/(x_1 - x_0)$ und können nun die Geradengleichung der zu zeichnenden Linie $y = y_0 + m \cdot (x - x_0)$ aufstellen. Nun ermitteln wir die einzuschwärzenden Pixel zwischen den beiden Endpunkten der Strecke einfach, indem wir *x* in ganzzahligen Inkrementen von 1 erhöhen, bis wir x_1 erreicht haben, und den zugehörigen *y*-Wert aus der Geradengleichung ermitteln und auf den nächstgelegenen ganzzahligen Wert runden. Ein mögliches Ergebnis dieses Rasterungs-Algorithmus für verschiedene Linien im 1. Quadranten der Ebene zeigt Bild 2.3.

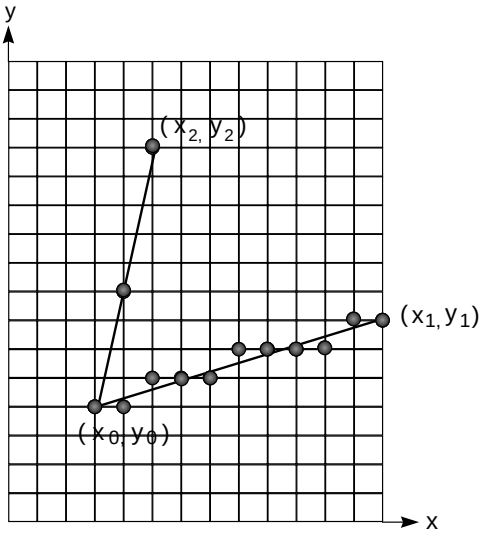

Bild 2.3: Rasterung einer Strecke. Erster Versuch zu einem Algorithmus.

Es ist sofort klar, daß dieser Algorithmus nur für $m \leq 1$ funktioniert, da sonst keine geschlossene Abfolge von benachbarten Pixeln mehr eingefärbt wird. Falls die Steigung der Geraden größer als 1 ist, müssen wir daher die Rollen von *x* und *y* vertauschen. Wir müssen also nicht nur jeden Quadranten, sondern jeden Oktanten der Ebene gesondert behandeln.

Wir unterscheiden dazu zwischen einer *treibenden* Achse und einer abhängigen Achse. Die treibende Achse ist bei $m \leq 1$ die *x*-Achse. Die Linie wird gezeichnet, indem wir, von (x_0, y_0) ausgehend, *x* jeweils um 1 erhöhen. Dazu ermitteln wir die zugehörige, auf ganzzahlige Werte gerundete *y*-Koordinate. Mit diesem Algorithmus können wir nunmehr Linien mit $0 \leq m \leq 1$ zeichnen; alle anderen Fälle lassen sich durch Symmetrieüberlegungen lösen. Dann wird ggf. die *y*-Achse zur treibenden und die *x*-Achse zur abhängigen Achse.

In der Computergraphik, aus der wir unseren Algorithmus entlehnt haben, spielt die *Effizienz* des Rasterungsverfahrens eine entscheidende Rolle; daher sind in der Praxis die für jedes einzelne Pixel erforderlichen *Rundungsoperationen* (d.h. Aufruf einer Funktion) unerwünscht, und man bevorzugt gegenüber unserer *reellwertigen Arithmetik* eine *rein ganzzahlige*.

Bresenham - Algorithmus

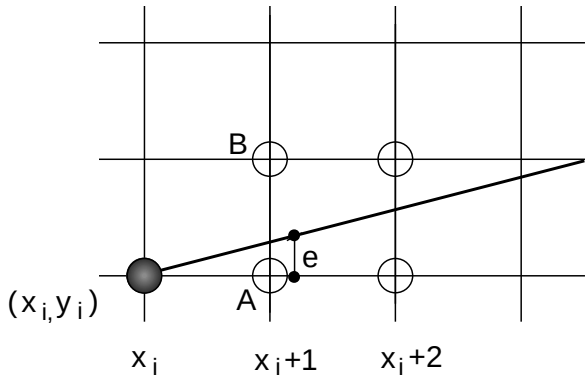

Bild 2.4: Rasterung einer Strecke mit dem BRESENHAM-Algorithmus.

Um den Aufruf der Rundungsfunktion zu vermeiden, geht man wie folgt vor: Man untersucht in jedem *x*-Inkrement, ob auch *y* inkrementiert werden muß. Am Beispiel von $0 \le m \le 1$ soll dies erläutert werden (Bild 2.4): Der *y*-Wert der exakten Linie mit Steigung *m* nimmt bei Inkrementieren von *x* um 1 gerade um *m* zu. Wenn der Punkt (x_i, y_i) exakt auf der Linie liegt, so ist der vertikale Abstand *e* zwischen $(x_i + 1, y_i)$ und der exakten Linie also gerade gleich *m*. Der *y*-Wert muß immer dann inkrementiert werden, wenn *e*>½ wird. Wir können diese Bedingung auch als 2*m*>1 oder 2*m*-1>0 formulieren. Wir multiplizieren mit Hinblick auf ganzzahlige Rechnung noch mit $x_1 - x_0$ durch und testen die so definierte Entscheidungsvariable $d = 2m(x_1 - x_0) - (x_1 - x_0) = 2(y_1 - y_0) - (x_1 - x_0)$. *y* wird inkrementiert, falls $d > 0$. Wir beobachten jetzt also anstelle des reellwertigen Abstandes *e* den ganzzahligen Wert *d*.

Falls wir den *y*-Wert unverändert gelassen haben (Punkt A in Bild 2.4), hat sich nach dem *x*-Inkrement (von Punkt x_i zu $x_i + 1$) der Abstand *e* um *m* vergrößert, im anderen Fall um *m*-1 (dieses Inkrement ist negativ!). Wir inkrementieren *d* somit im Fall A um $2(y_1 - y_0)$, im Fall B um $2(y_1 - y_0) - 2(x_1 - x_0)$ und können somit auch für Punkt $x_i + 2$ entscheiden, ob *y* inkrementiert werden muß oder nicht. Dieser rundungsfreie, rein ganzzahlige Algorithmus wird als BRESENHAM-Algorithmus der Computergraphik bezeichnet. Für jeden Punkt der Linie ist lediglich eine `if`-Verzweigung mit dem Test $d > 0$? sowie die entsprechende Aktualisierung von *d* notwendig.

Mit Hilfe des genannten Algorithmus kann man die Rasterdarstellung beliebiger Strecken erzeugen. Die mit dem unserem Algorithmus erzeugte Zelldarstellung der Linie entspricht im Limit unendlich feiner Zelleinteilung der mathematischen Idee einer Strecke, also einer Linie mit der Breite 0. Es ist nicht ganz offensichtlich, kann aber bewiesen werden, daß die mit dem BRESENHAM-Algorithmus gefundene Pixelfolge in der Tat bezüglich der Summe der Abstandsquadrate zwischen Strecke und Pixel optimal ist.

Die Zellzerlegung eines geschlossenen Polygonzugs kann jetzt natürlich ganz einfach durch Aneinanderhängen der Zelldarstellungen der einzelnen Kanten des Polygonzugs erzeugt werden. Damit haben wir eine wichtige Vorarbeit für die Ermittlung der *Zelldarstellung des Polygons* erledigt. Jetzt muß nur noch das Innere des Polygonzugs "ausgefüllt" werden.

Für diese Aufgabe steht uns der Algorithmus "Saatfüllen" (*Seedfill*) zur Verfügung. Die Grundidee des Algorithmus ist sehr einfach: Man sucht zunächst eine *beliebige* Zelle, die sich *im Inneren* der Fläche befindet. Diese Zelle wird als "innen" liegend markiert. Die Zelle gibt sodann den Markierungsbefehl an ihre vier direkten Nachbarn weiter, die links, rechts, oberhalb und unterhalb der Startzelle gelegen sind. Die diagonal benachbarten Zellen bleiben unberücksichtigt.

Für jede der Nachbarzellen führen wir folgende einfachen Schritte aus: Ist die Zelle bereits als "innen" (bzw. "Rand") markiert, so sind wir fertig. Im anderen Fall markieren wir die Zelle als "innen" und geben die Aufgabe wiederum an die vier Nachbarn weiter.

Der Algorithmus arbeitet sich also *rekursiv* durch die Pixelmatrix hindurch und kommt automatisch zum Stillstand, wenn alle inneren Zellen markiert sind. Der zuvor durchgeführte BRESENHAM-Algorithmus stellt sicher, daß die Zelldarstellung der Berandung der Fläche *lückenlos* ist, so daß der *Seedfill*-Algorithmus nicht nach außen über den Rand vordringen kann, jedenfalls nicht, wenn jeder Zelle die Markierungsanweisung nur an ihre *direkten* und nicht an ihre *diagonalen* Nachbarn weitergibt.

Der *Seedfill*-Algorithmus ist sehr einfach zu programmieren und liefert die gesuchte Zellzerlegung, obwohl durch die Weitergabe des Markierungsauftrags an alle Nachbarn der zuletzt untersuchten Zelle viele Zellen bis zu viermal getestet werden.

Es sei nicht verschwiegen, daß der Saatfüll-Algorithmus somit nicht gerade der effizienteste Algorithmus ist. Außerdem ist durch die rekursive Formulierung für die praktische Durchführung ein großer Rücksprungstapel erforderlich, so daß der Algorithmus bei hoher Auflösung schnell an seine Grenzen stoßen wird.

Für *räumliche* Zellzerlegungen kann man ähnliche Algorithmen entwickeln, die jedoch den Rahmen dieser Vorlesung sprengen würden.

Natürlich kann man durch die Zellzerlegungstechnik räumliche Objekte nur mit einer bestimmten, durch die Maschenweite der Zelleinteilung vorgegebenen, begrenzten Genauigkeit (in der Graphik als *Auflösung* bezeichnet) beschreiben. Die Darstellung wird zu ungenau, wenn man ein zu grobes Zellgitter verwendet. Man kann die Genauigkeit jedoch jederzeit erhöhen, indem man ein feineres Gitter verwendet. Dabei wächst der Speicherbedarf allerdings in 2D quadratisch, in 3D gar kubisch an, so daß Zellzerlegungen mit hoher Auflösung sehr teuer werden.

Wir wollen uns jetzt der Frage des *Speicherbedarfs* einer Zerlegung der Struktur in Normzellen zuwenden. Es liegt nahe, den hohen Speicherplatzbedarf dadurch zu reduzieren, daß man nur in solchen Bereichen, wo es wirklich notwendig ist, ein feines Gitter verwendet, in anderen Bereichen hingegen grobe Gitter. Zum Beispiel ist es unnötig, Bereiche mitten im Inneren des Körpers durch eine große Zahl von Zellen zu beschreiben. In der Nähe des *Randes* des Körpers sind feine Zelleinteilungen jedoch notwendig. Man möchte also anstelle der *Normzellen* gerne Zellen *unterschiedlicher* Größe verwenden, um lokale Effekte besser auflösen zu können.

In diesem Fall wird der 3D-Körper durch eine hierarchische, rekursive Zerlegung des Raumes in quaderförmige Zellen beschrieben, denen die Attribute "innen", "außen" oder "Rand" zugeordnet werden. Durch lokale Verfeinerung der Zellzerlegung läßt sich eine verbesserte Darstellung erzielen. Man fährt fort, zu verfeinern, bis man entweder eine Verfeinerungsgrenze erreicht oder nur noch Pixel bzw. Voxel gleichen Typs vorliegen hat.

Bild 2.5 zeigt ein typisches Beispiel. Zunächst wird ein Quadrat ermittelt, das die darzustellende Figur ganz umschließt. Für dieses Quadrat wird eine Normzellen-Einteilung gewählt, die einer Zweierpotenz entspricht – in unserem Beispiel $2^6 = 64$. Dieses Raster wird nun zunächst in der Mitte (in unserem Beispiel also bei $(x; y) = (32;32)$ geteilt. Dabei entstehen vier Quadranten, die wir mit *NW* ($0 \le x < 32$ und $0 \le y < 32$, links oben), *SW* (links unten), *SE* (rechts unten) und *NE* (rechts oben) bezeichnen, in Anlehnung an die Himmelsrichtungen Nordwest, Südwest, Südost und Nordost.

Jeder dieser Quadranten wird nun seinerseits wieder in vier Quadranten zerlegt, und so weiter, bis wir im 6. Schritt gerade die Einteilung in $64 \cdot 64$ Normzellen erreichen. Auf jeder Ebene ist wieder derselbe Algorithmus anzuwenden, den wir auch auf der obersten Ebene verwendet hatten (sogenannte *Rekursion*).

Im nächsten Schritt können wir nun wieder jeweils vier nebeneinanderliegende Zellen zu einer einzigen Zelle zusammenfassen, wenn alle vier Zellen dieselbe Färbung besitzen. Dieses Vorgehen der Vergröberung setzen wir bis zur obersten Ebene fort, falls möglich.

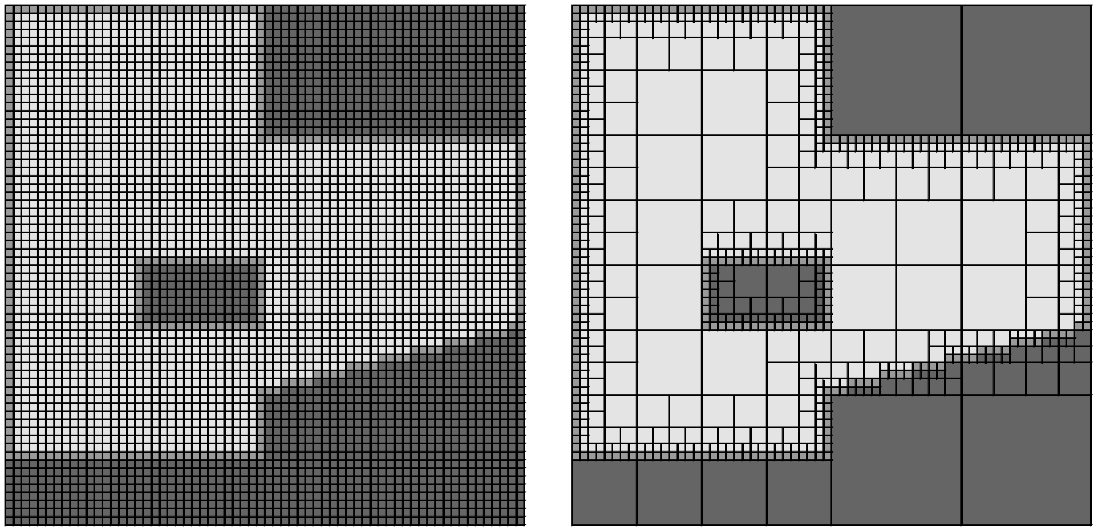

Bild 2.5: Vergleich zwischen einer Zerlegung einer Struktur in Normzellen (links) und einer hierarchischen Zerlegung (*Quadtree*-Datenstruktur, rechts).

Wir erhalten somit die in Bild 2.5 rechts dargestellte Zellenzerlegung, die nur noch dort eine hohe Feinheit besitzt, wo dies tatsächlich notwendig ist, nämlich in Randnähe. In großen Bereichen mit gleicher Einfärbung kommen wir hingegen mit wenigen Zellen aus. In unserem Bild enthält die gleichmäßige Normzellenzerlegung $64 \cdot 64 = 4096$ Zellen, während die rekursiv verfeinerte bzw. vergröberte Darstellung nur 989 Zellen enthält.

Um die rekursiv verfeinerte Darstellung der geometrischen Figur in eine Datenstruktur zu übersetzen, merken wir uns für jede Ebene die jeweils vier neu entstandenen Quadranten. Außerdem müssen wir uns irgendwo die Größe des Startquadrates (in Pixeln) merken.

Für die Figur in Bild 2.5 entsteht eine Datenstruktur, die aus Platzmangel hier nicht dargestellt werden kann. In Bild 2.6 ist jedoch eine gröbere Darstellung derselben Figur ($16 \cdot 16$ Pixel) dargestellt, für die auch die zugehörige Datenstruktur gezeichnet werden kann.

Es entsteht eine *Baumstruktur*. Die Wurzel des Baums entspricht der ersten Teilung unseres Bildes an $(x; y) = (8;8)$. Die Wurzel des Baums liegt in der Informatik in zeichnerischen Darstellungen immer oben. Von der Wurzel gehen vier Pfeile aus, die zu den entstandenen Quadranten weisen. Jeder dieser Unterquadranten verweist seinerseits wieder auf seine eigenen Unterquadranten. Die Elemente, die durch die Pfeile des Baums verbunden werden, werden als *Knoten* bezeichnet, die Pfeile auch als *Kanten*.

Es entsteht also ein Baum, bei dem von jedem Knoten bis zu vier Pfeile ausgehen. Ein solcher Baum wird als Quadrantenbaum oder *Quadtree* bezeichnet. Die Anzahl der Pfeile, die man von der Wurzel bis zum am weitesten entfernten Knoten durchlaufen muß, wird als *Höhe* des Baums bezeichnet. Der Baum in Bild 2.6 hat also die Höhe 3. Die äußeren Knoten des Baums, von denen keine weiteren Pfeile mehr ausgehen, werden auch als *Blätter* bezeichnet, die anderen Knoten als *innere* Knoten. Ein vollbesetzter Quadtree ist ein Baum, dessen Blätter alle gleich weit von der Wurzel entfernt sind und in dem von jedem inneren Knoten genau vier Pfeile ausgehen. Die Höhe eines vollbesetzten Quadtrees wächst nur logarithmisch mit der Anzahl der Knoten, die der Baum enthält.

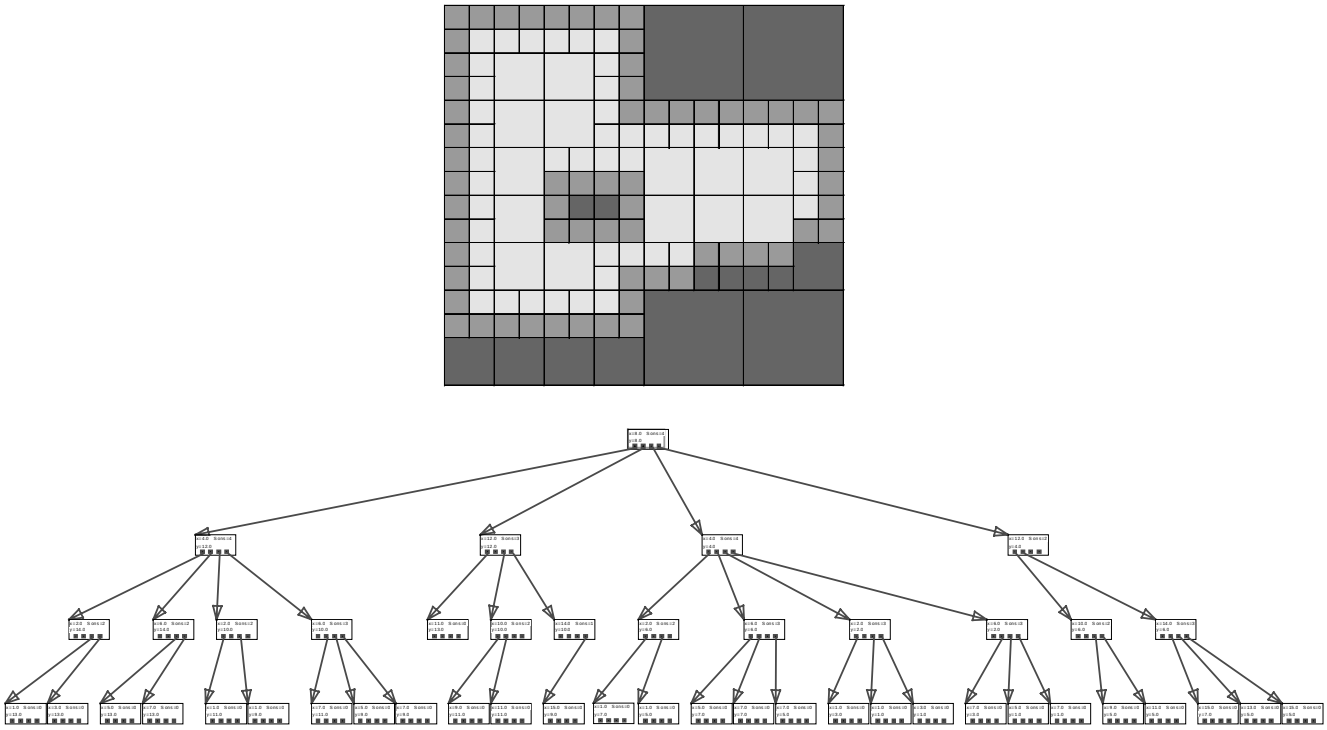

Bild 2.6: Zerlegung eines Objekts in 16·16 Zellen und zugehöriger Quadtree. Dieser Baum ist von links nach rechts in der Reihenfolge der Quadranten *SW, SE, NW, NE* geordnet.

Ein Knoten *j,* der von einem Knoten *i* aus über einen Pfeil erreicht werden kann, wird als *Nachfolger* von *i* bezeichnet.

Die Darstellung eines geometrischen Gebildes mit Hilfe eines Quadtrees lohnt sich dann, wenn der Baum nicht voll besetzt ist, sondern sich viele Quadranten zu größeren Einheiten zusammenfassen lassen. Das Beispiel in Bild 2.6 zeigt, daß hier schon einige Vereinfachungen getroffen werden konnten, der Baum aber noch noch recht viele Einträge enthält. Je feiner wir unsere geometrische Figur auflösen, desto mehr lohnt sich die Darstellung als Quadtree.

Man beachte, daß die Quadtree-Darstellung in Bild 2.6 entsteht, wenn man die unteren beiden Ebenen des Quadtrees aus Bild 2.5 einfach abschneidet. Beim Abschneiden der Nachfolger jedes Knotens weist man dem Knoten entweder die Kennzeichnung "Rand" zu, wenn der Knoten Nachfolger verschiedenen Typs hatte, oder die Kennzeichnung "innen" bzw. "außen", wenn der Knoten ausschließlich Nachfolger des entsprechenden Typs hatte. Der Quadtree erlaubt also je nach Bedarf eine hierarchische Vergröberung bzw. Verfeinerung der Darstellung. Hierarchisch bedeutet dabei, daß die höheren Ebenen des Baums nicht neu analysiert werden müssen, wenn man in einer tieferen Ebene die Auflösung verändert.

Nun noch zum Speicherplatzbedarf eines Quadtrees. Rein rechnerisch müssen wir für jeden Knoten die Farbe des Knotens sowie die vier Verweise auf andere Knoten, also insgesamt fünf Informationen, speichern. Wir sparen also erst dann gegenüber der Darstellung als Matrix Speicherplatz ein, wenn die Anzahl der Knoten weniger als ein Fünftel der Anzahl der Einträge einer Speichermatrix bei der Normzellenzerlegung beträgt. Da wir normalerweise wesentlich feinere Zerlegungen als in unseren Abbildungen verwenden, ist der Punkt, an dem die Quadtree-Darstellung vom Speicheraufwand her günstiger wird, jedoch schnell erreicht (in unserem Beispiel schon bei 128·128 Pixeln).

Speicherplatz ist aber nicht die einzige Überlegung, die zum Einsatz von Quadtrees führen kann. Wir wollen die Zellzerlegung ja zu irgend einem Zweck verwenden, d.h. wir wollen Operationen mit der Datenstruktur durchführen. Je weniger Elemente zur Darstellung unseres geometrischen Gebildes notwendig sind, desto schneller können die Operationen erledigt werden, die das geometrische Gebilde manipulieren.

Bäume sind Spezialfälle von Datenstrukturen, die in der Informatik eine bedeutende Rolle spielen, nämlich Spezialfälle von *Graphen*. Wir werden uns später in diesem Kapitel noch allgemein mit der Darstellung von Graphen durch Datenstrukturen im Rechner beschäftigen.

Es gibt natürlich auch Bäume, bei denen die Anzahl der von jedem Knoten weiterführenden Pfeile größer oder kleiner als vier sein kann. Bei Bäumen, in denen jeder Knoten maximal zwei Nachfolger hat, sprechen wir von *Binärbäumen*, bei Bäumen, in denen der maximale *Knotengrad* (die maximal zulässige Anzahl von Nachfolgern) acht beträgt, spricht man von *Octrees*.

Octrees können in einer Verallgemeinerung unserer Überlegung zur Zelldarstellung mit zweidimensionalen Pixels zur speichersparenden und effizienten Speicherung von *dreidimensionalen* Objekten eingesetzt werden. Die Vorgehensweise ist dabei völlig analog. Man stellt das Objekt mit Hilfe eines gleichmäßigen Rasters aus Normwürfeln dar und faßt dann gegebenenfalls jeweils acht benachbarte Würfel zu einem neuen Würfel doppelter Kantenlänge zusammen. Da der Speicheraufwand einer Normzellenzerlegung in 3D kubisch mit der Auflösung wächst, ist eine effizientere Speicherung, also der Einsatz von Octrees, fast unumgänglich.

Eine Zellzerlegungsdarstellung eines geometrischen Objektes kann für viele Aufgaben der Visualisierung gut genutzt werden, vor allem, wenn man bedenkt, daß die üblichen Graphik-Ausgabegeräte allesamt *Rastergraphik*-Geräte sind und daher ohnehin nur eine begrenzte Auflösung bieten. Die Auflösung eines Rastergerätes ist begrenzt, so daß es keinen Sinn hat, die 3D-Geometrie genauer zu beschreiben, als es der Darstellungsgenauigkeit eines solchen Gerätes entspricht.

Auch für Aufgaben wie die Ermittlung des Volumens komplexer Körper ist die Zellzerlegungsdarstellung gut geeignet. Für die Generierung linienorientierter Informationen (z.B. Pläne) sind Zellzerlegungsdatenstrukturen hingegen ungeeignet.

Zellzerlegungsdarstellungen, die auf Normzellen beruhen, gehören zu den wenigen Datenstrukturen, die eine *eindeutige* Objektbeschreibung für 3D-Körper gestatten, d.h., zwei gleiche Objekte besitzen in einer Zerlegung in regelmäßige Zellen auch dieselbe Darstellung.

Zellzerlegungsdarstellungen – Zerlegungen in Normzellen, aber auch Quadtree- bzw. Octree-Darstellungen – ermöglichen eine sehr einfache und direkte Implementierung von Mengenoperationen.

Viele numerische Verfahren, z.B. Finite-Differenzen-, Finite-Volumen- und Finite-Elemente-Methoden, erfordern eine Diskretisierung des geometrischen Objektes mit Hilfe eines Gitters. Eine Zellzerlegungsdarstellung eines Objektes bietet sich als Berechnungsgitter für solche Verfahren an.

Reguläre Gitter (also Normzellenzerlegungen) sind insbesondere für Finite-Differenzen-Methoden wünschenswert. Finite-Differenzen-Methoden sind einfach zu implementierende Methoden zur schnellen näherungsweisen Lösung von partiellen Differentialgleichungen. Besonders leicht sind solche Verfahren zu realisieren, wenn man eine Darstellung des zu untersuchenden Gebietes in Form eines Gitters mit konstanter Maschenweite besitzt.

2.4 Parametrisierte Grundkörper

Anstatt beliebige Körper als Bestandteile eines geometrischen Modells zuzulassen, kann man sich natürlich von Anfang an auf einen begrenzten Satz einfacher Grundkörper beschränken. Eine Szene wird in solchen Modellen aus Instanzen vorgegebener Grundkörper zusammengesetzt.

Nun entsteht zunächst die Frage nach einer geeigneten Auswahl von Grundkörpern und nach einer geeigneten Parametrisierung. Natürlich könnte man z.B. als Grundkörper einen Quader vorsehen und diesen mit Breite, Länge und Höhe parametrisieren. Außerdem muß man mit Hilfe der Parametrisierung angeben, an welcher Stelle des Raumes der Quader positioniert werden soll, und wie er gedreht werden soll.

Es ist sinnvoll, eine einheitliche Strategie zur Parametrisierung und Positionierung der Grundkörper zu entwickeln. Dazu wollen wir uns zunächst eine Technik zur Beschreibung allgemeiner geometrischer Transformationen ansehen.

2.4.1 Transformationen und homogene Koordinaten

Wir besprechen die Transformation *ebener* Figuren. Die Verallgemeinerung auf 3D-Objekte ist offensichtlich und wird daher dem Leser überlassen.

Die wichtigsten geometrischen Transformationen sind Translation (Verschiebung), Rotation (Verdrehung), Skalierung (Dehnung) und Scherung. Alle diese Transformationen sowie ihre Zusammensetzung zu komplexeren Transformationen können einheitlich durch Transformationsmatrizen dargestellt werden, wenn man von den gewöhnlichen *kartesischen Koordinaten* zu sogenannten *homogenen Koordinaten* übergeht.

Die im folgenden besprochenen Abbildungen sind linientreu, d.h. Geraden werden wieder in Geraden abgebildet. Daher reicht es zur Transformation geradlinig begrenzter Gebilde aus, die Schnittpunkte der Geraden zu transformieren und wiederum geradlinig zu verbinden, um das Bild einer transformierten Figur zu erhalten.

In kartesischen Koordinaten werden Punkte durch ein Koordinatenpaar (x, y) dargestellt. Bei der Darstellung in *homogenen* Koordinaten erweitern wir dieses Koordinatenpaar durch eine dritte Komponente t: $\begin{bmatrix} x \\ y \\ t \end{bmatrix}$ stellt jetzt also einen *Punkt* im zweidimensionalen Raum dar. Alle Koordinatentripel, die sich durch Multiplikation aller Komponenten mit einem skalaren Faktor ineinander überführen lassen, stellen denselben Punkt im zweidimensionalen Raum dar. Beispielsweie stellen die Tripel $\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ und $\begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix}$ denselben Punkt dar. Sofern $t \neq 0$, können wir alle Komponenten des Tripels durch t dividieren und kommen damit zur sogenannten homogenisierten Form der Darstellung: $\begin{bmatrix} x/t \\ y/t \\ 1 \end{bmatrix}$. Punkte, für die $t = 0$ ist, werden *Punkte im Unendlichen* genannt. Man kann die homogenen Koordinaten eines Punktes im zweidimensionalen Raum als Gleichung einer Geraden durch den

Ursprung des dreidimensionalen Raumes auffassen, $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} cx \\ cy \\ ct \end{bmatrix}$. Die kartesischen Koordinaten des Punktes erhält man, wenn man den Durchstoßpunkt dieser Geraden durch die Ebene $Z = 1$ bestimmt. Punkte, die auf einer Geraden liegen, die parallel zur Ebene $Z = 1$ verläuft, also die unendlich fernen Punkte, können nicht in kartesische Darstellung überführt werden. Wählt man alle drei Komponenten der homogenen Darstellung zu 0, d.h. $c = 0$, so wird durch diese Darstellung keine Gerade im dreidimensionalen Raum beschrieben; dieser Fall ist daher unzulässig. Um einen Punkt aus der Darstellung in homogenen Koordinaten in kartesische Koordinaten zu überführen, braucht man also lediglich die Darstellung zu homogenisieren und die *t*-Koordinate wegfallen zu lassen.

2.4.2 Translation

Die Matrixdarstellung der Translation eines Punktes *P* in der Ebene lautet in homogenen Koordinaten:

$$\begin{bmatrix} x' \\ y' \\ t' \end{bmatrix} = \begin{bmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ t \end{bmatrix} \text{ oder } P' = \mathbf{T} \cdot P$$

Bild 2.7 zeigt die Anwendung einer Translation auf alle Punkte einer ebenen Szene.
Die Hintereinanderausführung mehrerer Translationen kann durch Verkettung der einzelnen Matrizen beschrieben werden: $P' = \mathbf{T}_1 P$ und $P'' = \mathbf{T}_2 P'$ können beschrieben werden durch $P'' = (\mathbf{T}_2 \cdot \mathbf{T}_1) \cdot P$. Für die Transformation gilt das Kommutativgesetz.

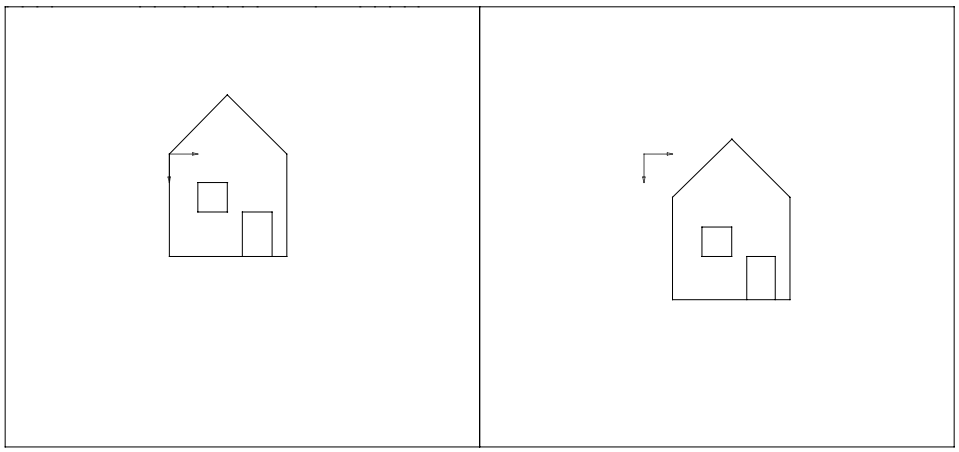

Bild 2.7: Anwendung einer Translation $\begin{bmatrix} 1 & 0 & 2 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{bmatrix}$ auf ein zweidimensionales Objekt. In diesem Beispiel weist die x-Achse von links nach rechts und die y-Achse von oben nach unten.

2.4.3 Rotation

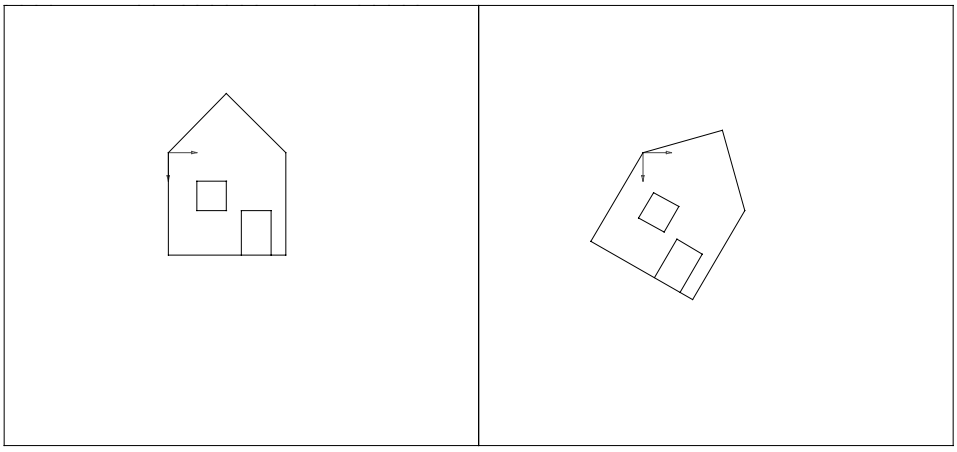

Bild 2.8: Anwendung einer Rotation $\begin{bmatrix} \cos\frac{\pi}{6} & -\sin\frac{\pi}{6} & 0 \\ \sin\frac{\pi}{6} & \cos\frac{\pi}{6} & 0 \\ 0 & 0 & 1 \end{bmatrix}$ auf ein Objekt. x-Achse horizontal, y-Achse von oben nach unten. Der positive Drehwinkel wird von der x- zur y-Richtung gemessen.

Eine Rotation *um den Ursprung des Koordinatensystems* wird in homogenen Koordinaten beschrieben durch

$$\begin{bmatrix} x' \\ y' \\ t' \end{bmatrix} = \begin{bmatrix} \cos\vartheta & -\sin\vartheta & 0 \\ \sin\vartheta & \cos\vartheta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ t \end{bmatrix} \text{ oder } P' = \mathbf{R} \cdot P .$$

Hierbei stellt ϑ den Drehwinkel dar, um den der Punkt verdreht werden soll. Er wird positiv von der x-Achse in Richtung auf die y-Achse gemessen. Bild 2.8 zeigt ein Beispiel für die Anwendung einer Rotationsmatrix auf alle Punkte einer ebenen Szene.

Auch Rotationen um den Ursprung können beliebig verkettet werden und sind untereinander kommutativ. Hintereinanderausführungen von Translationen und Rotationen (um den Ursprung) sind hingegen nicht kommutativ. Will man ein Objekt um einen vom Ursprung verschiedenen Punkt (x_p, y_p) rotieren, so muß man zunächst eine Transformation durchführen, die den Ursprung des Koordinatensystem in den gewünschten Punkt verlegt ($d_x = -x_p$, $d_y = -y_p$), dann die Rotationsmatrix anwenden, und schließlich den Ursprung wieder in die vorherige Position verlegen ($d_x = x_p$, $d_y = y_p$).

2.4.4 Skalierung

Eine *Skalierung* eines Bildes bedeutet, alle Strecken in x- und y-Richtung mit einem konstanten Faktor zu multiplizieren. Dabei können die Faktoren für x- und y-Richtung gleich oder auch verschieden sein.

Die Skalierung wird beschrieben durch die Transformation

$$\begin{bmatrix} x' \\ y' \\ t' \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ t \end{bmatrix} \text{ oder } P' = \mathbf{S}P .$$

Bild 2.9 zeigt ein Beispiel für die Anwendung einer Skalierung mit verschiedenen Faktoren für x- ud y-Richtung. Eine Scherung mit $s_x = s_y$ wird als *verzerrungsfrei* bezeichnet. Dabei bleiben die Längenverhältnisse der Figur auch nach der Transformation erhalten. Diese Transformation entspricht einer zentrischen Streckung mit dem Ursprung als Zentrum.

Durch spezielle Wahl der Skalierfaktoren s_x und s_y lassen sich auch andere häufig benötigte Operationen wie Spiegelung an der x-Achse ($s_x = 1,\ s_y = -1$), Spiegelung an der y-Achse ($s_x = -1,\ s_y = 1$) und Spiegelung am Ursprung ($s_x = -1,\ s_y = -1$) darstellen.

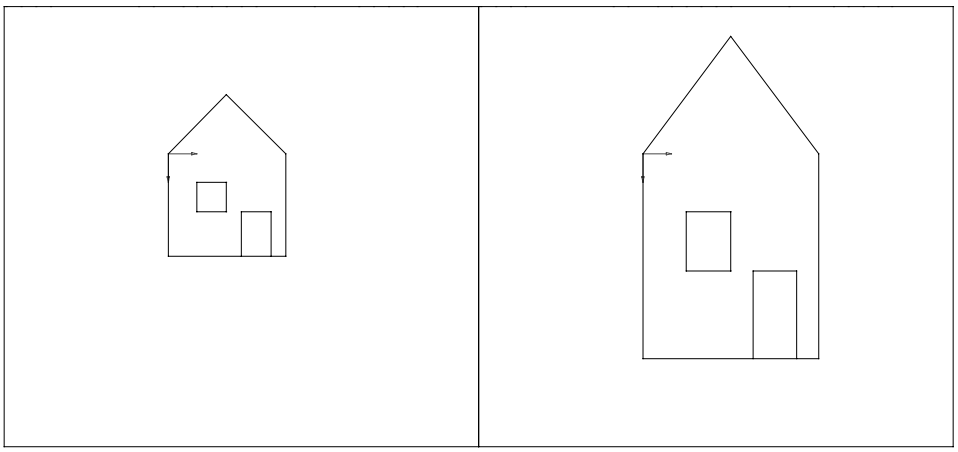

Bild 2.9: Skalierung eines ebenen Objektes mit der Transformationsmatrix $\begin{bmatrix} 1.5 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$.

2.4.5 Scherung

Die bisher eingeführten Transformationen bilden das Objekt winkeltreu in ein neues Objekt ab. Die letzte hier zu besprechende Transformation bildet das Objekt lediglich unter Erhaltung der Parallelität paralleler Geraden, jedoch nicht winkel- und längentreu ab (affine Transformation):

$$\begin{bmatrix} x' \\ y' \\ t' \end{bmatrix} = \begin{bmatrix} 1 & q_x & 0 \\ q_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ t \end{bmatrix} \text{ oder } P' = \mathbf{Q}P$$

Bild 2.10 zeigt die Anwendung einer Scherung auf ein ebenes Objekt.

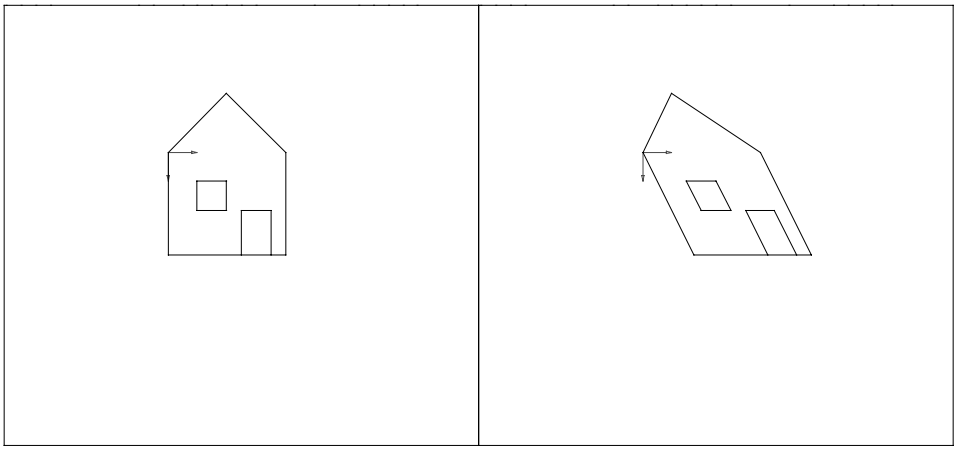

Bild 2.10: Anwendung einer Scherung in x-Richtung $\begin{bmatrix} 1 & 0.5 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ auf ein ebenes Objekt.

2.4.6 Anwendung auf die geometrische Modellierung

Die Transformation dreidimensionaler Körper erfolgt völlig analog zu der zweidimensionaler Körper. Man verwendet vierdimensionale homogene Koordinaten $\begin{bmatrix} x \\ y \\ z \\ t \end{bmatrix}$.

Somit kann eine beliebige räumliche Transformation durch eine 4x4-Matrix beschrieben werden, so daß mit 16 Parametern eine Fülle von Körpern im Raum aus jedem Grundkörper erzeugt werden kann.

Beispielsweise kann man allein mit Hilfe dieser 16 Parameter einen Einheitswürfel in den drei Richtungen beliebig skalieren und somit in einen Quader verwandeln und diesen dann beliebig im Raum positionieren.

Der Würfel ist aber nicht nur zur Erzeugung von Quadern geeignet. Durch Anwendung von Scherungen lassen sich auch hexahedrale Parallelepipede erzeugen.

Ähnlich lassen sich durch anisotrope Skalierung aus einer Einheitskugel beliebige Ellipsoide gewinnen, und aus einem Einheitszylinder mit Höhe und Durchmesser 1 lassen sich schiefe Zylinder und Zylinder mit elliptischer Grundfläche erzeugen.

Man kann also mit nur wenigen, völlig identisch parametrisierten Einheitskörpern bereits viele praktisch relevante Fälle erledigen. Bei anderen Körpern (z.B. Torus) sind gegebenenfalls zusätzlich zu den 16 Transformationsparametern weitere, spezielle Parameter notwendig (z.B. beim Torus Verhältnis von "Schlauchdurchmesser" und "Reifengröße").

In Spezialanwendungen kann man neben den bisher vorrangig betrachteten elementaren Grundkörpern auch komplexe, durch eine Vielzahl von anwendungsspezifischen Parametern beschriebene Primitive verwenden. Man denke etwa an Walzprofile im Stahlbau, an Verbindungselemente wie Winkel oder

Schrauben, etc. In CAD-Systemen, die auf solche Anwendungen spezialisiert sind, werden parametrisierte Ausgangselemente für solche Aufgaben bereitstehen. Die Flexibilität, beliebige Objekte darzustellen, oder auch nur geringfügige Änderungen an den vorhandenen Objekttypen vorzunehmen, geht dabei allerdings verloren.

2.5 Constructive Solid Geometry

Komplizierte Körper, wie sie im Ingenieurwesen an der Tagesordnung sind, gehören meist nicht zum Formenvorrat einer Sammlung parametrisierter Grundkörper. Man kann solche komplexen Objekte jedoch häufig durch *Kombination* elementarer Objekte mit den Operatoren der *Mengenlehre* erzeugen.

Dies ist der Grundgedanke der Methode *CSG*. Hinter diesem Kürzel verbirgt sich der Begriff *constructive solid geometry* oder *combinatorial solid geometry*. Aus den Grundkörpern werden durch Anwendung BOOLE*scher* Mengenoperationen neue Körper generiert. Zur Anwendung kommen die Operationen Vereinigung ($\cup$), Schnittmenge ($\cap$), Differenz (*a**b* bedeutet: "ziehe *b* von *a* ab") und Komplement ($\overline{a}$, bezeichnet den "Außenraum" des Körpers *a*). Alle diese Operationen verknüpfen je einen oder maximal zwei Operanden.

Die Bilder 2.11 bis 2.13 zeigen, wie man mit Hilfe dieser Verknüpfungen aus einfachen Körpern eine komplizierte Figur erzeugen kann:

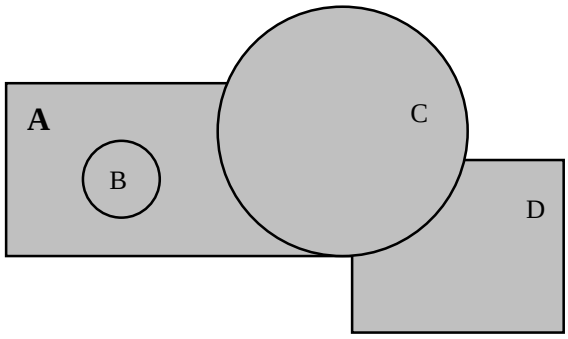

Bild 2.11: Ausgangspunkt eines CSG-Modells sind parametrisierte Grundkörper (hier mit A, B, C und D bezeichnet).

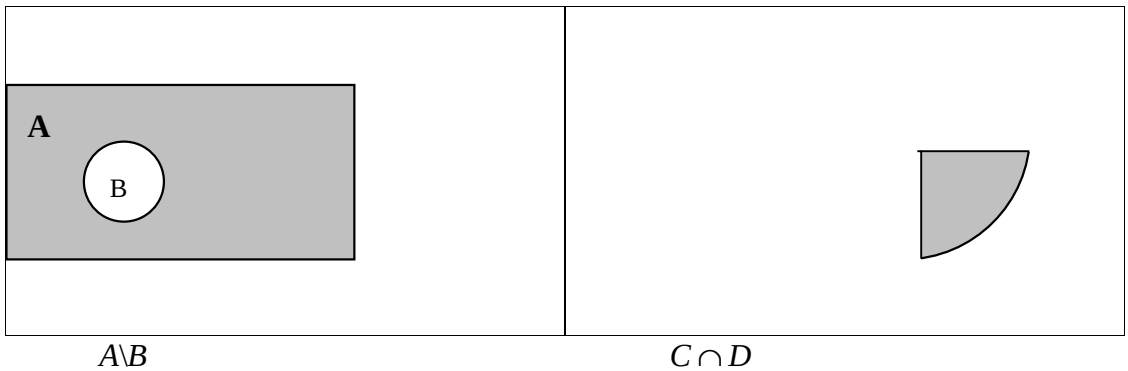

Bild 2.12: In einem ersten Schritt werden die Objekte *A**B* und $C \cap D$ gebildet.

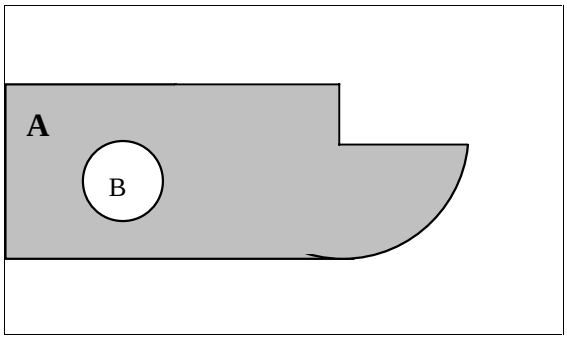

Bild 2.13: Schließlich werden die Objekte $A \backslash B$ und $C \cap D$ vereinigt. Es entsteht das Objekt $(A \backslash B) \cup (C \cap D)$.

Bei der Anwendung der BOOLEschen Operatoren tritt in manchen Fällen noch ein *Problem* auf, das wir am zweidimensionalen Beispiel verdeutlichen wollen: Wenn zwei zweidimensionale Objekte keinen Flächenbereich, sondern nur einen Punkt oder einen linienförmigen Bereich gemeinsam überdecken, so können unerwünschte Ergebnisse entstehen. Beispielsweise (Bild 2.14) kann das Ergebnis der Schnittbildung zweier flächiger Gebilde dann ein Punkt- oder linienförmiges Gebilde entstehen, oder mit dem Subtraktionsoperator wird der Rand des entstehenden Gebildes sozusagen "abgeschnitten".

Diese Ergebnisse sind unerwünscht. Wenn man zwei Zahlen mit Hilfe eines Operators verknüpft (zum Beispiel +, -, /, *), so erwartet man, daß das Ergebnis dieser Verknüpfung wiederum eine Zahl ist und nicht ein Gebilde eines anderen Typs. Die BOOLEschen Operatoren, so wie wir sie oben vorgestellt haben, erfüllen in der Anwendung auf geometrische Gebilde diese Forderung der *Abgeschlossenheit* nicht, d.h. das Ergebnis einer Verknüpfung zweier Objekte desselben Typs kann ein Objekt anderen Typs zum Ergebnis haben.

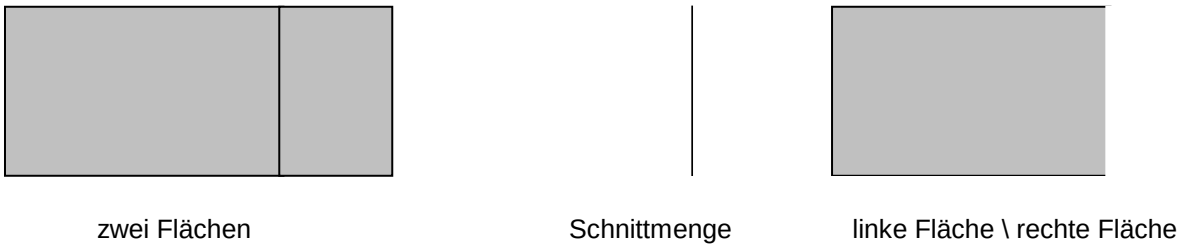

Bild 2.14: Bei zwei sich berührenden Flächen führen Schnittbildung und Subtraktion zu unerwarteten Ergebnissen.

Um dieses Problem zu vermeiden, werden die BOOLEschen Operatoren *regularisiert*. Diese bedeutet, daß man Fälle wie die in Bild 31 per Definition ausschließt. Die regularisierten BOOLEschen Operatoren werden durch einen Querstrich über dem Operatorsymbol gekennzeichnet, $\overline{\cup}$, $\overline{\cap}$, usw. Die Schnittmenge in Bild 2.14 ist bei *regularisierten* BOOLEschen Operatoren leer, der Rand der linken Fläche bleibt bei der Subtraktion der rechten Fläche erhalten.

Um die *regularisierten* Ergebnisse zu erhalten, führt man zunächst die Booleschen Operationen aus und entfernt dann in einem zweiten Schritt ungültige Elemente, z.B. Kanten ohne angrenzende Fläche oder Kanten im Inneren einer Fläche. Kanten, die nicht zum *Rand* des Objektes gehören, werden also entfernt.

Der Konstruktionsvorgang, der zur Herstellung einer komplexen Struktur führt, kann im CSG-Modell in eine Reihe von Konstruktionsvorgängen, bei denen nur je zwei Teilstrukturen beteiligt sind, zerlegt werden.

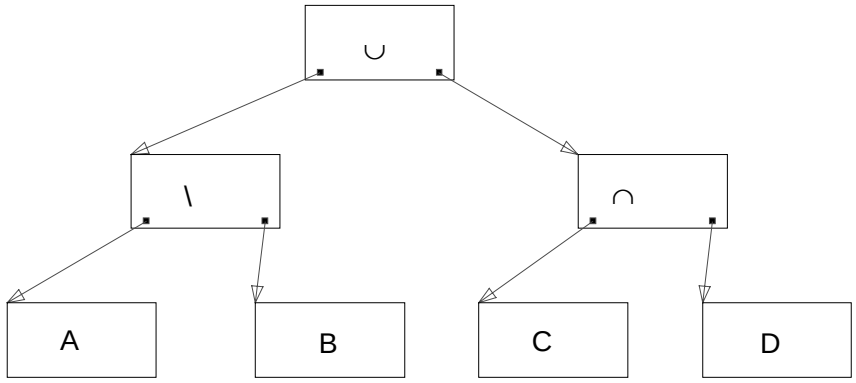

Bild 2.15: Darstellung des Objektes aus Bild 2.13 in Form eines Binärbaums (CSG-Baum)

Damit ist eine Darstellung einer Struktur in einem CSG-Modell als *Binärbaum* möglich. Ein Binärbaum ist ein Baum, dessen maximaler Knotengrad (Anzahl der von einem Knoten ausgehenden Kanten) zwei ist.

Bild 2.15 zeigt den CSG-Baum, der zu dem Objekt in Bild 2.13 gehört. In jedem *Blatt* des Baumes finden wir einen parametrisierten Grundkörper, hier also einen der Körper *A, B, C, D*. In jedem inneren Knoten des Baums ist ein *Operator* gespeichert, der eine BOOLEsche Verknüpfung der beiden Körper angibt, die in seinen beiden Teilbäumen gespeichert sind. Zu beachten ist, daß die BOOLEschen Mengenoperationen im allgemeinen nicht kommutativ sind, so daß der CSG-Baum ein *geordneter* Baum ist.

Um die Gestalt eines durch einen CSG-Baum beschriebenen Objektes zu bestimmen, müssen die Teilobjekte und Operationen von unten nach oben abgearbeitet werden. Dazu benötigt man einen Algorithmus, der alle Knoten des Baums in der gewünschten Reihenfolge aufsucht ("Traversieren des Baums"). Ein solcher Algorithmus kann rekursiv beschrieben werden durch:

> Suche den linken Teilast auf
> Suche den rechten Teilast auf
> Verknüpfe den linken Teilast mit dem rechten

Diese Traversierung des Baums wird als *Postorder*-Traversierung bezeichnet. Für jeden Knoten wird rekursiv derselbe Algorithmus durchgeführt, sofern dieser noch Teiläste besitzt. Man kann dies auch so darstellen, daß man die zugrundeliegende Formel $(A \backslash B) \cup (C \cap D)$ in "umgekehrter polnischer Notation" anschreibt: In dieser Notation (sie war früher auf HP-Taschenrechnern üblich) gibt man zuerst die Operanden, dann die Verknüpfung an. Die Szene aus Bild 2.15 wird in diesem Fall durch die Notation $A\ B \backslash C\ D \cap \cup$ beschrieben. Das Zwischenergebnis $A \backslash B$ wird dabei zunächst auf einem "Stapel" abgelegt, bis auch das Resultat aus der Verknüpfung $C \cap D$ bekannt ist. Die beiden Resultate werden schließlich durch den Operator $\cup$ verknüpft, um das Endergebnis zu erhalten.

Ein Problem der CSG-Datenmodelle besteht darin, daß der CSG-Baum *nicht eindeutig* ist. So kann z.B. auch die Struktur in Bild 2.13 in einer anderen Reihenfolge oder auch aus anderen Bestandteilen zusammengebaut werden, als in der Abbildung gezeigt.

Das CSG-Modell setzt voraus, daß die Grundkörper durch eine eigene Modellierungstechnik beschrieben werden. Beispielsweise kann man zu diesem Zweck parametrisierte Grundkörper verwenden.

Die Objekte, die durch BOOLEsche Verknüpfungen entstehen, lassen sich gut durch Normzellenschemata oder durch Quadtree-Darstellungen beschreiben. Es ist einfach, aus zwei in Normzellendarstellung gegebenen Körpern mittels einer CSG-Operation einen neuen Körper, wiederum in Zellendarstellung, zu erzeugen. Weitaus aufwendiger ist es, das Endprodukt einer CSG-Darstellung als Boundary Representation zu beschreiben.

2.6 Boundary-Representation-Modelle

Eine wichtige Methode, um Modelle geometrischer Gebilde aufzubauen, besteht darin, solche Gebilde über ihre *Oberfläche* zu beschreiben. Modelle, die auf eine Beschreibung der Oberfläche aufbauen, werden als *boundary representation* oder kurz als *B-Rep-Modelle* bezeichnet. Der Ausdruck "Oberfläche" steht hier ganz allgemein für die Berandung des Gebildes. Die Berandung eines zweidimensionalen geometrischen Gebildes besteht in dem umschließenden Linienzug des Gebildes.

Der Rand eines geometrischen Gebildes setzt sich aus Elementen des Typs Knoten (Eckpunkt, *vertex*, v), Kante (*edge*, e) und Facette (Fläche, *face*, f) zusammen. Diese Objekttypen werden als *topologische Primitive* bezeichnet. Will man auch komplizierte Gebilde mit eingeschlossenen Hohlräumen beschreiben, so muß man die Liste der topologischen Primitive noch um die Typen Schale (*shell*, zusammenhängende Oberfläche) und *region* (Bereich) erweitern. Beispielsweise stellt die Oberfläche eines Würfels ein Objekt des Typs *shell* dar, und dieses Objekt trennt das Innere des Würfels (*region* "innen") von dem unendlich ausgedehnten Außenraum (*region* "außen").

Jedes der genannten topologischen Primitive kann Träger *geometrischer Attribute* sein. Zum Beispiel kann eine Oberflächenfacette gekrümmt sein, und auch die Position eines Knotens im Raum wird durch geometrische Zusatzattribute, in diesem Fall durch die Koordinaten des Knotens, angegeben.

Die Berandung des Körpers zu beschreiben bedeutet aber mehr, als nur die geometrische Gestalt der einzelnen Primitive durch Atrribute zu beschrieben.

Eine wesentliche Rolle spielen die Nachbarschaftsbeziehungen zwischen den topologischen Primitiven. Die Nachbarschaftsbeziehungen werden auch als *Adjazenzrelationen* bezeichnet. Eine typische Nachbarschaftsbeziehung wäre beispielsweise die zwischen einem Knoten und dem von ihm ausstrahlenden Kanten. Ebenso häufig werden wir uns beispielsweise dafür interessieren, welche Knoten auf dem Rand einer Fläche liegen, oder welche Flächen durch eine gegebene Kante verbunden sind.

Da insgesamt 5 Typen topologischer Primitive vorliegen, sind insgesamt 25 Nachbarschaftsbeziehungen möglich. In der folgenden Tabelle sind die 9 möglichen Beziehungen der Primitive *vertex*, *edge* und *face* aufgelistet:

	v	e	f
v	Nachbarn des Knotens	Kanten, die vom Knoten ausgehen	Flächen, die am Knoten aneinanderstoßen
e	Knoten, die zu der	Kanten, die an die	Flächen, die an die Kante

	Kante gehören	Kante anschließen	anschließen
f	Knoten, die zum Rand der Fläche gehören	Kanten, die zum Rand der Fläche gehören	Flächen, die an die Fläche anschließen

Die allgemeinste Methode, gegenseitige Beziehungen von Objekten zu beschreiben, und somit ein Schlüssel zur *B-Rep*-Modellierung, ist die *Graphentheorie*. Ihr wollen wir uns deswegen zunächst kurz zuwenden.

2.6.1 Graphentheorie

Ein Graph setzt sich aus einer Menge von *Ecken E* und einer Menge von *Pfeilen* (*Kanten*) *K* zusammen. Jede Kante ist dabei durch zwei Ecken ("Endpunkte") definiert. Die Objekte, die in der Ecken- und Pfeilmenge enthalten sind, müssen nichts mit geometrischen Objekten zu tun haben. Vielmehr ist die Graphentheorie ein Hilfsmittel zur Beschreibung beliebiger Beziehungen zwischen Objekten beliebigen Typs. Aufgaben, die mit der Graphentheorie gelöst werden können, sind z.B.:

- Ist in einem System von vernetzten Rechnern Rechner *B* auch dann noch von Rechner *A* aus erreichbar, wenn eine der Netzwerkverbindungen gekappt wird (Erreichbarkeitsproblem)?
- Suche in einem Straßennetz die kürzeste Verbindung von *A* nach *B* (Wegsuche).
- Ermittle die minimale Bauzeit eines Bauprojektes (Terminplanung).

Im einzelnen unterscheidet man sogenannte *ungerichtete* Graphen, d.h. Graphen, in denen die Richtung der Pfeile (z.B. von Ecke *a* nach Ecke *b* oder umgekehrt) uninteressant ist, von *gerichteten* Graphen (*Digraphen=directed graphs*). Die Richtung der Pfeile spielt z.B. bei der Beschreibung eines Straßennetzes, das auch Einbahnstraßen enthält, eine Rolle. Als Beispiele gerichteter Graphen haben wir bereits Binärbäume, Quad- und Octrees kennengelernt.

Graphen werden *anschaulich* durch Linienzüge dargestellt. In vielen Anwendungen ist es notwendig, die *Kanten* eines Graphen zu *bewerten*. Beispielsweise setzt sich ein Rohrleitungsnetz im allgemeinen nicht aus lauter Rohren gleichen Durchmessers zusammen. Der Rohrdurchmesser spielt also eine wichtige Rolle als *Attribut* der Kanten. Viele Graphenalgorithmen verwenden solche *Attribute* zu Kanten. Man spricht von *bewerteten* Graphen.

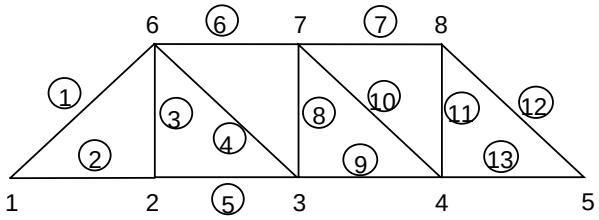

Bild 2.16: Beispiel eines Graphen. Die Knoten sind durch einfache Nummern, die Pfeile durch Nummern in Kreisrahmung gekennzeichnet.

Im *Rechner* können Graphen durch verschiedene *Datenstrukturen* beschrieben werden. Im einfachsten Fall läßt sich ein Graph durch eine *Matrix* (sogenannte *Adjazenzmatrix*) beschreiben, in der jedem Knoten eine Spalte und eine Zeile zugeordnet ist.

In Bild 2.16 ist als Beispiel ein Graph abgebildet. Seine Darstellung als *Adjazenzmatrix* sieht aus wie folgt:

Knoten	1	2	3	4	5	6	7	8
1	1	1				1		
2	1	1	1			1		
3		1	1	1		1	1	
4			1	1	1		1	1
5				1	1			1
6	1	1	1			1	1	
7			1	1		1	1	1
8				1	1		1	1

Alle Felder der Matrix, die leer gelassen sind, sind mit einer Null besetzt. Alle Felder, die eine 1 enthalten, stellen eine Verknüpfung zwischen den beiden zugehörigen Knoten her. Der in Bild 2.16 dargestellte Graph ist ungerichtet, so daß jede Kante zweimal auftaucht, jeweils in beiden Richtungen. Die Adjazenzmatrix eines ungerichteten Graphen ist, wie man hier sieht, stets symmetrisch. Es würde also ausreichen, nur die obere oder untere Hälfte der Matrix zu speichern.

Im Fall eines bewerteten Graphen könnte man anstelle der Einsen die Bewertung der Pfeile in die Adjazenzmatrix eintragen, z.B. die Längen der Wege zwischen den Knoten.

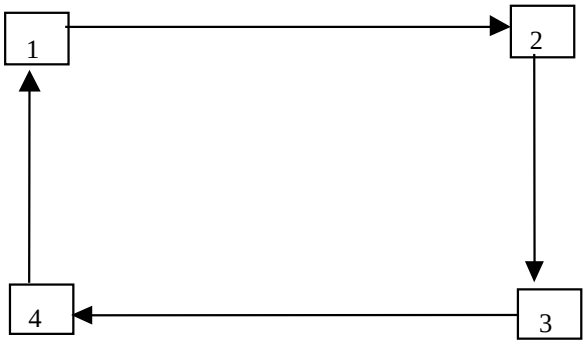

Bild 2.17: Beispiel zum Erreichbarkeitsproblem.

Die Adjazenzmatrix läßt eine sehr einfache Lösung des Erreichbarkeitsproblems zu: Wenn man für einen Graphen mit einer Knotenanzahl v die Adjazenzmatrix A aufstellt, so stellt die v-te Potenz dieser Matrix die Adjazenzmatrix der *transitiven Hülle* des Graphen dar. In der Adjazenzmatrix der transitiven Hülle sind alle Einträge ungleich Null, die einer direkten oder *indirekten* Verbindung zwischen zwei Knoten entsprechen. In unserem in Bild 2.16 gezeigten Graphen sind alle Knoten indirekt mit jedem anderen Knoten verbunden. Die Adjazenzmatrix der transitiven Hülle enthält daher keine Nullen mehr. Bei einem solch kleinen Beispiel kann man das noch recht schnell direkt sehen. Für die allgemeine Untersuchung solcher Fragestellungen bei Graphen mit einer großen Anzahl von Ecken und Pfeilen ist die schematische Berechnung der transitiven Hülle ein wichtiges Hilfsmittel. Die Berechnung über die v-te Potenz der Adjazenzmatrix ist allerdings nicht sehr effizient, da sie den Aufwand v^4 verursacht. Es gibt jedoch einen Algorithmus, der diese Aufgabe in weniger als v^3 Schritten löst.

Wir wollen diesen Algorithmus anhand des Beispiels in Bild 2.17 darstellen. Wir wollen alle Knoten j – von 1 bis 4 – der Reihe nach untersuchen. Wir schreiben jeweils die Knoten i an, von denen aus man zu Knoten j kommen kann, und alle Knoten k, die von j aus erreichbar sind. Offenkundig sind diese beiden Knotenmengen indirekt über Knoten j verbunden. Unsere Analyse läuft in folgenden Schritten ab: Zunächst untersuchen wir Knoten 1. Wir finden:

Knoten 1:

erreichbar von:	verbunden mit:
4	2

Also können wir nunmehr auch die Verbindung von 4 nach 2 in unseren Graphen aufnehmen. Er sieht damit aus wie folgt (Bild 2.18):

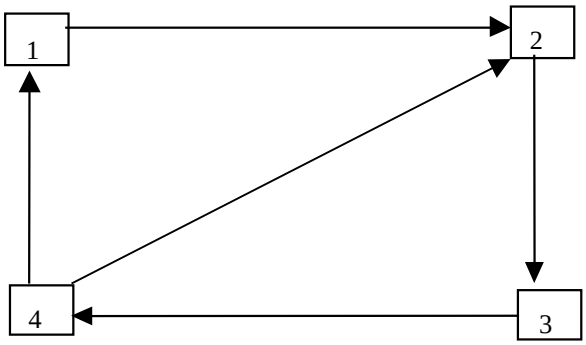

Bild 2.18: Schrittweise Ermittlung der transitiven Hülle des Graphen aus Bild 2.17. Zustand nach Untersuchung von Knoten 1.

Nun gehen wir weiter zu Knoten 2. Dort finden wir nunmehr:

Knoten 2:

erreichbar von:	verbunden mit:
1	3
4	

Wir können also auch die Pfeile 1-3 und 4-3 in unseren Graphen eintragen. Er erhält damit folgende Gestalt (Bild 2.19):

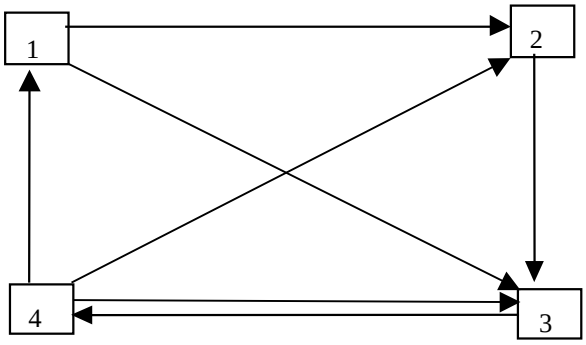

Bild 2.19: Schrittweise Ermittlung der transitiven Hülle des Graphen aus Bild 2.17. Zustand nach Untersuchung von Knoten 2.

Nun gehen wir weiter zu Knoten 3. An dieser Stelle können wir für unseren erweiterten Graphen nun feststellen:

Knoten 3:

erreichbar von:	verbunden mit:
1	4
2	
4	

Das erbringt uns die neuen Pfeile 1-4 und 2-4. Der Pfeil 4-4 entspricht der 1 auf der Diagonale der zugehörigen Adjazenzmatrix. Unser Graph hat nun folgende Gestalt (Bild 2.20):

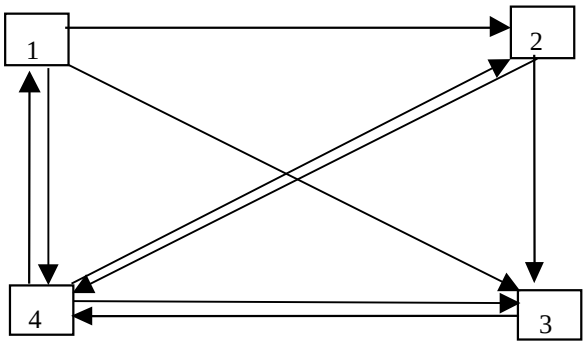

Bild 2.20: Schrittweise Ermittlung der transitiven Hülle des Graphen aus Bild 2.17. Zustand nach Untersuchung von Knoten 3.

Schließlich müssen wir auch noch Knoten 4 untersuchen. Dort finden wir:

Knoten 4:

erreichbar von:	verbunden mit:
1	1
2	2
3	3

Damit ergeben sich noch die neuen Pfeile 2-1, 3-1 und 3-2, so daß sich die transitive Hülle unseres Graphen als voll besetzter Graph, in dem jeder Knoten mit jedem anderen verbunden ist, herausstellt.

Die Vorgehensweise, die wir dargestellt haben, läßt sich mit Hilfe der Adjazenzmatrix *A* in folgendem Algorithmus wiedergeben:

```
for j=1 to Knotenanzahl
   for i=1 to Knotenanzahl
     if A(i,j)=1 then
       for k=1 to Knotenanzahl
         if A(j,k)=1 then A(i,k)=1
       next k
     end if
   next i
next j
```

Wie man leicht einsieht, benötigt dieser Algorithmus infolge der drei geschachtelten Schleifen, deren innerste aber nicht jedesmal durchlaufen wird, maximal (`Knotenanzahl`)^3 Operationen. Die äußerste Schleife entspricht der Untersuchung aller Knoten. Die nächste Schleife sucht alle jene Knoten heraus, die von Knoten *j* aus erreichbar sind, ermittelt also jeweils die 1. Spalte unserer Tabellen. Die innerste Schleife schließlich ermittelt die rechte Spalte unserer Tabellen und führt die neuen Verbindungen ("Abkürzungen" von *i* nach *k*) ein.

Für Graphen, die zwar viele Knoten, aber nur wenige Pfeile enthalten, ist die Adjazenzmatrix keine besonders effiziente Datenstruktur. Die Darstellung eines Graphen als Adjazenzmatrix hat aber den Vorteil, daß man sehr einfach alle von einem Knoten ausgehenden und vor allem auch die an einem Knoten endenden ("*inzidenten*") Pfeile ermitteln kann. Letzteres ist für unseren Algorithmus "transitive Hülle" entscheidend.

Einige andere wichtige Datenstrukturen für Graphen sind zwar wesentlich weniger speicherintensiv, lassen aber dafür keine so einfache Ermittlung inzidenter Pfeile zu.

Sehen wir uns zunächst die Darstellung eines Graphen als *Kantenliste* an (Bild 2.21). Der Graph hat dieselbe Anzahl von Knoten und Pfeilen wie der in Bild 2.16, ist aber nun ein *gerichteter* Graph.

Kante	von	nach
1	1	6
2	1	2
3	2	6
4	3	6
5	2	3
6	6	7
7	7	8
8	3	7
9	3	4
10	4	7
11	4	8
12	5	8
13	4	5

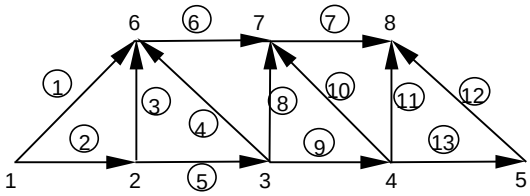

Bild 2.21: Darstellung eines Graphen durch eine Kantenliste.

Alle Kanten sind in einer Liste bzw. Tabelle aufgeführt. Jede Kante verweist per Nummer auf eine Anfangs- und Endecke. Die Nachbarschaftsrelationen des Graphen sind durch diese Darstellung in einer einzigen Liste vollständig beschrieben. Auch die geometrischen und sonstigen Zusatzattribute der Kante (Form, Strichstärke, Linienart, Farbe) können wir durch Hinzufügen weiterer Spalten zur Kantentabelle in dieser Datenstruktur unterbringen. Wir wollen jedoch auch noch geometrische Attribute zu den *Ecken* speichern, z.B. Koordinaten. Daher benötigen wir in der Regel wie bei der Datenstruktur Adjazenzliste noch ein zweites Feld, in dem diesmal die Knoten nach Nummer sortiert mit ihren Attributen aufgeführt sind.

Die Speicherung des Graphen als Kantenliste hat den Nachteil, daß es recht aufwendig ist, die von einem Knoten ausgehenden Pfeile bzw. die in einem Knoten inzidenten Pfeile zu ermitteln. Für Wegalgorithmen oder Erreichbarkeitsprobleme ist diese Datenstruktur also nicht besonders günstig.

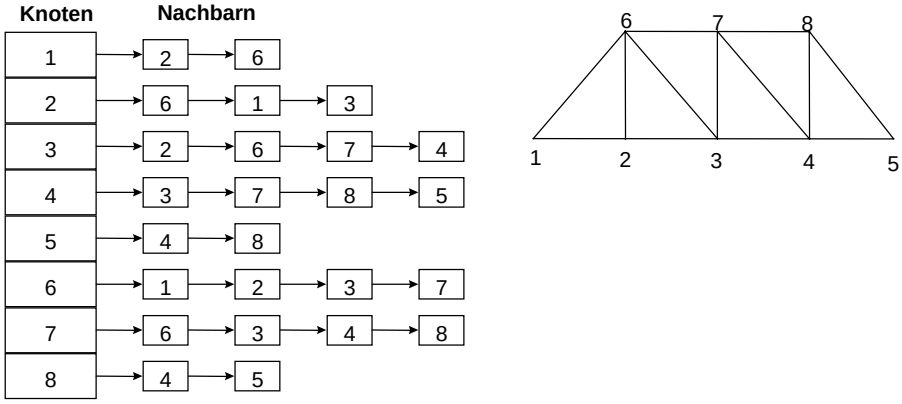

Bild 2.22: Darstellung eines Graphen durch eine Adjazenzliste (Nachbarschaftsliste).

Eine Datenstruktur, die den enormen Speicheraufwand der Adjazenzmatrix vermeidet und dennoch einen effizienten Zugriff auf einen Teil der Nachbarschaftsinformation gestattet, ist die *Nachbarschaftsliste* (*Adjazenzliste*, Bild 2.22). Dabei verwaltet jede Ecke des Graphen eine Liste aller Ecken, mit denen sie direkt durch eine Kante verbunden ist. In unserem Bild haben wir wiederum einen ungerichteten Graphen gewählt. In diesem Fall kommt jede Kante doppelt vor, einmal als Nachbarschaftsbeziehung von Ecke *a* nach Ecke *b* und einmal als Verbindung von *b* nach *a*.

Im Gegensatz zur Darstellung "Kantenliste" macht die Datenstruktur "Adjazenzliste" die Ermittlung der von einem Knoten ausstrahlenden Pfeile recht einfach. Die Ermittlung inzidenter Pfeile ist allerdings sogar noch aufwendiger als bei der Kantenliste.

Je nach Einsatzzweck des Graphen muß man eine geeignete Darstellungsform wählen; ein generelles Rezept läßt sich nicht angeben.

2.6.2 Anwendung auf die geometrische Modellierung

Wir greifen noch einmal die Tabelle aus der Einleitung zu Kapitel 2.6 auf:

	v	e	f
v	Nachbarn des Knotens	Kanten, die vom Knoten ausgehen	Flächen, die am Knoten aneinanderstoßen
e	Knoten, die zu der Kante gehören	Kanten, die an die Kante anschließen	Flächen, die an die Kante anschließen
f	Knoten, die zum Rand der Fläche gehören	Kanten, die zum Rand der Fläche gehören	Flächen, die an die Fläche anschließen

Zur geometrischen Modellierung eines Objektes mittels der B-Rep-Methode gehört es, die Nachbarschaftsrelationen zwischen den einzelnen topologischen Primitiven zu beschreiben. Die Beschreibung von Nachbarschaftsrelationen kann mit Hilfe von Graphen erfolgen, wie wir gesehen haben. Jede Zelle unserer Tabelle könnten wir also durch Einsatz eines *Graphen* ausfüllen.

Eine solche Vorgehensweise wäre aber nicht sehr sinnvoll, denn der Speicherplatzbedarf wäre schon bei einfachen Strukturen gewaltig.

Eine erste Idee, solche Probleme zu vermeiden und dennoch die Graphentheorie einzusetzen, besteht darin, alle topologischen Primitive bis auf die Knoten außer Acht zu lassen und nur noch die topologische Relation v-v zu speichern.

Damit kommen wir zu einem sogenannten *Drahtmodell*. Die Grundidee des Ansatzes "Drahtmodell" besteht also darin, die gesamte geometrische Information in den Eckpunkten des Körpers zu konzentrieren, also jedem Eckpunkt einen Ortsvektor im dreidimensionalen Raum zuzuordnen, und die Topologie des Objektes auf die Relation v-v zu reduzieren.

In diesem Modell treten Oberflächen und Volumina also nicht auf, die geometrische Beschreibung des Objektes wird auf die Speicherung der Verschneidungskanten von ebenen Oberflächen reduziert.

Drahtmodell

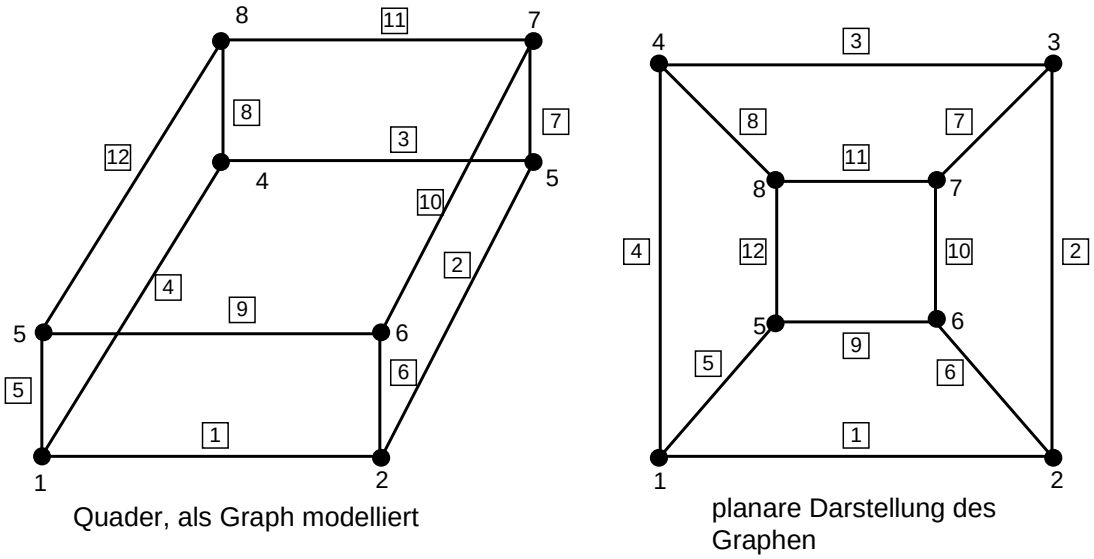

Bild 2.23: Beschreibung eines "Quaders" durch einen ungerichteten v-v-Graphen ("Drahtmodell"). Beide gezeigten Darstellungen des Quaders sind äquivalent! Aus der Datenstruktur kann nicht entnommen werden, ob ein Kantenzug eine Fläche umschließt oder nicht.

Zur Speicherung des Drahtmodells können wir die Strategien, die wir bei der Beschreibung von Graphen kennengelernt haben, unverändert übernehmen.

Bild 2.23 zeigt links eine "perspektivische" Darstellung eines Quaders als Drahtmodell. Für den Betrachter ist sofort intuitiv erkennbar, daß es sich bei dem modellierten Objekt um einen Quader handeln soll, nicht aber für den Rechner! Dies wird durch die rechte Darstellung in Bild 2.23 demonstriert, die genau dieselbe Information trägt wie die linke Darstellung. Der Graph aus dem linken Bild ist lediglich in eine "planare", also ebene, überschneidungsfreie Darstellung überführt worden, gibt aber nach wie vor dieselben Nachbarschaftsrelationen (z.B. Kante 5 verbindet Knoten 1 und 5) wieder.

Aus dieser Datenstruktur kann nicht ermittelt werden, ob die Kanten 1, 5, 6 und 9 eine Fläche umschließen oder nicht, ob der Quader eine "hohle Schachtel" ist oder ein ausgefülltes Volumen, ob eine oder mehrere Flächen offen sind oder nicht, usw. Wir können vom Rechner nicht automatisch bestimmen lassen, ob sich ein Punkt im Inneren, auf der Oberfläche oder außerhalb eines Körpers befindet.

Da die Flächen- und Volumeninformationen gänzlich fehlen, kann ein solches Modell auch nicht zur automatisierten Generierung von perspektivischen Ansichten oder Schnitten eingesetzt werden, weil es dann ja eine Rolle spielt, ob man auf einer Seite in den Quader "hineinsehen" kann oder nicht, usw. (Allenfalls kann man anhand der geometrischen Daten der Eckpunkte ermitteln, ob in einer perspektivischen Ansicht eine Kante "vor" oder "hinter" einer anderen liegt, und die hintere Kante dann für ein kurzes Stück unterbrechen. Dies entspricht der Unterscheidung zwischen einem Quader, bei dem Ecke 6 am nächsten am Betrachter liegt, und einem zweiten Quader mit gleicher graphischer Darstellung, bei dem jedoch Knoten 4 dem Betrachter am nächsten liegt. Diese Information kann aus Bild 2.23 allein ebenfalls nicht entnommen werden, sondern macht die Analyse der dreidimensionalen Ortsvektoren der Eckpunkte 6 und 4 nötig!).

Da heute CAD-Programme im Zentrum einer Vielzahl von Anwendungen und Nutzungen stehen, die Dienste wie die automatische Volumenberechnung oder Generierung von Schnitten erforderlich machen, sind Drahtmodelle als Basis eines CAD-Systems heute unzureichend.

Will man an die Stelle des Drahtmodells eine umfassendere und ergiebigere Beschreibung von 3D-Gebilden setzen, so bietet sich eine Reihe von Alternativen.

Zunächst sollten wir berücksichtigen, daß wir keineswegs *alle* topologischen Relationen aus unserer Tabelle direkt speichern müssen. Vielmehr können wir, wenn unser Modell einen Teil der Relationen speichert, die übrigen Relationen unter Umständen *berechnen*. Kennen wir beispielsweise die Relation f-e (alle Kanten einer Fläche) und die Relation e-v (Start- und Endknoten einer Kante), so können wir auch die Relation f-v (alle Knoten einer Fläche) berechnen. Relationen, die wir selten benötigen, werden wir also nicht direkt speichern, sondern aus anderen Relationen berechnen.

Ferner werden wir uns bemühen, bevorzugt solche Relationen zu speichern, die sehr einfach zu speichern sind. Beispielsweise hat eine Kante eines geometrischen Gebildes immer eine Beziehung zu zwei Knoten, während die Beziehung Knoten-Kanten oder Flächen-Kanten wesentlich komplizierter zu speichern ist.

Man beachte, daß nicht jede formal mögliche Zusammensetzung von Relationen in der oben angegebenen Tabelle auch tatsächlich das erwünschte Ergebnis liefert. Zum Beispiel erhält man durch Zusammensetzen (Skalarprodukt) der Relationen f-v und v-f *nicht* die Relation f-f. Vielmehr werden durch f-v $\times$ v-f viel mehr Flächen miteinander verknüpft als durch f-f! Man muß also durchaus aufpassen, welche Relationen man für die direkte Speicherung auswählt. Die Relationen über die Kanten bieten sich hierfür an.

Außerdem kann man sich die geometrische Modellierung natürlich leichter machen, indem man die Vielfalt der mit einer Datenstruktur darstellbaren Objekttypen beschränkt. Am leichtesten fällt die geometrische Modellierung mit B-Rep-Techniken, wenn man sich auf echte Polyeder beschränkt.

Für jeden echten Polyeder gilt folgende Beziehung (sogenannte EULER-*Formel*):

$$v + f - e = 2$$

Dabei bezeichnet *v* die Anzahl der Knoten der Struktur, *e* die Anzahl der Kanten und *f* die Anzahl der Facetten. Für allgemeine Körper mit Löchern läßt sich eine ähnliche Formel aufstellen.

Die Oberflächen von einfachen Polyedern sind sogenannte 2-*Mannigfaltigkeiten*. Dies sind, informell gesprochen, Oberflächen, bei denen sich die Umgebung jedes beliebigen Punktes der Oberfläche bijektiv in eine "Tangentialebene" abbilden läßt, also gewissermaßen "abwickelbare" Flächen.

Berühren sich zwei Körper nur in einer einzigen Ecke oder Kante, so ist die resultierende Gesamtoberfläche jedoch keine 2-Mannigfaltigkeit mehr.

2.5.3 Winged-Edge-Datenstruktur

Für geometrische Gebilde, deren Berandung die Forderungen einer 2-Mannigfaltigkeit erfüllt, gibt es eine sehr schöne und effiziente Datenstruktur, die *Winged-Edge-Datenstruktur* (nach B. G. Baumgart, "Geometric Modeling for Computer Vision", Stanford University, Stanford, CA, 1974).

In der reinen Form der Winged-Edge-Datenstruktur sind *mehrfach zusammenhängende* Körper, also Polyeder mit Löchern, nicht zugelassen.

Den Flächen wird ein *Umlaufsinn* zugeordnet, der so definiert ist, daß der zugehörige Drehvektor in dieselbe Richtung weist wie die aus dem Körper heraus zeigende Flächennormale. Von außen (außerhalb des Körpers) betrachtet, ist der positive Umlaufsinn der Fläche also der Gegenuhrzeigersinn.

Jeder Kante wird eine willkürlich gewählte Richtung zugeordnet. Somit kann jede Kante von einer Fläche benutzt werden, deren Umlaufsinn mit der Kante übereinstimmt, und von einer Fläche, deren Umlaufsinn der Kantenrichtung entgegengerichtet ist.

Die Winged-Edge-Datenstruktur verwendet nur die Elemente *vertex*¸ *edge* und *face*. Der besondere Trick besteht nun darin, die Nachbarschaftsrelationen zwischen diesen Objekten derart zu speichern, daß die B-Rep-Darstellung ausschließlich mit *Tabellen mit fester Spaltenanzahl* zu bewerkstelligen ist. Alle Relationen, die nicht direkt gespeichert werden können, sollen außerdem möglichst effizient berechnet werden können.

Bild 2.24 zeigt als Beispiel die Modellierung eines Quaders in der Winged-Edge-Datenstruktur. Wir benötigen drei Tabellen: eine Kantentabelle, eine Knotentabelle und eine Flächentabelle.

Die Kantentabelle enthält e-v-, e-f- und einen Teil der e-e-Beziehungen. Für jede Kante werden insgesamt vier Verweise auf andere Kanten gespeichert: Jede Kante kennt einen "Nachfolger im Gegenuhrzeigersinn" (*eccwn=counterclockwise next edge*), einen "Nachfolger im Uhrzeigersinn" (*ecwn=clockwise next edge*), einen "Vorgänger im Gegenuhrzeigersinn" (*eccwp=counterclockwise previous edge*) und einen "Vorgänger im Uhrzeigersinn" (*ecwp=clockwise previous edge*). Außerdem wird bei der Kante ein Verweis auf den Anfangsknoten (*vs*) und den Endknoten (*ve*) gespeichert; dadurch ist eine Kantenrichtung definiert. Desweiteren verwaltet die Kante zwei Verweise auf Flächen, nämlich einen Verweis auf die Fläche, die in Kantenrichtung gesehen "links" angrenzt (*fccw=counterclockwise face*), und einen Verweis auf die in Kantenrichtung "rechts" anschließende Fläche (*fcw=clockwise face*). Man macht sich also zunutze, daß es bei einer 2-Mannigfaltigkeit keine Kanten geben kann, an die mehr als zwei Flächen anschließen.

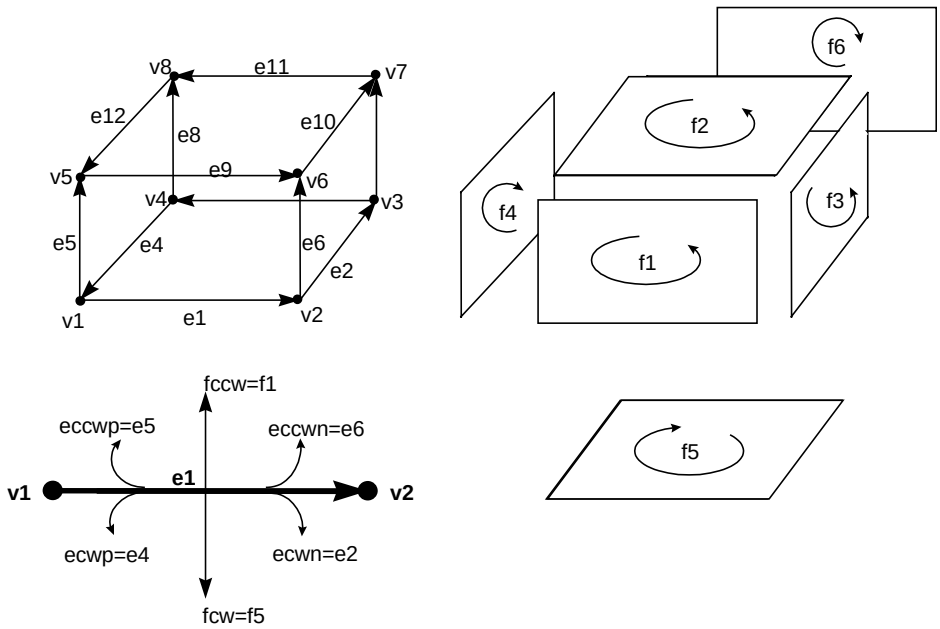

Bild 2.24: Darstellung eines Quaders in der Winged-Edge-Datenstruktur

Diese selektiv gespeicherten Nachbarschaftsbeziehungen der Kante müssen nun nur noch durch einen minimalen Satz von Nachbarschaftsbeziehungen für die Knoten und Flächen ergänzt werden, um die Ermittlung aller topologischen Relationen zu ermöglichen:

Es genügt, in der *Knotentabelle* für jeden Knoten nur *einen einzigen* Verweis auf *irgendeine* Kante zu speichern, die von dem betreffenden Knoten ausgeht oder an ihm endet. Ebenso genügt es, für jede Fläche einen Verweis auf *irgendeine* Kante ihres Randes zu speichern. Wir bezeichnen den jeweils gespeicherten Verweis mit *es* ("Startkante").

In den nachfolgenden Tabellen sind die Verweise für alle topologischen Primitive des Beispiels in Bild 2.24 angegeben:

e	*vs*	*ve*	*eccwn*	*ecwn*	*eccwp*	*ecwp*	*fccw*	*fcw*
1	*v1*	*v2*	*e6*	*e2*	*e5*	*e4*	*f1*	*f5*
2	*v2*	*v3*	*e7*	*e3*	*e6*	*e1*	*f3*	*f5*
3	*v3*	*v4*	*e8*	*e4*	*e7*	*e2*	*f6*	*f5*
4	*v4*	*v1*	*e5*	*e1*	*e8*	*e3*	*f4*	*f5*
5	*v1*	*v5*	*e12*	*e9*	*e4*	*e1*	*f4*	*f1*
6	*v2*	*v6*	*e9*	*e10*	*e1*	*e2*	*f1*	*f3*
7	*v3*	*v7*	*e10*	*e11*	*e2*	*e3*	*f3*	*f6*
8	*v4*	*v8*	*e11*	*e12*	*e3*	*e4*	*f6*	*f4*
9	*v5*	*v6*	*e10*	*e6*	*e12*	*e5*	*f2*	*f1*
10	*v6*	*v7*	*e11*	*e7*	*e9*	*e6*	*f2*	*f3*
11	*v7*	*v8*	*e12*	*e8*	*e10*	*e7*	*f2*	*f6*
12	*v8*	*v5*	*e9*	*e5*	*e11*	*e8*	*f2*	*f4*

v	*es*
1	*e1*
2	*e2*
3	*e3*
4	*e4*
5	*e5*
6	*e6*
7	*e7*
8	*e8*

f	*es*
1	*e1*
2	*e9*
3	*e2*
4	*e12*
5	*e1*
6	*e11*

Die Winged-Edge-Datenstruktur speichert also die Beziehung e-v als "Adjazenzliste" (die Listenlänge beträgt hier immer zwei!). Auch die Beziehung v-e ist als Adjazenzliste gespeichert, wenngleich dies nicht so offensichtlich ist, da die Verkettung der Kanten an einem Knoten implizit über die Kantenliste erfolgt. Auch die Beziehung f-e ist als Adjazenzliste gegeben, wobei wiederum die Kantentabelle geschickt ausgenutzt wird.

Mit Hilfe der in der Winged-Edge-Datenstruktur gespeicherten Informationen kann man die übrigen topologischen Relationen des Modells *berechnen*: Will man beispielsweise alle Begrenzungskanten der Fläche *f1* ermitteln, so geht man vor wie folgt: Die Fläche *f1* verweist auf die Startkante *e1*. Dies ist bereits die erste Berandungskante von *f1*. Wollen wir nun z.B. alle weiteren Berandungskanten im Uhrzeigersinn bestimmen, so müssen wir zunächst ermitteln, ob die Kante *e1* in der Richtung des Umlaufsinns von *f1* liegt oder in die Gegenrichtung weist. Dazu prüfen wir, ob *f1* die *fccw*-Fläche oder die *fcw*-Fläche von Kante *e1* ist. Ist *f1* die *fcw*-Fläche von *e1*, so ist *e1* entgegen dem Umlaufsinn von *f1* orientiert. In unserem Fall ist *f1* jedoch die *fccw*-Fläche von *e1*, und somit wissen wir, daß *e1 in Richtung* des Umlaufsinns von *f1* weist. Um die nächste Berandungskante von *f1* zu finden, folgen wir daher dem *eccwn*-Verweis der Kante *e1* und finden so als nächste Randkante von *f1* aus der Kantentabelle die Kante *e6*. Diese weist wiederum in Richtung des Umlaufsinns der Fläche *f1*, so daß wir wiederum dem *eccwn*-Verweis folgen können und so die Kante *e9* finden. Diese Kante verweist allerdings *entgegen* dem Umlaufsinn, der für die Fläche *f1* definiert ist, wie wir anhand des Umstands erkennen, daß die Fläche *f1* die *fcw*-Fläche der Kante *e9* ist. Somit dürfen wir jetzt nicht weiter dem *eccwn*-Verweis folgen, sondern müssen nun dem Verweis *ecwp* der Kante *e9* folgen, um die nächste Randkante von *f1* zu ermitteln. So finden wir die Kante *e5*. Diese ist wiederum entgegen dem

Umlaufsinn von *f1* orientiert, so daß wir wieder mit dem *ecwp*-Verweis fortfahren müssen. Im *ecwp*-Verweis von *e5* finden wir die Kante *e1*. Diese Kante haben wir allerdings schon in unserer Sammlung der Randkanten der Fläche *f1*; daran erkennen wir, daß wir die Fläche einmal ganz umrundet haben und also fertig sind.

Ganz ähnlich lassen sich auch Algorithmen zur Bestimmung aller Kanten, die an einem Knoten zusammentreffen, oder aller Knoten, die auf dem Rand einer bestimmten Fläche liegen, bestimmen. Alle diese Anfragen können so ähnlich wie oben dargestellt *rekursiv* beantwortet werden. Dem Leser sei empfohlen, selbst einige solche Beispiele durchzuspielen. Der Gesamtaufwand zur Bewältigung aller dieser Anfragen an die Datenstruktur hängt lediglich linear von der Anzahl der gefundenen Kanten bzw. Knoten ab, nicht jedoch von der Gesamtanzahl der Knoten, Kanten und Flächen. Auch der Speicheraufwand wächst nur linear mit der Gesamtanzahl topologischer Primitive im System.

Seltener benötigte Relationen, z.B. v-f, machen etwas mehr Aufwand: Für die v-f-Relation müssen wir zunächst die v-e-Relation auswerten, und für alle damit gefundenen Kanten dann die Beziehung e-f. Dabei erhalten wir jedoch alle *faces* doppelt, so daß ein Zusatzaufwand zum Heraussortieren der doppelten Angaben notwendig ist. Ähnliches gilt auch für die Beziehung e-e. Sie ist in der Winged-Edge-Datenstruktur nur teilweise direkt gespeichert, und um alle Nachbarkanten zu erhalten, müssen wir über e-v auf die v-e-Beziehung zugreifen.

Bei anderen B-Rep-Datenstrukturen, die beispielsweise mehr oder weniger topologische Relationen direkt speichern, wächst entweder der Rechenaufwand für einzelne Nachbarschaftsanfragen übermäßig an, oder der Speicheraufwand wird erheblich größer als bei der Winged-Edge-Datenstruktur.

Alle diese Vorteile haben wir uns freilich mit einigen wesentlichen Einschränkungen der darstellbaren geometrischen Objekte erkauft. Meist werden im CAD auch Körper benötigt, in deren Oberfläche oder Innerem Löcher enthalten sind. Solche Objekte werden als *mehrfach zusammenhängend* bezeichnet.

Will man auch Körper mit mehrfach zusammenhängenden Flächen beschreiben, so muß die Winged-Edge-Datenstruktur erweitert werden. Man führt dazu ein neues, künstliches topologisches Primitiv ein, den *loop*: Ein *loop* beschreibt einen geschlossenen Kantenzug, der zum Rand einer *face* gehört. Jede Fläche (*face*) besitzt mindestens einen *äußeren loop*, gegebenenfalls aber noch zusätzliche innere *loops*, also Löcher. Jede Kante wird von zwei *loops* benutzt, von denen je einer zu jeder der an die Kante anschließenden Flächen gehört.

In einer erweiterten Winged-Edge-Datenstruktur, die auch mehrfach zusammenhängende Körper zuläßt, ersetzt man die Flächen-Kanten-Tabelle durch eine Loop-Kanten-Tabelle. Die Zuordnung zwischen Flächen und Loops wird in einer eigenen Tabelle festgehalten. Dabei entsteht allerdings der Nachteil, daß man entweder eine Tabelle variabler Spaltenzahl (*face-loop*-Tabelle) erhält, oder aber bei der Abfrage "Gib alle *loops* der Fläche" eine unter Umständen große Tabelle durchsuchen muß (*loop-face*-Tabelle).

Es sei noch erwähnt, daß man die Winged-Edge-Datenstruktur nicht nur zur Beschreibung der *Oberfläche* dreidimensionaler Körper einsetzen kann, sondern auch zur Beschreibung *netzartiger ebener* Strukturen zweckentfremden kann. Man betrachtet dazu einfach die ebene netzartige Struktur als "Abwicklung" eines räumlichen Gebildes. Auch in einem ebenen netzartigen Gebilde schließen an jede Kante maximal zwei Flächen an. Für die *topologische* Beschreibung ist es völlig unerheblich, ob das dargestellte Gebilde die Oberfläche eines dreidimensionalen Objektes ist oder nicht.

So einfach es ist, ein Objekt mit Hilfe der Winged-Edge-Datenstruktur zu beschreiben, so schwierig ist es, Konstruktionsoperatoren zu schreiben, die die Winged-Edge-Datenstruktur manipulieren. Man kann selbstverständlich nur solche Operationen durchführen, deren Ergebnis wiederum in einer Winged-Edge-Datenstruktur darstellbar ist. Solche Operationen überführen also ein Objekt, das der EULER-Formel genügt, wiederum in ein Objekt, das dieser Formel gehorcht. Operatoren, die diese Bedingung erfüllen, werden als EULER-Operatoren bezeichnet.

Kompliziertere Operationen, wie sie in einem CAD-System bereitstehen sollten, muß man dann aus den elementaren EULER-Operatoren zusammenbauen. Ein Problem dabei ist, daß sich nicht notwendigerweise alle Zwischenstadien einer beliebigen Konstruktion als EULER-Polyeder darstellen lassen. Es läßt sich aber zeigen, daß es immer einen Konstruktionsweg gibt, der ausschließlich mit EULER-Operatoren auskommt, wenn das Endergebnis in der Einged-Edge-Struktur darstellbar ist.

Will man zu Nichtmannigfaltigkeiten übergehen, also auch gegenseitige Berührungen von Körpern längs einer Kante oder an einem einzelnen Knoten zulassen, so sind kompliziertere Datenstrukturen notwendig. Ein Beispiel für eine solche Datenstruktur ist die *Radial-Edge-Datenstruktur*, die dem Modellierkern *ACIS* des kommerziellen CAD-Programms AutoCAD ab Version 13 zugrundeliegt. Details dieser Datenstruktur zu besprechen, würde den Rahmen dieser Vorlesung sprengen; der Leser sei auf die Arbeit "Topological Structures for Geometric Modeling" von Kevin Weiler, Rensselaer Polytechnic Institute, Troy, NY, 1986 verwiesen.

Der Übergang von anderen Modelltypen zu Boundary-Representation-Modellen ist alles andere als einfach. Beispielsweise entstehen bei Anwendung BOOLEscher Operationen auf Oberflächen Schnittkurven höherer Ordnung, die nicht notwendigerweise mit den Mitteln des gegebenen B-Rep-Modells dargestellt werden können. Auch aus einer Zellzerlegungsdarstellung kann man nur schwer eine B-Rep-Darstellung ableiten. Da für photorealistische Darstellungen, z.B. mit Spiegelungen, Oberflächendarstellungen fast unverzichtbar sind, stößt man hier schnell auf schwer zu überwindende Probleme.

Ein Ausweg in kommerziellen CAD-Programmen besteht darin, daß *mehrere* Modelliersysteme nebeneinander bestehen. So kann der Benutzer die Struktur beispielsweise mit CSG-Techniken oder über Oberflächenbeschreibungen eingeben, intern aber hält das System zusätzlich auch noch eine entsprechende Zellzerlegungs- oder Quadtree-Darstellung. Dabei entsteht die Schwierigkeit, daß nunmehr dieselben Informationen gleichzeitig in unterschiedlicher Form gehalten werden, und daß die verschiedenen Modelle stets konsistent gehalten werden müssen (Problem der *redundanten* Datenhaltung). Da es aber keine Modellierungstechnik gibt, die allen Anforderungen gleichzeitig gerecht wird, die an ein integriertes System für Entwurf, Planung und Konstruktion zu stellen sind, ist der Griff zu einem hybriden, mehrere Techniken gleichzeitig integrierenden System fast unausweichlich.

3. Visualisierung und Interaktion

3.1 Projektionen

Die *Modell*-Komponente eines CAD-Systems stellt eine rechnerorientierte Beschreibung eines Ausschnitts der sichtbaren Welt dar. Im weiteren wird die Gesamtheit der im *Modell* beschriebenen Objekte einfach als *Welt* oder als *Szene* bezeichnet.

Gebäude sind dreidimensionale geometrische Gebilde. Zur interaktiven Bearbeitung und zur Visualisierung werden diese dreidimensionalen Körper in zweidimensionalen Ansichten dargestellt. Dabei kann ein einziges Ausgabegerät durchaus gleichzeitig mehrere "Ansichten" darstellen.

Die Abbildung dreidimensionaler Szenen in zweidimensionalen Ansichten beruht auf der Anwendung von *Transformationsmatrizen* auf die Daten des Modells.

Alle Abbildungen, die wir hier besprechen, sind *planare geometrische Projektionen*, d.h. Abbildungen in eine *Bildebene* mit Hilfe geradliniger *Projektionsstrahlen*.

Da alle planaren geometrischen Projektionen Geraden wieder in Geraden abbilden, reicht es aus, alle Schnittpunkte aller Geraden einer Szene (also alle Eckpunkte ebenflächig begrenzter Körper) abzubilden, und die abgebildeten Punkte im Bild wieder durch gerade Linien zu verbinden.

Es sind zweierlei Arten der Projektion üblich: *Parallelprojektionen* und *Zentralprojektionen* (Perspektiven).

Die *Parallelprojektion* verwendet parallele Projektionsstrahlen und entspricht somit einer Projektion mit Licht, das aus einer unendlich fernen Lichtquelle stammt. Die Projektionsstrahlen können dabei entweder senkrecht auf der Bildebene stehen (*orthogonale* Projektion) oder aber einen beliebigen Winkel mit der Bildebene einschließen.

Bei allen Parallelprojektionen werden Längen und Winkel von Strecken, die sich in Ebenen parallel zur Bildebene befinden, getreu abgebildet. Strecken, die auf Geraden liegen, die die Bildebene schneiden, werden verkürzt dargestellt.

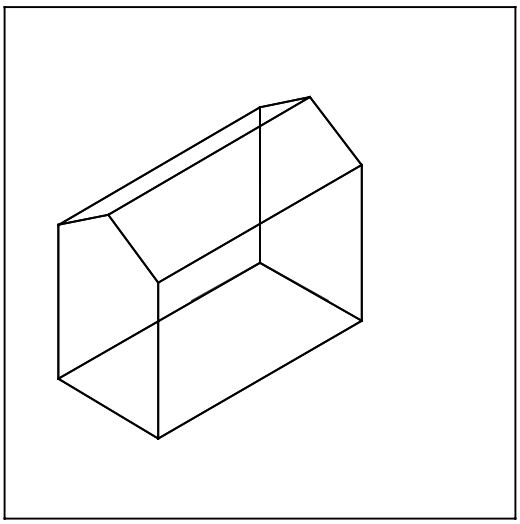

Bild 3.1: Isometrische Projektion eines Objekts

Eine *orthogonale* Projektion, bei der die Bildebene relativ zum globalen *x*-*y*-*z*-Koordinatensystem so ausgerichtet ist, daß alle drei Achsen in derselben Verkürzung erscheinen, wird als *Isometrie* bezeichnet. Die Bilder der drei Koordinatenachsen schließen dabei in der Bildebene den Winkel von 120° miteinander ein (Bild 3.1). Da alle hauptachsenparallelen Strecken in derselben Verkürzung erscheinen, werden die Längenverhältnisse auch in der Projektion getreu wiedergegeben. Für technische Anwendungen ist diese Projektionsart daher beliebt.

Weitere in der Praxis beliebte Parallelprojektionen sind die *Kavalliersprojektion* und die *Kabinettprojektion*. Beide arbeiten mit parallelen Projektionsstrahlen, die jedoch *nicht* orthogonal zur Bildebene stehen. Die Bildebene steht dabei jedoch senkrecht auf einer der drei Hauptachsen des abzubildenden Raumes. Im Fall der *Kavalliersprojektion* wird der Winkel β zwischen Projektionsstrahlen und Bildebene so gewählt, daß Geradenstücke, die senkrecht zur Bildebene stehen, unverkürzt abgebildet werden ($\tan\beta = 1$, siehe Bild 3.2). Der Winkel α gibt den Winkel an, den die orthogonalen Projektionen der Projektionsstrahlen mit der *u*-Achse bei dieser Abbildung einschließen.

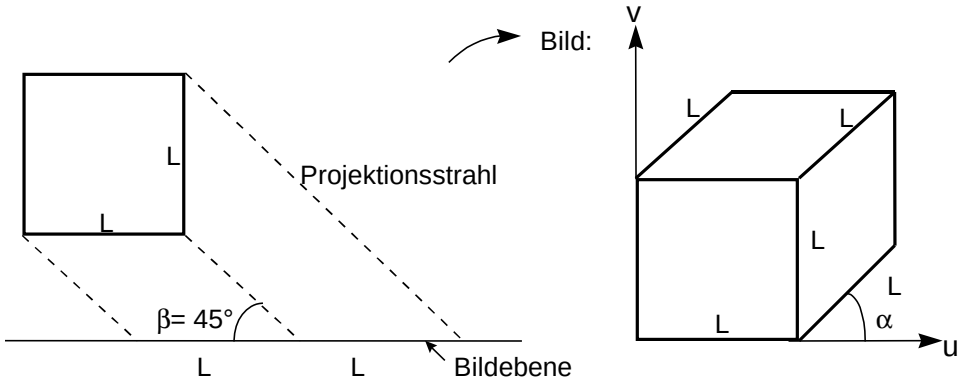

Bild 3.2: Kavalliersprojektion

Bei der *Kabinettprojektion* wird der Winkel β so gewählt, daß die dritte Hauptachse genau im Verkürzungsverhältnis ½ erscheint ($\beta \approx 63°$, Bild 3.3).

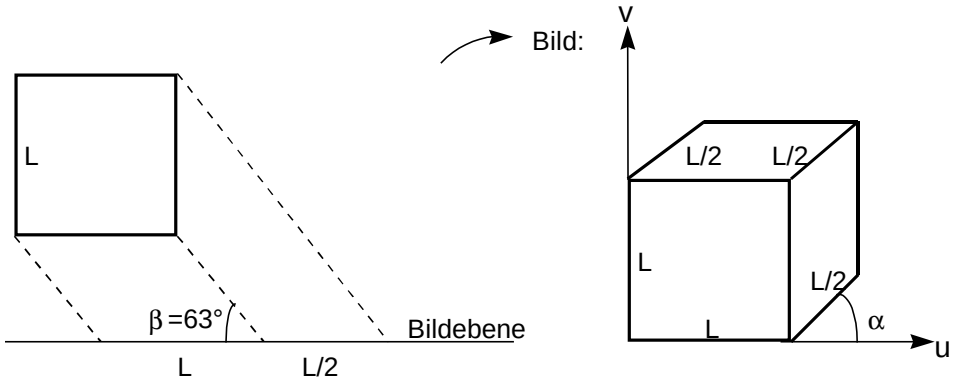

Bild 3.3: Kabinettprojektion

Parallelprojektionen bilden alle Objekte einer Szene im gleichen Maßstab auf die Bildebene ab, unabhängig von ihrer Entfernung zur Bildebene. Daher entsteht kein sehr realistischer Bildeindruck.

Realistischere Bilder ergibt die *Zentralprojektion* (*Perspektive*). Im folgenden konzentrieren wir uns auf diese Art der Projektion. Beispiele für Zentralprojektionen zeigt Bild 3.5.

Zur *mathematischen Beschreibung einer Projektion* muß zunächst die Lage der *Bildebene* im Raum angegeben werden. Wir werden im folgenden ein kartesisches *u*-*v*-*w*-Koordinatensystem verwenden, das so orientiert ist, daß die *w*-Achse senkrecht zur Bildebene steht. Dieses Koordinatensystem kann im Raum beliebig orientiert sein. Über die Art und Richtung der Projektion sagt die Lage der Bildebene allein noch nichts aus.

Bild 3.4 zeigt die Situation bei der *Zentralprojektion*. Wir nehmen an, daß die Bildebene die Ebene *w=d* unseres *u*-*v*-*w*-Koordinatensystems ist. Der Ursprung *Z* dieses Koordinatensystems wird als *Projektionszentrum* bezeichnet. Er entspricht dem *Standpunkt des Beobachters*. Wächst der Abstand *d* zwischen Bildebene und Betrachterstandpunkt ins Unendliche, so geht die Zentralprojektion in eine Parallelprojektion über. Der Durchstoßpunkt *A* der *w*-Achse durch die Bildebene wird *optischer Achsenpunkt* genannt.

Auf der Bildebene ist ein *Bildfenster* (*view window*) festgelegt. Das Bildfenster umgrenzt als u-v-achsenparalleles Rechteck denjenigen Ausschnitt der Bildebene, den wir nachher tatsächlich betrachten werden. Das Bildfenster braucht keinesfalls um den optischen Achsenpunkt zentriert zu sein; dieser kann vielmehr sogar außerhalb des Bildfensters gelegen sein.

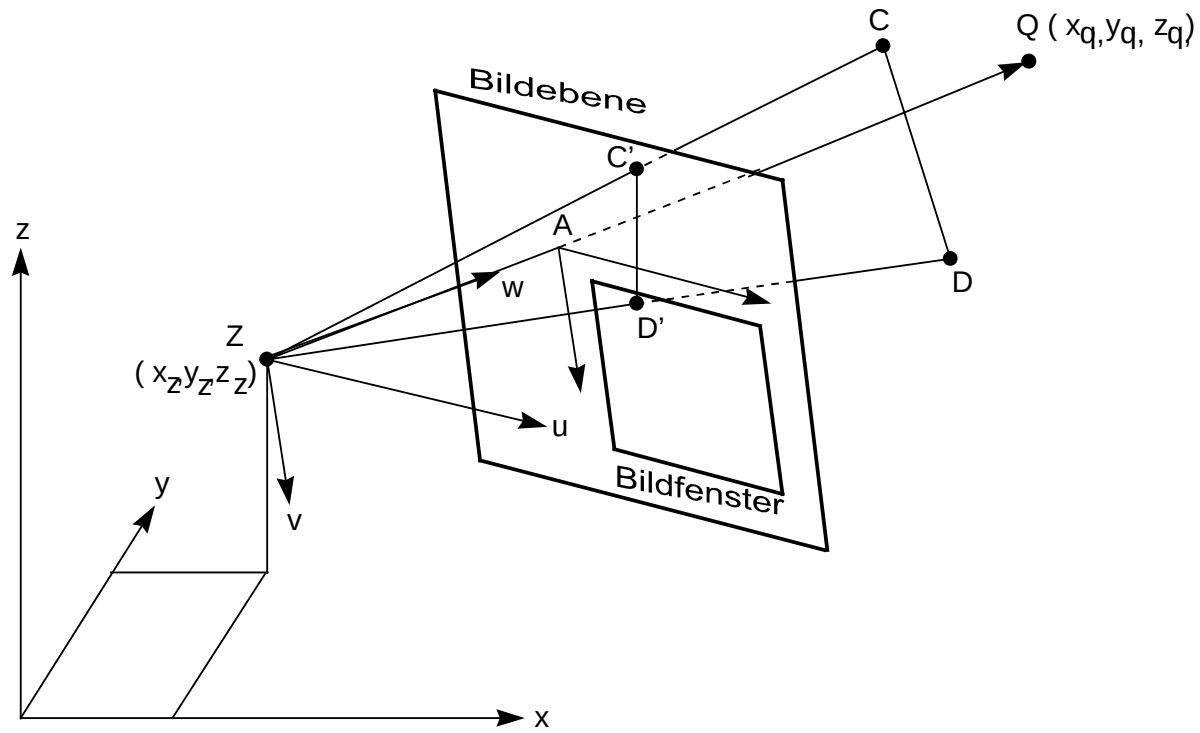

Bild 3.4: Zentralprojektion. Definition des Bild- und Weltkoordinatensystems

Objekte, die sich in der Ebene $w = 0$, also in der Parallelebene zur Bildebene durch den Beobachterstandpunkt befinden, werden auf der Bildebene in unendlich ferne Punkte abgebildet. Objekte, die sich sehr nahe vor oder hinter dem Betrachter befinden, werden entsprechend sehr stark verzerrt und sehr groß abgebildet, während Objekte, die sich in großer Entfernung vom Projektionszentrum befinden, auf dem Bild im Vergleich zu näher gelegenen Objekten fast zu Punkten zusammenschrumpfen.

Die Zentralprojektion entspricht der Abbildung einer Szene durch eine Lochkamera, z.B. das menschliche Auge. Im Gegensatz zur in Bild 3.4 dargestellten Situation befindet sich die Bildebene dort jedoch "hinter" dem Projektionszentrum ($d < 0$), so daß das Bild auf dem Kopf steht.

Die Projektion kann in einfacher Weise durch Angabe des Betrachterstandorts $Z(x_z, y_z, z_z)$, eines weiteren Punktes $Q(x_q, y_q, z_q)$, auf den die "Kamera" gerichtet ist, und den Abstand *d* der Bildebene vom Betrachter (Projektionszentrum) definiert werden. Damit ist jedoch noch nicht die Lage der *u*- und *v*-Achse festgelegt, sondern das Koordinatensystem (*u, v, w*) kann noch beliebig um die *w*-Achse (optische Achse) rotiert werden.

Bei Anwendungen aus dem Bauwesen ist meist eine globale *z*-Achse definiert, die in Richtung oder gegen die Richtung der Schwerkraft weist. Wir wollen im folgenden die Transformationsmatrizen herleiten, die für eine Projektion erforderlich sind, bei der die *u*-Achse parallel zur *xy*-Ebene eines globalen Koordinatensystems ist, dessen *z*-Achse entgegen der Richtung der Schwerkraft orientiert ist. Von den 6 Freiheitsgraden, die wir bei der Wahl des Betrachterkoordinatensystems haben, verwenden wir dabei nur 5 und verzichten auf eine Rotation um die optische Achse. Wir arbeiten wieder wie in Abschnitt 2.4 mit homogenen Koordinaten.

Zunächst führen wir eine Translation vom Koordinatensystem *x-y-z* in das Koordinatensystem *x'-y'-z'* durch, die den Ursprung des Koordinatensystems in den Betrachterstandpunkt *Z* verlegt:

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_z \\ 0 & 1 & 0 & -y_z \\ 0 & 0 & 1 & -z_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Als nächstes rotieren wir das Koordinatensystem um die neue *z'*-Achse, so daß die *y"*-Achse in Richtung auf den Fokussierpunkt *Q* verweist:

$$\mathbf{R_1} = \begin{bmatrix} \cos\theta_z & \sin\theta_z & 0 & 0 \\ -\sin\theta_z & \cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \text{ wobei}$$

$s_x = x_q - x_z$, $s_y = y_q - y_z$,

$s_{xy} = \sqrt{s_x^2 + s_y^2}$ und

$\cos\theta_z = s_y / s_{xy}$ für $s_{xy} \neq 0$, sonst $\cos\theta_z = 1$

$\sin\theta_z = -s_x / s_{xy}$ für $s_{xy} \neq 0$, sonst $\sin\theta_z = 0$

Jetzt muß noch eine Rotation um die zur *xy*-Ebene parallele *x"*-Achse erfolgen, so daß die *z"*-Achse in die *w*-Achse überführt wird. Die *x"'*-Achse wird dann zur *u*-Achse und die *y"'*-Achse zur *v*-Achse:

$$\mathbf{R_2} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \sin\varphi_x & -\cos\varphi_x & 0 \\ 0 & \cos\varphi_x & \sin\varphi_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \text{ wobei}$$

$s_z = z_q - z_z$,

$s_{xyz} = \sqrt{s_x^2 + s_y^2 + s_z^2}$

$\cos\varphi_x = s_z / s_{xyz}$

$$\sin\varphi_x = s_{xy} / s_{xyz}$$

Der Drehwinkel, um den hier gedreht wird, ist $-(90°-\varphi_x)$. Daher sind die Rollen von sin und cos und die Vorzeichen in der Transformationsmatrix vertauscht. φ_x gibt die Neigung der Verbindungsgerade von *Z* nach *Q* bezüglich der *xy*-Ebene an.
Verbinden wir alle diese drei Transformationen, so gelangen wir zu der Transformationsmatrix, die das *x*-*y*-*z*-System in das *x'''*-*y'''*-*z'''*-System (bzw. *u*-*v*-*w*-System) überführt:

$$\mathbf{U} = \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{T}$$

Im *u*-*v*-*w*-Koordinatensystem können wir nun die Projektion auf die Bildebene *w*=*d* durchführen. Sehen wir uns zunächst die Abbildungsmatrix für die Zentralprojektion an:

$$\mathbf{M}_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

Durch diese Transformationsmatrix werden in homogenen Koordinaten dargestellte Punkte $(u; v; w; t)$ des dreidimensionalen euklidischen Raums auf die Bildebene $w = d$ projiziert.

Die entsprechende Transformationsmatrix für eine Parallelprojektion mit Projektionsstrahlen, die parallel zur *w*-Achse verlaufen (orthogonale Parallelprojektion auf die *u*-*v*-Ebene), lautet:

$$\mathbf{M}_P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Die Lage der Bildebene bezüglich der *w*-Achse spielt hier keine Rolle. Wir können also ohne Einschränkung der Allgemeinheit die Bildebene durch *w*=0 beschreiben.

Die Verkettung der Abbildungsmatrizen ergibt schließlich die Gesamttransformation **P**, die eine dreidimensionale Szene auf die Ebene abbildet:

$$\mathbf{P} = \mathbf{M} \cdot \mathbf{R}_2 \cdot \mathbf{R}_1 \cdot \mathbf{T}$$

Je nach der Lage der Bildebene bezüglich des Weltkoordinatensystems wird bei der *Zentralprojektion* nach *Einpunktperspektive, Zweipunktperspektive* und *Dreipunktperspektive* unterschieden. Dabei wird die Anzahl der Hauptachsen gezählt, die die Bildebene schneiden. Die Schnittpunkte zwischen Achse und Bildebene werden als *Fluchtpunkte* bezeichnet. Bild 3.5 zeigt eine Szene in Einpunkt-, Zweipunkt- und Dreipunktperspektive.

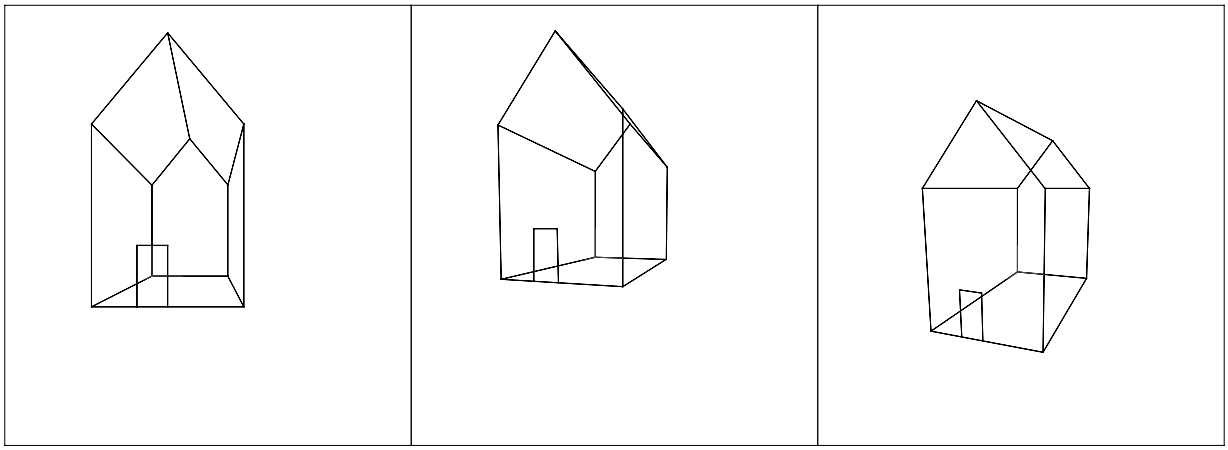

Bild 3.5: Darstellung einer Szene in verschiedenen Zentralprojektionen. Von links nach rechts: Einpunktperspektive, Zweipunktperspektive, Dreipunktperspektive.

Zur Erzeugung von Computergraphiken mit Zentralprojektionen ist es empfehlenswert, das Projektionszentrum außerhalb der Szene zu wählen, um stark verzerrte Darstellungen zu vermeiden. Den Blickzielpunkt *Q* wählt man dann am besten innerhalb der Szene, und den Abstand *d* sollte man so festlegen, daß die Bildebene etwa in der Mitte der Szene liegt. Dann lassen sich die Größenverhältnisse der Abbildung gut abschätzen, und man kann sinnvolle Einstellungen für das Bildfenster wählen.

Wird bei außerhalb der Szene gelegenem Projektionszentrum *Z* der Abstand *d* klein gewählt, so befindet sich die Bildebene weit von der Szene, und die Szene wird entsprechend klein dargestellt. Wählt man ein entsprechend kleines Bildfenster, so kann die Szene dennoch bildfüllend dargestellt sein. Man erhält dann allerdings eine Projektion mit fast parallelen Strahlen, so daß der Eindruck einer "Teleobjektiv-Aufnahme" entsteht (vgl. Bild 3.6).

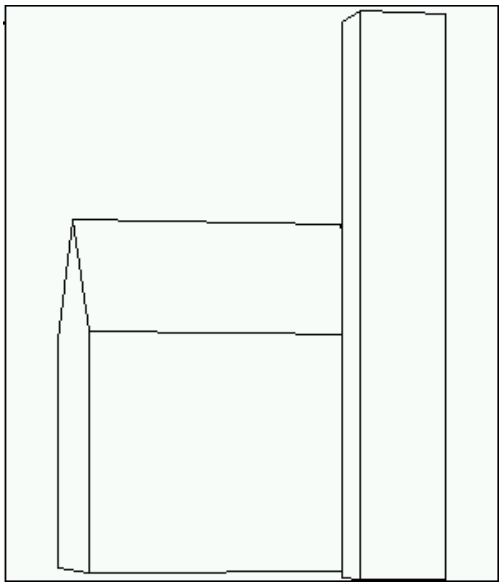

Bild 3.6: Zentralprojektion, bei der sich Betrachter und Bildebene weit von der Szene entfernt befinden. Der räumliche Eindruck ist schwach ausgeprägt, da die Projektionsstrahlen fast parallel verlaufen.

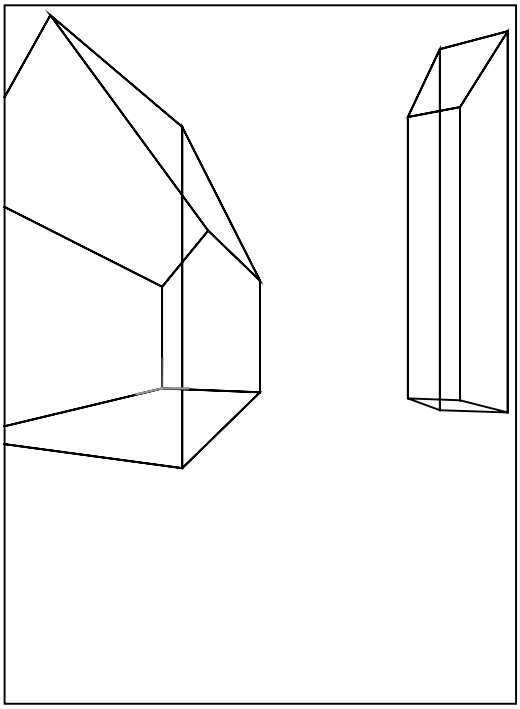

Bild 3.7: Projektion derselben Szene wie in Bild 3.6, aber mit nahe an der Szene (aber außerhalb der Szene) gelegenem Projektionszentrum *Z*. Die Bildebene ist so gewählt, daß sie die Objekte schneidet.

Demgegenüber zeigt Bild 3.7 eine Zentralprojektion, bei der das Projektionszentrum *Z* zwar nahe an der Szene, aber doch außerhalb des einhüllenden Quaders der Szenenobjekte gewählt wurde. Der Abstand *d* zur Bildebene wurde so festgelegt, daß die Bildebene die Objekte der Szene schneidet. Es entsteht eine Darstellung mit guter Tiefenwirkung, etwa entsprechend einer "Weitwinkel-Aufnahme". Die dargestellte Szene ist dieselbe wie in Bild 3.6.

Abschließend sei noch darauf hingewiesen, daß eine Umkehrung der Abbildung **P** vom Bildraum in den *x*-*y*-*z*-Raum nicht möglich ist, da bei der Projektion eine Dimension verlorengeht. Beim Konstruieren dreidimensionaler Szenen bereitet dies Schwierigkeiten mit der Gestaltung der Benutzungsschnittstelle. Um einen Punkt im dreidimensionalen Raum eingeben zu können, kann man beispielsweise fordern, daß der Punkt in zwei verschiedenen Projektionen der Szene angegeben wird (z.B. Konstruktion in zwei orthogonalen "Rissen", d.i. Parallelprojektionen). Eine Alternative besteht darin, im dreidimensionalen Raum eine "Arbeitsebene" zu definieren, und alle in der Bildebene vom Benutzer eingegebenen Punkte auf diese "Arbeitsebene" zu beziehen: In diesem Fall transformiert man den vom Benutzer auf der Bildebene angegebenen Punkt ins Weltkoordinatensystem, verbindet ihn mit dem Projektionszentrum *Z* (bei Parallelprojektion mit einem unendlich fernen Projektionszentrum) und bringt diese Verbindungsgerade zum Schnitt mit der definierten "Arbeitsebene". So erhält man dann durch eine einzige Mauseingabe einen eindeutig definierten Punkt im Weltkoordinatensystem. Es gilt freilich zu beachten, daß es bei dieser Art der Konstruktion dreidimensionaler Punkte mit Hilfe einer einzigen Projektion in jedem Falle Punkte auf der Bildfläche gibt, für die die Verbindungsgerade zum Projektionszentrum parallel zur "Arbeitsfläche" verläuft, so daß für diese Punkte der angegebene Algorithmus nicht eingesetzt werden kann. Es ist dann Aufgabe des Benutzers, Bildebene und Projektionszentrum geeignet zu wählen, um für den gerade aktuellen Bereich der Konstruktion eine eindeutige Zuordnung zwischen Bildpunkt und Weltpunkt zu gewährleisten.

3.2 Darstellung auf dem Bildschirm

Der erste Schritt zur Erzeugung eines Bildes der gewählten Szene auf dem Bildschirm besteht in der Projektion der dreidimensionalen Szene auf die Bildebene.

Im nächsten Schritt legen wir ein "Bildfenster" (*view window*) auf der Bildebene fest. Das Bildfenster (vgl. Bild 3.8) gibt an, welcher Ausschnitt der Bildebene letztendlich auf den Bildschirm übertragen werden soll.

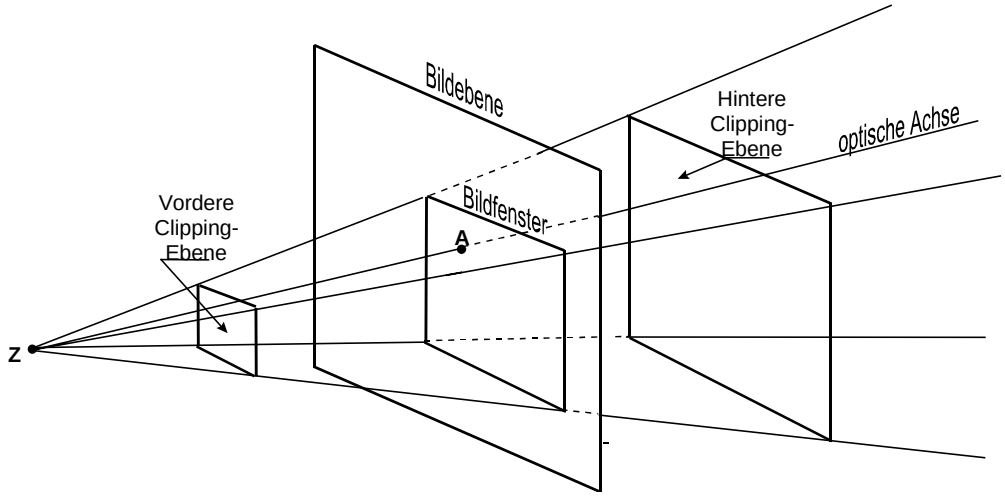

Bild 3.8: Sichtvolumen bei einer Zentralprojektion.

Durch das Projektionszentrum *Z* und das Bildfenster ist im Objektraum ein pyramidenförmiger Ausschnitt der Welt definiert, das sogenannte *Sichtvolumen* (Bild 3.8).

Objekte, die aus dem Sichtvolumen hinausragen, können vor der Durchführung der Projektion abgeschnitten werden (*Clipping*). Es ist sinnvoll, das Sichtvolumen außerdem zusätzlich durch eine *vordere* und *hintere Clippingebene* abzuschneiden, so daß man einen *Pyramidenstumpf* erhält (vgl. Bild 3.8): Bei Zentralprojektion werden Punkte, die sich in der Parallelebene zur Bildebene durch das Projektionszentrum (Ebene w=0) befinden, in unendlich ferne Punkte auf der Bildebene abgebildet. Objekte, deren Kanten diese Ebene durch das Projektionszentrum schneiden, werden bis zur Unkenntlichkeit verzerrt abgebildet. Auch Objekte, die sehr nahe am Projektionszentrum liegen, werden stark verzerrt (siehe Bild 3.9). Daher sollte man den in einer Projektion darzustellenden Ausschnitt der Welt so begrenzen, daß die Bildebene außerhalb des gewählten Ausschnitts liegt. Dieser Aufgabe dient die *vordere Clippingebene*.

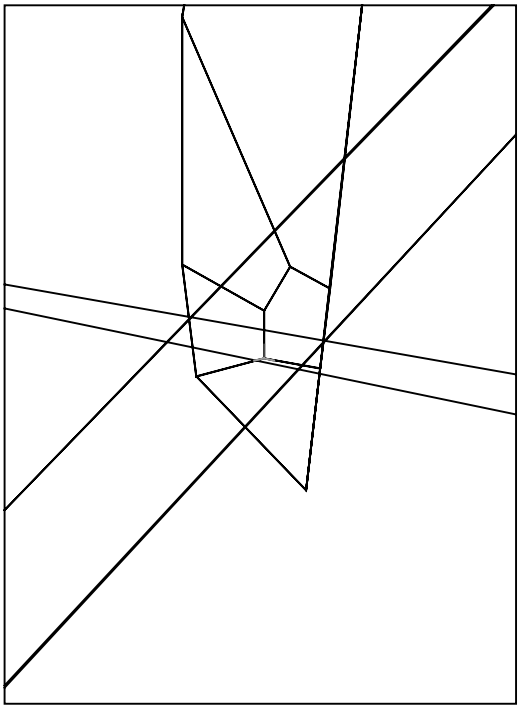

Bild 3.9: Objekte, die sich nahe an der Ebene w=0 befinden oder die diese Ebene schneiden, werden stark verzerrt abgebildet. Dargestellt ist dieselbe Szene wie in Bild 3.6 und 3.7.

Objekte, die sich in sehr großer Entfernung vom Projektionszentrum befinden, werden bei Zentralprojektion fast nur noch als Punkt abgebildet. Solche Objekte sollten vor der Projektion ebenfalls entfernt werden. Dieses Ziel verfolgt man mit der *hinteren Clippingebene.*

Die Lage der *vorderen Clippingebene* $w = w_1$ ist willkürlich; man wählt in der Regel $0 < w_1 < d$; ebenso willkürlich ist die genaue Lage der *hinteren Clippingebene* $w = w_2$ (in der Regel $d < w_2$).

Überdies ist die Projektion der Weltpunkte auf die Bildebene eine teure Operation, die für Objekte, die im Bildfenster gänzlich unsichtbar sind, eingespart werden sollte. Man sollte also wenigstens gänzlich außerhalb des Sichtvolumens gelegene Objekte vorab eliminieren.

Nach der Projektion der verbleibenden Objekte auf die Bildebene müssen alle Linien und Polygone, die den Rand des Bildfensters überschneiden, derart zurechtgestutzt werden, daß nur noch der innerhalb des Bildfensters liegende Teil verbleibt. Diese Operation wird als *Clipping* bezeichnet.

Ein gängiger Algorithmus zum *Clipping von Linien* ist der Algorithmus von *Cohen und Sutherland*. Auf diese Aufgabe beschränken wir uns hier.

Das zum Zeichnen freigegebene Gebiet sei ein achsenparalleles Rechteck, das durch die linke untere Ecke $(x_{\min}, y_{\min})$ und die rechte obere Ecke $(x_{\max}, y_{\max})$ definiert ist. Es soll nun eine Strecke *PQ* am Rand dieses Rechtecks gekappt und der innerhalb des Rechtecks (einschließlich des Randes) liegende Teil gezeichnet werden.

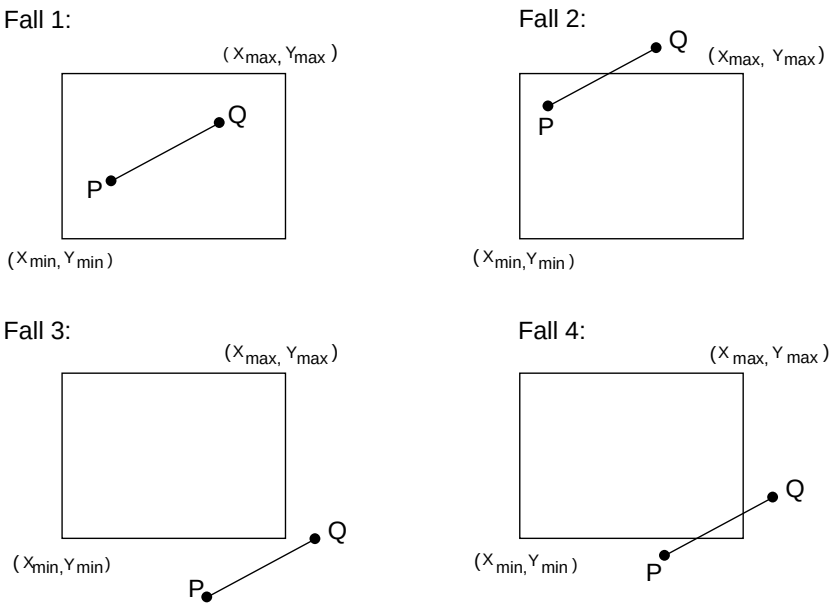

Bild 3.10: Mögliche Fälle der Lage einer zu zeichnenden Strecke *PQ* bezüglich eines Viewport-Rechtecks beim Clipping-Problem.

Dabei sind vier Fälle zu unterscheiden (Bild 3.10):

- Beide Endpunkte *P* und *Q* der Strecke liegen innerhalb des Rechtecks. Die gesamte Strecke kann gezeichnet werden. (Fall 1)
- Genau einer der beiden Endpunkte der Strecke liegt innerhalb des Rechtecks. In diesem Fall muß der Schnittpunkt zwischen der Strecke und dem Rand des Rechtecks ermittelt werden, und die Strecke wird auf den innerhalb des Rechtecks gelegenen Teil verkürzt. Dieser kann gezeichnet werden. (Fall 2)
- Beide Endpunkte liegen außerhalb des Rechtecks. In diesem Fall sind zwei Unterfälle möglich:
    - Die Strecke liegt ganz außerhalb des Rechtecks und braucht nicht gezeichnet zu werden. (Fall 3)
    - Die Strecke schneidet das Rechteck, und es müssen zwei Schnittpunkte berechnet werden, um den sichtbaren Teil zu bestimmen. (Fall 4)

Im trivialen 1. Fall kann die Strecke sofort gezeichnet werden, in allen anderen Fällen ist die Bestimmung von Schnittpunkten der die Strecke enthaltenden Geraden mit den achsenparallelen Geraden $x = x_{\min}$, $y = y_{\min}$, $x = x_{\max}$ und $y = y_{\max}$ erforderlich. Der Algorithmus von Cohen und Sutherland verfolgt das Ziel, die Anzahl von Schnittpunktsberechnungen durch vorgeschaltete einfache Tests zu minimieren. Dazu wird die gesamte Zeichenebene in neun Bereiche (siehe Bild 3.11) eingeteilt. Diese Bereiche werden durch einen vierstelligen binären Code charakterisiert. Dabei ist 0001 der Code für den Bereich $y > y_{\max}$ ("oben"), 0010 der Code für den Bereich $y < y_{\min}$ ("unten"), 0100 der Code für den Bereich "rechts"($x > x_{\max}$) und 1000 der Code für den Bereich "links" ($x < x_{\min}$). 0000 ist der Code für "innen". Durch Addition der Codes für die x- und y-Richtung werden die Codes für alle neun Teilbereiche ermittelt.

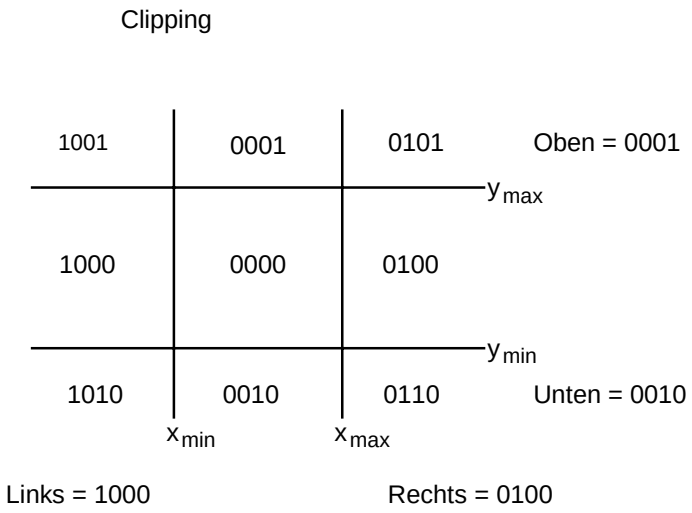

Bild 3.11: Zum Clipping-Algorithmus von Cohen und Sutherland.

Jedem Endpunkt *P* und *Q* einer zu zeichnenden Strecke wird der Code des Bereichs zugeweisen, in dem er liegt (z.B. $c(P)$ oder $c(Q)$. Die Strecke liegt genau dann ganz innerhalb des Rechtecks, wenn die Summe der Bereichscodes beider Endpunkte 0000 ergibt.

Die Situation einer Strecke, deren beide Endpunkte außerhalb des Rechtecks liegen und die das Rechteck nicht schneidet, tritt ein, wenn beide Endpunkte zugleich "links", "rechts", "oben" oder "unten" gelegen sind. Verknüpft man die Codes $c(P)$ und $c(Q)$ *für jede Stelle einzeln* mit der Operation "und zugleich" ($\wedge$), so erhält man ein Ergebnis, in dem nur noch die Stellen den Wert 1 haben, die in *beiden* Codes von 0 verschieden sind. Falls das Ergebnis $c(P) \wedge c(Q)$ *nicht* 0000 ist, liegt die Strecke, wie in Bild 3.11 leicht nachgeprüft werden kann, ganz außerhalb des Fensters und kann vernachlässigt werden.

Sind diese beiden trivialen Fälle abgehandelt, so bleiben die Strecken übrig, die verkürzt werden müssen, um korrekt dargestellt werden zu können. Sei *P* der Endpunkt der Strecke, der außerhalb des Rechtecks liegt, so müssen je nach Code des Punktes *P* folgende Schnittpunkte berechnet werden:

- Falls $c(P)$ den Wert 0001 enthält (*P* oberhalb des Rechtecks), muß der Schnitt mit $y = y_{max}$ berechnet werden, und der neue Punkt ersetzt *P*.
- Falls $c(P)$ den Wert 0010 enthält, muß der Schnittpunkt der Strecke mit der Geraden $y = y_{min}$ bestimmt werden und ersetzt *P*.
- Falls $c(P)$ den Wert 0100 enthält, ist der Ersatzpunkt für *P* durch Schnitt der Strecke mit der Geraden $x = x_{max}$ zu bestimmen.
- Falls $c(P)$ den Wert 1000 enthält, muß der Schnittpunkt zwischen der Strecke und der Geraden $x = x_{min}$ ermittelt werden.

Die so modifizierte Strecke wird im nächsten Schritt weiter untersucht. Beispielsweise könnte ja auch der Endpunkt *Q* außerhalb des Rechtecks liegen, oder der als Ersatz für einen Punkt *P* oder *Q* ermittelte Punkt könnte nach wie vor außerhalb des Rechtecks liegen.

Kurz zusammengefaßt ergibt sich so der folgende *iterative* Algorithmus:

```
done=FALSCH
wiederhole, solange done ungleich WAHR:
      Bestimme c(P) und c(Q)
```

```
        Falls c(P)+c(Q)=0000:
              Zeichne Strecke PQ (Strecke ganz innerhalb)
              done=WAHR;
        Sonst, falls c(P) ♈ c(Q):
              done=WAHR; (Strecke ganz außerhalb)
        Sonst:
              Wähle außerhalb des Rechtecks gelegenen Punkt R
              (Falls c(P) ungleich 0000, R=P, sonst R=Q).
              Berechne je nach c(R) den Schnittpunkt S und
              ersetze R durch S, so daß neue Strecke PS bzw. SQ.
        Ende
  Ende der Wiederholung
```

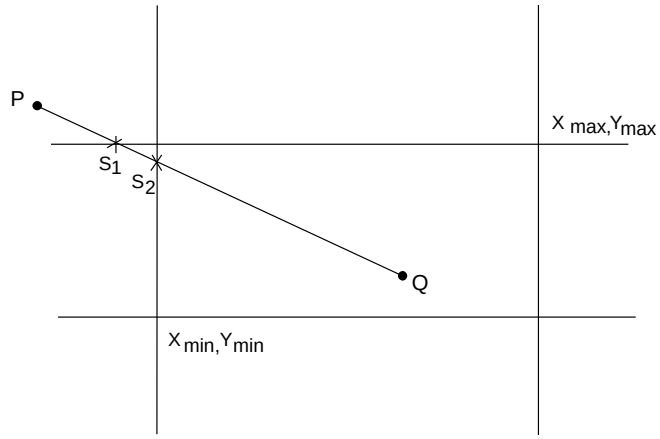

Bild 3.12: Schrittweise Verkürzung einer Strecke beim Clipping mit dem Cohen-Sutherland-Algorithmus. Unnötigerweise wird zunächst der Punkt S_1 berechnet, obwohl auch dieser außerhalb des Clippingrechtecks gelegen ist.

Der Algorithmus von Cohen und Sutherland berechnet zwar unter Umständen unnötigerweise mehrere Schnittpunkte, bevor die Strecke tatsächlich auf das ganz innerhalb des Cliprechtecks gelegene Stück verkürzt ist (vgl. das Beispiel in Bild 3.12), ist aber dennoch heute der beliebteste Algorithmus zur Lösung des Clipping-Problems.

Schließlich können wir das zurechtgestutzte Bild des Bildfensters auf einen entsprechenden achsenparallel begrenzten rechteckigen Bereich unseres Ausgabegerätes (z.B. Bildschirm) übertragen. Der Darstellungsbereich auf dem Ausgabegerät wird als *Gerätefenster* (z.B. *Bildschirmfenster*) oder *Viewport* bezeichnet.

Bei der Transformation vom Bildfenster auf den Viewport ist es notwendig, das Bild so zu skalieren, daß es in den Viewport hineinpaßt und diesen möglichst optimal ausfüllt. Eine anisotrope Skalierung ist bei CAD-Anwendungen unerwünscht, so daß bei nicht zusammenpassenden Proportionen von Bildfenster und Bildschirmfenster das Bild horizontal oder vertikal in das Bildschirmfenster eingepaßt werden muß.

3.4 Abbildungstechnik für Drahtmodelle

Im einfachsten Fall ist das Geometriemodell eines CAD-Systems ein Drahtmodell. Solche Modelle bieten aufgrund der fehlenden Eindeutigkeit kaum Möglichkeiten zur Generierung realistischer Ansichten. Etwas Übersicht läßt sich allerdings schaffen, indem Linien, die "nahe am Betrachter" liegen (*w* klein), mit größerer Strichstärke gezeichnet werden als entfernt liegende Linien. Diese

Methode, die recht einfach zu implementieren ist, wird als *Depth Cueing* bezeichnet. Bild 3.13 zeigt als Beispiel eine dreidimensionale Ansicht eines Objekts.

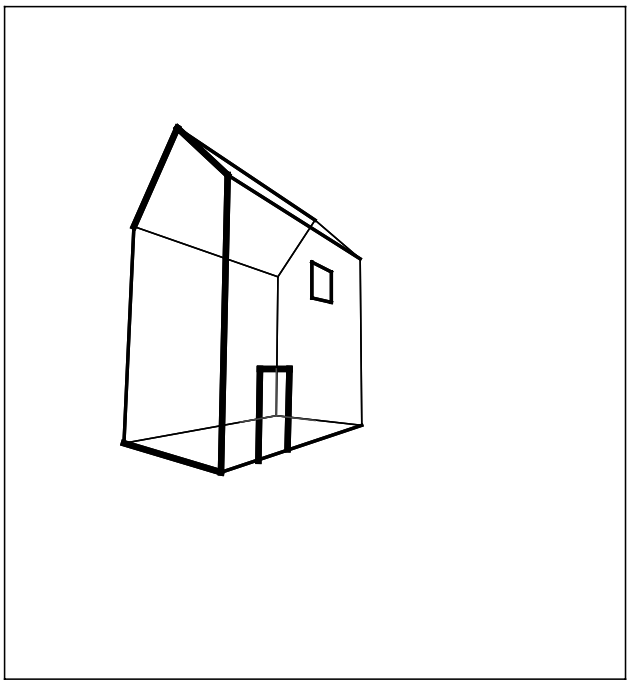

Bild 3.13: Darstellung eines Drahtmodells mit "Depth Cueing". Näher am Betrachter liegende Linien werden mit größerer Strichstärke dargestellt.

Solche Tricks können allerdings nicht darüber hinwegtäuschen, daß zur Erzeugung wirklich übersichtlicher Graphiken ein Flächenmodell verwendet werden muß, und daß wir Algorithmen zur Ermittlung der von einem Betrachterstandpunkt aus sichtbaren Oberflächen einer Szene benötigen.

3.5 Bestimmung verdeckter Oberflächen

Geometrische Modelle, die über das Drahtmodell hinausgehen, lassen die Erzeugung von Bildern zu, die der Seherfahrung durch Verdeckung unsichtbarer Teile der Objekte entsprechen.

3.5.1 Entfernen von unsichtbaren Rückseiten

Die einfachste Idee zur Elimination verdeckter Flächen einer computergenerierten perspektivischen Ansicht besteht darin, beim Zeichnen alle Flächen wegzulassen, die sich aus Sicht des Beobachters "auf der Rückseite" der dargestellten Körper befinden (*back face culling*).

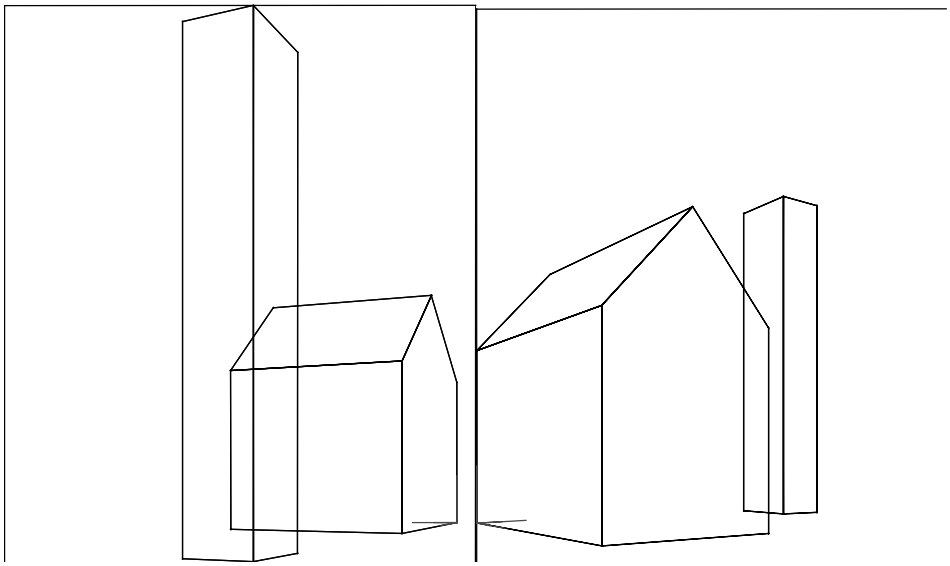

Bild 3.14: Perspektivische Darstellungen einer Szene mit entfernten Rückseiten.

Dazu wird jeder Facette *f* des Körpers ein Umlaufsinn zugewiesen, der so definiert ist, daß die Fläche, von außen betrachtet, gegen den Uhrzeigersinn umfahren wird. Die Normale $\vec{n}$ der Fläche weist dann nach außen. Um festzustellen, ob eine Fläche vom Betrachter aus gesehen auf der "Rückseite" des Körpers liegt, vergleicht man die Richtung $\vec{b}$ eines Sehstrahls, der vom Betrachter zu einem beliebigen Punkt im Inneren der untersuchten Fläche weist, mit der der Flächennormale. Sind beide gleichgerichtet, so ist das Skalarprodukt $\vec{n} \cdot \vec{b}$ der zugehörigen Richtungsvektoren positiv, und die Fläche kann vernachlässigt werden. Falls die Fläche sichtbar ist, ist das Skalarprodukt negativ oder Null. In letzterem Fall wird die Fläche als Strecke abgebildet.

Diese Methode ist auf die Anwendung bei *konvexen Polyedern* beschränkt. Dies sind ebenflächig begrenzte Körper ohne einspringende Ecken oder Löcher. Bei solchen Körpern kann man sich sicher sein, daß "auf der Rückseite" liegende Flächen durch andere Flächen desselben Körpers verdeckt werden.

Wie Bild 3.14 zeigt, wird der visuelle Eindruck einer Szene durch Rückseitenentfernung schon etwas besser, jedoch kann man mit dieser Methode das *gegenseitige Verdecken* verschiedener Körper selbstverständlich nicht erfassen. Die Methode ist dennoch nützlich, um die Zahl der Flächen zu vermindern, die bei den folgenden, aufwendigeren Algorithmen untersucht werden müssen.

3.5.2 Tiefensortierung

Der Algorithmus "Tiefensortierung" ist ein Algorithmus zur Ermittlung sichtbarer Oberflächen, der im Gegensatz zu den anderen nachfolgenden Algorithmen ausschließlich im *Objektraum* arbeitet und dessen Aufwand daher nur von der Anzahl der Polygone (Oberflächenfacetten) in der betrachteten Szene abhängt, nicht jedoch von der verwendeten Bildschirmauflösung.

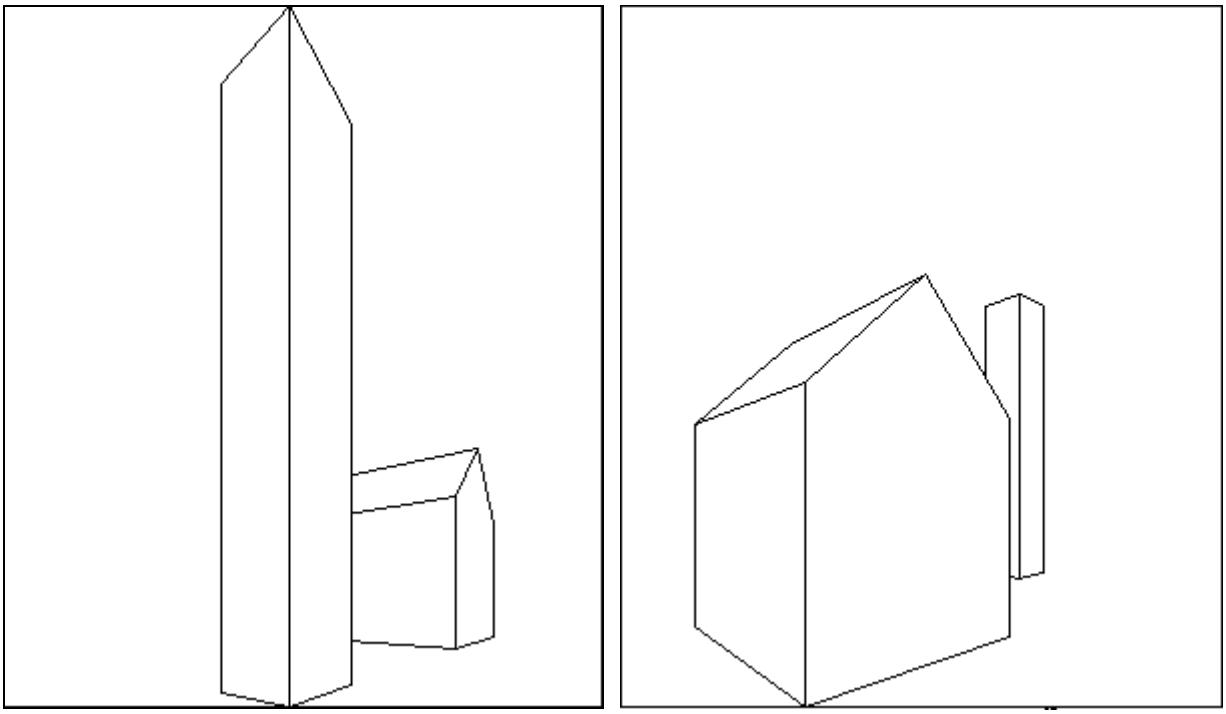

Bild 3.15: Darstellung einer Szene mit verdeckten Linien (Tiefensortierungsalgorithmus).

Bild 3.15 stellt eine Szene in verschiedenen Ansichten mit verdeckten Flächen dar. Beide Bilder wurden mit Hilfe des Algorithmus "Tiefensortierung" generiert.

Die Grundidee dieses Algorithmus ist einfach: Man transformiert alle Oberflächenfacetten einer Szene in das Koordinatensystem des Betrachters und sortiert die Flächen nach ihrer maximalen *w*-Koordinate (dies ist die Koordinatenachse, die vom Betrachter zur Szene verweist). Dann zeichnet man die Flächen in einer Reihenfolge, die abnehmender Entfernung vom Betrachter entspricht. Dabei werden weiter entfernt liegende Flächen durch näher am Betrachter gelegene Flächen einfach übermalt, weshalb der Algorithmus auch als *Maler*-Algorithmus (*painter algorithm*) bekannt ist. Probleme treten jedoch auf, wenn sich Flächen nicht eindeutig nach ihrer Entfernung vom Betrachter sortieren lassen, sondern denselben Tiefenbereich einnehmen. In diesem Fall muß zunächst untersucht werden, ob sich die Projektionen der zugehörigen Flächen überhaupt überlappen, oder ob die beiden betrachteten Flächen unabhängig voneinander nebeneinander gezeichnet werden können.

Diese Untersuchung kann z.B. mit Hilfe eines *Box*-Tests, also einer Prüfung der Überlappung der umschließenden achsenparallelen Rechtecke der beiden projizierten Polygone in der Bildebene erfolgen. Überschneiden sich die umgrenzenden Rechtecke (*bounding rectangles*) der Projektionen beider Flächen nicht, so ist es irrelevant, welche der beiden Flächen näher am Betrachter liegt.

Andernfalls ist eine genauere Analyse erforderlich. Schlimmstenfalls überlappen sich mehrere Flächen zyklisch, oder zwei Flächen überdecken sich jeweils gegenseitig teilweise. In solchen Flächen müssen die untersuchten Polygone in Teilflächen zerlegt werden, für die eine eindeutige Analyse möglich ist. Solche Fälle sind allerdings selten und treten z.B. im Bild 3.15 nicht auf. Dennoch ist es schwierig, den Tiefensortierungsalgorithmus allgemeingültig zu implementieren.

Obwohl der Algorithmus ausschließlich im Objektraum arbeitet, ist zur praktischen Realisierung des Algorithmus "Tiefensortierung" ein rasterorientiertes Ausgabegerät erforderlich, bei dem das Bild pixelweise im Arbeitsspeicher aufgebaut wird, bevor die eigentliche Ausgabe erfolgt, denn nur so ist

gewährleistet, daß später gezeichnete Oberflächen früher ausgegebene tatsächlich "übermalen". Außerdem sind zusätzlich zu den in diesem Skript erläuterten Algorithmen zum Zeichnen und Clipping von Linien entsprechende Funktionen zum Füllen und Clippen von Polygonen erforderlich.

3.5.3 z-Buffering-Algorithmus

Im Gegensatz zum Tiefensortieralgorithmus geht der z-Buffering-Algorithmus direkt von der Darstellung der Szene auf einem zweidimensionalen Rasterausgabegerät aus. Der Algorithmus wird von Operationen im *Bildraum* dominiert.

Der Algorithmus setzt neben einem Bildspeicher, der groß genug ist, um für jedes Pixel des Bildes die zugehörige Farbe zu speichern (z.B. Matrix von 640*480 Farbwerten bei Auflösung 640*480 Pixel), eine Matrix gleicher Dimension voraus, die für jeden Bildpunkt einen skalaren "Tiefenwert" *z* aufnehmen kann. Diese Matrix wird als *z-Puffer* bezeichnet. In einem ersten Schritt werden alle Tiefenwerte des Bildes mit einem sehr großen Wert z_{max} initialisiert, und allen Pixeln wird die Farbe "Hintergrund" zugewiesen.

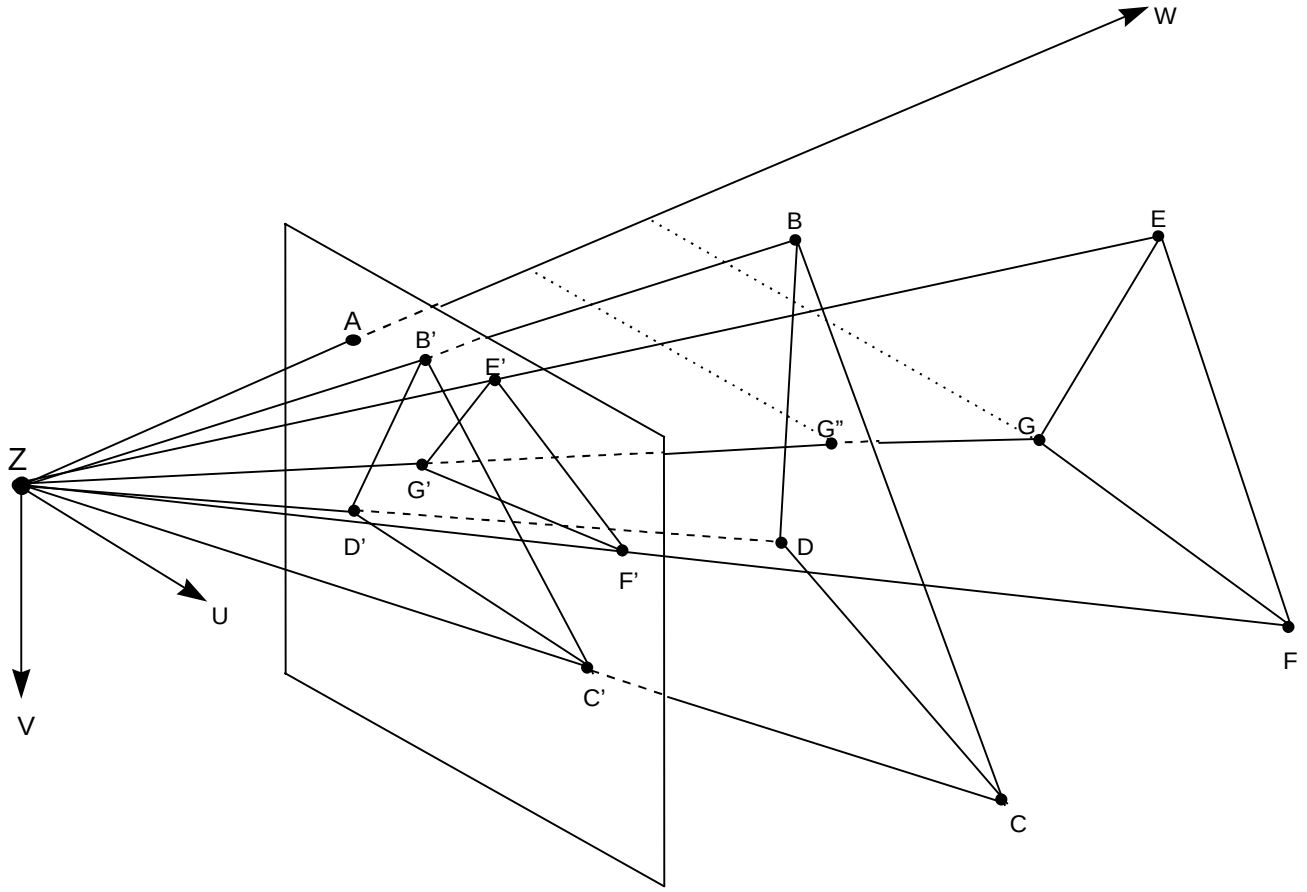

Bild 3.16: Ermittlung der sichtbaren Oberflächen mit dem z-Buffering-Algorithmus. Punkt G" besitzt eine geringere Tiefe (*w*-Koordinate) als Punkt G. Daher wird dem Pixel G' die Flächenfarbe der Fläche BCD und nicht die der Fläche EFG zugewiesen.

Jedes Polygon der Szene wird nun zunächst auf die Bildebene projiziert und in eine Rasterdarstellung konvertiert. Für alle Pixel, die innerhalb des projizierten Polygons liegen, wird ein Strahl durch das Projektionszentrum Z und den betreffenden Punkt der Bildebene (z.B. Punkt G' in Bild 3.16) "geschossen" und der Durchstoßpunkt dieses Strahls durch die originale Polygonfläche im

dreidimensionalen Objektraum berechnet. Der diskrete Bildpunkt wird also auf die Polygonfläche *zurückprojiziert* (Punkte G bzw. G"). Sodann wird die Entfernung des zurückprojizierten Punktes von der Bildebene bzw. vom Betrachterstandort bestimmt (*w*-Koordinate des Punktes im Betrachterkoordinatensystem). Ist diese Entfernung *w* (identisch mit dem Tiefenwert *z*) kleiner als der im z-Puffer gespeicherte Wert des Pixels, so wird die Farbe des aktuellen Pixels auf die Farbe der aktuellen Fläche gesetzt und der z-Puffer mit der *w*-Koordinate des Punktes im Objektraum belegt, da in diesem Fall der untersuchte Punkt (hier G") aus Sicht des Betrachters *vor* einem evtl. bereits vorher untersuchten Punkt (hier G) liegt und diesen daher verdeckt.

Die Hauptschleife dieses Algorithmus ist also eine Schleife über alle Polygone der Szene. In diese Schleife ist eine zweite Schleife eingebettet, die alle Pixel innerhalb der gerasterten Projektion des Polygons auf die Bildebene bearbeitet.

Der z-Puffer-Algorithmus ist einfach zu programmieren. Der Rechenaufwand ist erheblich und wächst bei ansonsten gleichen Abbildungsparametern mit zunehmender Bildauflösung, da die Anzahl zu untersuchender Pixel bzw. Punkte je Polygon bei Erhöhung der Auflösung quadratisch wächst. Außerdem wächst der Aufwand direkt proportional zur Anzahl der darzustellenden Polygone, weshalb eine Vorabeliminierung von Rückseiten konvexer Polygone ratsam erscheint. Sehr viele Graphikkarten unterstützen heute durch entsprechende Hardware (d.h. durch zusätzlichen, auf der Graphikkarte angeordneten, schnellen Speicher für die *z*-Werte) das z-Buffer-Verfahren. Das Verfahren ist sehr gut parallelisierbar, da jedes Polygon unabhängig von allen anderen untersucht werden kann; die Parallelisierung setzt allerdings die Möglichkeit eines gleichzeitigen Zugriffs aller aktiven Prozessoren auf den Bildspeicher und z-Puffer voraus (shared memory).

3.5.4 Scangeraden-Algorithmus

Bei diesem Algorithmus wird der Rasterbildschirm zeilenweise abgetastet. Zunächst wird eine Liste aller Polygone erstellt, deren Projektion von der aktuellen Bildschirmzeile geschnitten wird ("aktive Polygone"). Für alle Begrenzungskanten dieser Polygone wird dann geprüft, ob ihre Projektion die aktuelle Bildschirmzeile schneidet. Wenn ja, wird der Schnittpunkt berechnet, und die zugehörige Kante wird in eine Liste eingetragen, die nach der horizontalen Koordinate der Schnittpunkte sortiert ist (Liste der "aktiven Kanten").

Beginnend mit dem ersten Pixel ganz links auf dem Bildschirm werden nun die Pixel der Bildschirmzeile bis zum ersten Schnittpunkt mit einer Kante abgetastet. Alle diese Pixel erhalten die Farbe des Hintergrunds bzw. ggf. die Farbe eines dort sichtbaren Polygons. Beim Schnitt mit einer Kante ändert sich das sichtbare Polygon und damit die Farbe, die den folgenden Pixeln zugewiesen werden muß. Um das neue sichbare Polygon zu ermitteln, werden alle Polygone aus der Liste der aktiven Polygone untersucht, und das Polygon mit der geringsten Tiefe wird als nächstes ausgewählt. Allen nachfolgenden Pixeln wird nunmehr die Farbe des neuen sichtbaren Polygons zugewiesen, bis wieder eine Kante geschnitten wird.

Das Verfahren ähnelt also dem z-Puffer-Verfahren. Im Gegensatz zum z-Puffer-Verfahren wird nun jedoch als äußerste Schleife nicht mehr eine Schleife über alle Polygone, sondern eine Schleife über alle Bildschirmzeilen bzw. über alle Pixel des Bildes durchgeführt. Erst in der inneren Schleife werden die aktiven Polygone behandelt. Bei jedem Sprung von einer Bildschirmzeile zur nächsten muß die Liste der aktiven Polygone aktualisiert werden.

Im Scangeraden-Algorithmus entfällt die Notwendigkeit eines z-Puffers, so daß weniger Speicherplatz als beim z-Puffer-verfahren benötigt wird. Auch in der Rechenzeit ist das Verfahren vorteilhaft, weil zusammenhängende Folgen gleich gefärbter Pixel in einem Zug gezeichnet werden können, so daß die

aufwendige Rückprojektion eines *jeden* Pixels in den Objektraum entfällt. Nur bei jedem *Schnittpunkt* der Scangerade mit einer Kante ist eine *Rückwärtsprojektion* zur Ermittlung des neuen sichtbaren Polygons erforderlich. Dafür ist die Ablaufsteuerung des Algorithmus und die Behandlung von Sonderfällen schwieriger als beim z-Puffer-Algorithmus, und eine Parallelisierung ist nicht so einfach wie beim z-Puffer-Verfahren.

3.6 Photorealistische Darstellung

Die bisher besprochenen Methoden der Computergraphik lassen zwar die Generierung von Projektionen unter Berücksichtigung verdeckter Oberflächen zu; für die Erzeugung *realistischer* Bilder, wie sie immer mehr für Präsentationszwecke auch im Ingenieurwesen gefordert werden, reichen diese Methoden jedoch nicht aus.

Zur Erzielung *photorealistischer* Effekte sind *Farben* und *Beleuchtungseffekte* zu berücksichtigen. Die Wiedergabe von Farb- und Beleuchtungseffekten wird häufig als *Rendering* bezeichnet. Allerdings steht dieser Ausdruck prinzipiell für jede Art der Ausgabe eines Bildes.

Die *Helligkeit*, mit der eine Fläche einem Betrachter erscheint, hängt von der Lage der Fläche bezüglich der Lichtquellen, von den umgebenden Körpern (absorbierend, transluzent) und von den Oberflächenparametern der Fläche (Reflexionskoeffizient) selbst ab.

Als primäre *Lichtquelle* kommt ein gleichmäßiges diffuses (ungerichtetes) *Umgebungslicht* (*ambientes Licht*) ebenso in Frage wie Licht aus *punktförmigen* Lichtquellen.

3.6.1 Farbmodelle

Eine Grundfrage, die vor der Erzeugung photorealistischer Bilder geklärt werden muß, ist die *Darstellung von Farben* durch eine *digitale* Kodierung.

Bei der Ausgabe von Graphiken auf dem *Bildschirm* entsteht die vom Auge wahrgenommene Farbe des Bildpunktes durch *additive Farbmischung*, d.h. durch Mischung von drei verschieden intensiven Lichtstrahlen in den drei Grundfarben. Die drei Grundfarben sind Rot, Grün und Blau.

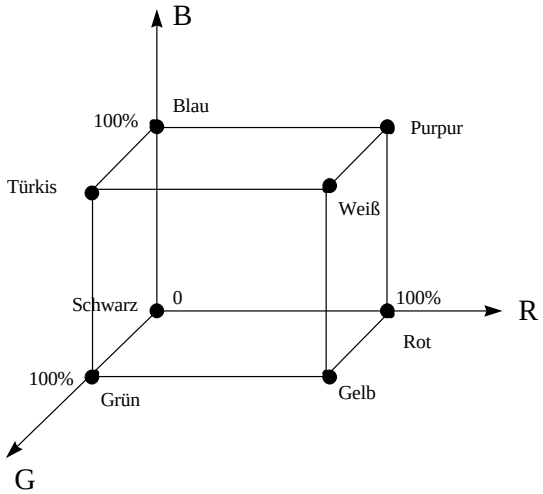

Bild 3.17: Additive Farbmischung (Mischung einer Farbe aus Lichtquellen in Rot, Grün und Blau).

Jedes Pixel des Bildschirms setzt sich bei genauer Betrachtung aus drei Punkten zusammen, die mit Pigmenten belegt sind, die beim Auftreffen des Elektronenstrahls der Röhre in den Farben Rot, Grün und Blau aufleuchten. Diese drei Punkte liegen jedoch so nahe beisammen, daß sie für das Auge zu

einem einzelnen Punkt verschmelzen. Die Farbe, die der Betrachter wahrnimmt, entsteht durch Vereinigung der drei erzeugten Lichtquellen.

Wir beschreiben Farben daher häufig im sogenannten RGB-Modell. Bild 3.17 zeigt, daß in diesem Modell jede beliebige Farbe als Vektor (Punkt) innerhalb eines Einheitswürfels dargestellt werden kann, längs dessen Achsen die Intensitäten der Grundfarben Rot, Grün und Blau aufgetragen sind.

Zur Darstellung der Farben wird die Intensität jeder der drei Grundfarben in diskreten Stufen kodiert. Typischerweise sind dabei je Grundfarbe z.B. 256 verschiedene Intensitäten zwischen 0 und 255 möglich. Sind alle Intensitäten 0, so erhalten wir die Farbe Schwarz. Setzen wir alle Intensitäten zu 255, so erhalten wir die Farbe weiß. Bei dieser Art der Farbbeschreibung benötigen wir je Pixel 3 Bytes Speicherplatz. Wir können damit rund 16 Millionen verschiedener Farben darstellen (256^3).

Je nach Leistungsfähigkeit und Speichergröße der Graphikkarte steht für jeden Bildpunkt eine gewisse Anzahl von Bits zur Verfügung, um die Farbe des betreffenden Bildpunktes zu beschreiben. Wenn wir für jeden Bildpunkt drei Bytes zur Farbdarstellung heranziehen, benötigen wir schon bei einer Auflösung von 1024*768 Pixeln einen VRAM von 2.4 MB. Häufig reicht es aus, in einem Bild weniger als 16 Millionen Farben gleichzeitig zur Verfügung zu haben. In der Regel verfügt die Graphikkarte über einen "Farbpalettengenerator", der in der Lage ist, durch Kombination der drei Grundfarben in verschiedenen Intensitäten eine große Anzahl verschiedener Farben darzustellen. Für die tatsächliche Bildschirmansteuerung wird jedoch nur ein kleiner Teil dieser Farben, z.B. 16 oder 256, gewählt. In diesem Fall ist für jeden Bildpunkt kodiert, welche Palettenfarbe zu ihm gehört. Der Graphikprozessor sieht dann in einer Lookup-Tabelle nach, wie die gewählte Farbe genau aussieht. Dadurch kann der Speicheraufwand im VRAM gering gehalten werden, und der Benutzer hat dennoch die Auswahl aus einer Vielzahl von Farben für die Darstellung.

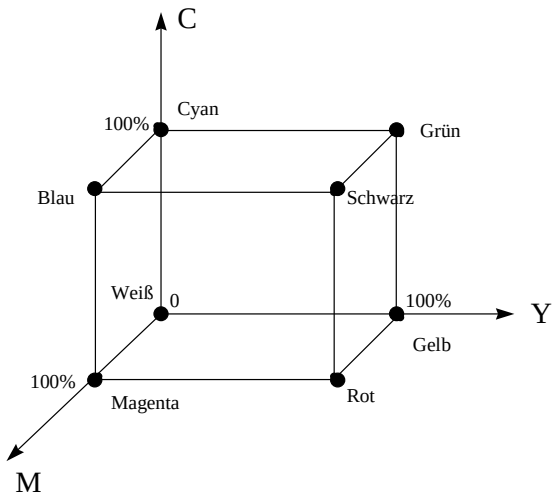

Bild 3.18: Subtraktive Farbmischung (Mischung einer Farbe aus den Körperfarben Gelb, Purpur (Magenta) und Türkis (Cyan)).

Beim *Drucken farbiger Bilder* wird die Farbe eines einzelnen Bildpunktes aus verschiedenen Farbschichten zusammengesetzt. Auch hier reichen wiederum drei Grundfarben. Die Farbmischung entsteht nunmehr jedoch dadurch, daß jedes Farbpigment einen Teil der einfallenden Lichtwellen absorbiert und nur die restlichen Farbbeiträge des weißen Lichtes reflektiert. Man spricht dann von *subtraktiver Farbmischung*. Die Grundfarben der subtraktiven Farbmischung sind Gelb, Magenta (ein tiefer, leicht bläulich angehauchter Rotton, auch als Purpur bekannt) und Cyan (ein türkisfarbener Ton). Das zugehörige Farbmodell kann wie das RGB-Modell im Rechner kodiert werden und wird als

CMY-Modell bezeichnet. Bild 3.18 stellt den CMY-Einheitswürfel und einige der zugehörigen Farben dar.

3.6.2 Beleuchtungsmodell

Zur photorealistischen Darstellung von Szenen gehört neben der Berücksichtigung der Farbigkeit der Objekte auch die Berechnung von Oberflächenhelligkeiten beleuchteter Objekte.

Ein beliebtes Modell zur Berechnung von Helligkeiten ist das PHONG-Modell.

Das PHONG-Reflexionsmodell ist ein heuristisches, nur lose an der physikalischen Realität orientiertes Modell zur Ermittlung der Lichtintensität bei Reflexion. Da das Modell aber zu guten Ergebnissen in der Praxis führt und einfach ist, wird es heute sehr häufig eingesetzt.

Das Modell beschreibt die Reflexion von Licht aus punktförmigen Lichtquellen in einer dreidimensionalen Szene.

Jede der punktförmigen Lichtquellen habe eine Intensität I_e (Intensität des einfallenden Lichtes). Handelt es sich um farbiges Licht, so kann die Farbe und Intensität dieses Lichtes durch die Intensität der drei Grundfarben Rot, Grün und Blau beschrieben werden.

Jede Oberfläche reflektiert einen gewissen Anteil des einfallenden Lichtes *diffus*, d.h., das einfallende Licht wird in alle Raumrichtungen mit gleicher Intensität abgestrahlt. Die Intensität dieses abgestrahlten Lichtes hängt vom Einfallswinkel des Lichtes von der Lichtquelle und von der Oberflächenbeschaffenheit ab.

Die Intensität des diffus abgestrahlten Lichtes wird beschrieben durch $I_d = I_e k_d \left(\vec{n} \cdot \vec{L}\right)$. k_d ist dabei ein materialabhängiger Reflexionskoeffizient $(0 \le k_d \le 1)$. $\vec{n}$ ist die normierte Flächennormale der beleuchteten Fläche und $\vec{L}$ ein Einheitsvektor, der vom betrachteten Oberflächenpunkt zur Lichtquelle weist (Bild 3.19).

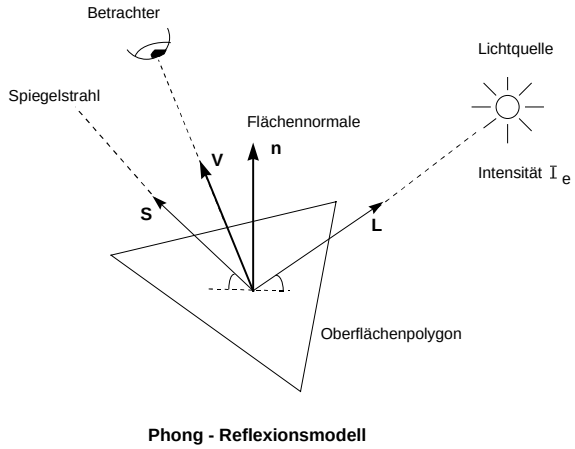

Bild 3.19: Zum PHONG-Reflexionsmodell.

Wird die Fläche durch mehrere Lichtquellen besonnt, so ergibt sich die Intensität des diffus abgestrahlten Lichtes einfach durch Addition: $I_d = k_d \sum_{i=1}^{k} I_{e,i} \left(\vec{n} \cdot \vec{L}_i\right)$.

In der physikalischen Realität nimmt die Lichtintensität einer punktförmigen Lichtquelle mit wachsendem Abstand r von der Quelle ab $(1/r^2)$, da die Lichtenergie sich auf eine Kugeloberfläche $4\pi r^2$ verteilt. Sei der Abstand Lichtquelle-Oberfläche r^* und der Abstand Oberfläche-Betrachter r, so ergibt sich eine Lichtintensität proportional zu $1/R^2$ mit $R = r^* + r$.

Diese quadratische Abnahme hat sich als für praktische Zwecke zu stark erwiesen. Daher berücksichtigt man im Phong-Modell den Abstandseffekt durch einen Abminderungsfaktor $f_a = 1/(r+k)$ *oder* $f_a = 1/(k_1 + k_2 R + k_3 R^2)$ mit $0 \le k_i \le 1$.

Das ambiente, diffuse Umgebungslicht wird im Phong-Ansatz einfach durch einen konstanten Zuschlag berücksichtigt. Verschattungseffekte und indirekte Beleuchtung durch benachbarte Körper werden also nicht erfaßt:

Somit erhalten wir eine Beziehung $I = I_a k_a + I_e f_a k_d (\vec{n} \cdot \vec{L})$ für die Intensität des von einer Fläche *diffus* reflektierten Lichtes. I_a stellt die Umgebungslichtintensität (ambientes Licht) dar, k_a einen "Umgebungsreflexionskoeffizienten". Besonders der Ansatz eines Koeffizienten k_a stellt eine grob vereinfachende Annahme dar, da dadurch der *globale* Effekt "Verschattung oder indirekte Beleuchtung durch Nachbarkörper" durch eine *lokale* Beschreibung ersetzt wird. Entsprechend "synthetisch" wirken daher Computervisualisierungen auf Basis dieses Modells. Schlag- und Halbschatten können im Phong-Modell nicht beschrieben und erklärt werden. Bild 3.20 zeigt zwei einfache Beispiele für die Anwendung der diffusen Reflexion (Bild 3.20 rechts in Grautönen, links in Farbe).

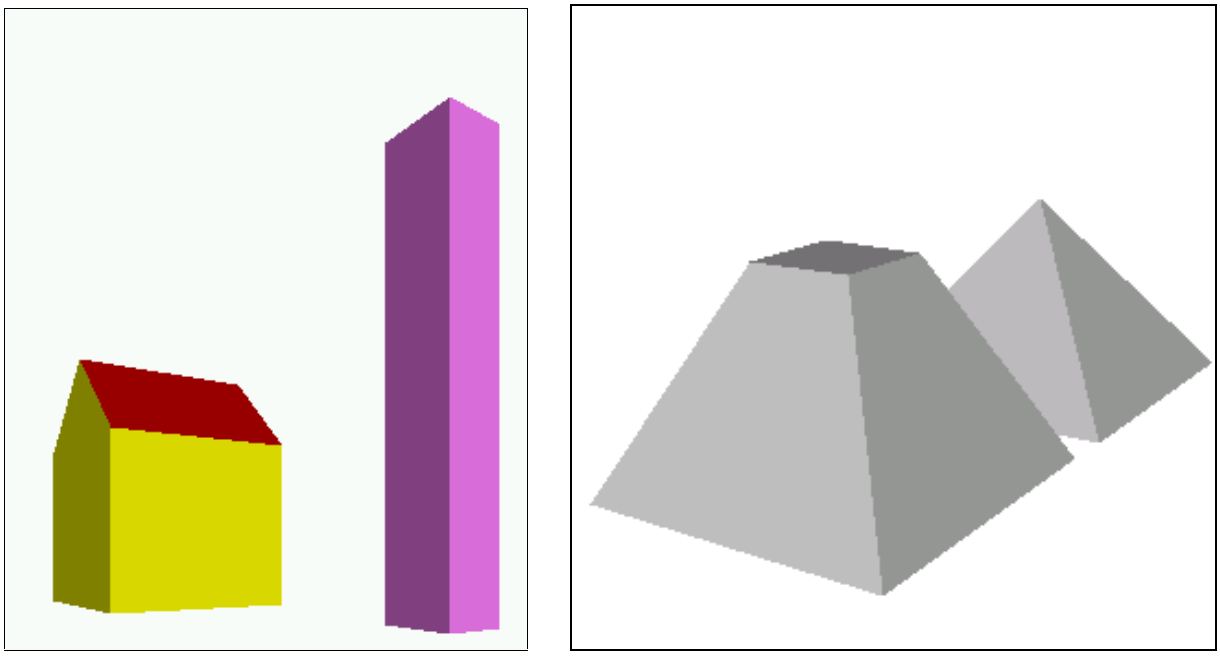

Bild 3.20: Schattierung zweier Szenen nach dem *Phong*-Beleuchtungsmodell. Nur *diffuse* Reflexionsanteile wurden berücksichtigt. Die Lichtquelle befindet sich im Unendlichen (paralleles Licht).

Soweit ist die Beschreibung der Reflexion aber noch nicht vollständig:

Ein Teil des auf eine Fläche einfallenden Lichtes wird nicht diffus, sondern *spiegelnd* reflektiert. Dies bedeutet, daß der einfallende Lichtstrahl bevorzugt in eine bestimmte Raumrichtung abgestrahlt wird (Einfallswinkel des Lichtstrahls = Ausfallswinkel des Spiegelstrahls). Befindet sich der Betrachter der

Szene in dieser Richtung, so wird die betreffende Stelle der Oberfläche als besonders hell wahrgenommen (*highlight*, Schlaglicht). Außerdem hat das spiegelnd reflektierte Licht die Farbe des einfallenden Lichts, während diffus reflektierte Strahlen durch die Flächenfarbe beeinflußt werden.

Für die Erzeugung realistischer Bilder geht man anstelle eines einzelnen Spiegelstrahls von einem Kegel gespiegelten Lichts aus, dessen Intensität sehr rasch mit der Abweichung des Betrachterstandorts von der strahlenoptisch ermittelten Richtung des Spiegelstrahls abnimmt.

Für spiegelnde Reflexion macht das Phong-Modell daher folgenden Ansatz:

Der Betrachter befinde sich vom Oberflächenpunkt aus gesehen in Richtung $\vec{V}$, die Richtung des Spiegelstrahls sei $\vec{S}$ (Bild 3.19):

Dann setzt man für die Intensität des spiegelnd reflektierten Lichtanteils $I_s = I_e k_s (\vec{V} \cdot \vec{S})^n$ mit $0 \le n \le \infty$. Der Parameter *n* gibt an, ob es sich um eine matte oder eine stark spiegelnde Oberfläche handelt. Z.B. steht *n*=1 für eine sehr matte Oberfläche, während *n*=100 einer stark spiegelnden Oberfläche entspricht (vgl. Bild 3.21). Je mehr die Richtungen vom Oberflächenpunkt zum Betrachter und die Richtung des Spiegelstrahls zusammenfallen (Skalarprodukt von $\vec{V}$ und $\vec{S}$ entspricht dem Cosinus des Zwischenwinkels), desto stärker ist die Intensität des vom Betrachter wahrgenommenen Lichts. Der Potenzansatz ist rein pragmatisch-heuristisch begründet und entbehrt jeder physikalischen Grundlage, die damit erzeugten Bilder entsprechen jedoch gut der Seherwartung.

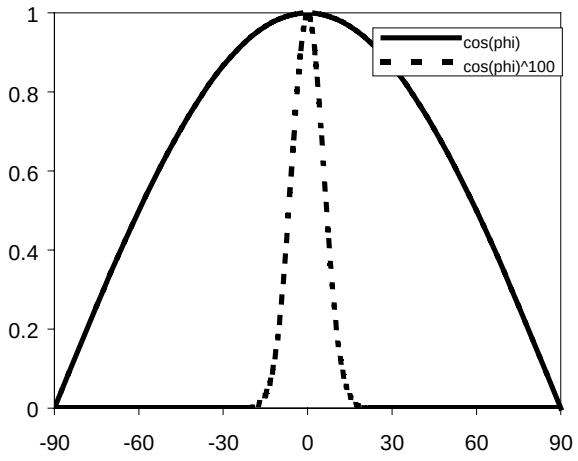

Bild 3.21: Berechnung der Intensität gespiegelten Lichts in Abhängigkeit vom Winkel zwischen Spiegelstrahl $\vec{S}$ und Betrachtungsrichtung $\vec{V}$.

Die Berechnung der Richtung des Spiegelstrahls ist etwas aufwendig. Um sie zu vermeiden, kann man anstelle des Vektors $\vec{S}$ einfach den Vektor $\vec{H} = \frac{1}{2}(\vec{L} + \vec{V})$ verwenden. Der Winkel zwischen diesem Vektor und dem Normalenvektor $\vec{n}$ der Fläche ist halb so groß wie der zwischen $\vec{S}$ und $\vec{V}$. Man erhält damit den Ansatz $I_s = I_e k_s (\vec{n} \cdot \vec{H})^n$ für die Intensität des beim Beobachter wahrgenommenen Spiegelstrahls. Da das Modell ohnehin heuristisch ist, kann man mit diesem Ersatzwinkel ebensogut arbeiten und paßt dann den Exponenten *n* entsprechend an.

Fassen wir alle Formeln zusammen, so gelangen wir zum Phong-Reflexionsmodell und bestimmen die Intensität eines im Betrachter wahrgenommenen Strahls (im Schwarzweiß-Fall) zu

$I = I_a k_a + I_e f_a (k_d (\vec{n} \cdot \vec{L}) + k_s (\vec{n} \cdot \vec{H})^n)$. Für farbiges Licht und farbige Oberflächen wird ein analoges Modell getrennt für jede der drei Grundfarben verwendet.

3.6.3 Ray Tracing

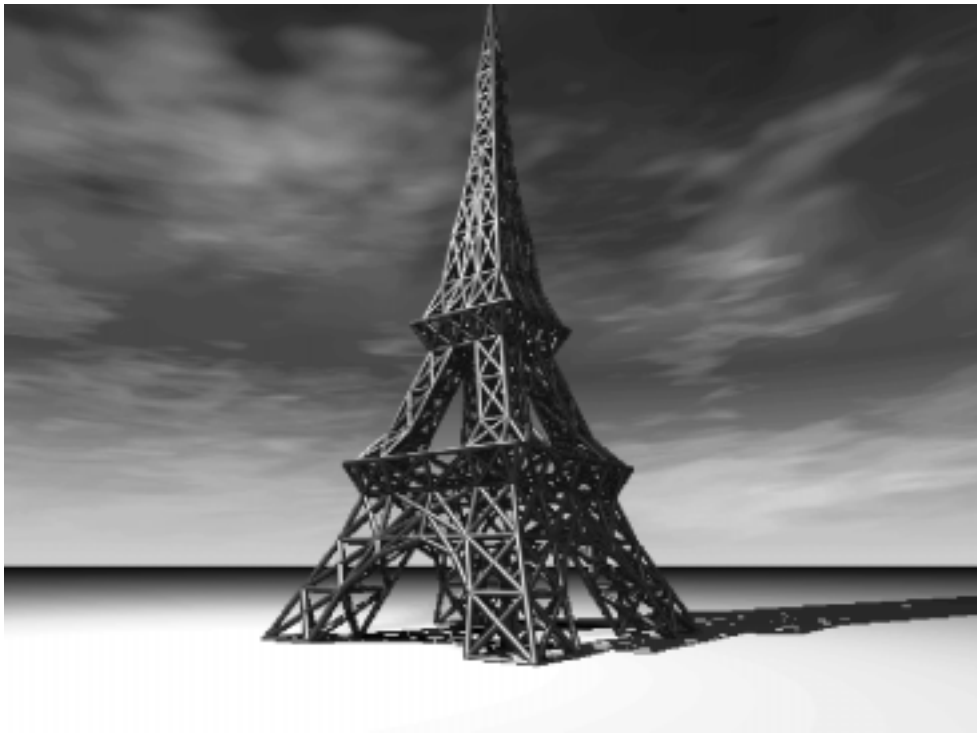

Bild 3.22: Visualisierung eines Objektes mit Hilfe des Ray-Tracing-Paketes Persistence of Vision™ Ray Tracer (POV-Ray™).

Die bisher geschilderten Methoden zur Ermittlung verdeckter Flächen und zur Schattierung ermöglichen die Erzeugung anschaulicher Darstellungen dreidimensionaler Szenen, aber keine Photorealistik. Einige Gründe wurden schon genannt, insbesondere fehlende Schatten. Die bisher diskutierten Modelle lassen aber auch keine Berücksichtigung von Oberflächentexturen und Oberflächenstrukturen zu und haben keinen Platz für transparente bzw. transluzente Objekte (durchsichtige oder durchscheindende Körper). Auch die indirekte Beleuchtung von Objekten durch Licht, das von anderen Objekten reflektiert worden ist, blieb bisher fast unberücksichtigt.

Methoden, die die Generierung photorealistischer Bilder gestatten, sind weitaus rechenaufwendiger als die bisher besprochenen Verfahren. Die wichtigsten Verfahren sind heute *Ray Tracing* und *Radiosity*.

Ray Tracing basiert auf der *Strahlenoptik* in Zusammenwirken mit dem heuristischen PHONG-Reflexionsmodell. Somit ist das *Ray-Tracing*-Verfahren nur bedingt physikalisch begründet. Demgegenüber baut das *Radiosity*-Verfahren auf die *Wellenoptik* auf und ist somit streng physikalisch begründet, kann aber typische Effekte der Strahlenoptik wie z.B. Spiegelung nicht wiedergeben.

Während das Verfahren Ray Tracing bei der Berechnung der Bilder den Beobachterstandpunkt benötigt, ist das Radiosity-Verfahren vom Standpunkt des Beobachters unabhängig. Für bewegte Beobachter ist das Radiosity-Verfahren daher vorzuziehen, obwohl es gegenüber dem ebenfalls aufwendigen Ray Tracing noch erheblich mehr Ressourcen verschlingt.

Die Bilder 3.22 und 3.23 zeigen die Visualisierung einer Szene mit dem Ray-Tracing-Verfahren (Strahlrückverfolgungsverfahren). Das Verfahren kombiniert diffuse und spiegelnde Reflexion, Transluzenz und Schlagschattenbildung.

Dazu geht das *Ray-Tracing*-Verfahren von einer Matrix diskreter Bildpunkte auf der Bildebene aus. Durch jedes Pixel des Bildfensters wird vom Projektionszentrum aus ein Strahl geschossen und rückverfolgt. Dieser Strahl verläuft also gerade entgegen der Richtung der Projektion und wird als *Pixelstrahl*, *Projektor* oder *Primärstrahl* bezeichnet.

Der Primärstrahl wird in den Objektraum verfolgt. Schneidet der Strahl kein Objekt der Szene, so wird das untersuchte Pixel auf die Hintergrundfarbe gesetzt. Werden mehrere Objekte geschnitten, so wird ermittelt, welche der getroffenen Flächen der Szene als erste von dem Strahl geschnitten wird, welcher Durchstoßpunkt also am nächsten am Projektionszentrum liegt.

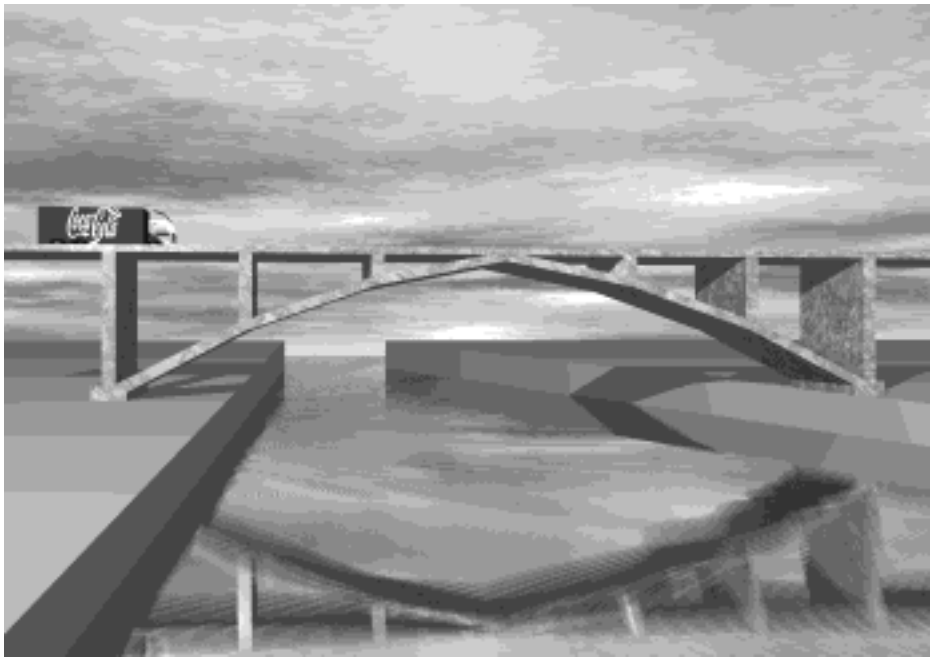

Bild 3.23: Darstellung einer Szene mit Hilfe des Ray-Tracing-Paketes Persistence of Vision™ Ray Tracer (POV-Ray™).

Dieser Durchstoßpunkt des Primärstrahls durch die Fläche kann Licht aus folgenden Quellen erhalten:
1. direktes Licht aus einer Lichtquelle
2. Licht, das von einer anderen Fläche reflektiert worden ist (indirekte Beleuchtung)
3. Licht, das von dem Körper, zu dem die Oberfläche gehört, oder von anderen Körpern durchgelassen worden ist (bei transparenten Körpern).

Um zu ermitteln, ob der untersuchte Punkt direktes Licht aus einer Lichtquelle erhält (1), kann man nun einfach von dem Punkt einen Strahl in Richtung auf die untersuchte Lichtquelle aussenden und prüfen, ob dieser einen anderen, opaken (lichtundurchlässigen) Körper schneidet oder nicht. Falls dies der Fall ist, erhält der betrachtete Punkt kein direktes Licht von der Lichtquelle. Im Falle, daß der geschnittene Körper transparent ist, kommt nur ein Teil des Lichtes der Lichtquelle am untersuchten Punkt an.

Der Teststrahl vom Oberflächenpunkt zu den Punktlichtquellen wird als *Schattenfühler* bezeichnet. Ist das Objekt matt und undurchsichtig, so reicht die Untersuchung der Schattenfühler aus. Man kann die

Intensität, mit der der untersuchte Punkt auf dem Bildschirm sichtbar wird, aus dem PHONG-Modell berechnen.

Der untersuchte Punkt kann aber auch auf einer spiegelnden Oberfläche oder einer transparenten Körper liegen. In diesen Fällen ist eine weitere Untersuchung notwendig. Man sendet zusätzlich zum Schattenfühler zwei weitere Sekundärstrahlen aus, nämlich einen *Reflexionsstrahl* und einen *Transmissionsstrahl*. Die Richtung des Reflexionsstrahls ergibt sich aus dem Gesetz "Einfallswinkel=Ausfallswinkel" der Strahlenoptik, die des Transmissionsstrahls aus den Beugungs- bzw. Brechungsgesetzen und den Materialparametern des Körpers, auf dessen Oberfläche der untersuchte Punkt liegt.

Reflexions- und Transmissionsstrahl werden weiterverfolgt, bis sie gegebenenfalls wiederum ein Objekt schneiden. Nun sind die Farbintensitäten dieses geschnittenen Objektes zu bestimmen. Je nach Reflexionsgrad bzw. Transluzenz des zuerst getroffenen Körpers trägt der mit dem Sekundärstrahl getroffene Körper zum Erscheinungsbild des untersuchten Punktes bei.

Für die mit den Transmissions- und Reflexionsstrahlen getroffenen Punkte können wiederum die gleichen Überlegungen wie für den vom Primärstrahl getroffenen Punkt angestellt werden (Rekursion, *rekursives Ray Tracing*). Da von jedem untersuchten Punkt bis zu drei verschiedene Arten von Strahlen (Schattenfühler, Transmissionsstrahl, Reflexionsstrahl) getestet werden müssen, die jeweils zu zusätzlichen Punkten der Szene (oder in den "Hintergrund") führen, steigt der benötigte Rechenaufwand exponentiell mit der Anzahl der Stufen der Strahlrückverfolgung an. Es entsteht ein *Strahlbaum*, dessen sämtliche Blätter aufgesucht werden müssen. Natürlich nimmt der Einfluß von Objekten, die durch die Strahlrückverfolgung aufgefunden werden, mit zunehmender Tiefe der Rückverfolgung ab, da ja jedes Objekt einen Teil des Lichtes absorbiert. Schneidet ein Teststrahl kein Objekt, so kann die Rekursion für diesen Strahl abgebrochen werden. In Szenen mit vielen Objekten wird die Strahlrückverfolgung häufig jedoch zu weiteren Objekten der Szene führen und damit einen gewaltigen Rechenaufwand verursachen.

Somit ist eine *Tiefenbeschränkung* bei der Strahlrückverfolgung sinnvoll und notwendig. Im einfachsten Fall wird immer nur die 1. Stufe des Verfahrens für jedes Pixel durchgeführt *(Ray Casting*). Damit wird nur der Sichtbarkeitsentscheid und eine einfache Intensitätsermittlung nach Phong sowie die Ermittlung von Schlagschatten durchgeführt, und man betrachtet implizit alle Objekte als matte Objekte. Nimmt man weitere Stufen der Rückverfolgung hinzu, wird der Bildeindruck immer realistischer (vgl. Bild 3.23).

Um den Rechenaufwand des Ray-Tracing-Verfahrens zu begrenzen, bieten sich verschiedene Techniken an:

- Komplexe Objekte mit einer Vielzahl von Oberflächenfacetten werden vereinfacht durch ein *umgebendes Volumen* (z.B. Kugel oder Quader) beschrieben. Bei dem Test, ob ein Strahl eine Facette des Objektes schneidet, wird zunächst geprüft, ob das umgebende Volumen geschnitten wird. Beispielsweise bei der Kugel als umgebendem Volumen ist dieser Test sehr einfach und damit effizient. Nur für den Fall, daß das umgebende Volumen geschnitten wird, werden die Facetten des Objektes getestet. Der Test mit dem umgebenden Volumen bringt nur dann wirkliche Rechenzeitersparnisse, wenn das umgebende Volumen den Originalkörper hinreichend dicht einhüllt. Für gedrungene Objekte sind daher umgebende Kugeln geeignet, für längliche Objekte hingegen eignen sich prismatische, ellipsoidale oder zylindrische umgebende Volumina besser als kugelförmige.
- Objekte, die sich nahe beeinander befinden, können gruppiert und ebenfalls durch ein umgebendes Volumen charakterisiert werden (*Clustering*). Solche Cluster können wie normale Objekte

untersucht werden und ihrerseits rekursiv bzw. hierarchisch zu größeren Clustern vereinigt werden (*hierarchisches Objektclustering*). Dabei entsteht ein *Clusterbaum*.

- Anstatt die Objekte zu gruppieren, kann man selbstverständlich auch den Objektraum rekursiv (z.B. analog zum Quadtree: Aufbau eines Octrees) unterteilen. Man muß dann nur noch alle jene Objekte gegen den Strahl testen, die in die vom untersuchten Strahl geschnittenen Volumenelemente hineinragen. Volumenelemente, die viele Objekte enthalten, können weiter aufgeteilt werden, um den Objektraum detaillierter zu erfassen.
- Anstatt eine feste Grenze für die Tiefe der Strahlrückverfolgung vorzugeben, kann man die Rückverfolgung an die Objekttypen anpassen und vom Einflußfaktor des untersuchten Strahls auf das aktuelle Pixel abhängig machen. Man gibt dann statt einer Tiefenschranke eine untere Schranke für den Einflußwert vor und gibt die Strahlrückverfolgung auf, wenn diese Einflußschranke unterschritten wird (*adaptives Ray Tracing*). Z.B. braucht der Reflexionsstrahl für eine nur schwach spiegelnde Oberfläche nicht besonders weit zurückverfolgt zu werden.

Abschließend sei noch angemerkt, daß beim rekursiven Ray Tracing die Rückseiten konvexer Körper nicht weggelassen werden können, da ja nicht nur vom Projektionszentrum ausgehende Stahlen untersucht werden müssen, sondern auch Strahlen, die von Punkten innerhalb der Szene ausgehen.

3.6.4 Radiosity

Im Gegensatz zum Phong-Modell und den daraus abgeleiteten Schattierungs- und Darstellungsverfahren baut das *Radiosity*-Verfahren auf die physikalischen Gesetze der Lichtausbreitung nach dem Wellenmodell auf. Dem Verfahren liegt eine aus den Wärmeübertragungs- und Energieerhaltungsgesetzen abzuleitende Gleichung zugrunde, die in die Kategorie der sogenannten *Fredholmschen Integralgleichungen* fällt. Zur näherungsweisen Lösung dieser Integralgleichung sind alle Oberflächen und Lichtquellen der Szene (analog zum Vorgehen bei der näherungsweisen Lösung partieller Differentialgleichungen der Strukturmechanik mit Finite-Elemente-Methoden oder Boundary-Element-Methoden) zu *diskretisieren*. Auf jedem Flächenelement wird dann ein *Ansatz* für die Lichtintensität (Strahlungsenergiedichte, *radiosity*) gewählt (z.B. elementweise konstant; auch polynomiale Ansätze und *Wavelet*-Ansätze werden erprobt). Die unbekannten *Koeffizienten* (bei konstantem Ansatz die Strahlungsdichten jedes Flächenelementes) dieses Ansatzes können durch die Lösung eines linearen Gleichungssystems, z.B. durch Gauß-Jacobi- oder Gauß-Seidel-Iteration, gelöst werden. Zur besseren Aufschlüsselung der Strahlungsenergiedichteverläufe in Bereichen mit starken Gradienten können dort verfeinerte Netze oder höherwertige Ansätze gewählt werden.

Das Radiosity-Verfahren kann nur *diffuse* Reflexion berücksichtigen, da nur die diffuse Reflexion durch die Energiedichteverteilung beschrieben werden kann. Glanzeffekte sind hingegen der Strahlenoptik zuzuordnen. Will man Glanzeffekte durch spiegelnde Reflexion berücksichtigen, so bietet sich theoretisch ein nachgeschaltetes einfaches Ray-Tracing mit geringer Rekursionstiefe an (Schlaglichter). Allerdings können die so ermittelten Intensitäten nicht einfach zu denen des Radiosity-Verfahrens addiert werden, da die beiden Modelle inkompatibel sind.

Lichtquellen werden vom Radiosity-Verfahren wie alle anderen Oberflächen behandelt, d.h., die räumliche Ausdehnung und Geometrie der Lichtquellen geht in die Berechnung ein. Eine Lichtquelle ist einfach eine Oberfläche mit *vorgebener* Strahlungsenergiedichte (Randbedingung der Differentialgleichung). Somit können beim Radiosity-Verfahren auch die Lichtquellen im Bild sichtbar sein, während man beim Ray Tracing implizit von einer Beleuchtung einer Szene durch "außerhalb" angeordnete Lichtquellen ausgeht. Für die Untersuchung von Beleuchtungsverhältnissen unter künstlichem Licht ist daher ausschließlich das Radiosity-Verfahren geeignet.

Da diffuses Licht ungerichtet ist, ist das Berechnungsergebnis des Radiosity-Verfahrens von der Position des Beobachters unabhängig. Damit ist das Radiosity-Verfahren das wichtigste Verfahren bei

Generierung von *virtueller Realität* (*virtual reality*, *VR*). Darunter wird das benutzergesteuerte Echtzeit-Navigieren in photorealistischen Visualisierungen räumlicher Szenen verstanden. Auch in der Architektur spielen heute VR-Anwendungen eine wichtige Rolle.

Radiosity-Bilder können von Ray-Tracing-Darstellungen durch die weicheren Konturen der Schatten unterschieden werden. Da das Radiosity-Modell auf die Wellengleichung aufbaut, gibt es Welleneffekte wie Beugung gut wieder und zeichnet Schatten deshalb nicht als scharf begrenzte Bereiche, sondern mit fließenden Übergängen.

3.7 Graphisch-interaktive, fensterorientierte Systeme

CAD-Programme sind *interaktive* Programme, d.h. ein CAD-Programm bearbeitet nicht einfach sequentiell eine vorgebene Anweisungsfolge, sondern wird vom Benutzer während der Laufzeit durch Eingaben in seinem Ablauf beeinflußt.

Der Dialog zwischen Mensch und Maschine bzw. zwischen Mensch und Programm wird als Mensch-Maschine-Interaktion (MMI) bezeichnet. Es sind unterschiedliche Formen (*Modi*) der Interaktion möglich, z.B. *sprachliche Interaktion* (z.B. Eingabe von Befehlen in einer Kommandosprache über Tastatur, natürlichsprachliche, gesprochene Anweisungen, usw.) oder *graphische* Interaktion.

Im CAD spielt die graphische Interaktion die größte Rolle.

3.7.1 Ereignisgesteuerte Systeme

Soll der Benutzer Gelegenheit erhalten, während der Laufzeit Eingaben zu tätigen und damit den Lauf des Programmes zu beeinflussen, so muß der Ablauf des Programmes regelmäßig *unterbrochen* werden, um die Benutzereingaben einzulesen und zu verarbeiten.

Dabei sind drei Eingabemodi möglich:

- *Anforderungsmodus (Request Mode).* Das Programm fordert Eingaben von einem Eingabegerät und wartet solange, bis der Benutzer die erwartete Eingabe vornimmt. Beispielsweise könnte das Anwendungsprogramm auf einen Mausklick oder einen Tastendruck warten. Das Programm gibt dabei den Zeitpunkt und die Form der Mensch-Maschine-Interaktion vor, indem es den Benutzer zu einer Eingabe auffordert. Solche Interaktionsmethoden sind für Programme mit einem starren Ablauf (z.B. Auskunftssysteme) tragbar, nicht jedoch für CAD-Anwendungen. Für CAD-Programme ist ein solcher Eingabemodus indiskutabel.
- *Sampling (Polling).* Das Programm fragt in regelmäßigen Abständen bei allen an den Rechner angeschlossenen Eingabegeräten nach, in welchem Zustand sich das jeweilige Eingabegerät befindet (z.B. Position und Tasten der Maus). Hat sich gegenüber einem früheren Zustand etwas geändert, so wird dies als Benutzeraktion interpretiert, und das Programm reagiert auf die Eingabe durch Aufruf entsprechender Funktionen. Durch das ständige Abfragen des Zustandes aller Eingabegeräte wird die Ausführung des eigentlichen Programmes verzögert, und damit kann die Ausführungsgeschwindigkeit zäh wirken. Daher ist auch dieser Interaktionsmodus heute veraltet.
- *Ereignismodus (event mode).* In diesem Modus wird das Auslösen einer Eingabe durch den Benutzer des Rechners als "Ereignis" interpretiert. Ereignisse können von jedem an den Rechner angeschlossenen Eingabegerät kommen. Typische Ereignisse sind z.B. das Drücken einer Taste oder das Bewegen der Maus um einen gewissen Mindestfahrweg. Tritt ein solches Ereignis ein, so wird eine sogenannte *Unterbrechungsanforderung* (*interrupt*) an das Betriebssystem gesandt. Das Betriebssystem hält dann alle Programme für kurze Zeit an und speichert die Information über das Ereignis in einer *Ereigniswarteschlange (event queue)*. Die Behandlung der Ereignisse durch einzelne Programme verläuft vollkommen unabhängig von der Aufnahme der Ereignisse in die Ereigniswarteschlange. Sobald eines der auf dem Rechner laufenden Programme "Zeit findet", sich

um eventuell eingetroffene Ereignisse zu kümmern, fragt es die Ereigniswarteschlange des Betriebssystems ab. Finden sich in der Warteschlange Ereignisse, die dem betreffenden Programm zugeordnet werden können, so werden die Ereignisse in der Reihenfolge ihres Eintreffens behandelt.

Heute ist nur noch der *Ereignismodus* üblich. Findet das Programm beim Abfragen der Ereigniswarteschlange kein Ereignis, so wartet das Programm bis zum Eintreffen des nächsten Ereignisses, oder es führt noch zu erledigende "ältere" Arbeiten fort. Treffen die Ereignisse schneller ein, als sie vom Programm verarbeitet werden können, so "hinkt" die Ausführung hinter den Ereignissen hinterher. Dieses Verhalten ist nicht erwünscht. Meist ist die Größe der Ereigniswarteschlange begrenzt, und Ereignisse, die bei voller Warteschlange eintreffen, werden ignoriert. Damit ist das Hinterherhinken der Applikation hinter den Ereignissen ebenfalls beschränkt. Ein Warnsignal sollte in solchen Fällen den Benutzer darauf aufmerksam machen, daß die Ereigniswarteschlange überfüllt ist.

3.7.2 Fensterorientierte Systeme

Graphisch-interaktive Benutzungsoberflächen (GUI=*graphical user interface*) sind symbolische graphische Darstellungen auf dem Computerbildschirm, die dem Benutzer die Interaktion mit einem Programm durch Auslösen graphischer Eingaben gestatten. Bild 3.24 zeigt als Beispiel die graphische Benutzungsoberfläche des CAD-Systems AutoCAD R. 14.

Ein graphisch-interaktives System baut auf die *Fenstertechnik* auf. Ein *Fenster* ist heute in aller Regel ein rechteckiger, achsenparalleler Teilbereich des Bildschirms. Fenster können sich auch überlappen. Ein *fensterorientiertes System* verwaltet eine Vielzahl von Fenstern auf dem Bildschirm.

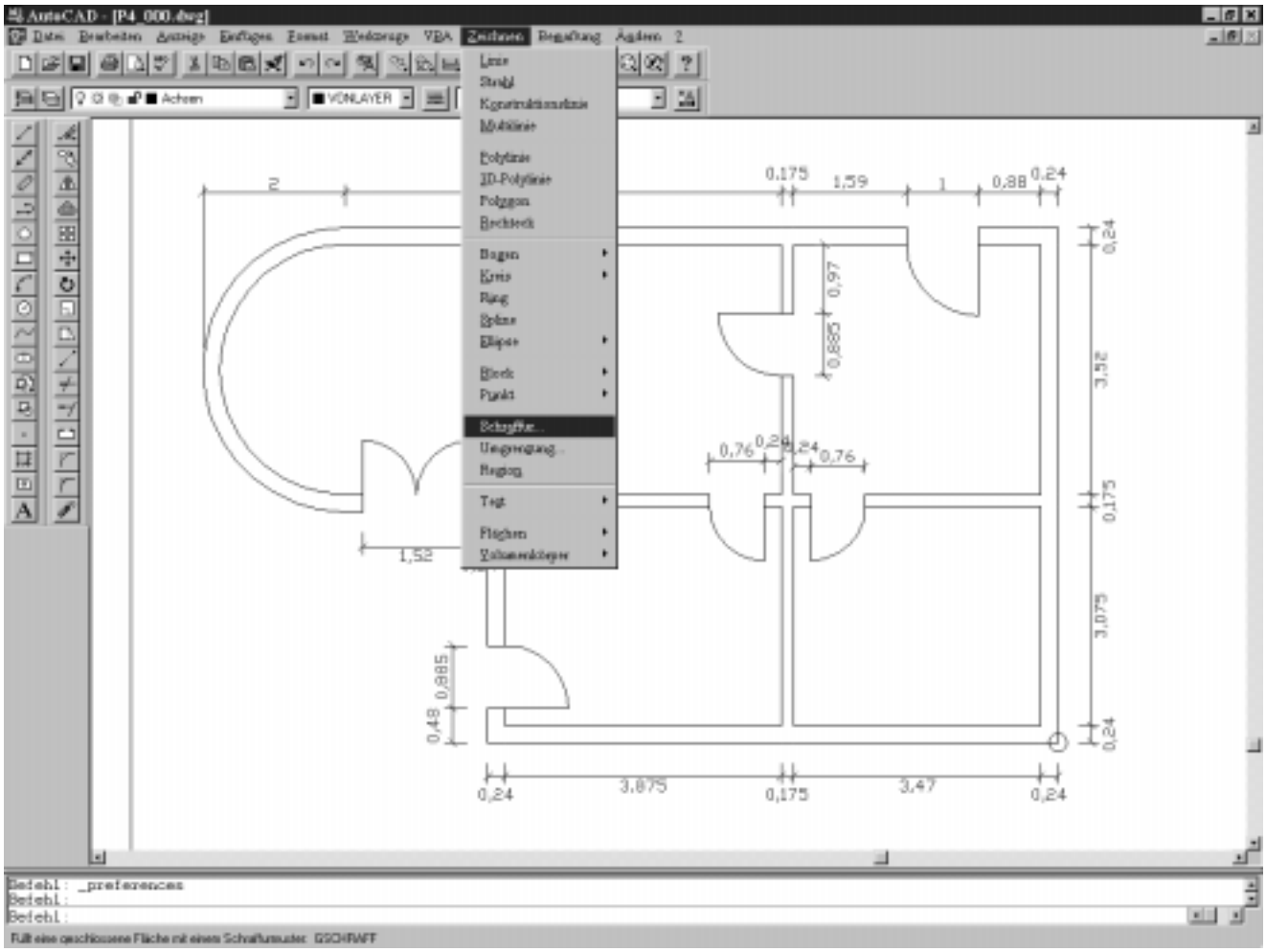

Bild 3.24: Graphische Benutzungsoberfläche (GUI) des CAD-Programmes AutoCAD R. 14 (AutoDesk).

In seltenen Fällen können Fenster auch kreisförmige oder beliebige andere Gestalt haben; für unsere weiteren Betrachtungen spielt die geometrische Form der Fenster keine Rolle.

Alle "Objekte", die zu einer graphisch-interaktiven Benutzungsoberfläche gehören, können abstrakt als "Fenster" angesprochen werden. Der Begriff "Fenster" deckt also nicht nur "Zeichenbereiche" oder "Eingabefelder" ab, sondern sämtliche Komponenten einer Benutzungsoberfläche, unabhängig von ihrem Aussehen (Knopf, Symbol, Menüleiste, Bildlaufleiste usw.).

Kern eines *fensterorientierten Systems* ist ein sogenannter *Window-Manager*. Er kann entweder in das Betriebssystem integriert sein (z.B. bei den Windows-Betriebssystemen) oder als ein eigenes Programm vorliegen (z.B. im System X). Seine Aufgabe ist es, die Fenster, aus denen sich die Benutzungsoberfläche zusammensetzt, zu verwalten, und die Zuordnung von eintreffenden Ereignissen zu den Fenstern herzustellen.

Jedes Fenster ist einer bestimmten Applikation (Anwendungsprogramm) zugeordnet. Wird durch den Benutzer ein *Ereignis* (z.B. Mausklick) ausgelöst, so wird von dem fensterorientierten System zunächst ermittelt, welchem Fenster bzw. welcher Applikation dieses Ereignis zuzuordnen ist.

Es gibt zwei verbreitete Modelle für die Zuordnung eines Ereignisses (z.B. Tastendruck) zu einem Fenster. Entweder ist maßgeblich, ob sich zum Zeitpunkt des Ereignisses der Eingabezeiger (*cursor*) innerhalb des Fensters befunden hat. In diesem Fall spricht man vom *Focus-follows-pointer-Modus* der Interaktion. Er ist im System *X* (s. unten) üblich. Eine andere Variante (z.B. in Windows) besteht darin, daß jeweils ein bestimmtes Fenster als "aktives", für alle Eingaben reserviertes Fenster auf dem Bildschirm markiert wird (*listener mode*). Alle Eingaben gehen dann zuerst an dieses aktive Fenster, und nur dann, wenn das betreffende Fenster mit dem Ereignis nichts anfangen kann, werden die Ereignisse an andere Fenster weitergeleitet. Solange ein Fenster aktiv ist, spricht man davon, das Fenster habe den *Eingabefokus*.

Der Fenster-Manager verwaltet eine Liste aller Fenster, die stets so geordnet ist, daß das "oberste" Fenster auf dem Bildschirm zuletzt gezeichnet, aber beim Zuordnen von Ereignissen ggf. zuerst gefunden wird. Die Fensterliste ist zu diesem Zweck nach *Prioritäten* sortiert (Bild 3.25). Das aktive Fenster ist üblicherweise das erste Fenster der Liste. Der Inhalt der Fenster wird in der umgekehrten Reihenfolge ihrer Priorität gezeichnet, so daß der Inhalt der Fenster mit der höchsten Priorität den der anderen Fenster übermalt und die Fenster höherer Priorität "oben" auf dem Bildschirm zu liegen scheinen. Das aktive Fenster wird dabei von keinem anderen Fenster verdeckt.

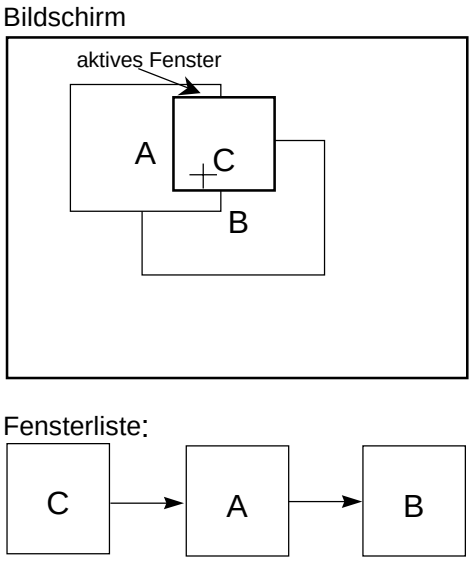

Bild 3.25: Fensterliste

Wird ein Fenster aus dem Hintergrund nach "vorne" geholt, so muß der Fensterinhalt restauriert werden. Dazu sendet der Window Manager dem Fenster eine Nachricht (*expose event*), und die Fensterliste muß reorganisiert werden (neue Prioritäten). Anstelle einer sortierten Liste wird in realen Implementierungen fensterorientierter Systeme eine Datenstruktur verwendet, die die Erhaltung eines sortierten Zustands auch bei einer Neubewertung der Fensterreihenfolge erleichtert (z.B. *Heap*-Datenstruktur).

Ein einzelnes Anwendungsprogramm kann über eine Reihe von Fenstern verfügen. In jedem der Fenster können Benutzerereignisse stattfinden. Abhängig vom Typ des Fensters werden die Ereignisse von einer Funktion als Anweisungen an das Programm gedeutet.

Eine graphische Benutzungsoberfläche setzt sich heute typischerweise aus *spezialisierten Fenstertypen* zusammen, deren Erscheinungsbilder schon die Funktionalität erahnen lassen. Beispiele sind:

- *Knopf, Schaltfläche* (*button*): Solche Fenster haben die äußere Erscheinungsform einer Taste, die mit einem Text oder einer Graphik gekennzeichnet ist. Durch Auslösen eines Mausklicks innerhalb eines Fensters dieses Typs wird in der Regel eine bestimmte Funktion aktiviert (*push button*). Kann von einer Reihe von Tasten jeweils nur eine gedrückt sein, während alle anderen Tasten der Gruppe gleichzeitig im Zustand "nicht gedrückt" sein müssen, so spricht man von einer Gruppe von *radio buttons*. Radio Buttons gestatten die Auswahl zwischen Alternativen. Können von einer Gruppe von Knöpfen mehrere gleichzeitig aktiv sein, so spricht man von *check buttons.*
- *Menü*: Ein Menü ist ein Fenster, das in mehrere Bereiche (Menüfelder) zerlegt ist, die mit verschiedenen Texten beschriftet sind. Abhängig davon, in welchem Teilbereich des Menüs der Benutzer eine Eingabe auslöst, werden verschiedene Kommandos angestoßen. Wird das Menü erst durch eine Benutzereingabe (z.B. Drücken der rechten Maustaste) auf den Bildschirm gebracht, so spricht man von einem *Pop-Up-Menü*. Wird ein Menü nach dem Wählen eines Menüfeldes durch weitere Menüs erweitert, so heißen diese Menüerweiterungen *Drop-Down-Menüs*.
- *Eingabefeld*: Ein Fenster dieses Typs akzeptiert Tastatureingaben, wenn sich die Eingabemarke innerhalb des Fensters befindet bzw. das Fenster den *Eingabefokus* hat (d.h. alle Tastatureingaben werden zunächst diesem Fenster zugeleitet), und stellt den eingegebenen Text (auch Zahlen) als Echo innerhalb seines Fensterbereiches dar.
- *Scroll Bar:* Unter diesem Begriff werden Fenster zusammengefaßt, die das äußere Erscheinungsbild eines "Schiebereglers" haben und es gestatten, durch graphische Interaktion einen skalaren Wert

festzulegen. Solche Schieberegler werden z.B. verwendet, um den Weltausschnitt, der in einem Fenster dargestellt ist, zu verschieben (Bildlaufleiste).

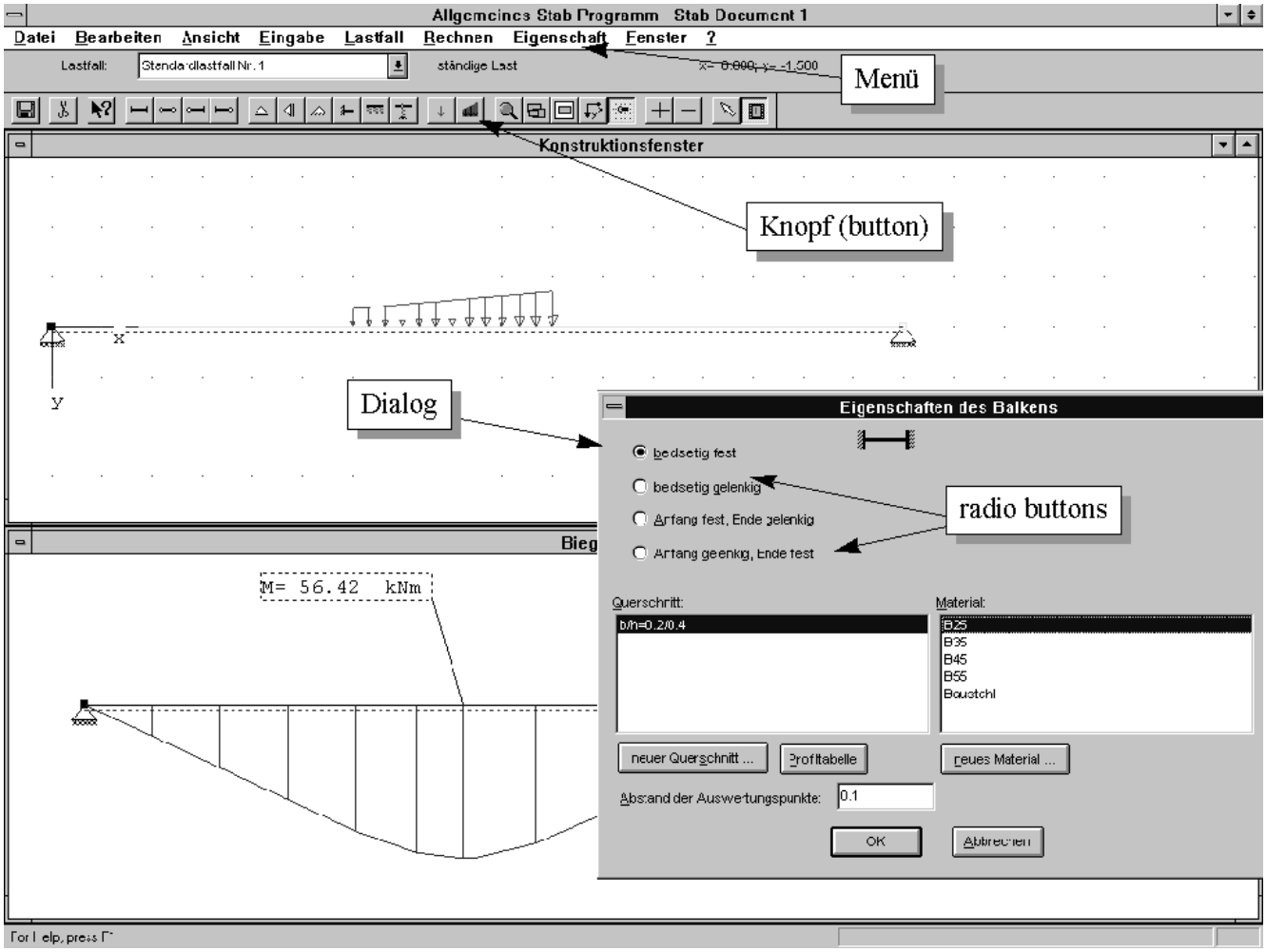

Bild 3.26: Elemente der graphischen Benutzungsoberfläche eines Statik-Programms

Fenster, die solche spezielle Gestalt haben und auf die Verarbeitung bestimmter Ereignisse spezialisiert sind, werden als *Widgets* bezeichnet (Diese Bezeichnung entstammt dem fensterorientierten Basissystem X11-Motif). Der Begriff *Widget* ist ein Kunstwort, das aus den Begriffen *Window* (Fenster) und *Gadget* (Instrument, Werkzeug) zusammengesetzt worden ist. Jedem Widget ist gegebenenfalls eine Reihe von Funktionen zugeordnet, die aufgerufen werden, sobald dem Widget-Fenster ein bestimmtes Ereignis zugeleitet wird, z.B. Drücken einer Maustaste innerhalb des Fensters. Diese Funktionen der graphisch-interaktiven Applikation werden als *Callback*-Funktionen bezeichnet.

Fenster können *hierarchisch* zusammengesetzt werden: Z.B. können mehrere Widgets in einem Rahmenfenster gruppiert werden. So stellt sich eine graphisch-interaktive Anwendung als *Baum* oder *Wald* (mehrere unverbundene Bäume nebeneinander) von Fenstern dar. Bild 3.26 (Bedienungsoberfläche eines Statik-Programms) zeigt eine Vielzahl von Fenstern, die als Behälter (*container*) für andere Fenster dienen, z.B. *Buttons*, die zu sogenannten *Tool Bars* (Werkzeugkästen) zusammengefaßt sind. Manche fensterorientierten Systeme (z.B. die GUI-Tools von Java) unterstützen ein automatisches Anordnen der Tochterfenster in einem Rahmenfenster und lassen auch ein dynamisches Verändern der Rahmenform und –größe zu.

Komplexere Interaktionen zwischen Anwender und Programm können durch sogenannte *Dialoge* abgewickelt werden. Dialoge sind Fenster (Widgets), die meist eine Reihe von anderen Widgets

enthalten (*container*). Ein Dialog sammelt mehrere Benutzereingaben und gestattet dem Benutzer den Zusammenbau einer Anweisung durch Auslösen verschiedener Ereignisse. In Bild 3.26 ist ein gerade geöffneter Dialog zu sehen, der dem Benutzer die Bearbeitung der Eigenschaften eines selektierten Objektes gestattet.

Kann der Benutzer den Dialog während der Dialogbearbeitung verlassen und Eingaben an andere Fenster senden, so wird von einem *nichtmodalen Dialog* gesprochen. Werden unabhängig von der Position des Mauszeigers nach dem Öffnen des Dialogfensters alle Ereignisse an den Dialog weitergeleitet, so daß der Benutzer den Dialog zuerst abschließen muß, bevor er in andere Fenster wechseln kann, so spricht man von einem *modalen Dialog*. Ein modaler Dialog reißt also den Eingabefokus an sich und behält ihn solange, bis eine komplexe Interaktion – z.B. das Ausfüllen eines Formulars – abgeschlossen ist. Mit solchen modalen Dialogen läßt sich eine streng sequentielle Komponente in ein graphisch-interaktives System einbringen, was für manche Eingaben und Interaktionen notwendig ist.

Graphisch-interaktive Anwendungen werden heute auf Basis vom kommerziellen Fenstersystemen entwickelt. Beispiele sind *Windows* (Microsoft), *Presentation Manager* (OS/2, IBM) sowie das System *X* (Unix-Systeme, heute *X11*, MIT), ergänzt z.B. durch den Window Manager und die *Widget Sets* von *Motif* (Open Software Foundation). Diese Systeme stellen jeweils ein *API* (*application programmer's interface*) zur Verfügung, so daß Anwendungsprogrammierer Dialogelemente und Widgets nicht mehr selbst entwickeln müssen.

Ein *ereignisgesteuertes* graphisch-interaktives System wird heute als *objektorientiertes System* entwickelt. Dabei wird die Funktionalität des fensterorientierten Systems nach dem *Message-Passing-Modell* beschrieben: Jedes Fenster der Applikation wird als eigenständiges Objekt (*event handler*) betrachtet, das mit anderen Objekten durch Austausch von *Nachrichten* kommunizieren kann. Findet ein *Ereignis* statt, so sendet der *Window Manager* allen von diesem Ereignis betroffenen Fenstern, nach ihrer Priorität sortiert, eine entsprechende *Meldung* bzw. Nachricht (*message*) in einem standardisierten Format (*kanonische Form*). Die Fenster ihrerseits interpretieren die eingetroffene Nachricht und senden gegebenenfalls neue Nachrichten an andere Komponenten der Benutzungsoberfläche oder der Applikation. Die Event Handler (Widgets oder auch unsichtbare Ereignisverarbeitungsobjekte) "synthetisieren" also neue "Ereignisse", die ganz analog wie die ursprünglichen, vom Benutzer ausgelösten Ereignisse weiterverarbeitet werden. Das prinzipielle Verhalten (Funktionalität) der einzelnen Event Handler wird durch eine allgemeine Beschreibung (Schablone, in der objektorientierten Programmierung "Klasse" genannt) festgelegt.

Die Benutzer-Programm-Interaktion kann in einem ereignisgesteuerten System durch Zustandsübergangsdiagramme beschrieben werden (*transition networks*).

Man kann die graphische Interaktion auch als Variante einer sprachlichen Interaktion verstehen und dann Benutzeroberflächen mit Techniken beschreiben, die aus der Theorie der formalen Sprachen (Syntaxdiagramme, Grammatiken, Backus-Naur-Notation) bekannt sind. Man kommt dann zum sogenannten *Seeheim*-Modell für graphisch-interaktive Anwendungen. In diesem Modell wird eine graphisch-interaktive Anwendung gedanklich in drei Prozesse oder Module zerlegt, die miteinander kommunizieren: Eine Darstellungskomponente, die die graphischen Benutzereingaben als "Wörter" einer Eingabesprache umdeutet (*lexikalische Analyse*), eine Dialogsteuerungskomponente, die die *syntaktische Analyse* der eingabesprachlichen Sätze übernimmt, die entsprechenden Befehle an die Applikation weiterleitet, und die Reaktion der Applikation in die graphisch übermittelbare Form zurückübersetzt, und die Applikationskomponente selbst, die die Befehle des Benutzers ausführt (*semantische Analyse*).

Geht man vom objektorientierten Modell der Ereignisverarbeitung aus, so bietet es sich ebenfalls an, eine Trennung der drei Bereiche Visualisierung, Interaktion und Applikation vorzunehmen. Bei der

Entwicklung graphisch-interaktiver Anwendungen hat sich das erstmals in der objektorientierten Programmiersprache *Smalltalk* eingeführte *Model-View-Controller-Paradigma* (MVC) durchgesetzt. Dabei werden die Objekte einer Anwendung in drei Domänen gruppiert: Die *View*-Komponente ist für die graphische Präsentation zuständig und enthält keinerlei applikationsspezifische Informationen; diese Daten und Funktionen sind vielmehr in den Objekten des *Modells* verankert. Die *Controller*-Komponente enthält Objekte, die Benutzer-Programm-Dialoge bereitstellen. Die strenge Trennung der drei Bereiche hilft dem Anwendungsentwickler, wiederverwendbare Programmkomponenten zu entwickeln. Häufig lassen sich allerdings View- und Controller-Komponente kaum trennen.

Dies gilt insbesondere für Anwendungen, die *direkte Manipulation* zulassen. Unter direkter Manipulation wird die Möglichkeit verstanden, die graphische Darstellung von Modell-Objekten am Bildschirm zu selektieren und dann eine Eingabeaktion durchzuführen, deren Auswirkung auf das Objekt direkt und in Echtzeit am Bildschirm mitverfolgt werden kann (*click, drag and drop*). Ein typisches Beispiel für die direkte Manipulation ist das Löschen oder Drucken einer Datei in graphisch-interaktiven Dateiverwaltungssystemen durch Anwählen eines Dateisymbols und Ziehen mit der Maus zu einem Abfallkorb- oder Druckersymbol. Zum Beispiel folgt die streng objektorientierte graphische Oberfläche des Betriebssystems OS/2 diesem Prinzip.

Plattform- und systemübergreifende Standardmethoden der direkten Manipulation (z.B. OS/2, Windows, Motif) sind *Selektieren* eines einzelnen Objektes durch Mausklick, *Verschieben* eines selektierten Objektes durch Verschieben der Maus bei gleichzeitig ständig gedrückter Maustaste, *Löschen* eines Objektes durch Verschieben auf ein Löschsymbol, *Erzeugen* eines Objektes durch Selektieren einer Objektschablone und Verschieben dieser Schablone an die gewünschte Stelle und *Aktivieren* eines Objekts (z.B. für die dialogische Bearbeitung oder zum Öffnen eines symbolisch dargestellten Fensters) durch Doppelklick mit der Maus. Diese plattformübergreifenden Direktmanipulationstechniken werden auch als *generische Funktionen* bezeichnet. Die einheitliche, konsistente Verfügbarkeit dieser Funktionen erleichtert Neulingen und nur gelegentlichen Anwendern den Einstieg in neue Programme erheblich.

In CAD-Anwendungen spielt heute das Prinzip "Click, Drag and Drop", d.h. die graphische Direktmanipulation von Objekten, eine herausragende Rolle. Dies bedeutet, daß man z.B. Zeichnungsobjekte direkt selektieren und durch Ziehen mit der Maus verschieben, rotieren, skalieren oder scheren kann. Die direkte Manipulation ist die unmittelbare Folge aus einer konsequenten Anwendung des Prinzips der *Objektorientierung*: Zuerst wird das zu manipulierende Objekt ausgewählt, und erst dann wird (in Abhängigkeit von den Manipulationsmöglichkeiten des konkret selektierten Objekts) eine Funktion ausgewählt, die auf das selektierte Objekt angewendet werden soll.

Allerdings trifft man in CAD-Anwendungen auch die *funktionsorientierte* Bearbeitung noch häufig an: In diesem Interaktionsprinzip wählt man zuerst (z.B. aus einem Menü) die gewünschte Aktion aus und spezifiziert erst dann (z.B. durch Mausklick) das Objekt, auf das diese Aktion angewendet werden soll.

Das optische Erscheinungsbild graphischer Benutzungsoberflächen und die Funktionalität der Widgets (*look and feel*) hat sich in den letzten Jahren plattformübergreifend angenähert; dennoch bestehen plattformspezifische Eigenheiten in Darstellung und Funktionalität. Sie werden für Applikationsentwickler in sogenannten *Style Guides* festgeschrieben. Kommerzielle Anwendungen sollten sich so eng als möglich an den Style Guides orientieren, um Neueinsteigern und gelegentlichen Anwendern den Einstieg bzw. die Orientierung möglichst zu erleichtern.

3.7.3 Algorithmen und Datenstrukturen zum Selektieren graphischer Objekte

CAD-Programme sind *interaktive* Programme, d.h., der Benutzer "ergreift" Objekte und gibt im "Dialog" mit dem Rechner an, was mit den ausgewählten Objekten geschehen soll.

Daher besteht die häufigste Aufgabe, die ein CAD-System zu lösen hat, darin, in möglichst kurzer Zeit alle Objekte aufzufinden, die sich (z.B. bei einem Mausklick) in der Nähe des Mauszeigers befinden, und diese zu selektieren. Sobald ein Objekt selektiert ist, können dann Operationen auf die Selektion angewendet werden. Wir wollen uns daher kurz mit den Grundlagen des Selektierens vertraut machen.

3.7.3.1 Selektieren von Punkten

Im einfachsten Fall gibt der Benutzer beim Selektieren eines Punktes mit dem Zeigegerät eine Position auf dem Bildschirm an. Durch diese Operation werden alle auf dem Bildschirm sichtbaren Punkte selektiert, die sich innerhalb eines gewissen Abstandes von der aktuellen Position des Mauszeigers befinden.

Dieser Abstand wird als *Fangradius* bezeichnet. Seine Größe wird auf die Maßeinheit des Bildschirms bezogen, ist also vom Abbildungsmaßstab unabhängig.

Die Umgebung des Mauszeigers, innerhalb derer Punkte erfaßt werden, muß nicht notwendigerweise kreisförmig sein; für praktische Zwecke reicht oft ein kleines, zu den Bildschirmkanten paralleles Quadrat als Suchbereich aus.

Will man beim Selektieren einen kreisförmigen Fangbereich verwenden, so muß man für jeden auf dem Bildschirm bzw. im betreffenden Fenster dargestellten Punkt den Abstand zum Mauszeiger bestimmen. Dazu transformieren wir die Position des Mauszeigers und den Fangradius in Weltkoordinaten (Punkt (x_c, y_c) und Radius r_{fang}) und berechnen $r = \sqrt{(x - x_c)^2 + (y - y_c)^2}$. Falls $r < r_{fang}$, gilt der untersuchte Punkt als selektiert und wird in die Liste der aktuell selektierten Punkte übertragen.

Die Berechnung der Quadratwurzel ist sehr zeitaufwendig. Da das Selektieren schnell gehen soll, formulieren wir die Bedingung für das Selektieren besser als $r^2 < r_{fang}^2$. Außerdem kann man die Funktion "Quadratwurzel" mit hinreichend guter Genauigkeit durch die ersten 1-3 Glieder einer Reihenentwicklung ersetzen und damit die Berechnung beschleunigen.

Noch einfacher fällt der Test aus, wenn wir eine quadratische Form des Fangbereichs verwenden: Wir brauchen dann nur maximal vier Vergleichsoperationen zwischen den Koordinaten des vom Benutzer angegebenen Punktes und den Bereichsgrenzen des Fangbereichs, um die Entscheidung "außen" oder "innen" treffen zu können. Bei der Mehrzahl der Punkte, die nicht innerhalb des Fangbereichs liegen, werden wir schon nach ein oder zwei Tests abbrechen können.

Auch die Mehrfachselektion mit Hilfe eines "Gummirechtecks" läßt sich auf die Selektion durch Mausklick an einer bestimmten Stelle zurückführen, wenn wir rechteckige Fangbereiche verwenden. Dabei entspricht die Größe des "Fangrechtecks" dann einfach der Größe und Ausdehnung des vom Benutzer "aufgezogenen" Suchbereichs (Gummirechtecks).

Sucht man in einer sehr großen Zeichnung Punkte in der Nähe des Mauszeigers, so ist für die Geschwindigkeit des Suchvorganges die interne Organisationsform der Daten im Programm weitaus wichtiger als die Rechenzeit, die zur Überprüfung des Abstandes erforderlich ist.

Im einfachsten Falle verwaltet man alle eingegebenen Punkte - unabhängig von ihrer Position - in einer Tabelle. Um festzustellen, welche Punkte vom Benutzer selektiert worden sind, muß man dann allerdings die gesamte Tabelle durchsuchen (lineare Suche) und für jeden Punkt den Abstand zur vom Benutzer eingegebenen Mausposition berechnen, um zu ermitteln, ob sich der Punkt innerhalb des Fangradius befindet. Der Suchaufwand wächst dann linear mit der Anzahl der Punkte der Zeichnung. Bei einer großen Anzahl von Objekten entsteht so sehr schnell ein übermäßig großer Suchaufwand, der auch bei leistungsfähigen Rechnern zu unakzeptabel großen Laufzeiten bzw. zu einer für das interaktive Arbeiten inakzeptablen "*Antwortzeit*" des Programmes (Zeit zwischen der Benutzeraktion und der Reaktion des Programmes) führen kann.

Zur effizienten Bearbeitung solcher *geometrischer* Anfragen ist es daher notwendig, die Punkte und Linien einer Zeichnung in einer entsprechenden Datenstruktur zu verwalten, die nach *geometrischen* Regeln aufgebaut wird. Eine solche Organisationsform der Daten ermöglicht es, schon nach wenigen Schritten einen großen Anteil der Punkte einer Zeichnung aus dem weiteren Suchprozeß auszuschließen.

Um eine solche günstige Organisation der Punktdaten zu erreichen, bauen wir auf eine schrittweise, hierarchische, *rekursive* Aufteilung der Zeichenfläche auf. Wir kommen damit wiederum zu einem *Quadtree* (vgl. Kapitel 2.3), jedoch zu einer speziellen Form dieser Datenstruktur, bei der die Geometrie der Quadranten durch die eingegebenen Punkte bestimmt wird.

Bild 3.27 zeigt als Beispiel einen Ausschnitt aus einer Menge von Punkten, die in einer zweidimensionalen Ebene angeordnet sind.

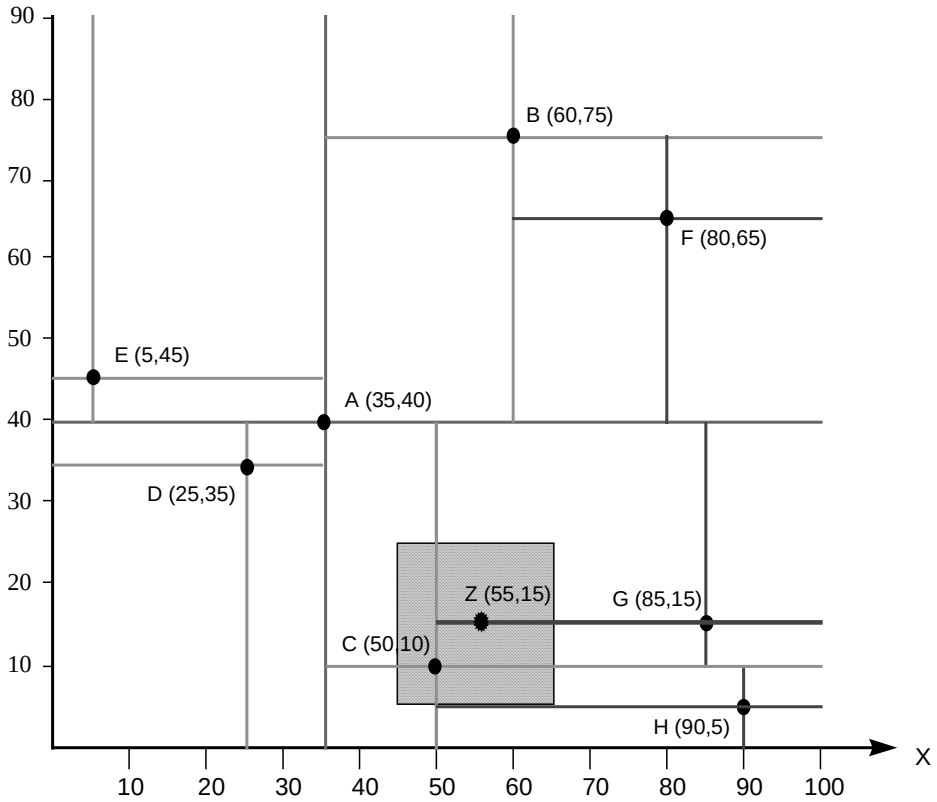

Bild 3.27: Zweidimensionaler Zeichenbereich mit 8 Punkten.

Um in einer großen Punktmenge effizient suchen zu können, muß die zu durchsuchende Punktmenge systematisch in kleinere Mengen aufgeteilt werden. Bild 3.27 zeigt eine rekursive Aufteilung der Ebene in "Quadranten", die entsteht, wenn die Punkte der Zeichnung in der Reihenfolge {A, B, C, D, E, F, G, H} eingegeben werden, wobei jeder neue Punkt eine weitere Unterteilung des Quadranten, innerhalb dessen er gelegen ist, definiert. Dadurch entsteht eine von den Eingabedaten und der Reihenfolge der Eingabedaten *abhängige*, "geschachtelte" Aufteilung der Ebene in "Quadranten". Die Quadranten haben jetzt unter Umständen stark unterschiedliche Größe, im Gegensatz zu den *a priori* festgelegten Quadranten, die wir im Kapitel 2.3 kennegelernt haben.

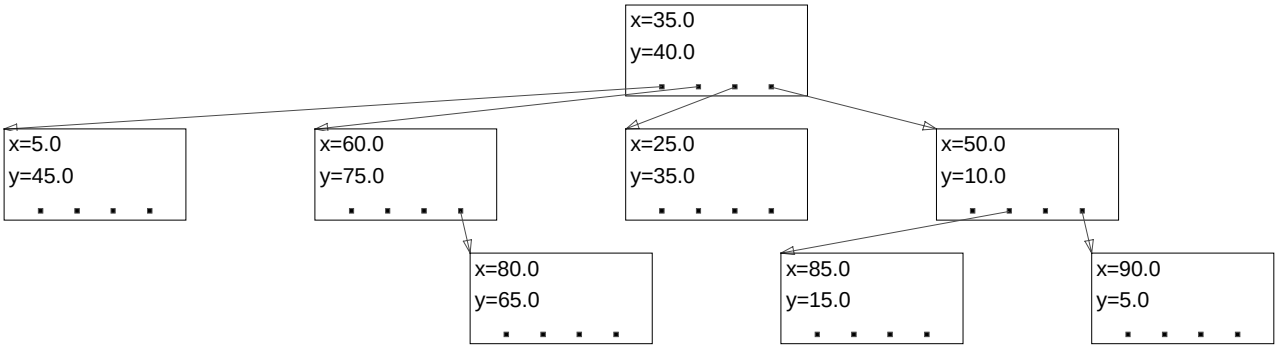

Bild 3.28: Zur Punktmenge in Bild 3.27 gehöriger Punkte-Quadtree.

Die in Bild 3.27 dargestellte Aufteilung der Ebene kann durch die bereits bekannte Datenstruktur "Quadtree" gespeichert werden. Bild 3.28 zeigt einen Quadtree, der die Flächenaufteilung in Quadranten aus Bild 3.27 wiedergibt. Für die Darstellung haben wir die Quadranten und Unterquadranten so durchnumeriert, daß der linke obere Quadrant der erste, der rechte obere der zweite, der linke untere der dritte und der rechte untere der vierte ist.

Jeder Knoten des Quadtrees verweist auf seine bis zu 4 Nachfolger. Da *a priori* bekannt ist, daß von jedem Knoten maximal 4 Kanten ausgehen, reicht es, genau 4 Speicherplätze für die Verweise auf die Nachbarn bereitzuhalten. Außerdem müssen wir nun für jeden Knoten des Baums die Lage des zugehörigen Punktes auf der Zeichenebene speichern, da die Quadranten bei der Eingabe neuer Punkte im allgemeinen nicht mittig aufgeteilt werden. Die spezielle Form des Quadtrees aus Bild 3.28 wird als *Punkte-Quadtree* bezeichnet, während eine gleichmäßige mittige rekursive Aufteilung der Quadranten wie in Bild 2.5 zu sogenannten *bucket quadtrees* führt.

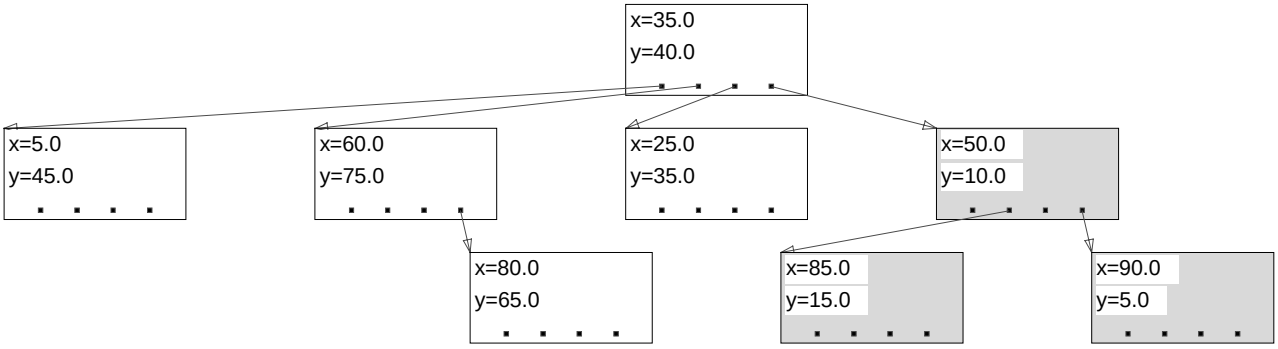

Bild 3.29: Teil des Quadtrees aus Abbildung 3.28, der bei der Suche nach Punkten innerhalb eines Suchquadrates der Seitenlänge 20 um den Punkt (55, 15) abgesucht werden muß. Es wird nur der Punkt (50,10) gefunden.

Geometrische Abfragen werden nun dadurch erleichtert, daß nur noch ein Teil der Punktmenge durchsucht werden muß. Sucht man nunmehr beispielsweise alle Punkte, die innerhalb des in Bild 3.27 schraffiert dargestellten Quadrates der Seitenlänge 20 mit dem Zentrum Z liegen (dieses Quadrat kann z.B. die "Fangumgebung" eines Cursors sein), so braucht man nur noch den 4. Quadranten (rechts unten) der Gesamtzeichnung zu untersuchen, da das Suchquadrat ausschließlich diesen Teil der Zeichnung abdeckt. Bild 3.29 zeigt den Teil der Datenstruktur, den wir nach Punkten absuchen müssen.

Um den betroffenen Teil des Baums zu ermitteln, müssen wir uns nur mit den Punkten A, C, G und H beschäftigen, während die anderen Punkte und Quadranten von vorneherein unberücksichtigt bleiben können. Auch für jede andere Position des Suchquadrates ist jeweils nur ein kleiner Ausschnitt des Baums von der Suche betroffen.

Um die von der Suchanfrage betroffenen Quadranten zu ermitteln, prüft man für die Wurzel des Baumes, ob sich seine Wurzel innerhalb des Suchrechtecks befindet oder nicht. Befindet sich die Wurzel nicht innerhalb des Suchrechtecks, so sind maximal zwei Quadranten des zugehörigen Teilbaums von der Suche betroffen; nur im Fall, daß die Wurzel innerhalb des Suchrechtecks liegt, müssen alle vier Teilquadranten abgesucht werden; in diesem Fall wird auf jeden Fall der Wurzelknoten gefunden, und weitere gesuchte Knoten können sich in den Teilquadranten befinden. Für jeden Teilquadranten wird der Suchalgorithmus wiederholt, bis man in jedem Teilast einen Knoten erreicht, der keine Nachfolger mehr besitzt.

Der Aufwand, der zur Durchführung einer Suche in einem Baum notwendig ist, hängt somit in erster Linie von der *Höhe* des Baums ab, also von der Anzahl der Ebenen des Baums. In einem Quadtree vervierfacht sich auf jeder Ebene die Anzahl der dort gespeicherten Elemente. In einem Quadtree der Höhe 2 können somit 21 Elemente, in einem Quadtree der Höhe 3 53 Elemente untergebracht werden. Die Anzahl der Ebenen, die bei einer Suche maximal durchlaufen werden müssen, wächst somit bei einem voll besetzten Quadtree proportional zu $\log_4 n$, wenn n die Anzahl der im Baum enthaltenen Elemente ist. In praktischen Anwendungen wird es meist nicht gelingen, den Baum so zu organisieren, daß er voll besetzt ist, so daß der Suchaufwand in der Praxis etwas höher ist. Verglichen mit dem proportional zu n wachsenden Suchaufwand in der einfachen Tabelle ergibt sich für großes n dennoch ein erheblicher Laufzeitvorteil.

Der Quadtree hat als Datenstruktur für die Punktsuche allerdings auch Nachteile: das Einfügen eines Punktes ist weitaus komplizierter als bei einer einfachen Tabelle und kostet damit auch mehr Rechenzeit. Noch viel schlimmer ist aber das Entfernen von Punkten aus der Datenstruktur. Der Baum muß reorganisiert werden, wenn der zu löschende Punkt an einer Ecke gespeichert war, die mehrere Nachfolger hatte. Die dazu erforderlichen Algorithmen sind kompliziert und würden den Rahmen der vorliegenden Vorlesung sprengen. Bei jedem Verschieben eines Punktes muß der Punkt zunächst aus der Datenstruktur gelöscht und dann an der neuen Position wieder eingefügt werden. Bei Eingabe von Punkten in ungünstiger Reihenfolge (z.B. Punkte, die alle auf derselben Geraden hintereinander folgen) kann der Punkte-Quadtree sehr unbalanciert geraten. Im Extremfall entartet der Baum dann zu einer linearen Liste. Einen Punkte-Quadtree durch Umsortieren der Punkte zu balancieren, ist eine extrem schwierige Aufgabe. Normalerweise ist bei üblichen Anwendungen ein Umsortieren jedoch nicht notwendig, da sich aus der Anwendung heraus nur selten derart ungünstige Punkteeingabefolgen ergeben.

Der Aufbau eines Quadtrees lohnt sich nur dann, wenn man erwarten kann, daß sehr häufig Selektiervorgänge in sehr großen Punktmengen vorgenommen werden sollen. Für kleine Punktmengen oder nur gelegentliche Selektionsvorgänge sind einfachere Organisationsformen der Daten vorteilhafter.

3.7.3.2 Selektieren von Linien und anderen Objekten

Im Fall der Selektion einer Linie müssen wir den minimalen Abstand zwischen dem Mauszeiger und der Linie ermitteln. Dabei müssen wir beachten, daß Linien einer Zeichnung keine Geraden sind, sondern begrenzte Länge besitzen.

Bei allen Objekten, die aus mehr als einem Punkt bestehen, ist es sinnvoll, zur groben Entscheidung, ob das Objekt bei einer Pick-Aktion des Benutzers selektiert worden ist, zunächst ein *bounding rectangle*, also ein das gesamte Objekt umhüllendes achsenparalleles Rechteck zu prüfen (Bild 3.30). Bei der Konstruktion dieses Rechtecks muß beachtet werden, daß es nach allen Richtungen um den Fangradius r_{fang} "aufgeblasen" werden muß. Liegt der Pickpunkt außerhalb des *bounding rectangle*, können wir getrost auf weitere Tests verzichten und wissen sofort, daß unser Objekt nicht selektiert worden sein kann. Um den Test auf das umhüllende Rechteck durchzuführen, vergleichen wir nacheinander die Koordinaten des Pickpunktes mit den Koordinaten des Rechtecks:

- Falls Pickpunkt "links" vom Rechteck, ist das Objekt nicht selektiert
- Falls Pickpunkt "rechts" vom Rechteck, ist das Objekt nicht selektiert
- Falls Pickpunkt "oberhalb" vom Rechteck, ist das Objekt nicht selektiert
- Falls Pickpunkt "unterhalb" vom Rechteck, ist das Objekt nicht selektiert

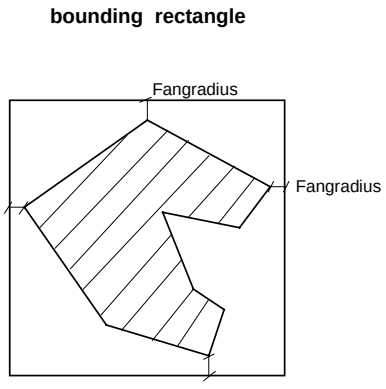

Bild 3.30: Untersuchung eines umgebenden Rechtecks als vereinfachte Vorstufe eines Pick-Tests

Erst dann, wenn wir alle diese Tests überstanden haben, müssen wir in eine detailliertere Prüfung einsteigen. Die vier Koordinatenvergleiche sind recht einfach und gehen schnell, während die weitere Prüfung aufwendig sein kann:

Ein Algorithmus zur Bestimmung des minimalen Abstands zwischen einem Punkt und einer Strecke könnte wie folgt lauten:

- Bestimme, ob die minimale Entfernung zwischen dem Pickpunkt $\begin{bmatrix} x_p \\ y_p \end{bmatrix}$ und einem der beiden Endpunkte der Strecke kleiner als r_{fang} ist. Falls ja, ist die Linie selektiert.

- Ermittle den Richtungsvektor $v = \begin{bmatrix} x_2 - x_1 \\ y_2 - y_1 \end{bmatrix}$ der Strecke
- Bilde das Lot auf diesen Vektor *v* (Vertauschen von x und y, Vertauschen eines der beiden Vorzeichen) und normiere das Lot. Das normierte Lot heiße *w*.
- Löse das Gleichungssystem $\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} + \lambda v = \begin{bmatrix} x_p \\ y_p \end{bmatrix} - \mu w$. Prüfe sodann, ob $0 \le \lambda \le 1$ und zugleich $|\mu| \le r_{fang}$. Falls beide Bedingungen erfüllt sind, ist die Linie selektiert, sonst nicht.

3.7.3.3 Selektieren von Polygonen

Auch bei Polygonen pflegen wir ein *bounding rectangle* und prüfen zunächst, ob der Pickpunkt innerhalb dieses Rechtecks liegt. Erst, falls dieser Test positiv ausfällt, werden wir einen aufwendigeren Punkt-in-Polygon-Test durchführen (Bild 3.31).

Man verwendet hierzu einen beliebigen Strahl (Halbgerade), der von dem Pickpunkt ausgeht, und zählt die Schnittpunkte zwischen diesem Strahl und der Begrenzungslinie des Polygons. Ist die Anzahl dieser Schnitte gerade, so liegt der Pickpunkt außerhalb des Polygons; liegt der Pickpunkt innerhalb, so ergibt sich eine ungerade Zahl von Schnittpunkten.

Dieser Test funktioniert auch bei nichtkonvexen Polygonen oder bei Polygonen mit Löchern. Am einfachsten ist er natürlich durchzuführen, wenn eine Halbgerade verwendet wird, die zur x- oder y-Achse parallel ist. Im Fall, daß der gewählte Strahl gerade durch eine Ecke des Polygons geht oder mit einer der Begrenzungskanten kollinear ist, sind zusätzliche Abfragen notwendig.

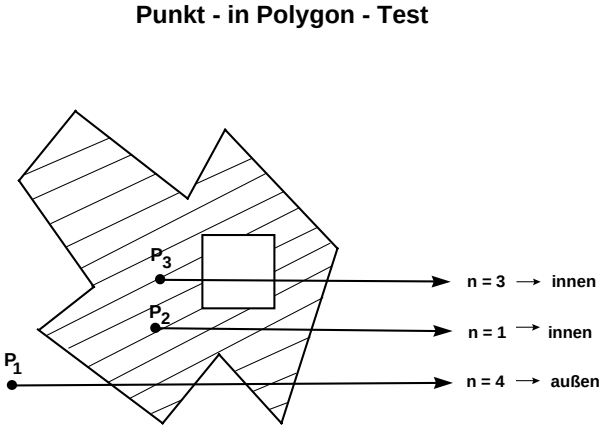

Bild 3.31: Punkt-in-Polygon-Test für beliebiges Polygon (auch mehrfach zusammenhängend, also mit Löchern).

Bei *konvexen Polygonen* ist ein einfacherer Test möglich: Ein konvexes Polygon ist die Schnittmenge einer Reihe von Halbräumen, die durch die Geraden, auf denen die Kanten des Polygons liegen, begrenzt werden. Daher reicht es zur Klärung der Frage, ob ein Punkt innerhalb des konvexen Polygons liegt, aus, das Polygon gegen den Uhrzeigersinn zu umfahren und für jede der dabei angetroffenen Kanten zu prüfen, ob der untersuchte Punkt auf der linken Seite der Kante liegt.

4. Integrierte Systeme

Die Gesamtheit aller Daten, die zur Beschreibung eines Aufgabenbereichs (z.B. Bauprojekt) notwendig sind, wird als *Datenbasis* (*data base*) bezeichnet. Die Datenbasis eines Bauprojektes kann eine beachtliche Größe erreichen, so daß automatisich die Frage nach der Verwaltung der Daten entsteht.

Will man diese Datenbasis tatsächlich als einheitliches Ganzes verwalten und nutzen, so bietet es sich an, zu diesem Zweck ein *Datenbankverwaltungssystem* (*data base management system, DBMS*) einzusetzen. Die Aufgabe eines solchen Systems ist die Steuerung des Zugriffs auf die Daten, einschließlich etwaiger Veränderungen an den Daten, und die Sicherung der Konsistenz (Widerspruchsfreiheit und Vollständigkeit) der gespeicherten Daten.

Wird eine Datenbasis durch ein Datenbankverwaltungssystem verwaltet, so spricht man von einer *Datenbank*.

Die Gründe, Daten in einem DBMS zu verwalten, sind vielfältig. Zum einen ist gerade in einem betrieblichen Umfeld die *Kontrolle der Zugriffsmöglichkeiten* wesentlich. Nicht jeder Mitarbeiter an einem Projekt hat gleiche Zugriffsrechte auf die Daten, jeder soll aber die ihm zugänglichen Daten optimal nutzen können. Ein DBMS bietet umfassende Möglichkeiten zur Regelung der Sicherheitsfragen.

Wichtiger noch ist die *Vermeidung von Redundanz und Inkonsistenz*. Werden Daten von verschiedenen Bearbeitern auf verschiedenen Rechnern eigenständig und unabhängig voneinander verwaltet, so ist es unvermeidlich, daß dieselben Informationen mehrfach gespeichert werden. Ändern sich zu einem späteren Zeitpunkt Teile der Daten, so ist es außerordentlich schwierig, alle Datenbestände, die von der Änderung betroffen sind, wieder auf den neuesten Stand zu bringen. So entstehen Widersprüche zwischen den einzelnen Dateien, mit der Folge potentiell schwerwiegender Fehler in der Projektbearbeitung. Dere Einsatz einer Datenbank vermeidet solche Fehler. Dabei ist es keineswegs erforderlich, daß die Datenbasis auf einem einzigen Rechner abgelegt ist, etwa nach dem Modell der Großrechenanlagen. Vielmehr kann eine einzige, logisch eine Einheit bildende Datenbasis durchaus auf mehrere Rechner verteilt sein (Verteilte Datenbank).

Datenbasen werden immer von einer Vielzahl von Personen genutzt. Verschiedene Personen können durchaus auch gleichzeitig auf eine Datenbasis zugreifen. Ohne Einsatz eines DBMS können hieraus beträchtliche Probleme entstehen, wenn z.B. zwei Benutzer denselben Datensatz gleichzeitig bearbeiten. Die Änderungen, die der erste Benutzer gemacht hat, gehen dabei verloren, wenn der zweite Benutzer seine geänderte Version des Datensatzes zurückspeichert. Ein DBMS vermeidet solche Anomalien.

Ganz gleich, welchem Aufgabenbereich eine Datenbasis dient, stets gilt eine Reihe von Bedingungen, denen die Daten genügen müssen. Dies können numerische Bedingungen sein (z.B. Volumen eines Baukörpers kann nie negativ sein), oder Bedingungen an die Beziehungen zwischen den Daten (z.B. Pfeiler kann erst erstellt werden, wenn das Fundament fertig ist). Die Einhaltung solcher Bedingungen, sogenannter *Integritätsregeln*, kann durch ein DBMS überwacht werden. Mehr noch, ein DBMS kann Änderungen an den Inhalten der Datenbank, die zu einer Verletzung von Integritätsregeln führen würden, ablehnen. Änderungen an Datenbanken werden im Rahmen von sogenannten *Transaktionen* durchgeführt. Eine Transaktion wird nur dann ausgeführt, wenn die Datenbank auch nach der Transaktion noch in einem konsistenten Zustand ist.

4.1 Aufbau einer Datenbank

In einer Datenbank ist die *logische Struktur* der Daten streng von der *physischen Speicherung* der Daten getrennt. Die logische Struktur wird durch ein *Datenbankschema* beschrieben. Sie gibt an, welche Typen von Daten (Datensätzen) in der Datenbank abgespeichert werden sollen, und welche Beziehungen zwischen diesen Daten bestehen. Das Datenbankschema gibt ausschließlich die *Struktur* der gespeicherten Daten an, sagt jedoch nichts über den aktuellen inhaltlichen Zustand (die aktuelle *Ausprägung*) der Datenbank.

Unter dieser *logischen Ebene* eines Datenbanksystems liegt die *physische Ebene*. Sie ist für den *Benutzer* oder *Anwender* einer Datenbank völlig uninteressant. In dieser Ebene wird angegeben, wie die Daten, die im allgemeinen auf dem Hintergrundspeicher (Plattenspeicher) liegen, im Speicher physisch organisiert sind. Natürlich hat die Art der Organisation der Daten entscheidenden Einfluß auf die Zugriffsgeschwindigkeit. Wenn der Entwickler einer Datenbank Änderungen an der physischen Ebene vornimmt, dürfen sich diese jedoch keinesfalls auf die logische Ebene auswirken.

Die wenigsten Benutzer einer Datenbank haben Zugriff auf die *gesamte* Datenbasis, sondern sind jeweils nur an Ausschnitten des Inhalts interessiert bzw. dazu berechtigt. Daher liegt bei einem Datenbanksystem zwischen der *logischen Ebene* und dem Benutzer im allgemeinen noch die Ebene der *Sichten*: Je nach Anwendungsbereich und Interesse des Benutzers wird diesem nur ein eigener, speziell aufbereiteter Ausschnitt des Inhalts der Datenbank geboten.

Der logische Aufbau einer Datenbank wird durch eine *Datendefinitionssprache* (*data definition language, DDL*) beschrieben. Für die Zugriffe auf die Informationen der Datenbank stellt diese eine *Datenmanipulationssprache* (*data manipulation language, DML*) zur Verfügung. Ein wichtiger Teilbereich der *DMA* ist die *Anfragesprache* (*query language*), mit der die Inhalte der Datenbank ermittelt und ausgewertet werdne können. Im folgenden werden wir noch die standardisierte DMA *SQL* (*structured query language*) kennenlernen.

Moderne Datenbanksysteme bieten über die "sprachliche" Kommunikation mittels einer DMA auch die Möglichkeit zu direkter graphisch-interaktiver Manipulation über eine graphische Benutzungsoberfläche (GUI).

4.2 Datenbankentwurf: Konzeptioneller Entwurf

Eine Datenbank dient der Speicherung v on Daten, die einen Ausschnitt der realen Welt beschreiben. Um von der Anschauung der realen Welt zu einem Datenbankschema zu gelangen, ist zunächst ein Abstraktionsschritt, die *konzeptuelle Modellierung*, erforderlich.

Die gängigste Methode, um ein konzeptuelles Modell zu entwerfen, ist der Einsatz eines *Entity-Relationship-Modells*. Die reale Welt wir dabei durch Gegenstandsmengen (*entities*) und Beziehungen (*relationships*) zwischen den Elementen dieser Mengen beschrieben. Sowohl die Entitäten als auch die Beziehungen können durch *Attribute* genauer beschrieben werden.

Jedes einzelne *Element* einer Entitätsmenge kann durch eine minimale Untermenge seiner Attribute eindeutig identifiziert werden. Diese Attributmenge wird als *Schlüssel* bezeichnet. In den meisten Datenbankanwendungen werden Entitäten durch ein einzelnes Schlüsselattribut identifiziert. Da natürliche Attribute wie Name und Vorname oft keine eindeutige Identifikation erlauben, werden meist künstliche Schlüssel (z.B. Matrikelnummer eines Studenten) eingeführt.

Eine Beziehung verbindet in der Regel zwei verschiedene Entitätsmengen. Es sind aber auch Beziehungen zulässig, die eine Entitätsmenge mit sich selbst verbinden (*rekursive Beziehung*).

Jede Beziehung kann bezüglich ihrer *Funktionalität* charakterisiert werden. Man spricht von einer 1:1-Beziehung zwischen zwei Entitätsmengen, wenn die Beziehung jedem Element der ersten Menge höchstens ein Element der zweiten Menge zuordnet und umgekehrt.

Bei einer 1:n-Beziehung gilt die Umkehrung der eindeutigen Zuordnung nicht: Dasselbe Element der ersten Menge kann beliebig vielen Elementen der zweiten Menge zugeordnet sein.

Bei einer n:1-Beziehung liegt dieselbe Situation, nur mit vertauschen Rollen der beiden Entitätsmengen, vor.

Von einer n:m-Beziehung schließlich spricht man, wenn jedes Element aus der ersten Entitätsmenge beliebig vielen Elementen der zweiten Menge zugeordnet sein kann und umgekehrt.

Die Funktionalität einer Beziehung stellt eine *Integritätsbedingung* für die Datenbank dar.

Die Funktionalität von Beziehungen wird häufig durch die (*min*,*max*)-Notation präziser charakterisiert.

4.3 Implementierung als relationale Datenbank

Die nächste Frage ist die, wie die ermittelten Entitätsmengen und Beziehungen durch die Strukturelemente einer DDL beschrieben werden können, wie also die Entitäten auf die in einer Datenbank vorhandenen Strukturen abgebildet werden können. Dieser Schritt wird als *Implementierung* bezeichnet.

In unserer Vorlesung wollen wir uns ausschließlich auf *relationale Datenbanken* beschränken. In solchen Datenbanken werden sowohl Entitätsmengen als auch Beziehungen durch *Tabellen* (*tables*) beschrieben. Diese Tabellen werden in einer relationalen Datenbank als *Relationen* bezeichnet. Das Aussehen der Tabelle wird durch das *Relationenschema* beschrieben.

Jede Zeile einer solchen Tabelle repräsentiert ein einzelnes Datenobjekt. Ein einzelnes Element einer Relation (eine Zeile) wird als *Tupel* bezeichnet. Die Anzahl von Tupeln, die in einer konkreten Ausformung der Tabelle enthalten sind, wird als *Kardinalität* der Relation bezeichnet.

Die Spalten der Tabelle entsprechen den Attributen des jeweiligen Objekts. Jedes einzelne Attribut entstammt einem bestimmten *Wertebereich* (*Gebiet, Domäne*). Attribute sind atomar. Mengen sind als Attribute nicht zulässig. Die Anzahl der Attribute, die zu jedem Tupel einer Relation gehören, wird als *Stelligkeit* oder *Grad* der Relation bezeichnet. Jedes Attribut einer Relation muß innerhalb der Relation (Tabelle) eindeutig durch einen *Namen* bezeichnet werden. Jedem Attribut ist in jedem Tupel ein *Attributwert* zugeordnet. Die Identifikation der Attribute erfolgt nicht über deren Spaltenposition, sondern ausschließlich über den Namen. Der Schlüssel, der ein Element einer Tabelle eindeutig identifiziert, wird als *Primärschlüssel* bezeichnet.

Eine Tabelle, die eine *Beziehung* modelliert, enthält als Attribute Schlüssel anderer Relationen. Solche Schlüssel werden als *Fremdschlüssel* bezeichnet. Eine wesentliche Anforderung an eine Datenbank, in der Relationen mit Fremdschlüsseln vorkommen, ist die, daß keine Relation einen Fremdschlüsselwert (außer dem symbolischen Wert *Null* für eine nicht vorhandene Beziehung) enthalten darf, der im dazugehörigen Primärschlüssel der zugeordneten Relation nicht existiert (*referentielle Integrität*).

Beim Datenbankentwurf können 1:1- und 1:n-Relationen, die denselben Primärschlüssel besitzen, vereinigt werden. Damit entsteht ein Relationenschema, in dem keine direkte Korrespondenz zwischen den Elementen des Entity-Relationship-Modells und den Relationen mehr besteht.

Relationen können durch *Operatoren* der *Relationenalgebra* bearbeitet werden. Die wichtigsten Operationen sind die gängigen Mengenoperationen Vereinigung (UNION), Schnitt (INTERSECT), Differenz (MINUS) und kartesisches Produkt (TIMES) sowie die speziellen Relationenoperatoren:

- Selektion (Restriktion). Auswahl von Tupeln (Zeilen), die eine Bedingung erfüllen, aus einer Relation. Beispiel: Auswahl aller Einträge aus einem Telefonbuch, die als Attributwert den Namen "Müller" enthalten. (SQL: WHERE)
- Projektion. Auszug von Attributen (Spalten) aus einer Relation. Duplikate werden eliminiert. Beispiel: Auswahl aller Orte aus einer Adreßdatenbank. (SQL: Attributauswahl
- Verbindung: Erzeugen einer neuen Relation durch Kombination gemeinsamer Schlüssel zweier Relationen (SQL: INNER JOIN ... ON)
- Division.
- Umbenennung einer Relation (SQL: AS). Erforderlich, wenn Attribute verschiedener Relationen g

4.4 Die relationale Anfragesprache SQL

SELECT: SELECT [ALL|DISTINCT] Spaltenliste
FROM Tabellenliste
[WHERE Bedingung]
[ORDER BY Spaltenliste desc|asc]
[GROUP BY Spaltenliste]

5. CAD-FEM-Kopplung

Ein Wunschtraum jedes Statikers und Konstrukteurs besteht noch immer darin, die rechnerbasierten Werkzeuge für Entwurf, Konstruktion und Berechnung direkt zu koppeln, also eine Integration von CAD-System und numerischen Berechnungsprogrammen auf Basis der Methode der finiten Elemente (*FEM=finite element method*) zu erreichen.

Das vorliegende Kapitel wendet sich den Chancen und Problemen einer solchen CAD-FEM-Kopplung zu. Dazu werden zunächst die Grundlagen der Finite-Elemente-Methode aus Sicht der Automatisierbarkeit kurz aufbereitet, sodann die Grundideen einer automatisierten Generierung von Finite-Elemente-Netzen vorgestellt und schließlich die Möglichkeiten und Grenzen einer CAD-FEM-Koppplung eingehend diskutiert.

5.1 Grundlagen einer automatisierten Finite-Elemente-Analyse

5.1.1 Methode der kleinsten Quadrate

Gegeben ist eine Funktion $f(x)$, die in einem Intervall $[0;L]$ "bestmöglich" zu approximieren ist, d.h. die durch eine einfachere Funktion ersetzt werden soll, die aber so wenig als möglich von der ursprünglichen Funktion abweicht. Als *Bestapproximierende* $g(x)$ bezeichnen wir eine Funktion, die wir so bestimmen, daß das Integral über die Abstandsquadrate $E = \int_0^L \left(f(x) - g(x)\right)^2 dx$ minimal wird. Dazu bauen wir die Approximation aus N linear unabhängigen Funktionen $p_i(x)$ auf: $g(x) = \sum_{i=1}^{N} p_i(x) a_i = \mathbf{p}^{\mathbf{T}}\mathbf{a}$ und bestimmen die Koeffizienten a_i derart, daß das Integral der Fehlerquadrate E minimal wird:

$$E = \int_0^L \left(f^2(x) - 2 f(x) g(x) + g^2(x)\right) dx = \int_0^L \left(f^2(x) - 2\,\mathbf{p}\, f(x) + \mathbf{a}^{\mathbf{T}}\mathbf{p}\,\mathbf{p}^{\mathbf{T}}\mathbf{a}\right) dx \geq 0$$

$$\frac{\partial E}{\partial a} = 0: \quad \int_0^L \mathbf{p}\,\mathbf{p}^{\mathbf{T}} dx\ \mathbf{a} = \int_{\mathbf{0}}^{\mathbf{L}} \mathbf{p}\, f(x)\, dx$$

oder: $\mathbf{A}\mathbf{a} = \mathbf{F}$

Die sogenannte GRAMsche Matrix der *Normalgleichungen* $A = \int_0^L \mathbf{p}\,\mathbf{p}^{\mathbf{T}} dx$ ist nach Konstruktion symmetrisch und positiv definit. Selbstverständlich können wir mit der genannten Methode nur solche Funktionen $f(x)$ approximieren, für die das Fehlerquadratintegral beschränkt bleibt. Das bedeutet insbesondere, daß $\int_0^L f^2 dx$ beschränkt sein muß.

5.1.2 Beurteilung der Approximationsqualität

Die erzielte Qualität der Approximation kann mit verschiedenen *Fehlermaßen* (Genauigkeitsmaßen) beurteilt werden. Am einfachsten können wir die Qualität einer Approximation nach der Methode der kleinsten Quadrate mit Hilfe des *mittleren Fehlers* beurteilen:

$$\|e\| = \sqrt{E}\ .$$

Das Fehlerquadratintegral E können wir aus dem Lösungsvektor **a** und der rechten Seite **F** ermitteln durch:

$$E = \int_0^L \left(f^2 - 2\mathbf{a}^\mathbf{T}\mathbf{p}f + \mathbf{a}^\mathbf{T}\mathbf{p}\mathbf{p}^\mathbf{T}\mathbf{a} \right) dx = \int_0^L f^2 dx - 2\mathbf{a}^\mathbf{T}\mathbf{F} + \mathbf{a}^T\mathbf{A}\mathbf{a} = \int_0^L f^2 dx - \mathbf{a}^\mathbf{T}\mathbf{F} \qquad \text{(I)}$$

Zum Vergleich verschiedener Probleme ist es sinnvoll, den Fehler auf $\int_0^L f^2 dx$ zu beziehen. Damit erhalten wir den *mittleren relativen Fehler*

$$\left\| e_r \right\| = \sqrt{\frac{\int_0^L f^2 dx - \mathbf{a}^\mathbf{T}\mathbf{F}}{\int_0^L f^2 dx}} \,. \qquad \text{(II)}$$

Wir untersuchen im folgenden die Approximationsqualität von Approximationen nach der Methode der kleinsten Quadrate in Abhängigkeit vom Typ der zu approximierenden Funktion *f(x)* und von der Wahl der Ansatzfunktionen $p_i(x)$. Als Testbeispiele untersuchen wir folgende Funktionen auf dem Intervall $[0;1]$:

a) $f(x) = \frac{1}{2}\left(\sin(\pi x) + \cos(\pi x)\right)$

b) $f(x) = x^{3/2}$

c) $f(x) = \sqrt{x}$

d) $f(x) = x^{-1/4}$

In Bild 5.1 sind diese vier Funktionen skizziert. Funktion (a) ist beliebig glatt, d.h. alle Ableitungen der Funktion haben im gesamten Intervall endliche Werte. Bei Funktion (b) hingegen ist nur die 1. Ableitung beschränkt, die höheren Ableitungen hingegen sind bei *x*=0 singulär. Bei Funktion (c) ist nur noch die Funktion selbst beschränkt, während alle Ableitungen singulär sind. Funktion (d) schließlich ist an der Stelle *x=0* singulär, jedoch so schwach singulär, daß $\int_0^L f^2 dx = 2$ noch existiert. Somit macht auch für Funktion (d) die Idee einer "Approximation mit der Methode der kleinsten Quadrate" noch Sinn.

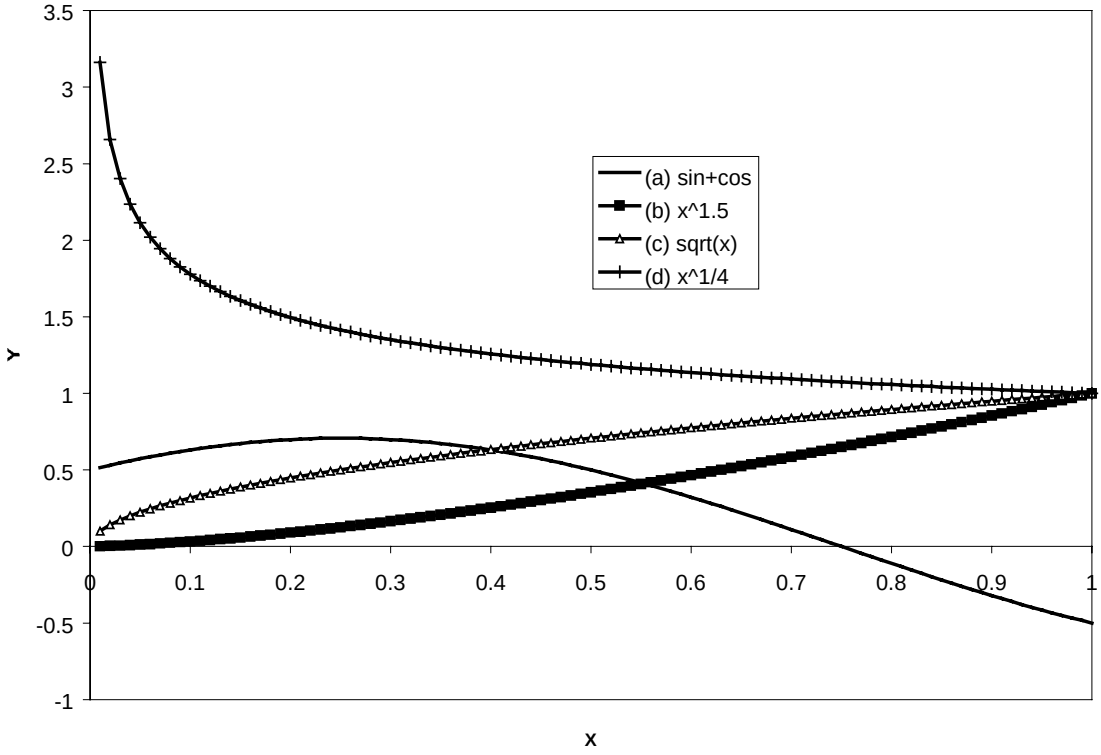

Bild 5.1: Verschiedene Funktionen, die mit der Methode der kleinsten Quadrate approximiert werden sollen.

5.1.3 *h*-Version: Approximation durch stückweise Polynome

In einem ersten Ansatz unterteilen wir das Intervall $[0;L]$ in *n* gleich lange Teilstücke, deren Länge wir mit *h=L/n* bezeichnen. In jedem dieser Teilintervalle approximieren wir die Funktion *unabhängig von den Nachbarintervallen* durch ein Polynom des Grades *p*, so bdaß *g(x)* durch insgesamt *N=n(p+1)* Koeffizienten bestimmt ist. An den Teilintervallgrenzen lassen wir jeweils einen *Sprung* in der Approximation zu.

Im einfachsten Falle wählen wir *p=0* und erhalten somit eine Approximation von *f* durch stückweise konstante Funktionen. In diesem Fall ist die Matrix **A** eine Diagonalmatrix.

Es geht nun darum, die zu erwartende Approximationsgenauigkeit abzuschätzen. Bei *p=0* ist die Gesamtzahl *N* der Koeffizienten gleich *n*. Wählen wir genügend viele Abschnitte *N*, so wird *h* genügend klein, um *f(x)* in der Umgebung $[x-h;x+h]$ genügend genau durch die nach dem linearen Glied abgebrochene Taylorentwicklung ersetzen zu können. Der Fehler hat dann näherungsweise stückweise linearen Verlauf. In der Umgebung eines Punktes x, der gerade an der Grenze zweier Abschnitte gelegen ist, bietet unsere Approximation die in Bild 5.2 dargestellte Situation:

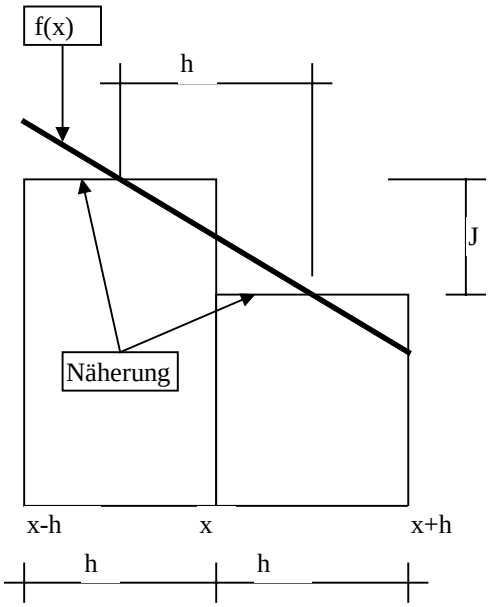

Bild 5.2: Zur Ermittlung einer Fehlerabschätzung für stückweise lineare Approximation

In jedem Abschnitt *h* muß der Fehler ausgeglichen sein, da wir die Approximation über die *N* Abschnitte aus *N* voneinander *unabhängigen* Teilaufgaben zusammengebaut haben (Das Fehlerquadratintegral wird minimal bei ausgeglichener Fläche der positiven und negativen Fehler). Wenn die Teilintervalle genügend klein sind, können wir die Funktion *f(x)* im Bereich zwischen *x-h* und *x+h* durch die Tangente in *x* ersetzen. Wenn das Fehlerquadrat in beiden Teilintervallen ausgeglichen sein soll, so ist dies dann nur noch dann möglich, wenn der Funktionswert an der Stelle *x* gerade der Mittelwert zwischen den beiden stückweise konstanten Approximationen in den angrenzenden Intervallen ist.

Bezeichnen wir den Sprung zwischen den stückweise konstanten Näherungen in den beiden Intervallen mit J_i, so können wir das Fehlerquadratintegral in dem Bereich zwischen den Mittelpunkten der beiden Abschnitte somit abschätzen durch:

$$\int_h (f-g)^2\,dx \approx 2\left(\frac{J_i}{2}\right)^2 \frac{1}{3}\frac{h}{2} = \frac{h}{12} J_i^2$$

Durch Auswertung aller Sprünge J_i zwischen den stückweise konstanten Approximationen benachbarter Intervalle können wir also näherungsweise den Gesamtfehler *E* der Approximation ermitteln: $E \approx \frac{h}{12}\sum_{i=1}^{N-1} J_i^2$. Lediglich an den Intervallenden können wir keine Sprünge ermitteln. Bei genügend vielen Intervallen fallen die Randintervalle jedoch kaum ins Gewicht.

Das folgende Bild 5.3 zeigt einen Vergleich zwischen dem exakten Fehler der Approximation (mit Hilfe von (I) ermittelt) und den durch Auswertung der Sprunggrößen näherungsweise ermittelten Fehlern für die Funktionen a) bis c) in Abhängigkeit von der Anzahl der Abschnitte *N* bei stückweise konstanter Approximation der jeweiligen Funktion *f(x)*.

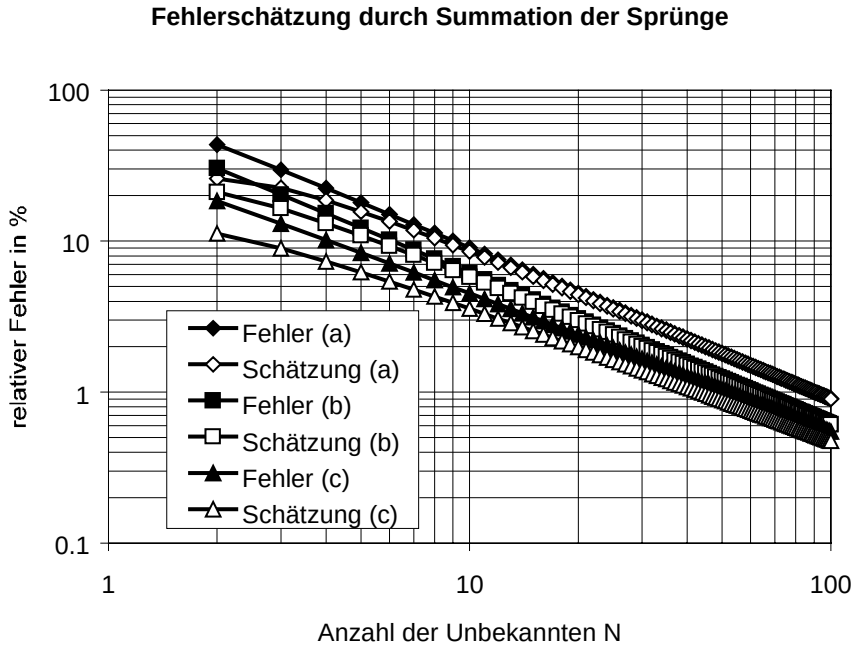

Bild 5.3: Vergleich zwischen Fehlerschätzung und wahrem Fehler.

Lediglich bei der Funktion (c) machen sich die in unserer Abschätzung nicht berücksichtigten Randintervalle deutlich bemerkbar, weil dort der größte Fehler ja gerade im Endintervall bei *x=0* auftritt.

In allen anderen Fällen konvergiert die mit Hilfe der Sprunggrößen geschätzte Genauigkeit schnell gegen den exakten Wert.

Wir haben in unserem einfachen Beispiel die Möglichkeit, den exakten Approximationsfehler zu ermitteln. In der Anwendung der *h*-Version bei der Finite-Elemente-Methode werden wir uns jedoch die Tatsache zunutze machen, daß wir den Approximationsfehler sehr gut durch Auswertung der Sprunggrößen auswerten können. Wir erhalten damit eine *A-posteriori*-Fehlerabschätzung.

Im zweiten Abschnitt soll es uns darum gehen, eine *A-priori*-Fehlerabschätzung zu bestimmen, d.h. zu einer Aussage zu kommen, mit welcher Konvergenzgeschwindigkeit wir bei Einsatz einer gleichmäßigen *h*-Verfeinerung bei abschnittsweise konstantem Polynomgrad *p* eigentlich rechnen können.

Um in diesem Fall zu einer Aussage zu kommen, müssen wir zunächst voraussetzen, daß die Steigung unserer Funktion *f(x)* im Intervall $[0; L]$ beschränkt ist, also innerhalb dieses Intervalls $\left|\frac{df}{dx}\right| \le C$ gilt.

Die maximale Größe eines Sprunges beträgt dann $J \le Ch$, und das Gesamtfehlerquadratintegral kann durch Addition der Einzelfehler aus den N Intervallen abgeschätzt werden zu

$$E \le \frac{h}{12}\sum_{i=1}^{N} J^2 = \frac{N h^3 C^2}{12} = \frac{L^3 C^2}{12 N^2} = \frac{k}{N^2} \quad .$$

Damit ergibt sich für große N folgender Zusammenhang für den mittleren Fehler (*algebraische Konvergenz*):

$$\|e\| = \frac{k}{N^{\beta}} \text{ mit } \beta = 1.$$

Ganz analog kann man sich auch vor Augen führen, daß der Fehler bei Approximation mit stückweise linearen, *an den Abschnittsgrenzen diskontinuierlichen* Funktionen von *quadratischer* Ordnung ist. Wir können wiederum jeden Abschnitt für sich betrachten, wenn wir die Approximationen in den Einzelabschnitten nicht stetig aneinanderbinden. Dieses Vorgehen mag hier etwas merkwürdig erscheinen, ist aber notwendig, um später die Analogie zur Finite-Elemente-Methode herstellen zu können.

Für unsere *h*-Version mit *p*=1 ergibt sich derselbe asymptotische Ausdruck für den mittleren Fehler wie bei *p*=0, jedoch mit $\beta = 2$. Bei gleicher Anzahl *n* von Abschnitten haben wir dann allerdings schon *2N* unbekannte Koeffizienten zu ermitteln. Trägt man den mittleren Fehler in Abhängigkeit von der Anzahl *N* der Unbekannten auf, so ergibt sich in einem bilogarithmischen Maßstab ein linearer Zusammenhang $\ln\|e\| = \ln k - \beta \ln N$. β gibt dabei die Neigung der Gerade an. Bild 5.4 zeigt die Konvergenz des mittleren relativen Fehlers für unsere vier Testfunktionen bei Einsatz der *h*-Methode:

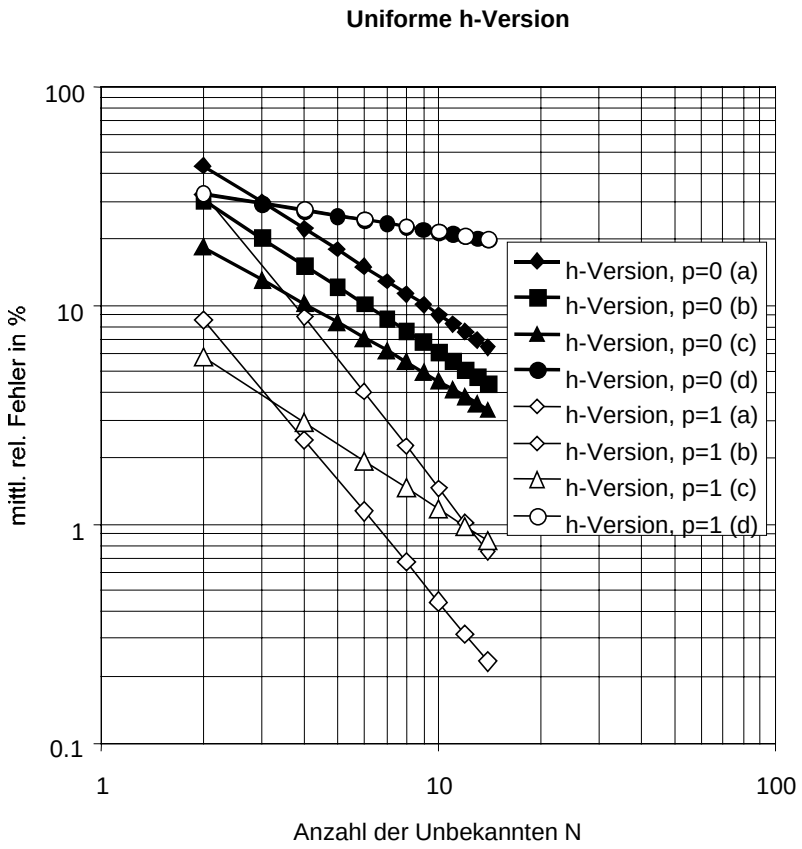

Bild 5.4: Konvergenz der *h*-Methode für die vier ausgewählten Funktionen.

Bild 5.4 zeigt deutlich, daß sich bei den genügend glatten Funktionen (a) und (b) tatsächlich die vorhergesagten Konvergenzraten $\beta = 1$ bzw. $\beta = 2$ einstellen. Der Fehler konvergiert also doppelt so schnell gegen 0, wenn wir statt stückweise konstanter Approximationsfunktionen stückweise lineare Ansätze verwenden.

Für die beiden anderen Testfunktionen gelten unsere Abschätzungen nicht, da für die Funktionen (c) und (d) die Voraussetzung der Beschränktheit der 1. Ableitung im untersuchten Intervall verletzt ist!

Dennoch sind auch diese Funktionen von Interesse. Experimentell können wir folgendes Konvergenzverhalten ermitteln (siehe Bild 5.4):

Bei Funktion (c) ist die Konvergenzrate nach oben beschränkt durch $\beta = \min\left(p+1, \lambda+\frac{1}{2}\right)$, worin $\lambda = \frac{1}{2}$ den Exponenten des am wenigsten glatten Anteils in *f(x)* bezeichnet. Bei Funktion (d) ist die Konvergenzrate $\beta = \frac{1}{4}$, also wiederum gleich dem Exponenten des führenden Anteils der Funktion an der Singularitätsstelle *x=0*.

5.1.4 *p*-Version - LEGENDRE-Polynome

Die Erkenntnisse des vorausgegangenen Abschnitts legen es nahe, die Anzahl der zu bestimmenden Koeffizienten nicht "unnötig" durch Unterteilung unseres Intervalles in die Höhe zu treiben, sondern lieber den effizienteren Weg der Wahl immer höherer Polynomgrade zu gehen, um unsere Approximationsqualität bis auf ein gewünschtes Maß zu verbessern. Wir verwenden im folgenden nur *ein einziges* Intervall.

Innerhalb dieses Intervalles verwenden wir unsere *N* Koeffizienten zur Beschreibung eines Approximationspolynoms der Ordnung *p=N*-1. Wir streben an, wiederum *N* voneinander unabhängige Gleichungen zu erhalten, so daß **A** wieder eine Diagonalmatrix ist. Wir müssen dann beim Übergang vom Polynomgrad *p* zu *p*+1 nicht alle *N* Koeffizienten neu berechnen, sondern jeweils nur den neuesten Koeffizienten, der zum Polynom mit dem Grad *p*+1 gehört!

Die gewünschte Eigenschaft der Orthogonalität besitzen auf dem Intervall $[-1;1]$ die LEGENDRE-Polynome L_i:

$$L_0 = 1$$
$$L_1 = x$$
$$L_2 = \tfrac{1}{2}\left(3x^2 - 1\right)$$
$$L_{i+1} = \frac{(2i+1)xL_i - i\,L_{i-1}}{i+1}$$

Es gilt, wie man leicht nachprüfen kann:

$$\int_{-1}^{1} L_i L_j dx = \frac{2}{2i+1}\delta_{ij}$$

Die ersten LEGENDRE-Polynome sind in Bild 5.5 dargestellt.

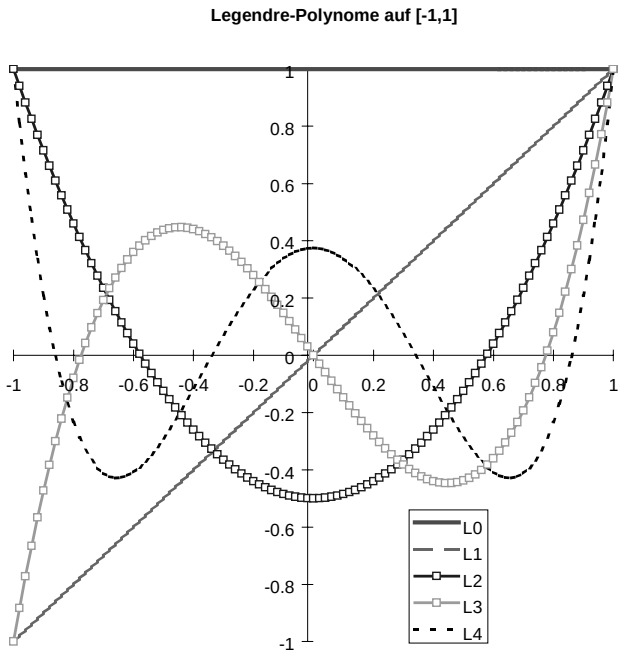

Bild 5.5: Verlauf der ersten fünf LEGENDRE-Polynome im Intervall [-1; 1].

Zum Aufbau unserer Approximation transformieren wir unser Intervall auf $[-1;1]$ und verwenden in diesem Einheitsintervall die LEGENDRE-Polynome.

Nunmehr können wir die Konvergenzcharakteristika für große *N* wie folgt abschätzen:

Bei der h-Version haben wir festgestellt, daß die Konvergenzrate β direkt proportional zum Polynomgrad *p* der Approximation war. Zwischen dem Logarithmus des mittleren Fehlers und *N* bestand folgender Zusammenhang:

$$\ln\|e\| = k - \beta \ln N = G(N)$$

Nach Differentiation nach *N* erhalten wir

$$\frac{d\,G(N)}{dN} = -\frac{\beta}{N}$$

In der *p-Version* auf einer festen Unterteilung unseres Intervalls (z.B. in ein einziges Teilintervall) "springen" wir gewissermaßen jeweils von der aktuellen Konvergenzkurve auf die nächststeilere, wenn wir von *p* zu *p*+1 weiterschalten.

β ist dann also nicht mehr konstant, sondern direkt proportional zu *N*, da *N*=*p*+1.

Setzen wir $\beta = CN$ in die Beziehung ein, so erhalten wir eine einfache Differentialgleichung für die Konvergenz des Logarithmus des Fehlers in der *p*-Version, *H(N)*, die wir verwenden können, um die Konvergenz der *p*-Version abzuschätzen:

$$\frac{d\,H(N)}{dN} = -C$$
$$H(N) = -CN + d$$
$$\|e\| = e^{-CN+d} = \frac{k}{e^{-CN}}$$

Die *p*-Version ist also - unter Voraussetzung ausreichend glatter Funktion *f(x)* - durch exponentielle Konvergenz gekennzeichnet. Dieses Verhalten spiegelt sich im doppeltlogarithmischen Maßstab als Krümmung der Konvergenzkurve nach unten wieder. Die exponentielle Konvergenz der p-Version ist für die Funktionen (a) bis (c) deutlich im Diagramm (Bild 5.6) zu erkennen.

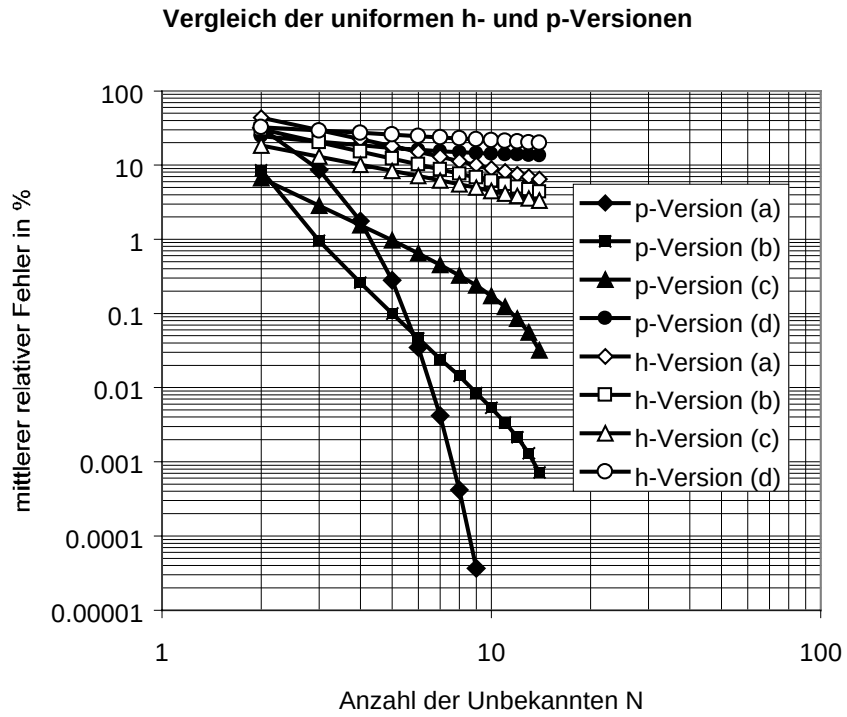

Bild 5.6: Konvergenz der *h*-Version und *p*-Version im Vergleich.

Bei der singulären Funktion d) bringt uns allerdings auch die *p*-Version keine Verbesserung gegenüber der *h*-Version. Dort konvergiert der mittlere Fehler nach wie vor nur algebraisch mit der konstanten, niedrigen Konvergenzrate $\beta = 0.25$. Dennoch sehen wir deutlich, daß die Konvergenzrate der *p*-Version in jedem Fall mindestens so gut ist wie die der *h*-Version. Es gibt bei unseren untersuchten Funktionen keinen Fall, in dem eine reine *h*-Version vorzuziehen wäre.

5.1.5 *hp*-Version

Um die singuläre Funktion (d) ebenso effizient approximieren zu können wie die Funktionen (a) bis (c), müssen wir eine andere Strategie wählen: Wir starten mit einem einzigen Intervall und p=0. Sodann fügen wir an der Singularitätsstelle x=0 eine "Netzverfeinerung" ein, d.h. wir teilen unser Intervall durch einen neuen Knoten bei $x = \sigma L$ auf, $0 < \sigma < 1$. In dem weiter von der Singularität weg gelegenen Teilstück erhöhen wir gleichzeitig den Polynomgrad auf p+1, während wir in dem Element, das direkt an die Singularität anschließt, p=0 beibehalten. Sukzessive führen wir weitere Elemente mit dem Verfeinerungsfaktor σ an der Singularitätsstelle ein und erhöhen gleichzeitig auf allen alten Elementen p um 1. Wir erhalten somit eine Netzverfeinerung in geometrischer Progression und eine Polynomgradverteilung, die "linear" von der Singularität nach außen hin anwächst.

Mit diesem Verfahren gewinnen wir auch für Funktion d) die exponentielle Konvergenz zurück, wie das Bild 5.7 zeigt.

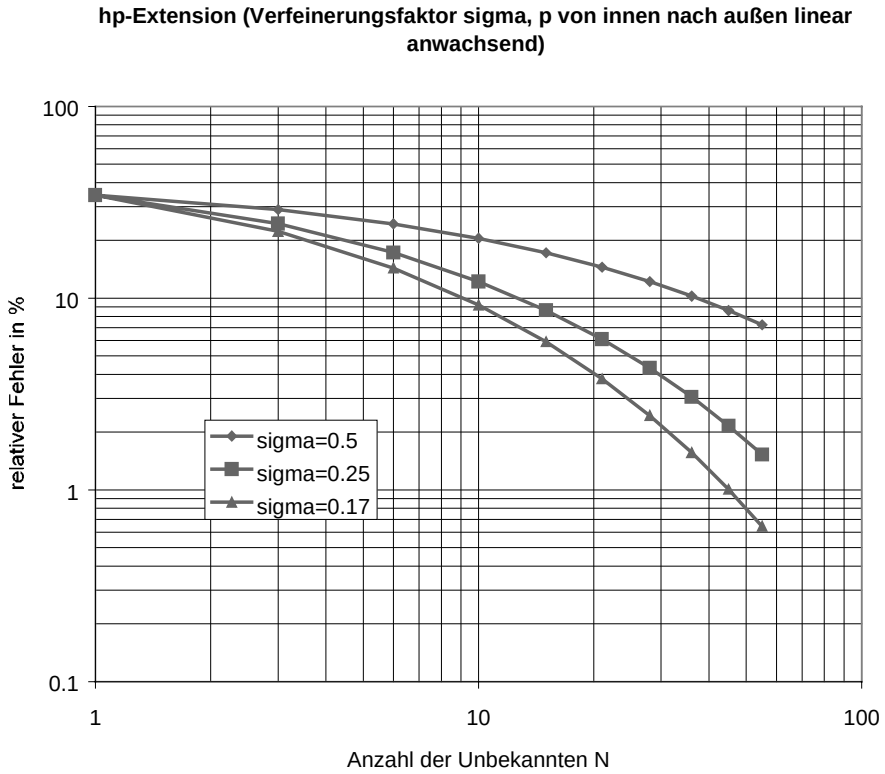

Bild 5.7: Konvergenz einer *hp*-Version bei der Approximation der singulären Funktion (d).

Man kann nun mit verschiedenen Verfeinerungsverhältnissen σ experimentieren, um den optimalen Verfeinerungsfaktor zu ermitten. Interessanterweise ist der optimale Verfeinerungsfaktor vom Grad der Singularität unabhängig und liegt bei $\sigma = 0.17$. Faktoren, die etwas kleiner als der optimale Faktor sind, sind ebenfalls gut geeignet. Auch schon wenig größere Faktoren sind hingegen deutlich ungünstiger.

5.1.6 Globale Fehlerschätzung durch Extrapolation

Bei allen bisher untersuchten Problemen konnten wir feststellen, daß die Konvergenz von $\|e\|$ mindestens algebraisch erfolgte. Im günstigsten Fall konnten wir jedoch auch exponentielle Konvergenz erzielen. Bei exponentieller Konvergenz wächst die Konvergenzrate β stetig an, während sie bei algebraischer Konvergenz konstant ist.

Um den Fehler einer Approximation abzuschätzen, können wir - auf der sicheren Seite liegend - unterstellen, daß nur algebraische Konvergenz vorliegt. Wir können dann aus drei verschiedenen Approximationen die Konvergenzrate β näherungsweise ermitteln, auch ohne daß uns $\Pi = \int_0^L f^2 dx$ explizit bekannt ist.

Wir unterstellen: $\|e\|^2 = \int_0^L f^2 dx - \mathbf{a}^{\mathbf{T}}\mathbf{F} = \left(\frac{k}{N^\beta}\right)^2 = \frac{k^2}{N^{2\beta}}$

Liegen uns drei verschiedene numerische Experimente mit $N_{i-2} < N_{i-1} < N_i$ Unbekannten Koeffizienten vor, so können wir drei Näherungswerte $\Pi_i = \mathbf{a}_i^T \mathbf{F}_\mathbf{i}$ für Π bestimmen. Mit Hilfe des unterstellten Zusammenhanges zwischen N und $\|e\|$ erhalten wir dann:

$$\frac{\ln\dfrac{\Pi-\Pi_i}{\Pi-\Pi_{i-1}}}{\ln\dfrac{\Pi-\Pi_{i-1}}{\Pi-\Pi_{i-2}}} = \frac{2\beta\ln\dfrac{N_{i-1}}{N_i}}{2\beta\ln\dfrac{N_{i-2}}{N_{i-1}}}$$

Aus dieser nichtlinearen Gleichung für Π läßt sich ein Schätzwert für Π ermitteln, und damit können wir sodann auch den mittleren Fehler abschätzen (RICHARDSON-Extrapolation).

Bei unserer ursprünglichen Approximationsaufgabe macht eine derartige Abschätzung wenig Sinn, da wir in aller Regel sowohl $\Pi = \int_0^L f^2 dx$ direkt bestimmen können, als auch β direkt bekannt ist. Bei der späteren Anwendung der *h*-, *p*- und *hp*-Versionen auf die Methode der finiten Elemente wird sich der Nutzen einer solchen Abschätzung aber deutlich zeigen. Diese Abschätzung durch Extrapolation kommt ohne irgendwelche lokalen Auswertungen von Ergebnissen aus. Bild 5.8 zeigt, daß auch bei Einsatz der *p*-Version bei glatten Funktionen oder der *hp*-Versionen bei singulären Funktionen, also in den Fällen, wo die Annahme $\beta = const$ nicht zutrifft, dennoch eine gute Übereinstimmung zwischen der Abschätzung und dem wahren Fehler erzielt wird.

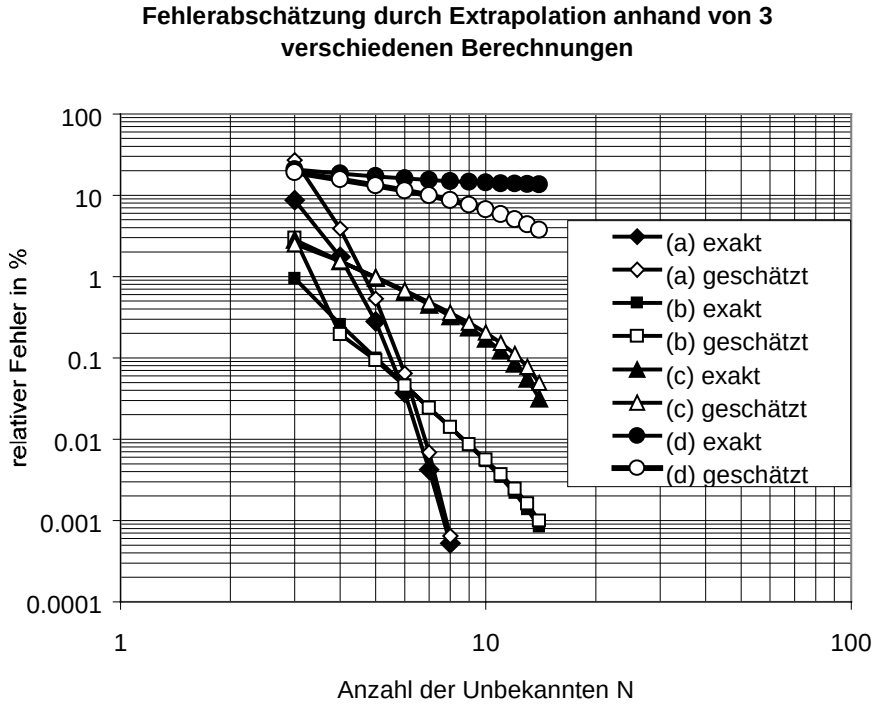

Bild 5.8: Vergleich zwischen Fehlerschätzung und wahrem Fehler.

5.1.7 *p*-Version auf einem graduierten Gitter

Als "pragmatische Variante" gegenüber der daten- und programmtechnisch vielleicht etwas verwickelten *hp*-Version untersuchen wir abschließend experimentell den Einsatz einer gleichmäßigen Polynomgraderhöhung auf einem nicht gleichmäßigen Gitter, sondern einem gemäß dem Vorgehen bei

der *hp*-Version erzeugten, dann aber während der Polynomgraderhöhung festen Gitter. - Das folgende Bild 5.9 zeigt die experimentellen Ergebnisse mit einem solchen Verfahren, eingesetzt bei unserer Testfunktion (d).

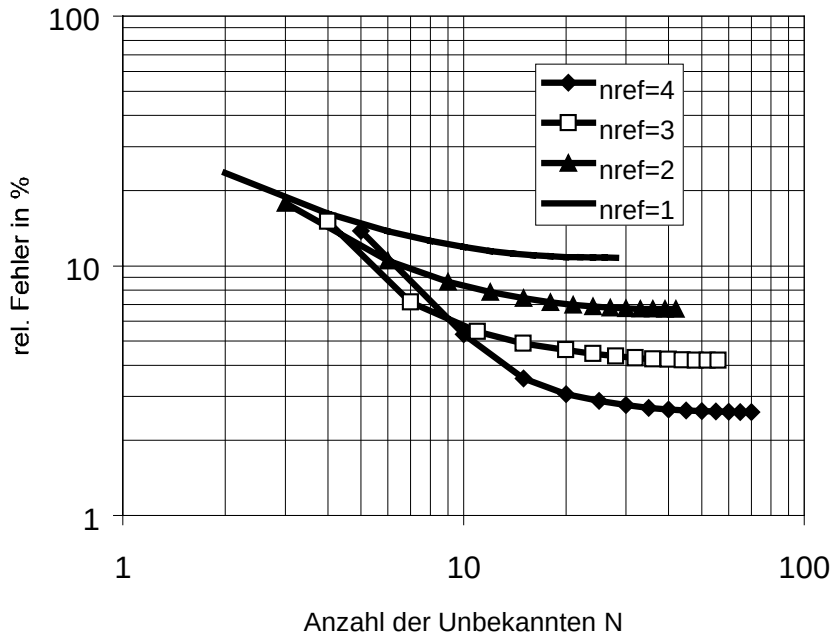

Bild 5.9: Konvergenz einer *p*-Version auf angepaßtem (graduiertem) Gitter

Selbstverständlich stellen sich bei dieser Variante asymptotisch (d.h. für $N \to \infty$) dieselben Konvergenzraten ein wie bei der ursprünglichen *p*-Version, d.h. algebraische Konvergenz mit $\beta = \frac{1}{2}$ in unserem Beispiel. Im Anfangsbereich, bei wenigen Unbekannten, verhält sich dieses modifizierte Verfahren allerdings ähnlich wie die echte *hp*-Version, erst bei zunehmendem Polynomgrad "fehlen" dann die sukzessiven Netzverfeinerungen an der Singularitätsstelle. Je nach der Anzahl vorhandener Verfeinerungsschichten biegen wir früher oder später in die flache asymptotische Konvergenz ein.

Welchen Vorteil könnte ein derartiges Verfahren haben? Datentechnisch ist die sukzessive Netzverfeinerung der *hp*-Version mühsam zu realisieren. Einfacher ist es, mit einer gleichbleibenden Geometrie zu arbeiten. Noch einfacher wird das Verfahren, wenn darüberhinaus alle Abschnitte denselben Polynomgrad besitzen. Wenn durch die "unnötigen", d.h. im Hinblick auf die Verminderung des Fehlers wenig effektiven Ansätze hoher Ordnung auf den kleineren Abschnitten kein überproportional großer Berechnungsaufwand entsteht, kann sich ein solches Verfahren im Rechenzeitaufwand mit einer echten *hp*-Methode messen. Zugleich erhalten wir uns einen der Hauptvorteile des *hp*-Verfahrens, nämlich die steile Konvergenz schon bei kleinem *N*, und können damit den Fehler schnell unter eine akzeptable Grenze drücken. Die geringe Konvergenzrate für $N \to \infty$ stört uns dann nicht mehr.

5.1.8 Finite Elemente in der Strukturmechanik: Analogie zum einfachen Approximationsproblem

Wir wollen nun unser Augenmerk der Finite-Elemente-Methode zuwenden. Als Modellproblem wählen wir den längsbelasteten elastischen Stab $[0; L]$, dessen Verschiebungen in Stabachse wir mit $u(x)$ bezeichnen (Bild 5.10).

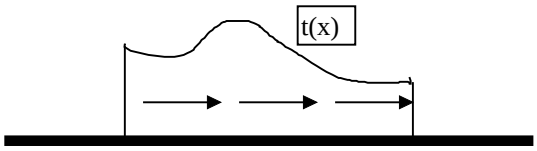

Bild 5.10: Der längsbelastete elastische Stab.

Der Stab habe konstanten Querschnitt und Elastizitätsmodul *EA* sowie homogene Dirichlet-Randbedingungen $u(0) = u(L) = 0$ (Stab an beiden Enden festgehalten). Längs des Stabes greifen beliebig verteilte Lasten $t(x)$ an.

$u(x)$ soll mit Hilfe von Ansätzen $\hat{u} = \mathbf{p}(x)^{\mathrm{T}}\,\mathbf{a}$ approximiert werden. Wir wählen für ***p(x)*** stetige Funktionen. **a** bezeichnet wiederum die Koeffizienten der Approximation. Wir erhalten folgendes Finite-Elemente-Gleichungssystem:

$$EA\int_o^L \left(\frac{d}{dx}\mathbf{p}\right)\left(\frac{d}{dx}\mathbf{p}\right)^T dx\,\mathbf{a} = \int_0^L \mathbf{p}\,t(x)\,dx$$

Betrachten wir zunächst die linke Seite. Ein Vergleich mit den Formeln aus (1.1) macht sofort deutlich, daß unser Finite-Elemente-Verfahren völlig analog zu dem Versuch ist, mit Hilfe der 1. Ableitungen der Ansatzfunktionen **p***(x)* eine Funktion zu approximieren. Um zu sehen, welche Funktion wir auf dem Intervall approximieren wollen, formen wir die rechte Seite der Finite-Elemente-Gleichung um. Wir interpretieren die Lastfunktion *t(x)* als 1. Ableitung einer Funktion *T(x)* und integrieren die rechte Seite partiell. Die Randterme fallen wegen der homogenen Randbedingungen weg, und wir erhalten:

$$EA\int_o^L \left(\frac{d}{dx}\mathbf{p}\right)\left(\frac{d}{dx}\mathbf{p}\right)^T dx\,\mathbf{a} = -\int_0^L \frac{d}{dx}\mathbf{p}\,T(x)dx$$

Die Lösung des Problems des längsbelasteten Balkens mit einer Finite-Elemente-Approximation $\hat{u} = \mathbf{p}(x)^{\mathrm{T}}\,\mathbf{a}$ ist also völlig gleichbedeutend mit der Aufgabe, die Stammfunktion *T(x)* der Belastungsfunktion *t(x)* mit den ersten Ableitungen der Finite-Elemente-Ansatzfunktionen im Sinne kleinster Quadrate zu approximieren.

5.1.9 Aufbau und Konvergenzrate der Finite-Elemente-Approximation

h-Version: Fordern wir lediglich eine einfache C^0-Stetigkeit der Approximation $\hat{u}$, so sind die Ableitungen der Ansatzfunktionen nur noch stückweise stetig. Wir haben dann genau denselben Fall wie in Abschnitt 2. Abhängig von der Glattheit der Funktion *t(x)* erhalten wir bei Einsatz von Finite-Elemente-Ansatzfunktionen der Ordnung *p* analog zu unseren Erkenntnissen in Abschnitt 2 eine Konvergenz wie

$\|e\| = \frac{k}{N^{\beta}}$ mit $\beta = \min\left(p, \lambda - \frac{1}{2}\right)$, worin λ im Falle nicht glatter Funktionen *t(x)* (nicht ganzzahliges λ) den führenden Exponenten von *t(x)* darstellt.

p-Version: Verwenden wir für $\hat{u}$ Ansätze der Ordnung *p*, so entspricht die Finite-Elemente-Lösung einer Least-Squares-Approximation von *T(x)* mit Polynomen der Ordnung *p-1*. Wir erhalten also in Analogie zu Abschnitt 3 in jedem Falle exponentielle Konvergenz, solange $\frac{d}{dx}T(x) = t(x)$ beschränkt bleibt. Auch bei Streckenlasten, deren Ordinate im Inneren des Balkens sprunghaft veränderlich ist, können wir also exponentielle Konvergenz erwarten. Interessant ist natürlich der Fall der Einzellast. Eine Einzellast entspricht einem Sprung in *T(x)*. Damit ist die Ableitung von *T* an einem Punkt nicht mehr endlich. Solange dieser Punkt jedoch mit einer Elementgrenze übereinstimmt (sogenannte "Knotenlast"), ist die Konvergenz nach wie vor exponentiell (ansonsten, wie ein einfaches numerisches

Experiment zeigt, algebraisch). Man muß die Finite-Elemente-Netzeinteilung also so wählen, daß an jeder Lasteinleitungsstelle einer Einzellast ein Knoten liegt.

Wenn die Ableitungen unserer Ansatzfunktionen Legendre-Polynome sind, unsere Ansätze also Stammfunktionen der Legendre-Polynome, können wir weiterhin die Vorteile der Orthogonalität ausnutzen. Um die C^0-Stetigkeit zu erreichen, benötigen wir mindestens p=1, und es ist zweckmäßig, die lineare Approximation aus zwei "Rampenfunktionen" zusammenzusetzen. Wir verwenden also die Ansätze

$p_1 = \frac{1}{2}(1-x)$

$p_2 = \frac{1}{2}(1+x)$

$p_3 = -\frac{1}{2}(1+x)(1-x)$

$p_4 = \frac{1}{2}\left(x^3 - x\right)$

usw.

hp-Version: Beim gewählten Modellproblem lassen sich keine für die Finite-Elemente-Methode zulässigen Fälle konstruieren, in denen eine *hp*-Methode notwendig wird, um exponentielle Konvergenz zu erreichen.

Bei allgemeineren Problemen treten gewöhnlich in unseren Finite-Elemente-Steifigkeitsmatrizen nicht nur die ersten Ableitungen der Ansatzfunktionen auf, sondern auch die Ansatzfunktionen selbst. In diesen Fällen ist die Analogie zwischen Least-Squares-Approximation und Finite-Elemente-Methode nicht mehr so offensichtlich. Dennoch lassen sich unsere Betrachtungen zur Konvergenz weitgehend auch auf wesentlich allgemeinere Fälle übertragen.

5.1.10 Finite-Elemente-Methoden für zweidimensionale Probleme

Auch bei zwei- oder mehrdimensionalen Problemen (Funktionen von zwei oder mehr Veränderlichen) können die genannten Verfahren zur Approximation nach der Methode der kleinsten Quadrate bzw. die Finite-Elemente-Methode angewendet werden. Bei der *h*-Version ist die Anzahl der Koeffizienten *N* dann proportional zur Gesamtanzahl der Elemente, bei der *p*-Version wächst sie mit dem Quadrat des Polynomgrades. Unter Berücksichtigung dieser Umstände ergeben sich die Konvergenzraten wiederum völlig analog zum 1D-Problem.

Das Analogon zu unserer *h-Version* beim eindimensionalen Problem ist in der sukzessiven, gleichmäßigen Verfeinerung eines weitgehend gleichmäßigen (quasiuniformen) Ausgangsnetzes zu sehen. Baut man die *h*-Version hierarchisch auf, so kann man die Fehlerschätzung durch Extrapolation anwenden. Außerdem erhält man dann eine Folge von hierarchisch verfeinerten Gittern. Das lädt dazu ein, die Konvergenz iterativer Gleichungslöser (Relaxationsverfahren) durch Einsatz von *Mehrgittermethoden* zu beschleunigen. So kann man auch in der *h*-Version extrem effiziente Programme erhalten. Die Idee des Mehrgitterverfahrens besteht darin, langwellige Lösungsanteile auf einem groben – somit schnell zu berechnenden – Gitter zu erfassen und die feineren Gitter nur zur Berechnung ("Glättung") der kurzwelligen Lösungsanteile heranzuziehen. Allerdings sind für das Mehrgitterverfahren – wie für Finite-Differenzen-Methoden – Gitter erforderlich, in denen z.B. die Anzahl der an jedem Knoten zusammentreffenden Elemente gleich ist (*reguläre Gitter*). Für allgemeine Problemgeometrie bedeutet dies eine wesentliche Einschränkung.

Ansätze für die *p-Version* in 2D gewinnt man durch Einsatz von Tensorprodukten der eindimensionalen Ansätze auf Viereckselementen. Aufgrund der im allgemeinen nicht rechtwinkligen Form der finiten Elemente in 2D und aufgrund der Vermischung von 0. und 1. Ableitungen der Ansätze in der Steifigkeitsmatrix kann man in aller Regel keine Elementsteifigkeitsmatrizen mehr

erzielen, die nur auf der Diagonale besetzt sind. Dennoch ist es auch weiterhin günstig, Stammfunktionen der LEGENDRE-Polynome als Ansätze zu benutzen, da das resultierende Gleichungssystem dann gut konditioniert ist und sich die Koeffizienten der niedrigen Ansätze bei Hinzunahme höherer Polynomgrade nur noch wenig ändern, was man z.B. in einem iterativen Gleichungslöser zur Bildung günstiger Iterations-Startvektoren ausnützen kann. Somit ist auch die *p*-Version sehr effizient implementierbar.

Abbildung 5.11 zeigt Ansatzfunktionen der *p*-Version in 2D.

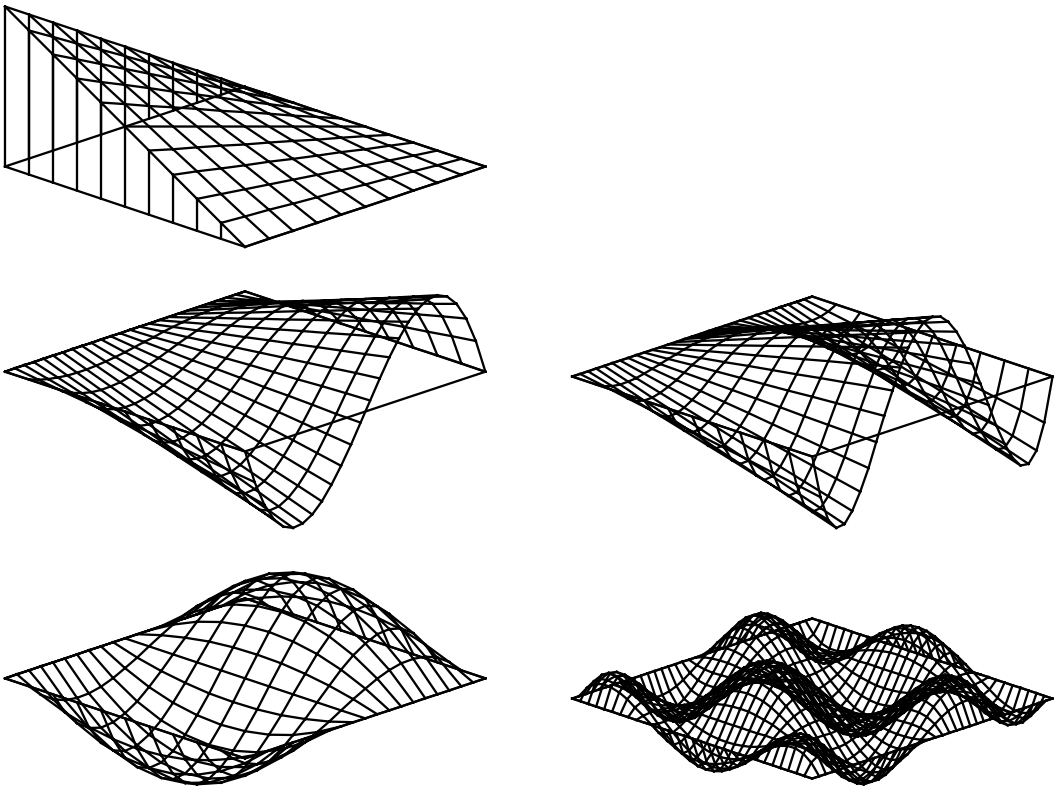

Bild 5.11: Hierarchische Ansatzfunktionen für die *p*-Version der FEM in 2D.

Die in Bild 5.11 gezeigten Tensorprodukt-Ansätze lassen sich in drei Gruppen einteilen:

1. "vertex modes": Das sind die bilinearen Ansatzfunktionen. Diese Ansatzfunktionen haben jeweils an einem Knoten den Wert 1. Bei einem regulären Viereckesnetz sind im Mittel jeweils ca. 4 Elemente durch derartige Ansätze gekoppelt.
2. "edge modes": Diese Funktionen entstehen aus dem Produkt eines höheren Polynoms in der einen mit einer linearen Funktion in der anderen Richtung. Diese Ansätze sind jeweils nur auf zwei benachbarten Elementen bzw. auf einer einzigen Kante von Null verschieden.
3. "interior modes": Diese Funktionen entstehen aus Produkten von Polynomen mit $p_i, p_j > 1$.

Es bieten sich zwei verschiedene Möglichkeiten zum Aufbau des Finite-Elemente-Ansatzraums:

1. Alle Produkte der Polynome eines bestimmten Grades werden angesetzt; bei Polynomgrad *p* entstehen auf jedem Element p^2 Ansätze.
2. Alternativ können auch lediglich die vollständigen Polynome der Ordnung *p* angesetzt werden.

In der Praxis hat sich der volle Produktansatz (1) als günstiger erwiesen.
Die skizzierte Vorgehensweise zum Aufbau von p-Approximationen bei finiten Elementen führt zu einer hierarchischen Ansatzgraderhöhung, d.h. beim Übergang von *p* nach *p+1* bleiben die Ansätze des niedrigeren Polynomgrades unverändert.

Die *hierarchische p-Version* erfordert eine aufwendigere Datenstruktur als die konventionelle, nicht hierarchische *h-Version*. Die Koeffizienten der Lösung können nicht direkt als Wert der Funktion an ausgewählten Stellen (den "Knoten") interpretiert werden. Zusätzlich zu bloßen Knoteninformationen sind auch kanten- und flächenbezogene Informationen erforderlich.

Eine *hp-Version* der finiten Elemente muß eingesetzt werden, um exponentielle Konvergenz auch bei Problemen zu erreichen, bei denen die exakte Lösung singuläre Stellen (Singularitäten in der 1. Ableitung) enthält. Prinzipiell können bei 2D-Problemen sowohl punkt- als auch linienförmige Singulärstellen auftreten. Für die Ecksingularitäten ist eine punktförmige Netzverfeinerung notwendig. Die Eckverfeinerung muß wiederum mit $\sigma = 0.17$ erfolgen. Durch rekursives Verfeinern dürfen die Elemente nicht immer spitzere Winkel erhalten, und konforme Netze sind erwünscht. Daher hat sich das nachfolgend skizzierte Verfeinerungsmodell eingebürgert (Bild 5.12).

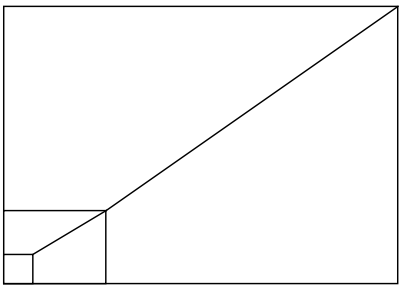

Bild 5.12: Verfeinerungsschema für die *hp*-Version der FEM in 2D bei Vorhandensein von Punktsingularitäten in der Lösung.

Beim Übergang von der singulären Stelle nach außen werden "Übergangselemente" zur schrittweisen Erhöhung des lokalen Polynomgrades benötigt. Diese können jedoch ganz einfach erzeugt werden, indem man solchen Elementen z.B. alle Ansatzfunktionen des Grades $p+1$ zuweist und lediglich diejenigen "edge modes" eliminiert, die auf der innen liegenden Kante von Null verschieden sind und längs dieser Kante höheren Polynomgrad als p aufweisen. Alternativ kann man auch das ganze Element mit den Ansätzen der Ordnung p belegen und lediglich an der äußeren Kante den "edge mode" mit $p+1$ hinzunehmen.

5.2 Automatische Netzgenerierung

5.3 Anwendungsbeispiele zur FEM bei Platten- und Scheibentragwerken

5.4 Integration von CAD und automatisierter FEM

5.4.1 Modellbildung

5.4.2 Datenaustausch

5.4.3 Bemessung und Konstruktion

6. Computergestützte Terminplanung