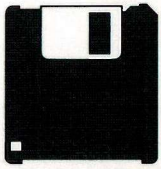
Markt&Technik

Axel Plenge

AMIGA 3D-GRAFIK UND ANIMATION

Grundlagen, Clipping, perspektivische Projektion, verdeckte Linien und Flächen, 3D-Funktionsplotter, Ray-Tracing, Spiegelungen, Reflexionen, Lichtquellen, Lichtbrechung, Schatten, Rotationskörper.



Auf 3½"-Diskette enthalten:
Alle Programmbeispiele in C und Amiga-BASIC

Amiga 3D-Grafik und Animation

Dreidimensionale Grafik bestaunen oft nicht nur Computerfreaks respektvoll und bewundernd in Schaufenstern oder auch auf dem eigenen Computer. Daß es gar nicht so schwierig ist, solche Grafiken auch selbst zu programmieren, wissen die wenigsten.

In diesem Buch erfahren Sie nun, wie Sie – angefangen bei den einfachsten Problemstellungen – solche professionellen 3-D-Grafiken auf Ihrem Commodore-Amiga planen, programmieren und darstellen. Es wendet sich an alle grafikbegeisterten Amiga-User, an die fortgeschrittenen Programmierer wie auch an Grafik-Neulinge – ein wenig mathematisches Basiswissen und Basic-(oder C-) Programmierkenntnisse vorausgesetzt.

Sämtliche theoretischen Grundlagen werden Ihnen sehr anschaulich und vor allem verständlich vermittelt. Zahlreiche Beispielprogramme in Basic und in C demonstrieren Ihnen die praktische Anwendung Ihres Wissens – in Farbe natürlich.

Alle Programme werden gleich auf einer **Diskette zum Buch** mitgeliefert!
Lästiges Eintippen und Fehlerkorrekturen bleiben Ihnen erspart.

Ausführlichkeit und vor allem Verständlichkeit haben in diesem Buch oberste Priorität! Nur so können die teilweise komplexen Sachverhalte durchschaubar gemacht werden.

Doch unter der Verständlichkeit darf natürlich das Niveau nicht leiden. Und so macht das Werk auch vor scheinbar so komplizierten Dingen wie Schattenbildung, Reflexionen, durchsichtigen Gegenständen, Vielfachspiegelungen oder »Ray-Tracing« nicht halt. Überzeugen Sie sich selbst! Ein **Farbteil** enthält Bildschirmfotos aller Demo-Programme. Hier einige ausgewählte Themen des Buches:

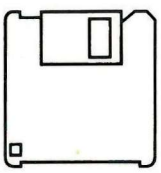
- 2-D-Grundlagen, Drehungen, Skalierung, Clipping
- 3-D-Koordinatensysteme
- Perspektivische Projektion
- Transformationen im Raum (Skalierung, Drehung ...)
- Verdeckte Linien und Flächen
- Ray-Tracing, Schattenbildung, Reflexionen und transparente Objekte
- Rotationskörper

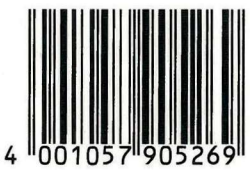
PETER WOLLSCHLAEGER, *bekannt als Fachbuchautor von zahlreichen Zeitschriftenartikeln und Computer-Büchern ist voll des Lobes für dieses seiner Meinung nach »gelungene« Grafikbuch:*
»Der Autor versteht es, schwierige mathematische Zusammenhänge allgemeinverständlich und angenehm lesbar zu erklären. Die Programme selbst sind immer von bester Qualität. Sie sind sauber programmiert, ausführlich kommentiert und bringen interessante Lösungen mit fantastischen grafischen Effekten.«

Hardware-Anforderungen:
Amiga 500, 1000 oder 2000
Software-Voraussetzung:
Amiga-Basic bzw. C-Compiler (Aztec C oder Lattice C)

ISB N 3-89090-526-9

Markt&Technik





DM 69,-
sFr 63,50
öS 538,20

Axel Plenge

AMIGA

3-D-Grafik und Animation

Grundlagen, Clipping, perspektivische Projektion, verdeckte Linien und Flächen, 3-D-Funktionsplotter, Ray-Tracing, Spiegelungen, Reflexionen, Lichtquellen, Lichtbrechung, Schatten, Rotationskörper.

Markt&Technik Verlag AG

CIP-Titelaufnahme der Deutschen Bibliothek

Plenge, Axel:
AMIGA-3-D-Grafik [AMIGA-drei-D-Grafik] und Animation :
Grundlagen, clipping, perspektiv. Projektion, verdeckte Linien u. Flächen,
3-D-Funktionsplotter, ray-tracing, Spiegelungen, Reflexionen,
Lichtquellen, Lichtbrechung, Schatten, Rotationskörper / Axel Plenge. –
Haar bei München : Markt-u.-Technik-Verl., 1988.
ISBN 3-89090-526-9

15 14 13 12 11 10 9 8 7 6 5 4 3
91 90 89

ISBN 3-89090-526-9

Einbandgestaltung: Grafikdesign Heinz Rauner
Druck: Schoder, Gersthofen
Printed in Germany

Inhaltsverzeichnis

Vorwort des Fachlektors

Eigentlich schreibe ich nur Vorworte in meinen eigenen Büchern, denn was habe ich davon, wenn ich die Produkte der Konkurrenz lobe? Nun, man sollte auch sachlich bleiben, und dieses Buch hat Lob verdient.

Im Gegensatz zu so manchem Grafik-Buch werden hier nämlich nicht nur ein paar Programme vorgeführt, mit denen der Leser spielen kann, sondern es wird Wissen vermittelt.

Wohl aus Angst, mögliche Käufer zu verschrecken, verschweigen andere Bücher, daß Grafik nichts weiter als angewandte Mathematik ist. Diese leider nicht zu ändernde Tatsache bestreitet der Autor nicht, sondern geht das Problem an.

Es ist ihm gelungen, schwierige mathematische Zusammenhänge allgemeinverständlich und angenehm lesbar zu erklären. Mathematik-Puristen mögen stellenweise etwas einzuwenden haben, die Aussagen sind aber immer sachlich korrekt und führen präzise zum eigentlichen Ziel, nämlich der programmtechnischen Lösung.

Die Programme selbst sind immer vom Feinsten. Sie sind sauber programmiert, ausführlich kommentiert und bringen interessante Lösungen mit fantastischen grafischen Effekten.

Die Programme bereichern das Buch natürlich; ich sehe sie aber nur als nützliche Zugabe. Primär ist positiv zu bewerten, daß dem Leser alle Kenntnisse von den Grundlagen bis hin zu den Tricks der Profis vermittelt werden, womit er selbst in die Lage versetzt wird, komplexe grafische Programme zu schreiben.

Peter Wollschlaeger

Kapitel 1

Einleitung

Haben Sie noch irgendwelche Zweifel, ob Sie mit Ihrem Amiga die richtige Wahl getroffen haben? Nun, dann schauen Sie sich doch einmal die Bilder in diesem Buch an...

Jetzt wo Sie der absolut festen Überzeugung sind, den besten Rechner der Welt zu Hause stehen zu haben, sollten Sie auch nicht vor einem adäquaten Buch zu dieser Hypermaschine haltmachen. Nein, schauen Sie nicht wieder ins Bücherregal, es liegt bereits in Ihren Händen!

So oder ähnlich könnte sich die Einleitung zu einem Buch-Erzeugnis zweiter Wahl anhören. Wir aber wollen uns hier natürlich mit ernsten Dingen befassen und bleiben deshalb gediegen und seriös!

Zugegeben, Spaß macht es allemal, das Letzte aus einem solchen Gerät herauszuholen, was gerade noch vertretbar ist; das Letzte an Geschwindigkeit, an Auflösung, an Farbe. Und dieses Buch wird Ihnen dabei helfen. Es macht Sie zum Profi, zum Spezialisten, was Grafik betrifft.

Oberste Prämisse: Verständlichkeit! Gerade das Gebiet der Grafik, das in der Mathematik eine entscheidende Rolle spielt, ist oft als schwierig und kompliziert verschrieen. Dieses Buch soll damit ein für allemal Schluß machen. Was Sie lesen, das sollen Sie auch von Anfang an verstehen! Aus diesem Grunde wurde erstens viel Wert auf eine systematische Gliederung der Themen und zweitens auf ausführliche Erklärungen gelegt, auch wenn dadurch zwangsläufig ab und zu einmal etwas doppelt erläutert wird (besser als einmal zuwenig).

Gleich danach ist das zweite Prinzip dieses Buches anzusiedeln: Hoher Informationsgehalt! Sie werden in diesem Buch Dinge finden, wie sie bisher stets nur »Eingeweihten« und »Insidern« bekannt waren.

Viele sehr gut dokumentierte und lehrreiche Beispielprogramme, die Ihnen gleichzeitig einige Freude bereiten werden, veranschaulichen die praktizierte Theorie und Sie werden merken, daß alles gar nicht so schwierig ist, wie Sie vielleicht bisher meinten. Beim Thema Grafik wird schließlich auch nur mit Wasser gekocht. Die Programme sind entweder in Amiga-Basic oder/und in der bekannten Programmiersprache C geschrieben, die immer mehr Freunde gerade auf dem Amiga und auch auf anderen Rechnern gewinnt. Alle Programme strotzen nur so vor Kommentaren und werden im Text ausführlichst beschrieben. Für C-Freunde ist es sicher eine gute Nachricht, daß sämtliche C-Programme sowohl mit dem Aztek- als auch mit dem Lattice-Compiler unverändert(!) kompiliert werden können.

Und das steht drin:

Sind Sie nun neugierig geworden? Na, dann hören Sie sich einmal an, was wir noch zu bieten haben. Zunächst beschäftigen wir uns mit einigen grundlegenden Grafikeigenschaften des Amiga, deren Kenntnis die Voraussetzung für die restlichen Kapitel des Buches sind. Es wird also sowenig wie nur möglich an Wissen und Fertigkeiten vorausgesetzt. Hier finden Sie Informationen über Farben, Grafikauflösungen, Grafikmodi und dergleichen. Zur selben Zeit erfahren Sie, wie Sie diese Grafikfeatures aus Amiga-Basic und aus C heraus am effektivsten nutzen können.

Nach dieser kleinen Einführung kommen wir gleich zur Sache. Es geht darum, die Techniken, die uns später in den dreidimensionalen Kapiteln ständig begleiten werden, hier zunächst auf »sicherem Boden« zweidimensional darzustellen: Punkte, Linien, Kreise, Ellipsen und »das,

was dahintersteckt«. Zentraler Punkt sind die sogenannten Transformationen: Hier erfahren Sie, wie Sie Objekte in der Ebene vergrößern, verschieben, drehen oder spiegeln. Die mathematischen Grundlagen hierzu werden ausführlich und verständlich erklärt, und ich garantiere Ihnen, daß Sie am Ende des Kapitels einiges hinzugelernt haben!

Das für die Fenstertechnik wichtige sogenannte Clipping von Linien – für die einen ein unbekannter Begriff, für die anderen ein Schreckgespenst – wird danach ebenso zu Ihrem Repertoire gehören wie die einfache Verformung von Bildschirmbereichen. Letzteres ist auch bekannt unter der »Mona-Lisa-auf-Cola-Flasche-Projektion« oder: Wie wickle ich mein Grafikbild um eine Tonne, um eine Kugel?

Nach der Einführung in die Grundbegriffe, die vor allem für dreidimensionale Grafiken von entscheidender Bedeutung sind, geht es nun tatsächlich ans »Eingemachte«; 3-D-Grafik, wie sie leibt und lebt. Ausführlich erfahren Sie alles über 3-D-Koordinatensysteme und wie man eine dreidimensionale Welt am besten organisiert, speichert und handhabt.

Was versteht man unter einer parallelen, was unter einer Zentralprojektion? Wie programmiere ich dreidimensionale Grafiken mit Vergrößerungen, Drehungen usw. im Raum unter Basic, wie unter C? Wie simuliere ich einen Beobachter, lasse ihn durch die Szenerie einer 3-D-Welt wandern und vieles mehr. Programme, die Sie sich ansehen müssen!

Im fünften Kapitel schließlich geht es darum, eigentlich verdeckte Linien und Flächen unsichtbar zu machen, Techniken zum Problem der »Hidden Lines and Surfaces« werden jeweils für unterschiedliche Problemstellungen vorgestellt. Hier finden Sie einen 3-D-Funktionsplotter, der es in sich hat, der Sie begeistern wird – und sämtliches Background-Wissen selbstverständlich.

Begeistern wird Sie aber auch die in Programm und Wort demonstrierte realistische Darstellung von dreidimensionalen Objekten mit verdeckten Flächen unter Berücksichtigung einer frei positionierbaren Lichtquelle etc.

Wenn Sie alles das hinter sich gelassen haben und sich denken, daß es ja jetzt gar nicht mehr besser kommen kann, dann muß ich Sie leider enttäuschen. Halten Sie sich fest! Haben Sie schon einmal dreidimensionale Grafiken höchster Perfektion eigenhändig programmiert unter Berücksichtigung von mehreren Lichtquellen, Schattenbildung, spiegelnder und diffuser Lichtreflexionen, total verspiegelter Objekte mit Mehrfachspiegelungen oder durchsichtiger Körper? Kennen Sie den Unterschied zwischen matten, polierten, metallischen oder nichtmetallischen Gegenständen? Ihre Bilder schon! Die Technik des Ray-Tracings macht es möglich: Basteln Sie sich Ihre eigene Welt, programmieren Sie sich Ihre eigenen Naturgesetze, Ihr eigenes Stilleben. Bilder, die Sie bisher höchstens in Diashows bewundert haben, können Sie in Zukunft selbst programmieren, selbst erstellen. Dieser absolute Höhepunkt des Buches wird nicht nur den Autor selbst faszinieren. Endlich können auch Sie eine »Jongleur-Demo« erstellen, nur noch perfekter.

Wenn Sie dann noch ein wenig Atem haben, dann sollten Sie sich das siebte Kapitel nicht entgehen lassen, das sich mit der erstaunlich realistischen Darstellung sogenannter Rotationskörper beschäftigt. Lichtquellen und Schattierungen sind dort bereits auch für Sie selbstverständliche Arbeitsmittel.

Was ist ein Buch ohne Anhang, ohne Nachschlagewerk? Im Anhang finden Sie noch einmal alle mathematischen Grundwerkzeuge kompakt zusammengestellt. Wir erklären auch grundlegende mathematische Prinzipien wie Winkelfunktionen usw. Als Vorwissen benötigen Sie nur Grundkenntnisse der Algebra.

Ebenfalls im Anhang finden Sie eine vollständige Zusammenstellung aller im Buch verwendeten Library-Funktionen mit Übergabeparametern und Beschreibung.

Ausführliche Literaturhinweise sowie ein umfangreiches Stichwortverzeichnis, in dem Sie alles sofort auf einen Blick wiederfinden, runden das Werk ab.

KAPITEL 2

Grafikgrundlagen für den Amiga

Daß der Amiga phänomenale Grafikeigenschaften besitzt, brauchen wir hier nicht wieder zu betonen. Wie Sie allerdings als Programmierer Gebrauch davon machen können, das sollten wir uns doch einmal genauer anschauen.

In diesem Kapitel erfahren Sie die allernötigsten Grundlagen, die Sie kennen sollten, wenn Sie mit dem Amiga Grafik erzeugen wollen. Angefangen von den grundsätzlichen Grafikmöglichkeiten (Auflösungen, Farben etc.) werden wir hier kurz und knapp beschreiben, wie Sie von Amiga-Basic aus oder aus der Sprache C die verschiedensten Grafikoperationen ansteuern können. Das hat etwas mit den Basic-Befehlen zur Grafik und natürlich mit der Grafik-Library zu tun. Hier werden Sie also die Grundlagen für die vielen Beispielprogramme des Buches vermittelt bekommen. Selbstverständlich ist es Ihnen überlassen, das Kapitel zu überspringen, wenn Sie bereits in Besitz der notwendigen Kenntnisse sind.

2.1 Die Farben des Amiga

Um die Farbgebung des Amiga – und im übrigen aller RGB-Rechner und -Monitore – zu verstehen, müssen wir einen kleinen Exkurs durch die »Allgemeine Farbenlehre« unternehmen.

Sie alle kennen sicher die Technik, mit Wasserfarben durch geschicktes Mischen aus den drei Grundfarben Rot, Blau und Gelb alle anderen Farben herzustellen. Violett beispielsweise entsteht dabei aus der Mischung von Rot und Blau, Grün durch Blau und Gelb und Schwarz durch Zugabe aller drei Farben. Ist uns ein Violett nicht blau genug, dann geben wir kurzerhand noch ein wenig von dieser Grundfarbe hinzu etc. Je mehr wir also von den Grundfarben hinzugeben, desto dunkler wird die resultierende Mischfarbe. Wir bezeichnen dieses Phänomen deshalb auch als Subtraktive Farbmischung.

Auch Goethe untersuchte diese Form der Farbmischung und entwickelte seinen berühmten Goethe-Farbkreis, den wir weiter unten – allerdings für unsere Zwecke modifiziert – skizziert haben.

Eine etwas andere Form der Farbmischung stellt die sogenannte Additive Farbmischung dar. Hierbei handelt es sich um die Mischung farbigen Lichtes. Auch hier existieren wieder drei Grundfarben – hier sind es Rot, Grün und Blau –, aus deren Kombination wir alle anderen Lichtfarben mischen können (das Ganze ist natürlich kein unerklärliches Naturphänomen, sondern hat mit der Art und Weise zu tun, wie unser Auge Farben identifiziert). Je intensiver jedoch hier die Grundfarben zugemischt werden, desto heller erscheint die resultierende Farbe (deshalb der Name Additive Farbmischung). Eine Kombination gleicher Teile aller drei Grundfarben beispielsweise ergibt die »Farbe« Weiß. Welche Farben aus welchen Grundfarben gemischt werden können, zeigt Ihnen die folgende Skizze.

Diese Tatsachen machen sich nun die modernen Farb-Monitore und -Fernseher zunutze. Jeder einzelne Punkt auf dem Bildschirm setzt sich in Wahrheit nämlich aus drei dicht nebeneinander

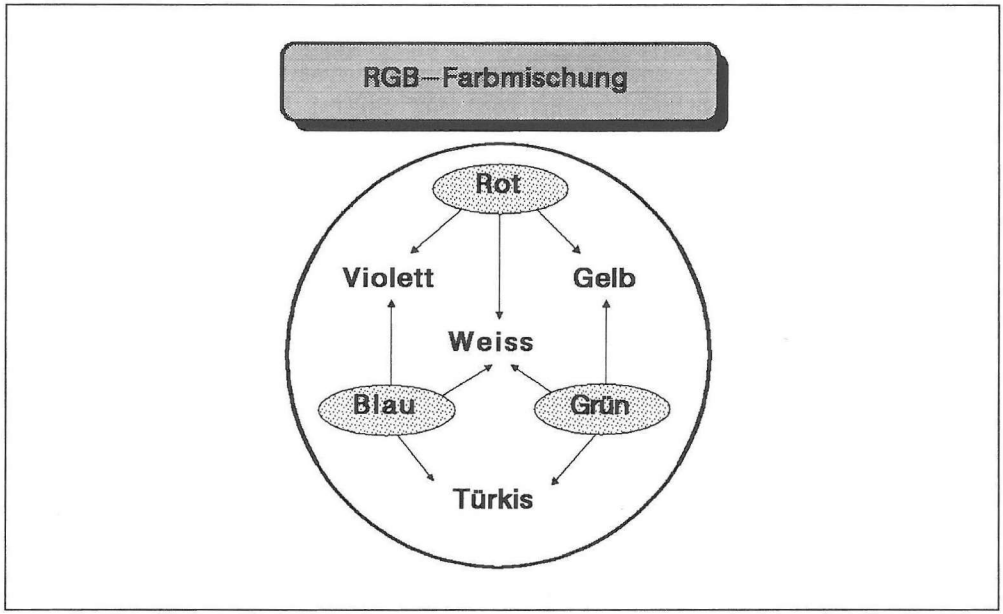


Bild 2.1: Additive Farbmischung

liegenden farbigen Punkten zusammen (schauen Sie einmal genau hin). Sie besitzen eben die drei Grundfarben Rot, Grün und Blau (RGB). Aufgrund der Entfernung unseres Auges von der Mattscheibe registrieren wir nur einen einzigen Punkt, dessen Farbe die Mischung der drei Einzelpunktfarben darstellt. Soll ein Punkt nun beispielsweise in Weiß erscheinen, so leuchten alle drei Einzelpunkte in gleicher Intensität auf. Für Gelb treten nur Rot und Grün in Aktion und bei Schwarz haben alle drei »Leuchter« Funkstille.

Da unser Amiga ein farbiges Monitorbild produzieren soll, muß auch er wissen, aus welchen Grundfarben sich die Farbe eines Punktes zusammensetzen soll. Dabei ist natürlich ebenfalls die Intensität der einzelnen Grundfarben wichtig. Der Programmierer hat nun die Möglichkeit, genau diese Informationen anzugeben. Der Amiga stellt dabei für jede Grundfarbe eine Palette von 16 Intensitätsstufen von Schwarz bis »Volle Leuchtkraft« zur Verfügung. Sie sind nun in der Lage, aus einer Kombination dieser drei Grundfarben mit jeweils beliebiger Intensität diejenige resultierende Farbe auszuwählen, die Ihnen gefällt. Sie können dabei aus einer Auswahl von 4096 Kombinationen aussuchen (für jede Grundfarbe 16 Intensitäten ergeben $16 \times 16 \times 16 = 4096$ Farben). Daraus ergibt sich eine Farbvielfalt, die gerade bei 3-D-Bildern besonders zum Tragen kommt und auch – man glaubt es kaum – voll ausgeschöpft wird. Teure und spezielle Grafikterminals erlauben oft noch um Größenordnungen feinere Farbwahlen (mehr als eine Million Farben etc.), was sich entsprechend auf die Bildqualität niederschlägt.

Es gibt beim Amiga zwar einen Modus, unter dem sich mit einem speziellen Trick auch alle 4096 Farben gleichzeitig darstellen lassen (in einem anderen Modus sind es 64), normalerweise aber kann er zur selben Zeit maximal 32 dieser 4096 Farben auf dem Bildschirm anzeigen. Welche Farben das nun sind, steht Ihnen natürlich wiederum völlig frei. Haben Sie sich 32 passende Farben ausgesucht, dann speichern Sie diese in 32 sogenannte Palettenregister (s. Skizze). Aus diesen Registern (von 0 bis 31 durchnumeriert) entnimmt Ihr Amiga dann die Information, mit welcher Farbe er einen bestimmten Punkt anfärben soll. Das Palettenregister Nummer 0 spielt dabei eine besondere Rolle. Es stellt stets das Hintergrund-Farbregister dar.

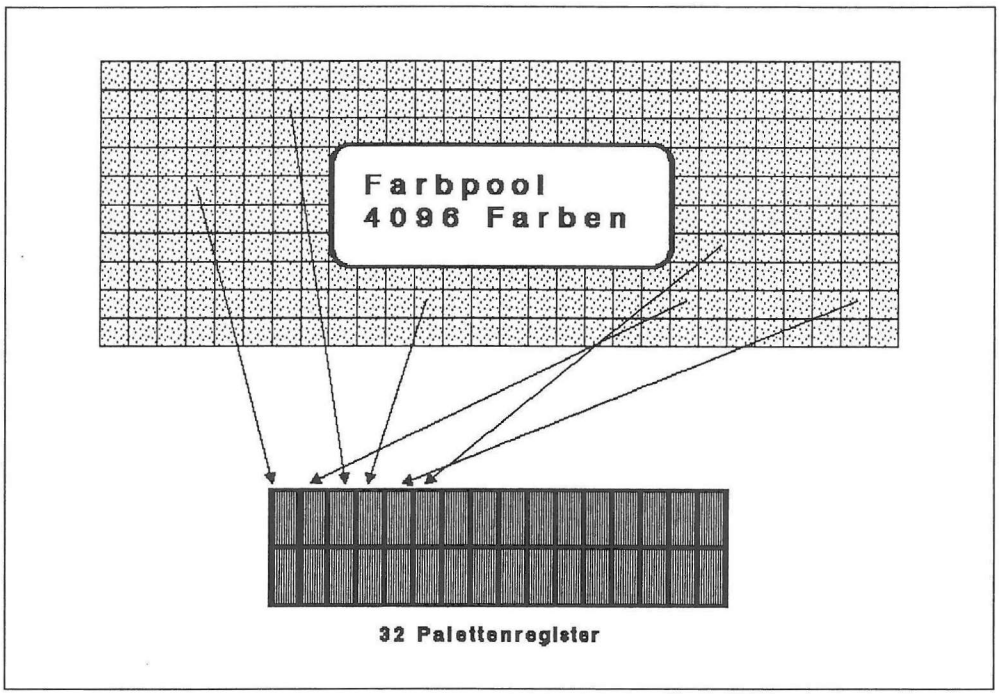


Bild 2.2: Auswahl von 32 Palettenfarben

Später beim Zeichnen (bzw. im Grafikspeicher selbst; s.u.) geben Sie dann für einen einzelnen Punkt nur noch die Nummer desjenigen Palettenregisters an, aus dem er seine Farbe beziehen soll. Die 32 Palettenregister spielen dann sozusagen die Rolle von 32 Pinseln, die in eine bestimmte Farbe getaucht wurden und dann zum Zeichnen Verwendung finden.

Wie wird nun die Farbe in einem solchen Palettenregister gespeichert?

Angegeben werden müssen ja in jedem Fall die Intensitätswerte (0–15) der drei Grundfarben Rot, Grün und Blau. Dies geschieht in Amiga-Basic auf einfache Weise durch den Befehl PALETTE. Hier können Sie die Intensitätswerte sogar wahlweise als Werte zwischen 0,00 und 1,00 angeben (prozentualer Anteil).

Auch in C bzw. bei der Verwendung der Graphics-Library-Funktion **SetRGB4()** bzw. **SetRGB4MC()** – wir werden auf die Library-Funktionen in diesem Kapitel noch zu sprechen kommen – brauchen Sie nur die Werte für Rot, Grün und Blau anzugeben.

In Wahrheit werden diese drei Werte aber zusammengepackt in ein 32-Bit-Register abgelegt, das den folgenden Aufbau hat:

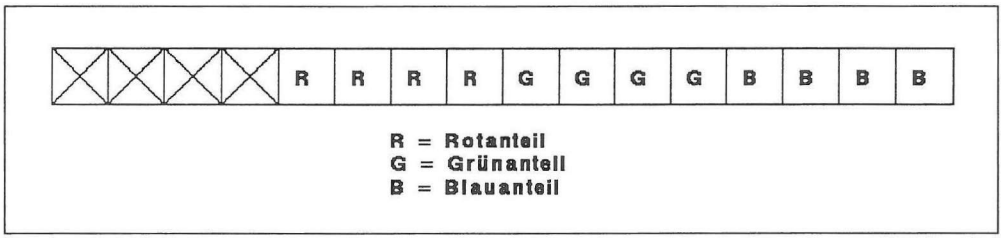


Bild 2.3: Aufbau eines Palettenregisters

Mit diesem Aufbau haben Sie jedoch beim Amiga kaum etwas zu tun, da Sie ja – wie gesagt – spezielle Funktionen verwenden können, die für die richtige Zuordnung bereits sorgen.

2.2 Grafikauflösungen und Bild-Koordinatensysteme

Wie viele Rechner besitzt auch Ihr Amiga verschiedene Grafikauflösungen für die verschiedensten Zwecke. Er verwendet dabei allerdings ein recht flexibles und variationsreiches System zur Wahl der richtigen Grafik- und Farb-Auflösung.

a) Die Standard-Auflösungen:
Unter »Standard-Auflösungen« verstehen wir hier zwei verschiedene Modi, die am einfachsten zu handhaben sind und damit wohl am weitaus häufigsten Anwendung finden. Es sind dies:

Niedrige Auflösung:	320x256 Punkte (320x200 unter NTSC) max. 32 Farben gleichzeitig
Hohe Auflösung:	640x256 Punkte (640x200 unter NTSC) max. 16 Farben gleichzeitig

(Unter 320x200 versteht man 320 Punkte in der Waagerechten bzw. x-Richtung und 200 Punkte in der Senkrechten oder auch y-Richtung. NTSC ist die amerikanische Fernsehnorm, wir arbeiten mit dem PAL-System.) Die Workbench beispielsweise arbeitet »serienmäßig« in der hohen Auflösung, in der 80 Zeichen pro Zeile möglich sind. Die niedrige Auflösung findet meist dann Verwendung, wenn es entweder um Speicherersparnis (jeder Punkt mehr auf dem Bildschirm kostet natürlich mehr Bildschirmspeicher) oder vor allem um mehr Farben geht. Den Zusammenhang werden wir sofort noch eingehender besprechen. Natürlich spielen die Fähigkeiten Ihres Monitors (Fernsehers?) ebenfalls eine Rolle bei der

Auswahl des richtigen Modus. Beachten Sie bitte, daß in der hohen Auflösung jeder Punkt halb so breit aber gleich hoch wie in der 320x200er ist. Die niedrige Auflösung stellt (fast) quadratische Punkte dar. Aus diesem Grunde kann es eventuell zu Verzerrungen kommen.

b) Interlace-Modus (Zeilensprungverfahren):
Die beiden Standard-Bildschirm-Auflösungen variieren – das ist leicht zu sehen – lediglich in der Waagerechten. Dies liegt an der Geschwindigkeit der Video-Prozessoren, die ja in der gleichen Zeit mehr Punkte bearbeiten und darstellen müssen, je höher die Auflösung ist.

Es gibt jedoch eine Möglichkeit, auch die y-Auflösung zu verdoppeln. Dies geschieht im sogenannten Zeilensprungverfahren (Interlace-Modus).

Normalerweise wird ein Bild auf dem Monitor 50mal (unter NTSC 60mal) pro Sekunde Zeile für Zeile von einem Elektronenstrahl neu aufgebaut, um ein möglichst flackerfreies Bild zu erhalten. Im Interlace-Modus geschieht dagegen folgendes: Wird ein Bild »zu Monitor« gebracht, so wird zunächst nur jede zweite Zeile abgetastet. Zwischen zwei Zeilen bleibt dann natürlich jeweils eine Leerzeile. Beim nächsten Bildaufbau aber werden dafür nur die fehlenden Zwischenzeilen vervollständigt. Ein ganzes Bild braucht also zwei Durchgänge, bevor alle Punkte dargestellt wurden – das ist dann nur noch 25mal (unter NTSC 30mal) pro Sekunde. Aus diesem Grunde kommt es – je nach Farbenwahl – zu einem mehr oder weniger starken Flackern des Bildes, dem man sich nicht gern länger aussetzt (ein Tip: drehen Sie den Kontrast herunter oder verwenden Sie kontrastarme Farben, das Tragen einer Sonnenbrille soll auch helfen).

Aber wir können die beiden Standard-Auflösungen erweitern:

Niedrige Auflösung, Interlace:	320x512 (bzw. 320x400) max. 32 Farben gleichzeitig
Hohe Auflösung, Interlace:	640x512 (bzw. 640x400) max. 16 Farben gleichzeitig

Die Anwendungen sind klar: Große Auflösung bringt großen Überblick und viele Details. Besonders bei CAD-Programmen, Tabellenkalkulationen oder Textverarbeitungen sind dies entscheidende Vorteile. Noch ein Vorteil des Interlace-Modus: Die Anzahl der verfügbaren Farben nimmt im Vergleich zu den Standard-Modi nicht ab! Bei Bildschirmfotos spielt das Flackern keine Rolle, da hier mit Belichtungszeiten von ca. 1 Sekunde gearbeitet wird.

c) Hold-and-Modify-Modus (HAM):
Hochinteressant wird es bei diesem Spezialmodus Ihres Amiga. Hier haben Sie tatsächlich die Möglichkeit, alle 4096 möglichen Farben gleichzeitig auf dem Bildschirm darzustellen. Dafür müssen wir allerdings auch einige Einschränkungen in Kauf nehmen:

Grundsätzlich arbeitet HAM nur unter einer einzigen Auflösung:

Hold-and-Modify: 320x256 (bzw. 320x200)

Bezüglich der Farbgebung der einzelnen Punkte sind zwei Fälle zu unterscheiden, die nebeneinander im gleichen Bild auftauchen können. Zum einen haben Sie die Möglichkeit, die Farbe für einen Punkt ganz normal aus 16 Palettenregistern auszuwählen (ähnlich wie

im hochaufgelösten Modus). Zum anderen aber können Sie auch Farben einstellen, die nicht in diesen 16 Registern stehen. Dies geht folgendermaßen vor sich:

Die Farbe eines Punktes ist stets abhängig von der Farbe des links danebenliegenden. Von diesem vorherigen Punkt ist nämlich noch die Farbe bekannt, die sich ja bekanntlich aus den drei Grundfarben Rot, Grün und Blau zusammensetzt. Für die Farbe des aktuellen Punktes dürfen Sie nun nur die Intensität einer einzigen Grundfarbe ändern (z.B. Rot), die aber beliebig. Wollen Sie dagegen zwei oder alle drei Grundfarben ändern, dann müssen Sie das schrittweise über ein oder zwei Zwischenpunkte tätigen.

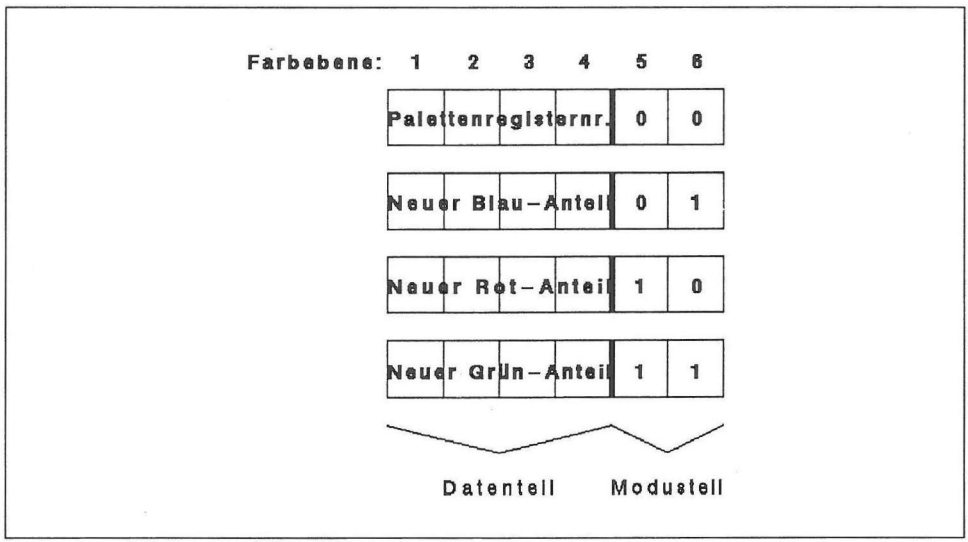


Bild 2.4: Das Prinzip des Hold-and-Modify-Modus

Beachten Sie bitte, daß der erste Punkt einer Bildschirmzeile stets in der Hintergrundfarbe erscheint (Inhalt des Palettenregisters 0). Es wird also nicht die Farbe des letzten Punktes der vorhergehenden Zeile übernommen.

Sie sehen, die Handhabung dieses Modus ist nicht ganz einfach. Aber gerade für Bilder, die viele verschiedene Farben oder Farbschattierungen enthalten sollen, ist er die einzige Möglichkeit.

d) Extra-Half_Brite-Modus:

In den neuen Amigas wurde von den Entwicklern Ihres Rechners noch ein weiterer Modus eingeführt, der sich ebenfalls geradezu hervorragend für 3-D-Effekte anbietet: Der Extra-Half_Brite-Modus. Er funktioniert nicht bei den alten Amiga 1000 und oft nicht richtig bei den älteren 500er-Modellen. Er ist in zwei verschiedenen Auflösungsstufen möglich:

Standard-Extra-Half_Brite:	320x256 (320x200) (64 Farben gleichzeitig)
Interlace-Extra-Half_Brite:	320x512 (320x400) (64 Farben gleichzeitig)

Wir haben es also mit einer Variation der niedrigen Auflösung zu tun. In dieser neuen Grafikart sind statt der maximal 32 Farben bei 320x256 insgesamt 64 verschiedene Farben gleichzeitig auf dem Bildschirm möglich. Irgendeinen Haken muß die Sache doch haben, oder? Richtig! Zum einen ist dieser Modus recht speicherintensiv, was noch zu verkraften wäre.

Zum anderen sind nicht alle 64 Farben völlig voneinander unabhängig. Die ersten 32 Farben (die normale Farbpalette) können Sie ganz normal mit irgendwelchen der 4096 zur Verfügung stehenden Farbmischungen belegen. Da es jedoch nur maximal 32 Farbregister gibt, wie wir gesehen haben, resultieren die oberen 32 Farben aus den Farben der unteren Farbregister. Hierzu teilt der Amiga die Farbintensitäten der unteren 32 Farbregister durch zwei. Die oberen 32 Farbregister besitzen also eigentlich die gleiche Farbe wie die unteren, nur halb so hell (daher der Name half brite = halb hell). Dabei bezieht Farbregister 32 die Farbe aus Register 0, Register 33 aus Register 1 usw. Beachten Sie, daß durch die Halbierung in den oberen 32 Farben nur Farbintensitätswerte von 0 bis 7 möglich sind.

Auflösung und Farborganisation:

Wir sind bisher noch nicht besonders ausführlich auf die Farbgebung in den einzelnen Auflösungen eingegangen. Das soll hier nachgeholt werden.

In allen Modi (außer HAM und Half_Brite) können Sie bestimmen, wie viele Farben Sie maximal gleichzeitig auf dem Bildschirm darstellen wollen. Dabei haben Sie folgende Werte zur Auswahl:

max. 2 Farben
max. 4 Farben
max. 8 Farben
max. 16 Farben
max. 32 Farben (nur für niedrige Auflösung)

Je mehr Farben Sie gleichzeitig darstellen möchten, desto mehr Speicherplatz verschlingt ein einziges Bild. Auch die Zeichengeschwindigkeit wird jeweils ein wenig abnehmen.

Die Farben beziehen sich bekanntlich stets auf die Palettenregister, d.h. ein Punkt mit Farbe 5 erhält die Farbe, die sich zur Zeit im Palettenregister 5 befindet. Im Bildschirmspeicher muß diese Nummer natürlich binär angegeben werden (5 entspräche dann der Bit-Kombination: %101). Je mehr Bits für die Farbangabe eines einzelnen Punktes benötigt werden (bei fünf: drei Bits), desto mehr Speicher wird insgesamt benötigt. Der Bildschirmspeicher ist nun in sogenannten Bildebenen organisiert.

Jede Bildebene enthält für alle Bildschirmpunkte jeweils ein Bit, das – zusammen mit den Bits der anderen Ebenen – die Punktfarbe angibt. Bildebene 0 also gibt alle nullten Bits der Farbregisternummer, Bildebene 1 alle ersten Bits usw. an (s. Bild 2.5).

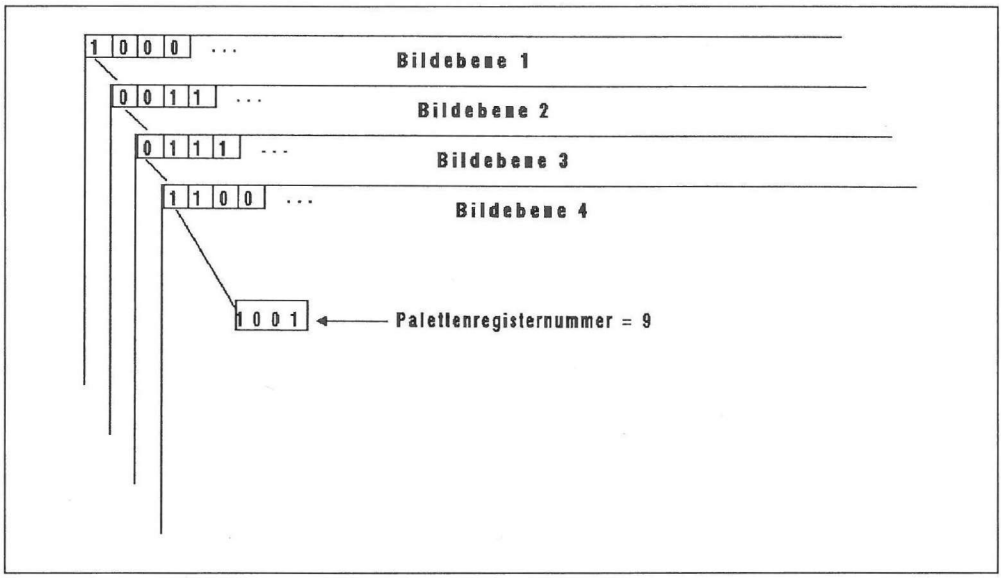


Bild 2.5: Bildschirmebenen und Farbkodierung

Ein Beispiel: Angenommen der oberste linke Punkt des Bildschirms soll seine Farbe aus Palettenregister 5 beziehen. Dann wird in Bildebene 0 im allerersten Bit eine 1 stehen, in Bildebene 1 eine 0, in Ebene 2 schließlich wieder eine 1.

Je nachdem, wie viele Farben Sie gleichzeitig darstellen wollen, benötigen Sie eine unterschiedliche Anzahl von Bildebenen:

max. 2 Farben – 1 Bildebene
max. 4 Farben – 2 Bildebenen
max. 8 Farben – 3 Bildebenen
max. 16 Farben – 4 Bildebenen
max. 32 Farben – 5 Bildebenen

Jede zusätzliche Ebene verbraucht zusätzlichen Speicherplatz. Der richtet sich natürlich nach der Anzahl der darstellbaren Punkte auf dem Bildschirm, also nach der Auflösung. Die folgende Tabelle gibt Ihnen den Speicherbedarf einer Ebene in den verschiedenen Auflösungen an:

Auflösung	Speicherplatz pro Ebene in Bytes
320x256 (320x200)	10240 (8000)
320x512 (320x400)	20480 (16000)
640x256 (640x200)	20480 (16000)
640x512 (640x400)	40960 (32000)

Tabelle 2.1: Speicherbedarf der Farbebenen in Abhängigkeit von der Auflösung (Werte in Klammer gelten für die amerikanische NTSC-Norm)

Der normale Workbench-Bildschirm beispielsweise besitzt eine Auflösung von 640x256 (640x200 in NTSC) Bildpunkten und läßt maximal 4 Farben gleichzeitig zu. Dazu werden also zwei Bildebenen benötigt, die jede eine Größe von 20480 (16000) Byte besitzt, macht zusammen einen Bildspeicherbedarf von 40960 (32000) Byte. Schalten Sie über »Preferences« den Interlace-Modus ein, dann verdoppelt sich der notwendige Speicher entsprechend.

Als absoluter Speicherfresser stellt sich dann natürlich ein Interlace-Bild mit 4 Bildebenen (max. 16 Farben) in einer Auflösung von 640x512 (320x400) heraus; Speicherbedarf: 163 840 (128 000) Byte pro Bild. Soviel Speicher besitzen manche moderne Computer nicht einmal. Ein Amiga 500 mit 512 Kbyte Speicher wird da schon ganz schön ins Schwitzen kommen, zudem er ja gleich eine Vielzahl von Bildschirmen (Screens) und Auflösungen gleichzeitig verwalten und auch gleichzeitig auf dem Monitor darstellen kann. Unter anderem ist deshalb auf kurz oder lang an einer Speichererweiterung wohl nicht vorbeizukommen.

Alle Bilder dürfen sich beim Amiga nur in den untersten 512 Kbyte befinden, sollen sie auf dem Bildschirm dargestellt werden. Denn nur auf diesen auch Chip-RAM genannten Speicherbereich haben auch die diversen Custom-Chips Zugriff, die als Coprozessoren u.a. dafür sorgen, daß Sie etwas sehen oder daß Ihre Grafiken schneller aufgebaut werden. Somit sind der Anzahl von Screens Grenzen gesetzt – wenn auch nicht allzu enge –, auch wenn Sie Ihren Arbeitsspeicher erweitern. Programme dürfen sich zwar auch in diesem Bereich tummeln, laufen aber – je nach Coprozessoraktivitäten – bis zu 30% langsamer ab, da die 68000-CPU stets auf die ebenfalls auf diesen Speicher zugreifenden Coprozessoren warten muß. Ideal wäre deshalb natürlich zusätzlicher Speicher – sogenannter Fast-RAM –, in dem die Programme ungestört für sich ablaufen können, wie das der Amiga 2000 bereits serienmäßig besitzt.

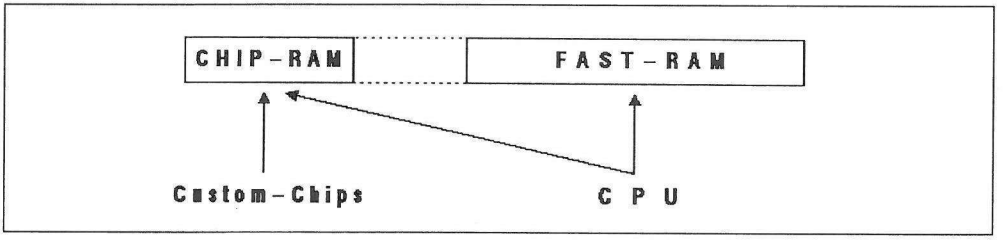


Bild 2.6: Speicherzugriff CPU und Coprozessoren

Die Farborganisation im HAM-Modus:

Eine Sonderrolle spielt – wie oben bereits erwähnt – die Farborganisation im Hold-and-Modify-Modus. Sie soll im folgenden kurz beschrieben werden:

Das Prinzip haben wir bereits kennengelernt. Die Farbe eines Punktes wird bestimmt, indem die Farbe des davorliegenden Punktes aufgegriffen und in einer der drei Grundfarben variiert wird. Wie aber teilen wir dem Rechner solches mit?

Nun, die Antwort liegt wieder in der Speicherorganisation eines HAM-Bildes: Jedes HAM-Bild besitzt nämlich 6 Bildebenen (Ebenen 0–5). Die ersten vier und die beiden letzten Ebenen gehören dabei jeweils zusammen. Ebenen 4 und 5 geben dabei an, was mit den vier Bits der Ebenen 0–3 zu geschehen hat:

Ebene 4	Ebene 5	Aufgabe der Ebenen 0–3
0	0	Die Bitkombination der Ebenen 0–3 gibt an, aus welchem Palettenregister der Punkt die Farbe beziehen soll (Register 0–15). Das enspricht der »normalen« Farbgebung.
0	1	Die Bitkombination der Ebenen 0–3 gibt den neuen Intensitätswert für die Grundfarbe Blau an. Die Intensitäten für Rot und Grün werden vom Vorpunkt übernommen.
1	0	Die Bitkombination der Ebenen 0–3 gibt den neuen Intensitätswert für die Grundfarbe Rot an. Die Intensitäten für Blau und Grün werden vom Vorpunkt übernommen.
1	1	Die Bitkombination der Ebenen 0–3 gibt den neuen Intensitätswert für die Grundfarbe Grün an. Die Intensitäten für Rot und Blau werden vom Vorpunkt übernommen.

Wahlweise können Sie dem HAM-Bild auch nur 5 Bildebenen zuweisen. Die Bits, die sonst aus der fehlenden Ebene (Ebene 5) kamen, werden dann vom Rechner als »0« angenommen. Ihnen sind dann also nur die Kombinationen 00 und 10 verfügbar.

Wie oben bereits gesagt, wird der erste Punkt einer Zeile stets in der Hintergrundfarbe erscheinen. Sie gehen also stets von dieser Farbe bei Ihren Manipulationen aus.

Die Farborganisation im Half_Brite-Modus:

Den ebenfalls bereits beschriebenen Half_Brite-Modus müssen wir genauso wie HAM einer Sonderbehandlung unterziehen. Der Aufbau ist allerdings ein wenig einfacher. Zusätzlich zu den maximal möglichen fünf Farbebenen der niedrigen Auflösung gesellt sich eine sechste. Das ist auch insofern logisch, als wir ja 64 verschiedene Farben kodieren wollen. Diese sechs Farbebenen sind im Half_Brite-Modus vorgeschrieben, Sie können also keine weglassen.

Mit den einzelnen Bits der sechs Ebenen verfährt Ihr Rechner ansonsten haargenauso wie in den normalen Farbauflösungen. Er ermittelt aus ihnen die Nummer desjenigen Farbregisters, aus dem der angesprochene Punkt seine Farbe beziehen soll. Dabei werden die Nummern 32–63 wie oben beschrieben ebenfalls aus den unteren 32 Palettenregistern gewonnen (allerdings mit halber Farbintensität).

2.3 Grafikansteuerung in Basic

Amiga-Basic besitzt bereits eine Anzahl von Befehlen zur Grafikerzeugung oder -manipulation. Aus diesem Grunde können wir problemlos einfache Grafiken auf den Bildschirm bringen. Im Anhang dieses Buches finden Sie eine kurze Zusammenstellung aller wichtigen reinen Grafikbefehle des Amiga-Basic. Detaillierte Informationen finden Sie in Ihrem Amiga-Basic-Handbuch, das jedem Amiga beigelegt sein sollte.

Was uns hier u.a. im besonderen interessieren soll, ist der Aufruf der Amiga-internen Bibliotheksfunktionen aus Basic heraus. Das Betriebssystem unseres Rechners nämlich stellt uns eine Reihe von grafischen Funktionen in seiner Graphics-Library (grafische Unterprogramm-Bibliothek) zur Verfügung. Teilweise greifen die Basic-Befehle bereits auf dieses Library zu (z.B. um eine Linie zu zeichnen oder Flächen zu füllen etc.).

Einige Funktionen der Graphics-Library sind allerdings auch in Basic nicht implementiert. In jedem Fall aber bedeutet der direkte Aufruf der Betriebssystem-Funktionen einen Geschwindigkeitsvorteil.

Hinzu kommt, daß Amiga-Basic mit der Version 1.1 des Betriebssystems rechnet. In der auch in Ihrem Rechner implementierten Version 1.2 jedoch sind einige Funktionen, wie das Zeichnen von Kreisen oder Ellipsen, hinzugekommen. Amiga-Basic behilft sich hier mit recht langsamen eigenen Routinen.

Zunächst einmal: Was sind überhaupt Libraries?

Ihr Amiga stellt dem Programmierer zu den verschiedensten Bereichen Funktionen zur Verfügung, mit denen er seinen Rechner steuern, Grafiken erzeugen, Bildschirmfenster kreieren oder mathematische Funktionen berechnen kann. Diese Funktionen sind zusammengefaßt in mehrere Bibliotheken. Hier die für unsere Belange wichtigsten Libraries:

Mathematische Bibliotheken:

- mathffp.library (ROM-resident)
- mathtrans.library (Disk-resident)
- mathieeedoubbas.library (Disk-resident)

Sie enthalten umfangreiche Befehle für Grundrechenarten, Winkelfunktionen usw. Nur »mathffp.library« befindet sich direkt im ROM. Die anderen Bibliotheken stehen auf Ihrer Workbench-Disk und müssen bei Bedarf erst vom Betriebssystem nachgeladen werden.

Grafische Bibliothek:

- graphics.library (ROM-resident)

Die Grafische Bibliothek enthält elementare Funktionen für Linien, Ellipsen, Flächen und sämtliche Animations-Befehle (Sprites, Bobs...).

Intuition-Bibliothek:

– intuition.library (ROM-resident)

Sie wird verwendet, um die bekannte Benutzeroberfäche mit Screens, Windows, Menüs, Requesters (Nachfragefenster) usw. auch für eigene Programme zu steuern.

Multitasking-Systembibliothek:

– exec.library (ROM-resident)

Diese Bibliothek steuert alle wesentlichen Systemfunktionen des Rechners bezüglich Multitasking, Speicheraufteilung, Libraryverwaltung etc.

Daneben existieren noch Bibliotheken wie »exec_support.library«, »clist.library«, »layers.library«, »dos.library«, »icon.library« und »diskfont.library«. Das Bibliothekensystem ist »offen«, d.h. es können jederzeit eigene Bibliotheken hinzugefügt werden. Das bietet dem Programmierer natürlich enorme Möglichkeiten für seine Programmentwicklungen.

Bevor Sie die einzelnen Funktionen einer Bibliothek aufrufen können, muß sie geöffnet werden. Dadurch wird eine eventuell auf Disk liegende Library in den Speicher geladen und die Sprungleiste für die jeweiligen Befehle freigegeben.

In Basic geschieht das durch den Befehl LIBRARY, z.B.:

```
LIBRARY "graphics.library"
```

Hier wird die Grafische Bibliothek geöffnet. Die einzelnen Funktionen sind wie ein normales Assemblerprogramm aufzurufen mit CALL, z.B.:

```
CALL SetDrMd(adr,mod)
```

Das Sprunglabel – in diesem Fall »SetDrMd« – zeigt dabei genau auf die Adresse der anzuspringenden Routine, der die in den Klammern stehenden Parameter übergeben werden.

Mit einem **LIBRARY CLOSE** werden alle (max. 5) offenen Libraries in Basic wieder geschlossen.

Der Befehl **LIBRARY** macht aber noch etwas anderes. Damit Amiga-Basic die Namen der verschiedenen Bibliotheksfunktionen (z.B. »SetDrMd«) bekannt sind, lädt es eine sogenannte bmap-Datei von der Disk ein, die Informationen darüber enthält, welche Funktionen es gibt, wie sie heißen und welche Parameter in welcher Reihenfolge übergeben werden müssen. Für die »graphics.library« hieße die zugehörige bmap-Datei »graphics.bmap«. Diese Datei muß dann natürlich auch auf Ihrer aktuellen Diskette im aktuellen Ordner vorhanden sein, sonst gibt es eine Fehlermeldung.

Damit muß für jede Bibliothek, die Sie von Basic aus aufrufen wollen, eine solche bmap-Datei auf der Disk stehen. Schauen Sie einmal auf Ihrer Amiga-Basic-Diskette nach. Im Ordner »BasicDemos« befinden sich bereits die Dateien »graphics.bmap« und »exec.bmap«. Diese beiden Dateien kopieren Sie also am besten auf Ihre Basic-Arbeitsdiskette. Wie aber erhalten wir die übrigen bmap-Dateien?

Nun, ebenfalls auf Ihrer Basic-Diskette befindet sich ein Ordner namens »FD1.2«. Von CLI aus können Sie sich den Inhalt dieses Ordners auslisten lassen. Dort finden Sie Files mit so wohlklingenden Namen wie »mathieeedoubbas_lib.fd« oder »mathieeesingbas_lib.fd«. Allen gemeinsam ist das Suffix ».fd«. Es handelt sich dabei um reine ASCII-Files. Sie enthalten wie die bmap-Dateien alle Informationen über Funktionsnamen, -aufruf und Parameterübergabe – diesmal aber editierbar in ASCII.

Mit Hilfe des Basic-Programmes »ConvertFD« (ebenfalls im »BasicDemos«-Ordner, lesen Sie bitte den Programm-Kopf!) können diese fd-Dateien in bmap-Dateien umgewandelt werden. Sie brauchen also nur für jede Bibliothek das Programm »ConvertFD« aufzurufen und die richtigen Namen einzugeben, und schon haben Sie die passenden bmap-Dateien für alle Bibliotheken (das brauchen Sie für »graphics.bmap« und »exec.bmap« natürlich nicht mehr zu tun).

Am besten machen Sie das gleich einmal und speichern das Ergebnis auf Ihre Arbeitsdiskette, so haben Sie alle Bibliotheken gleich parat.

Die etwa 500 Funktionen der verschiedenen Libraries und noch einiges mehr werden übrigens in den vier (englischsprachigen) Büchern von Addison Wesley zum Amiga beschrieben: »ROM Kernel Reference Manual: Libraries and Devices«, »ROM-Kernel Reference Manual: Exec«, »Intuition Reference Manual« und »Hardware Reference Manual«. Im ersten und dicksten (Libraries and Devices) finden Sie übrigens eine vollständige Liste aller Bibliotheksfunktionen mit Beschreibung, Aufruf und Übergabeparametern. Es ist damit unabdingbare Voraussetzung für fortgeschrittenere Programmierer. Sicherlich wird es mit der Zeit adäquate Beschreibungen auch anderer Verlage geben (teilweise gibt es sie bereits, s. Anhang). Wenn in diesem Buch Gebrauch von den Funktionen gemacht wird, dann werden sie natürlich entsprechend erläutert.

Hier ein kleines Beispielprogramm, das eine Reihe von Ellipsen auf den Bildschirm bringt. Es erscheint hier in zwei Versionen. Zum ersten mit dem originalen Basic-Befehl **CIRCLE**, zum zweiten mit der Graphics-Funktion **DrawEllipse()**. Vergleichen Sie einmal die Zeichengeschwindigkeiten:

Programm 1:

```
FOR x=1 TO 400 STEP 2
  xc% = x
  yc% = 40*SIN(x/30)+100
  CIRCLE (xc%,yc%),50,,,,.6
NEXT x
```

Programm 2:

```
LIBRARY "graphics.library"
FOR x=1 TO 400 STEP 2
  xc% = x
  yc% = 40*SIN(x/30)+100
  CALL DrawEllipse(WINDOW(8),xc%,yc%,50,35)
NEXT x

LIBRARY CLOSE
```

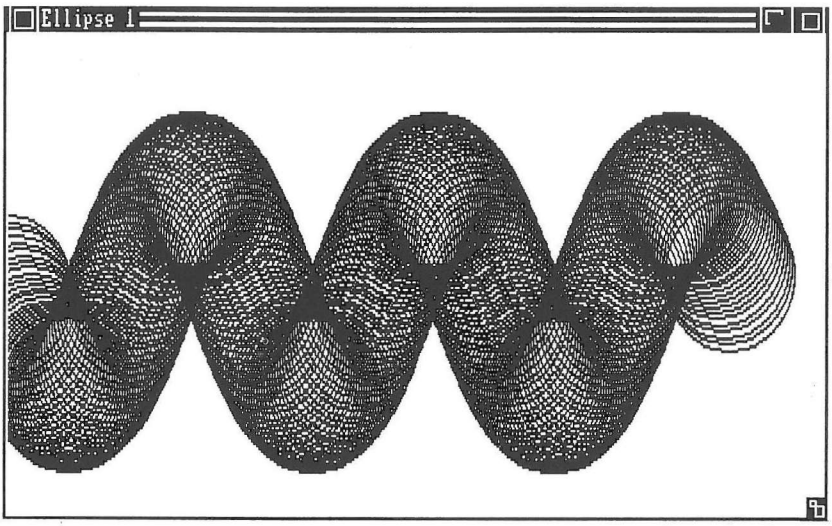


Bild 2.7: DrawEllipse in Amiga-Basic

In diesem Zusammenhang müssen wir noch auf die Programmierung der Screens (Bildschirme) und Windows (Fenster) eingehen. Diese Aufgabe gestaltet sich unter Amiga-Basic besonders einfach: Der Befehl **SCREEN** initialisiert einen neuen Bildschirm beliebiger Auflösung und Farbenzahl und weist ihm eine Kennziffer zu. Mit **SCREEN CLOSE** kann der Bildschirm wieder geschlossen werden (s. Amiga-Basic-Handbuch). Leider ist es unter Amiga-Basic zur Zeit nur möglich, Bildschirme mit maximal 200 Zeilen (400 im Interlace-Modus) zu erzeugen.

Grafikausgaben beziehen sich in Basic jedoch stets auf ein Fenster. Also müssen wir innerhalb des neuen Screens auch ein neues Fenster öffnen. Das erreichen wir durch Einsatz des Befehles **WINDOW**. WINDOW öffnet ein Fenster wählbarer Größe. Sie können dem Befehl sowohl die Kopfzeile als auch die Bestückung mit den verschiedenen Gadgets (Schließ-Knopf, Vergrößerungs-Knopf etc.) angeben. Jedes Fenster bezieht sich auf die Screen-Kennziffer des Bildschirms, in dem es geöffnet werden soll. Auch ein Fenster erhält eine entsprechende Kennziffer, die benötigt wird, um die Ausgabebefehle richtig zu leiten. Beispiele zur Programmierung von Screens und Windows werden wir zur Genüge in den späteren Kapiteln dieses Buches antreffen. Auch Fenster haben in Amiga-Basic maximal 200 Zeilen (400 im Interlace-Modus).

Aktuelle Informationen über das momentane Ausgabefenster erhalten Sie durch die wichtige Basic-Funktion **WINDOW(n)** (n bestimmt dabei die Art der Information, die wir erfragen wollen). Schauen Sie hierzu bitte in Ihrem Handbuch nach. Besonders interessieren uns die Werte 7 und 8 für n:

WINDOW(7) übergibt uns die Adresse eines Datensatzes (in C auch Struktur genannt), der ständig die aktuellen Eigenschaften des Fensters enthält (teilweise auch mit anderen WINDOW()-Aufrufen erfahrbar), so z.B. die momentane Maus-Cursor-Position etc. Diese Adresse

muß bei vielen Fenster-Operationen der Intuition-Library angegeben werden. Für weitere Informationen empfehle ich die Lektüre des »Intuition Reference Manual« (s.o.).

WINDOW(8) teilt uns die Adresse eines weiteren Datensatzes mit: RastPort. **RastPort** ist eine sehr wichtige Zusammenfassung von Variablen und Eigenschaften von Fensterinhalten. Er gibt Auskunft über den aktuellen Grafikspeicher, der für das jeweilige Fenster reserviert wurde etc. Für jedes Fenster existiert ein solcher RastPort. WINDOW(8) teilt uns die Adresse für das aktuelle Ausgabefenster mit. Für uns ist **WINDOW(8)** nur deshalb wichtig, da diese Adresse als Übergabeparameter bei fast jeder Funktion der Graphics-Library angegeben werden muß, damit das Betriebssystem weiß, in welches Fenster es zeichnen soll. Die Anwendung können Sie dem obigen Ellipsen-Beispiel entnehmen.

2.4 Grafikansteuerung in C

Bevor Sie dieses Kapitel lesen, sollten Sie sich kurz einmal mit der Grafikansteuerung in Basic beschäftigen, da sich vieles auch auf C übertragen läßt.

Die Ansteuerung von Grafik, Screens und Windows gestaltet sich in C aufgrund der vielen Deklarationen ein wenig komplizierter als in Basic. Im Prinzip läuft aber vieles genauso ab. Die Standard-Libraries brauchen Sie natürlich nicht mehr selbst zu schaffen. Sie stehen auf Ihrer C-Diskette und werden vom Linker automatisch eingelinkt. Aufgrund der Vielzahl von Strukturen, die Sie zum Aufrufen der Library-Routinen benötigen (als Übergabeparameter) und die teilweise aktuelle Systemzustände widerspiegeln, existieren eine Reihe von Include-Files für jede Bibliothek (sie sind auf Ihrer C-Diskette am Suffix ».h« zu erkennen). Sie enthalten die Initialisierungen für die verschiedensten Strukturen und definieren verschiedene Präprozessoranweisungen. Am besten listen Sie alle Include-Files einmal aus und heften sie geordnet ab. So können Sie jederzeit Strukturnamen und -inhalte etc. nachschlagen. Sie finden die kommentierten Include-Files aber auch in den vier erwähnten Büchern von Addison Wesley im vorhergehenden Abschnitt.

Alle C-Programme in diesem Buch laufen, das haben wir bereits in der Einleitung erwähnt, sowohl auf dem Aztek C-Compiler als auch mit dem Lattice-Compiler ab der Version 3.10. Damit das gewährleistet ist, müssen Sie allerdings einige Dinge beachten:

Zum Aztek-Compiler:

Die Version, die dem Autor zur Verfügung stand und für die die Garantie übernommen werden kann, daß alles funktioniert, lautet: V3.4a. Normalerweise sollten aber auch bei anderen Versionen keine Probleme auftauchen. Kompilieren und assemblieren können Sie auf ganz normale und gewohnte Art und Weise. Das so entstandene Objekt-File (Endung ».o«) muß dann nur noch mit den Libraries gelinkt werden. Verwenden Sie dabei folgende Link-Syntax:

```
ln filename.o -lc -lm
```

Damit werden die normalen Library-Funktionen und die FFP-Library zusammengelinkt.

Zum Lattice-Compiler:

Bis auf das Ray-Tracing-Programm werden alle anderen C-Programme anstandslos von allen Lattice-Versionen kompiliert. Da die Einbindung der FFP-Fließkommaroutinen aber erst seit der Version V3.10 problemlos funktioniert, müssen Sie das Ray-Tracing-Programm – sollten Sie in Besitz einer niedrigeren Version sein – erst ein wenig umschreiben.

Kompilieren Sie ab V3.10 grundsätzlich mit den folgenden Befehlssequenzen:

```
lc1 -f -i:include/ -i:include/lattice filename.c
lc2 filename
blink lib:c.o+filename.o LIBRARY lib:lc.lib+
lib:amiga.lib+lib:lcmffp.lib TO filename
```

Es kann sein, daß Sie statt **lib:c.o** das Objektfile **lib:lstartup.obj** einlinken müssen.

Das folgende kleine Demo-Programm soll Ihnen die Programmierung von Screens, Windows und Grafik ein wenig näher bringen:

```
/*************************************/
/**                                 **/
/**    Programmierung von Screens,  **/
/**      Windows und Grafik in C    **/
/**                                 **/
/*************************************/


#include <exec/types.h>
#include <intuition/intuition.h>

/* Funktionsdeklarationen (nur für Aztek-C-Compiler): */
/* wahlweise auch: #include <functions.h>            */
/*****************************************************/
VOID                    CloseLibrary();
VOID                    CloseScreen();
VOID                    CloseWindow();
VOID                    Draw();
VOID                    Exit();
struct Message *        GetMsg();
VOID                    Move();
struct Library *        OpenLibrary();
struct Screen *         OpenScreen();
struct Window *         OpenWindow();
VOID                    RectFill();
VOID                    ReplyMsg();
VOID                    SetAPen();
VOID                    SetDrMd();
```

```
LONG                    Text();
LONG                    Wait();
/*******************************************************/


struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

 /* Struktur für neuen Font initialisieren: */
struct TextAttr NeuerFont =
{
   "topaz.font",        /* Name des Fonts    */
   TOPAZ_SIXTY,         /* Fontgröße         */
   FS_NORMAL,           /* Stil              */
   FPF_ROMFONT,         /* Voreinstellungen */
};

 /* Struktur für einen neuen Bildschirm
    initialisieren (kann natürlich auch
    innerhalb des Programmes erfolgen):
    */
struct NewScreen NeuerBildschirm =
{
   0,                   /* x-Koord. linke obere    */
                        /* Ecke (immer 0)          */
   0,                   /* y-Koord. linke obere    */
                        /* Ecke                    */
   320,                 /* Bildschirmbreite        */
   256,                 /* Bildschirmhöhe          */
   2,                   /* Bildebenenzahl          */
   0,                   /* Farbe der Details       */
   1,                   /* Farbe der Flächen       */
   NULL,                /* Grafikmodus: 320x200    */
   CUSTOMSCREEN,        /* Bildschirmtyp           */
   &NeuerFont,          /* Zeiger auf eigenen Font */
   "Demo-Bildschirm",   /* Text im Bildschirm-Kopf */
   NULL,                /* unbenutzt, immer NULL   */
   NULL,                /* keine eigene BitMap     */
};

/* Struktur für ein neues Fenster initialisieren: */
struct NewWindow NeuesFenster =
{
```

```
    20,                 /* x-Koord. linke obere Ecke */
    20,                 /* y-Koord. linke obere Ecke */
    300,                /* Fensterbreite             */
    100,                /* Fensterhöhe               */
    0,                  /* Farbe der Details         */
    1,                  /* Farbe der Flächen         */
    CLOSEWINDOW |       /* Rückmeldung bei:          */
     NEWSIZE,           /* Fenster zu, Fenstergröße  */
                        /* geändert                  */
    WINDOWCLOSE |       /* Fensterelemente und -typ  */
     SMART_REFRESH |    /* wählen                    */
     ACTIVATE |
     WINDOWSIZING |
     WINDOWDRAG |
     WINDOWDEPTH |
     NOCAREREFRESH |
     GIMMEZEROZERO,
    NULL,               /* keine eigenen Gadgets     */
    NULL,               /* CheckMark                 */
    "Demo-Fenster",     /* Text im Fenster-Kopf      */
    0,                  /* Adresse Screen-Struktur   */
                        /* Muß innerhalb des         */
                        /* Programmes nach dem       */
                        /* Öffnen eines Screens      */
                        /* initialisiert werden      */
    NULL,               /* kein SuperBitmap-Fenster  */
    100,                /* Mindestbreite             */
    25,                 /* Mindesthöhe               */
    320,                /* Maximalbreite             */
    256,                /* Maximalhöhe               */
    CUSTOMSCREEN,       /* Bildschirmtyp             */
};


main()
{
    struct Screen *Bildschirm;
    struct Window *Fenster;
    struct IntuiMessage *Meldung;
    struct RastPort *rasterport;

    LONG    i;
    LONG    fensterbreite,
            fensterhoehe;
    ULONG   class;
```

```
/* Öffnen der Intuition- und der Graphics-          *
 * Bibliotheken. Die Funktion OpenLibrary()         *
 * gibt einen Pointer auf die Bibliothek zurück,    *
 * der gespeichert werden muß, damit das            *
 * C-Programm darauf zugreifen kann. Ist dieser     *
 * Pointer gleich Null, dann liegt ein Fehler vor:  *
 */
IntuitionBase =
(struct IntuitionBase *)OpenLibrary("intuition.library",0L);
if (IntuitionBase == NULL)  Exit(FALSE);

GfxBase =
(struct GfxBase *)OpenLibrary("graphics.library",0L);
if (GfxBase == NULL)
{
   CloseLibrary(IntuitionBase);   /* Intuition-Library close */
   Exit(FALSE);                   /* Ausgang                 */
}


/* Nun wird der Bildschirm geöffnet. Die Funktion   *
 * OpenScreen() gibt dabei einen Pointer auf die    *
 * aktuelle Screen-Struktur aus, die wir uns merken *
 * müssen. Ist er Null, liegt ein Fehler vor:       *
 */
Bildschirm =
(struct Screen *)OpenScreen(&NeuerBildschirm);
if (Bildschirm == NULL)
{
   CloseLibrary(GfxBase);         /* Graphics-Library close  */
   CloseLibrary(IntuitionBase);   /* Intuition-Library close */
   Exit(FALSE);                   /* Ausgang                 */
}


/* Ein Fenster wird durch OpenWindow geöffnet. Die  *
 * Funktion liefert einen Pointer auf die aktuelle  *
 * Window-Struktur. Ist er Null, liegt ein Fehler vor: *
 */
NeuesFenster.Screen = Bildschirm; /* Adresse der Bildschirm-
                                     struktur setzen      */
Fenster =
(struct Window *)OpenWindow(&NeuesFenster);
if (Fenster == NULL)
{
```

```
   CloseScreen(Bildschirm);        /* Bildschirm schließen    */
   CloseLibrary(GfxBase);          /* Graphics-Library close  */
   CloseLibrary(IntuitionBase);    /* Intuition-Library close */
   Exit(FALSE);
}


/* Damit befindet sich ein neues Fenster in einem      *
 * neuen Bildschirm. Jetzt wird zur Demonstration      *
 * eine kleine Grafik mit Text ausgegeben:             *
 */

/* Adresse des Rasterports: */
rasterport = Fenster->RPort;

do
{
   fensterbreite = Fenster->Width - 1;
   fensterhoehe = Fenster->Height - 1;

   SetAPen(rasterport, 2L);         /* Zeichenfarbe setzen    */
   SetDrMd(rasterport, (LONG)JAM1);/* Zeichenmodus auf JAM1  */

   for (i=0; i < fensterbreite+1; i=i+3)
   {
      /* Grafikcursor positionieren und Linie zeichnen: */
      Move(rasterport, fensterbreite-i, fensterhoehe);
      Draw(rasterport, i, 0L);
   }

   for (i=0; i < fensterhoehe+1; i=i+3)
   {
      /* Grafikcursor positionieren und Linie zeichnen: */
      Move(rasterport, fensterbreite, i);
      Draw(rasterport, 0L, fensterhoehe-i);
   }

   /* Text einschreiben:              */
   SetDrMd(rasterport, (LONG)JAM2); /* Zeichenmodus auf JAM2  */
   Move(rasterport, fensterbreite/2-45L, fensterhoehe/2+3L);
   Text(rasterport, "Fensterdemo",11L);
```

```
        /* Auf Message warten:              */
        Wait(1L << Fenster->UserPort->mp_SigBit);

        /* Empfangene Meldung bearbeiten: */
        Meldung = (struct IntuiMessage *)
        GetMsg(Fenster->UserPort);          /* Meldeadresse       */
        class = Meldung->Class;             /* Art der Meldung    */
        ReplyMsg(Meldung);                  /* Meldung quittieren */

        if ((class & NEWSIZE) != 0)         /* Fenstergröße wurde*/
        {                                   /* verändert!         */
           SetAPen(rasterport, 0L);         /* Zeichenfarbe=0     */
           SetDrMd(rasterport, (LONG)JAM1); /* Zeichenmodus: JAM1*/
           RectFill(rasterport, 0L, 0L,     /* Fenster löschen    */
                    fensterbreite, fensterhoehe);
        }

    } while ((class & CLOSEWINDOW) == 0);   /* bis Fenster        */
                                            /* geschlossen        */

    CloseWindow(Fenster);                   /* Fenster schließen */
    CloseScreen(Bildschirm);                /* Bildschirm ebenso */
    CloseLibrary(GfxBase);                  /* Graphics-Library  */
    CloseLibrary(IntuitionBase);            /* Intuition-Library */

    Exit(TRUE);                             /* Ausgang           */
}
```

Dieses Programm öffnet in einem neuen 320x200-Bildschirm ein Fenster, in dem eine Grafik und ein Text ausgegeben werden. Das Fenster ist frei beweglich und kann von Ihnen vergrößert bzw. verkleinert werden, wobei sich der Fensterinhalt stets an die neue Größe anpaßt. Um das Programm zu beenden, betätigen Sie einfach das CLOSER-Gadget (schließen also das Fenster).

Da wir mit den Exec-, Graphics- und Intuition-Libraries arbeiten wollen, benötigen wir zur Definition der verschiedenen Strukturen die Include-Files »exec/types.h« und »intuition/intuition.h«. Letzeres lädt die notwendigen Include-Files für die Graphics-Library automatisch nach.

Jetzt beginnt die Definition globaler Variablen und Strukturen. Hier sind zunächst die Pointer »*IntuitionBase« und »*GfxBase« zu nennen, die die Zeiger auf Strukturen liefern, die für jede Library angelegt werden. Sie werden später u.a. dafür genutzt, die Libraries wieder zu schließen.

Als nächstes wird die Struktur für einen neuen Zeichensatz initialisiert. Der Zeichensatz wird bei Bedarf automatisch nachgeladen. Da wir einen neuen Bildschirm öffnen wollen, müssen wir dem Betriebssystem gleichfalls eine Struktur für diesen neuen Bildschirm vorbereiten. Hier

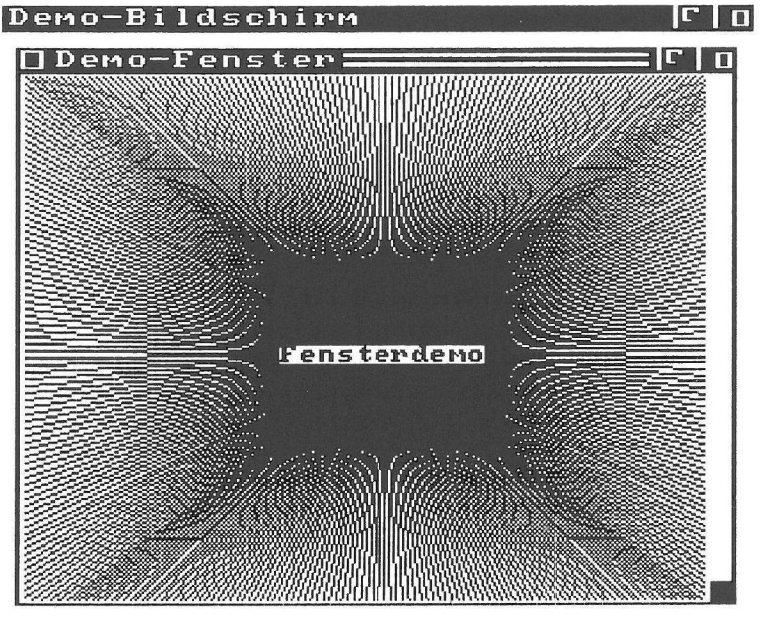


Bild 2.8: Screens, Windows, Grafik in C

geben wir die unterschiedlichen Merkmale (Auflösung, Größe, Farbenzahl, Font etc.) an, die er später annehmen soll. Wir hätten diese Struktur natürlich auch im Programm initialisieren können, indem wir jedem einzelnen Feld den entsprechenden Wert zugewiesen hätten. So geht es natürlich viel einfacher. Außerdem brauchen wir auf diese Weise die vielen Feldnamen nicht zu wissen (die bekanntlich bei der Strukturdefinition stehen, also in den Include-Files).

Ähnlich verfahren wir mit der erforderlichen NewWindow-Struktur. Dort allerdings müssen wir mindestens einen Parameter später einsetzen. Es ist dies die Adresse der zugehörigen Screen-Struktur, die wir erst nach dem eigentlichen Öffnen des Bildschirms erfahren. In dieser Struktur müssen wir eine Reihe von Flags angeben. Wir wählen hier zwei Rückmeldeflags (IDCMP-Flags). Unserem Programm soll also mitgeteilt werden, wenn der Benutzer das Fenster schließt (CLOSEWINDOW) und wenn er es vergrößert bzw. verkleinert (NEWSIZE). Die Fensterart geben wir mit einem anderen Flag an. Unser Fenster besitzt demnach ein Schließ-Gadget (WINDOWCLOSE), ein Gadget zur Größenänderung (WINDOWSIZING), zur Änderung der Fensterüberlagerungstiefe (WINDOWDEPTH) und es kann frei positioniert werden (WINDOWDRAG). Ferner wurde SMART_REFRESH eingeschaltet (Fensterinhalt muß nur bei Vergrößerung/Verkleinerung erneuert werden), ACTIVATE (Fenster ist nach dem Öffnen aktiv), NOCAREREFRESH (keine Refresh-Meldungen erwünscht). Gleichzeitig ist unser Fenster ein sogenanntes Gimmezerozero-Fenster, d.h. innerhalb des Fensters wird noch ein unsichtbares zweites Fenster geöffnet. Es umfaßt den gesamten Fensterbereich, außer die Bereiche, die durch Gadgets etc. eingenommen werden. Grafikausgaben können so relativ zu diesen Fensterausmaßen vorgenommen werden, wodurch wir verhindern, Gadgets etc. zu übermalen.

Dann kann es auch schon losgehen. Nach der Initialisierung der erforderlichen lokalen Strukturen und Variablen versuchen wir die beiden Libraries Intuition-Library und Graphics-Library mit der Exec-Funktion **OpenLibrary( )** zu öffnen (die Exec-Library ist bereits nach dem Programmstart offen, braucht also von uns nicht mehr geöffnet zu werden). Die rückgemeldeten Pointer merken wir uns für später. Ist dieser Pointer allerdings gleich Null, so hat irgend etwas nicht hingehauen (z.B. zu wenig Speicher, die Bibliothek ist nicht verfügbar, weil sie z. B. nicht auf Disk steht und nachgeladen werden muß etc.). In diesem Fall beenden wir unser Programm einfach mit der Funktion **exit( )**, wobei wir natürlich nicht vergessen dürfen, eventuell offene Libraries zu schließen.

Das Öffnen des Bildschirms mit **OpenScreen( )** und des Fensters mit **OpenWindow( )** geht völlig analog vonstatten. Diesen beiden Funktionen müssen Sie lediglich einen Zeiger auf die oben initialisierten Strukturen übergeben (NeuerBildschirm bzw. NeuesFenster). Zurück bekommen wir dann den Zeiger auf die interne Bildschirm- bzw. Fensterstruktur. Beachten Sie, daß wir vor dem Öffnen des Fensters noch schnell die Adresse der Bildschirmstruktur eingesetzt haben.

Damit haben wir es geschafft, ein Fenster in einem neuen Bildschirm zu öffnen. Damit wir auch etwas davon haben, zeichnen wir nun eine kleine Grafik ein. Viele Grafikfunktionen benötigen die Angabe eines Pointers, den wir in der internen Fenster-Struktur finden, den Rasterport. Der Rasterport ist eine Struktur, die – vereinfacht ausgedrückt – dem Betriebssystem angibt, wo es hinzeichnen soll (in welches Fenster etc.). In unserem Programm übernehmen wir diesen Pointer der Einfachheit halber in die Variable »rasterport«.

Nun können die verschiedenen Grafikfunktionen eingesetzt werden. Am Ende unserer Zeichnungen müssen wir nun auf eine Reaktion des Anwenders warten. Dies geschieht mit der Exec-Funktion **Wait( )**, die auf eine Meldung im Message-Port des Fensters wartet (während des Wartens können andere Prozesse und Task arbeiten). Wie das genau funktioniert, würde hier den Rahmen sprengen und sollte von Ihnen nachgelesen werden. Die Art der Meldung finden wir jedenfalls im User-Port des Fensters. Wir übernehmen sie in die Variable »class« und quittieren die Meldung mit **ReplyMsg( ).**

Nun können wir – je nach Art der Meldung – reagieren. Wurde das Fenster vom Benutzer vergrößert oder verkleinert (NEWSIZE), so löschen wir den Fensterinhalt und zeichnen alles neu. Wurde das Schließ-Gadget angeklickt (CLOSEWINDOW), so bedeutet das für uns: Programmende. Wir schließen Fenster, Bildschirm und Libraries (Reihenfolge beachten!) und beenden mit **Exit( )**.

In die Strukturen für neue Bildschirme und Fenster können wir eine Reihe von Flags einsetzen, wie wir gesehen haben. Im folgenden haben wir alle möglichen Flags für diese beiden Strukturen einmal kurz zusammengestellt. Jeder Flag-Name stellt eine Bit-Kombination dar. Wollen Sie zwei oder mehr Flags gleichzeitig setzen, so sind sie mit ODER (in C also: »|«) zu verknüpfen (s. Beispielprogramm). Wie die einzelnen Strukturen aussehen – insbesondere auch die eigentlichen Bildschirm- und Fenster-Strukturen (struct Screen und struct Window), die durch den Aufruf von **OpenScreen()** bzw. **OpenWindow()** initialisiert werden – können Sie den verschiedenen Include-Files auf Ihrer C-Compiler-Diskette entnehmen. Hier also die Tabellen:

a) Die Flags für die Bildschirmstruktur:

ViewModes (Grafikmodi):

NULL	– Niedrig aufgelöste Grafik 320 Punkt/Zeile
HIRES	– Hoch aufgelöste Grafik 640 Punkte/Zeile
LACE	– Einschalten des Interlace-Modus
HAM	– Einschalten des Hold-and-Modify-Modus
EXTRA_HALF_BRITE	– Einschalten des Extra-Half_Brite-Modus
DUALPF	– Dual-Playfield-Modus: Ein Spezialmodus, in dem eine Grafik durch eine andere hindurchscheint
PFBA	– Flag für Dual-Playfield. Es legt fest, welche der zwei Grafiken durch die andere hindurchscheint
SPRITES	– In dem Bildschirm sollen Sprites verwendet werden
VP_HIDE	– Der Bildschirm wird erzeugt, aber nicht auf dem Monitor dargestellt
GENLOCK_VIDEO	– Spezialmodus für ein Genlock-Interface. Ein Videobild scheint durch eine Grafik

ScreenTypes (Bildschirm-Arten):

WBENCHSCREEN	– Workbench-Screen
CUSTOMSCREEN	– Unabhängiger neuer Bildschirm
SHOWTITLE	– Anzeigen der Titelleiste
BEEPING	– Flag wird von Intuition gesetzt, wenn der Bildschirm blinkt
CUSTOMBITMAP	– Es wird eine eigene BitMap verwendet

b) Die Flags für die Fensterstruktur:

IDCMP-Flags (Intuition-Mitteilungsflags):

Die IDCMP-Flags setzen Sie in der Fensterstruktur, wenn Ihr Programm benachrichtigt werden soll, falls ein bestimmtes Ereignis eintritt. Auf ein beliebiges Ereignis warten Sie mit der Funktion **Wait()** (s. Beispielprogramm). Welches Ereignis tatsächlich eingetreten ist, erfahren Sie im Class-Feld der Struktur IntuiMessage (s. ebenfalls Beispielprogramm). Hier wird noch einmal das IDCMP-Flag des verantwortlichen Ereignisses gesetzt. Die einzelnen Informationen zu dem Ereignis finden Sie – je nach Art der Meldung – in den verschiedenen anderen Strukturfeldern wie Code, MouseX, MouseY etc. Hier sind nun die Flags:

MOUSEBUTTON	– Meldung, falls ein Mausknopf betätigt wurde. Welcher Mausknopf betätigt wurde, steht im Code-Feld der IntuiMessage–Struktur (SELECTDOWN, SELECTUP, MENUDOWN, MENUUP)
MOUSEMOVE	– Meldung, falls die Maus bewegt wurde (nur zusammen mit REPORTMOUSE zu setzen)

DELTAMOVE	– wie MOUSEMOVE. Es werden allerdings relative Mauskoordinaten übermittelt
GADGETDOWN	– Meldung, wenn ein Gadget angeklickt wurde
GADGETUP	– Meldung, wenn ein Gadget losgelassen wird
CLOSEWINDOW	– Meldung, wenn das Fenster geschlossen wurde
MENUPICK	– Meldung, wenn Menüpunkt selektiert wurde. Der Menüpunkt steht im Code-Feld der IntuiMessage-Struktur
MENUVERIFY	– Meldung, wenn der Benutzer einen Menüpunkt selektieren möchte (Meldung vor der Selektion)
REQSET	– Meldung, wenn sich der erste Requester im Fenster öffnet
REQCLEAR	– Meldung, wenn sich der letzte Requester im Fenster schließt
REQVERIFY	– Meldung, bevor sich der erste Requester im Fenster öffnet
NEWSIZE	– Meldung, wenn die Fenstergröße verändert wurde
REFRESHWINDOW	– Meldung, wenn Teile des Fensters erneuert werden müssen
SIZEVERIFY	– Mitteilung, wenn der Benutzer ein Fenster vergrößern möchte (Mitteilung bevor er es vergrößert!)
ACTIVEWINDOW	– Meldung, wenn Fenster aktiviert wird
INACTIVEWINDOW	– Meldung, wenn Fenster deaktiviert wird
RAWKEY	– Meldung bei Tastendruck
NEWPREFS	– Meldung, wenn die Preferences geändert wurden
DISKINSERTED	– Meldung, wenn eine Diskette eingeschoben wurde
DISKREMOVED	– Meldung, wenn eine Diskette entnommen wurde
INTUITICKS	– Intuition-Timermeldung (ca. 10mal die Sekunde)

Fenster-Flags:

WINDOWSIZING	– Das Fenster soll ein Vergrößerungs-Gadget besitzen
WINDOWDRAG	– Das Fenster soll verschoben werden können
WINDOWDEPTH	– Das Fenster soll Tiefenwechsel-Gadgets besitzen
WINDOWCLOSE	– Das Fenster soll ein Schließ-Gadget besitzen
SIZEBRIGHT	– Das Größen-Gadget soll am rechten Rand liegen
SIZEBBOTTOM	– Das Größen-Gadget soll am unteren Rand liegen
SMART_REFRESH	– Das Fenster wird nach einer Überlagerung durch andere Fenster automatisch wiederhergestellt
SIMPLE_REFRESH	– Nach einer Überlagerung durch andere Fenster muß das Programm die Wiederherstellung der überlagerten Bildschirmteile selbst vornehmen
SUPER_BITMAP	– Das Fenster ist nur ein Ausschnitt einer großen Grafik (SuperBitmap) und wird automatisch refreshed etc.
BACKDROP	– Hintergrundfenster (liegt stets hinter allen anderen Fenstern)
REPORTMOUSE	– Jede Mausbewegung soll gemeldet werden (s. auch IDCMP-Flags)

GIMMEZEROZERO	– Innerhalb eines Fenster wird ein zweites unsichtbares Fenster eröffnet. Alle Grafikkoordinaten beziehen sich dabei auf das innere Fenster. Vorteil: Die Koordinaten z.B. 0,0 liegen nicht mehr im Rahmenteil des Fensters.
BORDERLESS	– Fenster wird ohne Rahmen dargestellt
ACTIVATE	– Fenster wird aktiviert, sobald es geöffnet ist.
WINDOWACTIVE	– Dieses Flag wird von Intuition gesetzt, wenn das Fenster das aktive Fenster ist
INREQUEST	– Dieses Flag wird von Intuition gesetzt, wenn sich das Fenster im Request-Modus befindet
MENUSTATE	– Dieses Flag wird von Intuition gesetzt, wenn das Fenster mit eingeschalteten Menüs aktiv ist
NOCAREREFRESH	– Keine Meldung, wenn Refresh durchgeführt wird.

KAPITEL 3

Wir fangen klein an: 2-D-Operationen

Bevor wir uns in die räumliche Welt mit allen ihren Höhen und Tiefen stürzen, sollten wir uns zunächst einmal die wesentlichen Grundlagen unter zwei Dimensionen anschauen. Vieles läßt sich dann später nämlich sehr viel einfacher auf die dreidimensionalen Gegebenheiten übertragen.

Manche Dinge sind Ihnen sicher bereits bekannt. Das ist auch gut so. Denn wir wollen ja von bekannten Dingen ausgehend das Unbekannte erforschen. Gleichzeitig werden wir die mathematischen Grundlagen schaffen, auf die wir später nicht mehr verzichten können. Grafik ist Mathematik! Das ist eine Tatsache, die nicht nur Ihnen sicher übel aufstoßen wird. Erinnerungen an dunkle Schulzeit wollen nicht gern angekratzt werden.

Bekommen Sie nun aber keinen Schreck! Es gibt Mittel und Wege, auch öde gelehrte Schulmathematik kurz und vor allem – und das ist der Unterschied zur Schule – verständlich zu beschreiben.

Doch fallen wir nicht gleich mit der Türe ins Haus. Gehen wir das Ganze einmal ruhig an und beginnen mit:

3.1 Die grafischen Grundelemente: Punkt, Linie, Kreis, Ellipse

Bevor wir uns nämlich ins Getümmel werfen, wollen wir uns noch ein wenig mit den Grundelementen jeder Grafik vertraut machen. Ganz zuoberst ist da natürlich der Punkt zu nennen. Da sich der Bildschirm unseres Rechners – was nicht selbstverständlich ist – aus einzelnen Punkten zusammensetzt, stellt der Punkt zwangsläufig die kleinste Grafikeinheit dar. Jede Linie, jeder Kreis, jede andere Figur besteht letztendlich aus vielen kleinen farbigen Punkten.

Das heißt allerdings nicht, daß Punkte die handlichsten Grafikeinheiten sind. Viel effektiver und auch bequemer ist es vielmehr, seine Grafiken aus Linien zusammenzusetzen. Und davon wird überall auch reger Gebrauch gemacht. Linien zu zeichnen geht recht schnell. Sie sind sowohl mathematisch als auch grafisch einfach zu handhaben, kurz, sie sind das Grafikwerkzeug.

Da es bereits einen erheblichen Aufwand bedeutet, Kreise oder gar Ellipsen in akzeptabler Geschwindigkeit zu zeichnen, finden wir die Figuren in fast jedem Befehlssatz eines guten Basic-Interpreters. Selbst einige Betriebssysteme (wie Intuition V1.2) sind in Besitz solcher Algorithmen.

Das folgende Basic-Programm soll Ihnen einige hübsche Beispiele für die Anwendung dieser Grafikfiguren demonstrieren. Zum Thema Ellipsen beachten Sie bitte auch das im Kapitel »Grafikansteuerung in Basic« Gesagte.

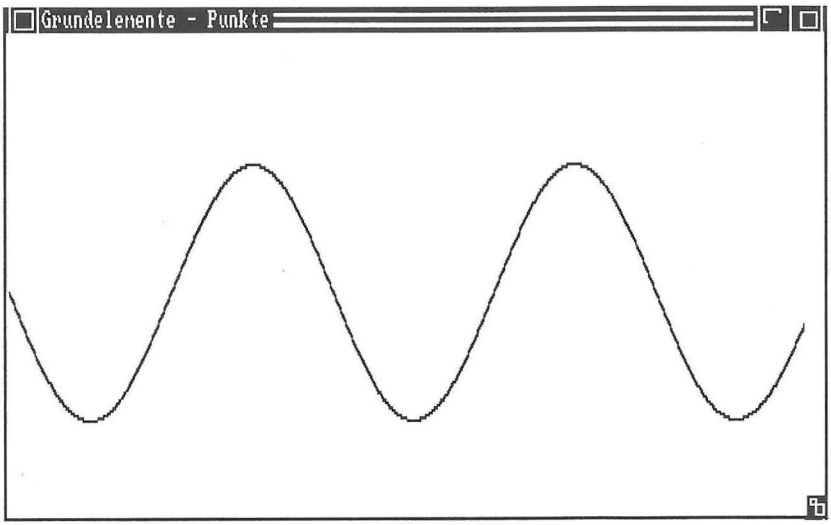


Bild 3.1: Punkte 1

```
' ************************
' **                    **
' **  Punkte, Linien,   **
' **  Kreise, Ellipsen  **
' **                    **
' ************************

' Punkte:

COLOR 2,0
FOR x=0 TO 639
  PSET (x,50*SIN(x/40)+100)
NEXT x

WHILE INKEY$=""
  SLEEP
WEND
```

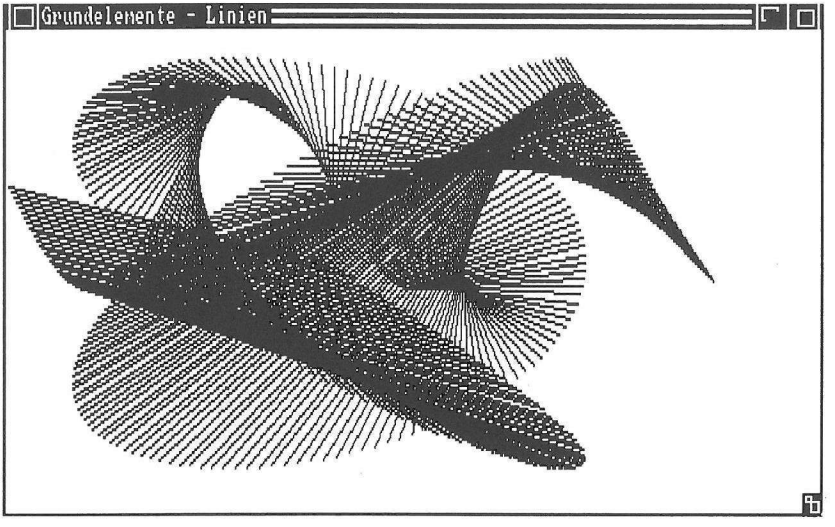


Bild 3.2: Linien

```
' Linien:

COLOR 3,2
CLS
FOR x=0 TO 550 STEP 2
  x1 = x
  y1 = 40*SIN(x/40)+60
  x2 = 200*SIN(x/40)+250
  y2 = 80*SIN(x/50)+90
  LINE (x1,y1)-(x2,y2)
NEXT x

WHILE INKEY$=""
  SLEEP
WEND


' Kreise und Ellipsen:

COLOR 3,0
CLS
```

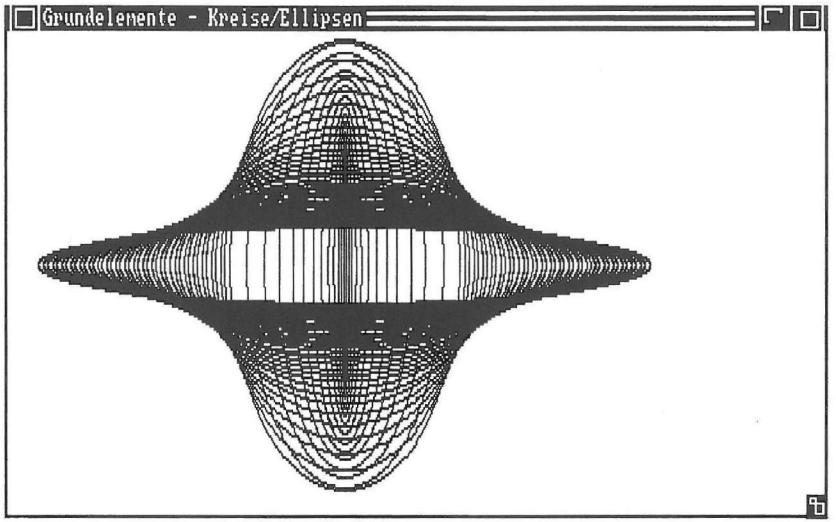


Bild 3.3: Kreise und Ellipsen

```
FOR x=1 TO 40 STEP .7
  CIRCLE (260,90),40+5*x,,,,100/x^2
NEXT x
WHILE INKEY$=""
  SLEEP
WEND


'Und noch einmal Punkte:

COLOR 2,0
CLS
FOR x=1 TO 250
  FOR y=-100 TO 100
    w = ATN(y/x)
    r = SQR(x*x+y*y)
    h = .5 + .5*SIN(2*w+LOG(r)*10)
    IF h>=RND(1) THEN
      PSET (250-x,100+y)
      PSET (249+x,100-y)
```

```
    END IF
  NEXT y
NEXT x

WHILE INKEY$=""
  SLEEP
WEND
```

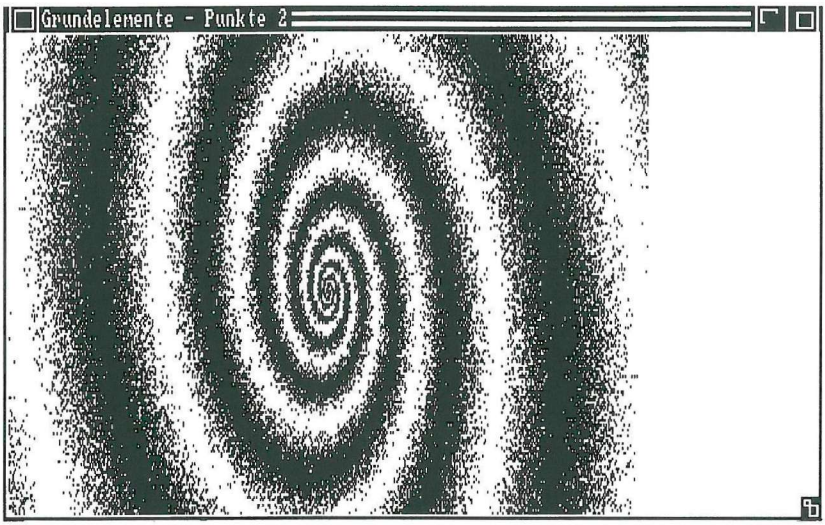


Bild 3.4: Punkte 2

Wir möchten hier dieses Programm nicht lange erläutern, das würde zu weit am Thema vorbei führen. Es soll Sie nur ein wenig einstimmen. Also setzen Sie sich hin, und schauen Sie sich das Ganze einfach an.

Oft – auch in diesem Buch – stehen wir vor der Aufgabe, nicht nur eine Linie zu zeichnen (das können Sie mit Hilfe der entsprechenden Befehle auch), sondern Schnittpunkte mit anderen Linien (oder gar Ebenen) zu berechnen, Linien abzuschneiden o.ä. Hierzu benötigen wir zumeist eine Spur von Mathematik.

Ganz grundlegend dabei ist natürlich die Geradengleichung, die wohl jeder schon einmal irgendwo gesehen hat:

```
a*x + b*y + c = 0                    (Allgemeine Form)
```

oder:

```
y = m*x + n                          (Normierte Form)
```

Bild I:
Ray-Tracing-Scene 1:
Spiegelungen, Reflexionen, Schatten...

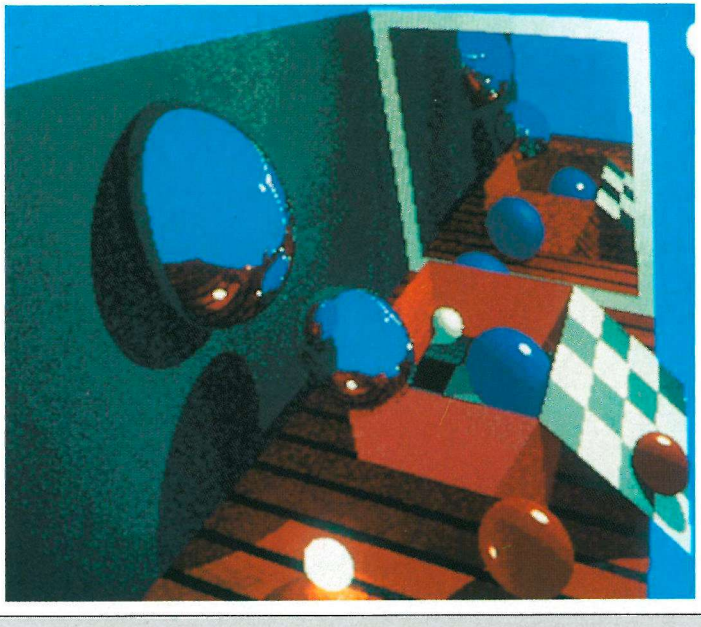


Bild II:
Ray-Tracing:
Lichteindrücke für das Auge

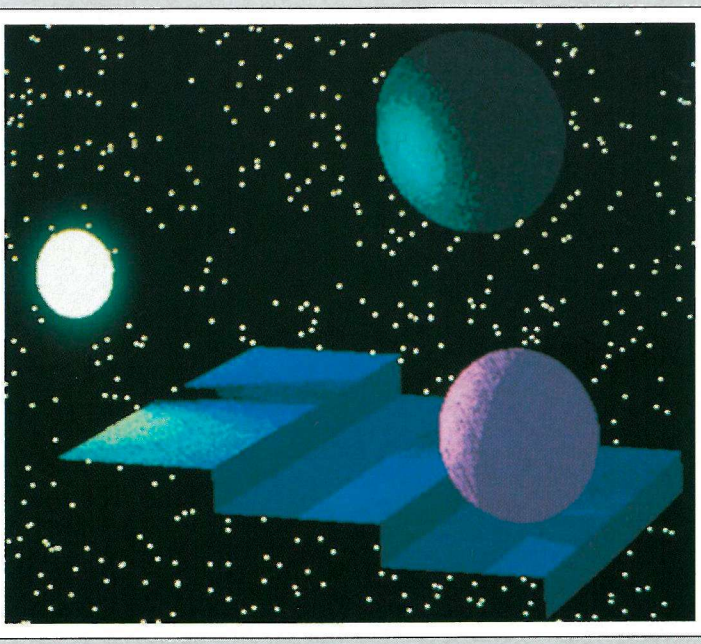


Bild III:
Rotationskörper 1
(Rot)

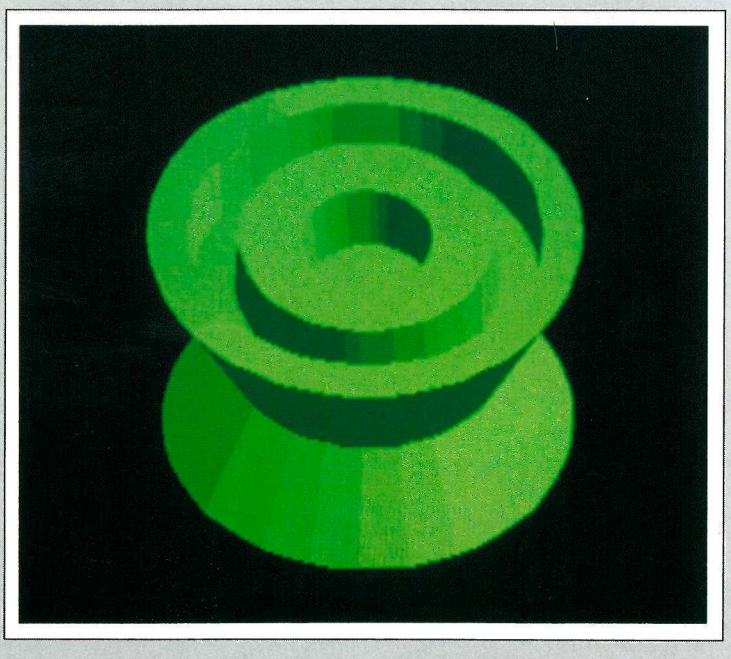


Bild IV:
Rotationskörper 2
(Grün)

Bild V:
Rotationskörper 4
(Blau)

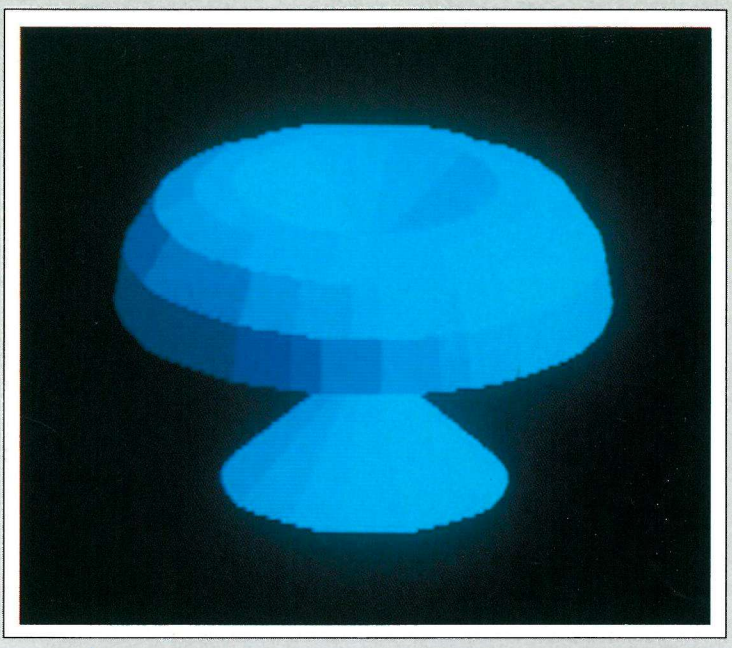
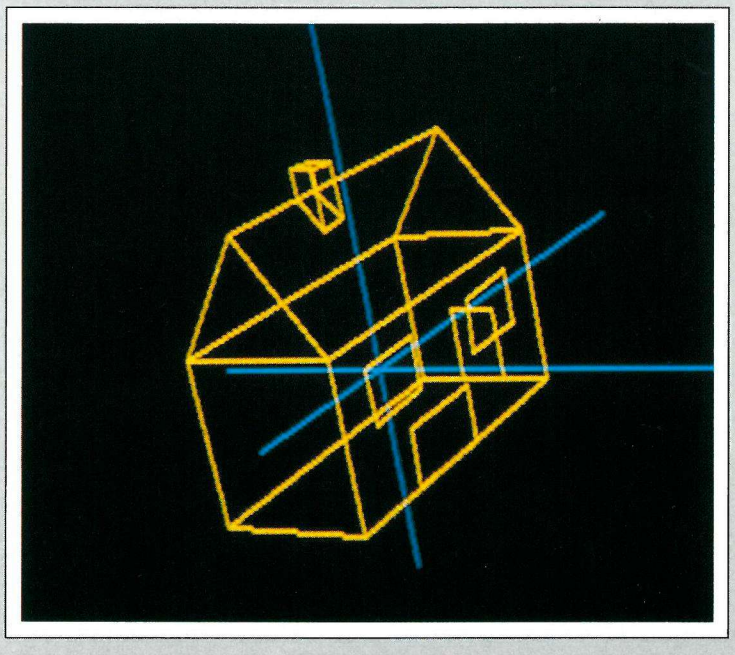


Bild VI:
Zentralprojektion



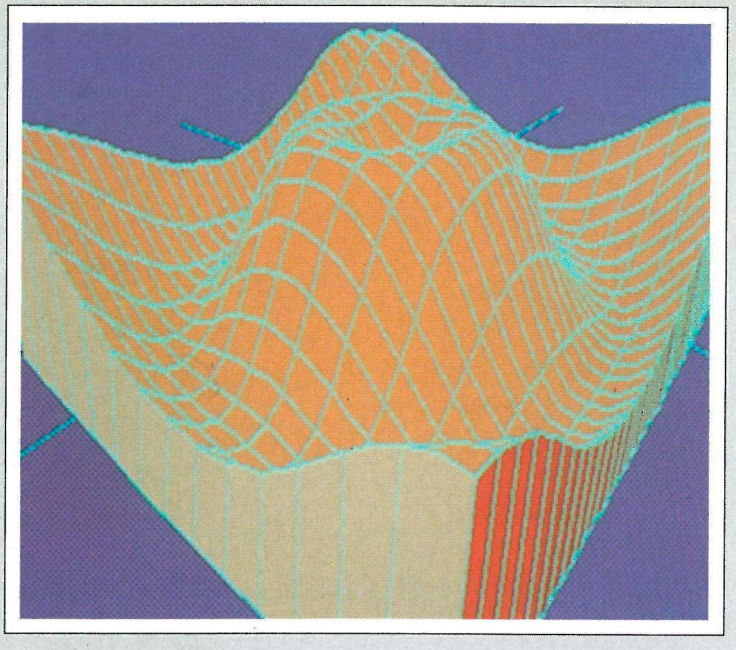


Bild VII: Funktionsplotter 1

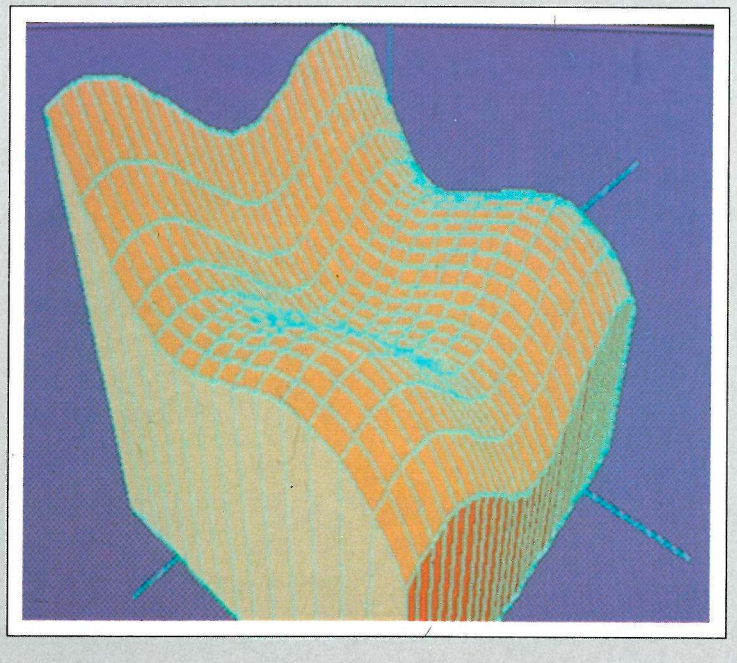


Bild VIII: Funktionsplotter 2

wobei m die Steigung und n den Schnittpunkt mit der y-Achse angeben, es gilt:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Auch diese Form ist möglich:

$$y = m*(x - x_1) + y_1 \qquad \text{(Punkt-Steigungs-Form)}$$

wobei:

x_1, y_1	Ein Punkt auf der Geraden
m	Steigung (s.o.)

Daraus leitet sich die nächste Form ab:

$$\frac{y - y_1}{y_2 - y_1} = \frac{x - x_1}{x_2 - x_1} \qquad \text{(Zweipunktegleichung)}$$

Eine letzte, nicht so alltägliche Form lautet:

$$\frac{x}{a} + \frac{y}{b} = 1 \qquad \text{(Achsenabschnittsform)}$$

Diese Gerade schneidet die x-Achse im Punkt P(a,0) und die y-Achse im Punkt Q(0,b).

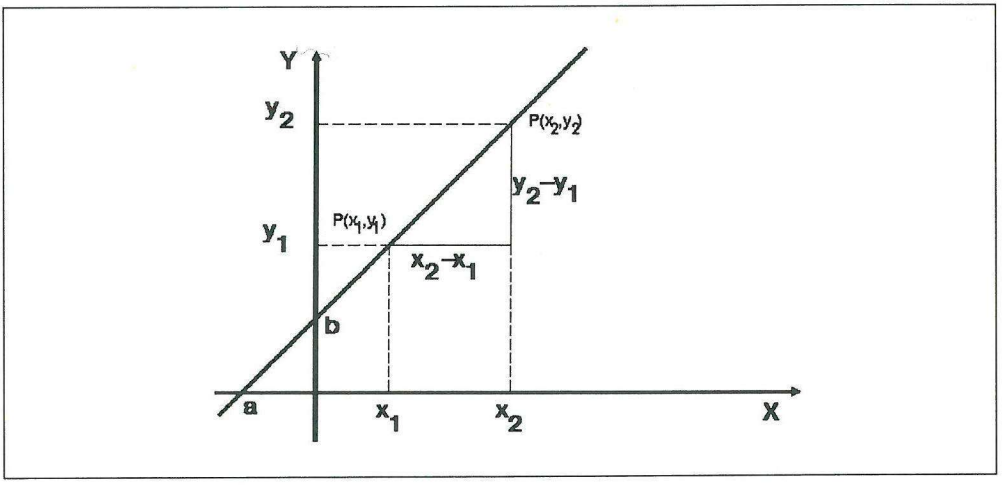


Bild 3.5: Die Steigung einer Geraden

Natürlich gibt es noch einige weitere Formen, die uns an dieser Stelle noch nicht interessieren sollen. Wir werden einigen dieser Gleichungen noch öfter begegnen. Sie sollten hier nur einmal gerafft vorgestellt werden. An den entsprechenden Stellen wird die jeweils verwendete Form natürlich noch einmal genauer erläutert.

Auch für Kreise und Ellipsen (der Kreis ist ja lediglich ein Sonderfall der Ellipse) gibt es mathematische Beschreibungen, die vielleicht schon etwas unbekannter sind:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \qquad \text{(Normalform)}$$

wobei:

a Radius der Ellipse in x-Richtung
b Radius der Ellipse in y-Richtung

für den Kreis gilt a = b = r und damit:

$$\frac{x^2 + y^2}{r^2} = 1$$

Eine andere, viel verwendete Form lautet:

$$x = a * \cos(w)$$
$$y = b * \sin(w) \qquad \text{(Parameterform)}$$

wobei:

a Radius der Ellipse in x-Richtung
b Radius der Ellipse in y-Richtung
w Winkel zwischen der Verbindung Ellipsenpunkt-Mittelpunkt und der x-Achse

Für den Kreis gilt natürlich wieder: a = b

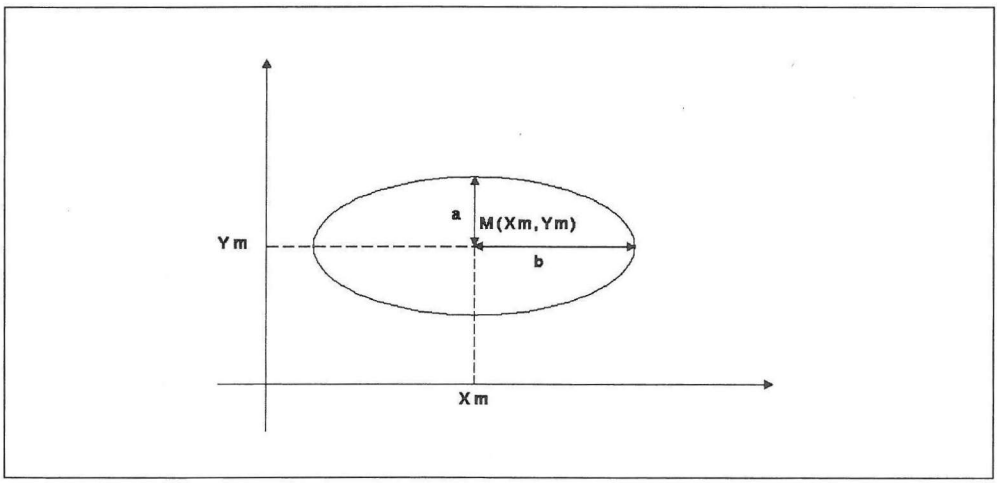


Bild 3.6: Ellipsengleichung

Nach diesem ganz kurzen Exkurs stürzen wir uns direkt in unser Thema, das uns in diesem Kapitel als erstes beschäftigen soll. Möchten Sie noch ein wenig mehr und detaillierter über die gerade vorgestellten Dinge erfahren, dann schlagen Sie ruhig einmal im Anhang nach. Dort werden sie noch einmal ausführlich erläutert.

3.2 Transformationen in 2-D

Auf dem Weg in räumliche Welten spielen sogenannte Transformationen eine wichtige Rolle. Unter Transformationen verstehen wir im weitesten Sinne »Umwandlungen« einer bestehenden Grafik. Angenommen, wir haben eine kleine Grafik (ein Haus o.ä.) auf dem Bildschirm gezeichnet. Wir kennen die Koordinaten der verschiedenen Eckpunkte und die Punkte, die miteinander durch Linien verbunden werden sollen. Wir wünschen uns nun einfache mathematische Operationen, die es uns erlauben, dieses Haus per Programm und ohne die einmal festgelegten Koordinaten mit der Hand verändern zu müssen, auf dem Bildschirm zu verschieben, es zu vergrößern oder zu verkleinern. Wir wollen es drehen und spiegeln, kurz: transformieren.

In vielen Anwendungen spielen Transformationen eine wesentliche Rolle. Architekten werden sehr viel Wert darauf legen, ihre Werke auch einmal aus einem anderen Winkel zu betrachten oder Details zu vergrößern. Trickfilmzeichner suchen sicher nach einer Möglichkeit, Figuren zu verschieben oder zu drehen usw.

Für diese Zwecke ist es sinnvoll, ein Bild nicht einfach Punkt für Punkt auf den Bildschirm zu zeichnen, wie wir das auf einfache Weise mit Malprogrammen wie »Deluxe Paint« o.ä. realisieren können. Vielmehr setzt sich für uns ein Bild aus einer Reihe von Linien zusammen, deren Endpunkte uns durch ihre Koordinaten bekannt sind. Wir haben die Koordinaten für unsere ganze kleine »Bildwelt« gespeichert und können sie so jederzeit durch mathematische Manipulationen, eben Transformationen verändern.

In diesem Kapitel geht es nun darum, solche Transformationen in der Ebene durchzuführen. Dabei werden wir die zugehörigen mathematischen Verfahren und den Umgang mit ihnen kennenlernen. Sie begleiten uns dann später durch einen großen Teil des Buches.

3.2.1 Mathematische Grundlagen – leicht gemacht

Bevor wir allerdings die verschiedenen Transformationen diskutieren wollen, werden wir ein paar mathematische Grundsteine legen müssen. Diese sollten Sie in jedem Falle verstehen und zumindest einigermaßen beherrschen, da sie uns durch das gesamte Buch begleiten. Ohne sie ist ein durchdringendes Verständnis der einzelnen Operationen und grafischen Manipulationen kaum möglich. Besonderes Augenmerk sollten Sie auf die Matrizenmultiplikation legen. Sie ist für uns mit eine der wichtigsten Operationen.

Ganz konkret sollen uns an dieser Stelle sogenannte Matrizen beschäftigen. Matrizenrechnung ist gar nicht so kompliziert, wie man Leute oft stöhnen hört. Sie ist aber von enormer Bedeutung für jede Art von Transformationen, da sie ein einfaches Mittel zu ihrer Realisierung darstellt.

Wenn Sie nach der Lektüre der folgenden Seiten noch kein Verständnis dafür haben, was uns diese neue Errungenschaft eigentlich bringt, so ist das nur natürlich. Warten Sie in diesem Falle einfach das Ende dieses 2-D-Kapitels ab.
Unter einer Matrix stellen wir uns eine rechteckige Anordnung von verschiedenen Zahlen vor:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & \dots & a_{3n} \\ a_{41} & a_{42} & . & . & . & \dots & . \\ . & . & . & . & . & \dots & . \\ . & . & . & . & . & \dots & . \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & a_{m5} & \dots & a_{mn} \end{pmatrix}$$

Jede Zahl a_{ik} (z.B. a_{11} oder a_{34}) stellt ein Element der Matrix dar. (Die Indizes an den Elementen oben sollen nur zur Unterscheidung dienen, die Namen für Matrizen sind vereinbarungsgemäß immer Großbuchstaben, hier A.) Die Matrix setzt sich zusammen aus Zeilen (waagerechte Reihe von Elementen) und Spalten (senkrechte Reihe von Elementen). Sie ist, so sagt man, vom Typ (m,n), da sie m Zeilen und n Spalten besitzt. Man spricht auch von einer (m,n)-Matrix. Die Elemente $a_{11}, a_{12}, a_{13}, \dots$ (oder kurz: a_{1k}) stellen die erste Zeile dar, alle Elemente a_{2k} die zweite Zeile etc. Analog verhält es sich mit den Spalten. Ein Element a_{ik} steht also in der Zeile i und in der Reihe k. Wir haben es hier sozusagen mit einem zweidimensionalen Array zu tun.

Die Anzahl von Elementen in einer Zeile kann (muß aber nicht) gleich der Anzahl der Elemente in einer Spalte sein. In diesem Fall sprechen wir von einer n-reihigen quadratischen Matrix, z.B.:

$$A = \begin{pmatrix} 1 & 5 & -6 \\ -7 & 13 & 8 \\ -1 & 2 & 22 \end{pmatrix}$$

Diese Matrix ist dreireihig und quadratisch. Quadratische Matrizen sind deshalb besonders hervorzuheben, da sie definitionsgemäß einer Zahl zugeordnet sind, der sogenannten Determinante. Determinanten werden beispielsweise verwendet, wenn Gleichungssysteme mit mehreren Unbekannten gelöst werden müssen.

Ein weiterer Sonderfall von Matrizen sind solche, die nur eine Zeile oder nur eine Spalte besitzen. Sie sind daher vom Typ (1,n) bzw. (m,1) (s.o.), z.B.:

$$A = (1\ 5\ 3\ -4)$$

Hier haben wir es mit einer Matrix zu tun, die nur eine einzige Zeile besitzt (hier mit 4 »Spalten«). Sie wird auch »Zeilenvektor« genannt. Der andere Fall ist ein sogenannter »Spaltenvektor«, z.B.:

$$A = \begin{pmatrix} -4 \\ 9 \\ 6 \end{pmatrix}$$

Man kann mit Matrizen nun auch rechnen. Wir können Additionen, Subtraktionen und Multiplikationen mit einer einfachen Zahl wie auch mit einer anderen Matrix durchführen. Wir

wollen uns hier aber keinen unnötigen Ballast aufladen und richten unseren Blick auf die Operationen, die wir dringend benötigen: »Die Matrix-Multiplikation zweier verketteter Matrizen« (für andere Rechenoperationen schauen Sie bitte im Anhang nach).

Hinter dem Begriff der »verketteten Matrizen« verbirgt sich eigentlich nur folgendes:

Man kann zwar zwei Matrizen miteinander multiplizieren, sie müssen dafür aber bestimmte Bedingungen erfüllen, sie müssen nämlich verkettet sein.

Zu deutsch: Wir können zwei Matrizen nur dann miteinander multiplizieren, wenn die erste genausoviele Spalten besitzt wie die andere Zeilen. Ein Beispiel einer erlaubten Multiplikation wäre dann:

$$A * B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

A besitzt drei Spalten, B drei Zeilen und damit wäre die Bedingung erfüllt. Wir sehen aber auch, daß die Multiplikation B*A nicht erlaubt ist. Auch wenn B*A erlaubt wäre, hieße das nicht, daß das Ergebnis das gleiche wäre wie A*B. Die Reihenfolge der Faktoren ist also stets wichtig! Beachten Sie, daß eine Matrixmultiplikation zwischen zwei quadratischen Matrizen (s.o.) immer erlaubt ist. Wir werden nämlich später vorzugsweise mit quadratischen Matrizen zu tun haben.

Das Ergebnis einer Matrixmultiplikation ist wieder eine Matrix. Sie besitzt die gleiche Anzahl von Zeilen wie die erste (A) und die gleiche Anzahl von Spalten wie die zweite Matrix (B). Das Ergebnis der obigen Multiplikation ((3,3)-Matrix mal (3,2)-Matrix) wäre also eine Matrix vom Typ (3,2) (3 Zeilen, 2 Spalten).

Die Berechnung dieser Ergebnismatrix C ist ein wenig kompliziert. Für die kundigen Leser schreiben wir die Berechnung eines einzigen Elementes in mathematischer Kurzschreibweise:

$$c_{ik} = \sum_{j=1}^{n} a_{ij} * b_{jk} = a_{i1} * b_{1k} + a_{i2} * b_{2k} + \ldots + a_{in} * b_{nk}$$

was soviel heißt wie: Das Element c_{ik} berechnet sich aus der Summe aller $a_{ij}*b_{jk}$ für $j = 1$ bis $j = n$.

wobei:

c_{ik}	ein Element der Ergebnismatrix C
a_{ij}	ein Element der Matrix A
b_{jk}	ein Element der Matrix B
n	Anzahl der Spalten von A, also auch: Anzahl der Zeilen von B

Damit können natürlich nur die wenigsten etwas anfangen, deshalb wollen wir das an einem Beispiel klar machen:

Wir berechnen, so sagt die Formel, ein Element c_{ik} der Ergebnismatrix, indem wir alle Elemente der i-ten Zeile(!) von A mit den jeweils korrespondierenden Elementen der k-ten Spalte(!) von B multiplizieren und von den Ergebnissen die Summe bilden (also addieren):

$$
\begin{aligned}
C &= A * B \\
&= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix} \\
&= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \\
&= \begin{pmatrix} 1*1 + 2*3 + 3*5 & 1*2 + 2*4 + 3*6 \\ 4*1 + 5*3 + 6*5 & 4*2 + 5*4 + 6*6 \\ 7*1 + 8*3 + 9*5 & 7*2 + 8*4 + 9*6 \end{pmatrix} \\
&= \begin{pmatrix} 22 & 28 \\ 49 & 64 \\ 76 & 100 \end{pmatrix} = \begin{pmatrix} c_{11} & c_{21} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{pmatrix}
\end{aligned}
$$

Das sieht ziemlich gewaltig aus, läßt einen Computer aber natürlich völlig kalt. Beachten Sie, daß in der vorletzten Zeile der Ausdruck »1*1 + 2*3 + 3*5« beispielsweise ein einziges Element darstellt. Versuchen Sie das Beispiel einmal nachzuvollziehen. Etwa so: Das oberste linke Element der Ergebnismatrix C, also das Element c11, wird errechnet durch:

$$c_{11} = a_{11}*b_{11} + a_{12}*b_{21} + a_{13}*b_{31}$$

Das Element c_{12} wird berechnet durch... usw.

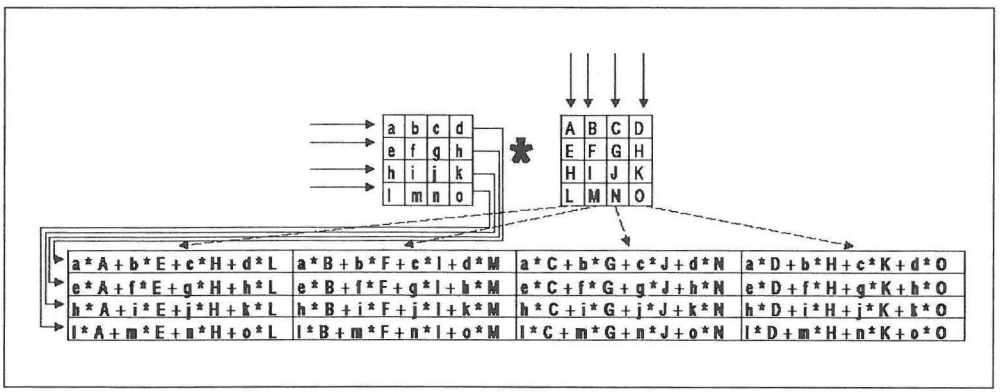


Bild 3.7: Schema der Matrixmultiplikation

Testen Sie Ihr Verständnis doch einmal an diesen beiden Übungsaufgaben:

Berechnen Sie die Ergebnismatrix C der Matrixmultiplikationen:

$$C = A*B = \begin{pmatrix} 6 & 8 \\ 3 & 9 \end{pmatrix} * \begin{pmatrix} 1 & -5 \\ 3 & 7 \end{pmatrix}$$

$$C = A*B = \begin{pmatrix} 3 & 5 & 2 & 8 \\ 7 & 0 & 1 & 1 \\ 5 & 5 & 3 & 2 \\ 1 & 2 & 0 & 5 \\ 4 & 4 & 4 & 4 \end{pmatrix} * \begin{pmatrix} 1 & 1 \\ 6 & -6 \\ 3 & 0 \\ 4 & 10 \end{pmatrix}$$

Einige Rechenregeln sollten wir uns noch einmal anschauen. Multiplizieren wir eine Reihe von Matrizen, so ist es gleichgültig, welche Multiplikationen wir zuerst durchführen, solange wir die Reihenfolge nicht verändern, mathematisch ausgedrückt:

$A*(B*C) = (A*B)*C$

Falsch wäre allerdings: $A*(B*C) = (A*C)*B$

Wie wir oben bereits festgehalten haben, gilt nicht – und das ist sehr wichtig: $A*B = B*A$

Es existieren eine Reihe von besonderen Matrizen, mit denen man eine andere Matrix multiplizieren kann, ohne daß sich etwas ändert. Es sind dies quadratische Matrizen, in denen alle Elemente gleich Null sind. Lediglich die Elemente der sogenannten Hauptdiagonale müssen den Wert Eins besitzen, man nennt sie auch quadratische Einheitsmatrizen E:

(1) oder $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ oder $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Es gilt:

$A*E = A$

Wie wir die Matrizen in der Computer-Grafik z.B. für Transformationen einsetzen (dort wird alles meist viel einfacher als es hier erscheint), das erfahren wir jetzt:

3.2.2 2-D-Vergrößerungen/Verkleinerungen

Es gibt natürlich neben den Matrizen noch weitere Möglichkeiten, die verschiedenen grafischen Transformationen mathematisch zu beschreiben. Es hat sich jedoch herausgestellt, daß keine so anschaulich und einfach zu handhaben sind, wie die nun folgenden Ableitungen.

Wie gesagt, geht es darum, die Koordinaten der einzelnen Punkte unserer Welt nachträglich mathematisch zu manipulieren (= zu transformieren). Damit wir aber mit Punkten und Matrizen nicht wie mit Äpfeln und Birnen rechnen, ist es notwendig, jeden Punkt so zu beschreiben,

daß er ins Matrizen-Schema paßt. Hierzu nehmen wir die beiden x- und y-Koordinaten eines Punktes und sehen sie quasi als (1,2)-Matrix an. Angenommen, ein Punkt besitzt die Koordinaten x und y, dann lautet seine Matrix-Schreibweise:

$$P(x,y) = (x\ y)$$

Dies ist natürlich eine rein willkürliche Definition, die nicht bewiesen werden muß, uns aber sehr hilfreich sein wird.

Wollen wir einen Punkt nun transformieren, so multiplizieren wir ihn mit einer sogenannten Transformationsmatrix. Haben wir diese Matrix korrekt gewählt, dann sollte die Ergebnismatrix die Koordinaten des transformierten Punktes angeben. Die Ergebnismatrix muß also wieder vom Typ (1,2) sein (und damit die Koordinaten enthalten).

Wir suchen also eine Transformationsmatrix T, die – multipliziert mit dem Ausgangspunkt P – den transformierten Punkt P' (bzw. dessen Matrix) bestimmt:

$$P' = P * T$$

Wenn wir nun alle Punkte unseres Gebildes mit T multiplizieren, dann haben wir unser ganzes Bild transformiert. Wie sieht es dann aber aus? Nun, das hängt natürlich von den Elementen der Matrix T ab. Nehmen wir an, T ist die Einheitsmatrix (s. Mathematische Grundlagen):

$$T = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

In diesem Falle bleibt das gesamte Bild so erhalten wie es ist, denn in unserem obigen Beispiel ist P gleich P'.

Angenommen, wir verwenden aber die folgende Matrix:

$$T_1 = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$$

Das Ergebnis:

$$\begin{aligned} P' &= P * T_1 \\ &= (x\ y) * \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} \\ &= (x*2 + y*0 \qquad x*0 + y*1) \\ &= (2*x\ \ y) \\ &= (x'\ y') \end{aligned}$$

Die letzten beiden Zeilen in uns vertrauter Parameterschreibweise:

$$\begin{aligned} x' &= 2*x \\ y' &= y \end{aligned}$$

Wir sehen: Jede neue x-Koordinate wird durch die Transformation doppelt so groß sein wie die ursprüngliche. Das bedeutet aber eine Verzerrung (oder Vergrößerung) in x-Richtung.

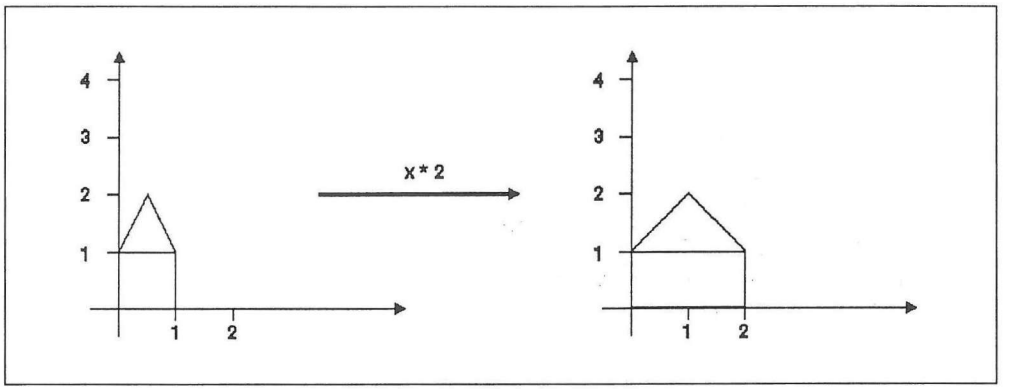


Bild 3.8: Verdoppelung aller x-Koordinaten

Entsprechend finden wir auf der Suche nach einer Matrix, die alle y-Koordinaten verdoppelt und damit das Bild in y-Richtung verzerrt:

$$T_2 = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$$

Wenn wir nun ein Bild sowohl in x- als auch in y-Richtung vergrößern (verkleinern) möchten, dann realisieren wir das in folgender Formel:

$$P' = (P * T_1) * T_2$$

Erst vergrößern wir es also in x-, dann in y-Richtung. Da wir nach den Rechengesetzen für Matrizen diese Formel auch

$$P' = P * (T_1 * T_2)$$

schreiben können, läßt sich sehr einfach eine Matrix $T_3 = T_1 * T_2$ finden, die beide Transformationen auf einmal ausführt:

$$\begin{aligned} T_3 &= T_1 * T_2 \\ &= \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix} \\ &= \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix} \end{aligned}$$

Wenn wir das Ganze verallgemeinern, dann erhalten wir die folgende Transformationsmatrix S für Vergrößerung/Verkleinerung (= Skalierung):

$$S(S_x, S_y) = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix}$$

Dabei bedeuten:
S_x – Skalierungsfaktor in x-Richtung
S_y – Skalierungsfaktor in y-Richtung

Je nach Wert für S_x oder S_y verändert sich das Bild in folgender Weise:

$S_{x/y} > 1$ – Vergrößerung
$S_{x/y} = 1$ – Keine Veränderung
$0 < S_{x/y} < 1$ – Verkleinerung
$S_{x/y} < 0$ – gleichzeitig Spiegelung um x/y-Achse

Werte kleiner oder gleich Null für die Skalierungsfaktoren sollten vermieden werden, da es sich dabei dann nicht mehr um eine einfache Vergrößerung bzw. Verkleinerung handelt.

Mit dieser einfachen Matrix haben wir unser Problem gelöst. Wir können mehrere Skalierungen, die hintereinander ausgeführt werden sollen, je nach Anwendung in einer Matrix zusammenfassen oder auch durch mehrere Matrixmultiplikationen realisieren. Im Rechner führen wir eine Skalierung eines Punktes P(x,y) mit der Transformationsmatrix S nach folgendem Schema aus:

$$
\begin{aligned}
P' &= P(x,y) * S(S_x,S_y) \\
&= (x\ y) * \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix} \\
&= (S_x*x\ \ S_y*y) \\
&= (x'\ y')
\end{aligned}
$$

Diese Matrixschreibweise unseres resultierenden Punktes P' lösen wir dann wieder auf in die Koordinatenschreibweise und erhalten:

$$
\begin{aligned}
x' &= S_x*x \\
y' &= S_y*y
\end{aligned}
$$

Das sind also die beiden Formeln, die der Computer zu berechnen hat. Der scheinbar komplizierte Umweg über Matrizen wird uns später allerdings sehr von Nutzen sein.

Das folgende kleine Basic-Programm soll Ihnen die Anwendung der Skalierungs-Transformation ein wenig näher bringen:

```
'***************************
'**                       **
'**  2-D-Transformation:  **
'**      Skalierung       **
'**                       **
'***************************

' Punkte und Linien einlesen:
READ anzp%, anzl%
DIM SHARED x.punkte(anzp%-1), y.punkte(anzp%-1)
DIM SHARED s.linien%(anzl%-1), e.linien%(anzl%-1)
```

```
'Skalierungsfaktoren:
sx = 1
sy = 1
Bildposition:
mx% = MOUSE(3)
my% = MOUSE(4)

'Punkte einlesen:
FOR i=0 TO anzp%-1
  READ x.punkte(i), y.punkte(i)
  'PRINT i": "x.punkte(i), y.punkte(i)
NEXT i
'Linien (Punktverbindungen) einlesen:
FOR i=0 TO anzl%-1
  READ s.linien%(i), e.linien%(i)
NEXT i


' Hauptschleife:
WHILE c<>27        'bis ESC

  CLS              'Bildschirm löschen

  ' Zeichenschleife:
  FOR i=0 TO anzl%-1
    x1% = x.punkte(s.linien%(i))
    y1% = y.punkte(s.linien%(i))
    x2% = x.punkte(e.linien%(i))
    y2% = y.punkte(e.linien%(i))
    LINE (mx%+x1%,my%-y1%)-(mx%+x2%,my%-y2%)
  NEXT i

  ' Auf Tastendruck warten:
  c$=""
  m%=0
  WHILE c$="" AND m%=0
    SLEEP          'Warten und Multitasking
                   'ermöglichen
    c$=INKEY$      'evt. Taste holen
    m%=MOUSE(0)    'oder Mausstatus
  WEND

  sx = 1
  sy = 1
```

```
  IF m%<>0 THEN    'Maustaste betätigt?
    mx% = MOUSE(1)  'ja -> neue Position
    my% = MOUSE(2)
  ELSE
    c = ASC(c$)
    IF c=28 THEN    'Cursor auf?
      sy = 1.2      'ja -> y-Vergrößerung
    END IF
    IF c=29 THEN    'Cursor ab?
      sy = .8       'ja -> y-Verkleinerung
    END IF
    IF c=30 THEN    'Cursor links?
      sx = 1.2      'ja -> x-Vergrößerung
    END IF
    IF c=31 THEN    'Cursor rechts?
      sx = .8       'ja -> x-Verkleinerung
    END IF
  END IF

  'Gesamtes Bild transformieren:
  CALL transformiere.bild(sx,sy)

WEND

END


' Alle Koordinaten des Bildes transformieren:
SUB transformiere.bild(sx,sy) STATIC
  SHARED i, anzp%
  FOR i=0 TO anzp%-1
    CALL s.transform(sx,sy)
  NEXT i
END SUB

' Matrixmultiplikation P2=P1*S:
SUB s.transform(sx,sy) STATIC
  SHARED i
  x.punkte(i) = sx*x.punkte(i)
  y.punkte(i) = sy*y.punkte(i)
END SUB
```

```
' Bilddaten:
' Anzahl der Punkte / Anzahl der Linien:
DATA 33
DATA 30

' Punktkoordinaten:
DATA  0, 3,   0, 7,   2, 8,   4, 8,   4,12
DATA  6,13,  20,13,  22,12,  22, 8,  24, 8
DATA 26, 7,  24,12,  26, 3,  24, 3,  24, 0
DATA 20, 0,  20, 3,   6, 3,   6, 0,   2, 0
DATA  2, 3,   6, 9,   6,12,  20,12,  20, 9
DATA 22, 5,  20, 6,  22, 7,  24, 6,   4, 5
DATA  2, 6,   4, 7,   6, 6

' Linien (Punktverbindungen):
DATA  0, 1,   1, 2,   2, 9,   9,10,   9,11
DATA 10,12,  12, 0,   3, 4,   4, 5,   5, 6
DATA  6, 7,   7, 8,  21,22,  22,23,  23,24
DATA 24,21,  13,14,  14,15,  15,16,  17,18
DATA 18,19,  19,20,  25,26,  26,27,  27,28
DATA 28,25,  29,30,  30,31,  31,32,  32,29
```

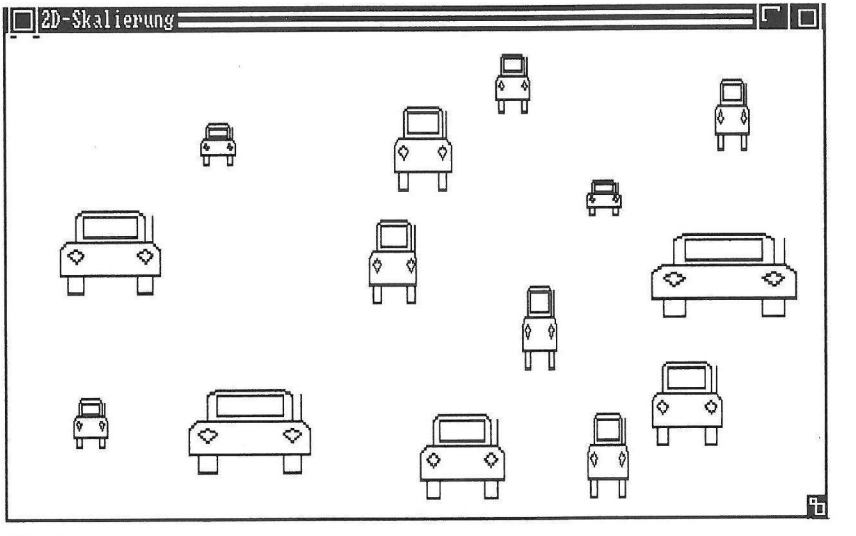


Bild 3.9: Skalierung

In diesem Programm werden Ihnen bereits die Grundzüge einer grafischen Datenverwaltung vorgestellt, die wir ausführlich aber erst im nächsten Kapitel bei der ersten Besprechung von dreidimensionalen »Welten« diskutieren wollen. Dieses Programm nun zeichnet ein kleines Gebilde auf den Bildschirm. Mit Hilfe der Cursortasten sind Sie nun in der Lage, das Bild in x- bzw. y-Richtung zu vergrößern und zu verkleinern. Dabei sind folgende Funktionen erreichbar:

Cursor auf	Vergrößerung in y-Richtung
Cursor ab	Verkleinerung in y-Richtung
Cursor rechts	Vergrößerung in x-Richtung
Cursor links	Verkleinerung in x-Richtung
Esc	Programm beenden
Maustaste	Bildobjekt gemäß Mausposition positionieren

Die Definition des Objektes finden Sie in den DATA-Zeilen am Ende des Programmes. Hier sind die x- und y-Koordinaten jedes einzelnen Eckpunktes festgehalten. Im Anschluß daran finden Sie die Nummern derjenigen Punkte, die jeweils eine Linie beschreiben. Die Daten werden in Arrays eingeladen und können so jederzeit manipuliert werden.

Für unser Thema sind dabei die Sub-Routinen **transformiere.bild( )** und **s.transform( )** wichtig. Letztere führt die Matrixmultiplikation eines einzelnen Punktes mit der Skalierungsmatrix S durch (mit den Elementen s_x und s_y). Der Parameter »i« gibt dabei die Nummer des jeweiligen Punktes im Punktearray an. Die andere der beiden Routinen transformiert alle Punkte des Bildes, indem sie **s.transform( )** für jeden einzelnen Punkt aufruft.

Die Matrix-Elemente S_x und S_y stammen aus dem Hauptprogramm und werden je nach Ihrer Cursor-Eingabe gesetzt. Der Rest des Programmes ist hauptsächlich schmückendes oder notwendiges Beiwerk, das Sie persönlich jederzeit verändern oder gar erweitern können. Das ist gleichzeitig der beste Weg, sich mit dem Stoff dieses Programmes vertraut zu machen.

3.2.3 Spiegelungen

Auch Spiegelungen um eine bestimmte Achse sind recht einfach durch Matrixmultiplikationen realisierbar. In diesem Fall ist es sogar recht einfach, eine Transformationsmatrix zu finden, da wir für Spiegelungen folgende Parametergleichungen verwenden:

Spiegelung um die x-Achse:

$$x' = x$$
$$y' = -y$$

Spiegelung um die y-Achse:

$$x' = -x$$
$$y' = y$$

Das erinnert uns an die Skalierung und schon haben wir die passenden Matrizen:

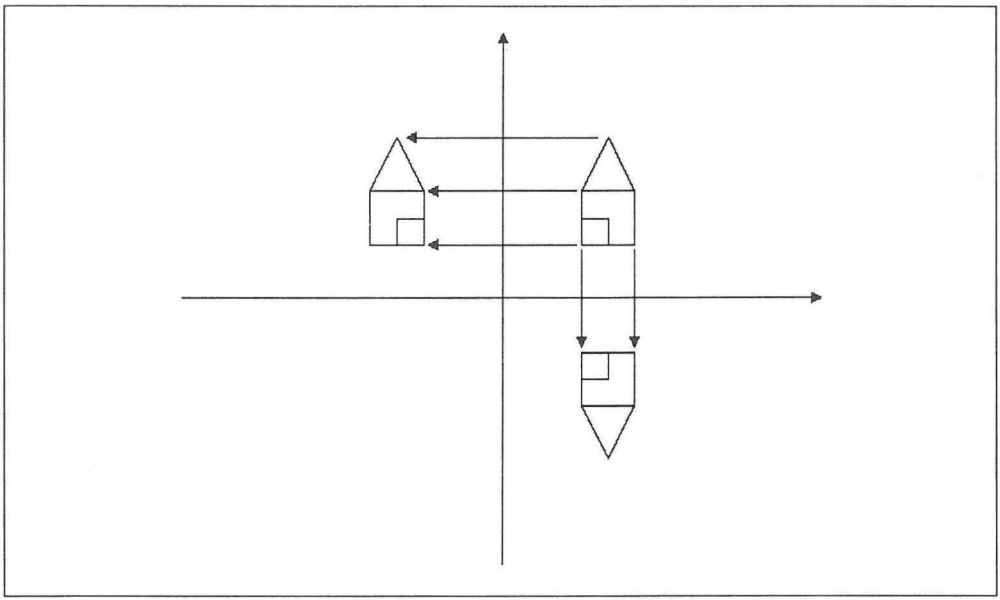


Bild 3.10: Spiegelungen

Spiegelung um die x-Achse:

$$M_x = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Spiegelung um die y-Achse:

$$M_y = \begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$

Wollen wir um beide Achsen gleichzeitig spiegeln, was einer Punktspiegelung gleichkäme, dann brauchen wir auch hier nur eine kleine Matrixmultiplikation durchzuführen (rechnen Sie ruhig nach):

$$M_{xy} = M_x * M_y = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$$

Da es sich hierbei um das gleiche Prinzip handelt, wie bei der Skalierung, möchten wir Sie auch nicht länger langweilen und machen gleich weiter beim nächsten Thema. Bauen Sie die Spiegelung doch einfach in das obige Programm ein (z.B. Spiegelung auf Tastendruck etc.).

3.2.4 Drehungen (Rotationen) in 2-D

Jetzt wird es wieder hochinteressant. Es geht nämlich um Drehungen (Rotation) einzelner Punkte um den Koordinatenursprung. Oft sollen nämlich einzelne Objekte oder ganze Bilder

gedreht werden, um sie von einem anderen Winkel aus zu betrachten etc. Die Grundlage zu einer solchen Drehung um einen beliebigen Punkt stellt die Drehung eines Punktes um den Koordinaten-Nullpunkt dar, die wir uns nun etwas genauer anschauen werden.

Uns interessiert nun natürlich eine Matrix, mit der wir eine solche Drehung durchführen können. Um diese Matrix zu finden, bedarf es zunächst aber der Ableitung der entsprechenden Parametergleichungen. Betrachten Sie hierzu bitte die folgende Abbildung.

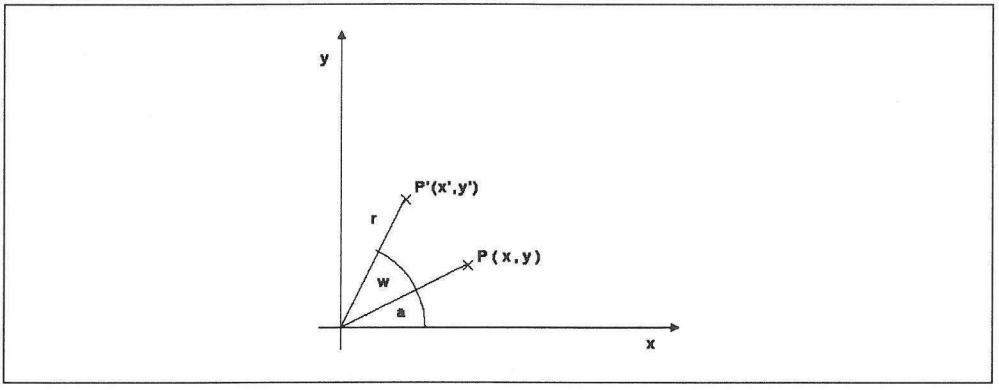


Bild 3.11: Drehen eines Punktes um den Nullpunkt

Sie wissen noch nicht – oder nicht mehr – genau, was es mit Winkelfunktionen (Sinus oder Cosinus etc.) auf sich hat? Kein Problem, schauen Sie in den Anhang! Dort finden Sie alles für uns Wissenswerte über dieses Thema, und da setzen wir die Kenntnis voraus, was unter Winkelfunktionen zu verstehen ist.

Ein Wort zur Abbildung: Dort wird ein Punkt P(x,y) um den Winkel w um den Nullpunkt gedreht. Der resultierende Punkt heißt P'(x',y'). Sowohl P als auch P' haben natürlich den gleichen Abstand r vom Drehzentrum. Die Verbindung zwischen P und dem Nullpunkt bildet mit der x-Achse den Winkel a. Damit bildet die Verbindung P' bis zum Nullpunkt mit der x-Achse den Winkel w + a.

Entsprechend den Definitionen für Sinus und Cosinus können wir aus unserer Skizze die folgenden vier Gleichungen ableiten:

$$\cos(a) = x/r$$
$$\sin(a) = y/r$$
$$\cos(a+w) = x'/r$$
$$\sin(a+w) = y'/r$$

Damit haben wir schon einmal einen Grundstock, auf den wir aufbauen können. Die beiden letzten Beziehungen lassen sich einfach umformen in:

$$x' = r*\cos(a+w)$$
$$y' = r*\sin(a+w)$$

x' und y' sind bekanntlich die beiden Koordinaten des gedrehten Punktes, auf deren Suche wir sind. x und y stellen die Koordinaten des Ursprungspunktes dar und w ist der Drehwinkel.

Etwas Kopfzerbrechen machen uns die Werte r und a. Sie sind uns nämlich unbekannt. Also werden wir versuchen, sie zu ersetzen. Dabei sind uns zwei sogenannte Additionstheoreme für Winkelfunktionen sehr hilfreich, die wir hier nicht beweisen wollen, da sie in jeder mathematischen Formelsammlung nachgeschlagen werden können:

$$\sin(s+t) = \sin(s)*\cos(t) + \cos(s)*\sin(t)$$
$$\cos(s+t) = \cos(s)*\cos(t) - \sin(s)*\sin(t)$$

Wir werden nun natürlich für s unser a und für t unser w einsetzen. Damit können wir die obigen Gleichungen leicht umformen:

$$x' = r * [\cos(a)*\cos(w) - \sin(a)*\sin(w)]$$
$$y' = r * [\sin(a)*\cos(w) + \cos(a)*\sin(w)]$$

Was fangen wir nun mit einem solchen Formelmonstrum an? Und a und r stören uns immer noch. Sofort fällt unser Blick auf die allerersten zwei Formeln, die wir noch nicht verwendet haben. Sie erlauben es uns, alle sin(a) durch y/r und alle cos(a) durch x/r zu ersetzen. Damit hätten wir alle Terme, in denen a vorkommt durch andere ausgetauscht:

$$x' = r * [x/r * \cos(w) - y/r * \sin(w)]$$
$$y' = r * [y/r * \cos(w) + x/r * \sin(w)]$$

Und siehe da, wir haben nur noch eine Unbekannte Größe: r. Doch die ist auch nur scheinbar. Denn wenn Sie sich einmal die Mühe machen und die eckigen Klammern auflösen, dann verschwindet dieser Störenfried wie von selbst:

$$x' = x*\cos(w) - y*\sin(w)$$
$$y' = x*\sin(w) + y*\cos(w)$$

Genau das ist es, was wir haben wollten: Gleichungen, mit denen wir aus den bekannten Parametern x und y (Punktkoordinaten) und w (Drehwinkel) die Koordinaten x' und y' des resultierenden Punktes errechnen können.

Jetzt können wir uns auf die Suche nach einer geeigneten Matrix begeben. Betrachten wir uns noch einmal die allgemeine Multiplikation einer Punktmatrix mit einer quadratischen Matrix, wie sie uns vorschwebt:

$$P' = P*M = (x\ y) * \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix}$$
$$= (x*a_{11} + y*a_{21} \qquad x*a_{12} + y*a_{22})$$

Im Falle der Drehung sollte die resultierende Punktmatrix wie folgt lauten:

$$P' = (\ x*\cos(w) - y*\sin(w) \qquad x*\sin(w) + y*\cos(w)\ )$$

Setzen wir folgendes gleich:

$a_{11} = \cos(w)$
$a_{21} = -\sin(w)$
$a_{12} = \sin(w)$
$a_{22} = \cos(w)$

dann erhalten wir die folgende Transformationsmatrix für die allgemeine Rotation (gegen den Uhrzeigersinn) um den Koordinatenursprung:

$$R(w) = \begin{pmatrix} \cos(w) & \sin(w) \\ -\sin(w) & \cos(w) \end{pmatrix}$$

Ein Beispiel:
Angenommen wir wünschen, den Punkt P mit den Koordinaten P(3,4) um den Winkel 60 Grad um den Nullpunkt zu drehen. (Wenn Sie nachrechnen, stellen Sie Ihren Taschenrechner auf DEGree für normale Gradberechnung. Normalerweise rechnen Computer nämlich mit RADiant.) Unsere Rotationsmatrix sieht dann so aus:

$$R(60) = \begin{pmatrix} \cos(60) & \sin(60) \\ -\sin(60) & \cos(60) \end{pmatrix} = \begin{pmatrix} 0.5 & 0.866 \\ -0.866 & 0.5 \end{pmatrix}$$

Den resultierenden Punkt P' errechnen wir dann auf die folgende Art und Weise:

$$P' = P(3,4) * R(60) = (3\ \ 4) * \begin{pmatrix} 0.5 & 0.866 \\ -0.866 & 0.5 \end{pmatrix}$$
$$= (3*0.5-4*0.866 \quad 3*0.866+4*0.5)$$
$$= (-1.964 \quad 4.598)$$

Die Koordinaten des resultierenden Punktes lauten also P'(−1.964, 4.598). So einfach geht das.

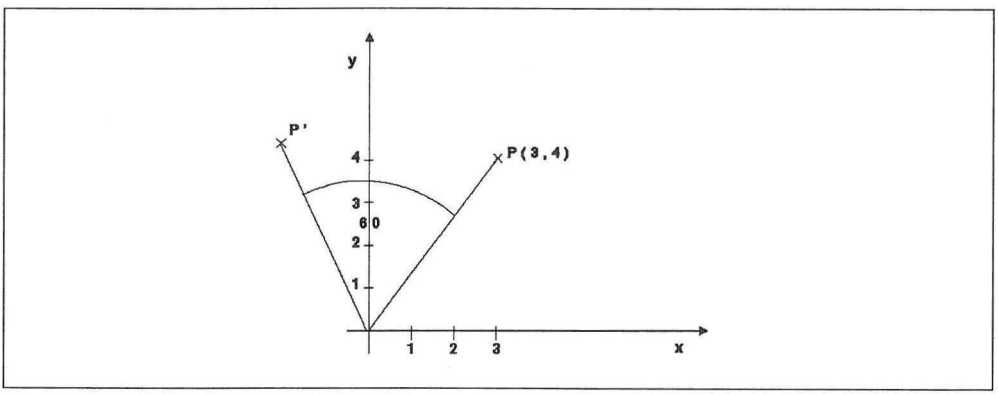


Bild 3.12: Rotation des Punktes P(3,4) um w = 60 Grad

Sehr einfach sind jetzt natürlich kombinierte Transformationen möglich. Angenommen Sie möchten Ihr Objekt sowohl drehen als auch vergrößern. Normalerweise würden wir diese beiden Transformationen nacheinander ausführen nach der Formel:

$P' = (P*R)*S$

Da aber, wie wir oben bereits gesehen haben, gilt:

$P' = (P*R)*S = P*(R*S)$

können wir auch eine einzige gemeinsame Transformationsmatrix M=R(w)*S(Sx,Sy) berechnen, die sowohl Rotation als auch Skalierung durchführt:

$$M = R(w)*S(S_x,S_y) = \begin{pmatrix} \cos(w) & \sin(w) \\ -\sin(w) & \cos(w) \end{pmatrix} * \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix}$$

$$= \begin{pmatrix} \cos(w)*S_x & \sin(w)*S_y \\ -\sin(w)*S_x & \cos(w)*S_y \end{pmatrix}$$

Auf die gleiche Weise lassen sich so mehrere Drehungen hintereinander in einer Matrix ausführen usw.

Das folgende kleine Beispielprogramm soll Ihnen das Prinzip der Rotation im Programm verdeutlichen:

```
'**********************************
'**                              **
'**  Rotation eines Kurvenzuges  **
'**                              **
'**********************************

'Funktion deklarieren:
DEF FNf(x)=SIN(x)/x
ON ERROR GOTO fehler

'Funktionsparameter:
a = 320             'x-Koordinate des Nullpunktes
b = 150             'y-Koordinate des Nullpunktes
f1 = 10             'x-Vergrößerungsfaktor
f2 = 50             'y-Vergrößerungsfaktor

'Schleife zur Veränderung des Rotations-Winkels
FOR wi=0 TO 30 STEP 5
  w = wi/180 * 3.141593       'Winkel in RAD umrechnen
```

```
  'y-Koordinatenachse einzeichnen:
  x = a : y = 200 : CALL rot.transform(w)
  PSET (xr%,yr%),2
  x = a : y =   0 : CALL rot.transform(w)
  LINE -(xr%,yr%),2

  'x-Koordinatenachse einzeichnen:
  x =   0 : y = b : CALL rot.transform(w)
  PSET (xr%,yr%),2
  x = 640 : y = b : CALL rot.transform(w)
  LINE -(xr%,yr%),2

  x = 0
  y = b - f2*FNf((x-a)/f1)     'Funktionswert für x=0 berechnen
  CALL rot.transform(w)
  PSET (xr%,yr%),3             'Punkt setzen

  'Berechnung der Funktionswerte:
  FOR x=0 TO 639
    y = b - f2*FNf((x-a)/f1)   'Funktionswert berechnen
    CALL rot.transform(w)      'und Rotieren
    LINE -(xr%,yr%),3          'Linie vom letzten Punkt
    onerr:
  NEXT x
NEXT wi

END

'Rotationstransformation für einen Punkt:
SUB rot.transform(w) STATIC
  SHARED x,y,xr%,yr%
  xr% =  x*COS(w) + y*SIN(w)
  yr% = -x*SIN(w) + y*COS(w)
END SUB
fehler:
  e=ERR
  IF e=11 OR e=6 THEN
    RESUME onerr
  END IF
ERROR e
```

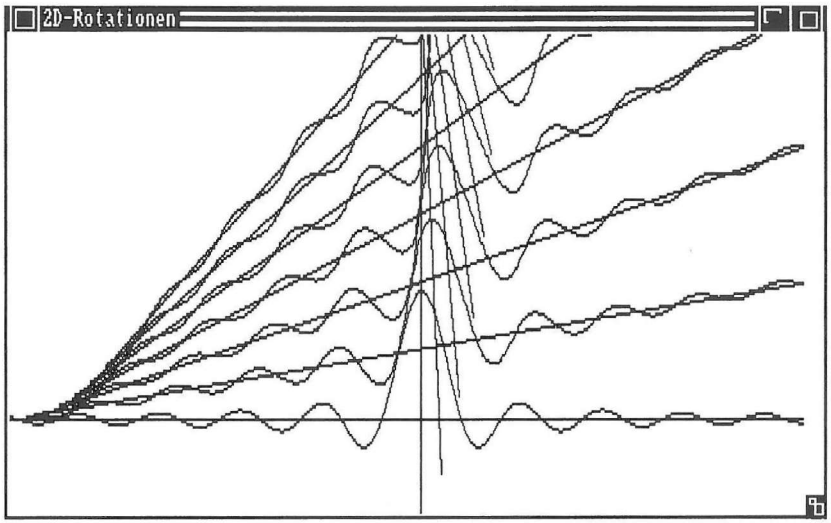


Bild 3.13: Rotation eines Kurvenzuges

Das Programm präsentiert Ihnen eine interessante Anwendung der Drehung. Hier wird nämlich die Kurve einer Funktion, die in der DEF-FN-Zeile definiert wurde (in diesem Programm: f(x) = sin(x)/x), um einen bestimmten Winkel w gedreht ins Ausgabefenster gezeichnet. Das Drehzentrum liegt allerdings im Koordinaten-Nullpunkt des Grafikfensters, da wir ja noch nicht um einen beliebigen Punkt drehen können.

Die für die Drehung zuständige Routine hat den Namen **rot.transform()** und führt mit den Koordinaten x und y eine Drehung um den übergebenen Winkel w aus. Die resultierenden Koordinaten liefert dieses Unterprogramm in xr und yr.

Alles umfassend zählt die äußerste FOR...NEXT-Schleife die Variable wi den Rotationswinkel (in Altgrad) schrittweise hoch. Die Koordinatenachsen werden kurz eingezeichnet (ebenfalls gedreht) und dann geht es auch schon an die Zeichnung der gedrehten Kurve. Dabei wird stets eine Linie vom jeweils zuletzt gezeichneten Punkt der Kurve zum aktuellen Punkt gezogen. Der erste Punkt für x = 0 muß deshalb getrennt berechnet werden. Lassen Sie sich bitte nicht von der umständlichen Formel zur Funktionswertberechnung irritieren. Die Parameter a, b, f1 und f2 dienen nur dazu, die Funktion korrekt und in der richtigen Größe ins Fenster zu rücken (es handelt sich hierbei übrigens ebenfalls um eine Transformation).

Aus Geschwindigkeitsgründen kann es in manchen Anwendungen von Vorteil sein, Sinus- und Cosinustabellen anzulegen, statt jedesmal diese Winkelfunktionen zu berechnen. Wird der Sinus eines bestimmten Winkels gesucht, so braucht das Programm einfach nur in dieser Tabelle nachzuschlagen.

3.2.5 Verschiebungen (Translationen) und homogene Koordinaten

Des öfteren hatten wir uns bereits eine Rotation um einen beliebigen Punkt gewünscht (bisher rotierten wir immer um den Koordinatenursprung). Eine solche Rotation könnten wir dadurch realisieren, daß wir den Koordinatenursprung auf den Punkt verschieben, den wir als Drehzentrum auserkoren haben. Nach der Drehung ist dann ein Zurückschieben notwendig. Es wäre also wünschenswert, eine Matrix für eine solche Verschiebung, eine sogenannte Translation, zu besitzen.

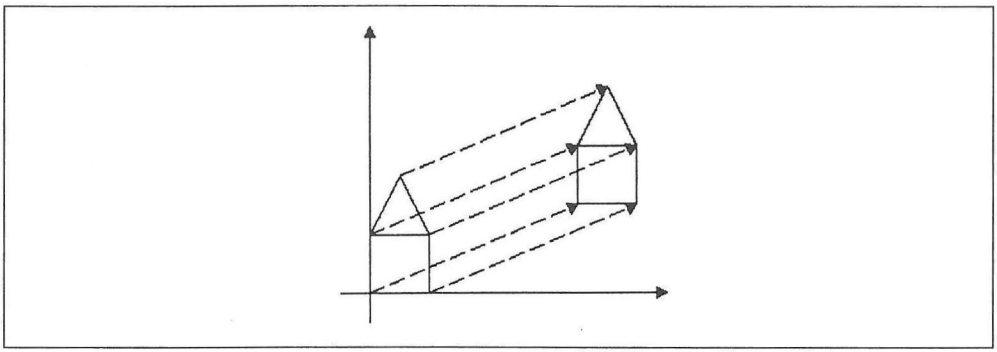


Bild 3.14: Translation (Verschiebung)

Die Verschiebung eines Punktes ist an und für sich eine einfache Sache:

$$x' = x+T_x$$
$$y' = y+T_y$$

wobei T_x und T_y die Werte der Verschiebung in x- und in y-Richtung darstellen. Dummerweise ist für diese Gleichungen keine 2x2-Transformationsmatrix zu finden, wie wir sie kennen. Aus diesem Grunde haben findige Mathematiker die sogenannten homogenen Koordinaten eingeführt. Statt unserer altbewährten 2x2-Matrizen werden wir in Zukunft nur noch mit 3x3-Transformationsmatrizen rechnen.

Bei den homogenen Koordinaten werden die Koordinaten eines Punktes nicht mehr – wie bisher üblich – durch zwei Werte angegeben. Vielmehr verwenden wir dabei drei. Unsere Punktmatrix sähe dann so aus:

$$P = (x*n \quad y*n \quad n)$$

Der Wert n ist dabei sozusagen eine Dummy-Koordinate, d.h. eine Koordinate, die in Wahrheit nicht existiert und nur aus rechnerischen Gründen eingeführt wird. Sie werden sehen, diese zusätzliche Koordinate wird sich automatisch immer in Luft auflösen, sobald wir eine Transformation durchgeführt haben. Die Einführung homogener Koordinaten ist nicht nur für dieses Problem sehr wichtig. Auch in den folgenden Kapiteln werden wir oft nicht um dieses Konstrukt herumkommen.

Sehr wichtig: Wir erhalten die richtigen Koordinaten x und y also stets durch eine Division der beiden ersten homogenen Koordinaten durch den dritten Wert n.

Ähnlich können wir unsere 2x2-Transformationsmatrizen erweitern. So verändert sich die Skalierungsmatrix

$$S(S_x,S_y) = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix}$$

für homogene Koordinaten in

$$S(S_x,S_y) = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Normalerweise werden wir n immer gleich 1 setzen. Das vereinfacht die Rechnung erheblich. Erst später bei der sogenannten Perspektivischen Projektion (3-D-Kapitel) kommen wir noch einmal darauf zurück. Eine Transformation eines Punktes führen wir dann z.B. so aus:

$$P' = P(x,y) * S(S_x,S_y)$$

$$= (x*n \quad y*n \quad n) * \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix} = (S_x*x*n \quad S_y*y*n \quad n)$$

Durch Division durch n rechnen wir um in die normalen Koordinaten:

$$P' = (S_x*x \quad S_y*y)$$

Das ist genau das Ergebnis, das wir bereits kennen. Da n gleich 1 ist, wird die Sache meist aber viel einfacher, als sie hier erscheint.

Wir können ebenso alle anderen Transformationsmatrizen in »homogene Matrizen« umwandeln, indem wir sie einfach durch »vier Nullen und eine Eins« erweitern:

$$R(w) = \begin{pmatrix} \cos(w) & \sin(w) & 0 \\ -\sin(w) & \cos(w) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Das Ergebnis bleibt natürlich stets dasjenige, das wir bereits kennen.

Jetzt interessiert uns aber natürlich brennend, wie denn nun die Transformationsmatrix für eine Translation lautet. Hier ist sie:

$$T(T_x,T_y) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{pmatrix}$$

Dabei sind T_x und T_y – wie oben – die Verschiebewerte in x- und in y-Richtung. Multiplizieren Sie doch einmal einen konkreten Punkt (z.B. P(5,9)) mit einer konkreten Translationsmatrix. Sie werden sehen, es stimmt! Jetzt haben wir den Grundstein für viele grafische Manipulationen gelegt, wie wir im nächsten Abschnitt gleich sehen werden.

3.2.6 Rotation um einen beliebigen Punkt

Die Lösung dieser Aufgabe sollte uns nun bereits vor Augen schweben (wir haben sie eben schon kurz angeschnitten). Um ein Objekt – oder besser einen Punkt – um einen beliebigen Punkt auf der Ebene rotieren zu lassen, benötigen wir drei Einzeltransformationen: Zunächst eine Verschiebung des Koordinatenursprunges zu dem Punkt hin, den wir als Drehzentrum wählen (andersherum: eine Verschiebung des Bildes mit dem Drehzentrum zum Nullpunkt); dann eine Rotation um den Koordinatenursprung (und damit um dieses Drehzentrum); und schließlich eine Rückschiebung des Bildes an die originale Position.

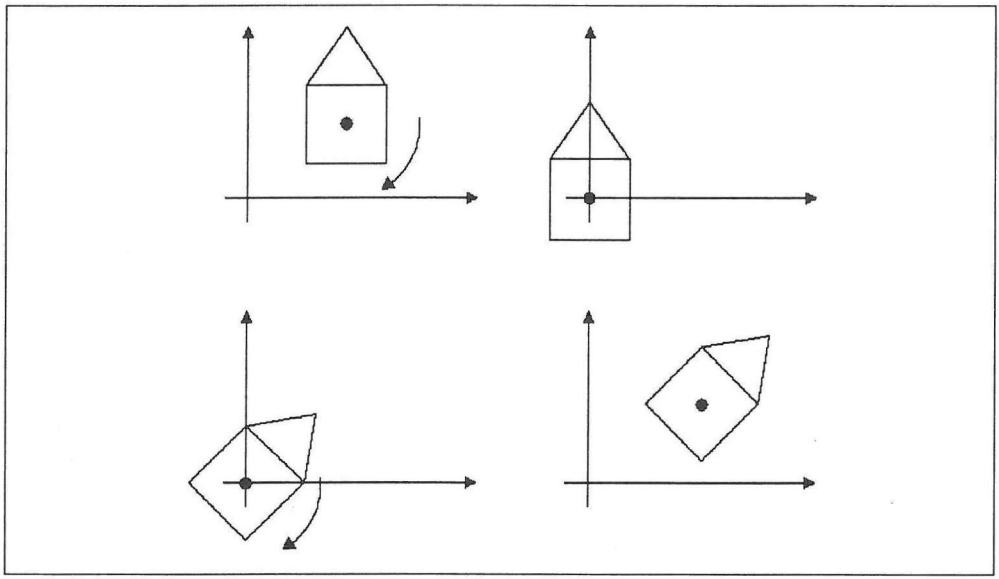


Bild 3.15: Zusammensetzung der Rotation um einen beliebigen Punkt aus Translationen und Rotation

Wir haben es also mit einer Translation, einer Rotation und einer anschließenden nochmaligen Translation zu tun:

$$P' = [(P*T_1)*R]*T_2 = P*(T_1*R*T_2)$$

Angenommen, das Zentrum der Rotation soll sein: $Z(x_z,y_z)$, dann lautet die Translationsmatrix, die Z an den Ursprung verschiebt:

$$T_1(-x_z,-y_z) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -x_z & -y_z & 1 \end{pmatrix}$$

Die Rotationsmatrix lautet, wie bekannt:

$$R(w) = \begin{pmatrix} \cos(w) & \sin(w) & 0 \\ -\sin(w) & \cos(w) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

und die Matrix, die das Drehzentrum wieder zurückschiebt, heißt:

$$T_2(x_z,y_z) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ x_z & y_z & 1 \end{pmatrix}$$

Wir brauchen nun nur noch das Produkt $T_1 * R * T_2$ dieser drei Matrizen zu bestimmen und erhalten:

$$R'(w,x_z,y_z) = T_1(-x_z,-y_z) * R(w) * T_2(x_z,y_z) =$$

$$\begin{pmatrix} \cos(w) & \sin(w) & 0 \\ -\sin(w) & \cos(w) & 0 \\ -x_z*\cos(w)+y_z*\sin(w)+x_z & -x_z*\sin(w)-y_z*\cos(w)+y_z & 1 \end{pmatrix}$$

Die Parameterschreibweise für die Rotation eines Punktes P(x,y) schaut dann so aus:

$$x' = x*\cos(w) - y*\sin(w) - x_z*\cos(w)+y_z*\sin(w)+x_z$$
$$y' = x*\sin(w) + y*\cos(w) - x_z*\sin(w)-y_z*\cos(w)+y_z$$

Die Rotationsmatrix R'(w) für Rotationen um beliebige Punkte sieht zwar ein wenig gewaltig aus, aber sie demonstriert Ihnen sicherlich sehr anschaulich, weswegen es gar keine schlechte Idee war, die Matrizenrechnung einzuführen. Auf diese Weise sind Sie in der Lage, jede beliebige Kombination von Transformationen in einer einzigen Matrix zusammenzufassen und damit später im Programm kompakt und effizient zu programmieren. Mit Transformationsmatrizen läßt sich eigentlich alles erfassen, was Sie mit einem Punkt machen wollen, sei es, ihn spiralförmig zu drehen oder ihn auf eine andere Funktion zu legen, wie wir es im Anschluß an diesen Abschnitt einmal vorstellen möchten. Gleichzeitig lassen sich diese Matrizen kombinieren usw. Durch die Matrizenschreibweise fällt Ihnen vielleicht auch etwas an der oben Matrix R' auf. Sie läßt sich auch durch eine Rotation mit anschließender Translation (also ohne Start-Translation) ersetzen. Die Translationsmatrix enthielte dann die Werte der dritten Reihe von R'.
Zum besseren Verständnis und natürlich zur Anschauung wollen wir unser gerade erworbenes Wissen gleich in einem kleinen Programm festigen:

```
'************************
'**                    **
'**  Rotation um einen  **
'**  beliebigen Punkt   **
'**                    **
'************************

Koordinaten des Rotationsobjektes:
'(Rechteck)
xr(0) = 100 : yr(0) = 100
xr(1) = 100 : yr(1) = 120
xr(2) = 160 : yr(2) = 120
xr(3) = 160 : yr(3) = 100
```

```
' Konstante Pi:
pi = 3.141593

' Drehzentrum Startkoordinaten:
xz = 130
yz = 110

' Drehwinkel:
w = pi/20

' Hauptschleife:
  WHILE INKEY$=""                'bis Taste

  IF MOUSE(0)=1 THEN             'Maustaste betätigt?
    xz = MOUSE(3)                'ja -> Mauskoordinaten
    yz = MOUSE(4)                'als neues Drehzentrum
  END IF

  '4 Eckpunkt-Koordinaten transformieren:
  FOR i=0 TO 3
    x = xr(i) : y = yr(i)
    CALL rot2.transform(w,xz,yz)
    xr(i) = xr : yr(i) = yr
  NEXT i
  CLS                            'Bildschirm löschen
  PSET (xz,yz),1                 'Drehzentrum zeichnen
  LINE (xr(0),yr(0))-(xr(1),yr(1)),3
  LINE -(xr(2),yr(2)),3          'Rechteck zeichnen
  LINE -(xr(3),yr(3)),3
  LINE -(xr(0),yr(0)),3
WEND

' Transformation (Rotation um beliebigen Punkt)
' w  - Rotationswinkel
' xz - x-Koordinate Drehzentrum
' yz - y-Koordinate Drehzentrum

SUB rot2.transform(w,xz,yz) STATIC
  SHARED x,y,xr,yr

  si = SIN(w)                    'Konstanten berechnen
  co = COS(w)
  xr = x*co-y*si-xz*co+yz*si+xz
  yr = x*si+y*co-xz*si-yz*co+yz
END SUB
```

Sobald Sie dieses kleine Programm starten, beginnt ein einzelnes Rechteck in dem Ausgabefenster zu rotieren. Ein kleiner Punkt markiert dabei das Drehzentrum. Nehmen Sie die Maus zur Hand, richten Sie den Mauszeiger auf einen beliebigen Punkt innerhalb des Fensters und betätigen Sie den linken Mausknopf. Sofort wechselt das Rotationszentrum zu dem von Ihnen angeklickten Ort usw. Möchten Sie das Programm verlassen, dann genügt ein einziger Tastendruck.

Zur Funktionsweise: Ganz zu Anfang der Routine werden die vier Eckpunkte des zu drehenden Rechteckes in zwei Arrays (xr( ) und yr( )) sowie das Drehzentrum in x_z und y_z, und der Drehwinkel in w definiert. Dann geht es in die Hauptschleife, die erst wieder verlassen wird, wenn Sie eine beliebige Taste betätigt haben. Die Funktion **MOUSE(0)** meldet ein Mausereignis (s. Basic-Handbuch), das dazu verwendet wird, neue Koordinaten für das Drehzentrum einzusetzen.

Die eigentliche Transformation der vier Eckpunkte geschieht in einer kleinen FOR...NEXT-Schleife, die ihrerseits das Unterprogramm **rot2.transform( )** für die Matrixmultiplikation aufruft. Die Konstanten si und co enthalten die Sinus- und Cosinuswerte von w. Wir hätten diese zwar auch einmal am Programmanfang definieren können (und hätten damit Rechenzeit gespart), da sich der Winkel nie verändert. Der Übersichtlichkeit halber aber haben wir das unterlassen.

Sind die vier gedrehten Punkte ermittelt, löscht das Programm den Bildschirm (bzw. das Ausgabefenster) und beginnt die Zeichenarbeit.

3.2.7 Ein kleiner Leckerbissen: Verformungen von Bildschirmbereichen

Für das Verständnis von 2-D- und später 3-D-Operationen ist das folgende Kapitel zwar nicht unbedingt notwendig (eilige Leser können es also überschlagen), bringt Ihnen aber einen interessanten Nebenaspekt hautnah ins Haus.

Sicher haben Sie das auch schon einmal gesehen: Ein Grafikprogramm projiziert einen ganzen Grafikblock auf eine Tonne oder verformt ihn längs einer Sinuskurve. »Wahnsinn!« werden Sie sagen. Doch wie einfach so etwas ist, das werden Sie im folgenden erfahren. Auch hierbei handelt es sich wieder um Operationen, die sich – meist – in Matrizenform niederschreiben lassen.

Oft werden die Werte einer normalen Funktion (z.B. die Sinusfunktion) berechnet und zu den x- bzw. y-Koordinaten jedes einzelnen Punktes eines (meist rechteckigen) Grafikblockes hinzuaddiert. Man könnte das folgendermaßen in Parameterschreibweise realisieren:

$$x' = x + f(y)$$
$$y' = y + g(x)$$

und in Matrixschreibweise:

$$P' = P * \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ f(y) & g(x) & 0 \end{pmatrix}$$

Statt f(y) und g(x) hätten wir genausogut f(x) und g(y) schreiben können. Wir werden hier ein einfaches Beispiel demonstrieren: Die Verformung eines Grafikblockes entlang einer Sinuskurve.

Dabei taucht allerdings noch ein weiteres Problem auf: Angenommen, der rechteckige Grafikblock, den wir verformen wollen, steht bereits irgendwo auf dem Bildschirm (oder im Speicher). Wollen wir ihn verformen, so müssen wir jeden einzelnen Punkt dieses Blockes abfragen und entsprechend transformieren (verformen). Dazu verwenden wir in Basic den Befehl **POINT,** der uns die Farbe eines Bildschirmpunktes zurückgibt. Näheres zeigt Ihnen sicherlich folgendes Programm:

```
'*************************
'**                     **
'**   Verformung von    **
'**  Bildschirmblöcken  **
'**                     **
'*************************

pi = 3.141593

'Zeichnen des zu verformenden Bereiches:

'Neues Füllmuster definieren:
DIM muster%(7)
muster%(0) = &H0
muster%(1) = &H7444
muster%(2) = &H4444
muster%(3) = &H6444
muster%(4) = &H4444
muster%(5) = &H7774
PATTERN &HFFFF,muster%

LINE (0,0)-(300,40),3,BF
PRINT : PRINT
PRINT TAB(4) "A u t o b a h n s c h l a n g e"

'Größenangabe des zu verformenden Quellblockes:
xq% = 0                 'x-Koordinate oben links
yq% = 0                 'y-Koordinate oben links
br% = 300               'Breite
ho% = 40                'Höhe
```

```
'Position des Zielblockes:
xz = 10
yz = 100

'Funktion der Verformungskurve g(x):
DEF FNg(x) = ampl*SIN(anzs*x/(br%/(2*pi)))

'Amplitude der Verformungskurve:
ampl = 20

'Anzahl der Schwingungsperioden pro Block:
anzs = 1

'Zeichenroutine:
FOR x0=xq% TO br%-1          'Punkt fÜr Punkt
  FOR y0=yq% TO ho%-1        'bearbeiten
    c% = POINT(x0,y0)        'Farbe des Quellpunktes ermitteln
    IF c%<0 THEN             'falls gleich -1 =>  außerhalb des Fensters
      COLOR 1
      PRINT "Fehler!"
      PRINT "Quellbereich liegt nicht"
      PRINT "vollständig im Fenster!!!"
      GOTO schluss
    END IF
    COLOR c%                 'Farbe entsprechend einstellen
    x = x0-xq%               'relative Quellkoordinaten berechnen
    y = y0-yq%
    y = FNg(x)+y             'und Transformation (Verformung) ausführen
    PSET (xz+x,yz+y)         'Zielpunkt zeichnen
  NEXT y0
NEXT x0

schluss:
COLOR 1

'Muster wieder normalisieren:
DIM mu%(1)
mu%(0) = &HFFFF
mu%(1) = &HFFFF
PATTERN ,mu%
```

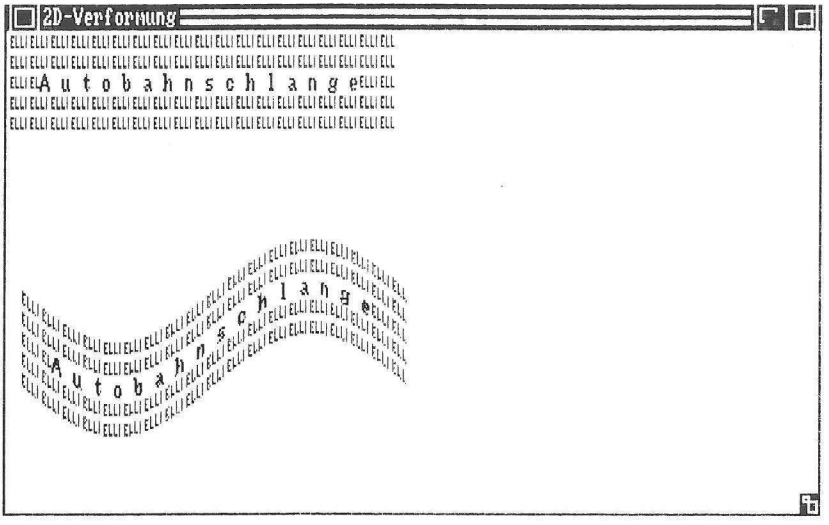


Bild 3.16: Verformung von Bildschirmbereichen

Hier wird zunächst der zu verformende Quellbereich gestaltet. Ein selbstdefiniertes Füllmuster und etwas Text sollen Ihnen den Effekt deutlich machen. Die genauen Abmaße des zu bearbeitenden Blockes werden festgehalten sowie die Position des Zielbereiches. Dann folgt die Definition der Verformungsfunktion.

Wie Sie vielleicht sehen, wurden hier für f(x) und g(x) folgende Funktionen eingesetzt:

$$f(y) = 0$$

$$g(x) = \text{ampl} * \sin(x * \frac{\text{anzs}}{\text{br}/(2*\text{pi})})$$

wobei:

ampl	Amplitude (»Ausschlag«) der Sinuskurve
anzs	Anzahl der Schwingungen in einem Grafikblock
br	Breite (in Punkten) des Grafikblockes
x	Relative x-Koordinate (relativ zum Blocknullpunkt)

Zum x-Wert jedes einzelnen durch POINT abgetasteten Punktes wird also nichts hinzugezählt. Zum y-Wert jedoch addiert das Programm den Funktionswert einer Sinuskurve. Das Ergebnis sehen Sie auf Ihrem Bildschirm. Der Ausdruck

$$\frac{\text{anzs}}{\text{br}/(2*\text{pi})}$$

stellt dabei den Stauchungs- (bzw. Streckungsfaktor) der Sinuskurve dar. Sein Wert bestimmt also, wie lang eine Schwingungsperiode ist. Er ist so gewählt, daß Sie angeben können, über wie viele Schwingungen der zu deformierende Grafikblock zu »modulieren« ist. br gibt dabei an, wie breit dieser Block ist (wie viele Punkte also die Periode umfaßt).

Sie dürfen die Funktion natürlich – wie immer – beliebig ändern. Sie brauchen sich eigentlich auch nicht so viel Mühe zu machen, wie wir das hier taten (Sie könnten ja auch feste Schwinungslängen und Amplituden vorsehen). Ersetzen Sie doch beispielsweise einmal die Zeilen vor PSET:

```
x = x0-xq%              'relative Quellkoordinaten berechnen
y = y0-yq%
y = FNg(x)+y            'und Transformation (Verformung) ausf.
```

durch:

```
xt = x0-xq%             'relative Quellkoordinaten berechnen
yt = y0-yq%
ampl = 10               'Amplitude für x-Koordinaten
br% = 40                'Höhe des Blockes
x = FNg(yt)+xt          'Verformung auch für x-Koordinaten
ampl = 20               'Amplitude für y-Koordinaten
br% = 300               'Breite des Blockes
y = FNg(xt)+yt          'und Transformation (Verformung) ausf.
```

In diesem Fall werden auch die x-Koordinaten transformiert. Das Ergebnis sollten Sie sich einmal anschauen. Das wirkt doch schon fast räumlich.

Denken Sie sich doch einmal andere Funktionen aus. Hier ein paar Tips:

Die Funktion zum Zeichnen einer halben Ellipse (s. Anfang des Kapitels), transformiert auf die y-Koordinate, erweckt den Anschein, der Grafikblock sei um eine Tonne gewickelt. Sie können auch später die perspektivische Projektion mit gleichzeitiger Drehung auf die Ebene des Quellblockes anwenden und damit den Block quasi in den Raum drehen etc. Projizieren Sie den Block doch einmal auf eine Kugel, auf einen Würfel o.ä. Ihrer Phantasie sind da kaum Grenzen gesetzt.

3.3 2-D-Clipping oder: Fenster zur Grafik

»Jede Grafik hat mal ein Ende« oder wie sollte dieses Kapitel eingeleitet werden? Es dreht sich um die Grafikgrenzen oder besser: Wie bestimme ich, wo ein Kreis, eine Linie, ein Rechteck endet? Nun, es geht, wie Sie vielleicht erkennen, um Grafikfenster. Wie erkenne ich, wo eine Linie enden muß, damit sie nicht aus einem bestimmten Grafikfenster hinausragt? Wo muß sie abgeschnitten, ge»clippt« werden?

Normalerweise haben wir mit dem sogenannten **Clipping** nicht viel zu tun. Sobald wir mit Fenstern (oder Layers) arbeiten, übernimmt das Betriebssystem diese Aufgabe. Trotzdem sind wir bei einigen Anwendungen gezwungen, diese Tätigkeit selbst zu übernehmen. Hier nun Lösungen für Punkte und für Linien:

3.3.1 Clippen von Punkten

Die zentrale Frage beim Punkt lautet: Darf er gezeichnet werden oder nicht. Es geht also um ja oder nein. Ein Punkt kann nicht geteilt werden wie ein Linie oder ein Kreis. Wie also testen wir, ob ein Punkt innerhalb oder außerhalb eines Fensters liegt?

Zur Vereinfachung nennen wir die Begrenzungskoordinaten eines Fensters wie folgt:

x_{min} x-Koordinate Ecke oben links
y_{min} y-Koordinate Ecke oben links
x_{max} x-Koordinate Ecke unten rechts
y_{min} y-Koordinate Ecke unten rechts

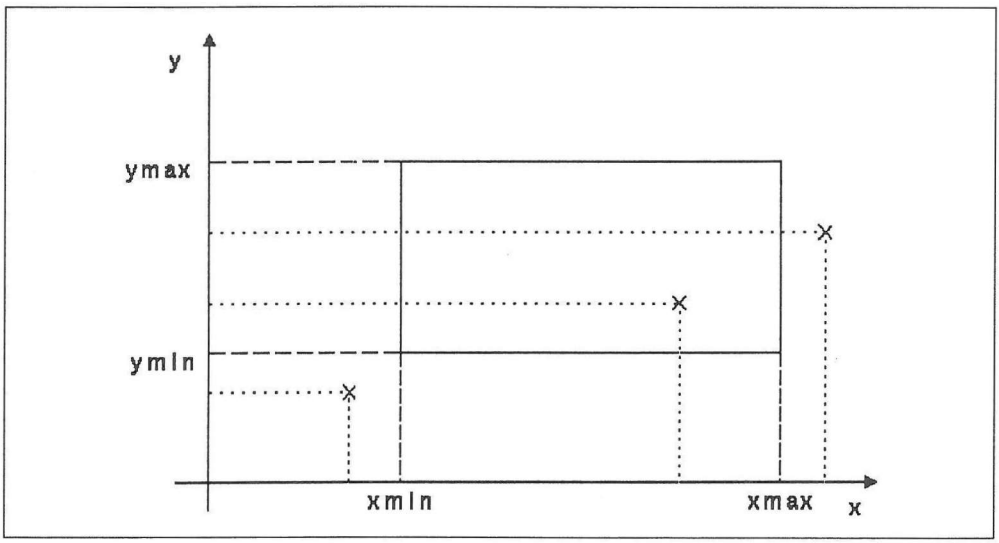


Bild 3.17: Punkt-Clipping

Ein einzelner Punkt mit den Koordinaten x und y ist genau dann innerhalb des Fensters, wenn alle folgenden vier Gleichungen wahr sind:

$$x \geqq x_{min} \qquad x \leqq x_{max}$$
$$y \geqq y_{min} \qquad y \leqq y_{max}$$

Wenn nur eine einzige dieser vier Gleichungen falsch ist, dann liegt der Punkt außerhalb des Fensters und ist somit unsichtbar. Diesen Punkttest könnten Sie selbstverständlich für jeden Punkt einer Linie, eines Kreises usw. durchführen. Aber wo bleibt da die Effizienz?

3.3.2 Clippen von Linien

Findige Leute haben aus diesem Grunde sehr schnelle Rechenvorschriften entwickelt, um z.B. Linien zu clippen. Ein solcher Algorithmus ist nach seinen Entwicklern Cohen und Sutherland benannt und soll nun vorgestellt werden.

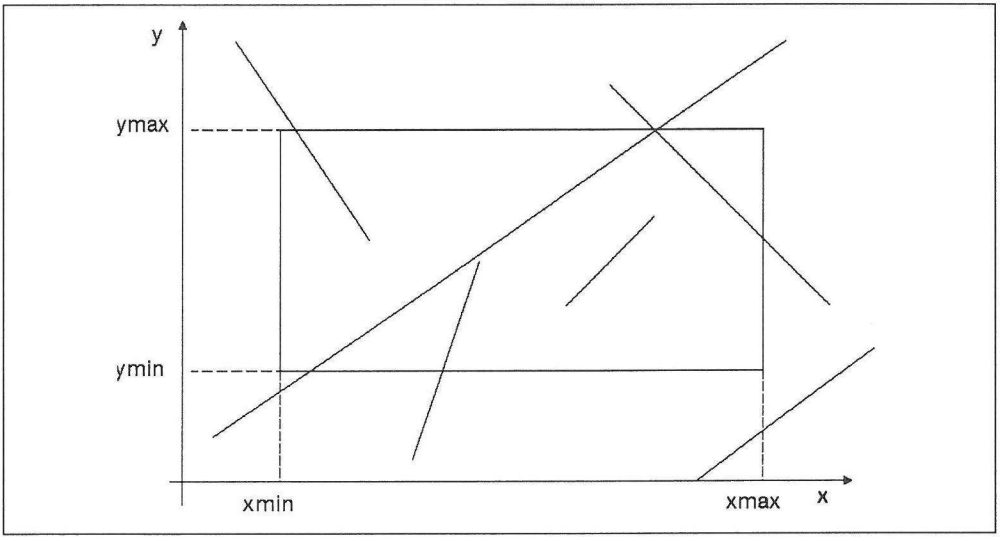


Bild 3.18: Linien-Clipping

Der Algorithmus:

Es gibt grundsätzlich drei verschiedene Kategorien von Linien (mit dem Startpunkt x_1, y_1 und dem Endpunkt x_2, y_2), die zu bearbeiten sind:

Kategorie 1:
Hierzu gehören Linien, die vollständig sichtbar sind, von denen also Start- und Endpunkte innerhalb des Fensters liegen. Für die beiden Punkte x_1, y_1 und x_2, y_2 gelten also die obigen Formeln zur Sichtbarkeit eines Punktes. Solche Linien können selbstverständlich sofort gezeichnet werden.

Kategorie 2:
Das sind Linien, die in jedem Fall völlig unsichtbar sind. Das ist der Fall, wenn auch nur eine einzige der folgenden Aussagen wahr ist:

a) $x_1 > x_{max}$ und $x_2 > x_{max}$
b) $x_1 < x_{min}$ und $x_2 < x_{min}$
c) $y_1 > y_{max}$ und $y_2 > y_{max}$
d) $y_1 < y_{min}$ und $y_2 < y_{min}$

Solche Linien brauchen also erst gar nicht gezeichnet zu werden.

Kategorie 3:
Zur Kategorie 3 gehören die Linien, die die obigen Bedingungen nicht erfüllen. Sie sind entweder völlig unsichtbar oder nur zum Teil sichtbar. Sie werden uns am meisten Kopfzerbrechen machen.

Im Cohen-Sutherland-Algorithmus wird nun festgestellt, zu welcher Kategorie eine Linie gehört. Dazu unterteilt er den Bereich, in dem die Linienendpunkte liegen können in 9 Abteilungen (s. Skizze):

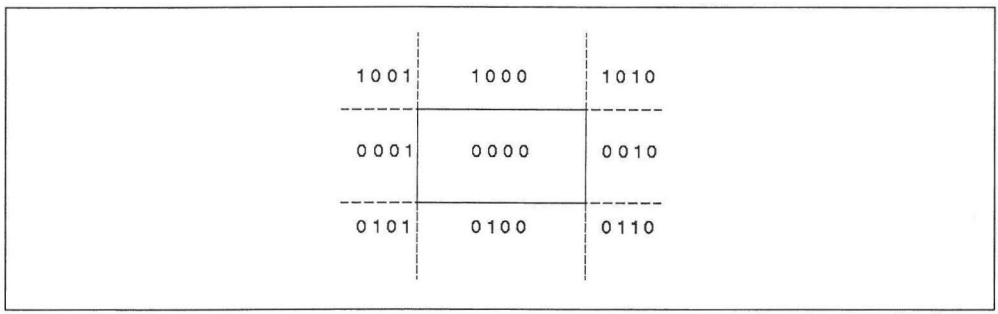


Bild 3.19: Die 9 Abteilungen im Cohen-Sutherland-Algorithmus

Nun wird für jeden Endpunkt der Linie ein vier Bit großer Speicher angelegt. Die einzelnen Bits für einen Punkt werden dann nach dem folgenden Schema gesetzt (Bit = 1) oder gelöscht (Bit = 0):

Bit 3	= 1 :	$y-y_{max}$	$\geqq$	0	(Punkt liegt über dem Fenster)
	= 0 :	$y-y_{max}$	$<$	0	(Punkt liegt nicht über dem Fenster)
Bit 2	= 1 :	$y_{min}-y$	$\geqq$	0	(Punkt liegt unter dem Fenster)
	= 0 :	$y_{min}-y$	$<$	0	(Punkt liegt nicht unter dem Fenster)
Bit 1	= 1 :	$x-x_{max}$	$\geqq$	0	(Punkt liegt rechts vom Fenster)
	= 0 :	$x-x_{max}$	$<$	0	(Punkt liegt nicht rechts vom Fenster)
Bit 0	= 1 :	$x_{min}-x$	$\geqq$	0	(Punkt liegt links vom Fenster)
	= 0 :	$x_{min}-x$	$<$	0	(Punkt liegt nicht links vom Fenster)

Bit 0 ist dabei das äußerste rechte Bit im Speicher. Falls für einen Punkt nach diesen vier Tests für alle 4 Bit der Wert 0 herauskommt, dann liegt er im Fenster.

Interessant wird es jetzt bei der Erkennung der richtigen Linienkategorie. Wir können nämlich jetzt einfach die so erlangten 4-Bit-Werte für die beiden Endpunkte einer Linie durch ein logisches UND verknüpfen und gelangen dann zu den folgenden Ergebnissen:

Kategorie 1:
Falls beide Werte gleich Null sind (logische ODER-Verknüpfung ist Null), dann gehört die Linie in die Kategorie 1, ist also vollständig sichtbar.

Kategorie 2:
Falls die logische UND-Verknüpfung beider Werte einen Wert ungleich Null ergibt, dann gilt für die Linie Kategorie 2, und sie ist nicht sichtbar.

Kategorie 3:
Ist das Ergebnis der UND-Verknüpfung gleich Null, dann muß die Linie wahrscheinlich geclippt werden.

Die Fälle 1 und 2 stellen kein Problem dar: Die Linie wird gezeichnet (Fall 1) oder oder sie wird es nicht (Fall 2). Schwieriger wird es nur bei Kategorie 3. Diese Linien müssen wir noch näher untersuchen. Hierzu unterteilt der Algorithmus jede Linie nun in immer kleinere Teile. Dies geht solange, bis er jeden der vielen Teile dieser Linie eindeutig in eine der ersten beiden Kategorien einordnen kann. Hierzu gibt es unter anderem die folgenden beiden Techniken:

Technik 1:
Dies ist die Technik, die uns vielleicht auf den ersten Blick einfällt. Man berechnet ganz einfach die Schnittpunkte der zu clippenden Linie mit den Rändern des Fensters. Das kann natürlich sehr zeitintensiv sein. Bei rechteckigen Fenstern allerdings, bei denen die vier Seiten parallel bzw. senkrecht zu den Koordinatenachsen stehen, vereinfachen sich die Tests um einiges: Wieder ziehen wir die 4-Bit-Codes der beiden Linienendpunkte von oben heran. Nach den folgenden Kriterien können Sie dann feststellen, mit welchen Fensterrändern die Linie geschnitten werden muß (für eine Linie können maximal vier Fälle gleichzeitig eintreten):

Fall 1: Bit3 = 1: Schnittpunkt mit der Linie $y = y_{max}$
Fall 2: Bit2 = 1: Schnittpunkt mit der Linie $y = y_{min}$
Fall 3: Bit1 = 1: Schnittpunkt mit der Linie $x = x_{max}$
Fall 4: Bit0 = 1: Schnittpunkt mit der Linie $x = x_{min}$

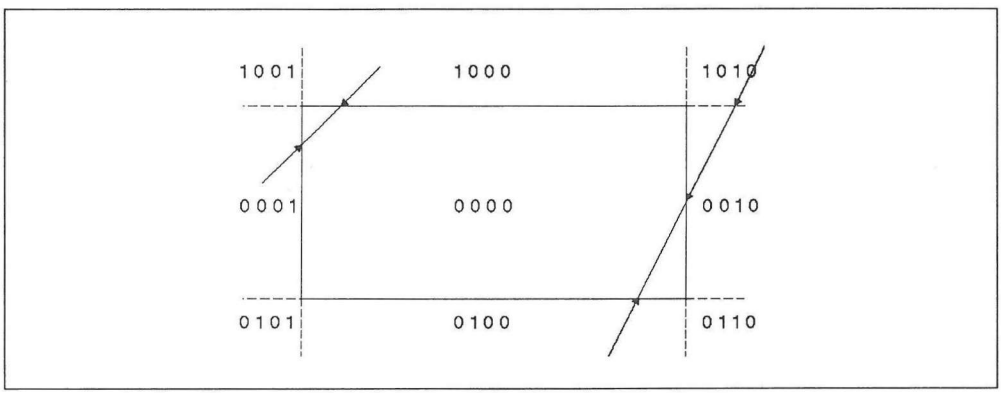


Bild 3.20: Die Schnittpunktberechnung nach Cohen-Sutherland

Nachdem Sie nun wissen, mit welchen Rändern die Linie geclippt werden muß, lassen Sie uns nun die Formeln präsentieren, nach denen die richtigen Schnittpunkte (also die neuen Endpunkte der Linie) ermittelt werden können. Ausgangspunkt ist dabei die im Anhang aufgeführte Zwei-Punkte-Geradengleichung.

Fall 1 und 2:
Hier kennen Sie die y-Koordinate y des Schnittpunktes bereits, nämlich y_{max} (Fall 1) oder y_{min} (Fall 2). Was bleibt, ist die unbekannte x-Koordinate x:

$$x = \frac{x_2 - x_1}{y_2 - y_1} * (y - y_2) + x_2$$

Fall 3 und 4:
Hier liegen die umgekehrten Verhältnisse vor. Die x-Koordinate des Schnittpunktes ist bekannt (x_{max} für Fall 3 und x_{min} für Fall 4). Gesucht ist die y-Koordinate y:

$$y = \frac{y_2 - y_1}{x_2 - x_1} * (x - x_2) + y_2$$

Für beide Formeln gilt:

x	x-Koordinate des Schnittpunktes
y	y-Koordinate des Schnittpunktes
x_1	x-Koordinate des Linienstartpunktes
y_1	y-Koordinate des Linienstartpunktes
x_2	x-Koordinate des Linienendpunktes
y_2	y-Koordinate des Linienendpunktes

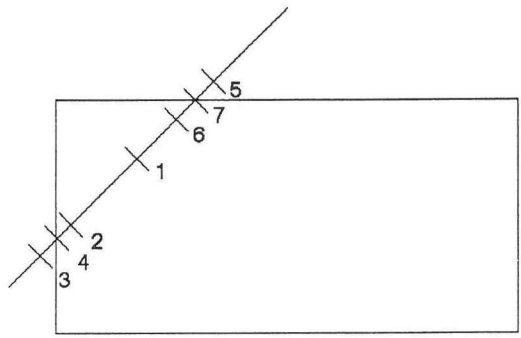
Für jede Linie müssen Sie dann also mindestens einen, höchstens aber vier Schnittpunkte berechnen, die die Linie in zwei bis fünf Teile zerlegen, von denen aber nur einer sichtbar sein kann. Welcher Linienteil das ist, das bestimmen Sie wieder mit Hilfe der 4-Bit-Berechnung. Die Endpunkte der verkürzten Linie kennen Sie, also sollten Sie nicht länger warten und das ermittelte Stück auf den Bildschirm bringen.

Technik 2:
Für normale Verhältnisse ist die beschriebene Rechenvorschrift eigentlich gut geeignet. In Assembler allerdings und bei Problemen, die möglichst schnell abzuarbeiten sind, reicht diese Form nicht aus. Um den richtigen Teil der Linie bzw. dessen Endpunkte zu errechnen, sind Divisionen und Multiplikationen mit Kommabehandlung notwendig. Das kann ein alter Assembler-Freak natürlich nicht dulden. Aus diesem Grunde verlangt es uns nach einer Methode, in der ausnahmslos Integer-Arithmetik verwendet werden kann. Mehr noch: Wir benötigen lediglich Integer-Additionen, -Subtraktionen und Verschiebebefehle (Division durch 2, 4 etc.).

Der Algorithmus läuft rekursiv ab. Zunächst teilen Sie dazu die Linie in zwei gleiche Teile. Den Mittelpunkt $Pm(x_m,y_m)$ einer Linie ermitteln Sie durch folgende Formeln, die leicht durch einfache Integer-Operationen durchzuführen sind:

$$x_m = \frac{x_1 + x_2}{2} \qquad y_m = \frac{y_1 + y_2}{2}$$

Sie haben nun zwei Linien, für die Sie wieder die Kategorisierung (s.o.) vornehmen. Diejenigen Linienteile, die in Kategorie 3 fallen, werden wieder halbiert usw. Das geht so lange, bis jeder kleinste Linienteil eindeutig einer der ersten beiden Kategorien zuzuordnen ist. Sie wissen nun, welche Linien Sie zeichnen dürfen und welche nicht! Gerade für die Sprache C, die ja Rekursionen voll unterstützt, eignet sich dieser Algorithmus vorzüglich. Aber auch unter 68000-Assemblern mit den Befehlen **LINK** und **UNLINK** sind ja sehr einfach lokale Variablen und damit Rekursionen möglich.



Mittenteilungsalgorithmus zum Linien-Clipping

Bild 3.21: Rekursive Linienhalbierung

Lassen Sie uns eine kleine Betrachtung über die Anzahl der Schritte vornehmen, um in etwa abzuschätzen, wie lange es maximal dauern könnte, bis die Schnittpunkte bestimmt sind. Die Schrittzahl hängt natürlich zunächst einmal von der Länge der Linie ab. Sie kann durch folgende Formel bestimmt werden:

$2^d = n \quad <=> \quad d = \log_2(n)$

wobei

d – Anzahl der Teilungen
n – Anzahl der Linienpunkte
$\log_2(n)$ – Zweier-Logarithmus von n

entspricht.

Ein Beispiel: Angenommen, eine Linie besitzt 256 Punkte, dann sind in maximal $\log_2(256) = 8$ Durchläufen die Clippunkte gefunden. Bei 512 Linienpunkten sind es maximal 9 und bei 1024 Punkten 10 Durchläufe. Das sind doch recht annehmbare Werte.

KAPITEL 4

Der Einstieg in die 3-D-Welt mit Ihrem Amiga

Nun sind wir endlich soweit. Der Sturz in die Tiefe des Raumes kann beginnen. Wir verfügen jetzt über die lebenswichtigen Voraussetzungen, um uns in der dreidimensionalen Welt behaupten zu können. Vieles, was nun folgt, wird Ihnen nicht mehr unbekannt erscheinen. Vielmehr werden Sie zahlreiche Bezüge zur zweidimensionalen Darstellung von Grafik feststellen. Wenn Sie also etwas hier nicht ganz verstehen, dann ist es sicher nützlich, ein paar Seiten zurückzublättern, um nachzuschauen, wie es denn in 2-D funktionierte.

Vor der Lektüre der folgenden Kapitel sollten Sie sich unbedingt mit diesen zweidimensionalen Grundlagen vertraut gemacht haben, da deren Kenntnis ab nun vorausgesetzt werden muß. Na, denn!

4.1 Alles ist relativ: Koordinatensysteme

Um unsere Welt bzw. unsere imaginäre Computerwelt, die wir im Endeffekt auf unserem Monitor, Drucker oder Plotter etc. darstellen möchten, zu beschreiben, bedarf es eines räumlichen Koordinatensystems. In diesem Koordinatensystem hat jeder Punkt, jede Ecke seine genau spezifizierte Position. Jedes Objekt kann so in seiner Lage und Größe angegeben werden. Seine Farbe oder andere wichtige Merkmale für seine Darstellung werden hier allerdings nicht erfaßt.

In einem zweidimensionalen Koordinatensystem, wie wir es bisher kennen, das nur alle Punkte innerhalb der Zeichenebene bezeichnen kann, benötigen wir zu Identifikation eines Punktes zwei Werte, die sogenannten Koordinaten. Diese Werte stellen, wie Sie alle wissen, den x- und y-Anteil der Position dar und wird an den senkrecht aufeinanderstehenden Koordinatenachsen abgelesen.

Im dreidimensionalen Koordinatensystem gesellt sich eine dritte, die z-Achse hinzu, die wiederum senkrecht auf den beiden anderen Koordinatenachsen steht, also sozusagen in die Zeichenebene hineinragt.

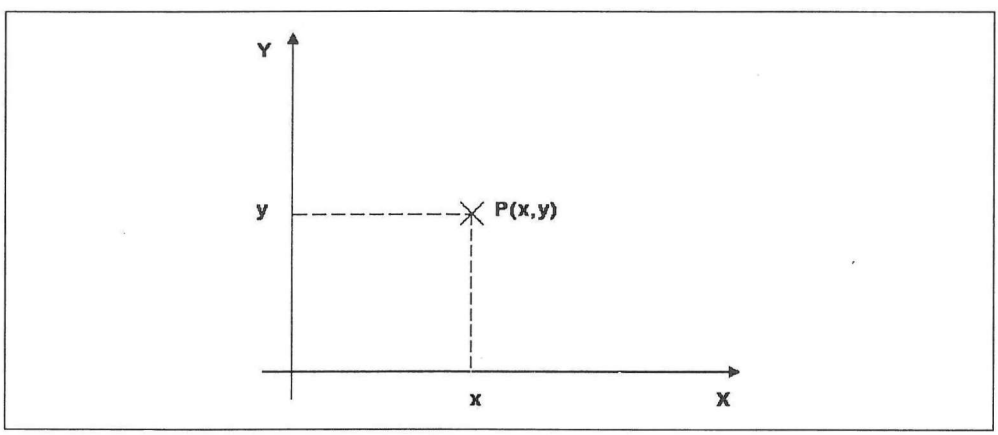


Bild 4.1: Zweidimensionales Koordinatensystem

Dabei existieren nun zwei verschiedene Möglichkeiten: Entweder die z-Achse ragt aus der Zeichenebene heraus (Rechtssystem) oder in sie hinein (Linkssystem).

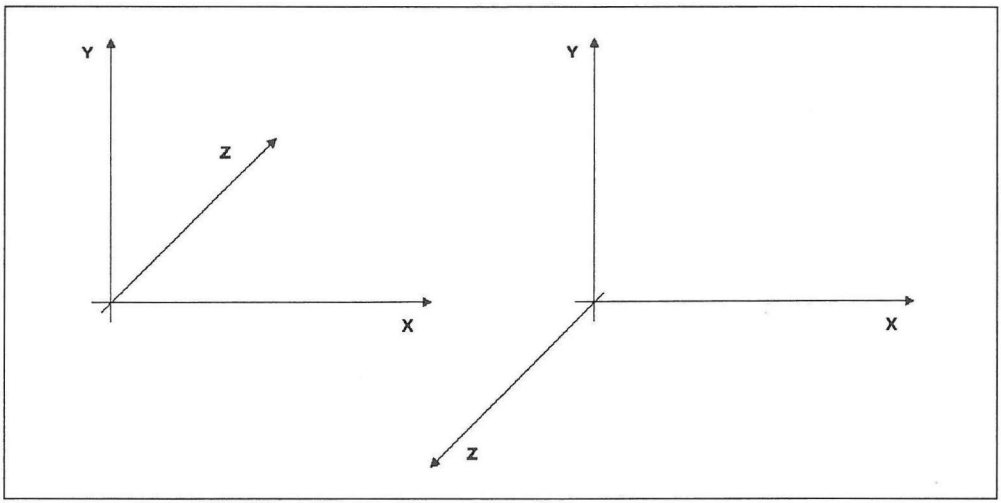


Bild 4.2: Rechts- und Linkssystem

Die z-Koordinaten der beiden Koordinatensysteme unterscheiden sich also nur in ihrem Vorzeichen. Wir werden in diesem Buch stets mit dem Linkssystem arbeiten, da es sich in der Computergrafik und auch in weiten Teilen der Mathematik eingebürgert hat. Es ist allerdings kein Problem, alle Rechnungen, Matrizen und Koordinaten in ein Rechtssystem zu übertragen (wie gesagt: x- und y-Koordinaten ändern sich nicht, nur die z-Koordinate wechselt ihr Vorzeichen).

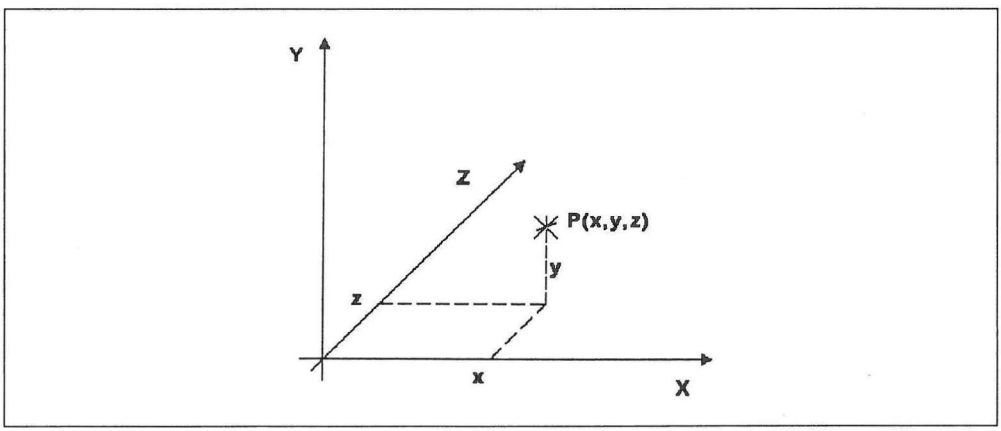


Bild 4.3: Eintragen eines Punktes in ein dreidimensionales Koordinatensystem

Wollen wir einen Punkt in einem räumlichen Koordinatensystem angeben, so benötigen wir dazu insgesamt drei verschiedene Werte: x-, y- und z-Koordinate (sie geben jeweils den Abstand des Punktes vom Koordinatennullpunkt in x-, y- und z-Richtung an). Wir geben einen 3-D-Punkt also wie folgt an: P(x,y,z). Im Gegensatz dazu lautete die Punktangabe unter 2-D-Verhältnissen stets: P(x,y).

Normalerweise kommen wir mit einem dreidimensionalen Koordinatensystem aus, um unsere ganze Welt bezüglich der Lage ihrer Elemente zu beschreiben. In der Computergrafik benötigen wir neben diesem sogenannten Welt-Koordinatensystem, das seinen Ursprung (Nullpunkt) irgendwo an einer beliebigen Stelle besitzt, mindestens noch ein weiteres Koordinatensystem zur Darstellung räumlicher Objekte: die Bild- oder Schirmkoordinaten. Wie der Name bereits andeutet, handelt es sich dabei um zweidimensionale Koordinaten, die sich lediglich auf den Bildschirm Ihres Rechners beziehen. Der Nullpunkt liegt dabei entweder unten oder oben links in der Ecke.

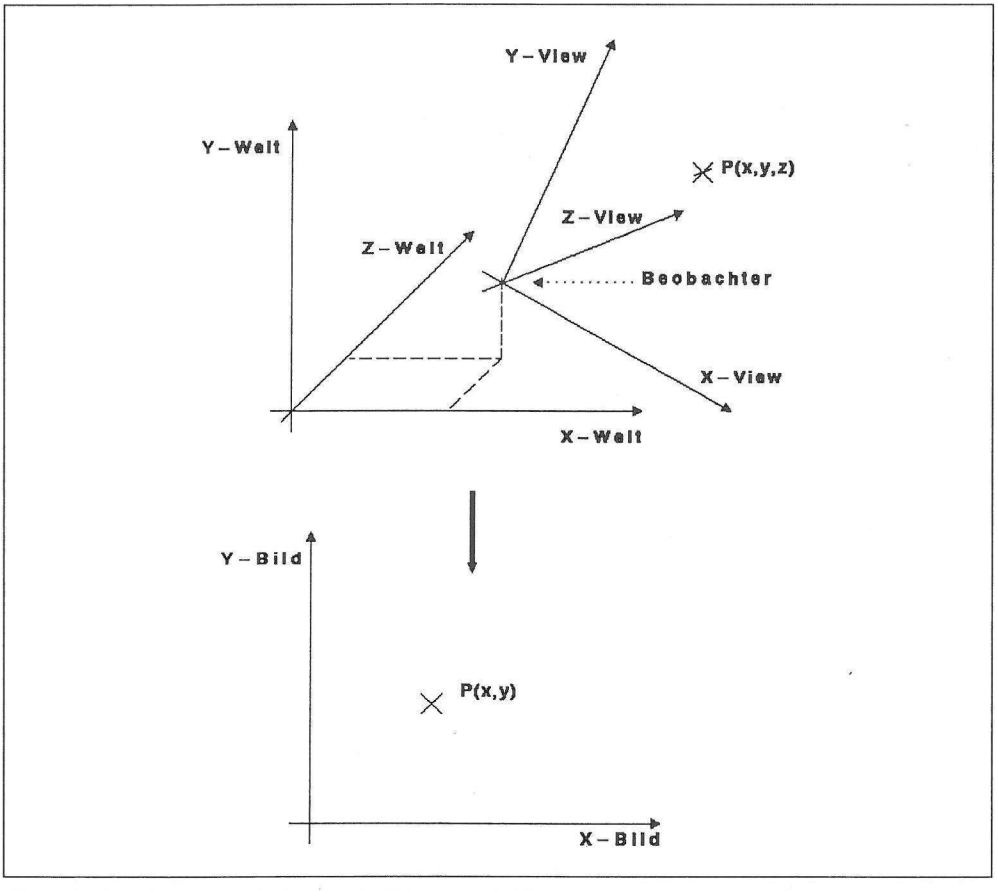


Bild 4.4: Beziehungen zwischen Welt-, View- und Bildsystem

Die dreidimensionalen Koordinaten des Weltsystems müssen also irgendwann einmal in die zweidimensionalen Schirmkoordinaten umgewandelt werden. Die dazu notwendigen mathematischen Operationen werden wir im Laufe dieses Kapitels noch kennenlernen. Natürlich spielen dabei wieder Transformationen wie Translation, Skalierung und Drehung eine entscheidende Rolle.

Wir bauen also unsere Welt in dem 3-D-Weltsystem auf und transformieren sie schließlich in die bescheidene 2-D-Welt unseres Bildschirms.

In vielen Computerprogrammen wird zudem noch Gebrauch von einem sogenannten View-Koordinatensystem gemacht. Das ebenfalls dreidimensionale View-Koordinatensystem unterscheidet sich vom normalen Weltsystem eigentlich nur in der Position seines Nullpunktes. Während die Position des Nullpunktes im Weltsystem an beliebiger Stelle liegen kann, die Koordinatenachsen entsprechend beliebig ausgerichtet sein können, befindet sich der Nullpunkt im Viewsystem genau dort, wo sich der Betrachter (View-Position) zur Zeit aufhält. Oft, nicht immer, liegt die z-Achse gleichzeitig noch in Blickrichtung, x- und y-Achsen verlaufen senkrecht bzw. waagerecht, bezogen auf den Beobachter. Damit besitzt der Benutzer einen anschaulichen Bezugspunkt, nämlich sich selbst. Gleichzeitig ist klar, welchen Teil der Welt der Beobachter betrachtet. Punkte mit negativen z-Koordinaten sind also unsichtbar.

4.2 Datenstruktur einer Raum-Welt

Im folgenden möchten wir den groben Aufbau einer dreidimensionalen Welt vorstellen, wie er in einem üblichen CAD-Programm verwirklicht wird. Bekommen Sie bitte keinen Schreck, wir brauchen natürlich in unseren »privaten« Anwendungen diesen Aufbau nicht vollständig zu übernehmen. Aufgezeigt wird hier quasi nur der Idealtyp. Es gibt natürlich noch eine Menge von Alternativen und Variationen, die hier nicht erfaßt werden können und zum Teil auch von der »kreativen Phantasie« des einzelnen Programmierers abhängen. Im Auge sollte man jedoch einige Standards der Datenspeicherung haben, die in diverser Fachliteratur beschrieben werden.

Wir wissen nun, wie wir die einzelnen Punkte einer räumlichen Welt definieren können. Wie aber bauen wir eine ganze Szene auf? Wir können ja schlecht jeden Punkt einzeln angeben, welch Aufwand! Wir haben bereits gesehen, daß wir zwei Punkte durch eine Linie verbinden können und dabei nicht jeden einzelnen Punkt dieser Linie anzugeben brauchen. Es werden lediglich die beiden Endpunkte gespeichert und angegeben, daß sie durch eine Linie verbunden werden. Aus vielen Linien errichten wir dann ein ganzes Bild. Um auch ungerade Konturen darstellen zu können, schließen wir andere Konturelemente mit ein: Kreis, Ellipse, Kurvenzug etc. Für diese Elemente reichen uns natürlich nicht zwei Punkte. Beim Kreis beispielsweise sind ein Mittelpunkt und ein Radius, bei der Ellipse ein Mittelpunkt und zwei Radien anzugeben, eventuell noch Start- und Endwinkel für Kreis- und Ellipsenbögen usw.

In räumlichen Systemen reicht das jedoch nicht mehr. Hier schließen 3-D-Programme viele solcher Konturelemente zu ganzen Flächen zusammen. Eine Fläche kann also von mehreren Linien oder Kombinationen von Linien und Kreisbögen etc. begrenzt werden. So, wie ein Punkt

zu mehreren Linien gehören kann, kann ein Konturelement von mehreren Flächen in Anspruch genommen werden. Viele dreidimensionalen Probleme (z.B. das Eliminieren von verdeckten Linien und Flächen oder die Schattierung von Flächen) setzen das Vorhandensein solcher Flächendefinitionen voraus.

Mehrere Flächen wiederum bilden zusammen ein sogenanntes Objekt (Quader, Kugel, Kegel, Pyramide...). Mehrere Objekte dürfen entsprechend zu höheren und komplexeren Objekten zusammengeschlossen werden. Damit haben wir ein recht kompliziertes, aber äußerst effizientes baumartiges Datensystem, das wir bei den räumlichen Berechnungen zu berücksichtigen haben. Dieses objektorientierte System hat entscheidende Vorteile. Objekte brauchen nur einmal entworfen zu werden. Sie können dann in eine Objektbibliothek auf Disk oder Harddisk gespeichert werden. Diese fertigen Objekte kann der Anwender dann jederzeit in sein Bild einbauen, sie vergrößern, verkleinern, verschieben, rotieren etc. Ein solches eingebautes Objekt kann aber auch jederzeit wieder geändert oder gar entfernt werden. Ein Maximum an Flexibilität ist gewährleistet.

In CAD-Systemen werden Bilder konstruiert, indem die verschiedensten Grundobjekte wie Quader, Würfel, Kegel etc. zu komplexen Gebilden zusammengesetzt werden. Die Anwender sind in der Lage, solche Gebilde (z.B. ein Haus) wiederum zu Objekten zu erklären und diese ihrerseits zur Konstruktion noch komplexerer Objekte zu verwenden (eine ganze Stadt beispielsweise) usw. Der Anwender eines CAD-Programmes sieht also nur die einzelnen Objekte und hat mit den Elementen eines Objektes (Flächen, Linien, Punkte nichts mehr zu tun).

Nehmen wir das Beispiel einer kleinen Häusersiedlung: Diese moderne Siedlung besteht aus lauter identischen Objekten, den Häusern. Jedes einzelne Haus ist, wie Häuser aus Bauklötzen, zusammengesetzt aus verschiedenen Grundobjekten (Dach, ein Quader als Gebäudeteil...) oder weiteren zusammengesetzten Objekten. Jedes dieser Grundobjekte wiederum besteht aus den begrenzenden Flächen, die ihrerseits von den umgebenden Kanten definiert werden. In der untersten Ebene stehen die Endpunkte der Kanten bzw. die anderen bestimmenden Parameter wie Radius bei Kreisen o.ä.

Da der Benutzer nur noch mit frei verschiebbaren, kopier- und duplizierbaren Objekten hantiert, ist es sinnvoll, für jedes Objekt ein eigenes Koordinatensystem einzurichten. Damit brauchen wir nur noch eine Koordinate anzugeben, wenn wir die Position dieses Objektes im Weltsystem oder, bei Unterobjekten, im übergeordneten Objektsystem festlegen möchten, nämlich die Koordinate des Objektsystem-Nullpunktes. Hinzu kämen noch die Winkel der Objektsystemachsen zu den jeweiligen Weltsystemachsen (bzw. zu den Objektsystemachsen des übergeordneten Objektes).

Innerhalb dieses Objekt-Koordinatensystems sind nun die verschiedenen Unterobjekte und Elemente (Flächen, Punkte,...) positioniert, aus denen sich das Objekt zusammensetzt (alles natürlich bezüglich des Objektsystems). Erst wenn die Welt tatsächlich gezeichnet werden soll, werden die objektinternen Koordinaten in die Koordinaten des übergeordneten Objektsystems

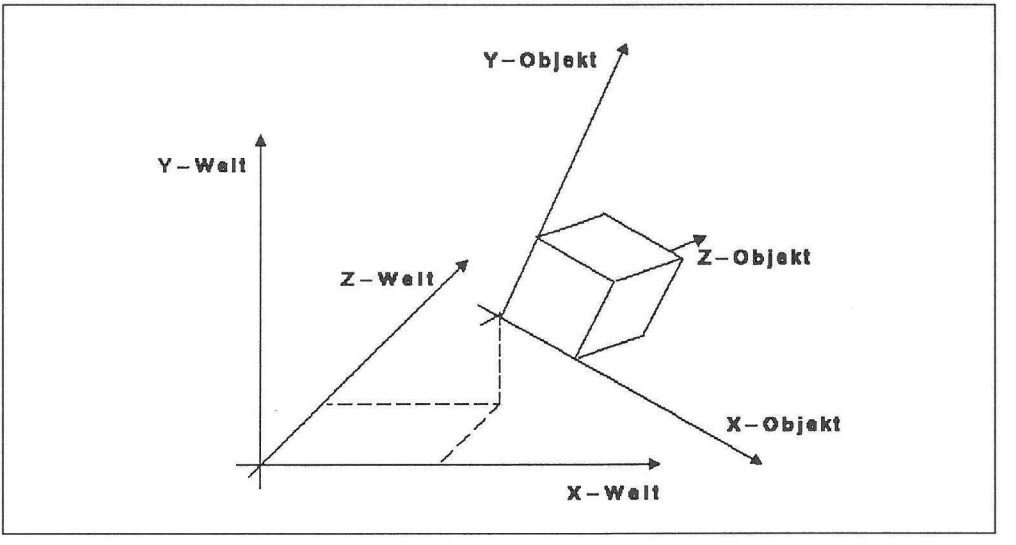


Bild 4.5: Ein Objekt im Weltsystem

und letztendlich in Weltkoordinaten umgerechnet. Auf diese Weise bleibt ein Objekt jederzeit sehr einfach in der »Welt« zugriffsfähig (verschiebbar, löschbar etc.).

Die Daten einer typischen »Welt« sehen also etwa so aus:

Zunächst existiert ein Hauptdatenblock namens »Welt«. Hier sind einfach die Namen (oder Nummern etc.) der Hauptobjekte eingetragen, aus denen die Welt besteht (das kann vielleicht nur ein einziges sein, das wiederum aus mehreren Objekten besteht). Hinzu kommen die Koordinaten, an denen sich diese Objekte im Weltsystem befinden. Aus jedem Eintrag muß erfahrbar sein, wo die Daten zu finden sind, die das jeweilige Objekt näher beschreiben (z.B. ein Zeiger auf eine Objekt-Datenstruktur):

Datenstruktur »Welt«:

- Kopfstruktur von »Welt« (z.B. Name der Welt, Anzahl der Objekte etc.)
- Name des ersten Objektes
- Zeiger auf die Datenstruktur dieses Objektes
- Koordinaten des Objektes
- Name des zweiten Objektes
- Zeiger auf die Datenstruktur dieses Objektes
- Koordinaten des Objektes
- Name des dritten Objektes
- ...

Jedes zusammengesetzte Objekt wird nun durch einen ähnlichen Datenblock beschrieben:

Datenstruktur »zusammengesetztes Objekt«:

- Kopfstruktur von »zusammengesetztes Objekt«
  z.B.:
  - Name des Objektes
  - Art des Objektes (zusammengesetzt oder Grundobjekt)
  - Anzahl der Unterobjekte
  - Flags wie: Objekt sichtbar? ja/nein
  - Translationsmatrix
  - Skalierungsmatrix
  - Rotationsmatrix

  etc.
- Name des ersten Unterobjektes
- Zeiger auf die Datenstruktur dieses Unterobjektes
- Koordinaten des Unterobjektes
- Name des zweiten Unterobjektes
- Zeiger auf die Datenstruktur dieses Unterobjektes
- Koordinaten des Unterobjektes
- Name des dritten Unterobjektes
- ...

Die Grundobjekte dagegen setzen sich nicht mehr aus Unterobjekten zusammen. Sie bestehen vielmehr aus einzelnen Flächen:

Datenstruktur »Grundobjekt«:

- Kopfstruktur von »Grundobjekt«
  z.B.:
  - Name des Objektes
  - Art des Objektes (zusammengesetzt oder Grundobjekt)
  - Flags wie: Objekt sichtbar? ja/nein
  - Translationsmatrix
  - Skalierungsmatrix
  - Rotationsmatrix

  etc.
- Zeiger auf die Definitionsstruktur der ersten Fläche
- Zeiger auf die Definitionsstruktur der zweiten Fläche
- ...

Die Kopfstruktur (Header) eines Objektes enthält zum Teil sehr wichtige Informationen. Da die Koordinaten der verschiedenen Teile eines Objektes (Unterobjekte, Punkte, Linien, Flächen etc.) fest und nicht (außer bei direkten Objektänderungen) veränderbar sein sollten, müssen wir angeben, um welche Werte das gesamte Objekt bezüglich dem übergeordneten Koordinatensystem bzw. dem übergeordneten Objekt (z.B. Weltsystem) verschoben, vergrößert oder rotiert sein soll. Dies geschieht mit Hilfe der angegebenen Transformationsmatrizen.

Dieses System hat entscheidende Vorteile. Zum einen bleiben alle Objekte fix und brauchen nur einmal definiert zu werden, auch wenn sie mehrmals im Bild auftauchen. Das spart einiges an Speicherplatz. Zum anderen brauchen nicht alle Koordinaten eines Objektes und aller seiner Unterobjekte bei einer Transformation des gesamten Objektes neu berechnet werden. Erst wenn ein Objekt auf dem Bildschirm dargestellt werden muß, werden die Punkte transformiert, die zu zeichnen sind. Das vereinfacht die Programmierung erheblich.

Oft ist es sinnvoller, die Transformationsmatrizen nicht in der Objektstruktur selbst, sondern vielmehr in der Struktur des übergeordneten Objektes anzugeben. Aber das hat der Programmierer nach seinen Erfordernissen selbst zu entscheiden.

In den beiden Objektstrukturen wird also ersichtlich, aus welchen Flächen sich das Grundobjekt zusammensetzt. Das kann z.B. auch nur eine einzige Fläche sein, wie bei einer Kugel oder einer einfachen Platte. Damit stehen wir vor dem Problem, eine Definitionsstruktur für eine beliebige Fläche zu schaffen:

Datenstruktur »Fläche«:

- Kopfstruktur von »Fläche«
  z.B. Art der Fläche:
  - Ebene, die sich aus Kanten zusammensetzt
  - Kugel, Ellipsoid, bei denen nur noch Mittelpunkt und Radien angegeben werden müssen
  - gebogene Fläche, die durch eine Funktion oder diverse Stützpunkte bestimmt wird (B-Splines, ...)
- Näher bestimmende Eigenschaften je nach Flächenart
  z.B.:
  - Zeiger auf die Definitionsstruktur der ersten Kante
  - Zeiger auf die Definitionsstruktur der zweiten Kante
  - ...

  oder:
  - Mittelpunktskoordinaten
  - Kugelradius

Wie Sie sehen, wird hier bereits nach der Art der Fläche differenziert. Die am meisten verwandten Flächenarten sind die ebenen, viereckigen Flächen, die sich aus vier Kanten zusammensetzen. Es sind jedoch noch eine ganze Menge anderer Flächentypen denkbar (z.B. Kugel, Halbkugel etc.). Auch sie müssen in der Flächenstruktur erfaßt werden. Meist aber beschränkt man sich, wie gesagt, auf die einfach zu handhabenden planen Flächen (Kugeln können z.B. ebenfalls aus solchen Flächen angenähert werden).

Diese planen Flächen setzen sich dann aus verschiedenen Kanten (mindestens drei) zusammen, für die ihrerseits je zwei Kantenendpunkte angegeben werden müssen:

Datenstruktur »Kanten«:

- Nummer des ersten Endpunktes
- Nummer des zweiten Endpunktes

Erst bei den durchnumerierten Endpunkten sind dann Koordinaten zu finden (natürlich relativ zum jeweiligen Objekt-Koordinatensystem).

Wenn wir in den obigen Strukturen von »Zeigern« gesprochen haben, dann dürfen das natürlich – wie bei den Kanten – auch Nummern o.ä sein. Auch dürfen Sie die verschiedenen Datenstrukturen zusammenfassen, wenn es Ihnen nutzt. Die höheren Strukturen (Objekte, Flächen etc.) sind auch wegzulassen, wenn es beispielsweise nur um ein sogenanntes Drahtmodell eines Objektes geht usw. Auch sollten Sie darauf achten, daß Ihr Objekte- und Strukturensystem nicht zu kompliziert wird, da das alles bei vielleicht noch hinzukommender etwas ungeschickter Organisation die Rechenzeit teilweise erheblich erhöhen kann.

Sie werden Teile des obigen Datensystems auch in diesem Buch immer wieder vorfinden, da es sich allgemein als sehr sinnvoll herausgestellt hat.

4.3 Mathematische Grundlagen zu 3-D-Berechnungen

Die Rechnung mit Matrizen und ihre Verwendung in der Computergrafik kennen Sie nun bereits aus dem 2-D-Kapitel. Für viele mathematische Ableitungen ist aber eine weitere Technik nicht fortzudenken: Die Vektorrechnung.

Es existieren zwar verschiedene mathematische Methoden zur Ableitung der unterschiedlichsten grafischen Figuren und Effekte. Die Vektorrechnung aber scheint mir (und nicht nur mir) dabei die anschaulichste und auch einfachste Form der Darstellung zu sein, wie Sie gleich sehen werden. Wir werden uns hier ein wenig intensiver mit ihr beschäftigen. Sollten Sie schon etwas auf diesem Gebiete bewandert sein, dann entschuldigen Sie bitte die oft recht unmathematische Ausdrucksweise, die dafür aber um so verständlicher ist.

a) Was ist ein Vektor?

Stellen Sie sich unter einem Vektor einfach einen Pfeil vor, der in eine bestimmte Richtung zeigt und eine festgelegte Länge besitzt. Um einen Vektor zu identifizieren, brauchen wir

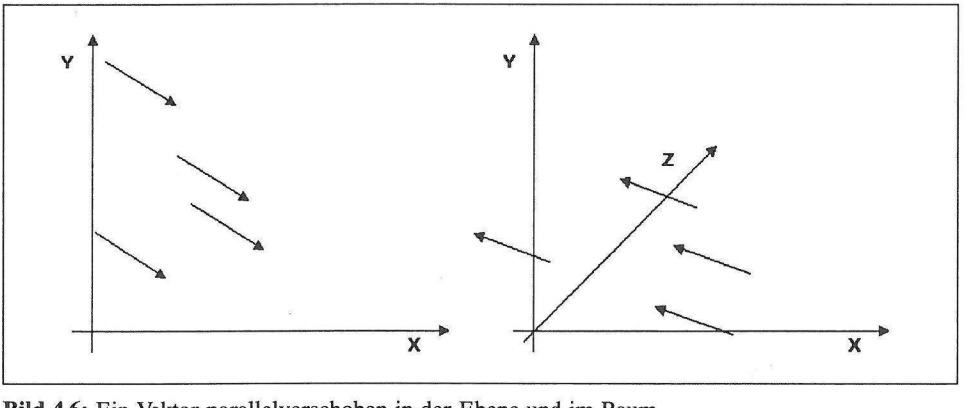


Bild 4.6: Ein Vektor parallelverschoben in der Ebene und im Raum

also nur zwei Parameter anzugeben: die Richtung und die Länge. Dieser Pfeil kann nun allerdings irgendwo auf der Zeichenebene (oder im Raum) liegen, ohne daß sich diese beiden Parameter ändern. Man spricht trotzdem stets vom selben Vektor (s. Abb.). Sie können also den Pfeil beliebig auf der Ebene oder im Raum verschieben. Es bleibt ein und derselbe Vektor, wenn sich nur die Länge und die Richtung nicht ändern. Man bezeichnet den Startpunkt eines Vektors auch als Fußpunkt, den Endpunkt als Spitze.

Wir werden in Zukunft Vektoren immer mit einem kleinen nach rechts gerichteten Pfeil (z.B.: $\vec{v}$) kennzeichnen. Einfache Zahlen sind hingegen normal gedruckt.

Hat ein Vektor die Länge Eins, dann nennt man ihn den Einheitsvektor. Der Nullvektor hat die Länge Null, seine Richtung ist unbestimmt.

Man kann mit Vektoren sogar rechnen: Die Richtung eines Vektors wechselt beispielsweise in die entgegengesetzte Richtung, indem wir sein Vorzeichen ändern ($-\vec{v}$). $\vec{v}$ und $-\vec{v}$ nennt man auch antiparallel (sie sind parallel und zeigen in entgegengesetzte Richtungen):

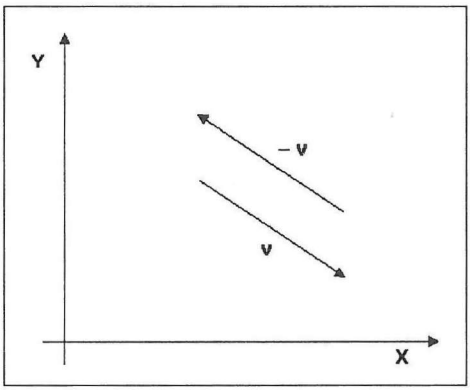


Bild 4.7: $\vec{v}$ und $-\vec{v}$

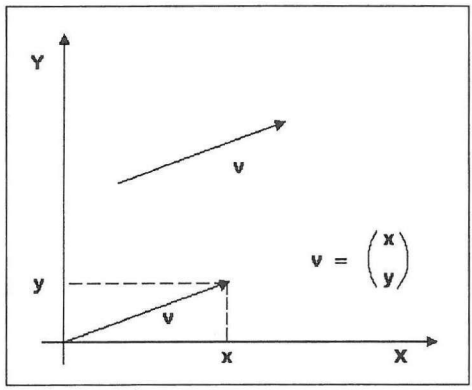


Bild 4.8: Koordinatenschreibweise eines Vektors

Man kann einen Vektor $\vec{v}$ auch alternativ durch einfache Koordinaten angeben. Dabei wird angenommen, daß sich der Fußpunkt des Vektors im Koordinaten-Nullpunkt befindet (wir können ihn ja beliebig verschieben). Als Koordinaten des Vektors werden dann nur die Koordinaten der Spitze angeführt (wir kommen also auch hier mit zwei Parametern für die Ebene und dreien für den Raum aus):

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

oder im Raum:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

Diese Koordinatenschreibweise ist sehr wichtig, da wir in Programmen letztendlich nur mit ihr rechnen werden. Mit Recht werden Sie Parallelen zu den Punktmatrizen feststellen. Ein Vektor kann auch als eine Art Matrize angesehen werden. Trotzdem sollten sie die beiden mathematischen Begriffe nicht verwechseln, da sie völlig unterschiedlich zu handhaben sind.

Die Länge eines Vektors ist sein Betrag (quasi eine Absolutfunktion für Vektoren). Sie kann durch folgende Formel berechnet werden, die wir uns gut merken sollten, da wir sie noch brauchen werden:

$$|\vec{v}| = \left| \begin{pmatrix} v_x \\ v_y \end{pmatrix} \right| = \sqrt{v_x^2 + v_y^2}$$

Zu Recht sehen Sie Ähnlichkeiten mit dem Satz des Pythagoras. Aus der obigen Skizze können Sie die Formel herleiten.

Bei räumlichen Vektoren sieht das Ganze so aus:

$$|\vec{v}| = \left| \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \right| = \sqrt{v_x^2 + v_y^2 + v_z^2}$$

Man kann mit Vektoren – wie gesagt – auch rechnen. Dabei gelten besondere Rechengesetze, die wir hier natürlich nicht beweisen möchten:

b) Multiplikation mit einer Zahl:

So ist es möglich, einen Vektor mit einer einfachen Zahl a zu multiplizieren. Das darf nicht mit dem weiter unten vorgestellten Skalarprodukt verwechselt werden! Grafisch ist darunter die Verlängerung oder Verkürzung des betreffenden Vektors $\vec{v}$ zu verstehen. Das Ergebnis ist also ebenfalls ein Vektor. Er besitzt die Länge $a*|\vec{v}|$. Ist a negativ, so kehrt sich gleichzeitig noch die Richtung des Vektors um. Bei $a = 0$ ist das Ergebnis der Nullvektor:

$$a*\vec{v} = a * \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} a*v_x \\ a*v_y \end{pmatrix}$$

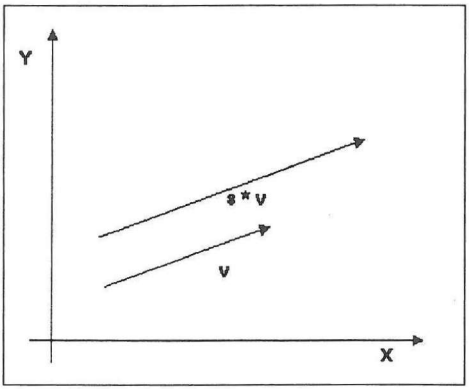


Bild 4.9: Multiplikation mit einer Zahl

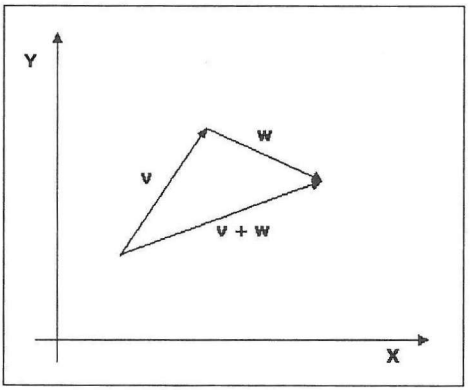


Bild 4.10: Vektoraddition

im Raum:

$$a*\vec{v} = a * \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} a*v_x \\ a*v_y \\ a*v_z \end{pmatrix}$$

In diesem Zusammenhang ist ein weiterer Begriff von Wichtigkeit: die lineare Unabhängigkeit von Vektoren. Zwei Vektoren sind genau dann linear unabhängig, wenn man den einen nicht durch eine Multiplikation mit einer einfachen Zahl in den anderen umwandeln kann. Mit anderen Worten: wenn sie nicht parallel oder antiparallel liegen. Linear abhängig oder besser kollinear sind demnach zwei Vektoren, die die gleiche Richtung (oder entgegengesetzte Richtungen), aber nicht unbedingt die gleiche Länge besitzen.

c) Vektoraddition/-subtraktion:

Wir können zwei Vektoren $\vec{v}$ und $\vec{w}$ auch addieren und subtrahieren. Das Ergebnis ist ebenfalls ein Vektor. Wollen wir eine solche Addition grafisch darstellen, dann brauchen wir nur den Fuß des Vektors $\vec{w}$ an die Spitze des Vektors $\vec{v}$ zu verschieben. Der resultierende Vektor geht dann einfach vom Fuß des Vektors $\vec{v}$ bis zur Spitze von $\vec{w}$ (s. Bild 4.10). In Koordinatenschreibweise sieht das Ganze dann so aus:

$$\vec{v}+\vec{w} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} + \begin{pmatrix} w_x \\ w_y \end{pmatrix} = \begin{pmatrix} v_x+w_x \\ v_y+w_y \end{pmatrix}$$

Entsprechendes gilt im Raum:

$$\vec{v}+\vec{w} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} + \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} v_x+w_x \\ v_y+w_y \\ v_z+w_z \end{pmatrix}$$

Das Subtrahieren zweier Vektoren geht ebenfalls sehr einfach und entspricht praktisch der Addition. Bei der Zeichnung kehren wir einfach das Vorzeichen des zweiten Vektors um ($-\vec{w}$ => Richtungswechsel) und addieren die beiden dann ($\vec{v}+(-\vec{w})$).

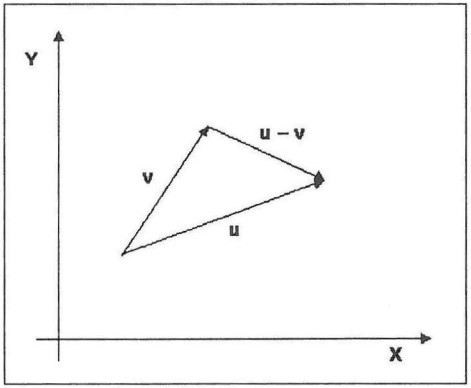


Bild 4.11: Vektorsubtraktion

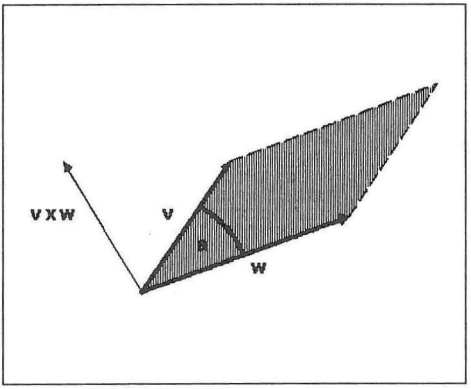


Bild 4.12: Vektorprodukt

$$\vec{v}+\vec{w} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} - \begin{pmatrix} w_x \\ w_y \end{pmatrix} = \begin{pmatrix} v_x-w_x \\ v_y-w_y \end{pmatrix}$$

Im Raum:

$$\vec{v}+\vec{w} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} - \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} v_x-w_x \\ v_y-w_y \\ v_z-w_z \end{pmatrix}$$

Rechengesetze:

$$\vec{v}+\vec{w} = \vec{w}+\vec{v} \qquad \vec{v}+(\vec{w}+\vec{u}) = (\vec{v}+\vec{w})+\vec{u}$$
$$a*(b*\vec{v}) = (a*b)*\vec{v} \qquad (a+b)*\vec{v} = a*\vec{v} + b*\vec{v}$$
$$a*(\vec{v}+\vec{w}) = a*\vec{v} + a*\vec{w}$$
$$|a*\vec{v}| = |a|*|\vec{v}|$$

d) Skalarprodukt:

Wir können zwei Vektoren $\vec{v}$ und $\vec{w}$ auch miteinander multiplizieren. Dabei gibt es allerdings zwei Möglichkeiten. Beim sogenannten Skalarprodukt $\vec{v}*\vec{w}$ (sprich: $\vec{v}$ mal $\vec{w}$) ist das Ergebnis eine einfache Zahl (Skalar), beim Vektorprodukt $\vec{v}\times\vec{w}$ (sprich: $\vec{v}$ kreuz $\vec{w}$) dagegen wieder ein Vektor.

Das Skalarprodukt ist wie folgt definiert:

$$\vec{v}*\vec{w} = |\vec{v}|*|\vec{w}|*\cos(a)$$

Dabei ist a der Winkel zwischen den beiden zu multiplizierenden Vektoren. Die Beträge von $\vec{v}$ und $\vec{w}$ sind – das ist bereits bekannt – die Längen der beiden Vektoren. Für die Koordinatenschreibweise gilt:

$$\vec{v}*\vec{w} = v_x*w_x + v_y*w_y$$

bzw. für räumliche Vektoren:

$$\vec{v}*\vec{w} = v_x*w_x + v_y*w_y + v_z*w_z$$

Das sieht schon einfacher aus und sind auch die Formeln, die später für uns wesentlich werden!

Eine wichtige Anwendung des Skalarproduktes ist die Berechnung des Winkels a zwischen zwei Vektoren durch Umstellung der obigen Formel:

$$\vec{v}*\vec{w} = |\vec{v}|*|\vec{w}|*\cos(a) \quad <=>$$
$$\cos(a) = \frac{\vec{v}*\vec{w}}{|\vec{v}|*|\vec{w}|} \quad <=>$$

$$\cos(a) = \frac{v_x*w_x + v_y*w_y + v_z*w_z}{\sqrt{(v_x^2 + v_y^2 + v_z^2)*(w_x^2 + w_y^2 + w_z^2)}}$$

Bei Vektoren in der Ebene setzen Sie v_z und w_z einfach gleich Null.

Sehr wichtig für unsere Belange ist auch eine andere Eigenschaft des Skalarproduktes. Sobald die beiden zu multiplizierenden Vektoren nämlich senkrecht aufeinanderstehen, wird der Cosinus ihres Winkels (Cosinus von 90 Grad) gleich Null und damit das gesamte Produkt:

$\vec{v} * \vec{w} = 0$ für: $\vec{v}$ steht senkrecht auf $\vec{w}$

Wir können so auf einfache Weise feststellen, ob zwei Vektoren aufeinander senkrecht stehen.

Es gibt natürlich unendlich viele Vektoren, die auf einem anderen Vektor senkrecht stehen. Es gibt in der Ebene aber nur zwei, die gleichzeitig dieselbe Länge besitzen (das kann man aus der Koordinatenschreibweise des Skalarproduktes berechnen):

Ausgangsvektor: $\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$

Darauf stehen senkrecht und haben die gleiche Länge:

$\vec{v}_1 = \begin{pmatrix} v_y \\ -v_x \end{pmatrix}$ und $\vec{v}_2 = \begin{pmatrix} -v_y \\ v_x \end{pmatrix}$

Für den Raum existieren allerdings weitaus mehr Möglichkeiten.

Rechengesetze:

$$\begin{aligned} \vec{v} * \vec{w} &= \vec{w} * \vec{v} \\ (a * \vec{v}) * \vec{w} &= a * (\vec{v} * \vec{w}) \\ \vec{v} * (\vec{w} + \vec{u}) &= \vec{v} * \vec{w} + \vec{v} * \vec{u} \\ \vec{v} * \vec{v} &= \vec{v}^2 \\ &= |\vec{v}|^2 \end{aligned}$$

Es gilt aber nicht unbedingt:

$\vec{v} * (\vec{w} * \vec{u}) = (\vec{v} * \vec{w}) * \vec{u}$

Mit anderen Worten: Sie müssen darauf achten, die Skalarmultiplikationen stets in der richtigen Reihenfolge auszuführen!

e) Vektorprodukt:

Das Vektorprodukt haben wir bereits erwähnt. Es ist die Art der Multiplikation zweier Vektoren, die wieder einen Vektor zum Ergebnis hat. Dieser neue Vektor $\vec{p} = \vec{v} \times \vec{w}$ hat eine Länge von:

$|\vec{p}| = |\vec{v} \times \vec{w}| = |\vec{v}| * |\vec{w}| * \sin(a)$

Dies ist übrigens auch der Flächeninhalt des von $\vec{v}$ und $\vec{w}$ aufgespannten Parallelogramms (s. Bild 4.12). $\vec{p}$ steht auf den beiden Vektoren $\vec{v}$ und $\vec{w}$ senkrecht, ragt also in den Raum hinein ($\vec{v}$, $\vec{w}$ und $\vec{p}$ ergeben ein rechtshändiges System). Einen senkrechten Vektor nennt man auch Normalvektor.

Mathematisch ist es wie folgt definiert:

$$\vec{p} = \vec{v} \times \vec{w} = \begin{pmatrix} v_y * w_z - v_z * w_y \\ v_z * w_x - v_x * w_z \\ v_x * w_y - v_y * w_x \end{pmatrix}$$

Rechengesetze:

$$\begin{aligned} \vec{v} \times \vec{w} &= -(\vec{w} \times \vec{v}) \qquad !!! \\ (a * \vec{v}) \times \vec{w} &= a * (\vec{v} \times \vec{w}) \\ \vec{v} \times (\vec{w} + \vec{u}) &= \vec{v} \times \vec{w} + \vec{v} \times \vec{u} \\ \vec{v} \times \vec{w} &= \text{Nullvektor (falls } \vec{v} \text{ zu } \vec{w} \text{ parallel bzw. antiparallel ist)} \\ \vec{v} \times \vec{v} &= \text{Nullvektor} \end{aligned}$$

f) Punktdarstellung durch Vektoren:

Das System der Vektoren ist sehr flexibel. Das zeigt uns z.B. die Art und Weise, wie wir einen einzelnen Punkt in einem Koordinatensystem auch durch die Vektorschreibweise ohne Probleme darstellen können (hier treffen sich Matrizen und Vektoren).

Der Trick ist ganz einfach: Man denke sich zunächst einen Punkt P mit den Koordinaten P(x,y). Weiter stelle man sich einen Vektor $\vec{v}$ vom Koordinaten-Nullpunkt bis zum gewünschten Punkt P im Koordinatensystem vor. Dieser Vektor kann in der Koordinatenschreibweise bekanntlich so angegeben werden:

$$\vec{v} = \begin{pmatrix} x \\ y \end{pmatrix} \qquad \text{bzw.:} \quad \vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

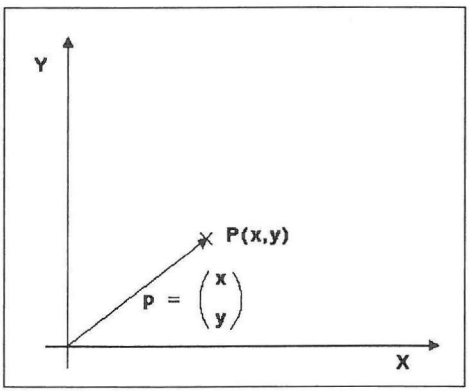


Bild 4.13: Punktdarstellung durch Vektoren

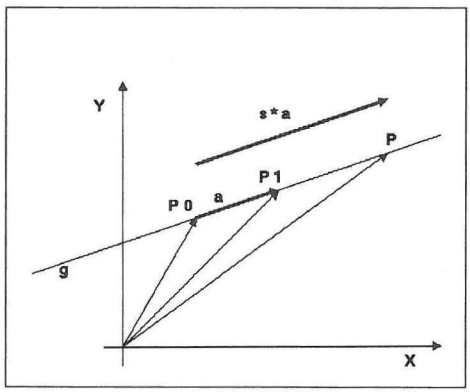


Bild 4.14: Vektorielle Geradengleichung

x, y und z sind schließlich die Koordinaten des Punktes. Statt mit dem Punkt selbst, rechnen wir also einfach mit dem Vektor zu diesem Punkt, ohne irgendwelche Probleme zu bekommen. Wir können sogar den Vektor $\vec{v}$ auch umbenennen in P:

$$P = \begin{pmatrix} x \\ y \end{pmatrix} \qquad \text{bzw.:} \quad P = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Mit P (steht ja eigentlich für den Punkt P) rechnen wir genauso wie mit normalen Vektoren – und davon werden wir regen Gebrauch machen.

g) Liniendarstellung durch Vektoren:

Wichtig für geometrische Berechnungen, wie wir sie vor uns haben, sind natürlich Linien bzw. Geraden. Aus diesem Grunde stellen wir hier einmal vor, wie eine zweidimensionale Gerade in Vektorform dargestellt werden kann. Betrachten Sie dazu am besten Bild 4.14.

Vektorielle Geradengleichung (sie gilt sowohl für die Ebene als auch für den Raum):

$$g: \vec{p} = \vec{p}_0 + t*\vec{a}$$

oder in Koordinatenschreibweise:

$$g: \begin{pmatrix} p_x \\ p_y \end{pmatrix} = \begin{pmatrix} p_{0x} \\ p_{0y} \end{pmatrix} + t*\begin{pmatrix} a_x \\ a_y \end{pmatrix}$$

$$= \begin{pmatrix} p_{0x} + t*a_x \\ p_{0y} + t*a_y \end{pmatrix}$$

Daraus kann man einfacher auch zwei Gleichungen machen, indem wir x- und y-Komponenten getrennt berechnen:

$$g: p_x = p_{0x} + t*a_x$$
$$p_y = p_{0y} + t*a_y$$

Im Raum gesellt sich die z-Koordinate hinzu:

$$g: p_x = p_{0x} + t*a_x$$
$$p_y = p_{0y} + t*a_y$$
$$p_z = p_{0z} + t*a_z$$

Dabei bedeuten:

- $\vec{p}$ – Vektor vom Koordinatenursprung zu dem zu berechnenden Punkt P der Geraden (oder einfach: Punkt P)
- $\vec{p}_0$ – Vektor vom Koordinatenursprung zu einem beliebigen Punkt P_0 der Geraden (oder einfach: Punkt P_0)
- $\vec{a}$ – »Richtungsvektor«, beliebiger Vektor parallel zur Geraden
- t – Faktor zur »Verlängerung« (oder Verkürzung) des Vektors $\vec{a}$, um Punkt P zu erreichen

Das kleine »g:« bedeutet, daß wir es hier mit einer Geradengleichung zu tun haben. Der Punkt (oder besser Vektor) $\vec{p}$ steht damit stellvertretend für alle Punkte der Linie.

Wir gehen also von einem Startpunkt P_0 auf der Geraden aus (es kann eigentlich jeder beliebige Punkt der Geraden dazu verwendet werden). Da Punkte jetzt und in Zukunft immer als Vektoren vom Koordinaten-Nullpunkt zum betreffenden Punkt ausgedrückt werden (s.o.), resultiert aus diesem Punkt der Vektor $\vec{p}_0$ (wir haben hier der Einfachheit halber und in völlig unmathematischer Weise Punkt und Vektor denselben Namen gegeben, da wir später nicht mehr zwischen Punkt und Vektor unterscheiden wollen). Von diesem Punkt aus ziehen wir einen zweiten Vektor $\vec{a}$ in Richtung der Geraden (auch dieser Vektor ist eigentlich beliebig gewählt und muß nur auf der Geraden liegen – egal wie lang er ist). Dieser Vektor gibt uns die Richtung an, in die die Gerade verläuft. Um nun einen beliebigen Punkt P auf der Geraden zu berechnen, müssen wir diesen Richtungsvektor nur noch mit einem Faktor t multiplizieren. Die beiden resultierenden Vektoren $\vec{p}_0$ und $t*\vec{a}$ werden addiert und ergeben den Ergebnisvektor $\vec{p}$, dessen Parameterform uns direkt die Koordinaten des Linienpunktes P angibt.

Um eine ganz bestimmte Gerade zu bestimmen, brauchen wir also lediglich die Parameter $\vec{p}_0$ und $\vec{a}$ anzugeben. Aus beliebigen Werten für t errechnen sich dann alle Punkte P (bzw. Vektoren $\vec{p}$) der Gerade.

h) Schnittpunkt zweier Geraden:

Wir können mit diesem Wissen auf einfache Weise den Schnittpunkt zweier Geraden in der Ebene berechnen. Angenommen wir haben zwei Geraden g und h mit den folgenden Gleichungen:

$$g: \vec{p} = \vec{p}_0 + t*\vec{a}$$
$$h: \vec{q} = \vec{q}_0 + s*\vec{b}$$

Da wir einen einzigen Punkt suchen, der auf beiden Geraden liegt, für den also $\vec{p}$ und $\vec{q}$ gleich werden, können wir die beiden Gleichungen auch gleichsetzen:

$$\vec{p}_0 + t*\vec{a} = \vec{q}_0 + s*\vec{b}$$

In Koordinatenschreibweise (in der Ebene können Sie die z-Parameter wie immer gleich Null setzen und damit weglassen):

$$\begin{pmatrix} p_{0x} \\ p_{0y} \\ p_{0z} \end{pmatrix} + t*\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} = \begin{pmatrix} q_{0x} \\ q_{0y} \\ q_{0z} \end{pmatrix} + s*\begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$$

oder:

$$\begin{aligned} p_{0x} + t*a_x &= q_{0x} + s*b_x \quad \text{und} \\ p_{0y} + t*a_y &= q_{0y} + s*b_y \quad \text{und} \\ p_{0z} + t*a_z &= q_{0z} + s*b_z \end{aligned}$$

Da wir die Parameter p_{0x}, p_{0y}, p_{0z}, a_x, a_y, a_z sowie q_x, q_y, q_z, b_x, b_y und b_z kennen (sie legen ja die beiden Geraden fest), müssen wir nur noch t und s bestimmen, um den Schnittpunkt P bzw. Q zu errechnen. Wir lösen zwei der Gleichungen also nach einer der beiden Variablen auf (hier nach t):

$$t = \frac{q_{0x} + s*b_x - p_{0x}}{a_x}$$

und:

$$t = \frac{q_{0y} + s*b_y - p_{0y}}{a_y}$$

Die beiden Gleichungen setzen wir wieder gleich, wodurch wir t eliminieren und s ausrechnen können. Das ausgerechnete s fügen wir dann in eine der Ausgangsgleichungen ein, um t zu ermitteln. s und t ergeben dann den Schnittpunkt, falls er existiert. Im Raum müssen wir die beiden Werte nämlich zunächst noch in die dritte Gleichung (hier die Gleichung mit den z-Parametern) einsetzen. Ergibt die Einsetzung eine wahre Aussage, so ist der Schnittpunkt perfekt. In der Ebene brauchen wir keine z-Koordinate und damit ist dieser letzte Test überflüssig. Zu keiner eindeutigen Lösung kommt es auch in der Ebene, wenn die beiden Geraden parallel verlaufen.

i) Ebenendarstellung durch Vektoren:

Speziell für unsere dreidimensionalen Belange ist es sinnvoll, auch Ebenen im Raum durch Vektoren zu definieren. Die Gleichung für eine Ebene ähnelt dabei der einer Geraden, wie wir sie oben kennengelernt haben:

$$e: \vec{p} = \vec{p}_0 + s*\vec{a} + t*\vec{b}$$

oder in Koordinatenschreibweise:

$$e: \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix} = \begin{pmatrix} p_{0x} \\ p_{0y} \\ p_{0z} \end{pmatrix} + s*\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} + t*\begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$$

$$= \begin{pmatrix} p_{0x} + s*a_x + t*b_x \\ p_{0y} + s*a_y + t*b_y \\ p_{0z} + s*a_z + t*b_z \end{pmatrix}$$

Daraus kann man – wie bei den Geraden – auch drei Gleichungen machen, indem wir x-, y- und z-Komponenten getrennt berechnen:

$$\begin{aligned} e: \quad p_x &= p_{0x} + s*a_x + t*b_x \\ p_y &= p_{0y} + s*a_y + t*b_y \\ p_z &= p_{0z} + s*a_z + t*b_z \end{aligned}$$

Dabei bedeuten:

- $\vec{p}$ – Vektor vom Koordinatenursprung zu dem zu berechnenden Punkt P der Ebene (oder einfach: Punkt P)
- $\vec{p}_0$ – Vektor vom Koordinatenursprung zu einem beliebigen Punkt P_0 der Ebene (oder einfach: Punkt P_0)
- $\vec{a},\vec{b}$ – Beliebige, linear unabhängige Vektoren auf der Ebene
- s,t – Faktoren zur »Verlängerung« (oder Verkürzung) der Vektoren $\vec{a}$ und $\vec{b}$, um Punkt P zu erreichen

Das kleine »e:« bedeutet, daß wir es hier mit einer Ebenengleichung zu tun haben. Der Punkt (oder besser Vektor) $\vec{p}$ steht damit stellvertretend für alle Punkte der Ebene.

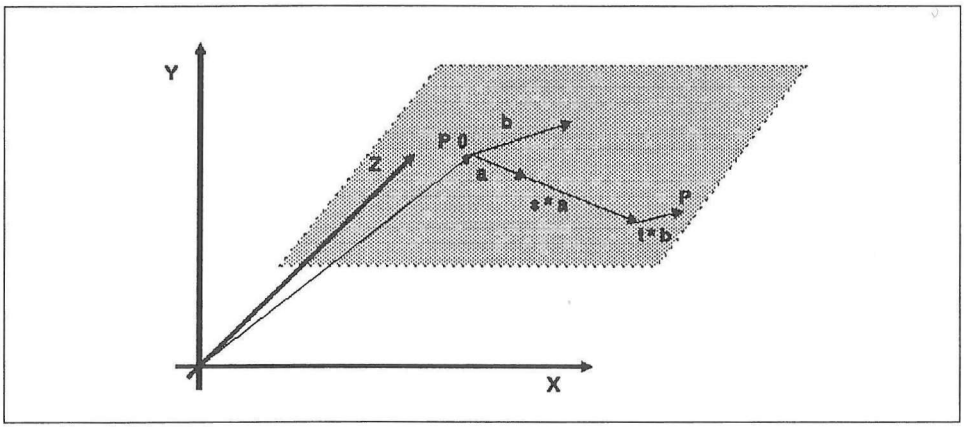


Bild 4.15 Vektorielle Ebenengleichung 1

Ähnlich wie bei der Definition einer Geraden (s.o.) gehen wir von einem beliebigen Startpunkt P_0 auf der Ebene aus. Ausgehend von diesem Punkt ziehen wir zwei Vektoren (s. Vektoraddition), die auf der gewünschten Ebene liegen und linear unabhängig sein müssen. »Linear unabhängig« (s.o.) heißt in diesem Fall, daß die beiden Vektoren nicht parallel oder antiparallel sein dürfen (die Länge spielt dabei keine Rolle). Die beiden Vektoren erreichen nun entsprechend verlängert oder verkürzt jeden beliebigen Punkt auf der Ebene (in etwa vergleichbar mit dem Zeichenarm eines Zeichenbrettes). Die Verlängerung bzw. Verkürzung erreichen wir durch entsprechende Wahl der Faktoren s und t. Die drei resultierenden Vektoren $\vec{p}_0$, $s*\vec{a}$ und $t*\vec{b}$ ergeben dann addiert den Ergebnisvektor $\vec{p}$ zu dem gesuchten Ebenenpunkt.

Um eine ganz bestimmte Ebene zu zeichnen, brauchen wir also lediglich die Parameter $\vec{p}_0$, $\vec{a}$ und $\vec{b}$ anzugeben. Aus beliebigen Werten für s und t errechnen sich dann alle Punkte P (bzw. Vektoren $\vec{p}$) der Ebene.

Diese Form der Ebenendarstellung ist besonders bei Schnittpunktberechnungen allerdings oft nicht besonders günstig, da wir stets mit drei Vektoren ($\vec{p}_0$, $\vec{a}$ und $\vec{b}$) rechnen müssen und meist ein unbequemes Gleichungssystem mit drei Unbekannten und drei Gleichungen erhalten. Dem wird meist durch eine andere Form der Ebenengleichung Rechnung getragen, die mit der Ebenennormalen (also dem senkrechten Vektor zur Ebene) operiert. Sie lautet:

$$e: (P-P_0)*\vec{n} = 0$$

oder eine dieser Gleichungen:

$$\begin{aligned} e:\ (P-P_0)*\vec{n} &= 0 \quad <=> \\ P*\vec{n} - P_0*\vec{n} &= 0 \quad <=> \\ P_0*\vec{n} - P*\vec{n} &= 0 \quad <=> \\ P*\vec{n} &= P_0*\vec{n} \end{aligned}$$

oder in Koordinatenschreibweise:

$$e:\ \begin{pmatrix} p_x - p_{0x} \\ p_y - p_{0y} \\ p_z - p_{0z} \end{pmatrix} * \begin{pmatrix} n_x \\ n_y \\ n_z \end{pmatrix} = 0$$

Da wir es hier mit einer Skalarmultiplikation zu tun haben, ist das Ergebnis von $(P-P_0)*\vec{n}$ eine Zahl. Wir erhalten gemäß der Definition des Skalarproduktes:

$$e:\ (p_x-p_{0x})*n_x + (p_y-p_{0y})*n_y + (p_z-p_{0z})*n_z = 0$$

oder:

$$\begin{aligned} e:\ & p_x*n_x + p_y*n_y + p_z*n_z = p_{0x}*n_x + p_{0y}*n_y + p_{0z}*n_z \quad <=> \\ e:\ & p_x*n_x + p_y*n_y + p_z*n_z = d \end{aligned}$$

wobei:

P	–	zu berechnender Punkt der Ebene
P_0	–	beliebiger Fixpunkt der Ebene
$\vec{n}$	–	Ebenennormale (senkrechter Vektor zur Ebene)
d	–	Kürzel für $d = p_{0x}*n_x + p_{0y}*n_y + p_{0z}*n_z$

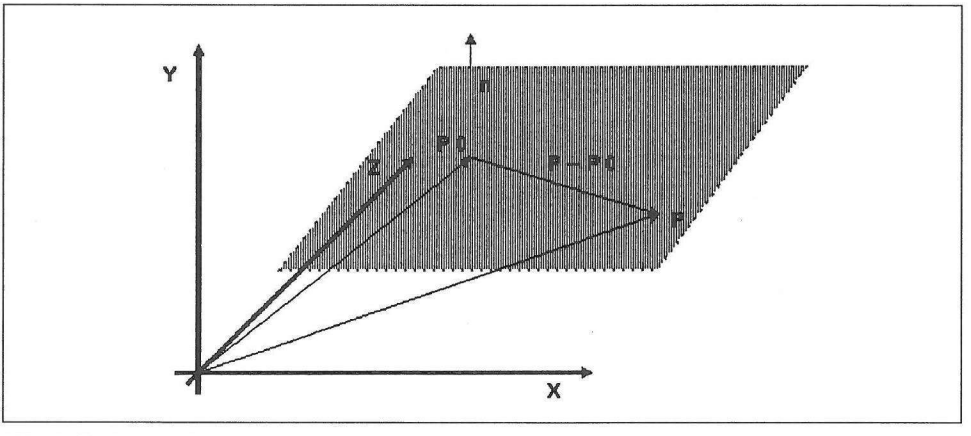


Bild 4.16: Vektorielle Ebenengleichung 2

Und das ist auch schon das Geheimnis dieser Ebenenform: Statt mit drei Gleichungen in der Parameterform haben wir es nur mit einer einzigen zu tun. Um eine bestimmte Ebene anzugeben, benötigen wir nur den Vektor $\vec{n}$ und den Punkt P_0 (das Skalarprodukt $P_0*\vec{n}$ ist

dann d). Ein Punkt P ist nur dann Punkt der Ebene, wenn die Gleichung erfüllt ist: ein ideales Kriterium für Schnittpunktberechnungen! Diese Gleichung ist allerdings nicht dazu geeignet, einen Punkt P der Ebene zu berechnen, da P ja bekannt sein muß, um die Gleichung aufzustellen.

Oft wünschen wir aber auch die alte Vektorform der Ebene, um eben einen Punkt P bestimmen zu können. Wie also können wir die beiden Gleichungen ineinander umformen?

Umrechnung: s-t-Gleichung -> Normalengleichung:

Nun, dazu ermitteln wir erst einmal den Normalenvektor $\vec{n}$. Er berechnet sich aus dem Vektorprodukt $\vec{a} \times \vec{b}$, da er senkrecht auf diesen beiden Vektoren der ersten Ebenengleichung steht:

$$\vec{n} = \vec{a} \times \vec{b} = \begin{pmatrix} a_y * b_z - a_z * b_y \\ a_z * b_x - a_x * b_z \\ a_x * b_y - a_y * b_x \end{pmatrix}$$

Damit hätten wir $\vec{n}$. P_0 bleibt selbstverständlich gleich P_0. s und t werden nicht benötigt, da wir ja die Normale und den Punkt P kennen.

Umrechnung: Normalengleichung -> s-t-Gleichung:

Die umgekehrte Umrechnung ist erheblich komplizierter, weil wir s und t aus der Normalen $\vec{n}$ und dem Punkt P bestimmen müssen. Sind uns die beiden Vektoren $\vec{a}$ und $\vec{b}$ nicht bekannt, dann müssen sie zunächst noch berechnet werden (es gibt ja unendlich viele Kombinationsmöglichkeiten für $\vec{a}$ und $\vec{b}$). Sind sie uns allerdings bekannt, dann sollten wir sie auch übernehmen (die Richtung und die Länge dieser Vektoren können ja für eine bestimmte Aufgabe wichtige Informationen enthalten).

Ermitteln wir also zunächst ein Paar $\vec{a}$ und $\vec{b}$ für den Fall, daß sie uns nicht bekannt sind. Wir wissen, daß beide Vektoren auf $\vec{n}$ senkrecht stehen. Für die Skalarprodukte gilt also:

$\vec{a} * \vec{n} = 0$ und
$\vec{b} * \vec{n} = 0$

und damit:

$a_x * n_x + a_y * n_y + a_z * n_z = 0$ und
$b_x * n_x + b_y * n_y + b_z * n_z = 0$

Da es für die Vektorparameter a_x, a_y etc. unendlich viele Lösungen gibt (außer $\vec{a} = \vec{b}$ oder $\vec{a}$ = Nullvektor oder $\vec{b}$ = Nullvektor), wählen wir eine spezielle aus, die uns am wenigsten Arbeit macht:

Zunächst ein Sonderfall:

Sind genau zwei Parameter von $\vec{n}$ gleich Null, dann setzen wir jeweils einen der entsprechenden Parameter von $\vec{a}$ gleich 1, den anderen gleich Null. Für $\vec{b}$ wählen wir die Parameter genau umgekehrt. Der Parameter von $\vec{n}$, dessen Wert ungleich Null war, sorgt

dafür daß die entsprechenden Parameter von $\vec{a}$ und $\vec{b}$ gleich Null werden (Beispiel: bei $n_x=0$, $n_y=5$, $n_z=0$ setzen wir: $a_x=1$, $a_y=0$, $a_z=0$ und $b_x=0$, $b_y=0$, $b_z=1$: da $n_x=n_z=0$, wird $a_x=b_z=1$ und $a_z=b_x=0$, da $n_y<>0$ ist, werden $a_y=b_y=0$).

Nachdem der obige Sonderfall überprüft wurde, wählen wir zwei Parameter von $\vec{n}$ aus, die nicht gleich Null sind. Nennen wir sie n_1, n_2. Der ausgelassene dritte Parameter, dessen Wert beliebig sein kann, heißt n_0. Die entsprechenden Parameter von $\vec{a}$ und $\vec{b}$ lauten dann: a_1, a_2 und a_0 bzw. b_1, b_2 und b_0. Dann wählen wir:

$$\begin{aligned} a_0 &= b_0 = 0 \\ a_2 &= b_1 = 1 \\ a_1 &= -n_2/n_1 \\ b_2 &= -n_1/n_2 \end{aligned}$$

Damit ist ein Paar $\vec{a}$ und $\vec{b}$ bestimmt.

Die Berechnung von s und t ist ein wenig aufwendiger. Auch hier ist eine Fallunterscheidung durchzuführen. Ausgangspunkt ist die Normalen-Form der Ebene:

$$(P-P_0)*\vec{n} = 0$$

Der Einfachheit halber setzen wir ab jetzt:

$$\vec{v} = P-P_0$$

Erhalten also:

$$\vec{v} * \vec{n} = 0$$

Die s-t-Form der Ebene formen wir so um:

$$\begin{aligned} P &= P_0 + s*\vec{a} + t*\vec{b} \quad <=> \\ s*\vec{a} &= (P-P_0) - t*\vec{b} \quad <=> \\ s*\vec{a} &= \vec{v} - t*\vec{b} \end{aligned}$$

Und in Koordinatenschreibweise:

$$\begin{aligned} s*a_x &= v_x - t*b_x \\ s*a_y &= v_y - t*b_y \\ s*a_z &= v_z - t*b_z \end{aligned}$$

v_x, v_y und v_z sind uns genauso bekannt wie a_x, a_y, a_z, b_x, b_y und b_z. Gesucht sind damit nur die beiden Variablen s und t. Dafür brauchen wir nur zwei Gleichungen, die wir uns aus den dreien aussuchen können (die Suffixe i und j stehen im folgenden für x, y oder für z):

$$\begin{aligned} s*a_i &= v_i - t*b_i \\ s*a_j &= v_j - t*b_j \end{aligned}$$

Da die Vektoren $\vec{a}$ und $\vec{b}$ keine Nullvektoren sein dürfen (damit ist jeweils mindestens ein Parameter von $\vec{a}$ und $\vec{b}$ ungleich Null), wählen wir uns als erste Gleichung diejenige aus, für die sowohl a_i als auch der Ausdruck $b_j*a_i - b_i*a_j$ (s.u.) ungleich Null sind. Das

vereinfacht die ganze Sache erheblich. Sollte das nicht möglich sein, dann liegt irgendwo ein Fehler vor: Die Vektoren $\vec{a}$ und $\vec{b}$ sind nicht linear unabhängig. Wir rechnen:

$$s*a_i = v_i - t*b_i \quad <=>$$
$$s = (v_i - t*b_i)/a_i$$

Das setzen Sie bitte in die zweite Gleichung ein:

$$(v_i - t*b_i)/a_i * a_j = v_j - t*b_j \quad <=>$$
$$v_i*a_j - t*b_i*a_j = v_j*a_i - t*b_j*a_i \quad <=>$$
$$t*(b_j*a_i - b_i*a_j) = v_j*a_i - v_i*a_j \quad <=>$$

$$t = \frac{v_j*a_i - v_i*a_j}{b_j*a_i - b_i*a_j}$$

Den letzten Schritt konnten wir nur deshalb ausführen, da wir bereits oben geprüft haben, ob bj*ai – bi*aj auch wirklich ungleich Null ist. t ist errechnet und braucht nur noch in die Gleichung für s eingesetzt zu werden. Die Ebenengleichung steht!

4.4 Projektionen – aus 3 mach 2

Wenn wir dreidimensionale Objekte auf unserem Bildschirm darstellen wollen, stehen wir vor dem Problem, auf welche Weise es wohl zu realisieren wäre, dort ein zweidimensionales Abbild zu erzeugen, der 3-D-Bildschirm ist schließlich noch nicht erfunden (wenn Techniken wie die Holographie hierzu auch das theoretische Fundament liefern könnten). Also müssen wir uns ein Verfahren ausdenken, das aus den dreidimensionalen Koordinaten eines einzigen Punktes zweidimensionale Koordinaten für den Bildschirm herstellt. Wir haben es mit dem Problem der Projektion zu tun. Sie können sich unter Projektion hier genau dasselbe vorstellen wie die Projektion der dreidimensionalen Welt auf den notwendigerweise zweidimensionalen Film eines Fotoapparates. In unserem Fall spielt nur der Bildschirm die Rolle des Filmes. Was dort sozusagen automatisch vonstatten geht, das müssen wir im Computer erst Punkt für Punkt berechnen.

Diese Analogie mit dem Fotoapparat sollten Sie sich in den folgenden Kapiteln immer wieder vor Augen halten, wenn Sie Probleme mit dem Verständnis haben. Im Prinzip finden nämlich hier wie dort stets ähnliche Prozesse statt.

Was die mathematischen Probleme anbelangt, so kennen wir das notwendige mathematische Rüstzeug für die grafischen Operationen, die wir in diesem Kapitel kennenlernen werden, bereits aus dem letzten Kapitel. Als einziges müssen wir diese Prinzipien auf dreidimensionale Verhältnisse übertragen. Alle Matrizen, Punkt- wie Transformationsmatrizen, werden wir also nur erweitern.

Einen Punkt in einem zweidimensionalen Koordinatensystem können wir auch mit dreidimensionalen Koordinaten (bei denen ja nur die z-Koordinate hinzukommt) ausdrücken. Die z-Koordinate wird dann einfach gleich Null. Die Ebene des zweidimensionalen Koordinaten-

systems liegt also genau in der x-y-Ebene des 3-D-Systems, die durch den Nullpunkt geht. Wir formen also 2-D-Koordinaten wie folgt in 3-D-Koordinaten um:

$P(x,y) = P(x,y,0)$

Es soll sich dabei vereinbarungsgemäß um ein und denselben Punkt handeln. Entsprechend formen wir die zugehörige Punktmatrix P um:

$P = (x\ y) = (x\ y\ 0)$

Und in homogenen Koordinaten (s. 3.2.5, unter »Translation«):

$P = (x*w\ \ y*w\ \ w) = (x*w\ \ y*w\ \ 0*w\ \ w)$

Auch hier also ist die z-Koordinate gleich Null. Allgemein schreiben wir für einen 3-D-Punkt also:

$P(x,y,z)$	– Koordinatenschreibweise
$P = (x\ y\ z)$	– Matrixschreibweise
$P = (x*w\ \ y*w\ \ z*w\ \ w)$	– Matrixschreibweise für homogene Koordinaten

Entsprechend müssen wir die Transformationsmatrizen erweitern. Die allgemeinen Formen für dreidimensionale Verhältnisse werden wir etwas weiter unten angehen. Hier seien nur die Matrizen für zweidimensionale Transformationen auf 3-D erweitert. Alle Matrizen werden um je eine Spalte und eine Reihe vergrößert. Aus der Skalierungsmatrix:

$$S(S_x,S_y) = \begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix}$$

wie wir sie in den 2-D-Kapiteln kennengelernt haben, wird:

$$S(S_x,S_y) = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

und in homogenen Koordinaten:

$$S(S_x,S_y) = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Entsprechend können wir alle anderen Transformationsmatrizen erweitern. Hier die erweiterte Translationsmatrix (rechnen Sie doch einmal nach):

$$T(T_x,T_y) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & 0 & 1 \end{pmatrix}$$

Beachten Sie bitte, daß es sich dabei immer noch um dieselben Transformationen handelt. Das heißt, es wird davon ausgegangen, daß die z-Koordinate stets gleich Null und z.B. eine Vergrößerung oder Verschiebung in z-Richtung nicht erwünscht ist.

Nach der Klärung dieser Dinge also wenden wir uns direkt der angekündigten Projektion dreidimensionaler Koordinaten auf die Ebene zu. Es gibt da mehrere Möglichkeiten, die sich unterschiedlich für die verschiedenen Zwecke eignen. Zwei von diesen Projektionsarten wollen wir hier besprechen:

- Parallelprojektion
- Zentralprojektion (auch perspektivische Projektion genannt)

Die erste Form ist mathematisch recht einfach zu realisieren und eignet sich besonders für maßstabsgerechte Abbildungen, die z.B. in der Architektur Verwendung finden. Die zweite erfordert bereits ein wenig mehr an Rechenaufwand. Es handelt sich dabei aber um die für das menschliche Auge realistischere Darstellung.

4.4.1 Der einfache Weg: Parallelprojektion

Auf dem Wege, dreidimensionale in zweidimensionale Koordinaten umzurechnen, könnten wir auf den Gedanken kommen, die störenden z-Koordinaten einfach fortzulassen, also für alle in Frage kommenden Punkte gleich Null zu setzen. Und in der Tat ist das bereits ein Sonderfall der parallelen Projektion. Sie erhielt ihren Namen aus der Tatsache, daß von allen 3-D-Punkten parallele Strahlen (bzw. Vektoren) bis zur xy-Ebene ($z = 0$) gezogen werden. Diejenigen Punkte, an denen diese Projektionsvektoren die Ebene treffen (schneiden), stellen die sogenannten Projektionspunkte dar. Für sie ist die z-Koordinate gleich Null und kann damit zugunsten eines ebenen Koordinatensystems weggelassen werden.

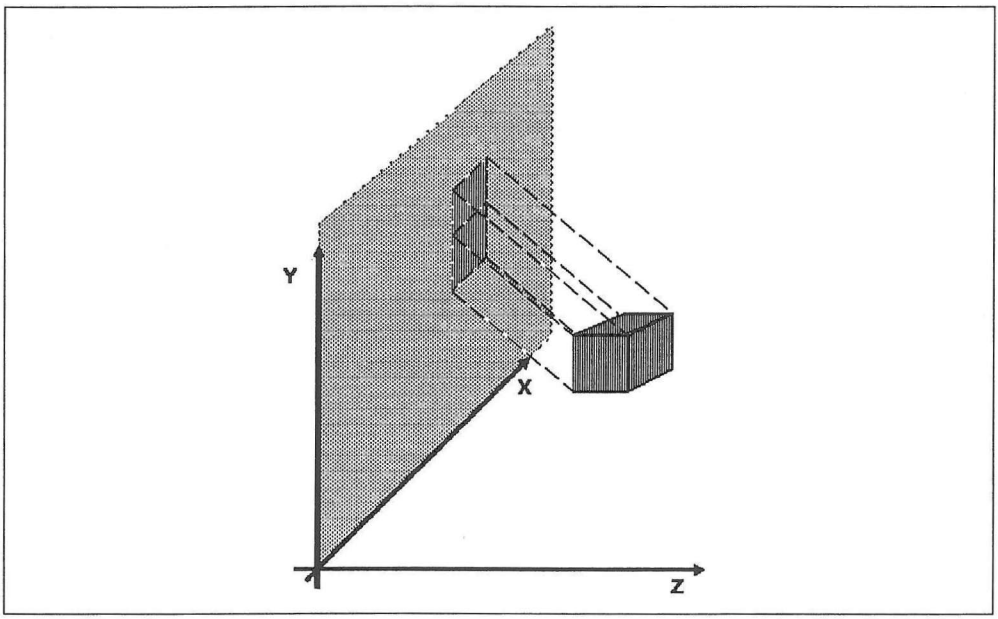


Bild 4.17: Konstruktion der Parallelprojektion

In dem angesprochenen Sonderfall, bei dem die z-Koordinaten eines jeden Punktes einfach gleich Null gesetzt werden, verlaufen die Projektionsvektoren zusätzlich noch parallel zur z-Achse, so daß sich x- und y-Koordinaten bei der Projektion nicht ändern, egal, wie groß die z-Koordinate war. Wir haben es hier mit einer Form der sogenannten orthografischen Projektion zu tun, in die Projektionsvektoren senkrecht (= orthogonal) zur Projektionsebene stehen.

Es gibt noch einige andere parallele Projektionen, die z.B. in technischen Zeichnungen Anwendung finden (z.B.: Kavalier- oder Kabinettprojektionen etc.). Bei ihnen liegt die Projektionsebene nicht immer in der xy-Ebene. Hier können auch xz-, yz-Ebenen oder beliebige andere Ebenen eingesetzt werden. Entsprechend werden sich natürlich die mathematischen Ableitungen von der nun folgenden unterscheiden. Meist jedoch handelt es sich um unwesentliche Änderungen, die leicht und schnell durchzuführen sind.

Die folgende Ableitung geht davon aus, daß die xy-Ebene die Projektionsebene darstellt. Die Projektionsvektoren – wir sprechen von Stund an nur noch von **dem** Projektionsvektor, da ja eigentlich nur seine Richtung eine Rolle spielt – sind dagegen beliebig, sofern sie nicht parallel zur Ebene verlaufen.

Rufen wir uns die vektorielle Form der Geradengleichung ins Gedächtnis:

$$\vec{P} = \vec{P}_0 + s*\vec{a}$$

Angenommen, unser Projektionsvektor $\vec{v}$ (also der Richtungsvektor aller Linien vom 3-D-Punkt zur Projektionsebene) lautet:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

Der zu projizierende 3-D-Punkt besitzt die Koordinaten:

$$\vec{Q} = \begin{pmatrix} Q_x \\ Q_y \\ Q_z \end{pmatrix}$$

Dann lautet die Gleichung der Geraden vom 3-D-Punkt bis zum Punkt auf der Projektionsebene:

$$\vec{P} = \vec{Q} + s*\vec{v}$$

oder in Parameterform:

$$P_x = Q_x + s*v_x$$
$$P_y = Q_y + s*v_y$$
$$P_z = Q_z + s*v_z$$

$\vec{P}$ ist dabei der gesuchte Punkt auf der Projektionsebene, der Punkt also, bei dem die Gerade die Ebene schneidet. Wir müssen nun also nur noch herausbekommen, welchen Wert s annehmen muß. Dann könnten wir $\vec{P}$ ausrechnen. Unsere Ebene soll ja vereinbarunsgemäß die xy-Ebene sein. Die z-Koordinate des Projektionspunktes auf der Ebene wird also gleich Null sein. Wenn Pz aber gleich Null ist, dann erhalten wir aus der dritten Parametergleichung:

$$0 = Q_z + s * v_z \quad <=>$$

$$s = -\frac{Q_z}{v_z}$$

Das können wir wiederum in die beiden ersten Gleichungen einsetzen und erhalten nach ein paar Umstellungen die zentrale Formel für die parallele Projektion:

$$P_x = Q_x - Q_z*(v_x/v_z)$$
$$P_y = Q_y - Q_z*(v_y/v_z)$$

mit:

P_x, P_y	–	Koordinaten des projizierten Ebenenpunktes
Q_x, Q_y, Q_z	–	Koordinaten des zu projizierenden Raumpunktes
v_x, v_y, v_z	–	Projektionsvektor

Daraus bestimmen wir die folgende Transformationsmatrix:

$$PP(v_x,v_y,v_z) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -v_x/v_z & -v_y/v_z \end{pmatrix}$$

oder in homogenen Koordinaten mit der Transformation eines Punktes $Q(Q_x,Q_y,Q_z)$:

$$(P_x \; P_y \; 0 \; 1) = (Q_x \; Q_y \; Q_z \; 1) * \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -v_x/v_z & -v_y/v_z & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In Computerprogrammen wird die parallele Projektion oft in Verbindung mit der Rotation um die x-Achse angewandt. Dabei gibt man dem Projektionsvektor ein möglichst einfaches Aussehen. Etwa eine Parallele zur z-Achse mit der Länge 1, wie wir es hier unternehmen:

$$\vec{v} = \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}$$

Die Transformationsmatrix PP(0,0,–1) vereinfacht sich dadurch natürlich rapide (rechnen Sie einmal nach). Die Folge: Es tritt wieder der sehr triviale Fall ein, bei dem x- und y-Koordinaten übernommen werden, die z-Koordinate einfach verschwindet (s.o.). Vor der Projektion wird dann allerdings das gesamte Bild um die y- und um die x-Achsen (möglich wäre natürlich noch eine Drehung um die z-Achse) rotiert. Damit hat die ursprüngliche z-Koordinate doch wieder Einfluß auf die resultierenden Projektionskoordinaten. Die Drehwinkel und die entsprechenden Winkelfunktionen werden zu Anfang der Projektion einmal berechnet und fortan stets als Konstanten in die Rechnung übernommen. Der Vorteil dieser Methode: Die Drehwinkel stellen praktisch den Sichtwinkel dar, mit dem der Betrachter auf das Bild schaut. Winkel sind in Programmen sehr viel leichter zu handhaben als solch unanschauliche Dinge wie ein Projektionsvektor. Da wir erst später auf Drehungen im Raum zu sprechen kommen, sei hier die resultierende Transformationsmatrix ohne Ableitung angegeben:

$$\vec{P} = \vec{Q} * R_y(b) * R_x(a) * PP(0,0,-1) =$$

$$(P_x \; P_y \; 0 \; 1) =$$

$$(Q_x \; Q_y \; Q_z \; 1) * \begin{pmatrix} \cos(b) & \sin(a)*\sin(b) & 0 & 0 \\ 0 & \cos(a) & 0 & 0 \\ \sin(b) & -\sin(a)*\cos(b) & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

und in Parameterform:

$$P_x = Q_x * \cos(b) + Q_z * \sin(b)$$
$$P_y = Q_x * \sin(a) * \sin(b) + Q_y * \cos(a) - Q_z * \sin(a) * \cos(b)$$

dabei bedeuten:

$P\ (P_x,P_y)$	–	Resultierender Punkt in der Ebene
$Q\ (Q_x,Q_y,Q_z)$	–	Ausgangspunkt im Raum
$R_y(b)$	–	Rotationsmatrix: Rotation um y-Achse mit dem Winkel b
$R_x(a)$	–	Rotationsmatrix: Rotation um x-Achse mit dem Winkel a
$PP(0,0,-1)$	–	Transformationsmatrix: Parallelprojektion mit Projektionsvektor $\vec{v} = (0\ 0\ -1)$
a	–	Drehwinkel um x-Achse
b	–	Drehwinkel um y-Achse

Man kann die Drehungen des Objektes auch als Drehung der Projektionsebene ansehen, die dann natürlich in die entgegengesetzte Richtung erfolgt.

Die oben angegebene Parameterform ist genau die Formel, die wir im folgenden Programm verwenden werden, um ein dreidimensionales Objekt auf den Bildschirm zu projizieren. Bitte schauen Sie sich das Programm ruhig einmal genauer an. Hier werden Sie das erste Mal sehen, auf welche Weise die viele graue Theorie, die wir uns bis jetzt angeeignet haben, in die Praxis umgesetzt werden kann:

```
'*************************
'**                     **
'**  Parallelprojektion **
'**                     **
'*************************

PI = 3.141593

'Translation der Ebenenkoordinaten:
'(Position des Koordinaten-Nullpunktes)
maus% = MOUSE(0)
v1    = MOUSE(3)  'Mauskoordinaten
v2    = MOUSE(4)  'holen
```

```
'Start-Rotationswinkel der Projektionsebene (Grad):
ag = 10             'Drehung um x-Achse
bg = -10            'Drehung um y-Achse

a = PI * ag/180     'Umrechnung nach RAD
b = PI * bg/180

'Drehinkrementwert (Grad):
dig = 5

di = PI * dig/180   'Umrechnung nach RAD

'Einen neuen Bildschirm mit Fenster öffnen:
SCREEN 2,320,200,2,1
WINDOW 2,"3-D-Parallelprojektion",(0,10)-(297,186),1+2+8,2

PALETTE 0,0,0,0      'Hintergrund: schwarz
PALETTE 2,1,.6,.67   'Farbe 1: kirschrot
PALETTE 3,1,1,.13    'Farbe 2: gelb
PALETTE 1,.4,.6,1    'Farbe 3: dunkelblau
COLOR ,0             'Hintergrundfarbe = 0

'Objekt-Daten in Arrays einlesen:
'xr%(), yr%(), zr%() - Koordinaten der Raumpunkte
'ls%(), le%()        - Linienverbindungen
get.objects

'Hauptschleife:
'**************

flag%=0
WHILE flag%<>1

  projektion        'Alle Punkte projizieren

  CLS               'Fenster löschen

  'Koordinatenachsen einzeichnen:
  PATTERN &HAAAA    'Linienmuster = gestrichelt
  COLOR 2
  FOR i=0 TO 2
    LINE (xe%(ls%(i)),ye%(ls%(i)))-(xe%(le%(i)),ye%(le%(i)))
  NEXT i

  PATTERN &HFFFF    'Linienmuster = durchgezogen
  COLOR 3
  'Linien ausgeben (Objekt zeichnen):
  FOR i=3 TO al%-1
    LINE (xe%(ls%(i)),ye%(ls%(i)))-(xe%(le%(i)),ye%(le%(i)))
```

```
  NEXT i
  maus% = 0 : flag%=0
  WHILE maus%<>1 AND flag%=0
    SLEEP           'Auf Ereignis warten

    'Mauskoordinaten als neue Nullpunkt-
    'Koordinaten übernehmen:
    maus%=MOUSE(0)

    IF maus%=1 THEN
      v1=MOUSE(3)
      v2=MOUSE(4)
    END IF

    c% = ASC(INKEY$+CHR$(0))
    IF c%=31 THEN  'Cursor links =>
      b=b+di       'y-Achsendrehwinkel erhöhen
      flag% = -1   'Flag für Neuzeichnen
    END IF
    IF c%=30 THEN  'Cursor rechts =>
      b=b-di       'y-Achsendrehwinkel erniedrigen
      flag% = -1   'Flag für Neuzeichnen
    END IF
    IF c%=28 THEN  'Cursor hoch =>
      a=a+di       'x-Achsendrehwinkel erhöhen
      flag% = -1   'Flag für Neuzeichnen
    END IF
    IF c%=29 THEN  'Cursor runter =>
      a=a-di       'x-Achsendrehwinkel erniedrigen
      flag% = -1   'Flag für Neuzeichnen
    END IF

    'Falls Fenster geschlossen -> Ende
    IF WINDOW(7)=0 THEN
      flag%=1
    END IF

  WEND
WEND

WINDOW OUTPUT 1
SCREEN CLOSE 2

END
```

```
'Objektdaten einlesen:
SUB get.objects STATIC

  SHARED ap%

  READ ap%
  DIM SHARED xr%(ap%),yr%(ap%),zr%(ap%)
  DIM SHARED xe%(ap%),ye%(ap%)

  FOR i=0 TO ap%-1
    READ xr%(i),yr%(i),zr%(i)
  NEXT i

  'Liniendefinitionen einlesen:
  SHARED al%

  READ al%
  DIM SHARED ls%(al%),le%(al%)

  FOR i=0 TO al%-1
    READ ls%(i),le%(i)
  NEXT i

END SUB


'Projektion aller Raum-Koordinaten in die Ebene:
SUB projektion STATIC
  SHARED ap%
  SHARED a,b,v1,v2

  'Konstanten für die Projektion berechnen:
  si.a=SIN(a)
  co.a=COS(a)
  si.b=SIN(b)
  co.b=COS(b)

  FOR i=0 TO ap%-1

    Qx = xr%(i)
    Qy = yr%(i)
    Qz = -zr%(i)  'ins Linkssystem umwandeln

    xe%(i) = v1 + Qx*co.b + Qz*si.b
    ye%(i) = v2 - (Qx*si.a*si.b + Qy*co.a - Qz*si.a*co.b)
```

```
  NEXT i
END SUB



'Objektdaten:
'************

'3-D-Punktkoordinaten:

'Anzahl der Punkte:
DATA 36

'Koordinatenachsen:
DATA -10,  0,  0,   80, 0,  0
DATA   0,-10,  0,    0,90,  0
DATA   0,  0,-10,    0, 0, 180

'Objektpunkte:
DATA  5,10, 5,   5,45, 5,  15,65, 5
DATA 25,45, 5,  25,10, 5,   5,10,55
DATA  5,45,55,  15,65,55,  25,45,55
DATA 25,10,55,  13,10, 5,  13,20, 5
DATA 17,20, 5,  17,10, 5,   7,33, 5
DATA  7,38, 5,  12,38, 5,  12,33, 5
DATA 18,33, 5,  18,38, 5,  23,38, 5
DATA 23,33, 5,  25,33,12,  25,38,12
DATA 25,38,23,  25,33,23,  25,33,32
DATA 25,38,32,  25,38,43,  25,33,43


'Linienverbindungen:

'Anzahl der Linien:
DATA 40

'Koordinatenachsen:
DATA  0, 1,   2, 3,   4, 5

'Objektlinien:
DATA  6, 7,   7, 8,   8, 9,   9,10
DATA 10, 6,   7, 9,  11,12,  12,13
DATA 13,14,  14,15,  15,11,  12,14
DATA  6,11,   7,12,   8,13,   9,14
DATA 10,15,  16,17,  17,18,  18,19
```

```
DATA 19,16,  20,21,  21,22,  22,23
DATA 23,20,  24,25,  25,26,  26,27
DATA 27,24,  28,29,  29,30,  30,31
DATA 31,28,  32,33,  33,34,  34,35
DATA 35,32
```

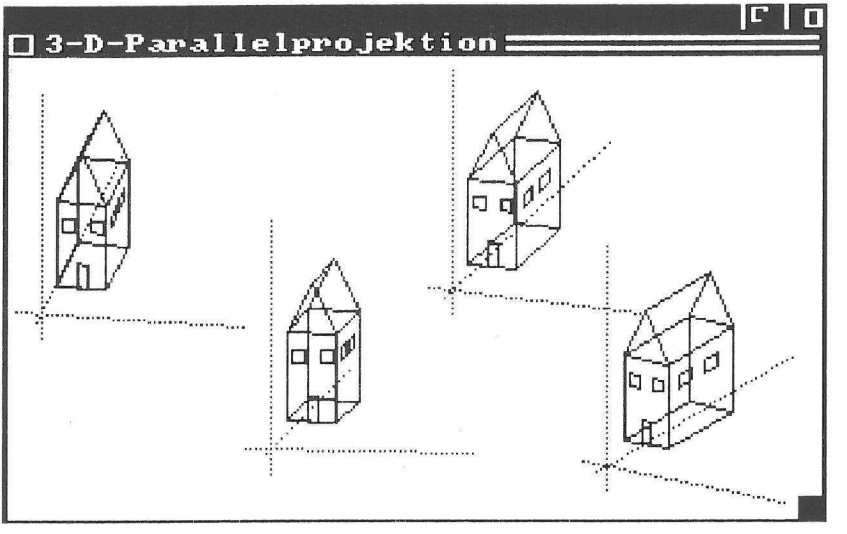


Bild 4.18: Parallelprojektion

Da steht es nun, unser erstes 3-D-Programm. Starten Sie es erst einmal und bewundern Sie unser Erzeugnis, das mit bisher einfachsten Mitteln auf den Bildschirm gezaubert wurde. Sie sehen ein Haus und ein Koordinatenkreuz. Sie befinden sich in einem neuen Screen niedrigster Auflösung – in einem Fenster. Sie fahren mit der Maus auf einen beliebigen Punkt des Fensters und betätigen den Mausknopf. Sie staunen? Das Haus samt Koordinatenkreuz wird gelöscht und an der von Ihnen angewählten Stelle neu gezeichnet. Sie betätigen eine Cursortaste. Sie staunen noch mehr? Das Objekt dreht sich! Sie möchten wissen, wie das funktioniert? Das wissen Sie bereits!

Wir haben lediglich einmal aus unserem bisherigen Wissen geschöpft und das ist das Ergebnis. Nun, es folgt eine Aufstellung der Manipulationsmöglichkeiten bei diesem Programm:

linke Maustaste – Positionierung des 3-D-Koordinatensystems
Cursor li/re – Drehung der Projektionsebene um die y-Achse
Cursor ob/un – Drehung der Projektionsebene um die x-Achse

Sie können sich das dargestellte Haus also von allen Seiten anschauen, indem Sie die Projektionsebene nach und nach mit Hilfe der Cursortasten drehen. Sollte sich das Objekt nach dem Starten des Programmes außerhalb des Fensters befinden, dann positionieren Sie es einfach mittels Maustaste.

Schauen wir doch einmal, wie das Programm dazu kommt, solche Dinge auf den Bildschirm zu zaubern. Zunächst werden – wie immer – einige grundsätzliche Variablen vorbelegt. Es handelt sich dabei um die Koordinaten des 3-D-Systems (v1 und v2), die Startwerte für die Drehung um die beiden Rotationsachsen (a und b), sowie den Wert di, den die Drehwinkel hoch- oder herunterzählen werden, sobald Sie eine Cursortaste betätigen. Hier sind natürlich alle Möglichkeiten der Änderung offen. Beachten Sie, daß alle Werte letztendlich in Radiant vorliegen müssen. Die Grad-Angaben rechnet dieses Programm deshalb auch sofort um.

Dann geht es auch schon zur Sache: Ein neuer Bildschirm wird geöffnet (Auflösung: 320x200, 2 Farbebenen = 4 Farben) und in ihm ein Fenster mit Schließ-Gadget, Sizer-Gadget und Move-Balken. Anschließend ordnen wir den 4 verfügbaren Farbregistern die entsprechenden Farbmischungen zu.

Nach diesen Formalitäten müssen die Daten für das zu zeichnende Objekt eingelesen werden. Die gestellte Aufgabe übernimmt die Routine »get.objects«. Hier definieren wir eine Reihe von Arrays, die objektspezifische Daten enthalten:

xr%( ),yr%( ),zr%( )	–	Raumkoordinaten aller Objektpunkte
xe%( ),ye%( )	–	Projizierte Ebenenkoordinaten aller Objektpunkte
ls%( ),le%( )	–	Nummern der Start- und Endpunkte aller Objektlinien

Gefüllt werden dann aber nur die Raumpunkt- und die Linien-Arrays. In die Raumpunkt-Arrays werden aus den am Programmende zu findenden DATA-Zeilen alle Eckpunkte des Objektes übernommen. Erinnern Sie sich vielleicht noch an die Ausführungen über die Datenstrukturen in einem CAD-Programm? Hier finden wir einen Teil realisiert. Unsere Welt besteht praktisch nur aus einem Objekt, dieses Objekt seinerseits aus vielen Linien (die Flächen haben wir hier ausgelassen). Die Linien wiederum werden durch ihre Endpunkte definiert. 36 Punkte reichen uns, um Koordinatensystem sowie Haus Eckpunkt für Eckpunkt darzustellen. 40 Verbindungen dieser Punkte speichern wir im zweiten DATA-Zeilensatz. Angegeben werden dabei pro Linie zwei Werte: die Nummer des Start- und die des Endpunktes. Wie wir ein solches Gebilde auf dem Papier entwerfen und die vielen verschiedenen Punkte systematisch ablesen und verbinden, das soll ein paar Absätze später gezeigt werden. Wollen wir mehrere Objekte einzeichnen, so können wir die bisher eindimensionalen Punkte- und Linien-Arrays noch um eine zusätzliche Dimension erweitern. Sie gibt dann an, zu welchem Objekt der jeweilige Punkt (die Linie) gehört.

Wir sollten uns aber weiter mit dem Programm beschäftigen. Punkte und Linien befinden sich nun also in den entsprechenden Arrays und die Routine »get.objects« kehrt zurück ins Hauptprogramm. Hier startet sofort eine große WHILE...WEND-Schleife, in der der gesamte Zeichen- und Eingabeprozeß stattfindet. Doch noch gibt es nichts zu zeichnen. Die Punkte unseres Objektes liegen uns bisher nur in dreidimensionaler Form vor. Was fehlt, ist die Projektion auf die Ebene.

Was nicht ist, das wird – in der Routine »projektion«. Hier sollte Ihnen bereits einiges bekannt vorkommen, wenn Sie noch einmal an den Anfang dieses Kapitels zurückdenken. Was hier getan wird, ist einfach die Anwendung der Formel für die Parallelprojektion mit den Drehungen der Projektionsebene um die x- und die y-Achsen. Sogar die Variablenbezeichnungen stimmen überein. Hier wird also Punkt für Punkt (die Anzahl der Punkte steht in der Variablen ap%)

auf die Ebene projiziert und in den Arrays xe%( ) und ye%( ) niedergelegt. Die Translationsvariablen v1 und v2 sind für eine einfache Verschiebung in der Ebene zuständig, wie wir sie bereits kennengelernt haben. Die Rückkehr ins Hauptprogramm folgt auf dem Fuße.

Da wir nun die ebenen Koordinaten jedes einzelnen Punktes kennen und frei in zwei Arrays verfügbar haben, können wir darangehen, das Ganze zu Bildschirm zu bringen. Da wird also zunächst das Koordinatenkreuz gezeichnet. Dafür werden die Farbe und die Linienform geändert. In einer Schleife geben wir dann jede der drei Achsen aus. Für das eigentliche Objekt ändern wir wiederum die Farbe und normalisieren die Linienform.

Die Linien des Objektes geben wir mit einem normalen LINE-Befehl aus:

```
LINE (xe%(ls%(i)),ye%(ls%(i)))-(xe%(le%(i)),ye%(le%(i)))
```

Die Start- und Endkoordinaten der Linien ermitteln sich wie folgt: Die Laufvariable i gibt an, um welche Linie es sich handelt. Sie dient als Index für die Arrays ls%( ) und le%( ). Die daraus ermittelten Werte sind bekanntlich die Nummern der jeweiligen Endpunkte und werden ihrerseits als Index für die Arrays xe%( ) und ye%( ) verwendet, die die gewünschten Ebenenkoordinaten enthalten.

Der Rest des Programmes ist eigentlich recht einfach zu verstehen. In einer inneren WHILE...WEND-Schleife wartet das Programm auf ein Ereignis (SLEEP) und ermöglicht in dieser Zeit Multitasking. Ist das Ereignis erfolgt, muß festgestellt werden, um welche Art von Ereignis es sich handelt. Leider bietet Amiga-Basic hierzu keine kompakte und komfortable Funktion, wie wir sie unter C kennengelernt haben. Wir müssen also alle Möglichkeiten, die uns interessieren, abklappern und entsprechend reagieren. Das Programm selbst dokumentiert Ihnen das ausreichend. Eine kleine Anmerkung vielleicht noch: Wenn Sie das Schließ-Gadget anwählen, dann schließt Amiga-Basic das Fenster automatisch, ohne dem Basic-Programm dies mitzuteilen oder »Rückfrage« zu halten. Aus diesem Grunde mußten wir einen kleinen Trick anwenden. Sobald nämlich das letzte Fenster des Bildschirms geschlossen ist, wird der Inhalt von WINDOW(7) gleich Null (Zeiger auf die dann ja nicht mehr vorhandene Window-Struktur des aktuellen Fensters). Wir reagieren entsprechend darauf.

Jetzt ist es an der Zeit, durchzusprechen, auf welche Weise Sie ein eigenes Objekt per Hand zeichnen und die Punktkoordinaten etc. ermitteln können. Schließlich eignet sich das vorliegende Programm hervorragend dazu, praktisch beliebige und beliebig komplizierte Objekte zu zeichnen, ohne etwas am Programm selbst zu ändern, indem wir andere DATA-Zeilen kreieren.

Nehmen wir das Beispiel unseres kleinen Hauses. Das Problem ist nun, daß wir nur sehr schwer dreidimensionale Koordinaten von einem auf die Ebene projizierten Gegenstand ermitteln können. Aus diesem Grunde müssen wir das Problem reduzieren, indem wir einfach mehrere Ansichten des gewünschten Objektes erstellen. Die Anzahl dieser Ansichten ist abhängig von der Komplexität des jeweiligen Objektes und von Ihrer räumlichen Vorstellungsfähigkeit. Bei unserem einfachen Haus käme man mit zwei Ansichten (eine von vorne, eine von der Fensterseite) aus. Das Optimale wären allerdings vier Ansichten, von jeder Seite eine (s. Bild 4.19).

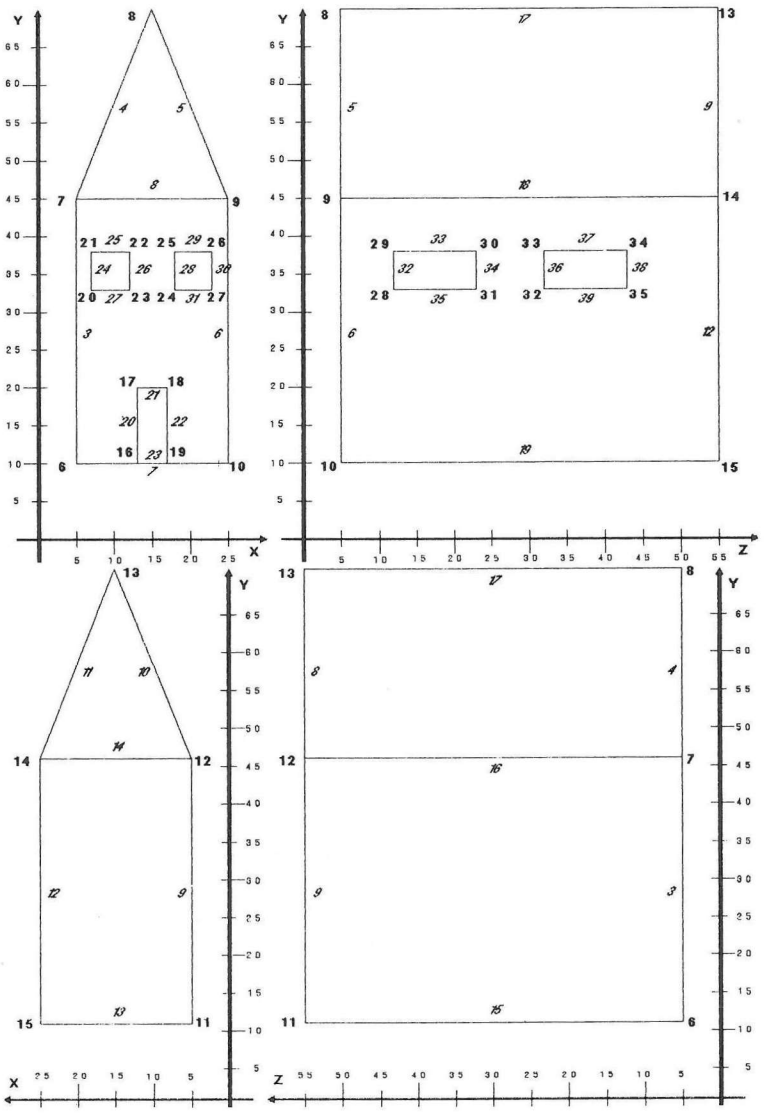


Bild 4.19: Vier Ansichten des Hauses

Jede Ansicht wird frontal genommen (also keine Betrachtung unter einem bestimmten Winkel, d.h. Projektionsebene und Projektionsvektor parallel zu den Achsen). Gleichzeitig haben wir ein Koordinatensystem einzuzeichnen (x-y, x-z oder y-z). Bei mehreren Ansichten müssen wir darauf achten, daß die Einheiten aller gleichnamigen Achsen und die Position zum Objekt übereinstimmen.

Der nächste Schritt ist die Numerierung aller Linien-Endpunkte, beginnend bei null (in unserem Programm wird das Koordinatensystem mit eingezeichnet. Also hier: beginnend bei sechs). Anschließend numerieren wir genauso die vielen verschiedenen Linien (am besten mit einer anderen Farbe). (In unserem Beispiel beginnen wir bei drei, da die drei Koordinatenachsen berücksichtigt werden müssen.) Später kommen noch die Flächen hinzu. Achten Sie darauf, daß Sie zwei identische Punkte in verschiedenen Ansichten auch identisch numerieren. Ansonsten hätten Sie viel zu viele überflüssige Eckpunkte!

Sind Sie soweit, dann stellen Sie zwei Tabellen auf. Die erste enthält alle Punkte mit ihren dreidimensionalen Koordinaten. Die zweite listet alle Linien auf. In ihr stehen die Nummern derjenigen Punkte, die jeweils eine Linie bilden. Jeder Punkt kann so oft auftauchen, wie Sie wünschen.

Die entsprechenden Tabellen zu dem Haus könnten dann wie folgt aussehen:

Punkttabelle:

Nr	x	y	z	Nr	x	y	z	Nr	x	y	z
6	5	10	5	16	13	10	5	26	23	38	5
7	5	45	5	17	13	20	5	27	23	33	5
8	15	65	5	18	17	20	5	28	25	33	12
9	25	45	5	19	17	10	5	29	25	38	12
10	25	10	5	20	7	33	5	30	25	38	23
11	5	10	55	21	7	38	5	31	25	33	23
12	5	45	55	22	12	38	5	32	25	33	32
13	15	65	55	23	12	33	5	33	25	38	32
14	25	45	55	24	18	33	5	34	25	38	43
15	25	10	55	25	18	38	5	35	25	33	43

Linientabelle:

Nr	P1	P2	Nr	P1	P2	Nr	P1	P2	Nr	P1	P2
3	6	7	13	15	11	23	19	16	33	29	30
4	7	8	14	12	14	24	20	21	34	30	31
5	8	9	15	6	11	25	21	22	35	31	28
6	9	10	16	7	12	26	22	23	36	32	33
7	10	6	17	8	13	27	23	20	37	33	34
8	7	9	18	9	14	28	24	25	38	34	35
9	11	12	19	10	15	29	25	26	39	35	32
10	12	13	20	16	17	30	26	27			
11	13	14	21	17	18	31	27	24			
12	14	15	22	18	19	32	28	29			

Vergleichen Sie das bitte mit den DATA-Zeilen im besprochenen Programm. Wenn Sie solche Objekte des öfteren entworfen haben, dann werden Sie sicherlich nach einem geeigneten 3-D-Editor schauen, mit dem das Ganze viel einfacher zu handhaben wäre. Doch lassen Sie uns erst einmal weitersehen, damit sich ein solches Unterfangen auch lohnt.

Ein kleiner Zusatz zum Ausprobieren: Versuchen Sie doch einmal, die folgenden DATA-Zeilen in das obige Programm einzusetzen (vergessen Sie die Daten für das Koordinatenkreuz nicht):

```
' Punkte:
' *******
'
' Anzahl Punkte:
DATA 170
'
' Koordinaten:
DATA   1,  7, 12,    1,  5, 12,    1,  5, 10,    1,  7, 10,    1,  7,  4
DATA   1,  5,  4,    1,  5,  2,    1,  7,  2,    1,  6,  9,    1,  2,  9
DATA   1,  2,  5,    1,  6,  5,    1,  6,  6,    1,  6,  7,    1,  6,  8
DATA   1,  2,  6,    1,  2,  7,    1,  2,  8,    1,  4, 14,    1,  3, 14
'
DATA   1,  3, 13,    1,  4, 13,    1,  4,  1,    1,  3,  1,    1,  3,  0
DATA   1,  4,  0,    1,  8, 14,    1,  2, 14,    1,  2,  0,    1,  8,  0
DATA  14, 12,  3,   14, 12, 11,    2,  8,  7,    2,  9,  6,    2, 10,  7
DATA   2,  9,  8,    8,  9,  1,    8,  9, 13,    8,  9, 11,   10, 10,  8
'
DATA   8,  9,  6,   10, 10,  3,    6,  8, 13,    8, 13, 13,    4,  2, 14
DATA   4,  2,  0,    4,  5, 14,    4,  5,  0,   10,  5, 14,   10,  5,  0
DATA  10,  2, 14,   10,  2,  0,   22,  2, 14,   22,  2,  0,   22,  5, 14
DATA  22,  5,  0,   28,  5, 14,   28,  5,  0,   28,  2, 14,   28,  2,  0
'
DATA  31,  2, 14,   31,  2,  0,   31,  2,  3,   32,  2,  3,   31,  6, 14
DATA  31,  6,  0,   27, 12, 11,   27, 12,  3,    5,  3, 13,    5,  3, 10
DATA   5,  3,  4,    5,  3,  1,    5,  1, 13,    5,  1, 10,    5,  1,  4
DATA   5,  1,  1,    6,  0, 13,    6,  0, 10,    6,  0,  4,    6,  0,  1
'
DATA  10,  8, 14,   10,  8,  0,   19, 12, 11,   19, 12,  3,    8,  0, 13
DATA   8,  0, 10,    8,  0,  4,    8,  0,  1,    9,  1, 13,    9,  1, 10
DATA   9,  1,  4,    9,  1,  1,    9,  3, 13,    9,  3, 10,    9,  3,  4
DATA   9,  3,  1,    8,  4, 13,    8,  4, 10,    8,  4,  4,    8,  4,  1
'
DATA   6,  4, 13,    6,  4, 10,    6,  4,  4,    6,  4,  1,   23,  3, 13
DATA  23,  3, 10,   23,  3,  4,   23,  3,  1,   23,  1, 13,   23,  1, 10
DATA  23,  1,  4,   23,  1,  1,   24,  0, 13,   24,  0, 10,   24,  0,  4
DATA  24,  0,  1,   19,  2, 14,   19,  2,  0,   19,  7, 14,   19,  7,  0
'
```

```
DATA  26,  0, 13,   26,  0, 10,   26,  0,  4,   26,  0,  1,   27,  1, 13
DATA  27,  1, 10,   27,  1,  4,   27,  1,  1,   27,  3, 13,   27,  3, 10
DATA  27,  3,  4,   27,  3,  1,   26,  4, 13,   26,  4, 10,   26,  4,  4
DATA  26,  4,  1,   24,  4, 13,   24,  4, 10,   24,  4,  4,   24,  4,  1
'
DATA   7,  2, 13,    7,  2,  1,   25,  2, 13,   25,  2,  1,   18,  7, 14
DATA  18,  7,  0,   18,  6, 14,   18,  6,  0,   15, 12, 11,   15, 12,  3
DATA  18, 12, 11,   18, 12,  3,   18,  8, 14,   18,  8,  0,   20, 12, 11
DATA  20, 12,  3,   20,  8, 14,   20,  8,  0,   28,  8, 14,   28,  8,  0
'
DATA  26, 12, 11,   26, 12,  3,   19,  8, 14,   19,  8,  0,   28, 10, 11
DATA  30,  7, 12,   30,  7,  2,   28, 10,  4,    6,  8, 14,    6,  8,  0
'
'
' Linien:
' *******
'
' Anzahl Linien:
DATA 192
'
' Verbindungen:
DATA   0,  1,    1,  2,    2,  3,    3,  0,    4,  5
DATA   5,  6,    6,  7,    7,  4,    8,  9,    9, 10
DATA  10, 11,   11,  8,   12, 15,   13, 16,   14, 17
DATA  18, 19,   19, 20,   20, 21,   21, 18,   22, 23
'
DATA  23, 24,   24, 25,   25, 22,   26, 27,   27, 28
DATA  28, 29,   29, 26,  158, 64,   30, 31,  159, 65
DATA  32, 34,   33, 35,   36, 37,   38, 39,   40, 41
DATA  42, 43,   27, 44,   44, 46,   46, 48,   48, 50
'
DATA  50, 52,   52, 54,   54, 56,   56, 58,   58, 60
DATA  60, 64,   64, 66,   66, 31,   31,168,  168, 26
DATA  28, 45,   45, 47,   47, 49,   49, 51,   51, 53
DATA  53, 55,   55, 57,   57, 59,   59, 61,   61, 65
'
DATA  65, 67,   67, 30,   30,169,  169, 29,   66, 67
DATA  64, 65,   60, 61,   68, 72,   72, 76,   76, 84
DATA  84, 88,   88, 92,   92, 96,   96,100,  100, 68
DATA  69, 73,   73, 77,   77, 85,   85, 89,   89, 93
'
DATA  93, 97,   97,101,  101, 69,   70, 74,   74, 78
DATA  78, 86,   86, 90,   90, 94,   94, 98,   98,102
DATA 102, 70,   71, 75,   75, 79,   79, 87,   87, 91
DATA  91, 95,   95, 99,   99,103,  103, 71,  140,141
```

```
'
DATA 104,108,  108,112,  112,120,  120,124,  124,128
DATA 128,132,  132,136,  136,104,  105,109,  109,113
DATA 113,121,  121,125,  125,129,  129,133,  133,137
DATA 137,105,  106,110,  110,114,  114,122,  122,126
'
DATA 126,130,  130,134,  134,138,  138,106,  107,111
DATA 111,115,  115,123,  123,127,  127,131,  131,135
DATA 135,139,  139,107,  142,143,   48, 80,   80,148
DATA 150,152,  152, 80,   82,162,  162,116,  118,144
'
DATA 144,146,  154,156,  156,158,  158,160,   49, 81
DATA  81,149,  151,153,  153, 81,   83,163,  163,117
DATA 119,145,  145,147,  155,157,  157,159,  159,161
DATA  62, 63,  164,165,  165,166,  166,167,  167,164
'
DATA  68, 69,   70, 71,   72, 73,   74, 75,   76, 77
DATA  78, 79,   84, 85,   86, 87,   88, 89,   90, 91
DATA  92, 93,   94, 95,   96, 97,   98, 99,  100,101
DATA 102,103,  104,105,  106,107,  108,109,  110,111
'
DATA 112,113,  114,115,  120,121,  122,123,  124,125
DATA 126,127,  128,129,  130,131,  132,133,  134,135
DATA 136,137,  138,139
```

4.4.2 Es geht auch realistischer: Zentralprojektion

Wenn uns das bisher Erreichte auch in Hochstimmung versetzen könnte, so bleiben jedoch noch einige Dinge, die zu beseitigen wünschenswert sind. Eines davon soll hier verbessert werden.

Wenn Sie im letzten Programm genauer hinschauen, dann wird Ihnen vielleicht auffallen, daß das Haus nach hinten hin größer zu werden scheint, obwohl die hintere Wand exakt genausogroß ist wie die vordere. Es handelt sich dabei nachweisbar um eine optische Täuschung. Sie rührt daher, daß wir normalerweise feststellen, daß die Gegenstände in unserer Umwelt immer kleiner erscheinen, je weiter sie von uns entfernt stehen. Die endlosen Eisenbahnschienen, die sich im Horizont treffen, sind das klassische Beispiel dafür. Man nennt dieses Phänomen auch die perspektivische Verzerrung. Bei der Parallelprojektion wurde das aber nicht berücksichtigt. Unser Auge (bzw. unser Gehirn) weiß von dem perspektivisches Effekt und zieht den Umkehrschluß: Wenn ein Objekt (die hintere Häuserwand) hinter einem anderen Objekt (die vordere Wand) liegt und die gleiche Größe besitzt, dann ist es in Wahrheit größer als das vordere. Deshalb erscheint uns die hintere Wand größer.

Wie können wir diesen Mangel beheben? Nun, eine sehr beliebte Technik ist die Einführung eines Fluchtpunktes (manche Maler bauen auch mehrere Fluchtpunkte in ihr Bild ein, um es noch realistischer erscheinen zu lassen. Das sind allerdings nur Hilfsmittel, die rein mathematisch auch durch einen einzigen Fluchtpunkt erreicht werden). Die verlängerten Kanten aller

Gegenstände, die wir in der Natur betrachten, scheinen sich nämlich in der Ferne in einem einzigen Punkt zu treffen, dem Fluchtpunkt (auch perspektivisches Zentrum genannt). Unsere Intention ist es nun, diesen Fluchtpunkt bei der Projektion der Raumkoordinaten auf eine Ebene zu berücksichtigen. Dies geschieht mit der sogenannten Zentral- oder perspektivischen Projektion.

Unser Fluchtpunkt liegt allerdings woanders. Er stellt quasi die Stelle dar, an der sich der Betrachter bzw. sein Auge befindet (oder der Fotoapparat), genauer gesagt: die Stelle, an der sämtliche Lichtstrahlen der betrachteten Objekte in einem Punkt zusammenfallen (s. Bild 4.20).

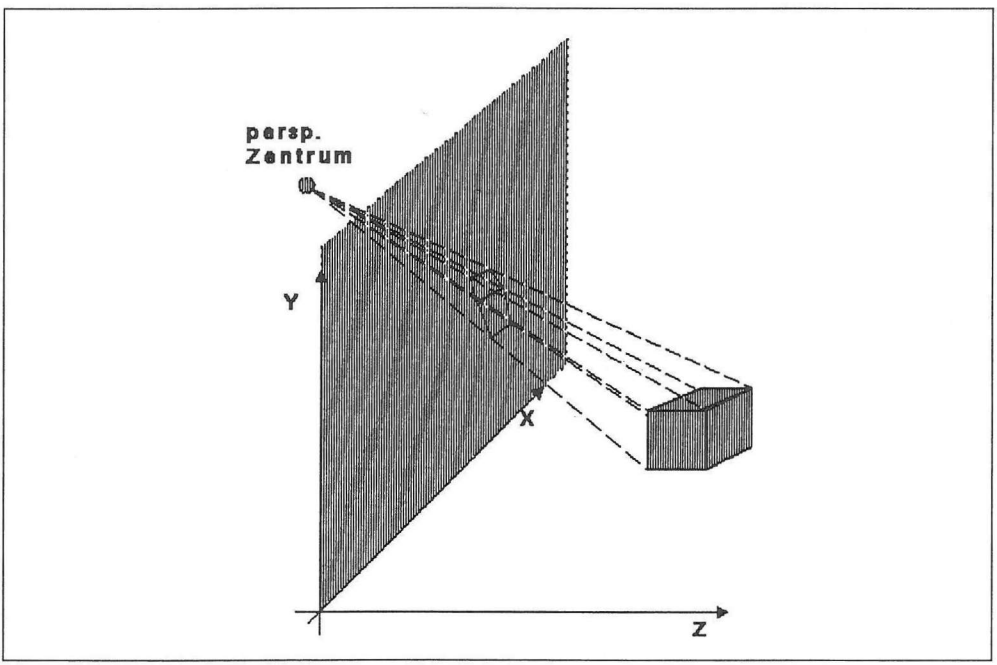


Bild 4.20: Die Zentralprojektion

Stellen Sie sich nun vor, Sie schauen durch eine matte Glasscheibe hindurch. Auf dieser Glasscheibe wird sich ein – wenn auch milchiges – Abbild der dahinterliegenden Welt abzeichnen (wir idealisieren hier natürlich, denn die Welt strahlt ja eigentlich durch die Scheibe hindurch). Der Weg der Lichtstrahlen geht also vom Objekt aus. Irgendwann treffen die Strahlen durch die Scheibe und dann in unser Auge.

Die Scheibe soll nun den Film bzw. unseren Bildschirm darstellen. Uns interessiert also, an welcher Stelle die Strahlen die Scheibe schneiden. Wir haben es mit dem gleichen Problem zu tun wie bei der Parallelprojektion. Die Strahlen sind allerdings nun nicht wie dort parallel, sondern treffen sich in einem Punkt.

Angenommen, das Zentrum der Projektion (also quasi der Fluchtpunkt) liegt im Punkt $Z(Z_x,Z_y,Z_z)$. Der zu projizierende 3-D-Punkt besitzt die Koordinaten $Q(Q_x,Q_y,Q_z)$. Mit der bekannten Geradengleichung in vektorieller Form:

$$\vec{P} = \vec{P}_0 + s*a$$

läßt sich dann folgendes ableiten: Z liegt auf der Geraden und wird für $\vec{P}_0$ eingesetzt. Fehlt nur der Vektor $\vec{a}$. Ihn erhalten wir durch die Verbindung der Punkte Z und Q:

$$\vec{a} = Q-Z = \begin{pmatrix} Q_x \\ Q_y \\ Q_z \end{pmatrix} - \begin{pmatrix} Z_x \\ Z_y \\ Z_z \end{pmatrix} = \begin{pmatrix} Q_x-Z_x \\ Q_y-Z_y \\ Q_z-Z_z \end{pmatrix}$$

Damit erhalten wir folgende Gleichung eines Strahles (einer Geraden) vom Objektpunkt zum Zentrum:

$$P = Z + s* \begin{pmatrix} Q_x-Z_x \\ Q_y-Z_y \\ Q_z-Z_z \end{pmatrix}$$

oder in Parameterform:

$$P_x = Z_x + s*(Q_x-Z_x)$$
$$P_y = Z_y + s*(Q_y-Z_y)$$
$$P_z = Z_z + s*(Q_z-Z_z)$$

Wie bei der Parallelprojektion ist $\vec{P}$ dabei der gesuchte Punkt auf der Projektionsebene, der Punkt also, bei dem die Gerade die Ebene (bzw. die Glasscheibe) schneidet. Uns interessiert jetzt, welchen Wert wir für s einsetzen müssen, damit $\vec{P}$ tatsächlich auf der Projektionsebene liegt. Auch hier helfen wir uns wieder, indem wir die Scheibe in die xy-Ebene setzen. Für den Punkt $\vec{P}$ bedeutet das, daß seine z-Koordinate Pz gleich Null ist. Damit erhalten wir aus der dritten Gleichung:

$$0 = Z_z + s*(Q_z-Z_z) \qquad <=>$$

$$s = -\frac{Z_z}{Q_z-Z_z}$$

Wir setzen auch dies in die beiden ersten Gleichungen ein und formen ein wenig um:

$$P_x = Z_x - Z_z * \frac{Q_x-Z_x}{Q_z-Z_z} \qquad \text{und}$$

$$P_y = Z_y - Z_z * \frac{Q_y-Z_y}{Q_z-Z_z}$$

Die lange Prozedur der Umformungen wollen wir Ihnen hier ersparen und teilen Ihnen das Ergebnis direkt mit:

$$P_x = \frac{Z_x*Q_z - Q_x*Z_z}{Q_z - Z_z} \qquad \text{und}$$

$$P_y = \frac{Z_y * Q_z - Q_y * Z_z}{Q_z - Z_z}$$

wobei:

P_x, P_y – Koordinaten des projizierten Ebenenpunktes
Q_x, Q_y, Q_z – Koordinaten des zu projizierenden Raumpunktes
Z_x, Z_y, Z_z – Zentrum der Projektion

Sie sind sicher bereits gespannt, wie die entsprechende Transformationsmatrix diesmal aussieht. Das ist in der Tat nicht ganz so einfach und wir benötigen hier in jedem Fall die homogenen Koordinaten, die wir glücklicherweise bereits früher eingeführt haben. Hier ist sie:

$$\begin{aligned}\vec{P} &= (P_x*P_n \quad P_y*P_n \quad P_z*P_n \quad P_n) \\ &= (Q_x*Q_n \quad Q_y*Q_n \quad Q_z*Q_n \quad Q_n) * \begin{pmatrix} -Z_z & 0 & 0 & 0 \\ 0 & -Z_z & 0 & 0 \\ Z_x & Z_y & 0 & 1 \\ 0 & 0 & 0 & -Z_z \end{pmatrix}\end{aligned}$$

Wir nennen die Matrix in Kurzform auch: $ZP(Z_x,Z_y,Z_z)$. Die Werte P_n und Q_n sind die »homogenen« Faktoren, die ja bekanntlich bei homogenen Koordinaten hinzugefügt werden müssen (wir haben sie bei der Einführung der homogenen Koordinaten »n« genannt). Sie werden normalerweise gleich 1 gesetzt (Qn könnten Sie in den folgenden Ausführungen ebenfalls gleich 1 setzen, aber wir wollen ja eine allgemeine Formulierung, denn schließlich könnte Q_n auch aus einer anderen Transformation stammen und wäre dann nicht mehr gleich 1). Hier spielen sie allerdings eine Rolle – allerdings nur übergangsweise. Hier ist P_n nämlich nicht gleich 1. Rechnen wir kurz nach, ob die Matrix wirklich stimmt. Die folgende Kontrollrechnung sollte Ihnen auch als Beispiel dienen, falls Sie sich eigene Transformationskombinationen (z.B. Zentralprojektion mit Drehung etc.) ableiten möchten:

Das Ergebnis des obigen Ausdruckes ist:

$$\begin{aligned}\vec{P} &= (P_x*P_n \quad P_y*P_n \quad P_z*P_n \quad P_n) \\ &= (-Q_x*Q_n*Z_z+Q_z*Q_n*Z_x \quad -Q_y*Q_n*Z_z+Q_z*Q_n*Z_y \quad 0 \quad Q_z*Q_n-Q_n*Z_z)\end{aligned}$$

Wenn Sie Probleme mit der Ausrechnung dieser Matrix haben, dann schauen Sie doch noch einmal kurz in den Anhang.

Um aus dieser Matrix die tatsächlichen Werte für P_x, P_y und P_z zu berechnen, müssen wir die einzelnen Matrixelemente durch P_n teilen (siehe Definition der homogenen Koordinaten). Das geht folgendermaßen:

Wir suchen bekanntlich die Werte für P_x, P_y, P_z und P_n. Aus der Gleichung für $\vec{P}$ können wir folgende Beziehungen ableiten (das sind einfach die entsprechenden Parametergleichungen):

$$\begin{aligned}P_x*P_n &= -Q_x*Q_n*Z_z + Q_z*Q_n*Z_x \\ P_y*P_n &= -Q_y*Q_n*Z_z + Q_z*Q_n*Z_y \\ P_z*P_n &= 0 \\ P_n &= Q_z*Q_n - Q_n*Z_z\end{aligned}$$

Die formen wir wie folgt um. Wir formen exemplarisch die erste unter Zuhilfenahme der letzten um:

$$P_x * P_n = -Q_x * Q_n * Z_z + Q_z * Q_n * Z_x \quad <=>$$

$$P_x = \frac{-Q_x * Q_n * Z_z + Q_z * Q_n * Z_x}{P_n} \quad <=>$$

$$P_x = \frac{-Q_x * Q_n * Z_z + Q_z * Q_n * Z_x}{Q_z * Q_n - Q_n * Z_z} \quad <=>$$

$$P_x = \frac{-Q_x * Z_z + Q_z * Z_x}{Q_z - Z_z}$$

Das ist genau die Formel, die wir oben abgeleitet haben. Die zweite Gleichung wird ganz genauso aufgelöst und ergibt ebenfalls das gewünschte Ergebnis. Die dritte ist ebenfalls leicht umgeformt:

$$P_z * P_n = 0 \quad <=>$$
$$P_z = 0$$

Sie sehen, die Behandlung der Zentralprojektion wird ein ganzes Stück komplizierter als die der parallelen Abbildung. Das liegt einfach an der Tatsache, daß wir es mit einer Division zu tun haben. Eine Division wird nur mit homogenen Koordinaten möglich, unter Zuhilfenahme des homogenen Elementes n (oder hier Q_n bzw. P_n).

Ein Programmbeispiel sparen wir uns auf, bis wir uns mit den Transformationen in 3-D auskennen. Bei der Parallelprojektion mußten wir ja ebenfalls bereits auf Drehungen vorgreifen. Genau das wollen wir hier vermeiden.

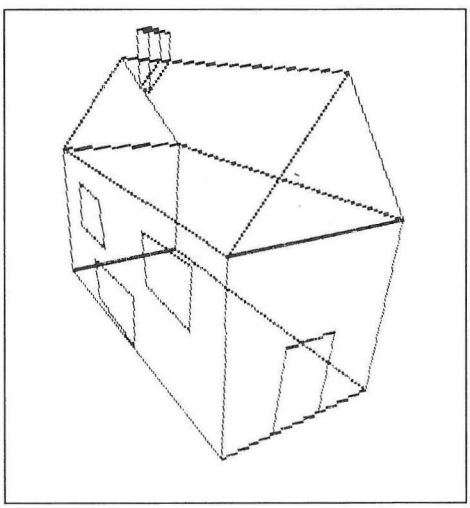


Bild 4.21: Zentralprojektion 1

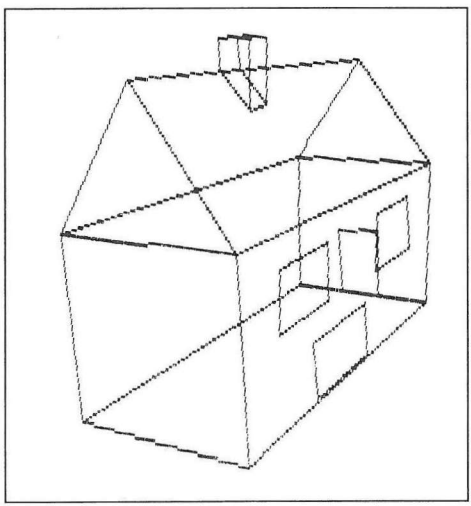


Bild 4.22: Zentralprojektion 2

4.5 Wir bringen Bewegung rein: Transformationen auch im Raum

In den letzten Abschnitten haben wir gesehen, wie es möglich ist, dreidimensionale Gebilde auf einen zweidimensionalen Bildschirm zu projizieren. Bei der Realisierung in Programmen vermißten wir jedoch die Möglichkeit von Transformationen im Raum teilweise sehr (erinnert sei an die notwendigen Drehungen beim Programm für die parallele Projektion).

Andererseits haben wir Transformationen in der Ebene bereits zur Genüge kennengelernt und möchten hierauf aufbauen. Dem soll hier nun entsprochen werden.

4.5.1 Verschiebungen, Vergrößerungen, Drehungen

Zum Anfang des Kapitels »Projektionen« haben wir bereits dargelegt, wie die 2-D-Transformationsmatrizen in die Form für dreidimensionale Koordinaten umgewandelt werden können. Nun möchten wir diese Matrizen so erweitern, daß sie auch im Raum funktionieren. Beginnen wir also bei der einfachsten Transformation, der Skalierung (Vergrößerung bzw. Verkleinerung).

Die 2-D-Matrix für diese Operation lautete:

$$\begin{pmatrix} S_x & 0 \\ 0 & S_y \end{pmatrix}$$

oder in homogenen Koordinaten:

$$\begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Im dreidimensionalen Raum muß entsprechend ein Vergrößerungsfaktor S_z für die z-Koordinaten eingebaut werden. Die zugehörige Matrix direkt in homogenen Koordinaten:

$$S(S_x,S_y,S_z) = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Um nun einen Punkt P(x,y,z) mit dieser Matrix zu vergrößern, verwenden wir – wie in der Ebene – die Matrixmultiplikation, eines unserer wichtigsten Arbeitsmittel überhaupt (zur Definition der Matrixmultiplikation s. Anhang):

$$\begin{aligned} P' &= (x'*n' \quad y'*n' \quad z'*n' \quad n') \\ &= P(x,y,z) * S(S_x,S_y,S_z) \\ &= (x*n \quad y*n \quad z*n \quad n) * \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= (x*S_x*n \quad y*S_y*n \quad z*S_z*n) \end{aligned}$$

und da n = 1 gesetzt werden kann:

$$= (x*S_x \quad y*S_y \quad z*S_z \quad 1)$$

Die Parameterformen schauen wieder etwas einfacher aus:

$$\begin{aligned} x' &= S_x*x \\ y' &= S_y*y \\ z' &= S_z*z \\ (n' &= n) \end{aligned}$$

Für die Translation (Verschiebung) benötigen wir in jedem Fall die homogenen Koordinaten. Die Ebenenmatrix lautete:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{pmatrix}$$

Im Raum erweitern wir wieder:

$$T(T_x,T_y,T_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix}$$

Die entsprechenden Parametergleichungen heißen dann:

$$\begin{aligned} x' &= x+T_x \\ y' &= y+T_y \\ z' &= z+T_z \\ (n' &= n) \end{aligned}$$

Fehlen uns nur noch die Drehungen (wenn wir einmal von den vielen anderen möglichen Transformationen absehen). Hier wird es ein wenig schwieriger. Unter zweidimensionalen Verhältnissen war eine Drehung eindeutig definiert als eine Drehung um den Nullpunkt (oder später um einen beliebigen anderen Punkt). Im Raum reicht die Aussage »Drehung um einen Punkt« nicht mehr aus. Schließlich gibt es unendlich viele Richtungen, um die dann gedreht werden kann. Im Raum müssen wir eine Achse angeben, um die wir ein Objekt rotieren wollen. Erst dann reduzieren sich die Drehrichtungen auf zwei Möglichkeiten: mit oder gegen den Uhrzeigersinn.

Jetzt können wir uns natürlich jede beliebige Achse denken, um die ein Gebilde gedreht werden kann. Der Einfachheit halber aber beschränken wir uns auf die drei Koordinatenachsen. Und Sie werden sehen, mit dieser Grundlage sind auch Drehungen um jede beliebige Achse möglich.

Übertragen wir doch erst einmal die Verhältnisse aus 2-D nach 3-D. Stellen wir uns vor, die 2-D-Ebene befindet sich im 3-D-Koordinatensystem genau in der x-y-Ebene (die z-Koordinate ist an allen Stellen der Ebene gleich Null). Eine Rotation der 2-D-Ebene wäre dann auf den

Raum übertragen eine Rotation um die z-Achse. Da die 2-D-Rotationsmatrix für Rotationen gegen den Uhrzeigersinn in homogenen Koordinaten wie folgt lautete (a gibt den Drehwinkel an):

$$\begin{pmatrix} \cos(a) & \sin(a) & 0 \\ -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

können wir die Raumdrehung um die z-Achse sehr einfach durch folgende Matrix ausdrücken:

$$R_z(a) = \begin{pmatrix} \cos(a) & \sin(a) & 0 & 0 \\ -\sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

oder in Parameterform für die Rotation des Punktes P(x,y,z):

$$\begin{aligned} x' &= x*\cos(a) - y*\sin(a) \\ y' &= x*\sin(a) + y*\cos(a) \\ z' &= z \\ (n' &= n) \end{aligned}$$

Wir sehen, die z-Koordinate ändert sich logischerweise nicht.

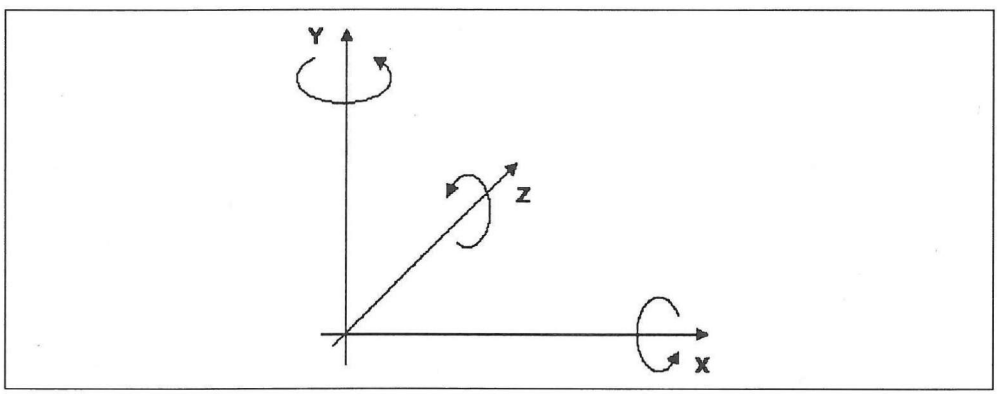


Bild 4.23: Rotationen um die Raumachsen

Bleibt uns die Beschreibung der Rotationen um die x- und die y-Achsen. Alle Rotationen sollen gegen den Uhrzeigersinn erfolgen. Dabei müssen Sie von der positiven Seite her auf die Spitze einer Achse schauen (s. Bild 4.23). Die Rotationsmatrizen lauten dann analog zur ersten für die Rotation um die x-Achse:

$$R_x(a) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & \sin(a) & 0 \\ 0 & -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

mit der Parameterform für die Rotation eines Punktes P(x,y,z):

$$
\begin{aligned}
x' &= x \\
y' &= y*\cos(a) - z*\sin(a) \\
z' &= y*\sin(a) + z*\cos(a)
\end{aligned}
$$

und um die y-Achse:

$$
R_y(a) = \begin{pmatrix} \cos(a) & 0 & -\sin(a) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(a) & 0 & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

mit der Parameterform für die Rotation eines Punktes P(x,y,z):

$$
\begin{aligned}
x' &= x*\cos(a) + z*\sin(a) \\
y' &= y \\
z' &= -x*\sin(a) + z*\cos(a)
\end{aligned}
$$

Die Transformationsmatrix für eine Rotation um alle Achsen mit den Winkeln a (x-Achse), b (y-Achse) und c (z-Achse) in der Reihenfolge: Rotation um x, um y, um z können wir durch folgende Matrixmultiplikation berechnen:

$$R_x(a)_y(b)_z(c) = R_x(a) * R_y(b) * R_z(c)$$

Wir haben uns die Mühe gemacht und das Ergebnis berechnet:

$$
R_x(a)_y(b)_z(c) = \begin{pmatrix} \cos(b)\cos(c) & \cos(b)\sin(c) & -\sin(b) & 0 \\ \sin(a)\sin(b)\cos(c) - \cos(a)\sin(c) & \sin(a)\sin(b)\sin(c) + \cos(a)\cos(c) & \sin(a)\cos(b) & 0 \\ \cos(a)\sin(b)\cos(c) + \sin(a)\sin(c) & \cos(a)\sin(b)\sin(c) - \sin(a)\cos(c) & \cos(a)\cos(b) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}
$$

Multiplizieren wir diese Mammutmatrix mit der eines Punktes, so wird dieser Punkt in alle drei Richtungen gedreht. Die resultierenden Parametergleichungen lauten:

$$
\begin{aligned}
x' &= x*A + y*B + z*C \\
y' &= x*D + y*E + z*F \\
z' &= x*G + y*H + z*I
\end{aligned}
$$

Für die Parameter A,B,... müssen Sie dann folgendes eingeben:

$$
\begin{aligned}
A &= \cos(b)*\cos(c) \\
B &= \cos(b)*\sin(c) \\
C &= -\sin(b) \\
D &= \sin(a)*\sin(b)*\cos(c) - \cos(a)*\sin(c) \\
E &= \sin(a)*\sin(b)*\sin(c) + \cos(a)*\cos(c) \\
F &= \sin(a)*\cos(b) \\
G &= \cos(a)*\sin(b)*\cos(c) + \sin(a)*\sin(c) \\
H &= \cos(a)*\sin(b)*\sin(c) - \sin(a)*\cos(c) \\
I &= \cos(a)*\cos(b)
\end{aligned}
$$

Falls Sie die Reihenfolge der Drehungen ändern wollen (z.B. zuerst um die z-Achse), dann müssen Sie die Matrix neu berechnen. Bei gleicher Reihenfolge können Sie selbstverständlich einen oder zwei Winkel gleich Null werden lassen. Das Objekt wird dann um die entsprechende Achse nicht mehr gedreht. Dadurch fallen einige Sinus und Cosinus in den obigen Gleichungen fort, da $\sin(0) = 0$ und $\cos(0) = 1$.

Wenn Sie die gedrehten Punkte noch auf eine Ebene projizieren möchten, um sie auf dem Bildschirm darzustellen, dann sind die Punkte nach der Drehung zusätzlich der entsprechenden Projektionsmatrix zu unterwerfen. Das geschieht am besten in einem separaten Arbeitsgang.

Was wir uns bis jetzt mühsam in grauer Theorie erarbeitet haben, das sollte nun auch einmal in der Praxis erprobt werden. Hierzu sehen Sie sich das folgende C-Programm an. Es realisiert die meisten der bisher entwickelten Transformationen samt Zentralprojektion etc. Dabei verwendet es in etwa die oben bereits skizzierte Struktur eines 3-D-CAD-Systems mit der Unterteilung in Welt, Objekte, Punkte, Linien etc. Jede dieser Einheiten wird in diesem Programm durch eine oder mehrere Strukturen repräsentiert.

```
/*************************************/
/**                                 **/
/**       Die große Demo zur        **/
/**                                 **/
/**          3-D-ANIMATION          **/
/**                                 **/
/**              mit                **/
/**   Zentralprojektion, Rotation,  **/
/**     Translation, Skalierung     **/
/**                                 **/
/**          Organisation:          **/
/**       objektorientiertes        **/
/**           Drahtmodell           **/
/**                                 **/
/*************************************/


#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/mathffp.h>


/* Funktionsdeklarationen (nur für Aztek-Compiler): */
/* wahlweise auch: #include <functions.h>          */
/*************************************************/
VOID                ClearScreen();
VOID                Draw();
VOID                Exit();
```

```
struct Message *      GetMsg();
VOID                  Move();
struct Library *      OpenLibrary();
struct Screen *       OpenScreen();
struct Window *       OpenWindow();
VOID                  ReplyMsg();
VOID                  ScreenToFront();
VOID                  SetAPen();
VOID                  SetRGB4();
LONG                  Wait();
/*****************************************************/


struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
LONG   *MathBase;


/* Struktur für 2 neue Bildschirme initialisieren: */
/*****************************************************/

struct NewScreen NeuerBildschirm =
{
   0,                   /* x-Koord. linke obere    */
                        /* Ecke (immer 0)          */
   0,                   /* y-Koord. linke obere    */
                        /* Ecke                    */
   640,                 /* Bildschirmbreite        */
   256,                 /* Bildschirmhöhe          */
   2,                   /* Bildebenenzahl          */
   0,                   /* Farbe der Details       */
   1,                   /* Farbe der Flächen       */
   HIRES,               /* Grafikmodus: 640x256    */
   CUSTOMSCREEN,        /* Bildschirmtyp           */
   NULL,                /* kein neuer Font         */
   "Zentral-Projektion",/* Text im Bildschirm-Kopf */
   NULL,                /* unbenutzt, immer NULL   */
   NULL,                /* keine eigene BitMap     */
};


/* Struktur für zwei neue Fenster initialisieren:  */
/*****************************************************/
```

```
struct NewWindow NeuesFenster =
{
   0,                    /* x-Koord. linke ob. Ecke */
   10,                   /* y-Koord. linke ob. Ecke */
   640,                  /* Fensterbreite           */
   246,                  /* Fensterhöhe             */
   0,                    /* Farbe der Details       */
   1,                    /* Farbe der Flächen       */
   MOUSEBUTTONS |        /* Rückmeldg. b. Mausknopf */
      RAWKEY,            /* und Taste               */
   ACTIVATE |            /* Fensterelemente und -typ*/
    BORDERLESS|          /* wählen: Randlos,        */
    NOCAREREFRESH |      /* keine Refresh-Meldungen */
    RMBTRAP,             /* keine Menüoperationen   */
   NULL,                 /* keine eigenen Gadgets   */
   NULL,                 /* CheckMark               */
   NULL,                 /* Text im Fenster-Kopf    */
   0,                    /* Adresse Screen-Struktur */
                         /* Muß innerhalb des       */
                         /* Programmes nach dem     */
                         /* Öffnen eines Screens    */
                         /* initialisiert werden    */
   NULL,                 /* kein SuperBitmap-Fenster*/
   640,                  /* Mindestbreite           */
   246,                  /* Mindesthöhe             */
   640,                  /* Maximalbreite           */
   256,                  /* Maximalhöhe             */
   CUSTOMSCREEN,         /* Bildschirmtyp           */
};


struct Screen *Bildschirm[2];
struct Window *Fenster[2];
struct RastPort *RastPort[2];
struct IntuiMessage *Message;


/* Palettenfarben: */
LONG palette[4][3] =
   {
      { 0, 0, 0}, {15, 0, 1},
      { 0, 4,15}, {14,10, 1}
   };
```

```
main()
{
 LONG do_program();

 LONG fehler;          /* Für die Typ-Abkürzungen s. unter */
                       /* Exec-Include-File: types.h       */
 ULONG i;


 /* Öffnen der Intuition-, Graphics-, Mathe-Bibliotheken */

 printf("3-D-Zentralprojektion / C-Demo\n");

 IntuitionBase =
 (struct IntuitionBase *)OpenLibrary("intuition.library",0L);
 if (IntuitionBase == NULL)            fehler = 1;
 else
 {
  GfxBase =
  (struct GfxBase *)OpenLibrary("graphics.library",0L);
  if (GfxBase == NULL)                 fehler = 2;
  else
  {
   MathBase =
   (LONG *)OpenLibrary("mathffp.library",0L);
   if (MathBase == NULL)               fehler = 3;
   else
   {
    /* Bildschirm 0 öffnen:   */
    Bildschirm[0] =
    (struct Screen *)OpenScreen(&NeuerBildschirm);
    if (Bildschirm[0] == NULL)         fehler = 4;
    else
    {
     /* Farbpalette setzen: */
     for (i=0; i<4; i++)
      SetRGB4(&Bildschirm[0]->ViewPort, i,
              palette[i][0], palette[i][1], palette[i][2]);

     /* Fenster 0 öffnen:     */
     /* Adresse der Screen-Struktur nachtragen:   */
     NeuesFenster.Screen = Bildschirm[0];
     Fenster[0] =
     (struct Window *)OpenWindow(&NeuesFenster);
     if (Fenster[0] == NULL)           fehler = 5;
```

```
    else
    {
     RastPort[0] = Fenster[0]->RPort; /* RastPort merken */

     /* Bildschirm 1 öffnen: */
     Bildschirm[1] =
     (struct Screen *)OpenScreen(&NeuerBildschirm);
     if (Bildschirm[1] == NULL)       fehler = 6;
     else
     {
      /* Farbpalette setzen: */
      for (i=0; i<4; i++)
       SetRGB4(&Bildschirm[1]->ViewPort, i,
              palette[i][0], palette[i][1], palette[i][2]);

      /* Fenster 1 öffnen:   */
      /* Adresse der Screen-Struktur nachtragen: */
      NeuesFenster.Screen = Bildschirm[1];
      Fenster[1] =
      (struct Window *)OpenWindow(&NeuesFenster);
      if (Fenster[1] == NULL)         fehler = 7;
      else
      {
       RastPort[1] = Fenster[1]->RPort; /* RastPort merken */

       fehler = do_program();      /* eigentliches Pro-     */
                                   /* gramm aufrufen        */

       CloseWindow(Fenster[1]);    /* Fenster 1 schließen   */
      }
      CloseScreen(Bildschirm[1]); /* Screen 1 schließen    */
     }
     CloseWindow(Fenster[0]);     /* Fenster 0 schließen   */
    }
    CloseScreen(Bildschirm[0]);   /* Screen 0 schließen    */
   }
   CloseLibrary(MathBase);        /* Mathe-Library close   */
  }
  CloseLibrary(GfxBase);          /* Graphics-Lib. close   */
 }
 CloseLibrary(IntuitionBase);     /* Intuition-Lib. close  */
}
Exit(fehler);                     /* Ausgang m. Fehlercode */
}
```

```
/*****************/
/* Hauptprogramm */
/*****************/


/* Globale Variablen und Strukturen:  */
/**************************************/

/* Addition von 90 zu einem Winkel w mit: 0 <= w < 360 */
/* Ergebniswinkel liegt wieder zwischen 0 und 360      */
/* (Verwendung für die Cosinusberechnung der folgenden  */
/*  Sinustabelle).                                      */

#define W_PLUS_90(w)   ( ((w) >= 270)? (w)-270 : (w)+90 )


/* Sinus/Cosinustabelle:                    */

/* Die Tabelle enthält in 1-Grad-Abständen */
/* Sinuswerte von 0 bis 359 Grad jeweils   */
/* multipliziert mit 2^15=32768.           */
/* Das oberste Bit ist das Vorzeichenbit.  */
/* Für die Ermittlung des Cosinus muß zum  */
/* Winkel vorher jeweils 90 Grad addiert   */
/* werden.                                 */

WORD sinus[] =
 {
  0x0000, 0x023C, 0x0478, 0x06B3, 0x08EE, /*  0- 4 Grad */
  0x0B28, 0x0D61, 0x0F99, 0x11D0, 0x1406, /*  5- 9 Grad */
  0x163A, 0x186C, 0x1A9D, 0x1CCB, 0x1EF7, /* 10-14 Grad */
  0x2121, 0x2348, 0x256C, 0x278E, 0x29AC, /* ...        */
  0x2BC7, 0x2DDF, 0x2FF3, 0x3203, 0x3410,
  0x3618, 0x381D, 0x3A1C, 0x3C18, 0x3E0E,
  0x4000, 0x41ED, 0x43D4, 0x45B7, 0x4794,
  0x496B, 0x4B3D, 0x4D08, 0x4ECE, 0x508E,
  0x5247, 0x53FA, 0x55A6, 0x574C, 0x58EB,
  0x5A82, 0x5C13, 0x5D9D, 0x5F1F, 0x609A,
  0x620E, 0x637A, 0x64DE, 0x663A, 0x678E,
  0x68DA, 0x6A1E, 0x6B5A, 0x6C8D, 0x6DB8,
  0x6EDA, 0x6FF4, 0x7104, 0x720D, 0x730C,
  0x7402, 0x74EF, 0x75D3, 0x76AE, 0x7780,
  0x7848, 0x7907, 0x79BC, 0x7A68, 0x7B0B,
  0x7BA3, 0x7C33, 0x7CB8, 0x7D34, 0x7DA6,
  0x7E0E, 0x7E6D, 0x7EC1, 0x7F0C, 0x7F4C,
  0x7F83, 0x7FB0, 0x7FD3, 0x7FEC, 0x7FFB,
```

```
0x7FFF, 0x7FFB, 0x7FEC, 0x7FD3, 0x7FB0, /* 90-94 Grad */
0x7F83, 0x7F4C, 0x7F0C, 0x7EC1, 0x7E6D, /* 95-99 Grad */
0x7E0E, 0x7DA6, 0x7D34, 0x7CB8, 0x7C33, /* ...        */
0x7BA3, 0x7B0B, 0x7A68, 0x79BC, 0x7907,
0x7848, 0x7780, 0x76AE, 0x75D3, 0x74EF,
0x7402, 0x730C, 0x720D, 0x7104, 0x6FF4,
0x6EDA, 0x6DB8, 0x6C8D, 0x6B5A, 0x6A1E,
0x68DA, 0x678E, 0x663A, 0x64DE, 0x637A,
0x620E, 0x609A, 0x5F1F, 0x5D9D, 0x5C13,
0x5A82, 0x58EB, 0x574C, 0x55A6, 0x53FA,
0x5247, 0x508E, 0x4ECE, 0x4D08, 0x4B3D,
0x496B, 0x4794, 0x45B7, 0x43D4, 0x41ED,
0x4000, 0x3E0E, 0x3C18, 0x3A1C, 0x381D,
0x3618, 0x3410, 0x3203, 0x2FF3, 0x2DDF,
0x2BC7, 0x29AC, 0x278E, 0x256C, 0x2348,
0x2121, 0x1EF7, 0x1CCB, 0x1A9D, 0x186C,
0x163A, 0x1406, 0x11D0, 0x0F99, 0x0D61,
0x0B28, 0x08EE, 0x06B3, 0x0478, 0x023C,

0x0000, 0xFDC4, 0xFB88, 0xF94D, 0xF712, /* 180-184 Grad */
0xF4D8, 0xF29F, 0xF067, 0xEE30, 0xEBFA, /* ...          */
0xE9C6, 0xE794, 0xE563, 0xE335, 0xE109,
0xDEDF, 0xDCB8, 0xDA94, 0xD872, 0xD654,
0xD439, 0xD221, 0xD00D, 0xCDFD, 0xCBF0,
0xC9E8, 0xC7E3, 0xC5E4, 0xC3E8, 0xC1F2,
0xC000, 0xBE13, 0xBC2C, 0xBA49, 0xB86C,
0xB695, 0xB4C3, 0xB2F8, 0xB132, 0xAF72,
0xADB9, 0xAC06, 0xAA5A, 0xA8B4, 0xA715,
0xA57E, 0xA3ED, 0xA263, 0xA0E1, 0x9F66,
0x9DF2, 0x9C86, 0x9B22, 0x99C6, 0x9872,
0x9726, 0x95E2, 0x94A6, 0x9373, 0x9248,
0x9126, 0x900C, 0x8EFC, 0x8DF3, 0x8CF4,
0x8BFE, 0x8B11, 0x8A2D, 0x8952, 0x8880,
0x87B8, 0x86F9, 0x8644, 0x8598, 0x84F5,
0x845D, 0x83CD, 0x8348, 0x82CC, 0x825A,
0x81F2, 0x8193, 0x813F, 0x80F4, 0x80B4,
0x807D, 0x8050, 0x802D, 0x8014, 0x8005,

0x8000, 0x8005, 0x8014, 0x802D, 0x8050, /* 270-274 Grad */
0x807D, 0x80B4, 0x80F4, 0x813F, 0x8193, /* ...          */
0x81F2, 0x825A, 0x82CC, 0x8348, 0x83CD,
0x845D, 0x84F5, 0x8598, 0x8644, 0x86F9,
0x87B8, 0x8880, 0x8952, 0x8A2D, 0x8B11,
0x8BFE, 0x8CF4, 0x8DF3, 0x8EFC, 0x900C,
```

```
  0x9126, 0x9248, 0x9373, 0x94A6, 0x95E2,
  0x9726, 0x9872, 0x99C6, 0x9B22, 0x9C86,
  0x9DF2, 0x9F66, 0xA0E1, 0xA263, 0xA3ED,
  0xA57E, 0xA715, 0xA8B4, 0xAA5A, 0xAC06,
  0xADB9, 0xAF72, 0xB132, 0xB2F8, 0xB4C3,
  0xB695, 0xB86C, 0xBA49, 0xBC2C, 0xBE13,
  0xC000, 0xC1F2, 0xC3E8, 0xC5E4, 0xC7E3,
  0xC9E8, 0xCBF0, 0xCDFD, 0xD00D, 0xD221,
  0xD439, 0xD654, 0xD872, 0xDA94, 0xDCB8,
  0xDEDF, 0xE109, 0xE335, 0xE563, 0xE794,
  0xE9C6, 0xEBFA, 0xEE30, 0xF067, 0xF29F,
  0xF4D8, 0xF712, 0xF94D, 0xFB88, 0xFDC4  /* 355-359 Grad */
 };


struct punkt      /* Struktur eines Raumpunktes */
{
   WORD  x;               /* x-Koord. des Punktes   */
   WORD  y;               /* y-Koord. des Punktes   */
   WORD  z;               /* z-Koord. des Punktes   */
};

struct linie      /* Liniendefinition           */
{
   WORD  p1;              /* Nr. des 1. Endpunktes  */
   WORD  p2;              /* Nr. des 2. Endpunktes  */
};

struct objekt     /* Objekt-Struktur            */
{
   UWORD  anz_pun;        /* Anzahl der Punkte      */
   struct punkt *punkte;  /* Zeiger auf Punktearray */
   UWORD  anz_lin;        /* Anzahl der Linien      */
   struct linie *linien;  /* Zeiger auf Linienarray */
   UWORD  farbe;          /* Objektfarbe            */
   char   *name;          /* Objektname             */
   UBYTE  sichtbar;       /* =0: unsichtbar         */
                          /* =1: sichtbar           */
   UBYTE  transform;      /* =0: Objekt wird nicht  */
                          /*     transformiert      */
                          /* =1: Objekt wird        */
                          /*     transformiert      */
};
```

```
struct welt       /* Alles, was in der Welt ist */
{
   UWORD  anz_obj;          /* Anzahl der Objekte     */
   struct objekt *objekte; /* Zeiger auf Objektarray */

   /* Transformationswerte für die gesamte Welt: */

   WORD   x_rot;            /* Rotationswinkel (Grad) */
   WORD   y_rot;
   WORD   z_rot;
   WORD   x_ska;            /* Skalierungswerte in    */
   WORD   y_ska;            /* Zehnteln               */
   WORD   z_ska;
   WORD   x_tra;            /* Translationswerte      */
   WORD   y_tra;
   WORD   z_tra;
   WORD   x_beo;            /* Koordinaten des        */
   WORD   y_beo;            /* Beobachters            */
   WORD   z_beo;

   WORD   xe_tra;           /* Ebenentranslation      */
   WORD   ye_tra;           /* nach der Projektion    */
   WORD   xe_ska;           /* und Ebenenskalierung   */
   WORD   ye_ska;           /* zur Anpassung an die   */
                            /* Bildschirmauflösung    */
} urwelt =
   { 2,0,
     340,20,0, 20,20,20, 0,0,0, 0,0,-500,
     200,100, 2,1 };

USHORT Fen_Nr_s = 1,        /* sichtbare Fensternr.   */
       Fen_akt  = 1,        /* Aktives Fenster        */
       Fen_Nr_v = 0;        /* verdeckte Fensternr.   */


/* Werte zur Erhöhung/Erniedrigung von */
/* Transformationswerten per Tastatur: */
#define SKALIER_INC     1  /* Inc-Wert Skalierung    */
#define DREH_INC        1  /* Inc-Wert Drehung       */
#define TRANSLA_INC     4  /* Inc-Wert z-Translation */
#define BEOB_INC       15  /* + z-Koord. Beobachter  */
```

```
/* RAW-Tastencodes definieren: */
#define SHIFT      (IEQUALIFIER_LSHIFT | IEQUALIFIER_RSHIFT)
#define PLUS       0x5e
#define PLUS2      0x1b
#define MINUS      0x4a
#define MAL        0x5d
#define GETEILT    0x5c
#define C_RECHTS   0x4e
#define C_LINKS    0x4f
#define C_AUF      0x4c
#define C_AB       0x4d
#define Z_C_AUF    0x3e
#define Z_C_AB     0x1e
#define Z_C_RECHTS 0x2f
#define Z_C_LINKS  0x2d
#define DEL        0x46
#define HELP       0x5f
#define Z_ENTER    0x43
#define Z_NULL     0x0f
#define Z_PUNKT    0x3c
#define ESC        0x45


LONG do_program()

{
   VOID  welt_init(),
         schaffe_welt(),
         init_ausgabe(),
         welt_verteiler(),
          pun_ska(),
          pun_tra(),
          pun_rot(),
          pun_zpj(),
          obj_zeichnen(),
         add_winkel();

   struct welt   welt;        /* Speicher für Welt-Struktur   */
   struct objekt *objekte;    /* Array aller Original-Objekte */
   struct objekt *trans_obj; /* Array transformierte Objekte */

   ULONG  class;              /* Speicher für die Intuition-  */
   USHORT code, qualifier;    /* Message-Struktur             */
   APTR   address;
   SHORT  mouse_x, mouse_y;
```

```
WORD x_rot_inc = 0,        /* aktuelle Rotationsincrements */
     y_rot_inc = 0,
     z_rot_inc = 0;

USHORT Fen_zwis;           /* Zwischenspeicher Double-Buff */
USHORT verd_flag = 1;      /* Flag für verdecktes/offenes  */
                           /* Zeichnen                     */


if (!verd_flag)
{
   Fen_zwis = Fen_Nr_v;    /* Für offenes Zeichnen         */
   Fen_Nr_v = Fen_Nr_s;
}

/* Weltstruktur initialisieren: */
welt_init(&welt);

/* Daten für die Welt laden:    */
schaffe_welt(&welt, &objekte, &trans_obj);

/* Zeichenschleife:             */
/*******************************/
code = 0;
while (code != ESC)        /* Schleife bis Esc betätigt    */
{
   REGISTER UWORD flag = 0;

   /* Objektdaten in Arrays für transformierte */
   /* Objekte übertragen:                      */
   init_ausgabe(&welt, trans_obj);

   /* Die ganze Welt transformieren: */
   welt_verteiler(1, &welt, trans_obj); /* Skalierung  */
   welt_verteiler(2, &welt, trans_obj); /* Translation */
   welt_verteiler(3, &welt, trans_obj); /* Rotation    */

   /* Die ganze Welt projizieren: */
   welt_verteiler(4, &welt, trans_obj);


   /* Die ganze Welt (evt. verdeckt) zeichnen: */

   Move(RastPort[Fen_Nr_v], 0L, 0L);     /* Fenster     */
   ClearScreen(RastPort[Fen_Nr_v]);      /* löschen     */
```

```
    welt_verteiler(5, &welt, trans_obj); /* zeichnen    */

    if (verd_flag)
    {
       /* Verdecktes Fenster nach vorne bringen */
       /* und Fen_Nr_v <-> Fen_Nr_s tauschen:   */

       ScreenToFront( Bildschirm[Fen_Nr_v] );

       Fen_zwis = Fen_Nr_v;
       Fen_Nr_v = Fen_Nr_s;
       Fen_Nr_s = Fen_zwis;
    }


    /* Input-Schleife: */
    /*******************/
    do
    {
/* Auf Taste testen und ggf. nach code einlesen:   */
/* Nur dann auf Zeichen warten, wenn keine Drehung */
/* Eingelesen wird der Tastencode (kein ASCII!)    */
       if (x_rot_inc == 0  &&
           y_rot_inc == 0  &&
           z_rot_inc == 0)
       {
          /* Auf Message warten: */
          Wait(1L << Fenster[Fen_akt]->UserPort->mp_SigBit);
          flag = 1;            /* Flag für warten     */
       }


       /* Meldungen abarbeiten: */
       /*************************/
       while (Message =
              (struct IntuiMessage *)GetMsg(Fenster[Fen_akt]->UserPort))
       {
          /* Daten aus Message-Struktur lesen: */
          class     = Message->Class;
          code      = Message->Code;
          qualifier = Message->Qualifier;
          address   = Message->IAddress;
          mouse_x   = Message->MouseX;
          mouse_y   = Message->MouseY;
          ReplyMsg(Message);   /* Message zurückgeben */
```

```
/* Bei Maustaste (nur wenn die linke */
/* Maustaste niedergedrückt wurde):  */
if (class == MOUSEBUTTONS && code == MENUDOWN)
{
   /* Welt auf Bildschirm positionieren: */
   welt.xe_tra = mouse_x;
   welt.ye_tra = mouse_y;

   flag = 0;
}
else    /* anderenfalls: Tastatur */
{
   switch (code)
   {
      case PLUS2:
      case PLUS:                /* Vergrößerung  */
         welt.x_ska += SKALIER_INC,
         welt.y_ska += SKALIER_INC,
         welt.z_ska += SKALIER_INC;
         flag = 0;
         break;
      case MINUS:               /* Verkleinerung */
         welt.x_ska -= SKALIER_INC,
         welt.y_ska -= SKALIER_INC,
         welt.z_ska -= SKALIER_INC;
         flag = 0;
         break;
      case C_RECHTS:
         if (SHIFT & qualifier) /* SHIFT?       */
         {
            /* ja:   Drehung um z-Achse rechts */
            z_rot_inc -= DREH_INC;
         }
         else
         {
            /* nein: Drehung um y-Achse rechts */
            y_rot_inc -= DREH_INC;
         }
         flag = 0;
         break;
      case C_LINKS:
         if (SHIFT & qualifier) /* SHIFT?       */
         {
```

```
            /* ja:   Drehung um z-Achse links  */
            z_rot_inc += DREH_INC;
         }
         else
         {
            /* nein: Drehung um y-Achse links  */
            y_rot_inc += DREH_INC;
         }
         flag = 0;
         break;
      case C_AUF: /* Drehung um x-Achse rechts */
         x_rot_inc -= DREH_INC;
         flag = 0;
         break;
      case C_AB:  /* Drehung um x-Achse links  */
         x_rot_inc += DREH_INC;
         flag = 0;
         break;
      case Z_C_RECHTS:  /* Welt entfernen       */
         welt.z_tra += TRANSLA_INC;
         flag = 0;
         break;
      case Z_C_LINKS:   /* Welt annähern        */
         welt.z_tra -= TRANSLA_INC;
         flag = 0;
         break;
      case Z_C_AUF:     /* Beobachter entfern. */
         welt.z_beo -= BEOB_INC;
         flag = 0;
         break;
      case Z_C_AB:      /* Beobachter annähern */
         welt.z_beo += BEOB_INC;
         flag = 0;
         break;
      case DEL:    /* Koord.-Kreuz ein/aus     */
         ((welt.objekte)+0)->sichtbar =
           (((welt.objekte)+0)->sichtbar)?  0 : 1;
         flag = 0;
         break;
      case HELP:   /* Alles stop und zurück    */
         /* Welt-Werte rücksetzen: */
         welt_init(&welt);
      case Z_ENTER:/* Nur Rotationen stoppen   */
         x_rot_inc =    /* Rotation stoppen    */
         y_rot_inc =
```

```
                    z_rot_inc = 0;
                    flag = 0;
                    break;
                case Z_NULL: /* Koordinatenkreuz trans-  */
                             /* formieren ein/aus        */
                    ((welt.objekte)+0)->transform =
                       (((welt.objekte)+0)->transform)?  0 : 1;
                    flag = 0;
                    break;
                case Z_PUNKT:/* Verd. Zeichnen ein/aus   */
                    if (verd_flag = (verd_flag)? 0 : 1)
                    {
                       Fen_Nr_v = Fen_zwis; /* ein       */
                    }
                    else
                    {
                       Fen_zwis = Fen_Nr_v; /* aus       */
                       Fen_Nr_v = Fen_Nr_s;
                    }
                    break;
                case ESC:                   /* Ende      */
                    flag = 0;
                    break;

             } /* switch */
          } /* else */
       } /* while */
       add_winkel(&welt.x_rot, x_rot_inc);
       add_winkel(&welt.y_rot, y_rot_inc);
       add_winkel(&welt.z_rot, z_rot_inc);

    } while (flag);          /* solange flag != 0       */
  } /* while */
  return((LONG)TRUE);
}


/* Addiere pos./neg. Wert zu einem Winkel: */
/*****************************************/

VOID add_winkel(winkel, inc)
WORD *winkel;
WORD inc;

{
```

```
   *winkel += inc;          /* Werte addieren       */
   if (*winkel >= 360)      /* Winkel auf Werte von */
      *winkel -= 360;       /* 0 bis 359 bringen    */
   if (*winkel < 0)
      *winkel += 360;
}


/* Initialisiere Welt-Struktur: */
/********************************/

VOID welt_init(w)
struct welt *w;

{
   urwelt.anz_obj = w->anz_obj; /* Anzahl Objekte und Ob-  */
   urwelt.objekte = w->objekte; /* jektpointer unverändert */

   *w = urwelt;                 /* Urwelt nach Welt        */
}



/*************************************/
/* Einrichtung aller Datenstrukturen: */
/*************************************/

/* Dieses Array gibt an, wieviele Punkte und */
/* und wieviele Linien die verschiedenen     */
/* Objekte besitzen, welche Farbe und welche */
/* Namen sie haben:                          */

struct new_objekt
{
   UWORD anz_pun;     /* Anzahl Punkte */
   UWORD anz_lin;     /* Anzahl Linien */
   UWORD farbe;       /* Farbe         */
   char *name;        /* Name          */
} new_objekt[] =
   {
      { 6, 3,                    /* Punkte/Linien Objekt 1 */
        2, "Koordinatensystem"}, /* Farbe/Name Objekt 1    */

      {34,43,                    /* Punkte/Linien Objekt 2 */
        3, "Haus"},              /* Farbe/Name Objekt 2    */
```

```
      {0,0,0,"\0"}                /* Ende-Kennzeichen       */
   };


/* Dieses Array enthält alle Punkte aller    */
/* Objekte. Die Reihenfolge der Objekte und  */
/* der Punkte muß stets eingehalten werden!  */

struct punkt _punkte[] =
   {
      /* Koordinatenkreuz: */
      {-15,  0,  0}, { 40,  0,  0}, {  0,-15,  0},
      {  0, 40,  0}, {  0,  0,-15}, {  0,  0, 40},

      /* Haus:             */
      {-6, 6, 14}, { 6, 6, 14}, { 6,-6, 14},
      {-6,-6, 14}, { 6, 6,-14}, {-6, 6,-14},
      {-6,-6,-14}, { 6,-6,-14}, { 0,14, 14},
      { 0,14,-14}, {-2,-6, 14}, {-2, 0, 14},
      { 2, 0, 14}, { 2,-6, 14}, { 6, 4, 10},
      { 6, 4,  4}, { 6, 0,  4}, { 6, 0, 10},
      { 6, 4, -4}, { 6, 4,-10}, { 6, 0,-10},
      { 6, 0, -4}, { 6,-2,  2}, { 6,-2, -6},
      { 6,-6, -6}, { 6,-6,  2}, { 2,12, -4},
      { 2,16, -4}, { 2,16, -6}, { 2,12, -6},
      { 0,14, -4}, { 0,16, -4}, { 0,16, -6},
      { 0,14, -6}
   };


/* Dieses Array enthält alle Linien aller    */
/* Objekte. Die Reihenfolge beachten!        */
/* Die Numerierung der Punkte ist stets      */
/* relativ zum ersten Punkt eines Objektes!  */
/* (Erster Objektpunkt gleich Null)          */

struct linie _linien[] =
   {
      /* Koordinatenkreuz: */
      {0,1}, {2,3}, {4,5},

      /* Haus:             */
      { 0, 1}, { 1, 2}, { 2, 3}, { 3, 0},
      { 1, 4}, { 4, 7}, { 7, 2}, { 7, 6},
      { 6, 5}, { 5, 4}, { 5, 0}, { 6, 3},
```

```
      { 8, 9}, { 0, 8}, { 8, 1}, { 4, 9},
      { 5, 9}, {10,11}, {11,12}, {12,13},
      {14,15}, {15,16}, {16,17}, {17,14},
      {18,19}, {19,20}, {20,21}, {21,18},
      {22,23}, {23,24}, {24,25}, {25,22},
      {26,27}, {27,28}, {28,29}, {29,26},
      {30,31}, {31,32}, {32,33}, {26,30},
      {29,33}, {27,31}, {28,32}
   };


/* Speicherreservierungen: */
/***************************/

  /* Anzahl der Objekte: */
#define ANZ_OBJ (sizeof new_objekt / sizeof(struct new_objekt) - 1)

  /* Anzahl der Punkte: */
#define ANZ_PUN (sizeof _punkte / sizeof(struct punkt))


struct objekt _objekte[ANZ_OBJ];   /* Reserv. für Objekte   */
struct objekt _trans_obj[ANZ_OBJ]; /* f.transformierte Obj. */
struct punkt _trans_pun[ANZ_PUN];  /* Speicherres. für      */
                                   /* transformierte Punkte */


/* Alle Welt-, Objekt-, Linien- und */
/* Punkt-Definitionen einlesen      */
/************************************/

VOID schaffe_welt(w, p_objekte, p_trans_obj)
struct welt *w;                /* Zeiger auf Weltstruktur   */
struct objekt **p_objekte;     /* Zeiger auf Objekt-        */
struct objekt **p_trans_obj;   /* Array-Zeiger              */

{
   REGISTER UWORD i;
   REGISTER UWORD p_ges = 0,  /* Aktuelle Gesamtpunktzahl  */
                  l_ges = 0;  /* Aktuelle Gesamtlinienzahl */

   *p_objekte = _objekte;     /* Adresse des Objekt-Arrays */
   *p_trans_obj = _trans_obj; /* Adresse des Objekt-Arrays */
                              /* für transformierte Arrays */
   w->objekte = _objekte;     /* Adresse des Objekt-Arrays */
```

```
                                /* in die Weltstruktur       */

   for (i=0; new_objekt[i].anz_pun != 0; i++)
   {
      /* Objektstruktur für jedes Objekt belegen: */
      _objekte[i].anz_pun = new_objekt[i].anz_pun; /*Punktez.*/
      _objekte[i].punkte  = &_punkte[p_ges];       /*P.-Array*/
      _objekte[i].anz_lin = new_objekt[i].anz_lin; /*Linienz.*/
      _objekte[i].linien  = &_linien[l_ges];       /*L.-Array*/
      _objekte[i].farbe   = new_objekt[i].farbe;   /*Farbe   */
      _objekte[i].name    = new_objekt[i].name;    /*Name    */
      _objekte[i].sichtbar= 1;                     /*sichtbar*/
      _objekte[i].transform = 1;                   /*transf. */

      p_ges += new_objekt[i].anz_pun; /* akt. Gesamtzahl Punkte */
      l_ges += new_objekt[i].anz_lin; /* akt. Gesamtzahl Linien */
   }
   w->anz_obj = i;       /* Anzahl der Objekte eintragen */
}


/* Ausgabe und Transformationen vorbereiten durch */
/* Übertragung der Original-Objektdaten in die    */
/* Arrays für transformierte Daten                */
/************************************************/

VOID init_ausgabe(w, trans_obj)
struct welt *w;                     /* Zeiger Weltstruktur  */
struct objekt *trans_obj;           /* Zeiger auf transfor- */
                                    /* mierte Objekte       */

{
   struct objekt *objekte;          /* Zeiger auf Objekte   */
   REGISTER UWORD i, k;             /* Indizes FOR-Schleife */
   REGISTER UWORD p_ges = 0;        /* akt. Gesamtpunktzahl */

   objekte = w->objekte;            /* Zeiger initialisieren*/

   for (i=0; i < w->anz_obj; i++) /* Jedes Objekt bearb.  */
   {
      trans_obj[i] = objekte[i];    /* Objekt-Struktur in   */
                                    /* Struktur für trans-  */
                                    /* formierte Objekte    */
                                    /* übertragen           */
```

```
      /* Adresse des entsprechenden Array-Elementes als    */
      /* Adresse des ersten Objekt-Punkte-Array-Elementes: */
      trans_obj[i].punkte = &_trans_pun[p_ges];

      p_ges += objekte[i].anz_pun;  /* Startpos. next Array */

      for (k=0; k < objekte[i].anz_pun; k++)
      {
         /* Punkte-Array ebenfalls übertragen: */
         trans_obj[i].punkte[k] = objekte[i].punkte[k];
      }
   }
}


/* Ganze Welt bearbeiten: */
/**************************/

VOID welt_verteiler(modus, w, trans_obj)
UWORD modus;               /* Art der Bearbeitung:  */
                           /* =1: Skalierung        */
                           /* =2: Translation       */
                           /* =3: Rotation          */
                           /* =4: Zentralprojektion */
                           /* =5: Zeichnung         */
struct welt *w;            /* Zeiger auf Weltstruktur      */
struct objekt *trans_obj;  /* Zeiger auf Strukturen-Array  */
                           /* für transformierte Objekte   */

{
   VOID  obj_transf(),
         pun_ska(),
         pun_tra(),
         pun_rot(),
         pun_zen(),
        obj_zeichnen();

   REGISTER UWORD i;

     /* Alle Objekte transformieren: */
   for (i=0; i < w->anz_obj; i++)
   {
      /* Alle Bearbeitungen am aktuellen Objekt vornehmen  */
      /* evt. Ergebnisse in trans_obj[] speichern          */
      switch (modus)
```

```
      {
         case 1:
         /* if (trans_obj[i].transform) *//* transformieren? */
            {
               obj_transf(pun_ska, &trans_obj[i],
                          w->x_ska, w->y_ska, w->z_ska);
            }
            break;
         case 2:
            if (trans_obj[i].transform) /* transformieren? */
            {
               obj_transf(pun_tra, &trans_obj[i],
                          w->x_tra, w->y_tra, w->z_tra);
            }
            break;
         case 3:
            if (trans_obj[i].transform) /* transformieren? */
            {
               obj_transf(pun_rot, &trans_obj[i],
                          w->x_rot, w->y_rot, w->z_rot);
            }
            break;
         case 4:
            obj_transf(pun_zen, &trans_obj[i],
                       w->x_beo, w->y_beo, w->z_beo);
            break;
         case 5:
            obj_zeichnen(w, &trans_obj[i]);
            break;
      } /* switch */
   } /* for */
}


/* Transformation eines Objektes: */
/**********************************/

VOID obj_transf(operation, objekt, t1, t2, t3)
VOID (*operation)();     /* Zeiger auf Funktion,     */
                         /* die die Transformation   */
                         /* durchführen soll:        */
                         /* pun_ska(), pun_tra(),    */
                         /* pun_rot(), pun_zen()     */
```

```
struct objekt *objekt;  /* Adresse des zu trans-    */
                        /* formierenden Objektes    */
WORD t1, t2, t3;        /* Transformationsparameter */

{
   struct punkt *punkte;       /* Zeiger auf Punkte-Array   */
   REGISTER UWORD i;

   punkte = objekt->punkte;    /* Adresse des Punkte-Arrays */

   /* Alle Punkte transformieren: */
   for (i=0; i < objekt->anz_pun; i++)
   {
      /* Transformationsroutine indirekt aufrufen: */
      (*operation)(&punkte[i], t1, t2, t3); /*1 Pkt transf.*/
   }
}


/* Skalierung eines Punktes: */
/*****************************/

VOID pun_ska(punkt, xs, ys, zs)
struct punkt *punkt;          /* Zeiger auf Punkt-Struktur */
WORD xs, ys, zs;              /* Skalierungsparameter      */

{
   punkt->x = (punkt->x * xs) / 10;  /* Skalierfaktor in   */
   punkt->y = (punkt->y * ys) / 10;  /* Zehntel            */
   punkt->z = (punkt->z * zs) / 10;
}


/* Translation eines Punktes: */
/******************************/

VOID pun_tra(punkt, xl, yl, zl)
struct punkt *punkt;
WORD xl, yl, zl;

{
   punkt->x += xl;
   punkt->y += yl;
   punkt->z += zl;
}
```

```
/* Rotation eines Punktes: */
/*************************/

VOID pun_rot(punkt, xr_w, yr_w, zr_w)
struct punkt *punkt;
WORD xr_w, yr_w, zr_w;          /* Rotationswinkel          */

{
   REGISTER LONG x,y,z;         /* Register-Vorschläge      */
   REGISTER LONG sin_w, cos_w;
   LONG zwis;

   x = punkt->x;        /* Register mit Koordinaten laden */
   y = punkt->y;
   z = punkt->z;

   /* Rotation um die x-Achse: */
   if (xr_w)                    /* x-Winkel != 0 ?    */
   {
      /* Sinus/Cosinus mal 2^15 aus Tabelle holen: */
      sin_w = sinus[xr_w];
      cos_w = sinus[ W_PLUS_90(xr_w) ];

      /* Drehmatrizen berechnen und durch 2^15 teilen */
      /* (Rückrechnung der Sinus-/Cosinuswerte)       */
      zwis = (y*cos_w - z*sin_w) >> 15;
      z    = (y*sin_w + z*cos_w) >> 15;
      y    = zwis;
   }

   /* Rotation um die y-Achse: */
   if (yr_w)                    /* y-Winkel != 0 ?    */
   {
      /* Sinus/Cosinus mal 2^15 aus Tabelle holen: */
      sin_w = sinus[yr_w];
      cos_w = sinus[ W_PLUS_90(yr_w) ];

      /* Drehmatrizen berechnen und durch 2^15 teilen */
      /* (Rückrechnung der Sinus-/Cosinuswerte)       */
      zwis = ( x*cos_w + z*sin_w) >> 15;
      z    = (-x*sin_w + z*cos_w) >> 15;
      x    = zwis;
   }
```

```
   /* Rotation um die z-Achse: */
   if (zr_w)                    /* z-Winkel != 0 ?   */
   {
      /* Sinus/Cosinus mal 2^15 aus Tabelle holen: */
      sin_w = sinus[zr_w];
      cos_w = sinus[ W_PLUS_90(zr_w) ];

      /* Drehmatrizen berechnen und durch 2^15 teilen */
      /* (Rückrechnung der Sinus-/Cosinuswerte)       */
      zwis = (x*cos_w - y*sin_w) >> 15;
      y    = (x*sin_w + y*cos_w) >> 15;
      x    = zwis;
   }
   punkt->x = x;          /* Koordinaten wieder zurück */
   punkt->y = y;
   punkt->z = z;
}

  /* Zentralprojektion eines Punktes */
/***********************************/

VOID pun_zen(punkt, xz, yz, zz)
struct punkt *punkt;      /* Zeiger auf Punktstruktur  */
WORD xz, yz, zz;          /* Koordinaten Beobachter    */

{
   REGISTER WORD zwis;

   if (zwis = punkt->z - zz)  /* nur bei zwis != 0    */
   {
      punkt->x = (LONG)xz - ( (LONG)zz * (LONG)(punkt->x - xz) )/zwis;
      punkt->y = (LONG)yz - ( (LONG)zz * (LONG)(punkt->y - yz) )/zwis;
      punkt->z = 0;
   }
   else
   {
      /* z-Koord. Punkt gleich z-Koordinate Beobachter: */
      punkt->x = xz;
      punkt->y = yz;
      punkt->z = 0;
   }
}
```

```
/* Objekt (verdeckt) auf Bildschirm zeichnen */
/*******************************************/

VOID obj_zeichnen(w, objekt)
struct welt *w;
struct objekt *objekt;    /* Zeiger auf zu              */
                          /* zeichnendes Objekt         */

{
   struct punkt *punkte; /* Zeiger auf Punkte-Array     */
   struct linie *linien; /* Zeiger auf Linien-Array     */
   REGISTER UWORD i;
   WORD x1, x2, y1, y2;  /* Endpunktkoordinaten         */

   if (objekt->sichtbar) /* Nur sichtbares Objekt       */
                         /* zeichnen                    */
   {
      punkte = objekt->punkte;
      linien = objekt->linien;

      /* Farbe setzen: */
      SetAPen( RastPort[Fen_Nr_v], (LONG)objekt->farbe );

      for (i=0; i < objekt->anz_lin; i++)
      {
         x1 = w->xe_tra + w->xe_ska * punkte[ linien[i].p1 ].x;
         y1 = w->ye_tra - w->ye_ska * punkte[ linien[i].p1 ].y;
         x2 = w->xe_tra + w->xe_ska * punkte[ linien[i].p2 ].x;
         y2 = w->ye_tra - w->ye_ska * punkte[ linien[i].p2 ].y;

         /* Linie zeichnen: */
         Move( RastPort[Fen_Nr_v], (LONG)x1, (LONG)y1 );
         Draw( RastPort[Fen_Nr_v], (LONG)x2, (LONG)y2 );
      }
   }
}
```

Starten Sie das Programm doch einfach einmal. Kaum wurde es geladen, öffnet sich ein neuer Bildschirm, die Farben wechseln, und es erscheint ein hübsches kleines Ferienhäuschen mit Türen und Fenstern in Vogelperspektive auf dem Monitor. Gehen Sie doch einmal näher heran: Betätigen Sie etwa 10–20mal die Plus-Taste (auf der amerikanischen Tastatur die Taste »]«). Das Ferienhäuschen wird zu einem stattlichen Ferienhaus. Mit der Minustaste des Ziffernblockes können Sie das Ganze wieder etwas auf Abstand bringen. Da Sie nun aber einmal davorstehen, sollten Sie es nicht versäumen, »einmal ums Haus zu gehen«: Drücken Sie kurz auf die Taste

<Cursor rechts>. Na, wer hätte das gedacht? Wir drehen uns tatsächlich ums Haus – oder dreht sich das Haus um uns oder um sich selbst? Drücken Sie doch noch einmal dieselbe Taste und noch einmal und noch einmal... Wird Ihnen schon schwindelig? Dann wäre es vielleicht besser, Sie betätigen einige Male die Taste <Cursor links> oder noch besser: <Enter> (Ziffernblock). Jetzt steht das Haus wieder.

Nehmen wir einen weiteren Anlauf: Je zweimal <Cursor links> und <Cursor auf> und <Shift> <Cursor rechts>. Drehungen um verschiedene Achsen überlagern sich. Und jetzt: Etwa 10mal die »4« des Ziffernblocks: Sie verlagern das Drehzentrum. Das Haus dreht sich nicht mehr um sich selbst, sondern um einen anderen Punkt usw. Probieren Sie doch ein wenig herum!

Neben den angesprochenen Tastenfunktionen bietet Ihnen das Programm eine Vielzahl von weiteren Eingriffsmöglichkeiten. (Das »Z« hinter einer Taste bedeutet: Diese Taste muß im Ziffernblock betätigt werden):

<Mausknopf re>	Positionieren der Objekte auf dem Bildschirm an die Mauszeigerposition
<Plus>	Vergrößern der Objekte
<Minus Z>	Verkleinern der Objekte
<4 Z>	Annähern der Objekte
<6 Z>	Entfernen der Objekte
<8 Z>	Entfernen des Beobachters (Fluchtpunkt)
<2 Z>	Annähern des Beobachters (Fluchtpunkt)
<Cursor re>	Drehung um die y-Achse rechts
<Cursor li>	Drehung um die y-Achse links
<Cursor auf>	Drehung um die x-Achse rechts
<Cursor ab>	Drehung um die x-Achse links
<Shift> <Crsr re>	Drehung um die z-Achse rechts
<Shift> <Crsr li>	Drehung um die z-Achse links
<Del>	Koordinatenkreuz ein/aus
<0 Z>	Koordinatenkreuz transformieren ein/aus
<Help>	alle Drehungen aus, Transformationen zurück auf die ursprünglichen Werte
<Enter Z>	alle Drehungen aus
<Punkt Z>	Double Buffering ein/aus
<Esc>	Programm beenden

Sie sollten jedoch vom Gebrauch des linken Mausknopfes Abstand nehmen, da es dann passieren kann, daß das Programm nicht mehr auf Ihre Eingaben reagiert (warum, das werden Sie sofort erfahren)! Sollte Ihnen das dennoch passiert sein, dann drücken Sie wiederholt so lange noch einmal auf den linken Mausknopf, bis das Programm wieder auf Sie zu sprechen ist.

Sicher werden Sie eine ganze Zeit Ihre Freude an dem Programm haben. Eines Tages jedoch gefällt Ihnen das Haus nicht mehr und es verlangt Sie nach einem Auto, einem Fahrrad oder einer ganzen Stadt. Nun, das Programm ist in der Beziehung äußerst flexibel. Es kann ohne Änderungen praktisch beliebig viele Objekte handhaben. Jedes Objekt kann seine eigene Farbe

besitzen, sichtbar oder unsichtbar sein etc. Jedes Objekt kann aus beliebig vielen Punkten und Linien bestehen. Das einzige, was Sie dafür machen müssen, ist, die notwendigen Daten (Objektbeschreibung, Punkte, Linien) in das Programm einzusetzen und neu zu kompilieren. Vielleicht lagern Sie die Daten ja auch aus (z.B. auf Disk) und können das eigentliche Programm stets unverändert halten.

Zur Zeit jedenfalls besteht die kleine Welt des Computers aus zwei Objekten: ein Koordinatenkreuz und ein Haus. Zur Zeit können auch nur alle Objekte gemeinsam oder einzelne Objekte gar nicht transformiert werden. Das liegt daran, daß die Transformationsdaten nur einmal in der Weltstruktur auftauchen. Wenn Sie diese (Rotation, Translation etc.) jeweils zusätzlich in die Objektstruktur jedes Objektes einsetzen, was programmtechnisch überhaupt kein Problem ist, dann können Sie jedes Objekt einzeln und getrennt für sich drehen, vergrößern usw. Die Koordinaten jedes Objektpunktes beziehen sich dann nur auf das objekteigene Koordinatensystem. Jedes Objektkoordinatensystem hat seine – ebenfalls variable – Position in der Welt. Selbstverständlich besitzt die Welt ihrerseits noch weiterhin ihre Transformationen und kann insgesamt gedreht, vergrößert werden etc.

Doch kommen wir zur eigentlichen Programmbeschreibung:

Was anderes sollte in einem Amiga-Programm zuerst passieren, als das Öffnen und Initialisieren der verschiedensten Systemkomponenten. Wir haben das alles bereits in einem früheren Kapitel kennengelernt. Da wären zunächst die Libraries. Das Programm benötigt – neben der bereits offenen Exec-Library – natürlich die Intuition- und die Graphics-Library. Vorsorglich für Ihre zukünftigen Erweiterungen an diesem Programm wird sogar noch – obwohl keinerlei Fließkommaarithmetik vorkommt – die MathFFP-Library geöffnet. Sie ist zuständig für die »kommahaltigen« Grundrechenarten.

Was dann folgt, kennen Sie ebenfalls bereits: Das Öffnen eines neuen Bildschirmes mit einem Fenster. Derer werden allerdings hier gleich zwei geöffnet. Das hängt mit dem sogenannten Double-Buffering, dem verdeckten Zeichnen zusammen, worauf wir sogleich zu sprechen kommen. Wundern Sie sich nicht, daß Sie keine Fenster sehen. Wir haben die Option **BORDERLESS** gewählt. Die Fenster besitzen also keinerlei Umrahmung. Ferner befinden sich keine Gadgets an den Fenstern. Sie sind also völlig unsichtbar. Mittels SetRGB4( ) werden gleich noch die Farben gesetzt. Erst dann kommt es zum eigentlichen Programmaufruf: **do_program( ).**

Bevor Sie sich dem allerdings zuwenden, sollten Sie noch kurz bei den Dingen verharren, die sich unter der Überschrift »Globale Variablen und Strukturen« befinden. Da wäre zunächst einmal eine riesige Tabelle: die Sinustabelle. Für die Drehmatrizen benötigen wir wohl oder übel eine Möglichkeit, Sinus und Cosinus von beliebigen Winkeln zu errechnen. Das könnten wir selbstverständlich sehr einfach mit Hilfe der Funktionen aus der transzendenten Mathematik-Library des Amiga bewerkstelligen. Aber wie lange wird das dauern? Wir möchten unsere Objekte schließlich einigermaßen schnell über den Bildschirm drehen. Nun, kein Problem. Da in dem Programm sowieso nur mit ganzzahligen Winkeln (also ohne Komma) gerechnet wird, erstellen wir einfach eine Tabelle mit 360 Werten (für die Winkel 0 bis 359). Jeder Wert gibt dabei den Sinus einer ganz bestimmten Gradzahl an. Sollte das Programm nun den Sinus

z.B. von 157 Grad benötigen, so schlägt es in dieser Tabelle nach und hat in Mikrosekundenschnelle den gewünschten Sinus. Eine komplizierte Berechnung ist nicht notwendig.

Mit der gleichen Tabelle können Sie übrigens auch den Cosinus eines Winkels ermitteln. Aus der Schule kennen Sie vielleicht noch die Gleichung:

cos(a) = sin(a+90)

oder in Radiant:

cos(a) = sin(a+PI/2)

Der Cosinus von 100 Grad ist also gleich dem Sinus von 100 + 90 = 190 Grad. 90 zum Winkel addiert, in der Sinustabelle nachgeschlagen, fertig. Worauf Sie allerdings achten müssen: Der Winkel darf nie größer als 359 Grad werden, da das die Tabelle sprengen würde. Das ist aber nicht so schwierig, denn 360 Grad sind genau eine Umdrehung, entspricht also 0 Grad. Wird ein Winkel also größer als 359 Grad, so ziehen wir einfach 360 ab.

Die Tabelle hätte auch viel kleiner ausfallen können. Ohne große Umrechnung hätten wir auch lediglich die Sinuswerte von 0 bis 179 oder gar von 0 bis 89 Grad erfassen brauchen. Aber der Speicher des Amiga ist ja groß, die Rechenzeit jedoch kostbar.

Wie aber speichern wir die Sinuswerte ab? Schließlich liegen alle Werte zwischen –1 und 1. Mit Komma zu rechnen kostet viel Zeit. Die Lösung: In der Tabelle sind eigentlich nicht direkt die Sinuswerte gespeichert. Vielmehr wurden alle Werte erst noch mit $2^{15} = 32768$ multipliziert. Das entspricht im Dualsystem einer Verschiebung um 15 Bit nach links. Damit passen sie genau in ein Wort (das oberste, 15. Bit, dient als Vorzeichenbit). Später im Programm wird das bei den Multiplikationen mit den Koordinaten eines Punktes wieder rückgängig gemacht (wenn auch ohne Rundung), dort wird (wohlgemerkt nach der Multiplikation!) wieder durch $2^{15} = 32768$ geteilt (15 Bit nach rechts geschoben). Diese Verschiebungen kann der Prozessor (auch in C) sehr schnell und einfach mit einem einzigen Befehl durchführen. Es ist also keine langwierige Division notwendig! Soweit zur Sinustabelle.

Es folgen die Strukturdefinitionen für einen Punkt, eine Linie, ein Objekt und die ganze Welt (ja, jeder Punkt, jede Linie ist eine Struktur, in der sich die Koordinaten bzw. die Nummern der Endpunkte befinden). Innerhalb der Objektstruktur befinden sich zwei Zeiger. Alle Punkte sowie alle Linien eines Objektes sind in zwei großen Arrays enthalten (jedes Array besteht also aus einer Vielzahl von Punkt- bzw. Linienstrukturen). Die Zeiger in der Objektstruktur zeigen eben auf diese Arrays. Weiterhin steht in dieser Struktur, aus wievielen Punkten und Linien sich das Objekt zusammensetzt, seine Farbe, ja, Sie können sogar einen Namen angeben. Vermerkt ist hier weiterhin, ob das Objekt auf dem Bildschirm überhaupt sichtbar und ob es den Welttransformationen unterworfen sein soll.

In der Weltstruktur findet sich schließlich ein Zeiger auf ein weiteres Array. Die Strukturen aller Objekte sind nämlich ebenfalls in einem Array zusammengefaßt. In der Weltstruktur erkennen Sie auch endlich die Werte für die verschiedenen Transformationen, die Koordinaten des Beobachters und die Ebenentransformationen für die Bildschirmanpassung.

Zwei weitere Arrays sollten nicht unerwähnt bleiben. Es sind dies ein zweites Objekt-Array und ein zweites Punkte-Array. (Die Namen: **trans_obj[ ]** (auch: _trans_obj[ ]) und _trans_pun[ ]). Sie sind haargenauso aufgebaut und auch genauso groß wie die oben beschriebenen. Sie werden zu Programmanfang sogar mit exakt denselben Werten beschrieben. Wenn allerdings ein Objekt mit allen seinen Punkten transformiert und projiziert werden soll, so werden nur in diesem zweiten Satz von Objekt- bzw. Punkte-Arrays die Zwischen- und Endergebnisse dieser Manipulationen gespeichert. Es wird also eigentlich nur mit dem zweiten Array-Satz gerechnet. Damit bleiben die ursprünglichen Punktkoordinaten stets erhalten.

Die Zusammenhänge veranschaulicht das folgende Bild:

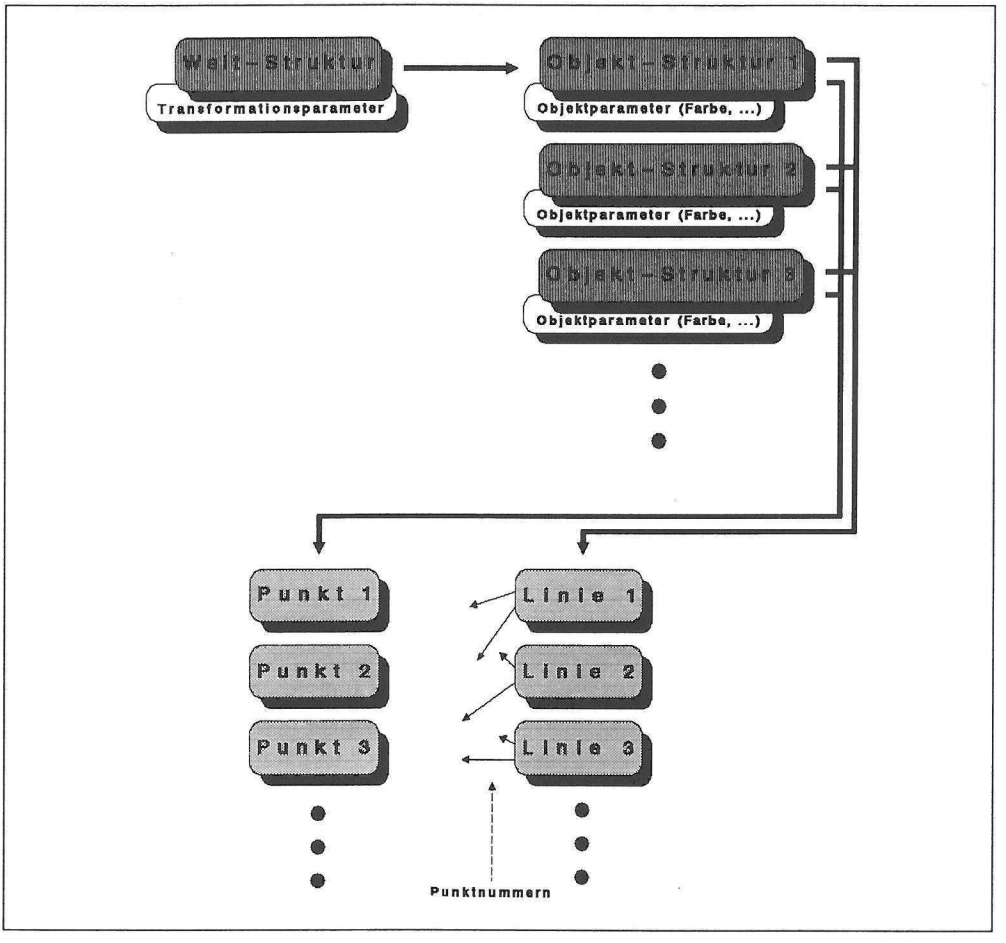


Bild 4.24: Datenorganisation des C-Programmes

Noch sind alle diese Strukturen nicht initialisiert (also mit Daten gefüllt). Das geschieht in der Funktion **schaffe_welt( )**, die gleich (fast) als erstes in **do_program( )** aufgerufen wird:

Bevor wir die Vorgänge in dieser Funktion verstehen, sollten wir uns zunächst mit der Dateneingabe für eine Weltsituation beschäftigen (schauen Sie also unter die Überschrift »Einrichtung aller Datenstrukturen«). Da befindet sich zunächst ein Array von Strukturen, die die neuen Objekte beschreiben (**new_objekt[ ]**). Hier geben Sie – falls Sie die Szenerie von Koordinatensystem und Haus ändern möchten – für jedes Objekt an, wie viele Punkte, wie viele Linien, welche Farbe und welchen Namen es hat. Die Anzahl der hier eingetragenen Array-Elemente bestimmt, wie viele Objekte die Welt hat (hier 2), die Reihenfolge bestimmt die Objektnummer.

Als nächstes kommt das Punkte-Array **_punkte[ ]**. Es enthält alle Punkte aller Objekte, geordnet in der Reihenfolge der Objekte. Die Reihenfolge der Punkte gibt definiert dabei die jeweilige Punktnummer. Für jedes Objekt beginnt die Numerierung allerdings wieder bei Null. In diesem Array tragen Sie die Koordinaten aller Ihrer Eckpunkte ein.

Das letzte Array, das Sie interessieren muß, wenn Sie neue Objekte eingliedern möchten, ist das Array aller Linien **_linien[ ]**. Hier sind die Nummern aller Start- und Endpunkte aller Objekte gespeichert. Der Aufbau ist ansonsten ähnlich dem Array _punkte[ ].

Jetzt werden Sie sicher die Funktion **schaffe_welt( )** besser verstehen. Hier nämlich wird das normale Objekt-Array (auf das der Funktion ein Zeiger übergeben wird) mit den notwendigen Werten aufgefüllt. Unter anderem mit den Adressen der Punkte- und Linienstarts für das jeweilige Objekt.

Jetzt, wo alles eingerichtet ist, kehren wir zurück zum Hauptprogramm **do_program( )**. Dort beginnt nämlich die große Hauptschleife, die bei jedem Neuzeichnen der Welt einmal durchlaufen wird, so lange, bis Sie die Taste <Esc> betätigt haben. Und was geschieht in der Schleife? Nun, nachdem die einzelnen Transformationen und die Projektion durchgeführt wurden, wird das Bild neu gezeichnet. Vorher aber ruft das Programm noch **init_ausgabe( )** auf. Hier werden die Arrays für Objekte und die für die Punkte in diejenigen Arrays übertragen, die wir eben als **trans_obj[ ]** und **_trans_pun[ ]** vorgestellt haben. Alle folgenden Rechnungen beziehen sich nur auf diese beiden Arrays.

Jetzt aber kann es richtig losgehen. Nacheinander werden die Funktionen für Skalierung, Translation, Rotation und Zentralprojektion aufgerufen. Alle führen die entsprechenden Matrixmultiplikationen mit allen Punkten der gesamten Welt aus. Alle werden aber auch über ein und dieselbe Routine aufgerufen: **welt_verteiler( )**. Ein übergebenes Flag zeigt an, welche Operation ausgeführt werden soll. Dieser Verteiler ruft nun für jedes einzelne Objekt wieder eine Routine auf, die die Operation ausführen soll.

Diese Routine ihrerseits ruft für jeden Punkt eines Objektes die Funktion auf, deren Adresse übergeben wurde (hier also die Transformationsroutinen). Jetzt endlich sind wir dort, wo ein einzelner Punkt transformiert wird. Es handelt sich um die Routinen **pun_ska( )**, **pun_tra( )**, **pun_rot( )** und **pun_zen( )**. Hier finden die tatsächlichen Matrizenmultiplikationen statt. Die Formeln werden Ihnen sicher bekannt vorkommen.

Sind endlich alle Punkte aller Objekte der Welt transformiert, kehrt das Programm wieder nach **do_program( )** zurück. Sind nun auch alle Transformationen ausgeführt, kann endlich gezeichnet werden. Vorher löschen wir noch eben den Bildschirm und dann geht die Post ab.

Gezeichnet wird ebenfalls über **welt_verteiler( )**, denn es soll ja die ganze Welt gezeichnet werden. Dann allerdings wird die Funktion **obj_zeichnen( )** angewählt, die ein ganzes Objekt in der richtigen Farbe zu Bild bringt.

Kaum sind wir zurück in **do_program( )**, überfällt uns schon ein neues Schicksal: »Verdecktes Fenster nach vorne bringen« steht da ganz lapidar im Kommentarteil, dann folgt ein Aufruf ScreenToFront( ), der einen Bildschirm zuoberst bringt. Ja, was denn jetzt? Ohne es bisher zu erwähnen, haben wir an verschiedenen Stellen im Programm eine Technik vorbereitet, deren Effekt Sie leicht testen können. Starten Sie das Programm noch einmal, betätigen Sie eine oder mehrere Cursortasten, so daß sich die Objekte zu drehen beginnen. Jetzt tippen Sie kurz auf den Dezimalpunkt Ihres Ziffernblockes. Sehen Sie den Unterschied? Das Bild bzw. die Objekte beginnen ziemlich stark zu flackern. Woran liegt das? Nun, irgendwann im Verlauf des Programmes müssen wir den Bildschirm löschen, um das neue Bild zu zeichnen. Bevor allerdings das gesamte neue Bild fertig ist, ist einige Zeit vergangen, in der sich nichts oder nur Teile des Objektes auf dem Bildschirm befanden. Der unerwünschte Effekt: Flackern.

Dem ist aber abzuhelfen. Der Trick heißt: Double Buffering oder zu gut deutsch: verdecktes Zeichnen. Ganz zu Anfang wurde bereits erwähnt, daß wir nicht einen, sondern zwei Screens mit je einem Fenster geöffnet haben. Es kann allerdings immer nur der oberste Screen betrachtet werden. Beim Zeichnen gehen wir nun so vor: Während auf dem sichtbaren Bildschirm das alte Bild unverändert verbleibt, wird im unsichtbaren Screen gelöscht, gezeichnet und sonstig gewerkelt. Erst wenn alles dort fertig ist, dann holen wir den bislang unsichtbaren Bildschirm eben mit ScreenToFront( ) nach vorne: keine Periode des Schwarzsehens, kein Flackern. Als nächstes wird der jetzt unsichtbare Bildschirm neu bearbeitet usw.

Im Programm geben die Variablen **Fen_Nr_s** und **Fen_Nr_v** die Nummer des sichtbaren bzw. unsichtbaren Bildschirms (Fensters) an. Diese Nummer bestimmt bei jeder Zeichenoperation, welcher Rasterport übergeben wird. Beim Bildwechsel wechseln wir auch die Nummern in diesen beiden Speichern aus. Mit dem Dezimalpunkt können Sie das Ganze aber auch abschalten. Schieben Sie doch einmal den Screen nach unten. Da ist er! Der zweite Screen. Zugegeben, es ist ein wenig unfein, für dieses Double-Buffering zwei Screens zu öffnen. Wir hätten auch einfach die Bitmaps des Fensters wechseln können. Das wäre allerdings ein wenig aufwendiger gewesen. So geht es doch auch (ein Nachteil: Sie dürfen nicht auf den linken Mausknopf kommen, da sonst das falsche Fenster aktiv werden könnte. Damit wäre das Programm von Ihren Eingaben abgeschnitten).

Aber weiter: Wir kommen zur Tastaturabfrage. Wird das Objekt zur Zeit nicht gedreht, so können wir mit **Wait( )** auf eine Taste warten (in der Fensterstruktur hatten wir angegeben, daß uns jeder Tasten- oder Maustastendruck gemeldet wird). Ansonsten überspringen wir die Warterei und stellen sofort mit **GetMsg( )** fest, ob eine Nachricht (Taste) angekommen ist. Die Nachricht besteht aus der IntuiMessage-Struktur mit:

Class	Message-Art (IDCMP-Flag)
Code	RAW-Code der Taste
Qualifier	gleichzeitig gedrückte Taste (Shift etc.)
Mouse_x	x-Position Maus (nur bei Mausknopf)
Mouse_y	y-Position Maus (nur bei Mausknopf)

Mit **ReplyMsg( )** geben wir die Message zurück. Anschließend führen wir die entsprechenden Aufgaben durch switch aus. Die Variable flag ist stets dann gleich Null, wenn nicht auf eine weitere Taste gewartet werden soll. Das ist dann der Fall, wenn eine gültige Taste betätigt wurde oder wenn das Objekt zur Zeit rotiert. Bei < Esc > endet die Routine und gibt einen Fehlercode an **main( )** zurück.

4.5.2 Rotation um beliebige Raumachsen

Nun können wir unsere Objekte bereits um die drei Koordinatenachsen drehen. Um aber einen allgemeinen Ausdruck zu erhalten, interessiert uns die Drehung um eine beliebige Achse im Raum. Das klingt zunächst ein wenig kompliziert, ist aber halb so schlimm, da wir hierzu eigentlich nur auf die bekannten 3-D-Transformationsmatrizen zurückgreifen müssen. Wir werden wieder einmal sehen, welchen Nutzen die Matrizenschreibweise für unsere Belange hat.

Zunächst aber müssen wir die Achse, um die sich alles drehen soll, definieren. Das unternehmen wir wieder mit der bekannten vektoriellen Geradengleichung:

$$P = P_0 + s*\vec{R}$$

in Parameterform also:

$$x = x_0 + s*R_x$$
$$y = y_0 + s*R_y$$
$$z = z_0 + s*R_z$$

wobei:

P – ein berechneter Punkt der Geraden mit den Koordinaten x,y,z
P_0 – ein beliebiger Punkt der Geraden mit den Koordinaten x_0,y_0,z_0
$\vec{R}$ – der Richtungsvektor mit den Parametern R_x,R_y,R_z
s – Verlängerungsfaktor des Richtungsvektors

Unsere Strategie entspricht ganz genau der Rotation um einen beliebigen Punkt in der Ebene (s. dort). Wir werden also versuchen, die Achse, um die gedreht werden soll, auf eine Koordinatenachse zu manövrieren. Dann können wir die gewünschte Drehung ganz einfach auf bekannte Drehungen um Koordinatenachsen zurückführen. Das werden wir erreichen, indem wir die Gerade zunächst so verschieben, daß sie mit einem beliebigen Punkt durch den Koordinaten-Nullpunkt verläuft. Als nächstes drehen wir sie so um die x-Achse, daß sie sich vollständig in der x-z-Ebene befindet. Es folgt eine Drehung um die y-Achse, so daß die Gerade mit der z-Achse zusammenfällt. Jetzt sind wir in der Lage, die eigentlich gewünschte Drehung auszuführen. Wir drehen das Objekt also mit dem gewünschten Winkel um die z-Achse. Zum Schluß müssen die beiden Drehungen um x- und y-Achse sowie die Translation wieder rückgängig gemacht werden – fertig. Die angeführte Abbildung 4.25 soll Ihnen den Zusammenhang noch einmal verdeutlichen.

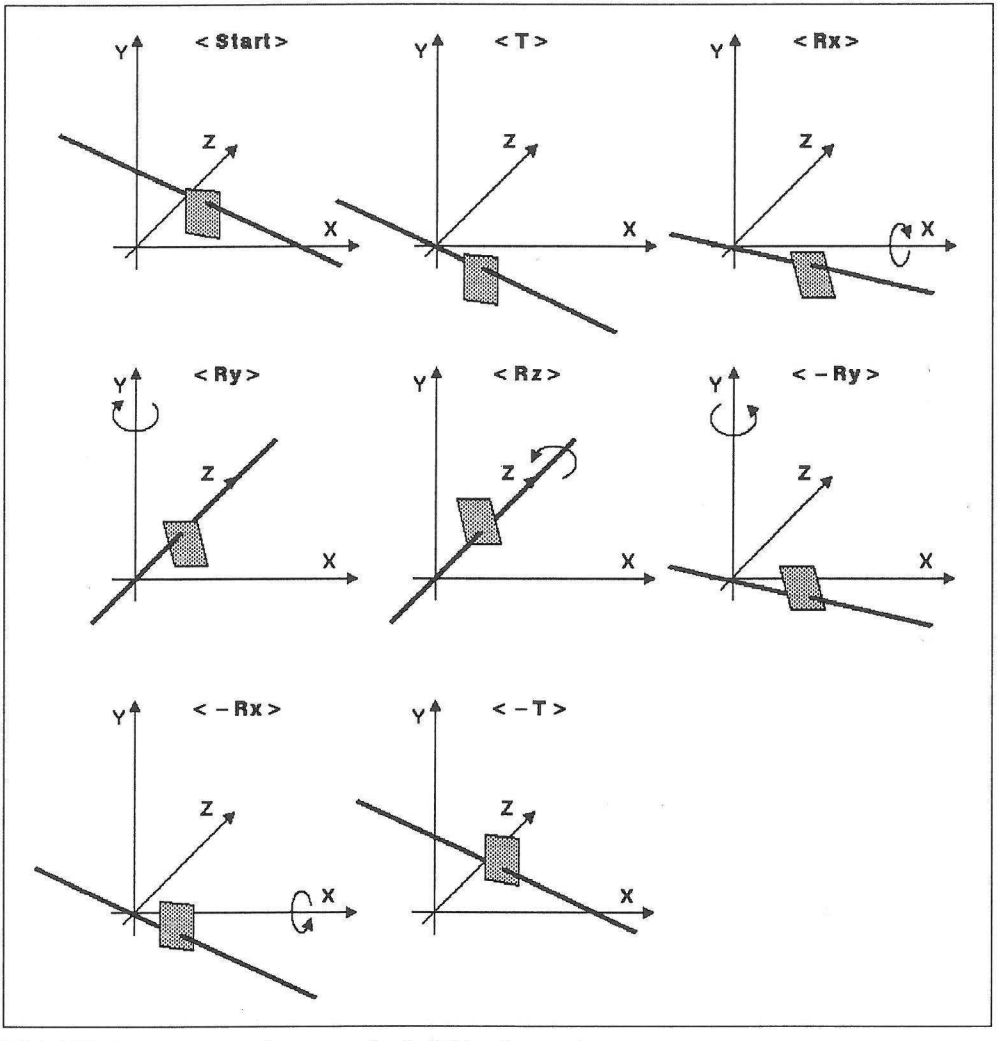


Bild 4.25: Schritte zur Drehung um eine beliebige Raumachse

Das Ganze beginnt also mit der Verschiebung der Linie in den Ursprung des Koordinatensystems. Dies erreichen wir mit der Translationsmatrix:

$$T(-x_0,-y_0,-z_0) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_0 & -y_0 & -z_0 & 1 \end{pmatrix}$$

Der Punkt $P_0(x_0,y_0,z_0)$ wird hier also in den Koordinaten-Nullpunkt verschoben. Am Ende unserer Drehung müssen wir diesen Punkt natürlich wieder mit der folgenden Translation zurückschieben:

$$T(x_0y_0,z_0) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_0 & y_0 & z_0 & 1 \end{pmatrix}$$

Der nächste Schritt wird ein wenig komplizierter: Die Gerade soll um die x-Achse auf die x-z-Ebene gedreht werden. Drehen können wir bereits perfekt, doch um welchen Winkel? Um das herauszubekommen, müssen wir ein wenig in unseren Geometrie-Kenntnissen kramen. Schauen Sie sich hierzu bitte einmal Bild 4.26 an.

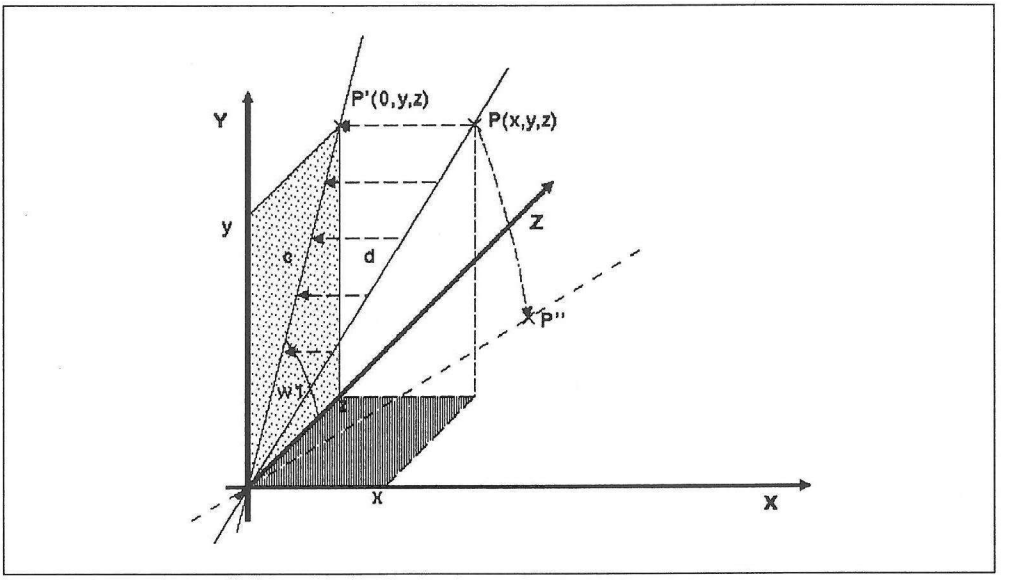


Bild 4.26: Drehung der Dreh-Achse auf die x-z-Ebene

Für die Bestimmung des Winkels wl projizieren wir die Gerade zunächst auf die y-z-Ebene. Beachten Sie, daß hier keine Drehung vorgenommen wird! Vielmehr haben wir es mit einer einfachen parallelen Projektion zu tun. Angenommen sei ein beliebiger Punkt der Geraden P(x,y,z). Von diesem Punkt ziehen wir also eine Parallele zur x-Achse. Der Punkt, in dem die Parallele die y-z-Ebene schneidet, ist der projizierte Punkt P' mit den Koordinaten (0,y,z). Der Winkel wl, der notwendig ist, um die Gerade auf die x-z-Ebene zu drehen, wird genausogut von der projizierten Geraden und der z-Achse eingeschlossen (s. Bild). Jetzt ist es allerdings viel einfacher, diesen Winkel zu berechnen.

Nach der Definition eines Winkels im rechtwinkligen Dreieck gilt nämlich (c ist die Verbindung: Nullpunkt -> Projektionspunkt P'):

$\sin(w_1) = y/c$
$\cos(w_1) = z/c$

c ist uns nicht völlig unbekannt, denn wir können für c nach dem Satz des Pythagoras folgendes einsetzen:

$$c^2 = y^2 + z^2$$

Damit könnten wir rein theoretisch den Winkel w_1 berechnen. Da das jedoch viel Rechenzeit benötigt (eine Wurzel und ein Arcussinus müßten berechnet werden), werden wir das allerdings vermeiden. Entwickeln wir also eine Transformationsmatrix für diese Drehung um die x-Achse. Normalerweise lautet die – wie bekannt:

$$R_x(w_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(w_1) & \sin(w_1) & 0 \\ 0 & -\sin(w_1) & \cos(w_1) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Für die Winkelfunktionen setzen wir jetzt die beiden obigen Gleichungen ein und erhalten:

$$R_x(w_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & z/c & y/c & 0 \\ 0 & -y/c & z/c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Sie sehen, wir benötigen gar keinen Winkel, müssen allerdings c durch Wurzelbildung errechnen. Die später benötigte Rücktransformation ist ebenfalls leicht ermittelt:

$$R_x(-w_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & z/c & -y/c & 0 \\ 0 & y/c & z/c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Nun liegt die Drehachse also in der x-z-Ebene. Der nächste Schritt wäre die Drehung dieser Geraden um die y-Achse auf die z-Achse. Das geht völlig analog zu unserer vorherigen Drehung, wie Sie der folgenden Zeichnung entnehmen können.

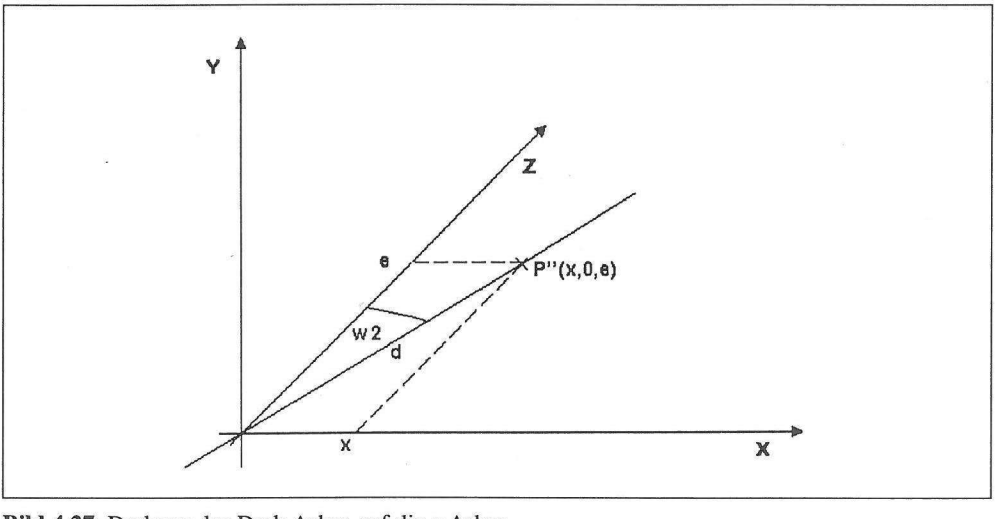


Bild 4.27: Drehung der Dreh-Achse auf die z-Achse

Auch hier ist wieder nach dem Drehwinkel w_2 gefragt.

Bei der bereits durchgeführten Drehung um die x-Achse ist die x-Koordinate des Punktes P logischerweise erhalten geblieben. Ebenfalls unverändert geblieben ist die Streckenlänge d zwischen dem Nullpunkt und P. Wir können sie berechnen aus (Satz des Pythagoras im Raum, betrachten Sie dazu noch einmal Bild 4.26)

$$\begin{aligned} d^2 &= x^2 + y^2 + z^2 \\ &= x^2 + c^2 \end{aligned}$$

(Auch hier berechnen Sie d selbst durch Ziehen der Wurzel aus dem rechten Teil der Gleichung). Sie können d also auch aus der seit der letzten Drehung bekannten Variablen c errechnen. Die y-Koordinate des gedrehten Punktes P'' ist selbstverständlich gleich 0 (schließlich liegt er in der x-z-Ebene). Die z-Koordinate errechnen wir leicht durch erneute Anwendung des pythagoräischen Lehrsatzes (s. Abbildung):

$$\begin{aligned} e^2 &= d^2 - x^2 \\ &= x^2 + c^2 - x^2 \\ &= c^2 \end{aligned}$$

e ist also gleich dem bereits oben berechneten c, was uns einige Rechenarbeit erspart. Um jetzt die gesuchte Rotationsmatrix zu finden, setzen wir nach Bild 4.27 auf:

$$\begin{aligned} \sin(w_2) &= x/d \\ \cos(w_2) &= c/d \end{aligned}$$

Die normale Rotationsmatrix für eine Drehung um die y-Achse sollte Ihnen bereits bekannt sein:

$$R_y(w_2) = \begin{pmatrix} \cos(w_2) & 0 & -\sin(w_2) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(w_2) & 0 & \cos(w_2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Mit den beiden letzten Gleichungen erhalten wir:

$$R_y(w_2) = \begin{pmatrix} c/d & 0 & -x/d & 0 \\ 0 & 1 & 0 & 0 \\ x/d & 0 & c/d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Und fertig ist die Rotationsmatrix. Wie oben, ersparen wir uns die Berechnung des Winkels w_2 per Arcussinus etc. Ermitteln wir noch eben die inverse Matrix:

$$R_y(-w_2) = \begin{pmatrix} c/d & 0 & x/d & 0 \\ 0 & 1 & 0 & 0 \\ -x/d & 0 & c/d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Endlich liegt die Rotationsachse auf der z-Achse des Koordinatensystems. Jetzt kann das Objekt ganz normal mit der Rotationsmatrix $R_z(a)$ um die z-Achse gedreht werden. Danach machen wir alle unsere Drehungen wieder rückgängig. Damit können wir die ganze Operation in einer langen Formel zusammenfassen:

$$R_{achse}(a) =$$

$$T(-x_0,-y_0,-z_0) * R_x(w_1) * R_y(w_2) * R_z(a) * R_y(-w_2) * R_x(-w_1) * T(x_0,y_0,z_0)$$

Beachten Sie, daß die Winkel w_1 und w_2 nie wirklich ausgerechnet werden müssen. Die Koordinaten x_0, y_0, z_0 geben einen Punkt auf der Drehachse an. Der Winkel a ist der eigentliche Drehwinkel um die Achse.

Selbstverständlich sind Sie in der Lage, aus diesen vielen Matrizenmultiplikationen eine einzige Matrix $R_{achse}(a)$ zu errechnen, die die gesamten Drehungen und Rückdrehungen etc. in einem durchführt. Mit dieser Matrix (ansonsten mit dem obigen Produkt) müssen Sie dann jeden zu drehenden Punkt multiplizieren.

KAPITEL 5

Verdeckte Linien und Flächen – das Problem und die Lösungen

Viel haben wir bereits geschafft. Wir sind in der Lage, komplexe räumliche Gebilde zu definieren, sie im Raum zu verschieben, zu vergrößern, um beliebige Achsen rotieren zu lassen, sie schließlich auf den Bildschirm zu projizieren unter Berücksichtigung von Fluchtpunkt, Beobachter etc. Eine Sache hat sicher auch Sie bisher stets gestört: Alle Objekte, ob klein, ob groß, waren durchsichtig. Die eigentlich verdeckten hinteren Kanten waren sichtbar, als hätten wir Modelle aus Draht in den Händen.

Die Darstellung dieser folgerichtig als Drahtmodelle bezeichneten Gebilde ist die einfachste Art und Weise der räumlichen Projektion, da es uns nicht zu interessieren braucht, welche Linien und Punkte wir nun tatsächlich zeichnen müssen und welche nicht. Um den räumlichen Eindruck jedoch noch perfekter zu gestalten, sollten wir Versuche unternehmen, hier die Initiative zu ergreifen.

Es existieren nun für die verschiedensten Anwendungen die unterschiedlichsten Algorithmen zur Verdeckung eigentlich versteckter Linien und Flächen. Auf den nächsten Seiten werden Sie einige kennenlernen.

5.1 Noch einfach: verdeckte Linien in 3-D-Funktionen

5.1.1 Das Prinzip

In diesem und in den nächsten Kapiteln wollen wir zwei Fliegen mit einer Klappe schlagen: Zum einen werden wir näher an unser Ziel der verdeckten Linien gelangen, zum anderen lernen Sie eine interessante Möglichkeit kennen, schöne Bilder durch dreidimensionale Funktionen zu erzeugen.

Normale zweidimensionale Funktionen sind Ihnen sicherlich ein Begriff. Wer hat nicht schon einmal eine Gleichung mit zwei Unbekannten (z.B. eine Geradengleichung) gesehen? Allgemein kann eine solche Funktion wie folgt ausgedrückt werden:

$$y = f(x)$$

Rechts steht dann ein beliebiger Ausdruck mit allen möglichen und unmöglichen Rechnungen, Sinus-, Logarithmus-Funktionen etc., die alle nur die eine Unbekannte x enthalten. Das Ergebnis eines solchen sogenannten Terms ist dann der Wert, der für y eingesetzt wird. Das Ergebnis wird aber davon abhängen, welchen Wert Sie für x einsetzen werden. Man sagt: y ist abhängig von x. Ein sehr einfaches Beispiel ist:

$$y = x$$

In diesem Fall nimmt y grundsätzlich den gleichen Wert an wie x.

Eine solche Funktion läßt sich sehr schön anschaulich auch grafisch darstellen. Hierzu verwendet man ein zweidimensionales Koordinatensystem. Auf der einen Achse zählen wir die y-Werte ab, auf der anderen die x-Werte. Das Ergebnis ist dann eine sogenannte Kurve (in dem obigen Fall eine Diagonale, die durch den Nullpunkt geht). Aber warum erzähle ich Ihnen das? Das wissen Sie ja bereits.

Jetzt wird es allerdings interessant: Es gibt nämlich auch Funktionen, die nicht mit den zwei üblichen Parametern x und y auskommen. In diesen Funktionen kommt ein dritter Parameter z ins Spiel. Solche Funktionen haben dann die Form:

$y = f(x,z)$

Der Funktionswert y hängt also von den zwei Variablen x und z ab (in manchen Büchern wird die Form auch mit $z = f(x,y)$ angegeben). Bei der zeichnerischen Darstellung solcher Funktionen kommen wir mit der einfachen zweidimensionalen Zeichentechnik nicht mehr aus. Wir führen vielmehr eine z-Achse ein, die uns die Dimension des Raumes eröffnet. Dabei müssen wir zwangsläufig auf die 3-D-Formeln zurückgreifen, die wir in den vorherigen Kapiteln kennengelernt haben.

In der Tat beschränkt sich diese Anwendung nicht nur auf das bloße Vergnügen der Betrachtung. Auf diese Weise lassen sich vielmehr recht komplexe Zusammenhänge anschaulich in einem Diagramm darstellen, die Sie sonst aus vielen einzelnen Bildern erahnen müßten. Nicht nur mathematische Gleichungen nämlich eignen sich für die hier besprochenen Wege, auch ganz normale statistische Tabellen sind streng mathematisch Funktionen, die auf diese Weise dargestellt werden können. Der einzige Unterschied: Bei Tabellen werden die einzelnen Werte nicht berechnet wie bei den Gleichungen.

Nehmen wir ein Beispiel: Angenommen, es interessieren Sie die jährlichen Umsätze Ihres Unternehmens (oder Ihrer Haushaltskasse). Nichts einfacher als das: Sie werden zunächst eine Tabelle anfertigen, in der Sie das Jahr und den dazugehörigen Umsatz niederschreiben. Alsdann folgt die Zeichnung mit den Jahren auf der waagerechten x-Achse, mit den Umsätzen auf der senkrechten y-Achse. Jeder y-Wert (Umsatz) ist damit abhängig vom zugehörigen x-Wert (Jahr).

Dreidimensionale Diagramme können gefordert sein, wenn Sie gleichzeitig noch die Umsätze Ihres Konkurrenzunternehmens bzw. Ihrer Familienmitglieder aufzeichnen möchten. Sie führen in diesem Fall die in den Raum hineinreichende z-Achse ein, auf der Sie dann die einzelnen zu betrachtenden Parteien auftragen. Auf diese Weise erhalten Sie einen optimalen Überblick über die Verhältnisse und können Ihrem Sohn, Ihrer Tochter oder Ihrem Mann rechtzeitig das Taschengeld kürzen.

Doch wenden wir uns wieder dem Hauptproblem zu. In unserem Falle werden wir die Techniken anhand mathematischer Gleichungen (Funktionen) kennenlernen, bei denen die einzelnen Funktionswerte errechnet werden. Das erspart uns aufwendige Tabellenerstellungen.

Den Graphen (oder die Kurve) einer Funktion erhalten wir durch die Erstellung einer sogenannten Wertetabelle. In dieser Tabelle richten wir im Geiste drei Spalten für x-, y- und z-Werte ein. In unserem Beispiel werden wir zunächst mit der folgenden Funktion hantieren:

$y = f(x,z) = x^2 + z^2$

Wie können wir die Berechnung dieser Funktion am effektivsten programmtechnisch realisieren? Hätten wir lediglich eine Funktion mit zwei Unbekannten (etwa $y = f(x) = x^2$) darzustellen, wäre das Problem recht einfach zu lösen. Unsere Aufgabe wäre es, einfach in einer FOR...NEXT-Schleife den Wert x von einem bestimmten Startpunkt aus bis zu einem Endwert

in festen Schritten hochzuzählen. Bei jedem Schleifendurchlauf berechnen wir dann in Abhängigkeit von x den fehlenden y-Wert. Die erhaltenen Wertepaare lassen sich dann sehr einfach auf den Bildschirm zeichnen. Das Schema eines solchen 2-D-Funktionsplotters:

```
FOR x = <Startwert> TO <Endwert> STEP <Schrittweite>
  y = f(x)
  PLOT x,y
NEXT x
```

Bei dreidimensionalen Funktionen hingegen haben wir zwei Variablen x und z, aus denen die dritte y ermittelt werden muß. In diesem Fall helfen wir uns mit zwei ineinander geschachtelten FOR...NEXT-Schleifen. Die eine zählt die x-Werte, die andere die z-Werte hoch:

```
FOR z = <z-Startwert> TO <z-Endwert> STEP <z-Schrittweite>
  FOR x = <x-Startwert> TO <x-Endwert> STEP <x-Schrittweite>
    y = f(x,z)
    PLOT x,y,z
  NEXT x
NEXT z
```

In diesem Fall haben wir z in der äußeren, x in der inneren Schleife verwandt. Das ist natürlich nicht unbedingt notwendig, empfiehlt sich aber oft, wie Sie sehen werden, beim Zeichnen von dreidimensionalen Objekten.

Den Befehl PLOT x,y,z werden Sie kaum in einem Basic-Sprachschatz finden. Einen dreidimensionalen Punkt müssen wir schließlich zunächst per Zentralprojektion unter Berücksichtigung aller anderen Transformationen auf die Ebene projizieren. Das folgende kleine Basic-Programm soll Ihnen das Prinzip verdeutlichen:

```
' ************************
' **                    **
' **  3-D-Funktionen 1  **
' **                    **
' ************************

PI = 3.141593

SCREEN 2,640,200,2,2          ' Neuer Bildschirm
WINDOW 4,,(0,0)-(631,186),0,2 ' Neues Fenster

PALETTE 0,  0,  0,  0         ' Farben setzen
PALETTE 1, .8,  0,.93
PALETTE 2,.47,.87,  1
PALETTE 3,  1, .6,.67

COLOR 2
```

```
' Transformationsparameter:
' Skalierung:
sx = 40
sy = 6
sz = 20

' Translation:
tx = 0
ty = 0
tz = 0

' Rotation:
rx = 15
ry = -45
rz = 0

' Beobachter:
bx = 0
by = 0
bz = -300

' Ebenentranslation:
tex = 300
tey = 90

' Konstanten für die Drehungen:
rx = rx*PI/180
ry = ry*PI/180
rz = rz*PI/180

si.x = SIN(rx) : co.x = COS(rx)
si.y = SIN(ry) : co.y = COS(ry)
si.z = SIN(rz) : co.z = COS(rz)

A = co.y * co.z
B = co.y * si.z
C = -si.y
D = si.x*si.y*co.z - co.x*si.z
E = si.x*si.y*si.z + co.x*co.z
F = si.x*co.y
G = co.x*si.y*co.z + si.x*si.z
H = co.x*si.y*si.z - si.x*co.z
I = co.x*co.y
```

```
' Start-/Endwerte/Schrittweiten:
sta.x = 3 : en.x = -3 : ste.x = -.01
sta.z = 4 : en.z = -3 : ste.z = -.5

' Funktion:
DEF FNfun(x,z) = x*x - z*z


' Zeichnung:

FOR z=sta.z TO en.z STEP ste.z
  FOR x=sta.x TO en.x STEP ste.x

    ' Funktionswert berechnen:
    y = FNfun(x,z)

    ' Transformationen:
    xt = sx*x + tx          ' Skalierung
    yt = sy*y + ty          ' und Translation
    zt = sz*z + tz

    xt = xt*A + yt*B + zt*C ' Rotationen
    yt = xt*D + yt*E + zt*F
    zt = xt*G + yt*H + zt*I

    ' Zentralprojektion:
    zwis = zt - bz
    xe = bx - bz * (xt-bx)/zwis
    ye = by - bz * (yt-bx)/zwis

    ' Zeichnen:
    PSET (tex + xe, tey - ye)

  NEXT x
NEXT z

WHILE (INKEY$ = "")
WEND

WINDOW CLOSE 4
SCREEN CLOSE 2
```

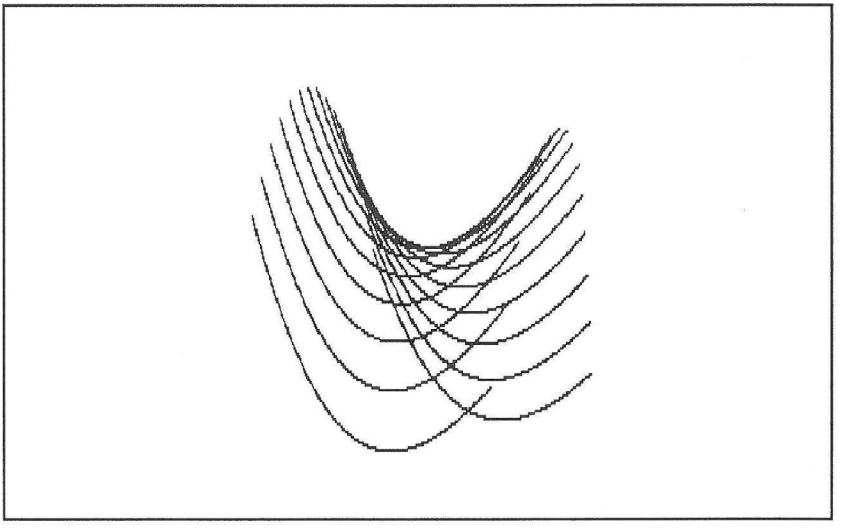


Bild 5.1: 3-D-Funktionen 1

Nach dem Öffnen eines neuen Bildschirms mit Fenster, nimmt das Programm einige Initialisierungen vor, die vor allem die verschiedenen Parameter für die Transformationen betreffen. Da die Routine in der Lage sein soll, den Graphen der Funktion um alle drei Raumachsen zu drehen, verwenden wir die im letzten Kapitel vorgestellte Gesamt-Rotationsmatrix für Rotationen um x-, y- und z-Achsen. Hierzu berechnen wir zuallererst die Rotationsfaktoren A, B, C, D, E, F, G, H und I, die über das gesamte Programm hindurch konstant bleiben (wir ersparen uns dadurch einige unnötige Berechnungen in den weiter unten liegenden Schleifen).

Bei diesen Transformationsparametern haben Sie selbstverständlich jeden Spielraum für Änderungen. Die momentan eingestellten Werte bringen eine recht schöne Grafik zu »Papier«.

Alsdann stellt das Programm die Start- und Endwerte sowie die Schrittweiten für die x- und z-Koordinaten ein, die in den folgenden FOR...NEXT-Schleifen hoch-(bzw. ab-)gezählt werden. Wenn Sie zunächst einen groben Überblick über das Bild bekommen möchten, dann empfiehlt es sich, ste.x (also die x-Schrittweite) um den Faktor von 10 bis 30 zu erhöhen, da die Erstellung der Zeichnung normalerweise einige Zeit in Anspruch nimmt.

Die Struktur der beiden ineinandergeschachtelten FOR...NEXT-Schleifen haben Sie bereits kennengelernt. Hier werden die entsprechenden Funktionswerte berechnet, vergrößert, verschoben, gedreht und per Zentralprojektion projiziert, anschließend bei den Ergebniskoordinaten ein Punkt gezeichnet. Das Ergebnis haben Sie nach einiger Zeit vor Augen. Ein Tastendruck genügt, um das Programm zu beenden.

5.1.2 Vernetzung (Crosshatching)

Noch wird Sie das Produkt unseres Programmes nicht besonders zufriedenstellen, geschweige denn beeindrucken. Aber lassen Sie uns etwas Zeit. Durch einen kleinen Trick ist es uns nämlich möglich, einen ganz besonders interessanten Effekt zu erreichen. Bisher stellen wir nur für bestimmte z-Werte einzelne Kurven dar. Aus der Fachliteratur jedoch kennen Sie vielleicht die schönen gebogenen Flächen, die ein Netzmuster ziert, das den räumlichen Eindruck besonders verstärkt.

Eine solche Vernetzung (Crosshatching) erreichen wir durch das Zeichnen einer um 90 Grad um die y-Achse gedrehten Kurvenschar auf das Gebilde von oben. Es handelt sich aber nur scheinbar um eine Drehung, denn das Gebilde bleibt in seiner Position eigentlich stehen, scheinbar deshalb, da einfach die Schrittweiten für die STEP-Kommandos der z- und x-Schleifen ausgetauscht werden. Jetzt stehen die Kurven genau senkrecht zu den zuerst gezeichneten. Das Ergebnis läßt sich bereits sehen!

```
' ************************
' **                    **
' **  3-D-Funktionen 2  **
' **                    **
' ************************

PI = 3.141593

SCREEN 2,640,200,2,2           ' Neuer Bildschirm
WINDOW 4,,(0,0)-(631,186),0,2 ' Neues Fenster

PALETTE 0,  0,  0,  0          ' Farben setzen
PALETTE 1, .8,  0,.93
PALETTE 2,.47,.87,  1
PALETTE 3,  1, .6,.67

COLOR 2

' Transformationsparameter:
' Skalierung:
sx = 40
sy = 6
sz = 20

' Translation:
tx = 0
ty = 0
tz = 0
```

```
' Rotation:
rx = 15
ry = -45
rz = 0

' Beobachter:
bx = 0
by = 0
bz = -300

' Ebenentranslation:
tex = 300
tey = 90

' Konstanten für die Drehungen:
rx = rx*PI/180
ry = ry*PI/180
rz = rz*PI/180

si.x = SIN(rx) : co.x = COS(rx)
si.y = SIN(ry) : co.y = COS(ry)
si.z = SIN(rz) : co.z = COS(rz)

A = co.y * co.z
B = co.y * si.z
C = -si.y
D = si.x*si.y*co.z - co.x*si.z
E = si.x*si.y*si.z + co.x*co.z
F = si.x*co.y
G = co.x*si.y*co.z + si.x*si.z
H = co.x*si.y*si.z - si.x*co.z
I = co.x*co.y

' Start-/Endwerte/Schrittweiten:
sta.x = 3 : en.x = -3 : ste.x = -.01
sta.z = 4 : en.z = -3 : ste.z = -.5

' Funktion:
DEF FNfun(x,z) = x*x - z*z


' Zeichnung:
```

```
FOR za = 1 TO 2

  FOR z=sta.z TO en.z STEP ste.z
    FOR x=sta.x TO en.x STEP ste.x

      ' Funktionswert berechnen:
      y = FNfun(x,z)

      ' Transformationen:
      xt = sx*x + tx          ' Skalierung
      yt = sy*y + ty          ' und Translation
      zt = sz*z + tz

      xt = xt*A + yt*B + zt*C ' Rotationen
      yt = xt*D + yt*E + zt*F
      zt = xt*G + yt*H + zt*I

      ' Zentralprojektion:
      zwis = zt - bz
      xe = bx - bz * (xt-bx)/zwis
      ye = by - bz * (yt-bx)/zwis

      ' Zeichnen:
      PSET (tex + xe, tey - ye)

    NEXT x
  NEXT z

  IF za = 1 THEN       ' Bei erstem Durchlauf werden
    zwis = ste.x       ' die Schrittweiten ausgetauscht
    ste.x = ste.z
    ste.z = zwis

    COLOR 3
  END IF

NEXT za

WHILE (INKEY$ = "")
WEND

WINDOW CLOSE 4
SCREEN CLOSE 2
```

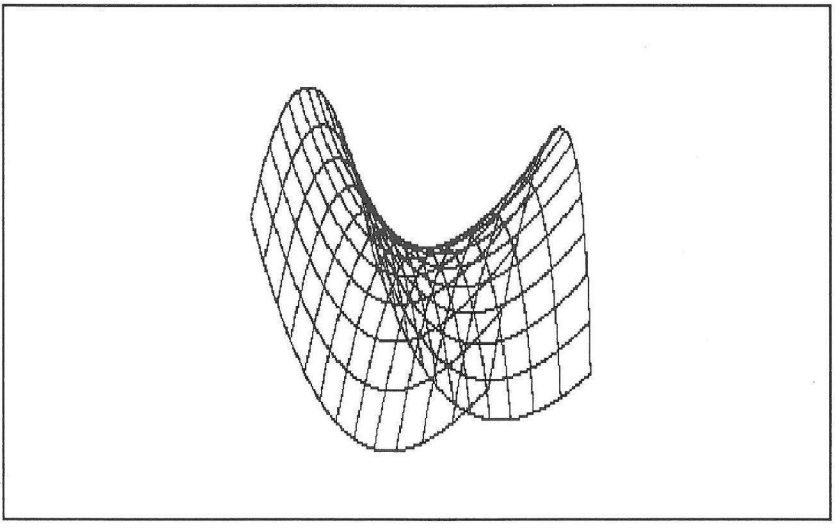


Bild 5.2: 3-D-Funktionen 2

Das Programm hat sich nicht sehr geändert. Lediglich eine alles umrahmende dritte FOR...NEXT-Schleife ist hinzugekommen. Der Graph wird also insgesamt zweimal auf zwei verschiedene Arten gezeichnet. Wenn Sie das Ergebnis immer noch zufriedenstellen sollte, dann lesen Sie gleich weiter.

5.1.3 Die verdeckten Linien

Bisher sind wir mit unseren Funktionen nicht über den Stand hinausgekommen, den wir bereits im letzten Kapitel mit allgemeinen Objekten erreicht hatten. Noch ist der dreidimensionale Graph durchsichtig, eigentlich verdeckte Linien sichtbar. Wir werden uns nun ein sehr einfaches Verfahren anschauen, diesem Makel ein Ende zu setzen.

Im täglichen Leben können wir normalerweise nicht durch die Dinge schauen, die wir betrachten, sofern sie nicht aus Glas sind. Wie Sie jedoch beim Zeichnen unserer Funktion sehen, ist dies hier der Fall. Unsere Linien durchdringen sich, obwohl die vorderen, die ja eine Fläche abstecken sollen, die hinteren verdecken müßten.

Wir machen uns dabei zunutze, daß wir die Zeichnung mit hohen z-Werten beginnen (also ganz weit weg) und den Graphen sozusagen von hinten nach vorne erstellen. Wir denken uns jeden z-Wert als eine Fläche parallel zur x-y-Ebene. Wenn wir nun diese Flächen von hinten nach vorne zeichnen, dann verdecken die vorderen stets die hinteren. Im Programm werden wir also stets die gesamte Fläche löschen, wodurch dann alle dahinterliegenden Flächen eliminiert werden. Wir löschen also vom gerade berechneten und gezeichneten Punkt der Kurve in einer senkrechten Linie bis zum unteren Fensterrand.

So erhalten wir eine Aufsicht auf die grafische Struktur der Funktion. Wenn Sie in das obige Programm ein paar Zeilen hinzufügen, wird Ihnen der Effekt deutlich:

```
' ************************
' **                    **
' **  3-D-Funktionen 3  **
' **                    **
' ************************

IF FRE(-1)+FRE(0) < 36000& THEN
  END
END IF

CLEAR ,36000&                  ' 36000 Byte für Amiga-Basic

DIM fenster%(14962)            ' Speicher für Bildspeicherung

PI = 3.141593

SCREEN 2,640,200,2,2           ' Neuer Bildschirm
WINDOW 4,,(0,0)-(631,186),0,2  ' Neues Fenster

PALETTE 0,  0,  0,  0          ' Farben setzen
PALETTE 1, .8,  0,.93
PALETTE 2,.47,.87,  1
PALETTE 3,  1, .6,.67

COLOR 2

' Transformationsparameter:
' Skalierung:
sx = 40
sy = 6
sz = 20

' Translation:
tx = 0
ty = 0
tz = 0

' Rotation:
rx = 15
ry = -45
rz = 0
```

```
' Beobachter:
bx = 0
by = 0
bz = -300

' Ebenentranslation:
tex = 300
tey = 90

' Konstanten für die Drehungen:
rx = rx*PI/180
ry = ry*PI/180
rz = rz*PI/180

si.x = SIN(rx) : co.x = COS(rx)
si.y = SIN(ry) : co.y = COS(ry)
si.z = SIN(rz) : co.z = COS(rz)

A = co.y * co.z
B = co.y * si.z
C = -si.y
D = si.x*si.y*co.z - co.x*si.z
E = si.x*si.y*si.z + co.x*co.z
F = si.x*co.y
G = co.x*si.y*co.z + si.x*si.z
H = co.x*si.y*si.z - si.x*co.z
I = co.x*co.y

' Start-/Endwerte/Schrittweiten:
sta.x = 3 : en.x = -3 : ste.x = -.01
sta.z = 4 : en.z = -3 : ste.z = -.5

' Funktion:
DEF FNfun(x,z) = x*x - z*z


' Zeichnung:

FOR za = 1 TO 2

  FOR z=sta.z TO en.z STEP ste.z
    FOR x=sta.x TO en.x STEP ste.x

      ' Funktionswert berechnen:
      y = FNfun(x,z)
```

```
      ' Transformationen:
      xt = sx*x + tx          ' Skalierung
      yt = sy*y + ty          ' und Translation
      zt = sz*z + tz

      xt = xt*A + yt*B + zt*C ' Rotationen
      yt = xt*D + yt*E + zt*F
      zt = xt*G + yt*H + zt*I

      ' Zentralprojektion:
      zwis = zt - bz
      xe = bx - bz * (xt-bx)/zwis
      ye = by - bz * (yt-bx)/zwis

      ' Zeichnen:
      PSET (tex + xe, tey - ye)
      LINE (tex+xe,tey-ye+1)-(tex+xe,200),0

    NEXT x
  NEXT z

  IF za = 1 THEN        ' Bei erstem Durchlauf werden
    zwis = ste.x        ' die Schrittweiten ausgetauscht
    ste.x = ste.z
    ste.z = zwis

    ' Fenster zwischenspeichern:
    GET (0,0)-(631,186),fenster%
    COLOR 3
    CLS                 ' und löschen
  END IF

NEXT za

PUT (0,0),fenster%,OR ' mit zweitem Fenster ORen

WHILE (INKEY$ = "")
WEND

WINDOW CLOSE 4
SCREEN CLOSE 2
```

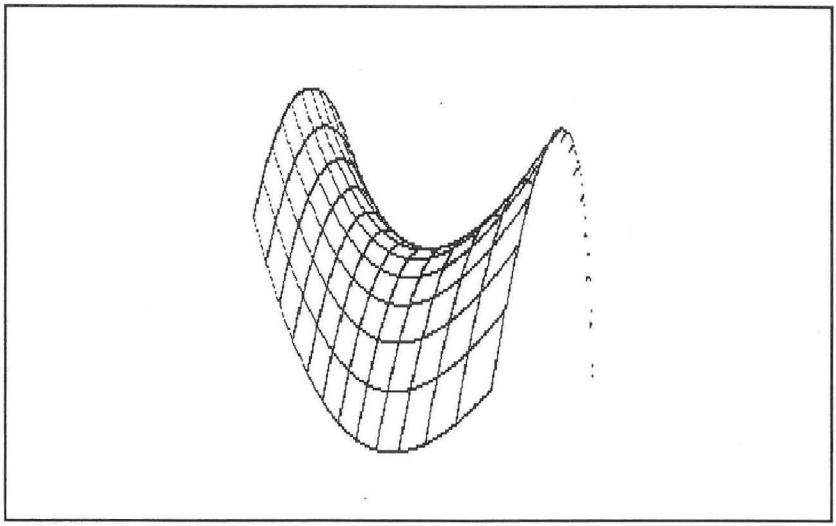


Bild 5.3: 3-D-Funktionen 3

Was wird gemacht? Nun, in diesem Programm werden nicht beide Bildteile einfach übereinandergezeichnet. Vielmehr speichert die Routine den ersten Teil zunächst zwischen. Dazu dient der GET-Befehl. Er überträgt den gesamten Fensterinhalt in das Integer-Array fenster%(), das zu Programmanfang entsprechend groß gewählt wurde (siehe Amiga-Basic-Handbuch unter GET). Dazu mußte allerdings erst genügend Speicher reserviert werden (CLEAR).

Der erste Teil wird also erstellt, die Punkte unter jedem Punkt mit einem LINE-Befehl gelöscht und das Ergebnis dann mittels GET zwischengespeichert. Der Bildschirm leert sich wieder und der zweite Teil der Grafik entsteht auf die gleiche Weise. Ist am Ende alles fertig, dann tritt der Befehl PUT in Aktion. Er verknüpft die beiden Grafiken per ODER-Funktion zu einer einzigen.

Was vielleicht noch nicht mit hundertprozentiger Zufriedenheit gelöst wurde, sind die Randbereiche. Man könnte sich wünschen, quasi auch von unten auf die Kurve sehen zu können. Dies läßt der hier angewandte Algorithmus allerdings nicht zu. Um dieses Manko auszugleichen, könnte man nun nicht bis zum unteren Bildrand, sondern lediglich zum darunterliegenden Punkt der nächsten Linie löschen. Damit wäre das Problem erledigt. Das Zeichnen würde mit dieser Methode jedoch um einiges länger dauern und erforderte mehrmaliges Zwischenspeichern. Vielleicht versuchen Sie einmal, das obige Programm so umzuschreiben.

Eine Idee zur Optimierung des Löschprozesses: Merken Sie sich jeweils die momentan niedrigste bisher gezeichnete y-Ebenen-Koordinate und löschen Sie nur bis dorthin.

Einen anderen noch interessanteren Algorithmus zum Löschen verdeckter Linien in 3-D-Funktionen stellen wir Ihnen im nächsten Abschnitt vor.

Es gibt einige schöne Funktionen, die Sie ausprobieren sollten. Versuchen Sie es evtl. mit jenen:

$y = x^2+z^2$
$y = 1/(1+x^2+z^2)$
$y = SQR(1-x^2/4-z^2/9)$
$y = 20*SIN(0.1*SQR(x^2+z^2)$
$y = SIN(x)/x+SIN(z)/z$
$y = SIN(1/x)/x+SIN(1/z)/z$

5.2 Kleines Intermezzo: ein komfortabler Funktionsplotter

Wir sind jetzt in der Lage, schöne dreidimensionale Funktionen auf dem Bildschirm darzustellen. Was uns noch fehlt, ist ein universales Programm, das sich für jede beliebige Funktion eignet und die Darstellung vielleicht noch ein wenig verfeinert. Was liegt da näher, als daß Sie sich mit dem folgenden Programm beschäftigen, einem weiteren kleinen Höhepunkt dieses Buches.

Starten Sie es ruhig einmal, und lassen Sie sich berauschen von der einzigartigen Schönheit der Amiga-Grafik:

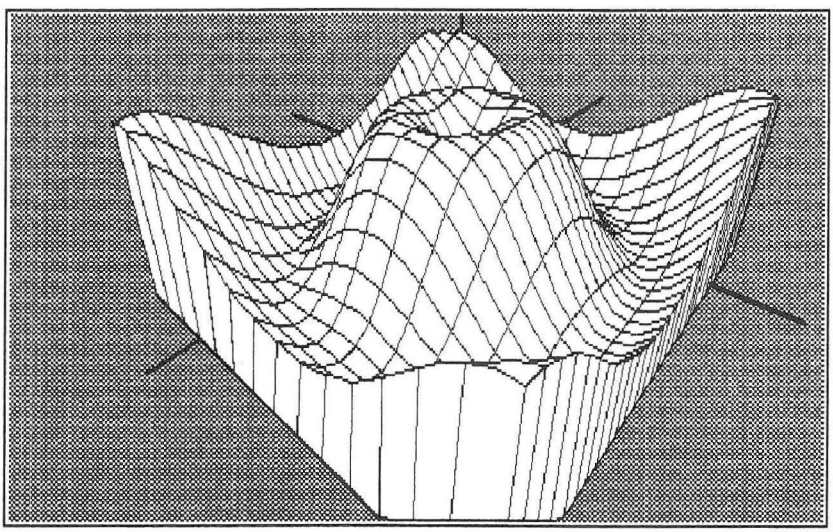


Bild 5.4: 3-D-Funktionsplotter 1

```
'*****************************
'**                         **
'**       Der große         **
'**  3-D-Funktionsplotter   **
'**                         **
'*****************************

Öffnen von Screen und Fenster / Setzen der Farben:
  SCREEN 2,640,200,3,2
wx% = 631
wy% = 186+10
WINDOW 4,,(0,0)-(wx%,wy%-10),0,2


PALETTE 0,  0,  0,  0
PALETTE 1, .8,  0,.93 'Farbe Hintergrundmuster
PALETTE 2,.47,.87,  1 'Farbe Netz
PALETTE 3,  1,.47,.53 'Farbe Netzfläche oben
PALETTE 4, .4, .6,  1 'Farbe Koordinatenkreuz
PALETTE 5,  1, .6,.67 'Farbe Rahmen vorne
PALETTE 6, .8,.33, .4 'Farbe Rahmen Seite

COLOR 2


'Das folgende Flag gibt an, ob Sie die
'verschiedenen Parameter per Input (flag%=0)
'oder direkt im Programm eingeben möchten (flag%=1):

flag% = 1


'Hier geben Sie die gewünschte 3-D-Funktion
'f(x,z) ein:

 DEF FNfun(x,z) = 20*SIN(.1*SQR(x*x+z*z))
'*****************************************


'Input-Eingaben:
'***************

LOCATE  1,15 : PRINT "Eingabe der Funktionsparameter:"
LOCATE  2,15 : PRINT "*******************************"
LOCATE  4,5  : PRINT "x-Intervall:      Start:"
```

```
LOCATE  5,24 : PRINT "Ende:"
LOCATE  6,5  : PRINT "y-Intervall:      Start:"
LOCATE  7,24 : PRINT "Ende:"
LOCATE  8,5  : PRINT "z-Intervall:      Start:"
LOCATE  9,24 : PRINT "Ende:"
LOCATE 11,5  : PRINT "Raum-Skalierung:  x-Faktor:"
LOCATE 12,23 : PRINT "y-Faktor:"
LOCATE 13,23 : PRINT "z-Faktor:"
LOCATE 15,5  : PRINT "Raum-Rotation:    x-Achse:"
LOCATE 16,23 : PRINT "y-Achse:"
LOCATE 17,23 : PRINT "z-Achse:"
LOCATE 19,5  : PRINT "Anz. d. ber. x-Punkte:"
LOCATE 20,5  : PRINT "Netzdichte x-Richtung:"
LOCATE 21,5  : PRINT "           z-Richtung:"
LOCATE 22,5  : PRINT "Umrahmung        (j/n):"
LOCATE 23,5  : PRINT "Koordinatenkreuz (j/n):"

IF flag% = 1 THEN
  GOTO start    'Parameter aus dem Programm
END IF

'Eingaben:
LOCATE  4,35 : INPUT xa
LOCATE  5,35 : INPUT xb
LOCATE  6,35 : INPUT ya
LOCATE  7,35 : INPUT yb
LOCATE  8,35 : INPUT za
LOCATE  9,35 : INPUT zb
LOCATE 11,35 : INPUT sx
LOCATE 12,35 : INPUT sy
LOCATE 13,35 : INPUT sz
LOCATE 15,35 : INPUT rx
LOCATE 16,35 : INPUT ry
LOCATE 17,35 : INPUT rz
LOCATE 19,35 : INPUT rech.gen
LOCATE 20,35 : INPUT netz.x
LOCATE 21,35 : INPUT netz.z
LOCATE 22,35 : INPUT umrahm$
LOCATE 23,35 : INPUT kreuz$

' noch einige Einstellwerte:
tx=0 : ty=0 : tz=0
beo.x = 0 : beo.y = 0 : beo.z = -400
```

```
' Ebenentransformationsparameter:
tex& = wx%/2      'etwa Fenstermittelpunkt
tey& = wy%/2.5
sex& = 1
sey& = 2

GOTO weiter


'Alternativ können Sie hier die verschiedenen
'Parameter auch direkt ins Programm eingeben:
start:
xa = -60          'x-Intervall Start
xb =  60          'x-Intervall Ende
ya = -30          'y-Intervall Start
yb =  60          'y-Intervall Ende
za = -60          'z-Intervall Start
zb =  60          'z-Intervall Ende
sx = 3            'Raum-Skalierung x-Richtung
sy = 3            'Raum-Skalierung y-Richtung
sz = 3            'Raum-Skalierung z-Richtung
tx = 0            'Raum-Translation x-Richtung
ty = 0            'Raum-Translation y-Richtung
tz = 0            'Raum-Translation z-Richtung
rx = 30           'Raum-Rotation um x-Achse
ry = -40          'Raum-Rotation um y-Achse
rz = 0            'Raum-Rotation um z-Achse
beo.x = 0         'Koordinaten Beobachter
beo.y = 0
beo.z = -400
rech.gen = 30     'Anzahl der zu berechnenden Punkte
                  'pro x-Intervall (Zeile)
netz.x = 15       'Netzdichte in Anzahl Linien pro
                  'x-Intervall (Zeile)
                  '(rech.gen muß ein ganzzahliges Viel-
                  ' faches von netz.x sein!)
netz.z = 30       'Netzdichte in Anzahl Linien pro
                  'z-Intervall
umrahm$ = "j"     'Umrahmung ein
kreuz$ = "j"      'Koordinatenkreuz ein

' Ebenentransformationsparameter:
tex& = wx%/1.8    'etwa Fenstermittelpunkt
```

```
tey& = wy%/3
sex& = 1          'Skalierung
sey& = 2


weiter:

PI = 3.141593

' Punkte-Arrays zum Zwischenspeichern von zwei ganzen Reihen
DIM merk.x&(rech.gen-1,1), merk.y&(rech.gen-1,1)

' Pattern-Speicher:
DIM feld1%(3), feld2%(3)

' Bodentiefe:
boden = ya

' Konstanten für die Drehung:
rx = rx*PI/180                  'Grad -> Radiant
ry = ry*PI/180
rz = ry*PI/180

A = COS(ry)*COS(rz)
B = COS(ry)*SIN(rz)
C = -SIN(ry)
D = SIN(rx)*SIN(ry)*COS(rz) - COS(rx)*SIN(rz)
E = SIN(rx)*SIN(ry)*SIN(rz) + COS(rx)*COS(rz)
F = SIN(rx)*COS(ry)
G = COS(rx)*SIN(ry)*COS(rz) + SIN(rx)*SIN(rz)
H = COS(rx)*SIN(ry)*SIN(rz) - SIN(rx)*COS(rz)
I = COS(rx)*COS(ry)

' Schrittweiten errechnen:
IF rech.gen < 2 THEN
  PRINT "Rechengenauigkeit zu klein!" : GOTO Stp
  END IF
ste.z = (zb-za)/(netz.z-1)
ste.x = (xb-xa)/(rech.gen-1)
IF ste.z<0 OR ste.x<0 OR ya>yb THEN
  PRINT "Fehler in Intervallangabe!"  : GOTO Stp
END IF

' Anzahl der Rechenschritte pro senkr. Netzlinie:
anz.netz% = INT(rech.gen/netz.x)
```

```
' Pattern für gesamten Bildschirm definieren:
feld1%(0) = &HCCCC
feld1%(1) = &H3333
feld1%(2) = &HCCCC
feld1%(3) = &H3333

' und für die Funktion:
feld2%(0) = &HFFFF      '&HAAAA
feld2%(1) = &HFFFF      '&H5555
feld2%(2) = &HFFFF      '&HAAAA
feld2%(3) = &HFFFF      '&H5555

PATTERN ,feld1%
LINE (0,0)-(wx%,wy%),1,BF    ' Bildschirm mit Muster füllen

PATTERN ,feld2%              ' Zweites Muster einstellen

POKE WINDOW(8)+27,2          ' Farbe des Area-OUTLINE-Pen
flags = WINDOW(8)+32
POKEW flags,PEEK(flags) OR 8 ' AreaOutline-Flag setzen

r.akt% = 0                   ' Nummer des aktuellen Reihen-Arrays
r.last% = 1                  ' Nummer des letzten Reihen-Arrays


' Zeichenroutine:

' Koordinatenkreuz einzeichnen:
IF kreuz$ = "j" THEN
  IF za<0 AND zb>0 AND xa<0 AND xb>0 AND ya<0 AND yb>0 THEN
    COLOR 4
    x=xa*1.3:y=0:z=0:GOSUB transform:PSET (x1&,y1&)
    x=xb*1.3:y=0:z=0:GOSUB transform:LINE -(x1&,y1&)
    x=0:y=ya*1.3:z=0:GOSUB transform:PSET (x1&,y1&)
    x=0:y=yb*1.3:z=0:GOSUB transform:LINE -(x1&,y1&)
    x=0:y=0:z=za*1.3:GOSUB transform:PSET (x1&,y1&)
    x=0:y=0:z=zb*1.3:GOSUB transform:LINE -(x1&,y1&)
  END IF
END IF

FOR z=zb TO za STEP -ste.z

  zwis% = r.akt%             ' Reihen-Array-Nummern tauschen
  r.akt% = r.last%
  r.last% = zwis%
```

```
n.zaehl% = -1                ' Zähler für Vernetzung
r.zaehl% = 0                 ' Zähler für Reihen-Array
COLOR 3

FOR x=xa TO xb STEP ste.x

  'Funktionswert berechnen:
  y = FNfun(x,z)

  GOSUB transform            ' Koordinaten transformieren
  GOSUB zeichne              ' evt. Feld zeichnen/Koord. merken etc.

NEXT x

IF z<>zb AND umrahm$="j" T

  rette = z                  ' z retten
  COLOR 6

  ' Rechte/linke Seitenwand zeichnen:

  FOR seite% = 1 TO 2

    IF seite% = 1 THEN       ' Linke Seitenwand
      ' Sichtbar?
    IF merk.x&(0,r.last%) >= merk.x&(0,r.akt%) THEN skip
      xvu = xa : zvu = z : yvu = boden
      xhu = xa : zhu = z+ste.z : yhu = boden
      xho = xa : zho = zhu : yho = FNfun(xho,zho)
      xvo = xa : zvo = zvu : yvo = FNfun(xvo,zvo)
    ELSE                     ' Rechte Seitenwand
      ' Sichtbar?
      IF merk.x&(r.zaehl%-1,r.last%) <= merk.x&(r.zaehl%-1,r.akt%)
      THEN skip
      xvu = x-ste.x : zvu = z : yvu = boden
      xhu = xvu : zhu = z+ste.z : yhu = boden
      xho = xvu : zho = zhu : yho = FNfun(xho,zho)
      xvo = xvu : zvo = zvu : yvo = FNfun(xvo,zvo)
    END IF

    ' Punkt vorne unten:
    x = xvu : y = yvu : z = zvu
      GOSUB transform
    IF  y1& >= wy%  THEN  y1& = wy%-1
```

```
      hold.x& = x1& : hold.y& = y1&
      GOSUB makearea
      ' Punkt hinten unten:
      x = xhu : y = yhu : z = zhu
      GOSUB transform
      IF  y1& >= wy%  THEN  y1& = wy%-1
      GOSUB makearea
      ' Punkt hinten oben:
      x = xho : y = yho : z = zho
      GOSUB transform
      GOSUB makearea
      ' Punkt vorne oben:
      x = xvo : y = yvo : z = zvo
      GOSUB transform
      GOSUB makearea

      AREAFILL 0                ' füllen

    skip:
    NEXT seite%
    z = rette
  END IF

NEXT z


' Vordere Wand zeichnen:
IF umrahm$ = "j" THEN

  zwis% = r.akt%              ' Reihen-Array-Nummern tauschen
  r.akt% = r.last%
  r.last% = zwis%
  n.zaehl% = -1               ' Zähler für Vernetzung
  r.zaehl% = 0                ' Zähler für Reihen-Array
  COLOR 5

  FOR x=xa TO xb STEP ste.x

    y = boden

    GOSUB transform          ' Koordinaten transformieren
    IF x+ste.x >= xb THEN x1&=hold.x& : y1&=hold.y&
    IF y1& >= wy% THEN y1& = wy%-1
    GOSUB zeichne            ' evt. Feld zeichnen/Koord. merken etc.
```

```
  NEXT x
END IF

Stp:
WHILE (INKEY$="")          ' Auf Taste warten
WEND

WINDOW CLOSE 4
SCREEN CLOSE 2

END


' Koordinaten im Array merken und
' evtl. Fläche zeichnen:
zeichne:
  'Ins Array speichern:
  merk.x&(r.zaehl%, r.akt%) = x1&
  merk.y&(r.zaehl%, r.akt%) = y1&

  n.zaehl% = n.zaehl% + 1
  ' Ein Netz-Kästchen einzeichnen:
  IF n.zaehl%=anz.netz% AND z<>zb AND y>=ya AND y<=yb THEN
    n.zaehl% = 0           ' Zähler rücksetzen

    ' Area-Polynom initialisieren:
    FOR ind = r.zaehl%-anz.netz% TO r.zaehl%
      x1& = merk.x&(ind,r.akt%)
      y1& = merk.y&(ind,r.akt%)
      GOSUB makearea
    NEXT ind
    FOR ind = r.zaehl% TO r.zaehl%-anz.netz% STEP -1
      x1& = merk.x&(ind,r.last%)
      y1& = merk.y&(ind,r.last%)
      GOSUB makearea
    NEXT ind

    AREAFILL 0             ' Polynom ausfüllen

  END IF
  r.zaehl% = r.zaehl% + 1  ' nächster Punkt
RETURN
```

```
' Transformationsroutine:
' transformiert 3-D-Koord. in x, y, z
' nach 2-D-Koord. in x1&, y1&

transform:
  'Transformationen durchführen:
  xt1 = sx*x + tx          ' Skalierung und Translation
  yt1 = sy*y + ty
  zt1 = sz*z + tz

  xt2 = xt1*A + yt1*B + zt1*C ' Rotationen
  yt2 = xt1*D + yt1*E + zt1*F
  zt2 = xt1*G + yt1*H + zt1*I

  'Zentralprojektion:
  zwis = zt2 - beo.z
  x1& = beo.x - beo.z * (xt2-beo.x)/zwis
  y1& = beo.y - beo.z * (yt2-beo.y)/zwis

  'Ebenentranslation:
  x1& = tex& + x1&/sex&
  y1& = tey& - y1&/sey&
RETURN


'Mini-Clipping und Area-Aufruf mit x1&, y1&:
  makearea:
  ' Mini-Clipping:
  IF x1&>=0 AND x1&<wx% AND y1&>=0 AND y1&<wy% THEN
    AREA(x1&,y1&)
  END IF
RETURN
```

Sagenhaft! werden Sie sagen, und mit Recht, doch gehen wir gleich in medias res. Dieses Programm zeichnet in der Form, wie Sie es hier sehen, eine ganz bestimmte Funktion in einer ganz bestimmten Art und Weise. Das liegt u.a. an der Variablen flag%. Hier wurde sie auf 1 gesetzt. Dieses Flag gibt nämlich an, ob Sie die vielen veränderbaren Parameter in dem Programm per INPUT nach jedem Starten des Programms neu eingeben wollen (flag% = 0), oder ob die Parameter verwendet werden sollen, die direkt im Programm stehen (flag% = 1). Das Flag wurde eingeführt, damit Sie sich beim Testen Ihrer eigenen Funktion mit Ihren eigenen Parametern einige Arbeit sparen können. Es wäre schließlich ziemlich umständlich, bei jedem Testlauf noch einmal sämtliche Parameter neu eingeben zu müssen, obwohl Sie vielleicht nur einen einzigen ändern möchten.

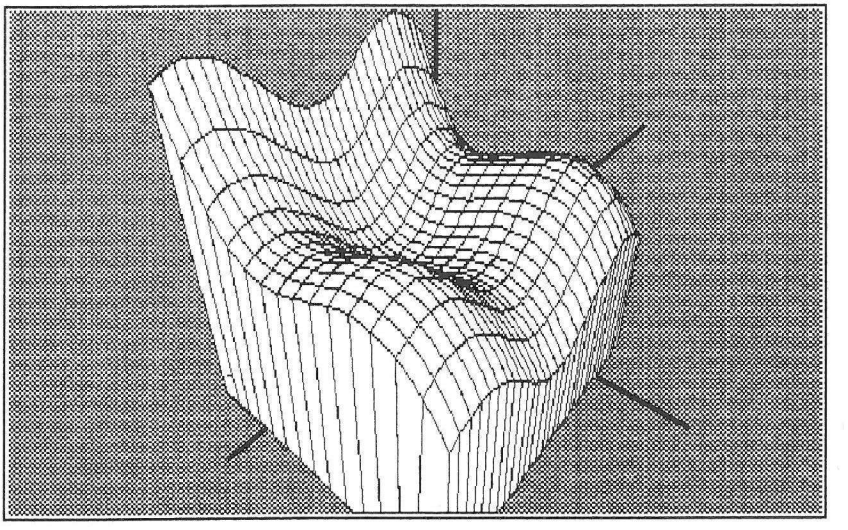


Bild 5.5: 3-D-Funktionsplotter 2

Ganz zu Anfang steht per DEF-Befehl die Definition der 3-D-Funktion, die Sie zeichnen möchten. Beachten Sie, daß sich nicht unbedingt jede Funktion dazu eignet, auf dem Bildschirm dargestellt zu werden. Bei Funktionen mit vielen Definitionslücken können eine Reihe von Problemen auftauchen. Wird z.B. im Laufe der Berechnungen der Nenner einmal gleich Null, so taucht natürlich die Fehlermeldung DIVISION BY ZERO auf. Negative Wurzeln oder zu große Werte bringen das Programm ebenfalls zum Stehen. Wählen Sie also die Werteintervalle entsprechend.

Möchten Sie nun aber unbedingt solche Funktionen ebenfalls an den undefinierten Stellen darstellen, so gibt es einen Trick: Entweder, Sie installieren eine entsprechende ON ERROR-Routine (Vorsicht! Die Funktion wird an mehreren Stellen im Programm berechnet) oder Sie nutzen die folgende Eigenschaft von Amiga-Basic aus: Geben Sie im Direktmodus ein:

```
PRINT 4=4;"/ ";4=2
```

Die Ausgabe: –1 / 0

Sie haben damit das Ergebnis eines logischen Vergleiches ausgegeben. Jeder logischen Operation weist Amiga-Basic nämlich einen Wert zu, mit dem man auch rechnen kann. Ist ein Ausdruck logisch wahr, erhält er den Wert –1, ist er falsch, ist das Ergebnis 0. Solche Vergleiche (auch mit »>«, »<«, »<>« etc.) dürfen Sie dann selbstverständlich in Ihre Funktionen mit einbauen.

Die Funktion f(x)=sin(x)/x beispielsweise ist an der Stelle x=0 nicht definiert. Damit das Programm diese Stelle jedoch ebenfalls errechnen kann, schreiben Sie:

```
f(x) = sin(x)/(x-(x=0))
```

Sobald x gleich Null wird, bekommt der Ausdruck (x=0) den Wert –1, der von x (also von 0) subtrahiert wird. Das Ergebnis für x=0: f(0)=sin(0)/(0–(–1)). Der Nenner wird gleich 1. Der resultierende Punkt ist zwar eigentlich kein gültiger Punkt der Kurve, das fällt aber bei geschicktem Einsatz nicht sonderlich auf.

Bei Wurzeln sollten Sie statt dessen vielleicht mit der Funktion ABS( ) arbeiten: f(x)=SQR(ABS(x)) liefert niemals ungültige Werte.

Kommen wir zu den vielen Parametern, die Sie ändern dürfen:

xa	x-Raumkoordinate des x-Intervallstartes
xb	x-Raumkoordiante des x-Intervallendes
ya	y-Raumkoordinate des y-Intervallstartes
yb	y-Raumkoordiante des y-Intervallendes
za	z-Raumkoordinate des z-Intervallstartes
zb	z-Raumkoordiante des z-Intervallendes
sx	Raum-Skalierung der Funktion in x-Richtung
sy	Raum-Skalierung der Funktion in y-Richtung
sz	Raum-Skalierung der Funktion in z-Richtung
tx	Raum-Translation in x-Richtung
ty	Raum-Translation in y-Richtung
tz	Raum-Translation in z-Richtung
rx	Raum-Rotation um die x-Achse
ry	Raum-Rotation um die y-Achse
rz	Raum-Rotation um die z-Achse
beo.x	x-Koordinate des Beobachters
beo.y	y-Koordinate des Beobachters
beo.z	z-Koordinate des Beobachters
rech.gen	Anzahl der zu berechnenden Punkte pro x-Intervall (Zeile)
netz.x	x-Netzdichte (Anzahl der Netzlinien pro x-Intervall; rech.gen muß stets ein ganzzahliges Vielfaches von netz.x sein!)
netz.z	z-Netzdichte (Anzahl der Netzlinien pro z-Intervall; gleichzeitig Anzahl der zu berechnenden Punkte pro z-Intervall)
umrahm$	Flag für Umrahmung ein/aus
kreuz$	Flag für Koordinatensystem ein/aus
tex&	Ebenentranslation in x-Richtung (x-Position der Grafik im Fenster)
tey&	Ebenentranslation in y-Richtung (y-Position der Grafik im Fenster)
sex&	Ebenenskalierung in x-Richtung (x-Koord. mal 1/sex&)
sey&	Ebenenskalierung in y-Richtung (y-Koord. mal 1/sey&)

Aus Platzgründen können Sie nicht alle davon auch per INPUT eingeben. Sie sehen, da können Sie schon einiges ändern. Am besten, Sie probieren einfach ein wenig herum. Auf diese Weise werden Sie mit den Auswirkungen jedes einzelnen Parameters am besten vertraut. Im übrigen sollten Ihnen einige bereits bekannt vorkommen (z.B. die Transformationsparameter). Wichtig ist, daß Sie die Intervall-Parameter so eingeben, daß gilt: ya < yb, xa < xb und za < zb. Ein Tip: yb sollten Sie möglichst groß wählen. Vermeiden Sie Winkel größer 45 oder kleiner –45 Grad (lieber Werte nahe bei Null wählen).

Zu Programmanfang öffnet die Routine erst einmal einen ganz neuen Bildschirm mit einem inliegenden Fenster. Der Screen wird mit drei Farbebenen eingerichtet. Es sind also maximal acht verschiedene Farben gleichzeitig verfügbar. In unserem Programm werden wir allerdings nur sieben benötigen, die gleich im Anschluß mit **PALETTE** definiert sind. Dann geht es los mit der Eingabe der Parameter.

Das eigentliche Programm beginnt allerdings erst hinter dem Sprunglabel »weiter:«. Hier werden einige wichtige Dinge initialisiert. So z.B. zwei Arrays, die uns über das gesamte Programm begleiten werden (**merk.x&(,)** und **merk.y&(,)**). Sie enthalten die gesamten Koordinaten von insgesamt zwei zu berechnenden z-Reihen der Funktion (s.u). Ferner errechnet das Programm die STEP-Werte für die weiter unten stehenden FOR...NEXT-Schleifen (ste.x und ste.z) aus den x- bzw. z-Intervallen und der angegebenen Rechengenauigkeit. Die Rechengenauigkeit rech.gen gibt an, wieviele Einzelpunkte in jeder y-z-Ebene berechnet werden sollen, die dann mit Linien verbunden werden, um einen zusammenhängenden Kurvenverlauf zu erzeugen. Die Variable netz.z enthält die gleiche Information für die x-y-Ebenen. netz.x dagegen gibt an, wieviele Netzlinien pro x-Intervall die waagerechten Netzlinien kreuzen sollen. **anz.netz%** bestimmt dann die Anzahl der zu berechnenden Punkte zwischen zwei Netzlinien. Aber das sollten Sie am besten selbst ausprobieren.

Als nächstes folgt das Koordinatenkreuz: Sechs Punkte werden berechnet, transformiert und projiziert (durch das Unterprogramm »transform«), drei Linien werden gezeichnet.

Dann erkennen Sie die beiden ineinanderverschachtelten FOR...NEXT-Schleifen, die Sie bereits in den Programmen der vorherigen Kapitel finden. Hier sind sie natürlich ein wenig umfangreicher. Das Prinzip aber bleibt.

Spätestens jetzt sollten wir uns einmal damit auseinandersetzen, wie in diesem Programm das Problem der verdeckten Linien gelöst wird. Das geschieht hier nämlich auf eine etwas andere Art und Weise, als wir das bisher kennengelernt haben.

Das Programm zeichnet nämlich nicht jeden errechneten Punkt der Funktion sofort auf den Bildschirm. Vielmehr werden die x- und y-Werte jedes Punktes einer x-Reihe in den oben erwähnten Arrays **merk.x&(n,r)** und **merk.y&(n,r)** abgelegt. n gibt dabei die Position des Punktes innerhalb einer Reihe an. r ist entweder 0 oder 1. Es werden nämlich stets zwei hintereinanderliegende x-Reihen in den Arrays gespeichert. Das hat einen bestimmten Grund. Es werden nicht einfache Linien von jedem Punkt nach unten gezogen. Vielmehr werden die Punkte in x- und auch in z-Richtung durch Linien miteinander verbunden (Vernetzung). Die Fläche, die von diesen Linien eingegrenzt wird (Netzflächen) füllt das Programm mit einer Farbe. Dadurch übermalt es alles, was jemals dahinter gezeichnet wurde. Das Bild wird also quasi Mosaikstein für Mosaikstein zusammengesetzt. Probleme kann das nur bei starken Drehungen rx, ry und rz in manchen Funktionen geben.

Nachdem also ein Funktionswert in der x-Schleife berechnet wurde, wird dieser Punkt in dem Unterprogramm »transform« auf altbekannte Weise transformiert und projiziert. Die fertigen Ebenenkoordinaten stehen dann in xl& und yl& und werden dem Unterprogramm »zeichne« übergeben, das sie in die beiden besagten Arrays ablegt. Jetzt entscheidet es sich, ob tatsächlich

gezeichnet werden soll: Beim ersten z-Schleifendurchlauf (z=zb) ist noch kein Array voll. Generell erst beim zweiten z-Durchlauf kann also gezeichnet werden. Sollen pro senkrechter Netzlinie noch mehrere Punkte errechnet werden, wird das Zeichnen ebenfalls so lange übersprungen, bis eine Netzlinie erreicht ist.

Jetzt ist es soweit: Eine Fläche soll eingezeichnet werden. Dazu verwenden wir den Befehl **AREA** bzw. **AREAFILL**, die ein beliebiges Polygon ausfüllen. Das Programm übergibt jeden Eckpunkt der Fläche – gewonnen aus den Arrays merk.x&(,) und merk.y(,) – dem Befehl AREA. Das sind mindestens vier Punkte, können aber je nach Rechengenauigkeit und Netzlinienzahl auch sechs, acht, zehn etc. sein.

Ist eine Linie abgearbeitet, so kann – je nach Flag umrahm$ und je nach Perspektive – noch eine rechte oder linke Seitenwand (ebenfalls mit AREAFILL) gezeichnet werden. Danach werden die Punkte der vorletzten Reihe nicht mehr benötigt. Sie werden nun durch die Punkte der nächsten Reihe überschrieben, bis das Ende der z-Schleife erreicht ist.

Haben Sie umrahm$="j" gewählt, gehört zu den Seitenwänden natürlich noch eine vordere Wand. Sie zeichnet das Programm auf die gleiche Weise wie beschrieben. Das Bild ist fertig, das Programm erwartet Ihren Tastendruck, schließt dann Screen und Fenster und kehrt zurück in Ihre Workbench.

An dem Programm werden Sie sicherlich Ihren Spaß haben. Testen Sie doch einmal andere Funktionen mit anderen Parametern. Ein Tip: Verschaffen Sie sich erst einmal durch großzügige Intervallangaben einen Überblick über Ihre Funktion, und grenzen Sie dann den interessanten Bereich ein. Wählen Sie zunächst kleine Drehwinkel, kleine Skalierungen und einen sehr entfernten Beobachter (beo.z=–4000). Schalten Sie die Umrahmung zunächst aus. Haben Sie die interessanten Teile gefunden, dann setzen Sie für weitere Verfeinerungen (aus Geschwindigkeitsgründen) die Rechengenauigkeit zunächst auf kleine Werte.

5.3 Verdeckte Linien und Flächen in objektorganisierten Welten

Was sich hinter diesem hochtrabenden Titel verbirgt, ist gar nicht so kompliziert, wie es sich anhört. In den letzten Kapiteln wurden Ihnen Möglichkeiten und Techniken demonstriert, mit denen Sie auf einfachste Weise und ohne viele komplizierte Berechnungen in dreidimensionalen Funktionsgraphen verdeckte Linien und Flächen realisieren können. Die gleiche Intention haben Sie allerdings sicherlich auch bei einfachen Objekten wie dem Haus aus dem letzten Kapitel etc. und genau das werden Sie hier finden.

Nun ist das Problem der sogenannten »Hidden Lines and Surfaces« eines der schwierigsten in der Computergrafik überhaupt. Es existieren eine ganze Reihe verschiedener Algorithmen, die entweder schnell sind oder möglichst viele Fälle von Objekt- und Flächenzusammenstellungen berücksichtigen. Beides gleichzeitig ist wohl kaum zu realisieren. Je mehr Fälle zu berücksichtigen sind, um so komplizierter und dementsprechend zeitaufwendiger werden die Routinen zur Flächenunterdrückung.

Da Sie später ein Verfahren kennenlernen werden, das hundertprozentig alle Eventualitäten erfaßt (das **Ray-Tracing**), aber auch entsprechend langsam ist, soll an dieser Stelle mehr Wert auf Geschwindigkeit gelegt werden. Wir schränken uns damit zwar in der Wahl der Objekte und Flächen ein, erhalten dafür aber eine Technik, die sogar noch unter Real-Time recht brauchbare Ergebnisse erzielt.

Unser hier vorgestelltes Verfahren unterteilt sich in zwei Schritte:

- Flächenrückenunterdrückung
- Tiefensortierung

Der erste Schritt bezieht sich stets auf Flächen innerhalb eines Objektes. Der zweite bearbeitet Flächen, die entweder ebenfalls in einem Objekt liegen können, die aber auch in unterschiedlichen Objekten vorkommen können. Damit versuchen wir also zunächst, so viele Flächen innerhalb jedes Objektes zu verdecken wie möglich. Im zweiten Schritt werden dann noch nicht erfaßte verdeckte Flächen innerhalb eines Objektes berücksichtigt und die Verdeckung von verschiedenen Objekten untereinander festgestellt.

In jedem Fall jedoch müssen Sie unser Modell von Objekten, Linien und Punkten erweitern. Jede einzelne Fläche eines Objektes muß erfaßt werden. Eine Fläche wird definiert durch die sie begrenzenden Linien, die ihrerseits durch Start- und Endpunkte festgelegt sind. Ein Objekt wiederum kann dann aus verschiedenen Flächen bestehen.

a) Die Flächenrückenunterdrückung

Nehmen wir ein sehr einfaches Beispiel eines Körpers, einen Quader (s. Bild 5.6). Dieser Quader besitzt sechs Flächen, von denen stets nur drei tatsächlich sichtbar sind. Wichtig zu bemerken ist, daß eine Fläche in diesem Fall entweder vollständig oder gar nicht sichtbar ist, denn nur solche Flächen findet der Algorithmus. Welche drei Flächen das sind, das hängt von der jeweiligen Perspektive ab. Wie aber ist es möglich, herauszufinden, welche Flächen sichtbar sind und welche nicht?

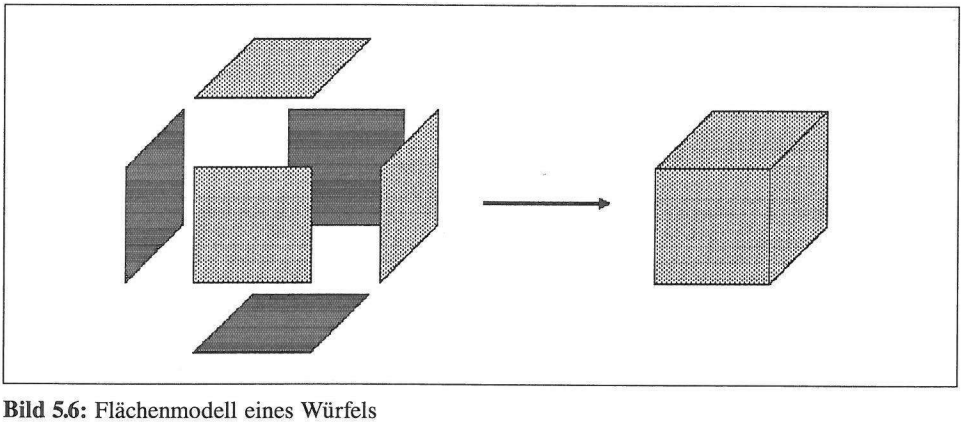


Bild 5.6: Flächenmodell eines Würfels

Das ist eigentlich gar nicht so schwer. Wir brauchen lediglich darauf zu achten, daß die Eckpunkte (oder die Grenzlinien) einer Fläche im Uhrzeigersinn numeriert sind. Schauen Sie sich also jede einzelne Fläche von vorne an und numerieren die Seiten entsprechend (s. Bild). Sollte später eine Seite vom Beobachter abgewandt sein, dann erschiene die Fläche in entgegengesetzter Numerierung auf dem Bildschirm. Genau das ist dann das Kriterium dafür, daß die Fläche unsichtbar ist.

Jede Fläche besitzt also quasi eine sichtbare und eine unsichtbare Seite (fast so wie ein halbdurchlässiger Spiegel). Wir stellen einfach fest, welche dieser beiden Seiten uns zu- und welche uns abgewandt liegt.

Beschreiten wir den mathematischen Weg! Wir suchen also die Richtung, in die die sichtbare Seite einer Fläche zeigt. Ein auf der Fläche senkrechter Vektor zeigt logischerweise genau in diese Richtung (s. Bild 5.7).

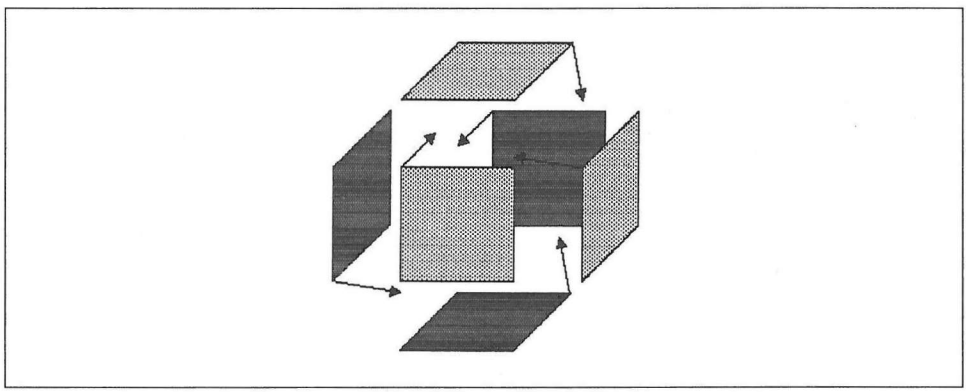


Bild 5.7: Vektorprodukt einer Fläche

Mathematisch ist dieser Vektor recht einfach zu berechnen, indem von zwei linear unabhängigen Vektoren der Fläche das Vektorprodukt gebildet wird (lineare Unabhängigkeit und Vektorprodukt haben Sie bereits kennengelernt, s. auch Anhang). Die linear unabhängigen Vektoren der Fläche sind sofort gefunden. Suchen Sie sich einfach irgendeinen Eckpunkt $P_1(p_{1x},p_{1y},p_{1z})$ der Fläche aus und bilden den Vektor zum nächstbesten Eckpunkt $P_2(p_{2x},p_{2y},p_{2z})$ – Vektor Nummer eins. Vektor Nummer zwei erhalten Sie, indem Sie einen zweiten Vektor vom gleichen Ursprungspunkt P_1 zum übernächsten Eckpunkt $P_3(p_{3x},p_{3y},p_{3z})$ ziehen. Das funktioniert immer, falls die drei Eckpunkte nicht auf einer Linie liegen (dann wären die beiden Vektoren nicht mehr linear unabhängig).

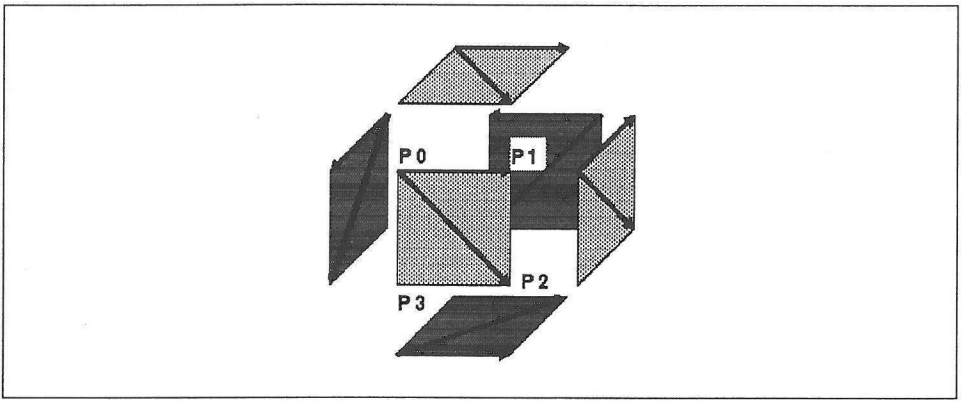


Bild 5.8: Ermittlung linear unabhängiger Vektoren einer Fläche

Nennen wir die beiden Vektoren $\vec{v}$ und $\vec{w}$ mit:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} p_{2x}-p_{1x} \\ p_{2y}-p_{1y} \\ p_{2z}-p_{1z} \end{pmatrix}$$

$$\vec{w} = \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} p_{3x}-p_{1x} \\ p_{3y}-p_{1y} \\ p_{3z}-p_{1z} \end{pmatrix}$$

Um nun den dazu senkrechten Vektor $\vec{p}$ zu berechnen, ist das Vektorprodukt »$\vec{v}$ kreuz $\vec{w}$« (nicht zu verwechseln mit dem Skalarprodukt) zu ermitteln:

$$\vec{p} = \vec{v} \times \vec{w} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \times \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix}$$

$$= \begin{pmatrix} v_y * w_z - v_z * w_y \\ v_z * w_x - v_x * w_z \\ v_x * w_y - v_y * w_x \end{pmatrix}$$

Falls dieser Richtungsvektor $\vec{p}$ von uns (als Betrachter) wegzeigt, dann ist die Fläche sichtbar, zeigt er auf uns, dann schauen wir auf die unsichtbare Seite. Um nun auch mathematisch zu ermitteln, in welche Richtung der Vektor zeigt, benötigen wir einen weiteren Vektor und das sogenannte Skalarprodukt beider Vektoren.

Bei dem zweiten Vektor, den wir $\vec{s}$ nennen wollen, handelt es sich um den Sichtvektor, der vom Beobachter zum jeweiligen zu projizierenden Punkt zeigen soll (s. Bild 5.9).

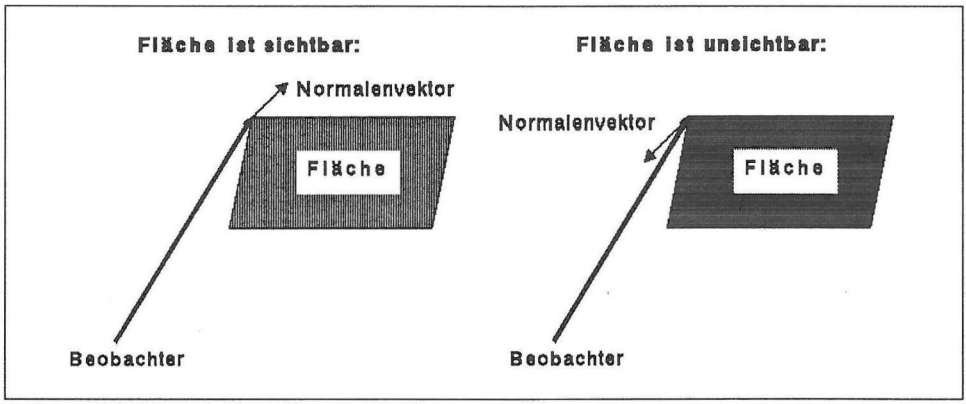


Bild 5.9: Sichtbarkeitsuntersuchung mittels Sicht- und Flächenrichtungsvektor

Für $\vec{s}$ gilt:

$$\vec{s} = \begin{pmatrix} s_x \\ s_y \\ s_z \end{pmatrix} = \begin{pmatrix} p_{1x}-beo_x \\ p_{1y}-beo_y \\ p_{1z}-beo_z \end{pmatrix}$$

mit:

beo_x, beo_y, beo_z: Koordinaten des Beobachters
p_{1x}, p_{1y}, p_{1z}: Koordinaten des Fußpunktes P_1 von $\vec{v}$ und $\vec{w}$

Zeigen also Sichtvektor $\vec{s}$ und Flächenrichtungsvektor $\vec{p}$ in etwa die gleiche Richtung, dann ist die Fläche sichtbar, andernfalls nicht. Um ein mathematisches Kriterium für diese Sichtbarkeit in der Hand zu haben, untersucht der Algorithmus den Winkel a zwischen den beiden Vektoren. Gilt für a: $-90 < a < 90$, dann zeigen beide Vektoren etwa in die gleiche Richtung, die Fläche ist sichtbar. Gilt dagegen: $a >= 90$ oder $a <= -90$, dann tritt der andere Fall in Kraft. Den Winkel zwischen zwei Vektoren berechnet man aber leicht durch das sogenannte Skalarprodukt q, das bekanntlich eine normale Zahl ergibt (keinen Vektor wie das Vektorprodukt):

$$\begin{aligned} q = \vec{s}*\vec{p} &= |\vec{s}|*|\vec{p}|*\cos(a) \\ &= s_x*p_x + s_y*p_y + s_z*p_z \end{aligned}$$

Ist a im Bereich –90 Grad bis +90 Grad, dann ist der Cosinus cos(a) positiv, im anderen Fall wird er gleich Null ($a = 90$ oder $a = -90$) oder negativ (s. Bild 5.10).

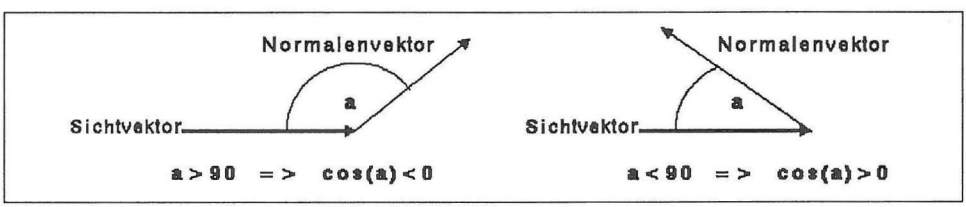


Bild 5.10: Sichtbar oder nicht sichtbar – eine Frage des Winkels

Da die beiden Faktoren $|\vec{s}|$ und $|\vec{p}|$ (Absolut) immer positiv sind, interessiert eigentlich nur das Vorzeichen des gesamten Skalarproduktes q. Wir brauchen also gar nicht umständlich mit Winkeln und Winkelfunktionen herumzurechnen, sondern benutzen lediglich die Parametergleichung des Skalarproduktes $q = sx * px + sy * py + sz * pz$, was natürlich viel einfacher zu berechnen ist. Wir unterscheiden also folgende zwei Fälle:

Fall 1: $q > 0$ => Die Fläche ist sichtbar
Fall 2: $q \leqq 0$ => Die Fläche ist unsichtbar

Der Algorithmus ist wie folgt zusammenzufassen:

1. Jedes Objekt hat eine Flächenliste mit im Uhrzeigersinn durchnumerierten Kanten
2. Jede Fläche wird der Sichtbarkeitsüberprüfung unterzogen
3. Ermittlung zweier linear unabhängiger Vektoren $\vec{v}$ und $\vec{w}$ der zu untersuchenden Fläche durch die Auswahl dreier aufeinanderfolgender Punkte P_1, P_2, P_3
4. Berechnung des senkrechten Vektors $\vec{p}$ durch das Vektorprodukt $\vec{p} = \vec{v} \times \vec{w}$
5. Berechnung des Sichtvektors $\vec{s}$ aus der Verbindung Beobachter -> P_1
6. Ermittlung der Richtung der Vektoren $\vec{s}$ und $\vec{p}$ zueinander durch Bestimmung des Vorzeichens des Skalarproduktes q der beiden Vektoren
7. Falls $q > 0$: Fläche als sichtbar kennzeichnen
   Falls $q \leqq 0$: Fläche als unsichtbar kennzeichnen

Dieser Algorithmus versagt bei Flächen, die zwar dem Beobachter zugewandt sind, von anderen Flächen desselben Objektes oder anderer Objekte allerdings verdeckt werden. Soll allein dieser Algorithmus Verwendung finden, dann müssen Sie sich auf ein Objekt

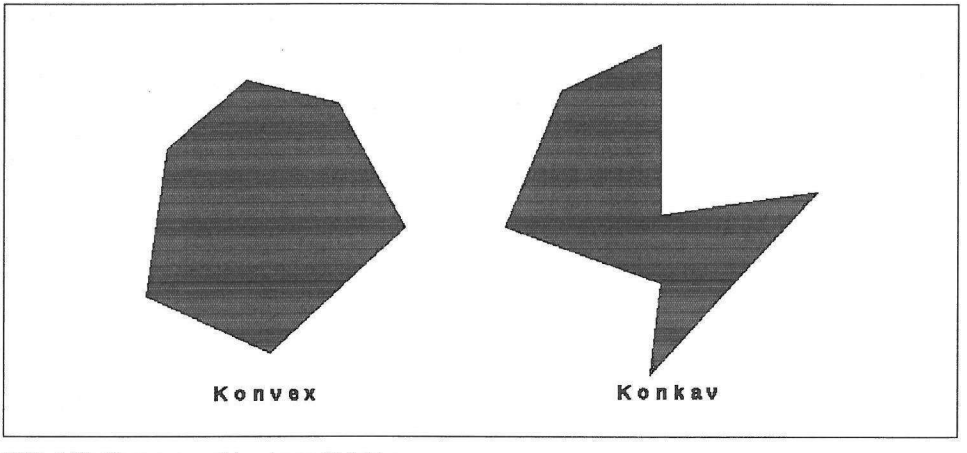


Bild 5.11: Konvexe und konkave Objekte

beschränken (oder die Objekte so anordnen, daß sie sich in keinem Fall überschneiden). Ferner muß das Objekt konvex sein (alle Innenwinkel sind kleiner 180 Grad). Ein konkaves Objekt hätte Einbuchtungen, die von anderen Flächen überdeckt sein könnten.

b) Tiefensortierung

Um diesen Mißstand zu beseitigen, behelfen wir uns einer Technik, die auch als Painters Algorithm (Algorithmus der Maler) weithin bekannt ist. In der Malerei nämlich wird diese Technik oft verwandt, um das Problem der verdeckten Flächen, das ja auch dort existiert, zu lösen. Dabei beginnt der Maler mit dem weit entfernten Hintergrund. Dann erst malt er die etwas näher liegenden Objekte ein. Dabei werden zwangsläufig Teile des Hintergrundes wieder überpinselt, bis er endlich zu den ganz nahen Figuren etc. kommt, die wiederum alles Dahinterliegende überdecken können.

In der Computertechnik geht das ganz analog: Erst werden die hinteren Objekte und Flächen gezeichnet, dann die vorderen, die die hinteren teilweise oder ganz wieder verdecken können. Die Technik eignet sich natürlich nur für Videos, wie ihn unser Amiga und die meisten Computer besitzen. Für Plotter beispielsweise, bei denen ja nicht einfach etwas überpinselt werden kann, werden sogenannte Hidden-Line-Algorithmen verwendet, die erst alle verdeckten Linien und Linienteile eliminieren und dann zeichnen.

Eigentlich ist das alles nichts Neues, denn genau das haben Sie ja bereits bei 3-D-Funktionen kennengelernt. Dort allerdings war es ein gehöriges Stück einfacher, herauszufinden, welche Fläche hinten und welche vorne lag. Wir fingen einfach bei hohen z-Koordinaten an zu zeichnen.

Bei normalen Flächen im Objektmodell allerdings müssen wir erst umständlich ermitteln, welche Flächen oder Flächenteile vor anderen Flächen liegen. Dabei kann es sogar vorkommen, daß ein Teil einer Fläche vor und ein Teil hinter einer anderen Fläche zu liegen kommt oder daß sich zwei Flächen durchdringen (s. Bild 5.12).

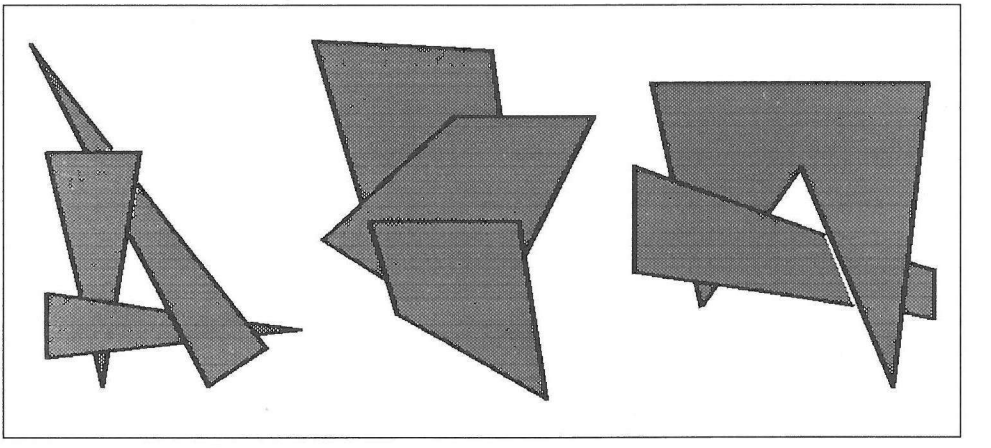


Bild 5.12: Zyklische Überlappung, Durchdringung und wechselnde Überlappung zweier Polygone

Ein Tiefensortieralgorithmus kann beliebig kompliziert werden, je mehr Fälle er zu berücksichtigen hat. Dabei muß er eventuell sogar einzelne Flächen teilen, um eindeutige Prioritäten herzustellen etc.

Verwendet werden bei dieser Technik selbstverständlich nur noch diejenigen Flächen, die der oben beschriebene Algorithmus als sichtbare Flächen übriggelassen hat, denn die dort als unsichtbar ermittelten Flächen sind ja in jedem Fall unsichtbar. Damit verringert sich die Anzahl der zu bearbeitenden Flächen natürlich erheblich (oft um die Hälfte).

Beim Tiefensortieralgorithmus wird für jede Fläche geprüft, ob es eine oder mehrere weitere Flächen gibt, die sie überlappen könnten. Dabei wird der Programmierer bemüht sein, erst mit einfachen Mitteln so viele Flächen wie möglich auszuschließen, damit er die komplizierteren Methoden möglichst spät an möglichst wenigen Flächen praktizieren muß.

Allgemein wird versucht, eine Prioritätenliste von Flächen zu erstellen, die angibt, in welcher Reihenfolge sie zu zeichnen sind. Hierzu existiert eine recht anschauliche Technik, die die folgenden Schritte vollzieht. Vorausgesetzt ist dabei stets, daß der Beobachter entlang der z-Achse schaut. Anderenfalls muß das durch entsprechende Translationen und Rotationen zunächst erreicht werden (ähnlich der Rotation um eine beliebige Raumachse):

1. Von allen Flächen wird der Eckpunkt ermittelt, der die höchste z-Koordinate z_{max} (am weitesten entfernt) und derjenige, der die niedrigste z-Koordinate z_{min} besitzt (am nächsten am Beobachter).
2. Alle Flächen werden zunächst nach fallendem z_{max} vorsortiert. An erster Stelle steht dann die Fläche mit dem höchsten z_{max}. Wir nennen sie $F_1, F_2, \ldots F_n$
3. Jetzt muß die erste Fläche F_1 mit allen anderen Flächen F_i (also: F_2 bis F_n) verglichen werden:

   a) Falls der nächstliegende Punkt von F_1 (F_{1zmin}) weiter entfernt liegt, als der am weitesten entfernte Punkt der zweiten Fläche F_2 (F_{2zmax}), also:

   $$F_{1zmin} \geqq F_{2zmax}$$

   dann kann kein Teil von F_1 einen Teil von F_2 oder einer anderen Fläche F_i der Liste überdecken. F_1 kann also an der ersten Stelle der Liste bleiben.

   b) Falls jedoch gilt:

   $$F_{1zmin} < F_{2zmax}$$

   dann ist es möglich, daß F_1 nicht nur F_2, sondern sogar alle anderen Flächen F_i verdeckt. Um das herauszufinden, werden eine Reihe von Tests durchgeführt. Dabei beginnt man mit den einfachsten und schnellsten Tests. Es muß dabei F_1 mit jeder anderen Fläche F_i getestet werden:

   - Untersuchung der einschließenden Rechtecke zweier Flächen F_1 und F_i: Projiziert man die beiden Flächen ganz normal auf die Ebene, dann kann man ein einschließendes Rechteck für jede Fläche finden, indem man einfach die niedrigsten und die höchsten x- bzw. y-Koordinaten der Eckpunkte ermittelt (s. Bild 5.13). Überlappen sich die einschließenden Rechtecke zweier Flächen nicht, dann können sich auch die Flächen selbst nicht überlappen. Dann geht es weiter mit der nächsten Fläche in der Liste. Überlappen sie sich doch, dann kommt der nächste Test:

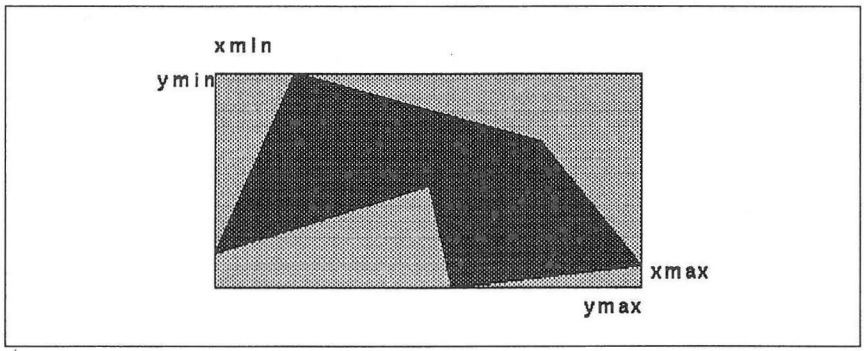


Bild 5.13: Einschließendes Rechteck bei Polygonen

– Liegen alle Eckpunkte der Fläche F_l hinter der von der zweiten Fläche F_i aufgespannten Ebene? Dazu verwenden wir wieder den bereits im obigen Algorithmus berechneten senkrechten Vektor $\vec{p}$ auf die jeweilige Ebene F_i. Dieser Vektor zeigt ja inzwischen bei allen übriggebliebenen Flächen in die gleiche Richtung wie der Beobachtervektor (s.o.). Er zeigt also in die Ferne (s. Bild 5.14). Ein Vektor von einem beliebigen Eckpunkt der Fläche F_i zu dem zu untersuchenden Eckpunkt von F_l (nennen wir ihn $\vec{f}$) zeigt genau dann in die gleiche Richtung wie $\vec{p}$, wenn der Eckpunkt von F_l hinter der Fläche F_i liegt. Liegt er vor ihr, dann zeigt er in entgegengesetzte Richtung. Liegt $\vec{f}$ dagegen senkrecht auf $\vec{p}$, dann liegt der Eckpunkt von F_l auf der Ebene (nicht unbedingt Fläche!), die von F_i aufgespannt wird.

  Die Richtungsbeziehung ermitteln wir wie beim obigen Flächentest mit dem Skalarprodukt $\vec{f}*\vec{p}$, von dem nur das Vorzeichen interessiert.

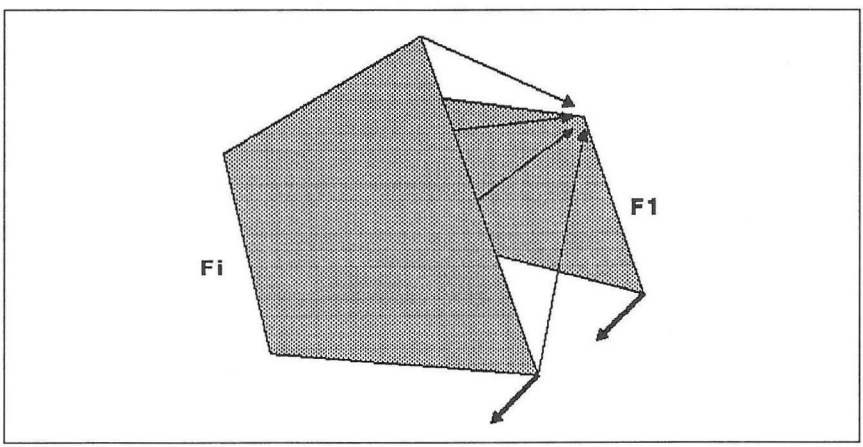


Bild 5.14: Test zur Polygonüberlappung

Alle Eckpunkte von F_1 werden auf diese Weise getestet. Liegen sie alle hinter der jeweiligen Fläche F_i, dann geht es weiter mit der nächsten Fläche F_{i+1}.

Ist das jedoch nicht der Fall, dann wird der nächste Test durchgeführt:

- Im Prinzip ist das der gleiche Test wie der vorige. Jetzt wird lediglich getestet, ob alle Eckpunkte der Fläche F_i vor der von F_1 aufgespannten Ebene liegen.

  Ist das der Fall, dann kann F_1 weiter an erster Position in der Tabelle bleiben und die nächste Fläche F_{i+1} wird überprüft.

- Der letzte Test, der sehr aufwendig werden kann, erfragt, ob F_1 von einer Fläche F_i überlappt wird. Hier müssen Schnittpunkte errechnet werden etc.

Wie gesagt, wird jede Fläche F_i bezüglich F_1 diesen Tests unterzogen. Übersteht F_1 alle Prüfungen, dann bleibt sie endgültig an erster Stelle in der Liste (oder wird direkt gezeichnet). Die Rolle von F_1 übernimmt dann die nächste Fläche F_2 usw.

Sollte F_1 jedoch einen einzigen Test nicht überstehen, dann wird es mit derjenigen Fläche getauscht, bei der der Test nicht funktionierte. Mit dieser neuen obersten Fläche wird dann die gesamte Prozedur von Anfang an wiederholt.

Gleichzeitig muß die ehemalige Position dieser Fläche gespeichert werden. Sollte es nämlich dazu kommen, daß derselbe Tausch noch einmal vorgenommen wird, dann liegt z.B. eine zyklische Überlappung vor, die nur durch Flächenteilung gelöst werden kann.

Reichlich kompliziert, werden Sie sagen. Da muß ich Ihnen leider zustimmen. Es handelt sich hierbei aber noch um einen relativ einfachen Algorithmus.

Wollten wir alle Schritte dieser Technik programmieren, dann wäre Schluß mit unserem Bestreben, Real-Time-Bewegungen vorzunehmen. Lassen wir also den ganzen Perfektionismus beiseite und begnügen uns mit einem verlockend einfachen Kriterium der Tiefensortierung, das jedoch eine ganze Menge Fälle berücksichtigt:

Ermittelt wird einfach ein mittlerer z-Wert jeder Fläche. Dazu addieren wir die z-Werte aller Eckpunkte der Fläche und teilen das Ergebnis durch die Anzahl der Eckpunkte – eine einfache Mittelwertsberechnung! Exakt ist das Ergebnis natürlich nur für Flächen, die parallel zur x-y-Ebene verlaufen. Je weiter eine Fläche in den Raum hineinragt, desto größer ist die Gefahr, daß Fehler auftauchen, Sie werden allerdings sehen: Gepaart mit der Sichtbarkeitsuntersuchung bringt diese Technik bereits verblüffende Ergebnisse (ganz besonders später bei den sogenannten Rotationskörpern, auf die Sie sich bereits jetzt schon freuen dürfen!).

Haben wir die Flächen nach diesem Kriterium nach ihrer mittleren z-Koordinate sortiert, ist es uns ein leichtes, sie nacheinander auf den Bildschirm zu zeichnen. Die später gezeichneten Flächen überdecken dann eventuell die vorherigen. Das folgende Basic-Programm wird Ihnen die ganze Technik veranschaulichen:

```
'************************************
'**                                **
'**  Verdeckte Linien und Flächen  **
'**                                **
'************************************

PI = 3.141593

'Einen neuen Bildschirm mit Fenster öffnen:
anz.farben% = 16
SCREEN 2,640,200,4,2
wx% = 631
wy% = 186
WINDOW 2,"Verdeckte Linien und Flächen",(0,0)-(wx%,wy%),4+8,2

blauf = 0      'Blaufaktor
gruenf = 0     'Grünfaktor
rotf = 1       'Rotfaktor
blauadd = .1   'Blau-Zusatz
gruenadd = .1  'Grün-Zusatz
rotadd = 0     'Rot-Zusatz

'Farben belegen:
FOR i=2 TO anz.farben%-1
  farbe = INT(i*100/15 + .5)/100
  rot   = farbe*rotf+rotadd
  gruen = farbe*gruenf+gruenadd
  blau  = farbe*blauf+blauadd
  IF rot>1 THEN rot=1
  IF gruen>1 THEN gruen=1
  IF blau>1 THEN blau=1

  PALETTE i,rot,gruen,blau
NEXT i

PALETTE 0,0,0,0      'Hintergrund: schwarz
PALETTE 1,1,.8,.13   'Rahmenfarbe


' Start-Transformationsparameter:
' Skalierung:
sx = 2
sy = 2
sz = 2
```

```
' Translation:
tx = 5
ty = 5
tz = 5

' Rotation:
rx = -20
ry = 20
rz = 0

' Beobachter:
bx = 0
by = 0
bz = 150

' Lichtquelle:
lq.x& = -900
lq.y& = 1000
lq.z& = -500

' Lichtintensität/Flächenintensität:
li.int = .7
fl.int = 1
hin.int = .3

' Ebenentranslation:
tex = 300
tey = 90

' Ebenenskalierung:
sex = 2
sey = 1

' Umrechnung der Drehwinkel in RAD:
rx = rx*PI/180
ry = ry*PI/180
rz = rz*PI/180

'Drehinkrementwert (Grad):
dig = 10

di = PI * dig/180              'Umrechnung nach RAD

'Beobachterinkrementwert:
binc = 20
```

```
POKE WINDOW(8)+27,1            'Farbe des Area-Outline-Pen
flags = WINDOW(8)+32
POKEW flags,PEEKW(flags) OR 8 'Area-Outline-Flag setzen

'Objekt-Daten in Arrays einlesen und
'folgende Arrays definieren:
'xr%(), yr%(), zr%() -- Koordinaten der Raumpunkte
'xe(),ye(),ze()      -- Transformierte Koordinaten
'fld%(,)             -- Flächendefinitionen
'flz%(,)             -- zu zeichnende Flächen
'anzeckd%()          -- Anzahl der Ecken jeder def. Fläche
'anzeckz%()          -- Anzahl der Ecken jeder zu
'                       zeichnenden Fläche
'sort.f%()           -- Sortierte Indizes der Flächen
'mit.z()             -- Array gemittelter z-Werte der Flächen
'farben()            -- Farbintensitäten aller zu
'                       zeichnender Flächen

get.objects


'Hauptschleife:
'***************

flag%=0
WHILE flag%<>1

  transform        'Alle Punkte transformieren
  verdecke         'Verdeckte Flächen ausfiltern
                   '(und Farben ermitteln)
  projektion       'Alle Punkte projizieren

  CLS              'Fenster löschen

  PATTERN &HFFFF   'Linienmuster = durchgezogen

  'Objekt zeichnen:
  FOR i=0 TO afz%-1                    'Alle Flächen
    flaech.nr% = sort.f%(i)            'zu zeichnende Flächennr.
      FOR k=0 TO anzeckz%(flaech.nr%)-1 'Alle Eckpunkte der Fläche
      punkt.nr% = flz%(flaech.nr%,k)   'Punktnummer
      x% = tex + sex*xe(punkt.nr%)     'Punktkoordinaten
      y% = tey - sey*ye(punkt.nr%)
```

```
    IF x%<0    THEN x% = 0
    IF x%>=wx% THEN x% = wx%-1
    IF y%<0    THEN y% = 0
    IF y%>=wy% THEN y% = wy%-1

    AREA (x%,y%)

  NEXT k

  COLOR INT(farbe(flaech.nr%)*14)+2 'Farbwert einstellen
  AREAFILL 0                        'Fläche zeichnen
NEXT i

maus% = 0 : flag%=0
WHILE maus%<>1 AND flag%=0
  SLEEP           'Auf Ereignis warten

  'Mauskoordinaten als neue Nullpunkt-
  'Koordinaten übernehmen:
  maus%=MOUSE(0)
  IF maus%=1 THEN
    tex = MOUSE(3)
    tey = MOUSE(4)
  END IF

  ch% = ASC(INKEY$+CHR$(0))
  IF ch%=31 THEN  'Cursor links =>
    ry=ry+di      'y-Achsendrehwinkel erhöhen
    flag% = -1    'Flag für Neuzeichnen
  END IF
  IF ch%=30 THEN  'Cursor rechts =>
    ry=ry-di      'y-Achsendrehwinkel erniedrigen
    flag% = -1    'Flag für Neuzeichnen
  END IF
  IF ch%=28 THEN  'Cursor hoch =>
    rx=rx+di      'x-Achsendrehwinkel erhöhen
    flag% = -1    'Flag für Neuzeichnen
  END IF
  IF ch%=29 THEN  'Cursor runter =>
    rx=rx-di      'x-Achsendrehwinkel erniedrigen
    flag% = -1    'Flag für Neuzeichnen
  END IF
  IF ch%=127 THEN 'Del => Flächenbegrenzungen ein/aus
    POKEW flags,PEEKW(flags) XOR 8
    flag% = -1
```

```
END IF
IF ch%=43 THEN  '+ => Skalierung auf
  sx = sx+1
  sy = sy+1
  sz = sz+1
  flag% = -1
END IF
IF ch%=45 THEN  '- => Skalierung ab
  sx = sx-1
  sy = sy-1
  sz = sz-1
  flag% = -1
END IF
IF ch%=56 THEN  '8 => Beobachter entfernen
  bz = bz-binc
  flag% = -1
END IF
IF ch%=50 THEN  '2 => Beobachter annähern
  bz = bz+binc
  flag% = -1
END IF
IF ch%=139 THEN 'Help => Rotation = 0
  rx = 0
  ry = 0
  rz = 0
  flag% = -1
END IF
IF ch%=8 THEN   'Backstep => Farbwechsel
  'Farbparameter rotieren:
  zwis     = rotf
  rotf     = gruenf
  gruenf   = blauf
  blauf    = zwis
  zwis     = rotadd
  rotadd   = gruenadd
  gruenadd = blauadd
  blauadd  = zwis

  'Farben belegen:
  FOR i=2 TO anz.farben%-1
    farbe = INT(i*100/15 + .5)/100
    rot   = farbe*rotf+rotadd
    gruen = farbe*gruenf+gruenadd
    blau  = farbe*blauf+blauadd
    IF rot>1 THEN rot=1
```

```
        IF gruen>1 THEN gruen=1
        IF blau>1 THEN blau=1

        PALETTE i,rot,gruen,blau
      NEXT i
    END IF

    'Falls Fenster geschlossen -> Ende
      IF WINDOW(7)=0 THEN
      flag%=1
    END IF

  WEND
WEND

WINDOW OUTPUT 1
SCREEN CLOSE 2

END


'Objektdaten einlesen:

SUB get.objects STATIC

  'Punktekoordinaten einlesen:
  SHARED ap%

  READ ap%              'Anzahl Punkte

  DIM SHARED xr%(ap%-1),yr%(ap%-1),zr%(ap%-1)
  DIM SHARED xe(ap%-1),ye(ap%-1),ze(ap%-1)

  FOR i=0 TO ap%-1
    READ xr%(i),yr%(i),zr%(i)
  NEXT i


  'Flächendefinitionen einlesen:
  SHARED afd%, afz%

  READ afd%             'Anzahl definierte Flächen
  afz% = afd%           'Anzahl zu zeichnende Flächen
  READ eckenmax%        'Max. Anzahl von Ecken einer Fläche
```

```
  DIM SHARED fld%(afd%-1,eckenmax%-1),flz%(afz%-1,eckenmax%-1)
  DIM SHARED anzeckd%(afd%-1),anzeckz%(afz%-1)
  DIM SHARED sort.f%(afz%-1),mit.z(afz%-1)
  DIM SHARED farbe(afz%-1)

  FOR i=0 TO afd%-1
    READ anzeckd%(i)  'Anzahl Ecken dieser Fläche
    FOR k=0 TO anzeckd%(i)-1
      READ fld%(i,k)
    NEXT k
  NEXT i

END SUB


'Transformation aller Raum-Punkte:
SUB transform STATIC
  SHARED ap%
  SHARED sx,sy,sz,tx,ty,tz,rx,ry,rz

  'Konstanten für die Drehung berechnen:
  si.x = SIN(rx) : co.x = COS(rx)
  si.y = SIN(ry) : co.y = COS(ry)
  si.z = SIN(rz) : co.z = COS(rz)

  A = co.y * co.z
  B = co.y * si.z
  C = -si.y
  D = si.x*si.y*co.z - co.x*si.z
  E = si.x*si.y*si.z + co.x*co.z
  F = si.x*co.y
  G = co.x*si.y*co.z + si.x*si.z
  H = co.x*si.y*si.z - si.x*co.z
  J = co.x*co.y

  FOR i=0 TO ap%-1
    ' Transformationen:
    xt = sx*xr%(i) + tx          'Skalierung
    yt = sy*yr%(i) + ty          'und Translation
    zt = sz*zr%(i) + tz

    xe(i) = xt*A + yt*B + zt*C  'Rotationen
    ye(i) = xt*D + yt*E + zt*F
    ze(i) = xt*G + yt*H + zt*J
  NEXT i

END SUB
```

```
'Projektion aller Raum-Koordinaten in die Ebene:
SUB projektion STATIC
  SHARED ap%
  SHARED bx,by,bz

  FOR i=0 TO ap%-1
      ' Zentralprojektion:
      zwis = ze(i) - bz
      xe(i) = bx - bz * (xe(i)-bx)/zwis
      ye(i) = by - bz * (ye(i)-by)/zwis
  NEXT i
END SUB


'Ausfiltern aller verdeckten Flächen
'und Sortieren der mittleren z-Werte:

SUB verdecke STATIC
  SHARED afd%, afz%
  SHARED bx,by,bz

  afz% = 0            'Anzahl zu zeichnender Flächen

  FOR i=0 TO afd%-1

    'Phase 1: Flächenrückenuntersuchung:
    '**********************************

    'Zwei (hoffentlich) linear unabhängige
    'Vektoren der Fläche ermitteln:
    'v = P2-P1 // w = P3-P1:
    'mit: P1,P2,P3 = Eckpunkte der Fläche
    P1% = fld%(i,0)
    P2% = fld%(i,1)
    P3% = fld%(i,2)
    P1.x& = xe(P1%)
    P1.y& = ye(P1%)
    P1.z& = ze(P1%)
    v.x& = xe(P2%) - P1.x&
    v.y& = ye(P2%) - P1.y&
    v.z& = ze(P2%) - P1.z&
    w.x& = xe(P3%) - P1.x&
    w.y& = ye(P3%) - P1.y&
    w.z& = ze(P3%) - P1.z&
```

```
'Vektorprodukt p = v x w ermitteln:
p.x& = v.y&*w.z& - v.z&*w.y&
p.y& = v.z&*w.x& - v.x&*w.z&
p.z& = v.x&*w.y& - v.y&*w.x&

'Sichtvektor s vom Beobachter zu P1 berechnen:
s.x& = P1.x& - bx
s.y& = P1.y& - by
s.z& = P1.z& - bz

'Skalarprodukt q = p * s berechnen:
q& = p.x&*s.x& + p.y&*s.y& + p.z&*s.z&

'Vorzeichen von q testen:
IF q&>0 THEN

  'Fläche als sichtbar kennzeichnen:
  anzeckz%(afz%) = anzeckd%(i)

  sum.z = 0
  FOR k=0 TO anzeckd%(i)-1
    flz%(afz%,k) = fld%(i,k) 'Flächendef. übertragen
    sum.z = ze(fld%(i,k)) + sum.z 'z-Summe bilden
  NEXT k


  GOSUB farbe          'Farbwert der Fläche errechnen


  'Phase 2: z-Mittelwert einsortieren:
  '**********************************

  'z-Mittelwert in z-Array speichern:
  mittel  = sum.z / (anzeckz%(afz%)-1)
  mit.z(afz%) = mittel

  'Nach z-Mittelwert den Flächenindex in Sort-Array einordnen:
  k = 0
  'Einsortierstelle suchen:
  WHILE (mittel <= mit.z(sort.f%(k)) AND k < afz%)
    k=k+1
  WEND
  IF k <= afz% THEN
    'Ab Einsortierstelle verschieben:
    FOR m=afz% TO k STEP -1
```

```
          sort.f%(m+1) = sort.f%(m)
        NEXT m
      END IF
      sort.f%(k) = afz%  'Index der Fläche merken

      afz% = afz%+1      'Anzahl sichtbare Flächen erhöhen

    END IF

  NEXT i                 'Nächste Fläche

  EXIT SUB

  'Flächen-Farbintensitäten errechnen:
  farbe:
    SHARED lq.x&, lq.y&, lq.z&
    SHARED li.int, fl.int, hin.int

    ' Vektor vom Flächenpunkt zur Lichtquelle:
    L.x& = P1.x& - lq.x&
    L.y& = P1.y& - lq.y&
    L.z& = P1.z& - lq.z&

    ' Intensität des einfallendes Lichtes:
    cos.a = (p.x&*p.x& + p.y&*p.y& + p.z&*p.z&)
    cos.a = cos.a * (L.x&*L.x& + L.y&*L.y& + L.z&*L.z&)
    cos.a = (p.x&*L.x& + p.y&*L.y& + p.z&*L.z&)/SQR(cos.a)

    farb = hin.int*fl.int          'Hintergrundintensität
    IF cos.a < 0 THEN               'Fläche dem Licht zugewandt?
      farb = farb - li.int*fl.int * cos.a ' JA! Lichteinfall
    END IF                                ' ermitteln
    farbe(afz%) = farb - INT(farb) 'nur zwischen 0 und 1!
  RETURN

END SUB

'Objektdaten:
'************

'3-D-Punktkoordinaten:

'Anzahl der Punkte:
DATA 10
```

```
'Objektpunkte:
DATA  0, 0, 0,   6, 0, 0,   6,10, 0
DATA  0,10, 0,   3,15, 0,   3,15,15
DATA  6,10,15,   6, 0,15,   0, 0,15
DATA  0,10,15


'Flächendefinitionen:

'Anzahl der Flächen:
DATA 9

'Maximale Anzahl an Ecken einer Fläche:
DATA 4

'Anzahl der Ecken + Eckpunktnummern der Flächen:
DATA 4,  3, 0, 1, 2
DATA 4,  2, 1, 7, 6
DATA 4,  6, 7, 8, 9
DATA 4,  9, 8, 0, 3
DATA 4,  0, 8, 7, 1
DATA 4,  4, 2, 6, 5
DATA 4,  5, 9, 3, 4
DATA 3,  3, 2, 4
DATA 3,  6, 9, 5
```

Recht lang ist das Programm, und trotzdem sollte Ihnen einiges bereits bekannt vorkommen. Viele Dinge kennen Sie nämlich bereits aus den anderen Kapiteln: Skalierungen, Translationen, Rotationen, Zentralprojektion und und und... Also scheuen Sie sich nicht, das Programm einmal genauer anzuschauen, es lohnt sich in jedem Fall!

Sobald Sie das Programm starten, erscheint – wie gewohnt – ein neuer Screen auf dem Monitor. Nicht nur das, nach kurzer Zeit sehen Sie... jawohl, das altbekannte Haus, diesmal ein wenig perfektioniert. Lassen Sie uns kurz – bevor Sie sich mit den Innereien des Programmes vertraut machen – an die Bedienung denken. Sie sind natürlich wieder einmal in der Lage, das Haus (oder jedes beliebige andere Objekt) per Tastendruck zu verändern. Folgende Tasten stehen Ihnen zur Verfügung:

<Maustaste li.>	Positionierung des Objektes
<Cursor links>	Drehwinkel um die y-Achse erhöhen
<Cursor rechts>	Drehwinkel um die y-Achse erniedrigen
<Cursor auf>	Drehwinkel um die x-Achse erhöhen
<Cursor ab>	Drehwinkel um die x-Achse erniedrigen
<Del>	Flächenumrahmung ein/aus
< + >	Objekt vergrößern

<->	Objekt verkleinern
<8>	Beobachter entfernen
<2>	Beobachter annähern
<Help>	alle Rotationswinkel = 0
<Backstep>	Farbwechsel
<Close-Gadget>	Programm beenden

Lassen Sie uns keine Zeit verlieren, steigen wir direkt in das Programm ein:

Am Anfang stehen natürlich wie immer die gewohnten Initialisierungen. Dabei wurde ein wenig mehr Aufwand bei der Farbgebung getrieben. Das Objekt soll nämlich später per Tastendruck in den Farben Rot, Grün oder Blau (eventuell mit jeweils anderen Farbzusätzen) erscheinen. Jede Farbe existiert in 16 Schattierungen (von 0 bis 15). Die 14 hellsten Intensitäten werden in 14 Palettenregistern gespeichert, so daß sie jederzeit zur Verfügung stehen. Die unteren zwei Register sind für die Rahmen- und Hintergrundfarben reserviert. Den zugehörigen Screen hat das Programm deshalb natürlich mit vier Farbebenen (16 Farben) eingerichtet.

Die beschriebene Farbinitialisierung findet in der ersten FOR...NEXT-Schleife des Programmes statt. Lassen Sie sich von der komplizierten Formel dort nicht irritieren. Hier werden einfach die Farbintensitätswerte 0–15 in die Werte zwischen null und eins (mit Rundung) umgerechnet, die das Amiga-Basic in seinem Palettenbefehl benötigt. Anschließend rechnet das Programm diese Intensitäten für jede der drei Grundfarben aus. Hier können Sie nach Belieben und Gefallen Änderungen vornehmen, falls die gewählten Farben nicht ganz Ihrem ästhetischen Empfinden entsprechen.

Die nächsten Schritte sollten Ihnen bekannt vorkommen. Hier werden Startwerte für die Skalierung, Translation, Rotation, für die Beobachterposition, die Ebenentranslation und -skalierung sowie einige Inkrementwerte zur tastenmäßigen Änderung dieser Parameter vorgewählt. Auch hier sollte Ihr Betätigungsfeld bei der »Erforschung« dieses Programmes sein, hier können Sie im wahrsten Sinne des Wortes schalten und walten.

Bevor wir endlich zum Kern der Sache vorstoßen, noch ein Hinweis auf eine interessante Passage in diesem Programm:

```
POKE WINDOW(8)+27,1
flags = WINDOW(8)+32
POKEW flags,PEEKW(flags) OR 8
```

Mit dem ersten POKE setzen wir die Farbe des sogenannten OUTLINE-Pens auf Palettenregister 1 (C-Freunden ist vielleicht die Funktion **SetOPen( )** (die eigentlich gar keine Funktion ist) ein Begriff). Dieser »Farbstift« wird von manchen Grafikfunktionen des Betriebssystems verwendet (z.B. von den Area-Befehlen), um beispielsweise Umrahmungen von Flächen zu zeichnen. Um diesen Funktionen mitzuteilen, daß sie ab sofort solche Rahmen zeichnen sollen, wird mit dem zweiten POKE das Area-OUTLINE-Flag gesetzt. Im Laufe des Programmes werden Sie eine zweite Stelle finden, an der dieses Flag als Reaktion auf einen Tastendruck (Del) wieder gelöscht wird. Jede Fläche wird von Stund an ohne Rahmen gezeichnet. Wir können uns dazu nicht der Funktion SetOPen( ) bedienen, da diese nur ein C-Makro darstellt und gar nicht im Befehlssatz des Betriebssystems vorhanden ist.

Aber jetzt endlich zum Wesentlichen! Eingeleitet wird die ganze Operation durch einen Sprung zur Routine get.objects. Wie Sie dem Kommentar im Programm entnehmen, werden hier zahlreiche Arrays eingerichtet, Daten aus DATA-Zeilen gelesen usf. Die Routine selbst sollte nicht allzu schwer zu verstehen sein. Allein die Funktion der Arrays muß eingehender untersucht werden. Die folgenden dort initialisierten Felder haben zentrale Bedeutung im ganzen Programm:

xr%( ),yr%( ),zr%( )	Raumkoordinaten aller Punkte
xe( ),ye( ),ze( )	Rechenarrays für Punktkoordinaten später: Ebenenkoordinaten
fld%(,)	alle definierten Flächen bestehend aus den Eckpunktnummern
anzeck%( )	Anzahl der Ecken aller definierten Flächen
flz%(,)	später: alle zu zeichnenden(!) Flächen
anzeckz%( )	Anzahl der Ecken aller zu zeichnenden Flächen
sort.f%( )	Nummern der sortierten Flächen
mit.z%( )	Sortierte mittlere Tiefen
farbe( )	Farbintensitätswerte aller Flächen

Diese teilweise erst später verwendeten Arrays enthalten alle Informationen, die das Programm benötigt, um alle gewünschten Objekte in der richtigen Art und Weise zu zeichnen. Wir werden sie im Laufe der Erläuterungen noch näher kennenlernen.

Wie Sie vielleicht bereits bemerkt haben, ist in diesem Programm ganz auf irgendwelche Liniendefinitionen verzichtet worden. Kein Wunder, wir haben es schließlich auch mit Flächen zu tun. Ein Objekt besteht also aus einer Vielzahl von Flächen. Jede Fläche wiederum besteht aus mehreren (mindestens drei) Eckpunkten, die in ganz genau bezeichneter Reihenfolge, im Uhrzeigersinn (Sie erinnern sich an den Hidden-Surface-Algorithmus!) aufgelistet sind. Jede Flächendefinition besteht also aus einer Zahl, die die Anzahl der Eckpunkte angibt (anzeck%( )) und einem Array von Eckpunkten. Um ein Objekt zu kreieren, müssen wir also zunächst einmal alle Eckpunkte sammeln und in dem Punktearray bereitstellen, dann sämtliche Flächen dieses Objektes numerieren und Fläche für Fläche, Eckpunkt für Eckpunkt auflisten.

Aber kehren wir doch zurück ins Hauptprogramm, das dort umgehend mit der großen Hauptschleife fortfährt. Sie wird erst dann beendet, wenn Sie das Ausgabefenster per CLOSE-Gadget schließen. Ohne große Umschweife ruft man hier drei große Routinen auf, die alle räumlichen Daten aufbereiten und zur Zeichnung fertigstellen:

transform	Transformation aller Raumpunkte
verdecke	Erkennen aller zu zeichnenden Flächen und Tiefensortierung (sowie Farbberechnung)
projektion	Umrechnung in ebene Punktkoordinaten

Beginnen wir bei der ersten. Hier findet nun nichts Spannendes mehr statt. Genau diese Routine sollten Sie bereits aus anderen Programmen kennen. Rotation, Translation und Skalierung aller Raumpunkte gehören zu ihren Aufgaben.

Hochinteressant geht es dagegen in der Routine »verdecke« zu; das ist unser Thema! Zur Einleitung der Flächenrückenuntersuchung wählt das Programm erst einmal zwei linear unabhängige Vektoren der Ebene aus. Das setzt allerdings voraus, daß Sie die Daten Ihrer Fläche korrekt

eingegeben haben. Sollten nämlich drei Eckpunkte (ausgerechnet die drei Punkte, die für die Rechnung herangezogen werden) auf einer Linie liegen, dann wird es haarig. Dann kann es nämlich vorkommen, daß die beiden ausgesuchten Vektoren eben nicht linear unabhängig sind. Das Ergebnis: recht ordentliches Chaos.

Ausgewählt werden der Vektor $\vec{v}$ vom ersten (P1) zum zweiten Eckpunkt (P2) und der Vektor $\vec{w}$ vom ersten zum dritten (P3). Ein Vektor berechnet sich bekanntlich aus der Differenz zweier Punktvektoren. Entsprechendes wird in der Routine ermittelt.

Im Anschluß daran muß gemäß Algorithmus der senkrechte Vektor auf die Fläche durch das Vektorprodukt $\vec{p} = \vec{v} \times \vec{w}$ bestimmt werden, dann der Sichtvektor $\vec{s}$ vom Beobachter zum Punkt P1 und dann das Skalarprodukt $q = \vec{p} * \vec{s}$. Jetzt sind wir dort, wo wir hin wollten. Jetzt können wir das Vorzeichen von q testen. Ist es negativ oder Null, dann ist die Fläche nicht sichtbar (die nächste Fläche kann untersucht werden), ist es dagegen positiv, dann können wir sie in das Array speichern, das alle zu zeichnenden Flächen enthält: **flz%(,)**. Das Eckenzahl-Array **anzeckz%( )** darf selbstverständlich nicht vergessen werden. Damit wäre die Realisierung des Flächenrückenalgorithmus bereits vollendet!

Als nächstes treffen Sie mit dem Unterprogrammaufruf »farben« auf ein kleines Intermezzo, das strukturell eigentlich gar nicht hierhin gehört. Da wir aber in Amiga-Basic ein wenig Wert auf Programmausführungszeiten legen müssen, können Sie vielleicht noch einmal ein Auge zudrücken. Es geht um die Farbgebung. Natürlich ist es sehr einfach, jeder zu zeichnenden Fläche die gleiche Farbe zu geben und fertig. Der Perfektionist wird aber gleich die Frage nach einer irgendwie gearteten Lichtquelle auf den Lippen haben, die für diverse Farbnuancen sorgt und den räumlichen Effekt enorm erhöht. Nun gut, soll er haben. In diesem Unterprogramm berechnet Ihr Amiga die Intensität des Lichtes, das auf die jeweilige Fläche trifft. Diese ist selbstverständlich unterschiedlich groß, je nachdem, in welchem Winkel die Fläche zum Licht steht (ein ähnliches Problem also wie mit dem Beobachter). Die dazu verwendete Formel (ja, es ist eine einfache Formel und kein komplizierter Algorithmus) soll hier nicht erläutert werden. Das geschieht nämlich in dem Ray-Tracing-Kapitel, dem Höhepunkt unseres Buches, den Sie sich nicht entgehen lassen sollten. Schlagen Sie also dort einmal nach, wenn Sie Näheres wissen wollen. Nur eins: Zur Berechnung der Formel bedarf es der Kenntnis des senkrechten Vektors auf die Fläche, den wir in der Sichtbarkeitsuntersuchung bereits ausgerechnet hatten und der jetzt gerade noch bereitsteht. Deshalb liegt die Routine ausgerechnet hier.

Die fertige Farbintensität (ein Wert zwischen Null und Eins) speichert das Programm dann in das Array farbe( ) für den späteren Gebrauch.

Und jetzt tritt bereits Phase 2 der Sichtbarkeitsuntersuchung in Aktion: Die Flächen werden nach Ihrer z-Priorität sortiert. Wie oben erwähnt, vereinfachen wir die Sache, indem wir lediglich einen mittleren z-Wert für jede Fläche ermitteln. Er errechnet sich aus der Summe der z-Werte aller Eckpunkte, dividiert durch die Anzahl der Eckpunkte. Exakt gilt der ermittelte z-Wert natürlich nur für die Flächenmitte, aber meist entstehen dadurch keine Probleme.

Aufgabe der zweiten Phase ist es nun, die Fläche so einzusortieren, wie es ihrem mittleren z-Wert entspricht. Später nämlich sollen dann alle Flächen in der Reihenfolge mit absteigendem z gezeichnet werden. Das Einsortieren geschieht durch einfaches Einschieben an die richtige

Stelle. Dabei wird allerdings nicht mit den Flächendefinitionen selbst, sondern nur mit den Nummern der Flächen gearbeitet. Im Array **sort.f%( )** stehen dann nachher die Nummern aller Flächen in der Reihenfolge, in der sie gezeichnet werden sollen.

Damit haben wir bereits alles bezüglich Sichtbarkeit oder Unsichtbarkeit einer Fläche getan, und Sie sollten sich wieder zurück ins Hauptprogramm begeben. Die nächste Funktion heißt »projektion«. Nicht mehr besonders schwierig, nicht besonders lang und auch nicht besonders unbekannt: die Zentralprojektion. Sparen wir uns lange Erklärungen.

Der nächste Schritt ist endlich die Zeichnung: Bildschirm löschen, Linienmuster einstellen, Fläche für Fläche zu Mattscheibe bringen.

Die Anzahl der zu zeichnenden Flächen steht in **afz%**. Die Nummern in der richtigen Reihenfolge in **sort.f%( )**. Für die jeweils aktuelle Fläche wird die Nummer nach flaech.nr% übertragen, die als Index (neben der Punktnummer k) für das Flächenarray **flz%(,)** dient. Aus diesem Array wird nun Punktnummer für Punktnummer nach **punkt.nr%** herausgeholt, die ihrerseits als Index für die projizierten Punktarrays xe( ) und ye( ) Verwendung finden.

Jeder Punkt wird geclippt und dem Befehl **AREA** übergeben. Ist das Maß voll, ist also jeder Eckpunkt bearbeitet worden, setzt das Programm die richtige Farbe, die ja im Array **farbe( )** für jede Fläche bereitsteht und... zeichnet: **AREAFILL.**

Die Arbeit ist bis auf weiteres erledigt, alle Flächen sind gezeichnet, das Bild steht. Nun wird auf die Benutzereingabe gewartet. Sie wird, so sie vorliegt, analysiert und entsprechende Reaktionen eingeleitet. Was soll ich Ihnen sagen, dann beginnt das Spiel von vorne – alles in Basic-Geschwindigkeit natürlich, aber was soll's?

Alles klar? Na dann stürzen Sie sich hinein, erstellen Sie einmal Ihr eigenes Objekt oder mehrere. Vielleicht erweitern Sie das Programm ja auch für Ihre eigenen Bedürfnisse. Viel Spaß!

KAPITEL 6

Es wird realistisch: perfekte 3-D-Grafik mit Lichteinfall, Reflexion und Schattierung durch Ray-Tracing – voller Farbeinsatz auf dem Amiga

Was unser Rechner bisher bezüglich Grafik auf den Bildschirm gezaubert hat, ist sicherlich recht beeindruckend. Beeindruckend natürlich auch die Geschwindigkeit unseres Rechners, der es schafft, komplizierteste Rechnungen auszuführen und trotzdem alle Objekte noch in Realzeit auf dem Bildschirm zu rotieren, zu vergrößern, zu verschieben und vieles mehr. Was uns bisher allerdings verwehrt blieb, war die echte, die totale Wirklichkeitsnachbildung mit allen Faktoren, die das Leben lebenswert machen.

Im normalen Leben gibt es Licht und Schatten, Gegenstände reflektieren Licht, sind matt oder glänzend, durchsichtig oder verspiegelt, blau, weiß, rot, metallfarben. Daß solche Effekte mit den Modellen, wie wir sie bisher kennengelernt haben, nur unter äußersten Anstrengungen mit den aufwendigsten mathematischen Methoden zu realisieren sind, braucht kaum erwähnt zu werden, wenn Sie sich nur einmal den Hidden-Surface-Algorithmus anschauen. Natürlich ist es möglich. Wir wollen an dieser Stelle aber eine Technik kennenlernen, die solche Dinge weitaus einfacher – wenn auch langsamer – erledigt. Selbst komplizierteste Effekte wie das Problem der verdeckten Linien oder Licht und Schatten werden mit dieser Technik zum Kinderspiel. Die Technik hat einen Namen: Ray-Tracing.

6.1 Die mathematischen Prinzipien des Ray-Tracing

Bei den Betrachtungen des Ray-Tracing wird intensiver Gebrauch von der Vektorrechnung gemacht, wie Sie sie bereits kennengelernt haben. Schauen Sie sich also die entsprechenden Kapitel am besten noch einmal an (Kapitel: »Mathematische Grundlagen zu 3-D-Berechnungen« oder im Anhang). Ohne die Vektorrechnung werden wir hier wohl nicht auskommen.

Ray-Tracing bedeutet in etwa soviel wie »Strahl-Verfolgung«. Verfolgt werden hier nämlich die Lichtstrahlen auf ihrem Weg von der Lichtquelle über die verschiedenen Objekte bis hin zum Auge des Beobachters. Mit einem kleinen Unterschied: Die Verfolgung geht genau den umgekehrten Weg. Sie startet an der Endstation der Lichtstrahlen, also beim Beobachter, und untersucht praktisch die Spur der Strahlen rückwärts zu den Objekten bis hin zur Lichtquelle. Damit ist gewährleistet, daß auch wirklich nur diejenigen Lichtstrahlen berücksichtigt werden, die tatsächlich das Auge (oder die Kamera etc.) treffen.

Technisch geht das Ganze so vonstatten: Stellen Sie sich vor, Sie sitzen vor Ihrem Bildschirm, schauen durch den Bildschirm hindurch und beobachten die dahinterliegende Welt (wir nehmen einmal an, der Bildschirm besteht nur aus einer einfachen Glasscheibe). Die Mattscheibe kann dann als Projektionsebene angesehen werden (s. Bild 6.1), auf der sich die dahinterliegende Welt abbildet.

In unserem Verfahren müssen wir nun für jeden Punkt dieses Bildschirms die passende Farbe ermitteln, insgesamt für z.B. 320x256 Punkte, wenn wir im niedrigaufgelösten Modus operieren. Die Berechnung der Farbe jedes einzelnen dieser Punkte ist also unsere Aufgabe. Dazu denken Sie sich bitte einmal einen Strahl vom Auge des Beobachters durch den zu berechnenden Punkt des Bildschirms hinaus in die Welt. Dieser Strahl wird entweder bis in die unendlichen Weiten des Raumes verlaufen oder ein bzw. mehrere Objekte, die sich in der Welt befinden, treffen.

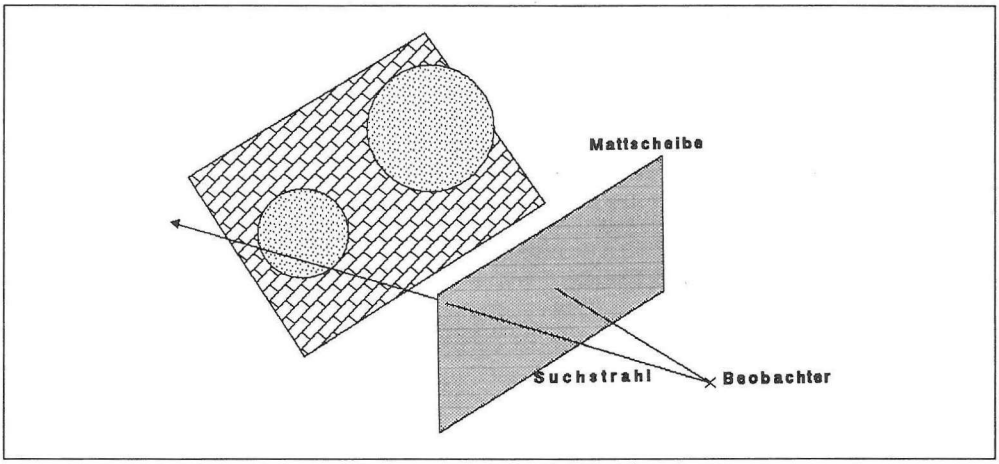


Bild 6.1: Das Prinzip des Ray-Tracing

Im ersten Fall erhält der Punkt ganz einfach die Farbe des Hintergrundes. Im zweiten Fall die Farbe desjenigen Objektes, das er zuerst trifft, das also am nächsten zum Beobachter liegt (womit übrigens das Problem der verdeckten Linien und Flächen sozusagen nebenbei gelöst ist). Es hängt nun von der Beschaffenheit dieses Objektes ab, was weiter geschieht. Um Ihnen die Technik noch ein wenig schmackhaft zu machen, gleich noch ein paar Aussichten:

Gibt es keine Lichtquelle im Raum (nur eine allgemeine diffuse Helligkeit), und handelt es sich bei diesem Objekt um ein ganz einfaches, mattes, z.B. blaues Gebilde, dann liegt die Farbe des jeweiligen Punktes auf der Mattscheibe bereits fest: blau. Das Objekt kann aber auch verspiegelt sein. In diesem Fall ist ein zweiter Strahl zu ermitteln, der nach dem Gesetz »Einfallswinkel gleich Ausfallswinkel« berechnet werden muß. Dieser Strahl kann nun wiederum in die Ferne verlaufen oder ein weiteres Objekt treffen. Trifft er ein normales Objekt, dann wird die Farbe dieses Objektes als Punktfarbe übernommen. Ist dieses Objekt ebenfalls verspiegelt, dann geht der Prozeß weiter und weiter und weiter ...

Das Objekt kann aber auch durchsichtig sein. Der Strahl wird also an der Oberfläche nach den Brechungsgesetzen gebrochen, tritt an der anderen Seite wieder aus und sucht sich das nächste Objekt.

Vielleicht gibt es auch eine oder gar mehrere Lichtquellen. In diesem Fall wird von dem Punkt, an dem der Strahl das Objekt getroffen hat, ein weiterer Strahl zur Lichtquelle gezogen. Dabei spielt dann der Winkel dieses Strahles zur Objektoberfläche sowie die Frage eine Rolle, ob sich zwischen Objektpunkt und Lichtquelle ein anderes Objekt befindet (Schatten).

Sie sehen, wie mit einfachen Mitteln komplizierteste Effekte erzielt werden können. Wir werden sie auf den folgenden Seiten kennenlernen und auch mathematisch beschreiben. Sie werden lernen, wie – und das sind auch schon die eigentlichen Probleme bei dieser Technik – die verschiedensten Objektformen behandelt werden können. Und Sie werden lernen, wie das alles programmtechnisch realisiert werden kann, was auch nicht immer ganz einfach ist.

Die Hauptarbeit eines Ray-Tracing-Programmes liegt in der Schnittpunktberechnung von Strahlen mit den verschiedenen Objekten der Welt. Etwa 75–95 % der Rechenzeit werden dafür benötigt. Die Geschwindigkeit eines Ray-Tracing-Programmes hängt also wesentlich von den Algorithmen ab, die für diese Schnittpunktberechnung mit den Objekten gewählt werden. Die Zeit zur Erstellung eines Bildes in der Größe des gesamten Amiga-Bildschirmes kann – je nach Anzahl und Komplexität der Objekte – bis zu mehrere Stunden betragen, da für jeden Bildpunkt Schnittpunkte mit jedem Objekt des Bildes errechnet werden müssen.

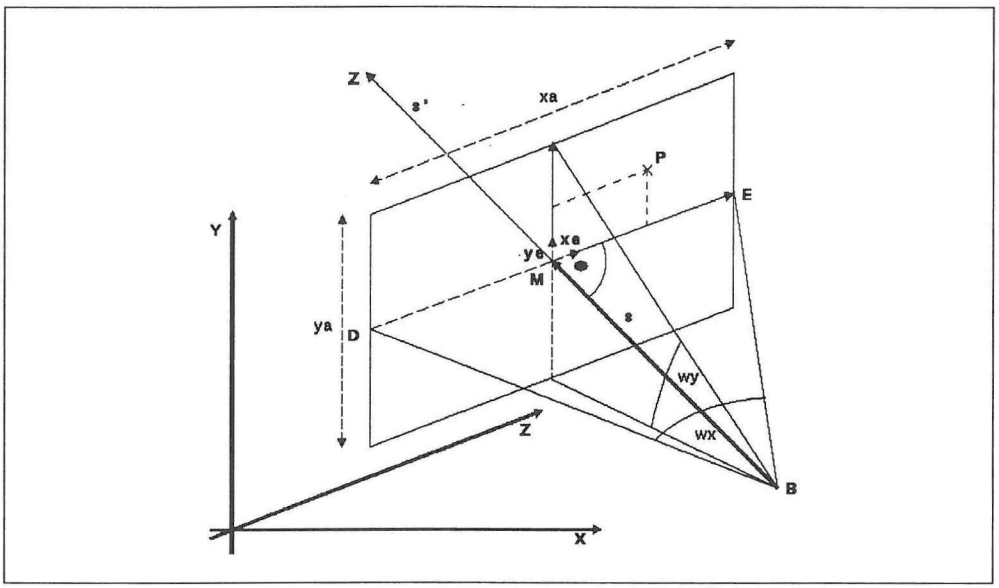


Bild 6.2: Mathematische Ableitung des Ray-Tracing

Zur mathematischen Ableitung schauen Sie sich Bild 6.2 an. Dort sind die Verhältnisse des Ray-Tracing noch einmal veranschaulicht. Sie sehen den Beobachter bei Punkt $B(b_x,b_y,b_z)$. Seine Position im räumlichen Koordinatensystem ist bekannt. Ferner ist der Punkt in der Welt bekannt, auf den der Beobachter gerade blickt: $Z(z_x,z_y,z_z)$. Das kann natürlich jeder beliebige Punkt der Landschaft sein, je nachdem, welches Objekt Sie später in der Mitte des Bildschirms plazieren möchten. Er darf allerdings nicht mit dem Beobachter B zusammenfallen.

Der Vektor von B nach Z, nennen wir ihn $\vec{s}'$, ist dann der sogenannte (verlängerte) Blickvektor, der Vektor also, der die Richtung angibt, in die der Beobachter schaut. Er steht in jedem Fall senkrecht auf der Mattscheibe. Bekannt sein muß ferner die Entfernung des Beobachters von der Mattscheibe. Dies ist die Strecke zwischen B und M (M ist der Mittelpunkt des Bildschirms). Alternativ können wir entweder direkt den verlängerten Blickvektor oder den Blickpunkt angeben. Beides hat seine Vor- und Nachteile. Während im ersten Fall bei einer Änderung der Beobachterposition der Blickpunkt stets konstant bleibt, also immer dasselbe Objekt anvisiert wird, sich dafür allerdings die Blickrichtung mit dem Beobachter ändert, haben wir es im zweiten Fall mit der umgekehrten Situation zu tun: Die Blickrichtung bleibt, der Blickpunkt ändert sich. Wir können sehr leicht beide Möglichkeiten offenhalten.

Den Vektor von B nach M nennen wir $\vec{s}$ (Sichtvektor). Die Länge dieses Vektors ist bekanntlich $|\vec{s}|$ ($\vec{s}$ absolut) und darf nicht Null sein. Er entspricht der Brennweite eines Objektivs und wird fix auf 1 gesetzt. In unserem Algorithmus verwenden wir noch zusätzlich einen sogenannten Beobachtungswinkel w_x. Er bestimmt den Bereich der Welt, der auf der Mattscheibe abgebildet wird. Den gleichen Effekt hätten wir auch mit dem Abstand des Beobachters von der Mattscheibe erreichen können, den wir ja fest auf 1 gesetzt haben. So ist es allerdings ein wenig anschaulicher.

Ein paar weitere Parameter muß Ihr Programm natürlich ebenfalls kennen: die x- und die y-Auflösung xa bzw. ya, sowie die x- und y-Punktgröße xpg und ypg, um das Bild nicht zu verzerren. Aus diesen Parametern errechnet sich übrigens auch der y-Blickwinkel wy, der allerdings nicht benötigt wird. xy und ya geben eigentlich nur die Anzahl der darzustellenden Punkte in x- und y-Richtung an, also praktisch die Fenstergröße. Je kleiner die beiden Werte gewählt werden, desto weniger Punkte werden berechnet und desto schneller ist das Bild fertig. Ferner vereinbaren wir zur Vereinfachung der Berechnungen, daß die Mattscheibe nicht um die z-Achse gedreht wird. Rechts und links gegenüberliegende Eckpunkte besitzen also stets dieselbe y-Koordinate, oben und unten gegenüberliegende Eckpunkte gleiche x-Koordinaten. Diese Vereinbarung ist nicht allzu tragisch, da Sie Ihren Monitor sicherlich nicht schrägstellen wollen.

Zunächst einmal möchten wir den Vektor $\vec{s}$ berechnen. Wir kennen nur seine Länge $|\vec{s}|$. Seine Richtung ist jedoch dieselbe wie $\vec{s}'$ (verlängerter Sicht- oder Blickvektor). Es gilt also:

$$\vec{s} = n * \vec{s}'$$

und nach der Definition von Vektorlängen:

$$|\vec{s}|^2 = s_x^2 + s_y^2 + s_z^2$$

Der Vektor $\vec{s}'$ ist leicht aus den Endpunkten B und Z zu ermitteln (beachten Sie, daß Punkte auch als Vektoren vom Nullpunkt zum Punkt angesehen werden können):

$$B + \vec{s}' = Z \quad <=> \quad \vec{s}' = Z - B$$

Es gilt also:

$$\vec{s} = n * (Z - B)$$

und damit:

$$|\vec{s}| = n * |Z - B| \quad <=>$$

$$n = \frac{|\vec{s}|}{|Z - B|}$$

Damit erhalten wir:

$$\vec{s} = \frac{|\vec{s}|}{|Z - B|} * (Z - B)$$

Die Länge |Z–B| des Vektors (Z–B) ist leicht durch die bekannte Längenformel zu errechnen:

$$|\vec{v}|^2 = v_x^2 + v_y^2 + v_z^2$$

$|\vec{v}|$ errechnet sich dann durch die Wurzel aus dem rechten Teil der Gleichung. Da $|\vec{s}|$ immer gleich 1 ist, vereinfacht sich die Gleichung entsprechend. Damit ist uns der eigentliche Blickvektor $\vec{s}$ bekannt.

Um einen beliebigen Punkt P auf der Mattscheibe anzusprechen, verwenden wir (neben dem Beobachterstandpunkt und dem Sichtvektor natürlich) die beiden Einheitsvektoren $\vec{xe}$ und $\vec{ye}$. Sie geben jeweils den x- bzw. y-Abstand zweier Punkte an. Der Abstand hängt natürlich auch von der Punktgröße xpg bzw. ypg ab. Ein Punkt P mit den Koordinaten x,y errechnet sich dann wie folgt:

$$P = B + \vec{s} + (x * xpg) * \vec{xe} + (y * ypg) * \vec{ye}$$

Beachten Sie, daß der Nullpunkt des Bildschirm-Koordinatensystems in der Bildmitte M liegt. x und y haben also Werte von $-xa/2$ bis $+xa/2$ bzw. $-ya/2$ bis $+ya/2$ (xa, ya sind die x- bzw. y-Auflösungen des Bildschirms).

Die Berechnung der Einheitsvektoren $\vec{xe}$ und $\vec{ye}$ wird Sie also bereits brennend interessieren. Nun, die ist ein wenig komplizierter. Bestimmen wir als erstes $\vec{xe}$: Da der Blickvektor $\vec{s}$ senkrecht auf der Mattscheibe stehen soll, muß das Skalarprodukt $\vec{s} * \vec{xe}$ gleich Null sein. Damit haben wir eine Aussage für die Richtung von $\vec{xe}$:

$$\vec{s} * \vec{xe} = 0 \quad <=>$$
$$s_x * xe_x + s_y * xe_y + s_z * xe_z = 0$$

Als zweites Kriterium, das sich mit der Länge von $\vec{xe}$ beschäftigt, gilt:

$$ME = (xa/2) * xpg * |\vec{xe}|$$

wobei:

ME	Strecke Mittelpunkt-> Punkt E
xa	x-Auflösung
xpg	x-Punktgröße
$\|\vec{xe}\|$	Länge des x-Einheitsvektor

Beide Kriterien müssen also erfüllt sein. Beginnen wir mit der Länge des Einheitsvektors $|\vec{xe}|$:

Betrachten Sie dazu noch einmal die Zeichnung. Das Dreieck, das von den Punkten B, M und E gebildet wird, ist rechtwinklig. Der rechte Winkel liegt im Punkt M. Der Winkel bei B ist der halbe Blickwinkel $w_x/2$. Gesucht sei die Länge der Strecke ME. Es gilt damit:

$$\tan(w_x/2) = ME / |\vec{s}| \quad <=>$$
$$ME = \tan(w_x/2) * |\vec{s}|$$

Für die Länge der Strecke ME haben wir bereits einen Ausdruck:

$$ME = (xa/2) * xpg * |\vec{xe}|$$

bei Gleichsetzung der beiden Gleichungen erhalten wir damit:

$$(xa/2)*xpg*|\vec{xe}| = \tan(w_x/2)*|\vec{s}| \quad <=>$$

$$|\vec{xe}| = \frac{2*\tan(w_x/2)*|\vec{s}|}{xa*xpg}$$

Die benötigte Länge des x-Einheitsvektors ist ermittelt.

Begeben wir uns jetzt daran, die Richtung von $\vec{xe}$ festzustellen. Ausgangspunkt ist dabei die obige Gleichung:

$$s_x*xe_x + s_y*xe_y + s_z*xe_z = 0$$

Erinnern Sie sich? Wir hatten vereinbart, daß der Rotationswinkel der Mattscheibe um die z-Achse gleich Null sein soll. Damit ist die y-Koordinate des Fußes von $\vec{xe}$ identisch mit der der Spitze (der Vektor ist parallel zur x-z-Ebene). Es gilt also zusätzlich:

$$xe_y = 0$$

Die Gleichung vereinfacht sich damit in:

$$s_x*xe_x + s_z*xe_z = 0$$

Hier müssen wir leider drei Fälle unterscheiden:

Fall 1: $s_x <> 0$:
In diesem Fall können wir die Gleichung einfach nach xe_x auflösen:

$$xe_x = -(s_z*xe_z)/s_x$$

Fall 2: $s_x = 0$ und $s_z <> 0$:
Da die Division durch Null verboten ist, ist es nicht möglich, die Gleichung nach xe_x aufzulösen. Es darf allerdings auch nicht jeder beliebige Wert für xe_x eingesetzt werden, da das zweite Kriterium (die Länge des Vektors $\vec{xe}$) ebenfalls erfüllt sein muß. Wir behelfen uns also mit der Auflösung nach xe_z:

$$xe_z = -(s_x*xe_x)/s_z$$

Fall 3: $s_x = 0$ und $s_z = 0$:
Tja, jetzt sind uns die Hände gebunden: Keine der beiden Auflösungen ist uns erlaubt. Das ist allerdings auch nicht so schlimm, da der Vektor $\vec{s}$ in diesem Falle parallel zur z-Achse verläuft (der Nullvektor darf er ja nicht sein, s_y ist also in diesem Fall ungleich Null!). Der gesuchte Einheitsvektor $\vec{xe}$ muß damit parallel zur x-Achse verlaufen. xe_z ist also gleich Null (xe_y sowieso), während xe_x dann gleich der Länge des Einheitsvektors sein muß:

$$xe_z = 0$$
$$xe_x = |\vec{xe}|$$

Fall 3 wäre damit bereits vorzeitig abgeschlossen und braucht nicht mehr berücksichtigt zu werden!

Die beiden Fälle 1 und 2 allerdings müssen wir noch ein wenig weiter verfolgen. s_x und s_z sind bekannt (Sichtvektor). Es gilt demnach, die jeweils andere Unbekannte xe_z für Fall 1 und xe_x für Fall 2 zu bestimmen. Da hilft uns wieder die Länge des Vektors $\vec{xe}$, die Sie ja bereits berechnen können:

$$|\vec{xe}|^2 = xe_x^2 + xe_y^2 + xe_z^2$$

und mit $xe_y = 0$ (s.o.):

$$|\vec{xe}|^2 = xe_x^2 + xe_z^2$$

Hier kann die jeweilige Gleichung von oben nun eingesetzt werden. Spielen wir das anhand des ersten Falles einmal durch (s_x ist also ungleich Null):

$$\begin{aligned} |\vec{xe}|^2 &= xe_z^2 + (-s_z * xe_z/s_x)^2 \quad <=> \\ |\vec{xe}|^2 &= xe_z^2 + xe_z^2 * (s_z/s_x)^2 \quad <=> \\ |\vec{xe}|^2 &= xe_z^2 * [1 + (s_z/s_x)^2] \quad <=> \end{aligned}$$

$$xe_z^2 = \frac{|\vec{xe}|^2}{1 + (s_z/s_x)^2} \quad \text{(Für Fall 1: } s_x <> 0\text{)}$$

Damit haben wir einen Ausdruck für xe_z. Den können Sie dann in den weiter oben angeführten Ausdruck für xe_x einsetzen.

Für den zweiten Fall erhalten wir ganz analog:

$$xe_x^2 = \frac{|\vec{xe}|^2}{1 + (s_x/s_z)^2} \quad \text{(Für Fall 2: } s_x = 0 \text{ und } s_z <> 0\text{)}$$

Auf die gleiche Weise ermitteln wir nun auch den y-Einheitsvektor. Auch hier sind drei Fälle zu unterscheiden. Bei ihm wird dann allerdings die x-Koordinate gleich Null. Wir kommen zu den folgenden Gleichungen:

Fall 1: $s_y <> 0$:

$$\vec{ye}_y = -s_z * ye_z/s_y$$

$$ye_z^2 = \frac{|\vec{ye}|^2}{1 + (-s_z/s_y)^2}$$

Fall 2: $s_y = 0$ und $s_z <> 0$:

$$ye_z = -s_y * ye_y/s_z$$

$$ye_y^2 = \frac{|\vec{ye}|^2}{1 + (-s_y/s_z)^2}$$

Fall 3: $s_y = 0$ und $s_z = 0$:

$$\begin{aligned} ye_z &= 0 \\ ye_y &= |\vec{ye}| \end{aligned}$$

für alle Fälle gilt dann noch:

$|\vec{ye}| = |\vec{xe}|$

Die Länge der beiden Einheitsvektoren ist selbstverständlich gleich, braucht also nur einmal am Anfang berechnet werden.

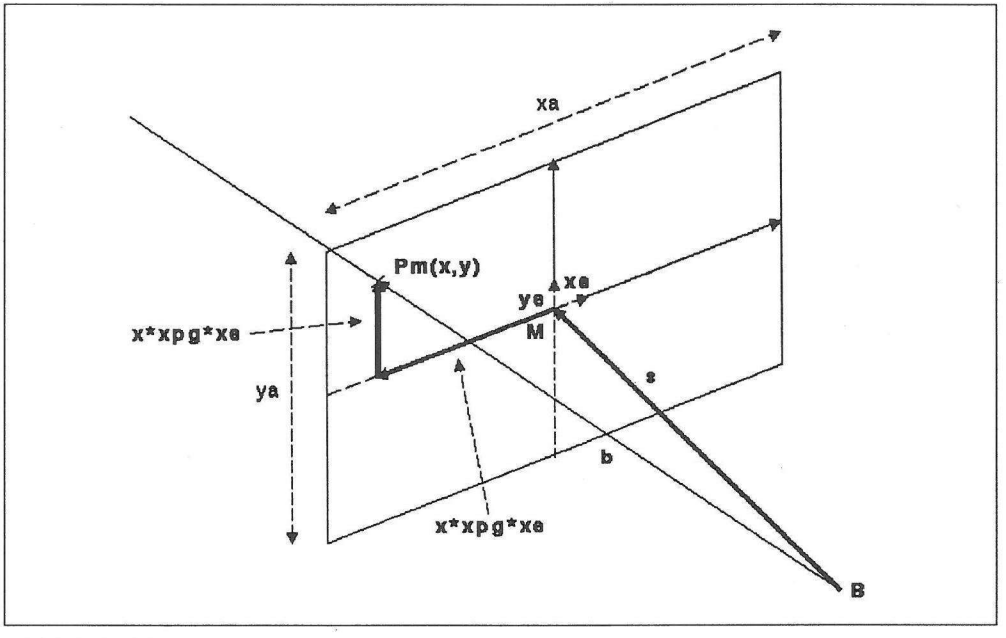


Bild 6.3: Strahlberechnung

Damit haben wir eigentlich die Grundlagen gelegt. Im Programm werden dann in zwei großen ineinanderverschachtelten FOR...NEXT-Schleifen alle Punkte von –xa/2 bis +xa/2 und von –ya/2 bis +ya/2 abgeklappert. Mit der oben bereits angeführten Formel:

$$P_m = B + \vec{s} + x*xpg*\vec{xe} + y*ypg*\vec{ye}$$

berechnet man den jeweiligen Punkt auf der imaginären Mattscheibe (s. Bild 6.3). Durch diesen Punkt muß dann ein Strahl vom Beobachter B in den Raum gezogen werden. Dieser Strahl (eine Gerade) berechnet sich nach der Vektorform der Geradengleichung:

$$P = P_m + k*\vec{b}$$

mit:

$$\vec{b} = P_m - B$$

also:

$$P = P_m + k*(P_m - B)$$

wobei:

P_m	Punkt auf der Mattscheibe
B	Beobachter
$\vec{b}$	Basisvektor von B nach P_m
k	Verlängerungsfaktor
P	Ein Punkt der Geraden

Nun müssen nacheinander alle in der Welt befindlichen Objekte darauf getestet werden, ob sie von der Geraden geschnitten oder berührt werden. Ist das bei mehreren Objekten der Fall, dann wird derjenige Schnittpunkt für die weiteren Berechnungen verwandt, der am nächsten zum Beobachter liegt, der also das kleinste k besitzt. Wie Sie bei den verschiedenen Objekten einzelne Schnittpunkte errechnen, das erfahren Sie im nächsten Kapitel.

6.2 Schnittpunktberechnungen an Körpern und Flächen

6.2.1 Die Kugel

Am einfachsten, und vor allem am schnellsten, gestaltet sich die Berechnung eines Schnittpunktes zwischen der Sichtgeraden und einem Körper bei der Kugel. Die Kugel ist deshalb das wohl am häufigsten verwendete Objekt beim Ray-Tracing.

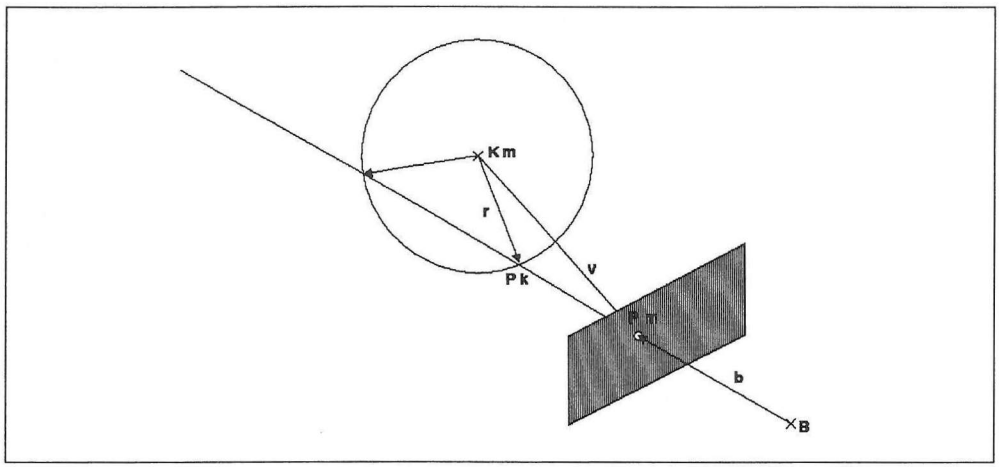


Bild 6.4: Schnittpunktberechnung bei der Kugel

Schauen Sie sich dazu Bild 6.4 an. Gesucht ist der Radiusvektor $\vec{r}$, dessen Länge dem Radius der Kugel mit dem Mittelpunkt K_m entspricht. $\vec{d}$ hat seinen Fuß im Kugelmittelpunkt K_m, seine Spitze auf der Geraden im Punkt P_k:

$$\vec{r} = P_k - K_m$$

Pk ist Punkt der Geraden durch den Mattscheibenpunkt P_m mit dem Richtungsvektor $\vec{b}$:

$$P_k = P_m + k*\vec{b}$$

also:

$$\vec{r} = P_m + k*\vec{b} - K_m \quad <=>$$
$$\vec{r} = (P_m - K_m) + k*\vec{b}$$

Gesucht ist also der Verlängerungsfaktor k, der den Radiusvektor $\vec{r}$ liefert. Da die Länge von $\vec{r}$ der Radius R der Kugel ist, gilt ferner:

$$|\vec{r}|^2 = R^2$$

Setzen Sie die obige Formel für r ein:

$$|(P_m - K_m) + k*\vec{b}|^2 = R^2$$

Für den Vektor $(P_m - K_m)$ setzen wir nun der Einfachheit halber: $\vec{v}$. Dann gilt:

$$v = P_m - K_m$$

und damit:

$$v_x = P_{mx} - K_{mx}$$
$$v_y = P_{my} - K_{my}$$
$$v_z = P_{mz} - K_{mz}$$

Es geht weiter mit:

$$|\vec{v} + k*\vec{b}|^2 = R^2 \quad <=>$$
$$\vec{v}^2 + 2*k*\vec{v}*\vec{b} + k^2*\vec{b}^2 = R^2 \quad <=>$$
$$(\vec{b}^2)*k^2 + (2*\vec{v}*\vec{b})*k + (\vec{v}^2 - R^2) = 0$$

Wir setzen nun:

$$a = \vec{b}^2 = b_x^2 + b_y^2 + b_z^2$$
$$b = 2*\vec{v}*\vec{b} = 2*(v_x b_x + v_y b_y + v_z b_z)$$
$$c = \vec{v}^2 - R^2 = v_x^2 + v_y^2 + v_z^2 - R^2$$

und erhalten:

$$a*k^2 + b*k + c = 0$$

Diese lange quadratische Gleichung können wir nach k auflösen. Verwendung findet dabei die Lösungsformel für quadratische Gleichungen:

$$k_{1,2} = \frac{-b \pm \sqrt{b^2 - 4*a*c}}{2*a}$$

Bedenken Sie, daß für die Koeffizienten a, b und c obige Ausdrücke eingesetzt werden müssen! Auch v_x, v_y und v_z sind zusammengesetzte Ausdrücke!

In diesem Stadium sind Sie bereits in der Lage zu entscheiden, ob die Kugel von der Geraden geschnitten wird oder nicht. Das bestimmt die Diskriminante $D = b^2 - 4*a*c$ (also der Term unter der Wurzel). Es gilt:

$D<0$ => Die Gerade schneidet die Kugel nicht
$D=0$ => Die Gerade berührt die Kugel in einem Punkt
$D>0$ => Die Gerade schneidet die Kugel in zwei Punkten

Das resultiert daraus, daß eine Wurzel aus einer negativen Zahl ($D<0$) im reellen Zahlenbereich nicht definiert ist. Ist $D=0$, dann wird auch die Wurzel gleich Null und der gesamte Wurzelterm fällt weg. k ist dann gleich $-b/(2*a)$. Ist $D>0$, dann ist die Wurzel relevant. Es existieren zwei Lösungen.

Der errechnete Parameter k ist ein Maß dafür, welchen Abstand der Schnittpunkt vom Beobachter hat. Im Normalfall entscheidet sich das Programm also für das kleinere k (den näheren Schnittpunkt), sofern es nicht negativ ist. In diesem Fall läge der Schnittpunkt nämlich hinter der Mattscheibe, wäre also strenggenommen gar nicht sichtbar. Wird das größere k einer Kugel verwendet, dann ist das derjenige Schnittpunkt, bei dem der Strahl wieder aus der Kugel heraustritt! Aus k kann bei Bedarf jederzeit der richtige Raumpunkt berechnet werden, was aber in den meisten Fällen nicht nötig ist.

6.2.2 Die ebene Fläche (Parallelogramm)

Die Schnittpunktberechnung an einer ebenen Fläche in Form eines Parallelogrammes (z.B. eines Rechtecks) gehört ebenfalls zu den noch einfacheren Aufgaben. Beliebige Polygone sind da schon eine gehörige Portion schwieriger zu bearbeiten. Unter einem Parallelogramm versteht man eine viereckige Fläche, bei der die jeweils gegenüberliegenden Seiten parallel zueinander verlaufen. Liegen die Seiten sogar in einem rechten Winkel zueinander, so haben wir es mit dem Sonderfall eines Rechtecks zu tun. Sind zusätzlich noch alle Seiten gleich lang, handelt es sich um ein Quadrat.

Das Prinzip ist klar: Es wird zunächst ein Schnittpunkt einer Geraden (des Sichtstrahles) mit einer Ebene errechnet. Dann muß überprüft werden, ob sich der Schnittpunkt innerhalb bestimmter Grenzen befindet.

Wie wäre ein Parallelogramm zu definieren? Bei der Kugel brauchten nur Mittelpunkt und Radius gespeichert werden. Bei der Fläche empfiehlt es sich, zunächst einmal die Parameter für die vektorielle Ebenengleichung festzulegen:

$$P = P_0 + k*\vec{v} + m*\vec{w}$$

Der Punkt P_0 und die Vektoren $\vec{v}$ und $\vec{w}$ müssen also abgespeichert werden. P_0 sollte dabei ein beliebiger Eckpunkt der Fläche sein. Der Vektor $\vec{v}$ hat dann seinen Fuß in P_0 und seine Spitze im benachbarten Eckpunkt. Der Vektor $\vec{w}$ fußt ebenfalls in P_0, verläuft aber zum anderen Punktnachbarn (s. Bild 6.5).

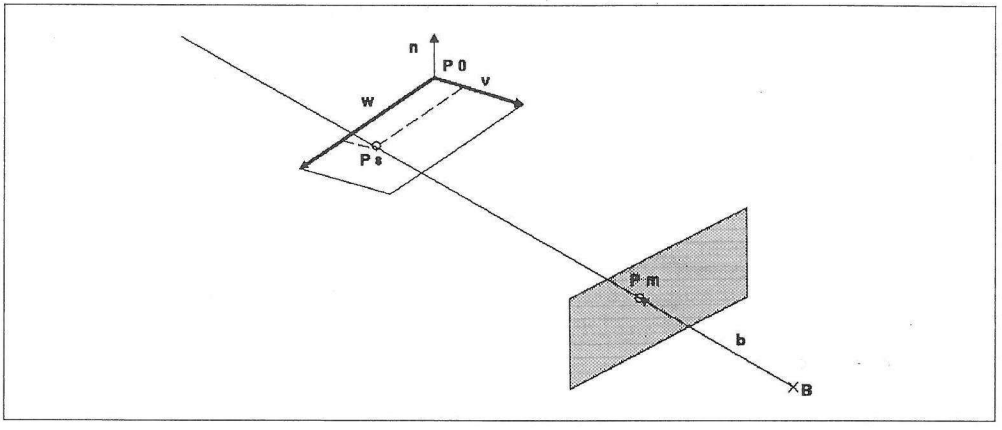


Bild 6.5: Schnittpunktberechnung bei Parallelogrammen

Mit dieser Anordnung kann sehr schnell nachgeprüft werden, ob ein Punkt innerhalb des Parallelogrammes liegt oder nicht. Müssen zur Berechnung des fraglichen Punktes für k oder für m oder gar für beide Werte eingesetzt werden, die größer als 1 oder kleiner als Null sind, dann liegt der Punkt außerhalb der Fläche. Gilt aber:

$0 \leqq k \leqq 1$ und $0 \leqq m \leqq 1$

dann liegt der Punkt im Parallelogramm.

Kommen wir zur eigentlichen Schnittpunktberechnung. Auf Bild 6.5 können Sie wieder alles mitverfolgen. Der spekulative Schnittpunkt soll P_s sein. Er liegt sowohl auf der Linie als auch auf der Fläche. Für ihn gelten also sowohl die Linien- als auch die Flächengleichung:

$P_s = P_m + s * \vec{b}$ und
$P_s = P_0 + k * \vec{v} + m * \vec{w}$

wobei:

P_s	Schnittpunkt?
P_m	Punkt auf Mattscheibe
$\vec{b}$	Geradenrichtungsvektor
s	Geraden-Verlängerungsfaktor
P_0	Eckpunkt des Parallelogramms und Fußpunkt der Vektoren $\vec{v}$ und $\vec{w}$
$\vec{v}$, $\vec{w}$	Ebenenrichtungsvektoren
k, m	Verlängerungsfaktoren für $\vec{v}$, $\vec{w}$

An dieser Stelle sollten Sie sich noch einmal Kapitel 4.3 anschauen. Dort finden Sie die alternative Form der Ebenengleichung, die Normalen-Gleichung und die Formeln zur Umrechnung der einen in die andere. Die Normalen-Gleichung lautet dann in unserem Fall:

$(P_s - P_0) * \vec{n} = 0$

$\vec{n}$ ist die Normale (der auf der Ebene senkrechte Vektor) und kann durch das Vektorprodukt aus $\vec{v}$ und $\vec{w}$ ermittelt werden:

$$\vec{n} = \vec{v} \times \vec{w}$$

Es wird im folgenden viel einfacher sein, mit dieser Form zu rechnen, auch wenn dazu erst ein Vektorprodukt berechnet werden muß.

Setzen wir also die Geradengleichungen in die für die Ebene ein:

$$\begin{aligned}
(P_m + s*\vec{b} - P_0) * \vec{n} &= 0 \quad <=> \\
(P_m - P_0)*\vec{n} + s*\vec{b}*\vec{n} &= 0 \quad <=> \\
s*\vec{b}*\vec{n} &= (P_0 - P_m)*\vec{n}
\end{aligned}$$

An dieser Stelle wird eine Fallunterscheidung notwendig. Jetzt nämlich können wir feststellen, ob ein Schnittpunkt überhaupt existiert:

Fall 1: $\vec{b}*\vec{n} = 0$:

In diesem Fall wird der linke Teil der Gleichung Null:

$$0 = (P_0 - P_m)*\vec{n}$$

Falls dieser Ausdruck wahr ist, dann können Sie beliebige Werte für s einsetzen, die Gerade liegt also auf der Ebene. Es existieren sozusagen unendlich viele Schnittpunkte.

Falls der Ausdruck jedoch falsch ist, dann verläuft die Gerade parallel zur Ebene, es gibt also keinen Schnittpunkt.

Fall 2: $\vec{b}*\vec{n} <> 0$:

Hier dürfen wir wie gewohnt weiterrechnen und erhalten für s:

$$s = \frac{(P_0 - P_m)*\vec{n}}{\vec{b} * \vec{n}}$$

s aber gibt den Verlängerungsfaktor für den Richtungsvektor der Gerade an. Der Schnittpunkt P_s kann errechnet werden.

Uns interessiert nun aber, ob sich P_s nicht nur auf der Ebene, sondern auch auf dem Parallelogramm befindet. Dazu müssen wir die Normalenform der Ebenengleichung wieder rückrechnen in die Form mit k und m. Die Vektoren $\vec{v}$ und $\vec{w}$ sind uns bereits bekannt:

$$P_s = P_0 + k*\vec{v} + m*\vec{w}$$

In Parameterform:

$$\begin{aligned}
p_{sx} &= p_{0x} + k*v_x + m*w_x \\
p_{sy} &= p_{0y} + k*v_y + m*w_y \\
p_{sz} &= p_{0z} + k*v_z + m*w_z
\end{aligned}$$

Nach der genannten Ableitung in Kapitel 4.3. wählen wir die beiden Gleichungen aus, für die gilt (i und j stehen für x, y oder für z):

$$v_i <> 0 \quad \text{und}$$
$$w_j * v_i - w_i * v_j <> 0$$

(z.B.: $v_x <> 0$ und $w_z * v_x - w_x * v_z <> 0$)

Sollte es keine Kombination von zwei Gleichungen geben, die die Bedingungen erfüllen, liegt ein Fehler vor. Die beiden Vektoren $\vec{v}$ und $\vec{w}$ sind nicht linear unabhängig.

Sind die beiden Gleichungen gefunden, die an diesen Bedingungen nicht scheitern (was normalerweise sofort die ersten beiden sind), dann errechnen Sie k und m einfach durch:

$$m = \frac{(p_{sj} - p_{0j}) * v_i - (p_{si} - p_{0i}) * v_j}{w_j * v_i - w_i * v_j}$$
$$k = (p_{si} - p_{0i} - m * w_i) / v_i$$

Testen Sie vor der Berechnung von k, ob m im erlaubten Bereich von 0 bis 1 liegt. Testen Sie ebenfalls, ob sich k in diesem Bereich befindet. Sind alle diese Bedingungen erfüllt, dann schneidet die Gerade das Parallelogramm tatsächlich. Ist s jedoch negativ, dann liegt der Schnittpunkt hinter der Mattscheibe und ist unsichtbar.

Oft werden die Ebenen in einer Welt an ganz bestimmte Stellen verlegt, um den Rechenaufwand zu verringern. So könnte ein Boden beispielsweise in der xy-Ebene mit $z = 0$ liegen. Die obigen Gleichungen vereinfachen sich entsprechend.

Steht erst einmal fest, daß der Strahl die Fläche schneidet, dann können für die Farbwahl sehr einfach noch andere Informationen hinzugezogen werden. So kann auf der Fläche z.B. ein Schriftzug oder gar eine kleine Grafik stehen. Das wäre z.B. dadurch zu schaffen, daß sich jeder Punkt der Fläche in binärer Pixelform (also genauso wie im normalen Bildspeicher) im Speicher befindet. Die Werte für k und m geben dann die Koordinaten des betreffenden Punktes auf der Fläche an. Im Speicher kann dann nachgeschaut werden, was sich an dieser Stelle gerade befindet. Das können natürlich auch Informationen über unterschiedliche Oberflächenbeschaffenheit sein. So könnten Sie beispielsweise die Fläche an den Stellen verspiegeln, an denen sich Grafik befindet etc.

Sie sind dann also in der Lage, eine Grafik oder einen Text mit einem normalen Grafikprogramm zu erstellen und abzuspeichern. Diese Grafik erscheint dann auf Ihrem Ray-Tracing-Bild auf der entsprechenden Fläche gedreht oder vergrößert im Raum. Oft werden auch die auf die Fläche projizierten Muster etc. rechnerisch bestimmt (z.B. Schachbrettmuster etc.)

Die Prozedur ist selbstverständlich auch für andere Objekte wie Kugeln etc. durchführbar – natürlich mit entsprechend mehr Aufwand für die jeweiligen Projektionen der Grafik auf das Objekt.

6.3 Lichtquellen: Reflexion, Glanz, Licht und Schatten

Bevor Sie endlich erfahren, wie man Ray-Tracing überhaupt programmiert, schauen Sie sich doch dieses Kapitel noch einmal an, es lohnt sich. Hier erfahren Sie nämlich, wie die Einflüsse des Lichtes in Ihrer Landschaft realisiert werden können. Hier lernen Sie spiegelnde Objekte, glänzende und matte Flächen sowie die Schattenbildung kennen. Das alles hängt zentral zusammen. Sie werden mit den verschiedenen physikalischen Gesetzen arbeiten, die hier hineinspielen. Sie können sich aber auch Ihre eigenen Naturgesetze schaffen, wie es Ihnen beliebt. Diese Gesetze, Formeln und Ableitungen gelten aber nicht nur für das Ray-Tracing. Sie können sie teilweise sehr leicht auch in andere 3-D-Algorithmen einbauen und damit verblüffende Effekte erzeugen.

Sicherlich können wir hier nicht alle Effekte berücksichtigen. Oft werden wir uns nur mit Näherungsformeln beschäftigen, die uns die Arbeit wesentlich vereinfachen. Denn so viele Dinge spielen bei der Berechnung von Farbe und Lichtintensität eine Rolle. Wir werden z.B. alle Farben gleich behandeln. In Wirklichkeit wird jede Wellenlänge anders reflektiert, gebrochen etc. Jedes Material, sei es Papier, Gold, Silber oder Kunststoff hat seine eigenen Charakteristika, die sich nicht berechnen lassen, sondern nur in aufwendigen Tabellen nachgeschlagen werden können. Nicht umsonst können wir auf Anhieb sagen, ob es sich bei einem Gegenstand um weißes Papier oder um weißes Glas handelt etc. Alle diese wesentlichen Kleinigkeiten werden wir hier nur annähern, um ein ästhetisch möglichst schönes Bild zu erhalten. Viele Werte, Konstanten usw., die im folgenden Verwendung finden, sind Erfahrungswerte, die Sie jederzeit ändern können und auch sollten. Doch nun ans Werk.

6.3.1 Diffuse Reflexion

Sicher hat jeder schon einmal irgendwo von dem zentralen Reflexionsgesetz gehört, das sowohl für Licht wie auch für Billardkugeln gilt: Einfallswinkel gleich Ausfallswinkel. Obwohl dieses idealisierte Gesetz natürlich immer gilt, sieht die Wirklichkeit sehr viel komplizierter aus. Trifft ein Lichtstrahl nämlich auf eine Oberfläche, so hängt es von der Beschaffenheit der Oberfläche ab, auf welche Weise und in welche Richtungen das Licht reflektiert wird. Eine polierte Oberfläche beispielsweise wird sich – mehr oder weniger – exakt an das Reflexionsgesetz halten. Eine rauhe, d.h. matte Oberfläche aber reflektiert den Lichtstrahl in alle möglichen Richtungen. Je matter die Oberfläche, desto breiter gefächert wird der Strahl reflektiert. Ein gewisser Anteil des Lichtes verteilt sich sogar in alle Richtungen. Man spricht bei diesem Anteil auch von der sogenannten diffusen Reflexion im Gegensatz zur spiegelnden Reflexion.

Bild 6.6 veranschaulicht die Verhältnisse. Bei der diffusen Reflexion, die ja in alle Himmelsrichtungen strahlt, spielt der Einfallswinkel der Lichtstrahlen auf eine Fläche nur insofern eine Rolle, als er die Helligkeit des jeweiligen Punktes bestimmt. Je senkrechter das Licht einfällt, desto heller wird auch der Punkt sein. Sie kennen das: Am Äquator fällt das Sonnenlicht fast senkrecht auf die Erde, bei uns leider in einem kleineren Winkel. Das ist auch der Grund dafür, weswegen es am Äquator wärmer ist.

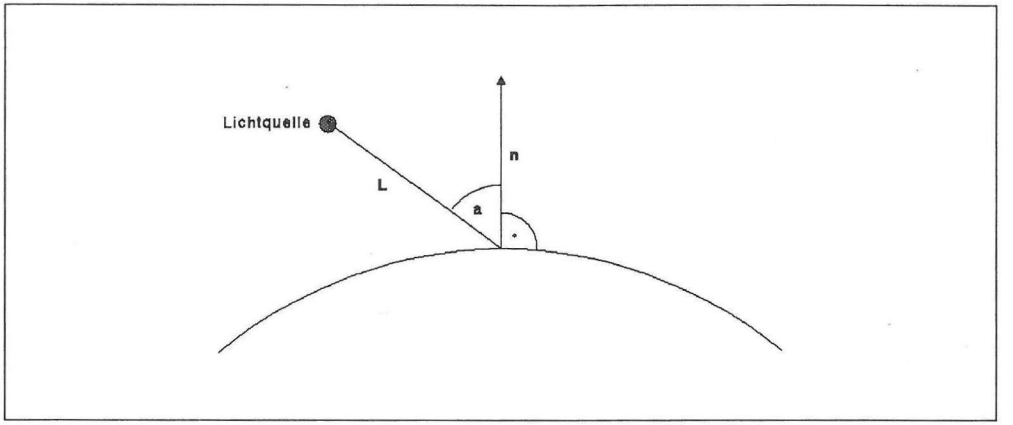


Bild 6.6: Diffuse Reflexion

Neben der einen Lichtquelle haben wir aber noch Licht zu berücksichtigen, das auf andere Gegenstände fällt und von dort wieder auf andere Gegenstände und/oder auf unseren Punkt ausgestrahlt wird. Sie können sich vorstellen, wie aufwendig die Berechnung wäre. Statt dessen nehmen wir einfach eine allgemeine diffuse Hintergrundhelligkeit an, die ebenfalls von unserem Punkt reflektiert wird. Die Lichtintensität, die aus dieser Hintergrundhelligkeit resultiert, addiert man zu dem Licht, das aus der Reflexion resultiert.

Wir kommen dann zu der folgenden Formel für die Intensität des von einem Punkt ausgestrahlten Lichtes:

$$I = I_h * K_{oh} + I_q * K_{od} * \cos(a)$$

wegen:

$$\begin{aligned} \vec{n} * \vec{L} &= |\vec{n}| * |\vec{L}| * \cos(a) && <=> \\ \cos(a) &= (\vec{n} * \vec{L}) \;/\; (|\vec{n}| * |\vec{L}|) && <=> \\ \cos(a) &= \vec{n}/|\vec{n}| * \vec{L}/|\vec{L}| && <=> \\ \cos(a) &= \vec{n}' * \vec{L}' \end{aligned}$$

gilt auch:

$$I = I_h * K_{oh} + I_q * K_{od} * \vec{n}' * \vec{L}'$$

wobei:

I	Intensität des vom Punkt ausgestrahlten Lichtes
I_h	Intensität der Hintergrundhelligkeit
K_{oh}	Reflexionskonstante für die Hintergrundreflexion mit $0 \leqq K_{oh} \leqq 1$ (Objekt- bzw. Farbhelligkeit) (auch Helligkeit oder Farbe des Objektes)
I_q	Intensität der Lichtquelle
K_{od}	Reflexionskonstante für die diffuse Reflexion mit $0 \leqq K_{od} \leqq 1$ (Objekt- bzw. Farbhelligkeit) (auch Helligkeit oder Farbe des Objektes)

a	Winkel zwischen dem Lichtvektor $\vec{L}$ und dem zur Oberfläche normalen (= senkrechten) Vektor $\vec{n}$
$\vec{L}$	Lichtvektor
$\vec{L}'$	Einheits-Lichtvektor (Länge = 1)
$\vec{n}$	Oberflächen-Normalvektor (senkrechter Vektor zur Oberfläche)
$\vec{n}'$	Einheits-Normalvektor (Länge = 1)

Die Einheit für die Intensitäten kann von Ihnen willkürlich festgelegt werden (z.B. 0–100 oder 0–1). Die Konstanten K_{oh} und K_{od} sind von den Reflexionseigenschaften des jeweiligen Objektes abhängig. Reflektiert das Objekt sehr stark, dann bewegt sich der Wert für K_{oh} bzw. K_{od} um 1. Reflektiert es sehr schwach, dann liegt er um 0. Meist sind die beiden Werte identisch ($K_{oh} = K_{od}$). Sie sind ein Maß für die Grundhelligkeit des Objektes. Jedes Objekt absorbiert (verschluckt) einen gewissen Anteil des Lichtes. Bei gleicher Beleuchtung ist ein Objekt heller, ein anderes dunkler.

Absorbiert ein Objekt alle Teile des Lichtes gleichermaßen zu einem bestimmten Anteil (also für uns die Rot-, Grün- und Blauanteile), dann erscheint es bei weißem Licht grau. Absorbiert es (fast) das gesamte Licht, ist es schwarz. Wird jedoch (fast) alles Licht zurückgestrahlt, dann wirkt es weiß.

Manche Objekte absorbieren nur ganz bestimmte Frequenzen des Lichtes. Das Objekt wird farbig. Werden beispielsweise alle Rot- und alle Blau-Anteile absorbiert, dann wird nur Grün zurückgeworfen. Das Objekt ist grün.

Da wir es immer mit farbigen Objekten zu tun haben werden, die ganz bestimmte Grundintensitäten für die Farben Rot, Grün und Blau besitzen, sind die Konstanten K_{oh} und K_{od} die Helligkeitswerte für diese Grundfarben. Jede Grundfarbe hat einen eigenen Helligkeitswert. Die obige Formel muß also für jede der drei Grundfarben berechnet werden. Auf diese Weise sind sogar farbige Lichtquellen möglich, da die Lichtquelle ebenfalls unterschiedliche Intensitätswerte I_q für die drei Farbanteile haben kann. Das Ergebnis I ist dann der Helligkeitswert des Punktes für die jeweilige Grundfarbe, z.B. Grün.

Ein Faktor wurde bisher noch nicht berücksichtigt: die Entfernung. Je mehr Weg das Licht zurücklegt, desto schwächer wird es, da es sich zerstreut. Normalerweise nimmt die Stärke des Lichtes umgekehrt proportional mit dem Quadrat der Entfernung ab, also: $I' = I/d^2$ mit d als Entfernungswert. Passender für die Bildästhetik hat sich jedoch die Formel $I' = I/(d * K_d)$ herausgestellt, mit K_d als willkürliche Konstante. Gemessen wird der Einfachheit halber stets die Entfernung Lichtquelle -> Objektpunkt, obwohl die Entfernung zum Beobachter natürlich auch eine Rolle spielen müßte. Angewandt auf den Lichtquellenteil unserer Formel hieße das:

$$I = I_h * K_{oh} + \frac{I_q * K_{od} * \vec{n}' * \vec{L}'}{d * K_d}$$

6.3.2 Spiegelnde Reflexion

Damit haben wir bereits ein sehr einfaches Helligkeitsmodell für unser Ray-Tracing-Programm zur Verfügung. Wir können aber noch weitergehen, zur spiegelnden Reflexion. »Spiegelnd« heißt hier nicht unbedingt, daß sich alle Objekte scharf auf dem »spiegelnden« Objekt abbilden, was der Fall wäre, wenn der Eintrittswinkel a exakt gleich dem Austrittswinkel wäre (oder anders formuliert: Für eine perfekte spiegelnde Reflexion muß der Winkel b immer gleich 0 sein). Die eintretenden Lichtstrahlen können vielmehr innerhalb eines bestimmten Winkelbereiches reflektiert werden (s. Bild 6.7). Das Maß der Reflexion nimmt mit der Entfernung des Sichtvektors $\vec{S}$ vom exakten Reflexionsstrahl $\vec{R}$ ab. Je größer also der Winkel b zwischen $\vec{R}$ und $\vec{S}$ ist, desto geringer wird die Reflexionsintensität, die durch folgende Formel berechenbar ist:

$$I_s = I_q * w(i,l) * \cos(b)^f$$

wegen:

$$\vec{R} * \vec{S} = |\vec{R}| * |\vec{S}| * \cos(b) \quad <=>$$
$$\cos(b) = (\vec{R} * \vec{S}) / (|\vec{R}| * |\vec{S}|) \quad <=>$$
$$\cos(b) = \vec{R}' * \vec{S}'$$

gilt auch:

$$I_s = I_q * r(a,l) * (\vec{R}' * \vec{S}')^f$$

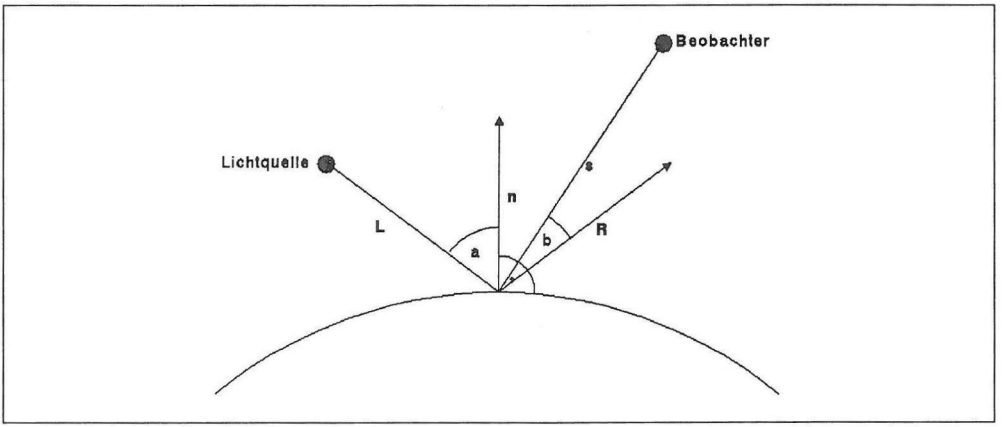


Bild 6.7: Spiegelnde Reflexion

wobei:

I_s	Intensität des Punktes durch spiegelnde Reflexion
I_q	Intensität der Lichtquelle
b	Winkel zwischen dem exakten Reflexionsvektor $\vec{R}$ und dem Sichtvektor $\vec{S}$
$\vec{R}$	Exakter Reflexionsvektor
R'	Einheits-Reflexionsvektor (Länge = 1)

$\vec{S}$ Sichtvektor (Punkt -> Beobachter). Beachten Sie, daß wir oben den Sichtvektor vom Beobachter zum Punkt gerichtet hatten. In dem Falle müssen Sie entsprechend umrechnen ($\vec{S} = -\vec{s}$)

$\vec{S}'$ Einheits-Sichtvektor (Länge = 1)

r(a,l) Reflexionsfunktion in Abhängigkeit vom Einfallswinkel a und der Wellenlänge des Lichtes l mit $0 \leqq r(a,l) \leqq 1$

f Fokussierung

Berücksichtigt wurde hier noch nicht der Abstand zur Lichtquelle. Die Reflexionsfunktion r(a,l) ist völlig materialabhängig. Die verschiedenen Stoffe, Metalle, Kunststoffe etc. besitzen nämlich einen unterschiedlichen Reflexionsgrad, je nachdem, welche Farben eingestrahlt werden, aus welchen Wellenlängen sich also das Licht zusammensetzt. Das heißt, die Materialien verändern das Licht, das sich in ihnen spiegelt, geringfügig. Aus dem gleichen Grunde sind z.B. manche Spiegel nicht farbecht. Zum zweiten hängt der Reflexionsgrad bei den unterschiedlichen Stoffen in unterschiedlicher Weise vom Einfallswinkel ab. Mit dieser recht komplizierten Funktion wollen wir uns hier gar nicht beschäftigen. Näherungsweise werden wir sie einfach durch eine Konstante K_r ersetzen, die je nach Bedarf gewählt werden kann:

$r(a,l) = K_r \qquad \text{mit } 0 \leqq K_r \leqq 1$

Die Variable f ist ebenfalls eine Materialkonstante. Sie gibt an, wie stark konzentriert der Reflexionsbereich ist, d.h. je größer f ist, desto schwächer wird die Reflexion bei gleicher Entfernung des Vektors $\vec{S}$ von $\vec{R}$. Bei $\vec{R}$ ist die Reflexion jedoch immer noch gleich hoch. f gibt also an, wie scharf eine Lichtquelle abgebildet wird. Setzen Sie für f riesige Werte ein, erhalten Sie eine perfekte Spiegelung, da der Ausdruck $\cos(b)^f$ nur noch für $b = 0 \Rightarrow \cos(b) = 1$ nennenswerte Beträge liefert. Metallische Gegenstände beispielsweise werden ein eher hohes f besitzen ($f = 50$ bis $f = 100$), matte Objekte dagegen eher niedrige Werte. Im Zweifelsfall geht hier Probieren, wie bei allen anderen Konstanten, auch über Studieren.

Spannend wird es jetzt, wenn wir die beiden Effekte der diffusen und der spiegelnden Reflexion zusammenpacken:

$$I = I_h * K_{oh} + \frac{I_q}{d * K_d} * [K_{od} * \cos(a) + K_r * \cos(b)^f] \quad <=>$$

$$I = I_h * K_{oh} + \frac{I_q}{d * K_d} * [K_{od} * (\vec{n}' * \vec{L}') + K_r * (\vec{R}' * \vec{S}')^f]$$

Die einzelnen Parameter sollten Ihnen nun schon bekannt sein. Die Konstanten K_{oh}, K_d, K_{od}, K_r und f geben Ihnen einigen Spielraum für Ihre eigenen Experimente. Die Parameter I_h und I_q sind Konstanten der Lichtquellen. d, a und b müssen für diese Formeln berechnet werden, wobei die Ausdrücke cos(a) und cos(b) wie gezeigt ersetzbar sind durch die Skalarprodukte der entsprechenden Einheitsvektoren (ein Einheitsvektor berechnet sich durch den Quotienten aus Vektor und Vektorlänge).

Die angegebenen Formeln beziehen sich lediglich auf eine einzige Lichtquelle. Sind in der Landschaft mehrere Lichtquellen vorhanden, so addieren sich die Einzelintensitäten (den Ausdruck $I_h * K_{oh}$ dürfen Sie allerdings nur einmal verwenden, da er ja die Hintergrundhelligkeit ermittelt).

6.3.3 Berechnung des Reflexionsstrahles

Bei den obigen Ausführungen haben wir bisher stets vorausgesetzt, daß uns die Richtung des Reflexionsvektors $\vec{R}$ bekannt war. Bei der spiegelnden Reflexion und auch bei der totalen Spiegelung muß dieser Vektor jedoch erst einmal berechnet werden. Sehen Sie dazu auf die beigefügte Skizze (Bild 6.8).

Voraussetzung ist die Bekanntheit des Normalvektors $\vec{n}$ in dem Punkt P, in dem sich der Strahl $\vec{L}$ spiegelt. Der Normalvektor ist bekanntlich ein Vektor, der auf der jeweiligen Oberfläche eines Gegenstandes senkrecht steht. Seine Berechnung ist natürlich von Oberfläche zu Oberfläche verschieden (bei einer Kugel ist es einfach der Radiusvektor zum Spiegelpunkt P, bei einer Ebene errechnet er sich aus dem Vektorprodukt zweier auf der Ebene liegenden linear unabhängigen Vektoren). Die Länge von $\vec{n}$ ist hier unerheblich. Bekannt sind also $\vec{L}$ (der Vektor zur oder von der sich spiegelnden Lichtquelle, die Richtung ist gleichgültig) und $\vec{n}$.

Wie Sie aus der Skizze erkennen können, errechnet sich der Reflexionsvektor $\vec{R}$ aus:

$$\vec{R} = -\vec{L} + 2*\vec{L}'$$

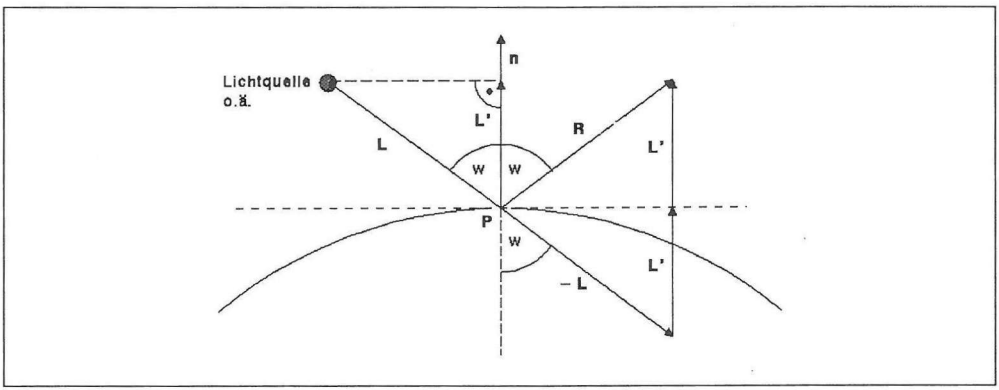


Bild 6.8: Berechnung des Reflexionsvektors

$\vec{L}'$ ist dabei derjenige Vektor, der entsteht, wenn man $\vec{L}$ parallel auf $\vec{n}$ projiziert. Seine Bestimmung ist nur ein wenig aufwendiger. $\vec{L}'$ liegt also auf $\vec{n}$, ist demnach nur eine Verlängerung oder Verkürzung von $\vec{n}$:

$$\vec{L}' = k * \vec{n}$$

Der Verlängerungsfaktor k ist die Unbekannte, nach der wir suchen müssen. In der Zeichnung erkennen Sie ein rechtwinkliges Dreieck mit den Seiten $|\vec{L}|$ (Hypotenuse) und $|\vec{L}'|$ (Kathete) und dem Winkel w zwischen diesen Seiten. Gemäß der Definition des Cosinus (s. Anhang) gilt:

$$\cos(w) = |\vec{L}'| / |\vec{L}| \quad <=>$$
$$|\vec{L}'| = |\vec{L}| * \cos(w)$$

Für $\vec{L}'$ setzen Sie dann bitte die obige Gleichung ein:

$$|k*\vec{n}| = |\vec{L}| * \cos(w) \quad <=>$$
$$k*|\vec{n}| = |\vec{L}| * \cos(w) \quad <=>$$
$$k = |\vec{L}|/|\vec{n}| * \cos(w)$$

Schön und gut. Jetzt kennen wir eine Formel für k. Der Winkel w bzw. sein Cosinus cos(w) ist uns aber ebenfalls nicht bekannt. Da springt das Skalarprodukt zweier Vektoren helfend ein:

$$\vec{L}*\vec{n} = |\vec{L}|*|\vec{n}|*\cos(w) \quad <=>$$

$$\cos(w) = \frac{\vec{L}*\vec{n}}{|\vec{L}|*|\vec{n}|}$$

Diesen Ausdruck, bei dem nun tatsächlich alles bekannt ist, setzen Sie natürlich bitte schleunigst für cos(w) ein:

$$k = \frac{|\vec{L}| * \vec{L}*\vec{n}}{|\vec{n}| * |\vec{L}|*|\vec{n}|} \quad <=>$$

$$k = \frac{\vec{L}*\vec{n}}{|\vec{n}|^2}$$

und Sie bekommen für $\vec{L}'$:

$$L' = \frac{\vec{L}*\vec{n}}{|\vec{n}|^2} * \vec{n}$$

Versuchen Sie nicht, $\vec{n}$ auf den Bruch zu heben und daraus $|\vec{n}|^2$ zu machen, um das dann auch noch mit dem Nenner zu kürzen. Vergessen Sie nicht, daß die Reihenfolge bei der Skalarmultiplikation eingehalten werden muß. Es gilt nämlich normalerweise nicht: $(\vec{a}*\vec{b})*\vec{c} = \vec{a}*(\vec{b}*\vec{c})$! Schauen Sie dazu vielleicht noch einmal im Anhang nach.

6.3.4 Schattenbildung

Die Berücksichtigung von Schatten, den Körper auf andere Objekte werfen, ist normalerweise dem Problem der verdeckten Flächen sehr ähnlich. Schließlich verdecken hier Flächen andere Flächen vor der Lichtquelle genauso wie vor dem Beobachter. Entsprechend kompliziert können die Lösungen sein, wenn mit mehreren Objekten hantiert wird. Beim Ray-Tracing allerdings ist auch hier die Handhabung ein Kinderspiel und kaum ein getrenntes Kapitel wert.

Das Prinzip ist schnell erklärt: Angenommen, Sie haben bereits den Schnittpunkt des Sichtstrahles mit einem Objekt in der Welt berechnet. Bilden Sie nun einen zweiten Strahl von diesem Punkt zur Lichtquelle (oder zu den Lichtquellen). Mit diesem Strahl verfahren Sie genauso wie mit dem Sichtstrahl. Berechnen Sie die Schnittpunkte dieser Gerade mit allen anderen Objekten. Hier brauchen Sie allerdings nur zu prüfen, ob es mindestens ein Objekt gibt, das der Strahl schneidet. Gibt es tatsächlich einen Schnittpunkt mit einem anderen Objekt als mit der angesteuerten Lichtquelle selbst, dann verdeckt dieses Objekt die Lichtquelle; der Punkt, dessen Intensität ermittelt werden soll, steht im Schatten. Er könnte höchstens noch von anderen Licht-

quellen angestrahlt werden. Berücksichtigt wurde hierbei nicht, daß ein Objekt auch spiegeln könnte, das Licht von einer anderen Lichtquelle also auf den Punkt strahlen könnte. Eine solche Betrachtung, die in komplizierten professionellen Programmen durchaus Berücksichtigung findet, werden wir hier der Einfachheit halber nicht vornehmen.

Normalerweise wäre dieser Punkt schwarz. Da jedoch eine Hintergrundhelligkeit existiert, erhält er trotzdem eine gewisse Intensität. Liegt ein Punkt bezüglich einer Lichtquelle im Schatten, dann fällt der gesamte Ausdruck für diese Lichtquelle, wie wir ihn im vorigen Kapitel abgeleitet haben, fort bzw. wird Null. Was bleibt, ist die besagte Hintergrundhelligkeit und das Licht von eventuell anderen Lichtquellen.

Die Tatsache, daß ein Objekt andere Objekte vor einer Lichtquelle abschirmen kann, können wir uns zunutze machen, indem wir z.B. an einer oder mehreren Seiten der Lichtquelle undurchsichtige Ebenen anbringen, die den Strahl des Lichtes eingrenzen. Dadurch wären beispielsweise Punktstrahler zu simulieren, die nur ganz bestimmte Bereiche eines Zimmers oder einer Landschaft beleuchten.

6.4 Das Programm

Endlich sind wir soweit. Endlich hat die graue Theorie ein Ende. Endlich können wir alles Gelernte direkt am Bildschirm ausprobieren. Vorhang auf für das sagenumwobene Ray-Tracing-Programm, natürlich in C. Bitte schön!

```
/**************************************/
/**                                  **/
/**       Computer und Realität      **/
/**                                  **/
/**          3-D-RAY-TRACING         **/
/**                                  **/
/**                mit               **/
/**      mehreren Lichtquellen,      **/
/**     diffuser und spiegelnder     **/
/**           Reflexion,             **/
/**        Totalverspiegelung        **/
/**       und  Schattenbildung       **/
/**                                  **/
/**           Organisation:          **/
/**        objektorientiertes        **/
/**           Volumenmodell          **/
/**                                  **/
/**************************************/
```

```
#include <exec/types.h>
#include <intuition/intuition.h>
#include <libraries/mathffp.h>

/* Funktionsdeklarationen (nur für Aztek-Compiler): */
/* wahlweise auch: #include <functions.h>          */

/*****************************************************/
VOID                 ClearScreen();
VOID                 Close();
VOID                 Exit();
struct Message *     GetMsg();
UWORD                GetRGB4();
LONG                 IoErr();
VOID                 Move();
struct FileHandle *  Open();
struct Library *     OpenLibrary();
struct Screen *      OpenScreen();
struct Window *      OpenWindow();
VOID                 ReplyMsg();
VOID                 SetAPen();
VOID                 SetRGB4();
LONG                 Wait();
LONG                 Write();
VOID                 WritePixel();

/*****************************************************/

#define WURZEL SPSqrt
#define SIN    SPSin
#define COS    SPCos
#define TAN    SPTan
#define LOG    SPLog
#define EXP    SPExp

#define MODE_NEWFILE 1006L

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
LONG   *MathBase;
LONG   *MathTransBase;
```

```
/* Hardware-Konstanten: */

#define  ANZ_FARBEN  64            /* Anzahl der versch. Farben   */
#define  ANZ_EBENEN  6             /* Anzahl der Farbebenen       */
#define  ANZ_INT     16            /* Anzahl d. Intensitätsstufen */
#define  X_AUFL      320           /* Hardware-x-Auflösung        */
#define  Y_AUFL      256           /* Hardware-y-Auflösung        */
#define  G_MOD    EXTRA_HALFBRITE  /* Grafik-Modus                */
#define  ANZ_PAL_FAB 32            /* Anzahl der Palettenreg.     */

/* Hier die Einstellungen für 32 Farben:   */
/* (Zusätzlich sollten dann noch die       */
/*  Palettenfarben (s.u.) geändert werden) */

/* #define  ANZ_FARBEN  32     */ /* Anzahl der versch. Farben   */
/* #define  ANZ_EBENEN  5      */ /* Anzahl der Farbebenen       */
/* #define  ANZ_INT     16     */ /* Anzahl d. Intensitätsstufen */
/* #define  X_AUFL      320    */ /* Hardware-x-Auflösung        */
/* #define  Y_AUFL      256    */ /* Hardware-y-Auflösung        */
/* #define  G_MOD       NULL   */ /* Grafik-Modus                */
/* oder: */
/* #define  G_MOD       LACE   */ /* für den Interlace-Modus     */
/* #define  ANZ_PAL_FAB 32     */ /* Anzahl der Palettenreg.     */


/* Struktur für neuen Bildschirm initialisieren: */
/*************************************************/

struct NewScreen NeuerBildschirm =
{
   0,                   /* x-Koord. linke obere    */
                        /* Ecke (immer 0)          */
   0,                   /* y-Koord. linke obere    */
                        /* Ecke                    */
   X_AUFL,              /* Bildschirmbreite        */
   Y_AUFL,              /* Bildschirmhöhe          */
   ANZ_EBENEN,          /* Bildebenenzahl          */
   32,                  /* Farbe der Details       */
   1,                   /* Farbe der Flächen       */
   G_MOD,               /* Grafikmodus             */
   CUSTOMSCREEN,        /* Bildschirmtyp           */
   NULL,                /* kein neuer Font         */
   "Ray-Tracing-Demo",  /* Text im Bildschirm-Kopf */
   NULL,                /* unbenutzt, immer NULL   */
   NULL,                /* keine eigene BitMap     */
};
```

```
/* Struktur für neues Fenster initialisieren:  */
/*********************************************/

struct NewWindow NeuesFenster =
{
   0,                   /* x-Koord. linke ob. Ecke */
   10,                  /* y-Koord. linke ob. Ecke */
   X_AUFL,              /* Fensterbreite           */
   Y_AUFL-10,           /* Fensterhöhe             */
   32,                  /* Farbe der Details       */
   1,                   /* Farbe der Flächen       */
   MOUSEBUTTONS |       /* Rückmeldg. b. Mausknopf */
    RAWKEY,             /* und Taste               */
   ACTIVATE |           /* Fensterelemente und -typ*/
    BORDERLESS|         /* wählen: Randlos,        */
    NOCAREREFRESH |     /* keine Refresh-Meldungen */
    RMBTRAP,            /* keine Menüoperationen   */
   NULL,                /* keine eigenen Gadgets   */
   NULL,                /* CheckMark               */
   NULL,                /* Text im Fenster-Kopf    */
   0,                   /* Adresse Screen-Struktur */
                        /* Muß innerhalb des       */
                        /* Programmes nach dem     */
                        /* Öffnen eines Screens    */
                        /* initialisiert werden    */
   NULL,                /* kein SuperBitmap-Fenster*/
   X_AUFL,              /* Mindestbreite           */
   Y_AUFL-10,           /* Mindesthöhe             */
   X_AUFL,              /* Maximalbreite           */
   Y_AUFL,              /* Maximalhöhe             */
   CUSTOMSCREEN,        /* Bildschirmtyp           */
};


struct Screen *Bildschirm;
struct Window *Fenster;
struct RastPort *RastPort;
struct IntuiMessage *Message;


VOID main()
{
 LONG do_program();
```

```
LONG fehler;          /* Für die Typ-Abkürzungen s. unter */
                      /* Exec-Include-File: types.h       */

/* Öffnen der Intuition-, Graphics-, Mathe-Bibliotheken */

printf("Ray-Tracing / C-Demo\n");

IntuitionBase =
(struct IntuitionBase *)OpenLibrary("intuition.library",0L);
if (IntuitionBase == NULL)            fehler = 1;
else
{
 GfxBase =
 (struct GfxBase *)OpenLibrary("graphics.library",0L);
 if (GfxBase == NULL)                 fehler = 2;
 else
 {
  MathBase =
  (LONG *)OpenLibrary("mathffp.library",0L);
  if (MathBase == NULL)               fehler = 3;
  else
  {
   MathTransBase =
   (LONG *)OpenLibrary("mathtrans.library", 0L);
   if (MathTransBase == NULL)
   {
      printf("Keine MathTrans-Library vorhanden!\n");
      fehler = 4;
   }
   else
   {
    /* Bildschirm öffnen:   */
    Bildschirm =
    (struct Screen *)OpenScreen(&NeuerBildschirm);
    if (Bildschirm == NULL)           fehler = 5;
    else
    {
     /* Fenster öffnen:     */
     /* Adresse der Screen-Struktur nachtragen:   */
     NeuesFenster.Screen = Bildschirm;
     Fenster =
     (struct Window *)OpenWindow(&NeuesFenster);
     if (Fenster == NULL)             fehler = 6;
     else
     {
```

```
        RastPort = Fenster->RPort; /* RastPort merken        */

        Move(RastPort, 0L, 0L);
        ClearScreen(RastPort);     /* Bildschirm löschen     */

        fehler = do_program();     /* eigentliches Pro-      */
                                   /* gramm aufrufen         */

        CloseWindow(Fenster);      /* Fenster schließen      */
       }
       CloseScreen(Bildschirm);    /* Screen schließen       */
      }
      CloseLibrary(MathTransBase); /* Mathe-Library close    */
     }
     CloseLibrary(MathBase);      /* Mathe-Library close    */
    }
    CloseLibrary(GfxBase);        /* Graphics-Lib. close    */
   }
   CloseLibrary(IntuitionBase);   /* Intuition-Lib. close   */
  }
  Exit(fehler);                   /* Ausgang m. Fehlercode */
}


/******************/
/* Hauptprogramm */
/******************/

/* Defines und Makros: */

/* Ermitteln des Maximums/Minimums zweier Werte: */
#define MAX(a,b)  ( (a) > (b) ? (a) : (b) )
#define MIN(a,b)  ( (a) < (b) ? (a) : (b) )

/* Ermitteln des Absolutwertes einer Zahl: */
#define ABS(a) ( (a) < 0  ?  -(a) : (a) )

/* Ermitteln der Potenz a^b:     */
/* nach p=a^b <=> p=e^(b*ln(a)) */
#define POTENZ(a,b) (EXP( (b)*LOG(a) ))

/* Vektoroperationen als Macros:    */
/* a, b, c müssen Vektornamen sein! */
```

```
/* Zuweisen eines Vektors: a=b: */
#define VEKLET(a,b) {a[0]=(b[0]); a[1]=(b[1]); a[2]=(b[2]);}

/* Zuweisen eines Farbarrays: a=b */
#define COLLET  VEKLET

/* Vektoraddition c=a+b zweier Vektoren: */
#define VEKADD(c,a,b) {c[0] = (a[0]) + (b[0]);\
                       c[1] = (a[1]) + (b[1]);\
                       c[2] = (a[2]) + (b[2]);}

/* Vektorsubtraktion c=a-b zweier Vektoren: */
#define VEKSUB(c,a,b) {c[0] = (a[0]) - (b[0]);\
                       c[1] = (a[1]) - (b[1]);\
                       c[2] = (a[2]) - (b[2]);}

/* Multiplikation b=z*a eines Vektors mit einer Zahl: */
#define MULVEK(b,z,a) {b[0] = (z) * (a[0]);\
                       b[1] = (z) * (a[1]);\
                       b[2] = (z) * (a[2]);}

/* Skalarprodukt a*b zweier Vektoren: */
#define SKAPRD(a,b) ( (a[0])*(b[0]) +\
                      (a[1])*(b[1]) +\
                      (a[2])*(b[2]) )

/* Vektorprodukt c=axb zweier Vektoren: */
#define VEKPRD(c,a,b) {c[0] = (a[1])*(b[2]) - (a[2])*(b[1]);\
                       c[1] = (a[2])*(b[0]) - (a[0])*(b[2]);\
                       c[2] = (a[0])*(b[1]) - (a[1])*(b[0]);}

/* Ermittlung der Geradengleichung P = P0 + s*a */
/* in der Struktur struct gerade g              */
/* aus zwei Punkten b und c: P0=b, a=b-c        */
/* Damit ist die Spitze des Richtungsvektors    */
/* gleichzeitig Fixpunkt der Gerade             */

#define GET_GERADE(g,b,c) {VEKLET(g.p0,b); VEKSUB(g.a,b,c);}


/* Berechnung eines Punktes aus der */
/* vektoriellen Geradengleichung:   */
/* P = P0 + s*a                     */
```

```
#define GERADEN_PKT(P,P0,s,a) {P[0] = (P0[0]) + (s)*(a[0]);\
                               P[1] = (P0[1]) + (s)*(a[1]);\
                               P[2] = (P0[2]) + (s)*(a[2]);}


/* Berechnung der Länge eines Vektors a: */
#define VEK_LAENGE(a) (WURZEL(SKAPRD(a,a)))


/* Objektkennziffern (Typen):*/
#define HINTERGRUND  0
#define LICHTQUELLE  1
#define KUGEL        2
#define FLAECHE      3

/* Lichtoperationstypen: */
#define NORMAL       0
#define VERSPIEGELT  1
#define DURCHSICHTIG 2

/* sonstige Parameter: */
#define REKUR_MAX    7         /* Max. Zahl Rekursionen */
#define UNENDLICH    1.0e10    /* Ferne                 */
#define SCHWARZ      0.10      /* Lichtgrenze           */
#define MINI_S       0.0001    /* kleinstes erlaubtes s */
                               /* (fast 0 wegen Rechen- */
                               /*  ungenauigkeit!)      */


/* Globale Variablen und Strukturen:  */
/**************************************/

/* Einzelner Objekt-Punkt: */
struct punkt
{
   FLOAT    pos[3];         /* Position des Punktes     */
   FLOAT    s;              /* s-Wert f. Geradengleichg.*/
   FLOAT    farbe[3];       /* Farbe                    */
   FLOAT    intens_b[3];    /* RGB-Intensitätsbeiwert   */
   FLOAT    user0[3];       /* sonstige Werte           */
   FLOAT    user1;
   FLOAT    user2;
};
```

```
/* Geradenparameter nach P = P0 + s*a: */
struct gerade
{
   FLOAT    p0[3];          /* Fixpunkt P0              */
   FLOAT    a[3];           /* Richtungsvektor a        */
};

/* Ebenenparameter nach P = P0 + s*a + t*b: */
struct ebene
{
   FLOAT    p0[3];          /* Fixpunkt P0              */
   FLOAT    a[3];           /* Richtungsvektor a        */
   FLOAT    b[3];           /* Richtungsvektor b        */
   BOOL     n_valid;        /* Flag: TRUE => Normalen-  */
                            /*       vektor n vorhanden */
                            /*       FALSE => n ist     */
                            /*       nicht vorhanden    */
   FLOAT    n[3];           /* Normalenvektor v. Ebene  */
};

/* Materialkonstanten für ein Objekt: */
struct material
{
   FLOAT    farbe[3];       /* RGB-Farbe des Objektes   */
   FLOAT    farbe2[3];      /* evt. Zweitfarbe          */
   FLOAT    farbe3[3];      /* evt. Drittfarbe          */
                            /* übernehmen die Rolle von */
                            /* Konstanten Koh und Kod   */
   FLOAT    spieg_ref[3];   /* Konstante zur spiegeln-  */
                            /* den Reflexion Kr         */
   FLOAT    fokus_ref[3];   /* Fokussierung der spie-   */
                            /* gelnden Reflexion f      */
   FLOAT    spiegel_int[3];/* Intensität des gespie-    */
                            /* gelten Lichtes           */
   BYTE     licht_typ;      /* Lichtoperationstyp:      */
                            /* =0: Normal               */
                            /* =1: Total Verspiegelt    */
                            /* =2: durchsichtig         */
   BYTE     oberf_typ;      /* Oberflächentyp           */
                            /* =0: Normal, einfarbig    */
                            /* =1: z.B. Karos           */
                            /* =2: z.B. Muster          */
   union                    /* Individuelle Parameter   */
   {
      struct
```

```
      {
         WORD anz_k,anz_m; /* Anzahl Muster in k/m-Rtg.*/
         WORD muster[16];  /* Musterdefinition         */
      } muster;
      struct
      {                    /* oder Integer-Tripel      */
         WORD user1[3];
         WORD user2[3];
         WORD user3[3];
      } int_user;
      struct
      {                    /* oder Float-Tripel        */
         FLOAT user1[3];
         FLOAT user2[3];
         FLOAT user3[3];
      } float_user;
   } user;
};

/* Hintergrund im "Unendlichen": */
struct hintergrund
{
   WORD     typ;           /* Typ für Hintergrund: 0   */
   FLOAT    unendlich;     /* Wert für die Position    */
   FLOAT    farbe1[3];     /* Farbe 1                  */
   FLOAT    farbe2[3];     /* Farbe 2                  */
};

/* Lichtquelle: */
struct lichtquelle
{
   WORD     typ;           /* Typ für Lichtquelle: 1   */
   FLOAT    pos[3];        /* Position der Lichtquelle */
   FLOAT    radius;        /* Radius                   */
   FLOAT    farbe[3];      /* Farbe des Lichts         */
   FLOAT    dist_konst;    /* Distanzkonstante Kd      */
};

/* Kugel: */
struct kugel
{
   WORD     typ;           /* Typ für Kugel: 2         */
   FLOAT    pos[3];        /* Position Mittelpunkt     */
   FLOAT    radius;        /* Radius                   */
   struct material material; /* Materialkonstanten     */
};
```

```
/* Fläche (Parallelogramm): */
struct flaeche
{
   WORD     typ;           /* Typ für Fläche: 3        */
   FLOAT    p1[3],p2[3],   /* drei Eckpunkte           */
            p3[3];
   struct ebene ebene;     /* Ebenenparameter mit      */
                           /* Normalenvektor!          */
   struct material material; /* Materialkonstanten     */
};

/* Beobachter: */
struct beobachter
{
   FLOAT    pos[3];        /* Position des Beobachters */
   FLOAT    blickp[3];     /* Punkt, auf den er schaut */
   FLOAT    blickvek[3];   /* alternativ: Blickvektor  */
                           /* ist der Blickvektor 0,   */
                           /* dann wird der Blickpunkt */
                           /* verwendet                */
   FLOAT    mattsch;       /* Abstand zur Mattscheibe  */
   FLOAT    x_punktgr;     /* Punktbreite              */
   FLOAT    y_punktgr;     /* Punkthöhe                */
   LONG     x_aufl;        /* x-Auflösung              */
   LONG     y_aufl;        /* y-Auflösung              */
   FLOAT    x_winkel;      /* horizontaler Sichtwinkel */

   /* später zu berechnende Variablen: */
   FLOAT    blickv[3];     /* Blickvektor mit Länge    */
                           /* Beobachter -> Mattscheibe*/
   FLOAT    x_step[3];     /* x-Schrittweitenvektor    */
   FLOAT    y_step[3];     /* y-Schrittweitenvektor    */
};

/* Die ganze Welt: */
struct welt
{
   WORD     anz_obj;          /* Anzahl aller Objekte  */
   WORD     anz_kugeln;       /* Anzahl aller Kugeln   */
   struct kugel *kugeln;      /* Alle Kugeln           */
   WORD     anz_flaech;       /* Anzahl aller Flächen  */
   struct flaeche *flaech;    /* Alle Flächen          */
   WORD     anz_licht;        /* Anzahl aller Lichter  */
   struct lichtquelle *licht; /* Alle Lichtquellen     */
   WORD     anz_hinten;       /* Anzahl Hintergründe=1 */
```

```
   struct hintergrund *hinten;/* Die Weite, das Nichts */
   FLOAT    hintlicht[3];  /* Untergrundbeleuchtung Ih */
};


/* Globale Variablen: */

FLOAT    sch_p[3];          /* aktueller Schnittpunkt   */
FLOAT    sch_m, sch_k;      /* Verlängerungsfakt. Ebene */
WORD     *z_objekt;         /* Zwischenvar. f. Objekt   */
WORD     rekur_zaehl;       /* Rekursionszähler         */

/* RAW-Tastencodes definieren: */
#define ESC        0x45
#define TAS_S      0x21


LONG do_program()
{
   VOID     init_welt(),
            init_farben(),
            get_konst(),
            get_punkt(),
            ray_trace(),
            plot(),
            save_iff();
   USHORT   taste();


   USHORT code = 0;

   struct punkt      punkt;  /* Objektpunkt          */
   struct gerade     strahl; /* Sichtstrahl          */
   struct welt       welt;   /* Weltstruktur         */
   struct beobachter beo;    /* Beobachterstruktur   */

   LONG     xa, ya;          /* halbe Auflösungen    */
   REGISTER LONG x, y;       /* Schleifenindizes     */
   FLOAT    akt_punkt[3];    /* Mattscheibenpunkt    */


   init_welt(&welt, &beo);   /* Datenstrukturen initialisieren */

   init_farben();            /* Initialisiere Farbpalette      */
```

```
/* Schrittweiten und Sichtvektor berechnen: */
get_konst(&beo);

rekur_zaehl = 0;          /* Rekursionszähler auf Null      */


/* Hauptschleifen: */
/*******************/

for (y = -(ya = beo.y_aufl/2); y<=ya & code!=ESC; y++)
{
   for (x = -(xa = beo.x_aufl/2); x<=xa & code!=ESC; x++)
   {
      /* Berechne aktuellen Mattscheibenpunkt: */
      get_punkt(&beo, x, y, &akt_punkt[0]);

      /* Ermittle Geradengleichung des Sichtstrahles */
      /* mit Mattscheibenpunkt als Fixpunkt:         */
      GET_GERADE(strahl, akt_punkt, beo.pos);

      punkt.intens_b[0] =  /* Intensitätsbeizahl initialis.  */
      punkt.intens_b[1] =
      punkt.intens_b[2] = 1.0;
      punkt.farbe[0] =     /* Punkt-Farbwert auf Null setzen */
      punkt.farbe[1] =
      punkt.farbe[2] = 0.0;

      /* Hier beginnt das Ray-Tracing (Strahlverfolgung) */
      /* Ermittlung der Punktfarbe:                      */
      ray_trace(&welt, &strahl, &punkt);

      /* Zeichne Punkt in der ermittelten Farbe: */
      plot(xa+x, ya-y, &punkt);

      /* Auf Taste testen: */
      code = taste();
   }
}
/* Auf ESC warten: */
while (code != ESC)
{
   /* Auf Message warten: */
   Wait(1L << Fenster->User-Port->mp_SigBit);
   code = taste();
```

```
      if (code == TAS_S)  /* bei "s":                          */
         save_iff();      /* Bild im IFF-Format abspeichern */
   }

   return ((LONG)TRUE);
}


/* Auf Taste testen: */
USHORT taste()

{
   ULONG  class;             /* Speicher für die Intuition- */
   USHORT code, qualifier;   /* Message-Struktur            */
   APTR   address;
   SHORT  mouse_x, mouse_y;

   code = 0;

   /* Meldungen abarbeiten: */
   /*************************/
   while (code != ESC  &&
          (Message =

            (struct IntuiMessage *)GetMsg(Fenster->User-Port)) )
   {
      /* Daten aus Message-Struktur lesen: */
      class     = Message->Class;
      code      = Message->Code;
      qualifier = Message->Qualifier;
      address   = Message->IAddress;
      mouse_x   = Message->MouseX;
      mouse_y   = Message->MouseY;
      ReplyMsg(Message);   /* Message zurückgeben */
   }
   return (code);
}

/* Speichere gesamtes Bild im IFF-Format */
/* unter dem Namen: "Ray_Trace.pic" in   */
/* das aktuelle Directory                */
/* Vorsicht! Eine bereits vorhandene     */
/* Datei mit dem gleichen Namen wird     */
/* überschrieben!!!                      */
```

```
VOID save_iff()

{
   LONG     i, row;
   UWORD    anz_farben;
   UWORD    farbe;
   ULONG    laenge;
   LONG     longword;        /* Datenspeicher                  */
   WORD     word;
   BYTE     byte;
   BYTE     *BitPlanes[8];   /* Zeiger auf maximal 8 Bitplanes */

   struct BitMap *BitMap;
   struct ColorMap *ColorMap;
   struct FileHandle *FileHandle;

   /* File öffnen: */
   FileHandle = Open("Ray_Trace.pic",(LONG)MODE_NEWFILE);

   if (FileHandle == 0)
   {
      printf("Fehler beim Öffnen der Bilddatei! Fehlernummer: %ld\n"
                                                        ,IoErr());
      return;
   }

   BitMap = RastPort->BitMap;   /* Pointer auf BitMap-Struktur */

   /* Zeiger auf BitPlanes ermitteln: */
   for (i=0; i<BitMap->Depth; i++)
      BitPlanes[i] = (BYTE *)BitMap->Planes[i] - BitMap->BytesPerRow;

   ColorMap = Bildschirm->ViewPort.ColorMap;
   anz_farben = ColorMap->Count;

   /* IFF-Header: */
   Write(FileHandle, "FORM", 4L);      /* "FORM" als Einleiter */
   laenge = 60 + 3*anz_farben +
            BitMap->BytesPerRow * BitMap->Rows * BitMap->Depth;
   Write(FileHandle, &laenge, 4L);     /* Länge des Files      */
   Write(FileHandle, "ILBM", 4L);      /* "ILBM" als Einleiter */

   /* Bitmap-Header: */
   Write(FileHandle, "BMHD", 4L);      /* "BMHD" als Einleiter */
   laenge = 20;
```

```
Write(FileHandle, &laenge, 4L);      /* Länge des BMHD       */
word = BitMap->BytesPerRow * 8;      /* Punkte pro Zeile     */
Write(FileHandle, &word, 2L);
Write(FileHandle, &BitMap->Rows, 2L); /* Punkte pro Spalte */
longword = byte = 0;
Write(FileHandle, &longword, 4L);
Write(FileHandle, &BitMap->Depth, 1L); /* Plane-Tiefe      */
Write(FileHandle, &byte, 1L);
Write(FileHandle, &longword, 4L);
byte = 10;                           /* x-/y-Verhältnis      */
Write(FileHandle, &byte, 1L);
byte = 11;
Write(FileHandle, &byte, 1L);
Write(FileHandle, &word, 2L);
Write(FileHandle, &BitMap->Rows, 2L);

/* View-Modes: */
Write(FileHandle, "CAMG", 4L);       /* "CAMG" als Einleiter */
longword = 4L;                       /* Länge                */
Write(FileHandle, &longword, 4L);
longword = Bildschirm->ViewPort.Modes;
Write(FileHandle, &longword, 4L);

/* Color-Map: */
Write(FileHandle, "CMAP", 4L);       /* "CMAP" als Einleiter */
longword = anz_farben * 3;       /* Länge der Farbdefinition */
Write(FileHandle, &longword, 4L);
/* Farben speichern: */
for (i=0; i<anz_farben; i++)
{
   farbe = GetRGB4(ColorMap, i);
   byte = ((farbe & 0xf00) >> 4);    /* Rot-Anteil * 16      */
   Write(FileHandle, &byte, 1L);
   byte = (farbe & 0x0f0);           /* Grün-Anteil * 16     */
   Write(FileHandle, &byte, 1L);
   byte = (farbe & 0x00f) << 4;      /* Blau-Anteil * 16     */
   Write(FileHandle, &byte, 1L);
}

/* Grafikdaten: */
Write(FileHandle, "BODY", 4L);       /* "BODY" als Einleiter */
laenge = BitMap->BytesPerRow * BitMap->Rows * BitMap->Depth;
Write(FileHandle, &laenge, 4L);
```

```
   for (row=0; row<BitMap->Rows; row++)
      for (i=0; i<BitMap->Depth; i++)
         Write(FileHandle, BitPlanes[i] += BitMap->BytesPerRow,
                           (LONG)BitMap->BytesPerRow);

   Close(FileHandle);       /* Datei schließen */
}


/* Berechne Schrittweiten für Mattscheibe */
/* und den Sichtvektor s:                 */
VOID get_konst(beo)
struct beobachter *beo;  /* Zeiger auf Beobachterstruktur */
{
   FLOAT    zahl;
   FLOAT    l_xe;
   FLOAT    vektor[3];
   REGISTER WORD x_z, z_x, y_z, z_y;

   /* Ermittlung des Sichtvektors s zur Mattscheibe nach: */
   /* s = |s|/|Z-B| * (Z-B)                               */
   /* Speicherung von s in Beobachter-Struktur            */
   /*****************************************************/

   /* Verlängerten Blickvektor ermitteln. */
   /* Falls schon angegeben, übertragen:  */
   if (beo->blickvek[0] || beo->blickvek[1] || beo->blickvek[2])
   {
      VEKLET(vektor, beo->blickvek);          /* Blickvektor */
   }
   else
   {
      VEKSUB(vektor, beo->blickp, beo->pos);  /* Z-B         */
   }

   zahl = beo->mattsch / VEK_LAENGE(vektor);  /* Quotient    */
   MULVEK(beo->blickv, zahl, vektor);         /* s           */


   /* Berechnung der Schrittweiten: */
   /*********************************/

   /* Länge des Einheitsvektors |xe| berechnen: */
   l_xe = 2 * TAN(beo->x_winkel / 2) * beo->mattsch /
             (beo->x_aufl * beo->x_punktgr);
```

```
/* Berechnung von x_step nach: */
/* x_step = xpg * xe (s. Buch) */

/* y-Koordinate von x_step berechnen: */
beo->x_step[1] = 0;     /* wegen xey = 0 */

if (beo->blickv[0] != 0  ||
    beo->blickv[2] != 0)        /* sx=0 und sz=0 ? */

{  /* Nein: => Fälle 1/2: */
   if (beo->blickv[0] != 0)     /* sx<>0 ?         */
   {  /* Ja: => Fall 1: */
      x_z = 0;
      z_x = 2;
   }
   else
   {  /* Nein: => Fall 2: */
      x_z = 2;
      z_x = 0;
   }

   /* z_x-Koordinate von x_step berechnen: */
   /* sz/sx  bzw.  sx/sz:                  */
   zahl = beo->blickv[z_x] / beo->blickv[x_z];

   beo->x_step[z_x] = WURZEL(l_xe*l_xe / (1 + zahl*zahl));

   /* x_z-Koordinate von x_step berechnen: */
   /* -xez*sz/sx  bzw.  -xex*sx/sz         */
   beo->x_step[x_z] = -beo->x_step[z_x] * zahl;
}
else
{  /* Ja => Fall 3: */
   beo->x_step[2] = 0;
   beo->x_step[0] = l_xe;
}

/* mit Punktgröße multiplizieren: */
MULVEK(beo->x_step, beo->x_punktgr, beo->x_step);


/* Berechnung von y_step nach: */
/* y_step = ypg * ye (s. Buch) */
```

```
   /* x-Koordinate von y_step berechnen: */
   beo->y_step[0] = 0;     /* wegen yex = 0 */

   if (beo->blickv[1] != 0  ||
       beo->blickv[2] != 0)        /* sy=0 und sz=0 ? */

   {  /* Nein: => Fälle 1/2: */
      if (beo->blickv[1] != 0)     /* sy<>0 ?         */
      {  /* Ja: => Fall 1: */
         y_z = 1;
         z_y = 2;
      }
      else
      {  /* Nein: => Fall 2: */
         y_z = 2;
         z_y = 1;
      }

      /* z_y-Koordinate von y_step berechnen: */
      /* sz/sy  bzw.  sy/sz:                  */
      zahl = beo->blickv[z_y] / beo->blickv[y_z];

      beo->y_step[z_y] = WURZEL(l_xe*l_xe / (1 + zahl*zahl));

      /* y_z-Koordinate von y_step berechnen: */
      /* -yez*sz/sy  bzw.  -yey*sy/sz         */
      beo->y_step[y_z] = -beo->y_step[z_y] * zahl;
   }
   else
   {  /* Ja => Fall 3: */
      beo->y_step[2] = 0;
      beo->y_step[0] = l_xe;
   }

   /* mit Punktgröße multiplizieren: */
   MULVEK(beo->y_step, beo->y_punktgr, beo->y_step);

}


/* Berechne die Koordinaten des  */
/* aktuellen Mattscheibenpunktes */
VOID get_punkt(beo, x, y, punkt)
struct beobachter *beo;     /* Zeiger auf Beobachterstruktur */
LONG     x, y;              /* Bildschirmkoordinaten         */
FLOAT    *punkt;
```

```
{
   REGISTER WORD xyz;

   /* Berechnung nach: P = B + s + x*xpg*xe + y*ypg*ye */
   /* wobei xpg*xe bzw. ypg*ye die Schrittweiten sind  */

   for (xyz=0; xyz<3; xyz++)
      punkt[xyz] = beo->blickv[xyz] + beo->pos[xyz] +
                   x*beo->x_step[xyz] + y*beo->y_step[xyz];
}

/*********************************/
/* Die große Ray-Tracing-Routine: */
/* Feststellung der Farbe des     */
/* nächsten Objektes auf dem      */
/* Sichtstrahl                    */
/*********************************/

VOID ray_trace(welt, strahl, punkt)
struct welt   *welt;   /* Zeiger auf Weltstruktur */
struct gerade *strahl; /* Zeiger auf Sichtstrahl  */
struct punkt  *punkt;  /* Zeiger auf Punktstrukt. */

{
   WORD     schnitt_objs();
   VOID     hintergrundfarbe(),
            lichtfarbe(),
            kugelfarbe(),
            flaechenfarbe();

   REGISTER WORD obj_typ; /* geschnittener Objekttyp     */
   WORD     *objekt;      /* Zeiger auf erstes Element   */
                          /* einer Objektstruktur        */

   /* Schnittpunkt des Strahles mit nächstem */
   /* Objekt ermitteln:                      */
   obj_typ =
   schnitt_objs(welt, punkt, strahl, &objekt);

   /* Ermittle Objektfarbe: */

   switch (obj_typ)
   {
      case HINTERGRUND:   /* kein Objekt geschnitten     */
         hintergrundfarbe(welt, punkt);
         break;
```

```
      case LICHTQUELLE:   /* wir sehen ein Licht!        */
         lichtfarbe(welt, objekt, punkt);
         break;
      case KUGEL:         /* wir sehen auf eine Kugel    */
         kugelfarbe(welt, objekt, strahl, punkt);
         break;
      case FLAECHE:       /* wir sehen auf eine Fläche   */
         flaechenfarbe(welt, objekt, strahl, punkt);
         break;
   }
}

/* Ermitteln des nächsten Schnittpunktes einer */
/* Geraden (Strahl) mit allen Objekten:        */
/* Return-Wert: geschnittener Objekttyp        */
/* in "objekt": Zeiger auf Objektstruktur      */
/*              des geschnittenen Objektes     */

WORD schnitt_objs(welt, punkt, strahl, objekt)
struct welt    *welt;   /* Zeiger auf Weltstruktur       */
struct punkt   *punkt;  /* Zeiger auf Punktstruktur      */
struct gerade  *strahl; /* Zeiger auf Strahl             */
WORD     **objekt;      /* Zeiger auf Zeiger auf erstes  */
                        /* Element einer Objektstruktur  */

{
   BOOL     schnitt_kugel_a(),
            schnitt_flaech();

   REGISTER WORD i;
   WORD     anzahl;
   FLOAT    s_min, s;   /* kleinstes s und s                    */
   WORD     *obj;       /* Zeiger auf 1. Objektstrukturelement */

   s_min = UNENDLICH;              /* Minimum initialisieren */

   /* Schnittpunkte mit Lichtquellen? */
   /***********************************/
   anzahl = welt->anz_licht;        /* Anzahl Lichtquellen    */
   for (i=0; i<anzahl; i++)
   {
         /* Berechne Schnittpunkt mit Lichtkugel */
      if (schnitt_kugel_a(obj=(WORD *)&welt->licht[i], strahl, &s))
      {
```

```
        /* Schnittpunkt existiert: */
        if (s < s_min)
        {
           s_min = s;                /* neues Minimum          */
           *objekt = obj;            /* Objektstruktur merken  */
        }
     }
  }

  /* Schnittpunkte mit Kugeln? */
  /*****************************/
  anzahl = welt->anz_kugeln;         /* Anzahl Kugeln          */
  for (i=0; i<anzahl; i++)
  {
        /* Berechne Schnittpunkt mit Kugel */
     if (schnitt_kugel_a(obj=(WORD *)&welt->kugeln[i], strahl, &s))
     {
        /* Schnittpunkt existiert: */
        if (s < s_min)
        {
           s_min = s;                /* neues Minimum          */
           *objekt = obj;            /* Objektstruktur merken  */
        }
     }
  }

  /* Schnittpunkte mit Flächen? */
  /******************************/
  anzahl = welt->anz_flaech;         /* Anzahl Flächen         */
  for (i=0; i<anzahl; i++)
  {
        /* Berechne Schnittpunkt mit Fläche: */
     if (schnitt_flaech(obj=(WORD *)&welt->flaech[i], strahl, &s))
     {
        /* Schnittpunkt existiert: */
        if (s < s_min)
        {
           s_min = s;                /* neues Minimum          */
           *objekt = obj;            /* Objektstruktur merken  */
           VEKLET(punkt->pos, sch_p); /* Schnittpunkt merken   */
           punkt->user1 = sch_m;     /* m und k merken         */
           punkt->user2 = sch_k;
        }
     }
  }
```

```
   /* Kein Schnittpunkt? */
   if (s_min == UNENDLICH)
      /* dann Hintergrund: */
      *objekt = (WORD *)&welt->hinten[0];

   /* Objekttyp und s_min übergeben: */
   punkt->s = s_min;
   return (**objekt);
}

/* Schnittpunktberechnungen: */
/*****************************/

/* Kugel:                                        */
/* Ermitteln des Schnittpunktes einer Kugel      */
/* mit einem von außen(!) auf die Kugel          */
/* treffenden Strahl (Gerade)                    */
/* Rückgabe: Schnittpunkt existiert/existiert    */
/*           nicht vor der Mattscheibe!          */

BOOL schnitt_kugel_a(kugel, strahl, s)
struct kugel   *kugel;      /* Zeiger auf Kugelstruktur */
struct gerade  *strahl;     /* Zeiger auf Gerade        */
FLOAT    *s;                /* Zeiger auf gesuchtes s   */

{
   FLOAT    v[3], *Pm, *bv, *Km;
   FLOAT    a, b, c, D;

   Pm = &strahl->p0[0];     /* Punkt Pm ist P0 der Gerade    */
   Km = &kugel->pos[0];     /* Punkt Km ist Kugelmittelpunkt */
   bv = &strahl->a[0];      /* bv ist Richtungsvektor Gerade */

   VEKSUB(v, Pm, Km);       /* v = Pm-Km                     */

   /* Berechnung der Koeffizienten für die */
   /* quadratische Gleichung von s:        */
   a = SKAPRD(bv, bv);      /* a = bx^2 + by^2 + bz^2        */
   b = 2*SKAPRD(v, bv);     /* b = 2*(vxbx + vyby + vzbz)    */

   c = SKAPRD(v, v) -       /* c = vx^2 + vy^2 + vz^2 - R^2  */
         kugel->radius * kugel->radius;

   /* Berechnung der Diskriminante für s:  *
   D = b*b - 4*a*c;         /* D = b^2 - 4*a*c               */
```

```
   /* Existiert ein Schnittpunkt? */
   if (D>0)
   {
      /* Ja: */
      D = WURZEL(D);
      *s = (-b-D)/(2*a);     /* kleineres s berechnen          */

      if (*s <= MINI_S)      /* s zu klein?                    */
         *s = (-b+D)/(2*a); /* ja => größeres s berechnen     */
      return (BOOL)(*s>MINI_S); /* s immer noch zu klein?     */
   }
   else
   {
      return (BOOL)(FALSE); /* Kein Schnittpunkt vorhanden    */
   }
}


/* Fläche:                                    */
/* Berechnen des Schnittpunktes einer         */
/* Gerade (Strahl) mit einem Parallelogramm   */
/* (z.B.: Rechteck oder Quadrat)              */
/* gesucht ist s als den Verlängerungsfaktor */
/* für den Richtungsvektor der Gerade         */

BOOL schnitt_flaech(flaeche, strahl, s)
struct flaeche *flaeche;    /* Zeiger auf Flächenstruktur */
struct gerade  *strahl;     /* Zeiger auf Geradenstruktur */
FLOAT    *s;                /* Zeiger auf gesuchtes s     */

{
   REGISTER WORD i, j;

   FLOAT    *n;            /* Normalenvektor der Fläche  */
   FLOAT    *P0;           /* Flächen-Fixpunkt           */
   FLOAT    *v, *w;        /* Flächen-Richtungsvektoren  */
   FLOAT    *Pm, *b;       /* Geradenparameter           */
   FLOAT    vektor[3];     /* Zwischenparameter          */
   FLOAT    nenner, q;

   P0 = &flaeche->ebene.p0[0];
   v  = &flaeche->ebene.a[0];
   w  = &flaeche->ebene.b[0];
```

```
Pm = &strahl->p0[0];
b  = &strahl->a[0];

n = &flaeche->ebene.n[0];

/* Bei Bedarf Berechnung des Normalenvektors n: */
if (NOT flaeche->ebene.n_valid)
{

   VEKPRD(n, v, w);              /* n = v x w        */
   flaeche->ebene.n_valid = TRUE;
}

/* Ist der Nenner (b*n) ungleich Null           */
/*          => Schnittpunkt mit Ebene existiert */
/* Ist der Nenner gleich Null                   */
/*          => kein Schnittpunkt mit Ebene      */

if ( (nenner = SKAPRD(b, n)) != 0.0)
{
   /* Schnittpunkt bei s ermitteln:      */
   VEKSUB(vektor, P0, Pm);
   *s = SKAPRD(vektor, n) / nenner;

   /* Koordinaten des Punktes berechnen: */
   GERADEN_PKT(sch_p, Pm, *s, b);


   /* Liegt der Punkt im Parallelogramm? */

   /* Passendes Gleichungspaar suchen:   */
   for (i=2; i>=0; i--)
      for (j=0; j<3; j++)

         /* vi<>0 und wj*vi-wi*vj<>0 ? */
         if ( i!=j && v[i] && (nenner=w[j]*v[i]-w[i]*v[j]) )
            goto weiter;

   return (FALSE);          /* bei Fehler zurück!         */

   weiter:  /* i und j sind die Nummern der Gleichungen */

   /* Berechnung der Ebenenfaktoren m und k: */
   sch_m = ( (sch_p[j]-P0[j]) * v[i] - (q=sch_p[i]-P0[i]) * v[j] )
           / nenner;
```

```
      if (sch_m >= 0.0  &&  sch_m <= 1.0)
         sch_k = (q - sch_m*w[i]) / v[i];
      else
         return (FALSE);        /* Schnittpunkt außerhalb des */
                                /* Parallelogramms            */
      if (sch_k >= 0.0  &&  sch_k <= 1.0)
         return (BOOL)(*s>MINI_S); /* s zu klein?              */
      else
         return (BOOL)(FALSE); /* Schnittpunkt außerhalb des */
                                /* Parallelogramms            */
   }
   else
   {
      return (BOOL)(FALSE);     /* kein Schnittpunkt          */
   }
}


/* Farbberechnungen: */
/*********************/

/* Hintergrund: */

VOID hintergrundfarbe(welt, punkt)
struct welt    *welt;   /* Zeiger auf Weltstruktur   */
struct punkt   *punkt;  /* Zeiger auf Punktstruktur  */

{
   struct hintergrund *hg; /* Zeiger auf Hintergr.-Strukt. */

   hg = &welt->hinten[0];

   /* Hintergrundfarbe holen: */
   COLLET(punkt->farbe, hg->farbe1);
}


/* Lichtquelle: */

VOID lichtfarbe(welt, lq, punkt)
struct welt    *welt;   /* Zeiger auf Weltstruktur   */
struct lichtquelle *lq; /* Zeiger auf Lichtstruktur  */
struct punkt   *punkt;  /* Zeiger auf Punktstruktur  */
```

```
{
   /* Lichtfarbe holen: */
   COLLET(punkt->farbe, lq->farbe);
}


/* Kugel: */

VOID kugelfarbe(welt, kugel, strahl, punkt)
struct welt    *welt;   /* Zeiger auf Weltstruktur   */
struct kugel   *kugel;  /* Zeiger auf Kugelstruktur  */
struct gerade  *strahl; /* Zeiger auf Strahlstruktur */
struct punkt   *punkt;  /* Zeiger auf Punktstruktur  */

{
   VOID     farbintens();

   FLOAT    normale[3]; /* Normalenvektor der Kugel  */

   /* Schnittpunkt nachträglich berechnen: */
   GERADEN_PKT(punkt->pos, strahl->p0, punkt->s, strahl->a);

   /* Normalenvektor berechnen: */
   /* nach: n = Schnittpunkt - Kugelmittelpunkt */
   VEKSUB(normale, punkt->pos, kugel->pos);

   /* Grundfarbe des Punktes bestimmen (Koh/Kod): */
   COLLET(punkt->farbe, kugel->material.farbe);

   /* Farbintensität bestimmen: */
   farbintens(welt, normale, strahl, punkt, &kugel->material);

}


/* Fläche: */

VOID flaechenfarbe(welt, flaeche, strahl, punkt)
struct welt    *welt;    /* Zeiger auf Weltstruktur    */
struct flaeche *flaeche; /* Zeiger auf Flächenstruktur */
struct gerade  *strahl;  /* Zeiger auf Strahlstruktur  */
struct punkt   *punkt;   /* Zeiger auf Punktstruktur   */

{
   VOID     farbintens();
```

```
REGISTER UWORD zahl1, zahl2;
FLOAT         fzahl1, fzahl2;
struct material *mat;

mat = &flaeche->material;

/* Grundfarbe des Punktes bestimmen (Koh/Kod): */
switch (mat->oberf_typ) /* je nach Oberflächentyp */
{
   case 0: /* normal, einfarbig: */
      COLLET(punkt->farbe, mat->farbe);
      break;

   case 1: /* kariert, zweifarbig: */
      /* Ermittlung des Feldes:            */
      /* INT(m * anzahl_karos_m) modulo 2) ergibt */
      /* die Feldart (0 oder 1) in m-Richtung     */
      /* INT(k * anzahl_karos_k) modulo 2) ergibt */
      /* die Feldart (0 oder 1) in k-Richtung     */

      zahl1 =
      ((ULONG)(punkt->user1 * mat->user.muster.anz_m
                              + 0.001)) % 2; /* m */
      zahl2 =
      ((ULONG)(punkt->user2 * mat->user.muster.anz_k
                              + 0.001)) % 2; /* k */
      if ( (zahl1 + zahl2) % 2 )
      {
         COLLET(punkt->farbe, mat->farbe);
      }
      else
      {
         COLLET(punkt->farbe, mat->farbe2);
      }
      break;

   case 2: /* 16x16-Musterdefinition */
      /* Ermittlung des aktuellen Bitstreifens:        */
      /* zahl = k*anz_muster_k bzw. zahl=m*anz_muster_m */
      /* streifennummer = INT(16* (zahl-INT(zahl) ))   */

      /* m: */
      zahl1 =
      fzahl1 = punkt->user1 * mat->user.muster.anz_m + 0.001;
```

```
        zahl1 =
        fzahl1 = 16 * (fzahl1 - (FLOAT)zahl1 );

        /* k: */
        zahl2 =
        fzahl2 = punkt->user2 * mat->user.muster.anz_k + 0.001;
        zahl2 =
        fzahl2 = 16 * (fzahl2 - (FLOAT)zahl2 );

        /* je nach gesetztem oder nicht gesetztem Bit */
        /* im Punktmuster Farbe 1 oder Farbe 2 setzen */
        if ( mat->user.muster.muster[zahl1]  &  (1L << zahl2) )
        {
           COLLET(punkt->farbe, mat->farbe);
        }
        else
        {
           COLLET(punkt->farbe, mat->farbe2);
        }
        break;
   }

   /* Farbintensität bestimmen: */
   farbintens(welt, flaeche->ebene.n, strahl, punkt, mat);

}


/* Reflexionsgesetz: Berechne den Reflexions-   */
/* vektor aus Einfallsvektor und Flächennormale */

VOID reflexion(refl_vek, einf_vek, normale)
FLOAT    *refl_vek;      /* Zeiger auf Reflexionsvektor */
FLOAT    *einf_vek;      /* Zeiger auf Einfallsvektor L */
FLOAT    *normale;       /* Zeiger auf Normalenvektor n */

{

   FLOAT    Lp[3];       /* Projizierter Einfallsvektor */
   FLOAT    q;

   /* 2 * Projizierter Einfallsvektor: Lp=2*(L*n)/n^2 * n */
   q = 2 * SKAPRD(einf_vek, normale) / SKAPRD(normale, normale);
   MULVEK(Lp, q, normale);
```

```
   /* Bestimme Reflexionsvektor nach: R=Lp-L */
   VEKSUB(refl_vek, Lp, einf_vek);
}


/*************************************************/
/* Farbintensität eines Objektpunktes berechnen */
/*************************************************/

VOID farbintens(welt, normale, strahl, punkt, material)
struct welt     *welt;     /* Zeiger auf Weltstruktur   */
FLOAT           *normale;  /* Zeiger auf Flächennormale */
struct gerade   *strahl;   /* Zeiger auf Strahlstruktur */
struct punkt    *punkt;    /* Zeiger auf Punktstruktur  */
struct material *material; /* Zeiger auf Materialstrukt.*/

{
   VOID     ray_trace(),
            reflexion();
   WORD     schnitt_objs();
   FLOAT    potenz();

   struct gerade  refl_strahl;  /* Reflexionsstrahl  */
   struct punkt   refl_punkt;   /* Reflexionspunkt   */
   FLOAT    farbe[3];

   /* Farbwertinitialisierung: */
   farbe[0] = farbe[1] = farbe[2] = 0.0;


   /* Spiegelungen bearbeiten: */
   /****************************/

   if (material->licht_typ == VERSPIEGELT)
   {
      /* Objekt ist verspiegelt:              */

      /* Intensität des gespiegelten Lichtes: */
      {
         REGISTER WORD  rgb;

         for (rgb=0; rgb<3; rgb++)
         {
            refl_punkt.intens_b[rgb] =
            punkt->intens_b[rgb] * material->spiegel_int[rgb];
```

```
      }
   }

   /* Abbruchkriterium: gespiegeltes Licht ist */
   /* zu dunkel oder: maximale Anzahl an       */
   /* Rekursionen ist erreicht                 */
   if ( (refl_punkt.intens_b[0] > SCHWARZ ||
         refl_punkt.intens_b[1] > SCHWARZ ||
         refl_punkt.intens_b[2] > SCHWARZ   ) &&
         rekur_zaehl < REKUR_MAX )
   {
      /* Licht ist noch hell genug =>             */
      /* Rekursion! Reflexionsstrahl wird weiter- */
      /* verfolgt. Gesucht ist die Farbe!         */

      /* berechne Reflexionsstrahl: */
      refl_strahl.a[0] = -strahl->a[0]; /* Einfallsstrahl */
      refl_strahl.a[1] = -strahl->a[1]; /* umdrehen       */
      refl_strahl.a[2] = -strahl->a[2];
      reflexion(refl_strahl.a, refl_strahl.a, normale);

      /* Fixpunkt des neuen Strahles = Schnittpunkt: */
      VEKLET(refl_strahl.p0, punkt->pos);

      /* Rekursionszähler erhöhen: */
      rekur_zaehl++;

      /* Hier ist sie: */
      ray_trace(welt, &refl_strahl, &refl_punkt);

      /* Rekursionszähler erniedrigen: */
      rekur_zaehl--;

      /* Ermittelte Farbe übertragen           */
      /* und Spiegelintensität berücksichtigen: */
      {
         REGISTER WORD rgb;

         for (rgb=0; rgb<3; rgb++)
            farbe[rgb] = refl_punkt.farbe[rgb] *
                         material->spiegel_int[rgb];
      }
   }
}
```

```
/* Spiegelnde und diffuse Lichtreflexion: */
/*****************************************/

{ /* Neuer Blockstart mit eigenen lokalen Variablen */
   REGISTER WORD rgb, lq;
   struct gerade licht_strahl; /* Strahl zum Licht          */
   struct lichtquelle *licht;  /* Zeiger auf Lichtq.-Struk. */
   FLOAT    cos_a, cos_b;
   FLOAT    nenner, potenz;
   WORD     *z_objekt;

   /* Fixpunkt der Licht- und Reflexionsstrahlen: */
   VEKLET(licht_strahl.p0, punkt->pos);
   VEKLET(refl_strahl.p0, punkt->pos);

   /* Alle Lichtquellen bearbeiten: */
   for (lq=0; lq < welt->anz_licht; lq++)
   {
      /* Adresse der aktuellen Lichtquellenstruktur:     */
      licht = &welt->licht[lq];

      /* Bestimme Strahl Schnittpunkt->Lichtquelle:      */
      /* P0=Schnittpunkt, a=(Lichtq-Schnittp)            */
      VEKSUB(licht_strahl.a, licht->pos, punkt->pos);

      /* Existiert ein Schnittpunkt vor der Lichtquelle? */
      schnitt_objs(welt, &refl_punkt, &licht_strahl, &z_objekt);

      if (z_objekt == (WORD *)licht)
      {
         /* nein: Objekt wird von Lichtquelle beschienen */
         /*       => Lichtintensität berechnen:          */

         /* Bestimme den Licht-Reflexionsvektor R:       */
         reflexion(refl_strahl.a, licht_strahl.a, normale);


         /* Berechne die RGB-Farbwerte des Punktes */
         /* für die Lichtquelle:                   */

         /* Intensitätsformel:                          */
         /* intens =                                    */
         /* Iq * [Kod*cos(a) + Kr*cos(b)^f] / (d * Kd) */
         /* mit:                                        */
         /*     Iq     = licht->farbe[]                 */
```

```
        /*      Kod     = punkt->farbe[]                */
        /*      cos(a)  = n'*L' = n*L/(|n|*|L|)         */
        /*      n       = normale                       */
        /*      L       = licht_strahl.a                */
        /*      Kr      = material->spieg_ref[]         */
        /*      cos(b)  = R'*S' = R*S/(|R|*|S|)         */
        /*      R       = refl_strahl.a                 */
        /*      S       = - strahl->a                   */
        /*      f       = material->fokus_ref[]         */
        /*      d       = VEK_LAENGE(licht_strahl.a)    */
        /*      Kd      = licht->dist_konst             */

        nenner = VEK_LAENGE(licht_strahl.a) *
                  licht->dist_konst;

        /* Cosinus berechnen: */
        cos_a = SKAPRD(normale, normale) *
                  SKAPRD(licht_strahl.a, licht_strahl.a);
        cos_a = SKAPRD(normale, licht_strahl.a) /
                  WURZEL(cos_a);

        cos_b = SKAPRD(refl_strahl.a, refl_strahl.a) *
                  SKAPRD(-strahl->a, -strahl->a);
        cos_b = SKAPRD(refl_strahl.a, -strahl->a) /
                  WURZEL(cos_b);

        for (rgb=0; rgb<3; rgb++)
        {
           potenz = POTENZ(cos_b, material->fokus_ref[rgb]);
           farbe[rgb] += licht->farbe[rgb] *
             ( punkt->farbe[rgb] * cos_a +
               material->spieg_ref[rgb] * potenz
             ) / nenner;

        } /* for rgb */
     } /* if */
  } /* for lq */

  /* Berücksichtigung der Hintergrundbeleuchtung */
  /* nach: I = Ih*Koh + sonst. Intensitäten      */
  for (rgb=0; rgb<3; rgb++)
  {
     punkt->farbe[rgb] =
     farbe[rgb] + welt->hintlicht[rgb] * punkt->farbe[rgb];
  }
```

```
   } /* Blockende */
}


/* Initialisiere Farbpalette: */
VOID init_farben()

{
   REGISTER UWORD reg, rgb;

   for (reg=0; reg<ANZ_PAL_FAB; reg++)
   {
      /* Farbpalette setzen: */
      SetRGB4(&Bildschirm->ViewPort, (LONG)reg,
              palette[reg][0],
              palette[reg][1],
              palette[reg][2] );

      /* Hardware-Intensitätswerte in programminterne */
      /* Werte umrechnen (von 0 - 2^16):              */
      for (rgb=0; rgb<3; rgb++)
      {
         if (G_MOD & EXTRA_HALFBRITE)
         {
            /* obere Farbregister haben die halbe Intensität: */
            /* berechne den Integer der halben Intensität:    */
            palette[reg+32][rgb] =
              ((palette[reg][rgb] & 0xfffffffeL) << 15 ) / ANZ_INT;
         }

         palette[reg][rgb] = (palette[reg][rgb] << 16) / ANZ_INT;
      }
   }
}


/* Zeichne einen Punkt auf den Bildschirm: */
/* Übergeben wird die RGB-Intensität       */
/* Gesucht wird das Palettenregister, das  */
/* dem RGB-Wert am nächsten kommt          */

VOID plot(x, y, punkt)
LONG           x, y;    /* Bildschirm-Koordinaten des Punktes */
struct punkt   *punkt;  /* Zeiger auf Punktstruktur           */
```

```
{
   WORD     zufall();

   REGISTER UWORD reg, rgb;
   UWORD    reg_min=0, zweit_reg_min=0;

   ULONG    farbe[3];

   ULONG    diff_sum_min = 0x7fffffff,  /* größt mögl. Wert */
            zweit_sum_min = 0x7fffffff,
            diff1_sum,
            diff2_sum;
   ULONG    diff_max_min = 0x7fffffff,
            zweit_max_min = 0x7fffffff,
            diff1_max,
            diff2_max;

   UWORD    reg2;
   LONG     diff;


   /* Punktfarbe zwischen 1 und 0 festlegen */
   /* und ins Palettenformat umwandeln:     */
   for (rgb=0; rgb<3; rgb++)
   {
      punkt->farbe[rgb] =
            MAX(0.0, punkt->farbe[rgb]);
      punkt->farbe[rgb] =
            MIN(1.0, punkt->farbe[rgb]);
      farbe[rgb] = (ULONG)(punkt->farbe[rgb] * 65536);
   }


   /* Passendes Palettenregister suchen: */

   /* bei EXTRA_HALFBRITE: doppelter Farbsatz */
   for (reg=0; reg<ANZ_FARBEN; reg++)
   {
      diff1_sum = 0;
      diff2_sum = 0;
      diff1_max = 0;
      diff2_max = 0;
```

```
    for (rgb=0; rgb<3; rgb++)
    {
       /* Summe der Differenzen der RGB-Intensitäten: */

       diff = palette[reg][rgb] - farbe[rgb];
       diff1_sum += diff = ABS(diff);

       /* und größte Differenz: */
       diff1_max = MAX(diff1_max, diff);
    }

    diff2_sum = diff1_sum;
    diff2_max = diff1_max;
    reg2      = reg;

    /* Minimum berechnen: */
    if (diff_sum_min > diff1_sum  ||
        (diff_sum_min == diff1_sum  &&  diff_max_min > diff1_max) )
    {
       diff2_sum = diff_sum_min;
       diff2_max = diff_max_min;
       reg2      = reg_min;
       diff_sum_min = diff1_sum;  /* Neues Minimum      */
       diff_max_min = diff1_max;
       reg_min      = reg;        /* Register merken    */
    }

    /* zweitkleinstes Minimum berechnen: */
    if ( (zweit_sum_min > diff2_sum  ||
          (zweit_sum_min == diff2_sum  &&  zweit_max_min > diff2_max) )

          && diff2_sum > diff_sum_min )
    {
       zweit_sum_min = diff2_sum; /* Neues Minimum      */
       zweit_max_min = diff2_max;
       zweit_reg_min = reg2;      /* Register merken    */
    }
}


/* Kleinste und zweitkleinste Summe zufällig mit */
/* gewichteter Wahrscheinlichkeit als Quelle für */
/* das jeweilige Farbregister auswählen:         */
if ( (FLOAT)zufall() + (zweit_sum_min << 8) /
                (zweit_sum_min+diff_sum_min) * 1.20 > 0xff )
```

```
   {
      /* Farbe einstellen: */
      SetAPen(RastPort, (LONG)reg_min);
   }
   else
   {
      /* Farbe einstellen: */
      SetAPen(RastPort, (LONG)zweit_reg_min);
   }

   /* Punkt plotten:            */
   /*****************************/
     WritePixel(RastPort, x, y);
   /*****************************/
}


/* Ermittlung einer Integer-Zufallszahl */
/* zwischen 0 und $100                  */

WORD zufall()

{
   static LONG zuf = 100001;

   zuf *= 125;
   zuf %= 2796203;
   return (WORD)( (zuf << 8)/2796203);
}


/*******************************/
/**                           **/
/**    D a t e n t e i l :    **/
/**                           **/
/*******************************/

/* Palettenfarben: */
ULONG palette[ANZ_FARBEN][3] =
   {
      { 1, 1, 1}, {14,14, 0},
      {15,15,15}, {13,13, 0},
      {15,13,13}, {12,12, 0},
      {15,10,10}, {11,11, 0},
      {15, 6, 6}, { 9, 9, 0},
```

```
      {15, 0, 0}, { 9, 9, 9},
      {14, 0, 0}, {13,13,15},
      {13, 0, 0}, {10,10,15},
      {12, 0, 0}, { 6, 6,15},
      {11, 0, 0}, { 0, 0,15},
      { 9, 0, 0}, { 0, 0,14},
      {13,13,13}, { 0, 0,13},
      {15,15,13}, { 0, 0,12},
      {15,15,10}, { 0, 0,11},
      {11,11,11}, { 0, 0, 9},
      {15,15, 0}, { 3, 3, 3},
   }; /* nur die ersten 32 Farben initialisieren */


/*********************************/
/* Sämtliche Daten für die Welt: */
/*********************************/

/* Die Urwelt: */
struct new_welt
{
   FLOAT    hintlicht[3];
} new_welt = { {0.32, 0.32, 0.32} };

/* Der Beobachter: */
struct beobachter new_beo =
{
   {21.0, 24.0,-29.0},      /* Position                */
   { 0.0,  0.0,  0.0},      /* Blickpunkt              */
   {-1.5, -1.5,  3.0},      /* alternativ: Blickvektor */
   1.0,                     /* Abstand zur Mattscheibe */
   1.00, 1.00,              /* Punktbreite/-höhe       */
   319L, 245L,              /* x-/y-Auflösung          */
   55.0,                    /* hor. Sichtwinkel (Grad) */
   {0.0, 0.0, 0.0},         /* später zu init. Werte   */
   {0.0, 0.0, 0.0},
   {0.0, 0.0, 0.0}
};

/* Der Hintergrund: */
struct hintergrund new_hint =
{
   HINTERGRUND,             /* Typ        */
   UNENDLICH,               /* "Position" */
   {0.00, 0.00, 0.80},      /* Farbe 1    */
```

```
   {0.00, 0.00, 0.00}        /* Farbe 2    */
};

/* Die Lichtquellen: */
struct lichtquelle new_licht[] =
{
   {
    LICHTQUELLE,             /* Typ        */
    {23.5,20.0, 20.0},       /* Position   */
     1.0,                    /* Radius     */
    {1.00, 1.00, 1.00},      /* Farbe      */
     0.18                    /* Kd         */
   },
   {
    LICHTQUELLE,             /* Typ        */
    { 7.0, 1.0, -6.0},       /* Position   */
     1.0,                    /* Radius     */
    {1.00, 1.00, 1.00},      /* Farbe      */
     1.9                     /* Kd         */
   }
};

/* Die Objekte: */

/* 1. Kugeln: */
struct kugel new_kugeln[] =
{
   {
    KUGEL,                   /* Typ        */
    { 6.0,  6.5,  3.0},      /* Position   */
    3.0,                     /* Radius     */
    { /* Material: */
     {0.00, 0.00, 0.00},     /* Farbe      */
     {0.00, 0.00, 0.00},     /* Farbe 2    */
     {0.00, 0.00, 0.00},     /* Farbe 3    */
     {0.00, 0.00, 0.00},     /* Kr         */
     { 5.0,  5.0,  5.0},     /* Fokus f    */
     {0.85, 0.85, 0.85},     /* Spiegelint */
     VERSPIEGELT,            /* Lichtop    */
     0                       /* Oberfl.typ */
     /* User */
    }
   },
   {
```

```
  KUGEL,                    /* Typ         */
  { 0.0, 13.0, 0.0},        /* Position    */
  5.0,                      /* Radius      */
  { /* Material: */
   {0.00, 0.00, 0.00},      /* Farbe       */
   {0.00, 0.00, 0.00},      /* Farbe 2     */
   {0.00, 0.00, 0.00},      /* Farbe 3     */
   {0.00, 0.00, 0.00},      /* Kr          */
   { 5.0,  5.0,  5.0},      /* Fokus f     */
   {0.85, 0.85, 0.85},      /* Spiegelint  */
   VERSPIEGELT,             /* Lichtop     */
   0                        /* Oberfl.typ  */
   /* User */
  }
 },
 {
  KUGEL,                    /* Typ         */
  {15.0,  2.0, -2.5},       /* Position    */
  2.0,                      /* Radius      */
  { /* Material: */
   {1.00, 0.00, 0.00},      /* Farbe       */
   {0.00, 0.00, 0.00},      /* Farbe 2     */
   {0.00, 0.00, 0.00},      /* Farbe 3     */
   {9.00, 9.00, 9.00},      /* Kr          */
   {60.0, 60.0, 60.0},      /* Fokus f     */
   {0.00, 0.00, 0.00},      /* Spiegelint  */
   NORMAL,                  /* Lichtop     */
   0,                       /* Oberfl.typ  */
   /* User */
  }
 },
 {
  KUGEL,                    /* Typ         */
  { 8.0,  2.5, 22.5},       /* Position    */
  2.5,                      /* Radius      */
  { /* Material: */
   {0.00, 0.00, 1.00},      /* Farbe       */
   {0.00, 0.00, 0.00},      /* Farbe 2     */
   {0.00, 0.00, 0.00},      /* Farbe 3     */
   {1.00, 1.00, 1.00},      /* Kr          */
   { 4.0,  4.0,  4.0},      /* Fokus f     */
   {0.00, 0.00, 0.00},      /* Spiegelint  */
   NORMAL,                  /* Lichtop     */
   0,                       /* Oberfl.typ  */
   /* User */
```

```
 }
},
{
 KUGEL,                    /* Typ         */
 { 6.1,  1.2, 18.9},       /* Position    */
 1.0,                      /* Radius      */
 { /* Material: */
  {1.00, 1.00, 0.00},      /* Farbe       */
  {0.00, 0.00, 0.00},      /* Farbe 2     */
  {0.00, 0.00, 0.00},      /* Farbe 3     */
  {9.50, 9.50, 9.50},      /* Kr          */
  {50.0, 50.0, 50.0},      /* Fokus f     */
  {0.00, 0.00, 0.00},      /* Spiegelint */
  NORMAL,                  /* Lichtop     */
  0,                       /* Oberfl.typ */
  /* User */
 }
},
{
 KUGEL,                    /* Typ         */
 {20.5,  2.2, 6.5},        /* Position    */
 1.5,                      /* Radius      */
 { /* Material: */
  {1.00, 0.00, 0.00},      /* Farbe       */
  {0.00, 0.00, 0.00},      /* Farbe 2     */
  {0.00, 0.00, 0.00},      /* Farbe 3     */
  {9.50, 9.50, 9.50},      /* Kr          */
  {50.0, 50.0, 50.0},      /* Fokus f     */
  {0.00, 0.00, 0.00},      /* Spiegelint */
  NORMAL,                  /* Lichtop     */
  0,                       /* Oberfl.typ */
  /* User */
 }
},
{
 KUGEL,                    /* Typ         */
 {12.0,  2.7, 11.0},       /* Position    */
 2.5,                      /* Radius      */
 { /* Material: */
  {0.00, 0.00, 1.00},      /* Farbe       */
  {0.00, 0.00, 0.00},      /* Farbe 2     */
  {0.00, 0.00, 0.00},      /* Farbe 3     */
  {5.00, 5.00, 5.00},      /* Kr          */
  {60.0, 60.0, 60.0},      /* Fokus f     */
  {0.00, 0.00, 0.00},      /* Spiegelint */
```

```
      NORMAL,                  /* Lichtop    */
      0,                       /* Oberfl.typ */
      /* User */
     }
    }
};

struct flaeche new_flaechen[] =
{
   { /* linke Seitenwand: */
     FLAECHE,                 /* Typ        */
     { 0.0,  0.0,-30.0},      /* 3 Ecken    */
     { 0.0, 20.0,-30.0},
     { 0.0,  0.0, 30.0},
     { /* Ebene, wird später initialisiert: */
      {0,0,0},{0,0,0},{0,0,0},FALSE,{0,0,0}
     },
     { /* Material: */
      {0.30, 0.30, 0.30},     /* Farbe      */
      {1.00, 0.00, 0.00},     /* Farbe 2    */
      {0.00, 0.00, 0.00},     /* Farbe 3    */
      {0.70, 0.70, 0.70},     /* Kr         */
      {1.00, 1.00, 1.00},     /* Fokus f    */
      {0.00, 0.00, 0.00},     /* Spiegelint */
      NORMAL,                 /* Lichtop    */
      0,                      /* Oberfl.typ */
      /* User: */
      {
       { /* Muster (nur nach Einstellung von Oberfl.typ = 2): */
        10,10,       /* Anzahl Muster m/k-Rtg. */
        {0x0000, 0x0000, 0x0000, 0x001c,
         0x003c, 0x005c, 0x009c, 0x011c,
         0x021c, 0x041c, 0x0ffc, 0x101c,
         0x201c, 0x401c, 0xe03e, 0x0000}
       }
      }
     }
   },
   { /* hintere Wand (verspiegelt): */
     FLAECHE,                 /* Typ        */
     { 1.3,  1.3, 29.9},      /* 3 Ecken    */
     { 1.3, 18.7, 29.9},
     {18.7,  1.3, 29.9},
     { /* Ebene, wird später initialisiert: */
      {0,0,0,},{0,0,0},{0,0,0},FALSE,{0,0,0}
```

```
  },
  { /* Material: */
   {0.00, 0.00, 0.00},   /* Farbe      */
   {0.00, 0.00, 0.00},   /* Farbe 2    */
   {0.00, 0.00, 0.00},   /* Farbe 3    */
   {0.00, 0.00, 0.00},   /* Kr         */
   { 1.0,  1.0,  1.0},   /* Fokus f    */
   {0.85, 0.85, 0.85},   /* Spiegelint */
   VERSPIEGELT,          /* Lichtop    */
   0,                    /* Oberfl.typ */
   /* User */
  }
 },
 { /* hintere Wand (unverspiegelt): */
  FLAECHE,               /* Typ        */
  { 0.0,  0.0, 30.0},    /* 3 Ecken    */
  { 0.0, 20.0, 30.0},
  {20.0,  0.0, 30.0},
  { /* Ebene, wird später initialisiert: */
   {0,0,0},{0,0,0},{0,0,0},FALSE,{0,0,0}
  },
  { /* Material: */
   {1.00, 1.00, 1.00},   /* Farbe      */
   {0.00, 0.00, 0.00},   /* Farbe 2    */
   {0.00, 0.00, 0.00},   /* Farbe 3    */
   {1.10, 1.10, 1.10},   /* Kr         */
   {99.0, 99.0, 99.0},   /* Fokus f    */
   {0.00, 0.00, 0.00},   /* Spiegelint */
   NORMAL,               /* Lichtop    */
   0,                    /* Oberfl.typ */
   /* User */
  }
 },
 { /* Tischplatte: */
  FLAECHE,               /* Typ        */
  { 0.0,  0.0,-30.0},    /* 3 Ecken    */
  { 0.0,  0.0, 30.0},
  {20.0,  0.0,-30.0},
  { /* Ebene, wird später initialisiert: */
   {0,0,0},{0,0,0},{0,0,0},FALSE,{0,0,0}
  },
  { /* Material: */
   {1.00, 1.00, 0.00},   /* Farbe      */
   {0.00, 0.00, 0.00},   /* Farbe 2    */
   {0.00, 0.00, 0.00},   /* Farbe 3    */
```

```
    {1.00, 1.00, 1.00},    /* Kr          */
    { 1.0,  1.0,  1.0},    /* Fokus f     */
    {0.00, 0.00, 0.00},    /* Spiegelint */
    NORMAL,                /* Lichtop     */
    2,                     /* Oberfl.typ */
    /* User: */
    {
     { /* Muster: */
      5,5,      /* Anzahl Muster m/k-Rtg. */
      {Oxeeee, Oxeeee, Oxeeee, Oxeeee,
       Oxeeee, Oxeeee, Oxeeee, Oxeeee,
       Oxeeee, Oxeeee, Oxeeee, Oxeeee,
       Oxeeee, Oxeeee, Oxeeee, Oxeeee}
     }
    }
   }
  },
  { /* Schachtelboden: */
   FLAECHE,                /* Typ         */
   { 5.0,  0.1,  3.0},     /* 3 Ecken     */
   {15.0,  0.1,  3.0},
   { 5.0,  0.1, 20.0},
   { /* Ebene, wird später initialisiert: */
    {0,0,0},{0,0,0},{0,0,0},FALSE,{0,0,0}
   },
   { /* Material: */
    {1.00, 1.00, 1.00},    /* Farbe       */
    {0.00, 0.00, 0.00},    /* Farbe 2     */
    {0.00, 0.00, 0.00},    /* Farbe 3     */
    {1.00, 1.00, 1.00},    /* Kr          */
    { 1.0,  1.0,  1.0},    /* Fokus f     */
    {0.00, 0.00, 0.00},    /* Spiegelint */
    NORMAL,                /* Lichtop     */
    1,                     /* Oberfl.typ */
    /* User: */
    {
     { /* Karos: */
      4, 4,      /* Anzahl Karos m/k-Rtg. */
      {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
     }
    }
   }
  },
  { /* Schachteldeckel: */
   FLAECHE,                /* Typ         */
```

```
  {15.0,  5.0,  3.0},     /* 3 Ecken    */
  {15.0,  5.0, 20.0},
  {22.0,  0.0,  3.0},
  { /* Ebene, wird später initialisiert: */
   {0,0,0},{0,0,0},{0,0,0},FALSE,{0,0,0}
  },
  { /* Material: */
   {1.00, 1.00, 1.00},    /* Farbe      */
   {0.30, 0.30, 0.30},    /* Farbe 2    */
   {0.00, 0.00, 0.00},    /* Farbe 3    */
   {1.00, 1.00, 1.00},    /* Kr         */
   { 1.0,  1.0,  1.0},    /* Fokus f    */
   {0.00, 0.00, 0.00},    /* Spiegelint */
   NORMAL,                /* Lichtop    */
   1,                     /* Oberfl.typ */
   /* User: */
   {
    { /* Karos: */
     4, 4,       /* Anzahl Karos m/k-Rtg. */
     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0}
    }
   }
  }
 },
 { /* vordere Schachtelwand: */
  FLAECHE,                /* Typ        */
  { 5.0,  0.1,  3.0},     /* 3 Ecken    */
  { 5.0,  5.0,  3.0},
  {15.0,  0.1,  3.0},
  { /* Ebene, wird später initialisiert: */
   {0,0,0},{0,0,0},{0,0,0},FALSE,{0,0,0}
  },
  { /* Material: */
   {1.00, 0.00, 0.00},    /* Farbe      */
   {0.00, 0.00, 0.00},    /* Farbe 2    */
   {0.00, 0.00, 0.00},    /* Farbe 3    */
   {0.80, 0.80, 0.80},    /* Kr         */
   { 1.0,  1.0,  1.0},    /* Fokus f    */
   {0.00, 0.00, 0.00},    /* Spiegelint */
   NORMAL,                /* Lichtop    */
   0,                     /* Oberfl.typ */
   /* User */
  }
 },
```

```
{ /* hintere Schachtelwand: */
 FLAECHE,                  /* Typ         */
 { 5.0,  0.1, 20.0},       /* 3 Ecken     */
 { 5.0,  5.0, 20.0},
 {15.0,  0.1, 20.0},
 { /* Ebene, wird später initialisiert: */
  {0,0,0},{0,0,0},{0,0,0},FALSE,{0,0,0}
 },
 { /* Material: */
  {1.00, 0.00, 0.00},      /* Farbe       */
  {0.00, 0.00, 0.00},      /* Farbe 2     */
  {0.00, 0.00, 0.00},      /* Farbe 3     */
  {0.90, 0.90, 0.90},      /* Kr          */
  { 1.0,  1.0,  1.0},      /* Fokus f     */
  {0.00, 0.00, 0.00},      /* Spiegelint  */
  NORMAL,                  /* Lichtop     */
  0,                       /* Oberfl.typ  */
  /* User */
 }
},
{ /* Schachtelwand rechts: */
 FLAECHE,                  /* Typ         */
 {15.0,  0.1,  3.0},       /* 3 Ecken     */
 {15.0,  5.0,  3.0},
 {15.0,  0.1, 20.0},
 { /* Ebene, wird später initialisiert: */
  {0,0,0},{0,0,0},{0,0,0},FALSE,{0,0,0}
 },
 { /* Material: */
  {1.00, 0.00, 0.00},      /* Farbe       */
  {0.00, 0.00, 0.00},      /* Farbe 2     */
  {0.00, 0.00, 0.00},      /* Farbe 3     */
  {0.90, 0.90, 0.90},      /* Kr          */
  { 1.0,  1.0,  1.0},      /* Fokus f     */
  {0.00, 0.00, 0.00},      /* Spiegelint  */
  NORMAL,                  /* Lichtop     */
  0,                       /* Oberfl.typ  */
  /* User */
 }
},
{ /* Schachtelwand links: */
 FLAECHE,                  /* Typ         */
 { 5.0,  0.1,  3.0},       /* 3 Ecken     */
 { 5.0,  5.0,  3.0},
 { 5.0,  0.1, 20.0},
```

```
    { /* Ebene, wird später initialisiert: */
     {0,0,0},{0,0,0},{0,0,0},FALSE,{0,0,0}
    },
    { /* Material: */
     {1.00, 0.00, 0.00},   /* Farbe      */
     {0.00, 0.00, 0.00},   /* Farbe 2    */
     {0.00, 0.00, 0.00},   /* Farbe 3    */
     {0.90, 0.90, 0.90},   /* Kr         */
     { 1.0,  1.0,  1.0},   /* Fokus f    */
     {0.00, 0.00, 0.00},   /* Spiegelint */
     NORMAL,               /* Lichtop    */
     0,                    /* Oberfl.typ */
     /* User */
    }
   }
};


/* Einrichtung aller Datenstrukturen: */

VOID init_welt(welt, beo)
struct welt       *welt;   /* Zeiger auf Weltstruktur */
struct beobachter *beo;    /* Zeiger auf Beobachter   */

{
   REGISTER WORD fl;

   /* Weltstruktur einrichten: */
   welt->anz_kugeln =
     sizeof new_kugeln / sizeof (struct kugel);
   welt->kugeln = (struct kugel *)new_kugeln;
   welt->anz_flaech =
     sizeof new_flaechen / sizeof (struct flaeche);
   welt->flaech = (struct flaeche *)new_flaechen;
   welt->anz_licht =
     sizeof new_licht / sizeof (struct lichtquelle);
   welt->licht = (struct lichtquelle *)new_licht;
   welt->anz_hinten = 1;
   welt->hinten = (struct hintergrund *)(&new_hint);

   welt->anz_obj = welt->anz_kugeln +
                   welt->anz_flaech +
                   welt->anz_licht + 1;

   COLLET(welt->hintlicht, new_welt.hintlicht);
```

```
    /* Beobachterstruktur einrichten: */
    *beo = new_beo;

    beo->x_winkel *= PI/180;      /* Winkel in Rad umrechnen */

    /* Ebenenstrukturen in allen Flächenstrukturen einrichten: */
    for (fl=0; fl<welt->anz_flaech; fl++)
    {
       /* Fixpunkt P0 und Richtungsvektoren a, b: */
       VEKLET(new_flaechen[fl].ebene.p0, new_flaechen[fl].p1);
       VEKSUB(new_flaechen[fl].ebene.a,
              new_flaechen[fl].p2, new_flaechen[fl].p1);
       VEKSUB(new_flaechen[fl].ebene.b,
              new_flaechen[fl].p3, new_flaechen[fl].p1);
    }
}
```

Na, ist das nichts? Da macht das Programmieren doch wieder richtig Spaß!

Was ist zur Bedienung zu sagen? Nun, eigentlich nicht viel. Das Wichtigste ist das Starten, dann geht alles ganz von alleine. Es dauert lediglich seine Zeit, bis das gesamte Bild fertig ist. Das hängt natürlich von der Anzahl und der Größe der Objekte, vor allem von der Anzahl der Lichtquellen und der Anzahl der verspiegelten Körper ab; und genauso selbstverständlich auch leider von dem Compiler, den Sie verwenden. Die fertig ausführbare Version auf der Diskette wurde mit dem Aztek-Compiler V3.4a kompiliert und ist aufgrund der vielen Floatingpoint-Operationen etwa zwei- bis dreimal schneller als das gleiche Programm, das mit dem Lattice-Compiler in einer alten Version vor V3.10 (z.B. V3.03) kompiliert wurde. Trotzdem können Sie sich bei dem hier voreingestellten Stilleben auf einige Stunden Rechenzeit gefaßt machen (obwohl das Programm noch recht schnell ist – können Sie sich vorstellen, wie viele Tage oder Wochen ein Amiga-Basic-Programm wohl daran arbeiten würde?). Glücklicherweise sind Lattice-C-Besitzer seit V3.10 nun ebenfalls in der Lage, die schnellen FFP-Routinen des Amiga-Betriebssystems ohne die früheren Probleme zu nutzen (s. Kapitel 2). Damit unterscheiden sich die Ausführgeschwindigkeiten der beiden Compiler in diesem, vor Floatingpoint-Operationen nur so strotzenden Programm kaum noch. Aber das Warten lohnt sich! Dauert Ihnen das Ganze zu lange, dann versuchen Sie es doch einmal mit nur einer einzigen Kugel und einer Lichtquelle. Das Bild ist dann in einigen Minuten fertig.

Wenn Sie Ihren Rechner irgendwann wieder ganz für sich haben möchten, dann betätigen Sie einfach den Esc-Knopf (auch während der Zeichenphase) und das Programm endet. Oder Sie schieben den Screen mit der Maus herunter. Wenn Sie dann noch die Task-Priorität unter CLI mit dem Kommando **CHANGETASKPRI** für das Ray-Tracing-Programm heruntergesetzt haben, können Sie ganz normal mit Ihrem Rechner weiterarbeiten, während das Ray-Tracing-Programm auf vollen Touren läuft. Nur abstürzen darf Ihnen der Rechner dann nicht.

Gehen wir doch einmal davon aus, Sie hätten nun das vollständige Bild ganz fertig gezeichnet auf dem Bildschirm stehen. Der Besuch, der sich Ihre Kreation einmal anschauen wollte, kommt, aber erst morgen. Nun, Sie brauchen den Rechner nicht die ganze Zeit eingeschaltet zu lassen. Ein Druck auf die Taste <S>, und schon speichert Ihr Amiga das Bild im IFF-Format auf Disk. Auf der beiliegenden Diskette befindet sich ein kleines Ladeprogramm, das Ihnen dieses Bild jederzeit wieder zum Vorschein holt. Haben Sie nicht gerade ein EXTRA_HALFBRITE-Bild, dann ist es ebenfalls kein Problem, Ihr fertiges Bild z.B. in Deluxe Paint einzuladen und nachträglich zu bearbeiten.

Für die Darstellung des Bildes wurde in der abgedruckten Version der Grafikmodus **EXTRA_HALFBRITE** gewählt, der es ermöglicht, insgesamt 64 verschiedene Farben gleichzeitig auf den Bildschirm zu zaubern. Für diejenigen unter Ihnen, deren Rechner nicht in der Lage ist, diesen Modus darzustellen (die Amiga 1000 kennen ihn noch nicht und einige 500er haben ebenfalls Probleme damit), und die keinen Compiler besitzen, um im obigen Programm die wenigen Änderungen vorzunehmen und neu zu kompilieren, befindet sich ein entsprechendes kompiliertes Programm für 32 Farben auf der beigelieferten Diskette. Die Farbqualität ist dabei natürlich erheblich verringert (bzw. Sie können weniger unterschiedliche Farbtöne nutzen). Ebenfalls auf Diskette finden Sie sowohl Source-File als auch das ausführbare Programm für ein Bild im HAM-Modus (Hold-and-Modify), bei dem alle 4096 Farben des Amiga gleichzeitig genutzt werden können (mit einigen Einschränkungen allerdings). Lediglich die Programmierung der Punkt-Setz-Routine mußte dabei ein wenig abgewandelt werden. Sehr kommt es bei allen Modi auf die geschickte Wahl der Farben in den Palettenregistern an. Sie bestimmt erst richtig und in letzter Instanz die Qualität des gesamten Bildes.

Es sollte für Sie ebenfalls kein Problem sein, den Interlace-Modus einzustellen und mit der doppelten Auflösung zu arbeiten. Ein Bild zu berechnen, dauert dann allerdings auch doppelt so lange. Das sollte Ihnen hier zunächst einmal erspart bleiben (obwohl ich Ihnen sagen muß, daß der Interlace-Modus ganz hervorragende Ergebnisse bringt). Vergessen Sie bei der Einstellung des Interlace-Modus nicht, die vertikale Punktgröße (also **ypg** in der Beobachterstruktur) zu halbieren, da ein Punkt schließlich nur halb zu hoch wie breit ist. Vernachlässigen Sie diese Aufgabe, erscheint Ihr Bild verzerrt auf dem Monitor.

Wie Sie Punkte im HAM-Modus setzen, das erfahren Sie z.B. im ROM-Kernel-Reference-Manual, Libraries and Devices (Beispielprogramm im Kapitel 1, Advanced Graphics Examples).

Wie Sie mit diesem Programm eine ganz neue Szenerie entwerfen und einstellen können, das erfahren Sie ein paar Zeilen tiefer – eine spannende Sache!

Kommen wir also zum Programm selber. Die notwendigen mathematischen und algorithmischen Grundlagen haben Sie sich bereits in den vorangegangenen Abschnitten erarbeitet. Sie sind jedenfalls Voraussetzung für die nun folgenden Erläuterungen.

Beginnen wir ganz vorne. Dort werden unter der Überschrift »Hardware-Konstanten« einige Konstanten per #define eingestellt, die festlegen, wie viele Farben erreichbar sind, welche Hardware-Auflösung und welcher Grafikmodus gewählt werden. Hier können Sie natürlich bei Bedarf jederzeit Änderungen vornehmen. Wie gesagt, zur Zeit sind eingestellt: 64 Farben bei

32 Palettenregistern, damit 6 Farbebenen, eine Auflösung von 320x200 im EXTRA_HALF-BRITE-Modus. Die maximal 16 Farbintensitätsstufen sind beim Amiga natürlich fest eingestellt und unveränderbar.

Als nächstes erfolgen altbekannte Definitionen und Initialisierungen: Strukturen für einen neuen Bildschirm, ein neues Fenster wird definiert und initialisiert, in **main( )** öffnet das Programm einige Libraries, den neuen Screen sowie das neue Window und springt dann zum eigentlichen Hauptprogramm **do_program( ),** mit dem wir uns gleich beschäftigen wollen.

Vorher aber haben Sie es noch mit einigen #defines zu tun. Definiert werden hier vor allem Makros, für die aus Geschwindigkeitsgründen kein separates Unterprogramm eingerichtet werden sollte, die aber (fast) genauso wie normale Funktionen verwendet werden können (wenn »Makros« in C Ihnen kein Begriff sind, vielleicht schauen Sie dann noch einmal in Ihr C-Buch zum Thema »Präprozessor« oder »#define«): Neben Maximum, Minimum, Absolutwert und Potenzierung werden hier sämtliche notwendigen Vektoroperationen definiert. Ein Vektor wird im Programm durchgehend als dreielementiges Array betrachtet. Jedes Element stellt eine Vektorkoordinate dar (z.B.: v[0], v[1] und v[2] sind die drei Elemente vx, vy und vz des Vektors v).

Sollen nun zwei Vektoren $\vec{a}$ und $\vec{b}$ addiert und als Vektor $\vec{c}$ gespeichert werden ($\vec{c} = \vec{a} + \vec{b}$), dann muß das Programm dazu gemäß der Definition der Vektoraddition, die Ihnen nun bereits geläufig sein sollte, jedes einzelne Element von von $\vec{a}$ zum entsprechenden Element von $\vec{b}$ addieren und als Element von $\vec{c}$ abspeichern:

```
c[0] = a[0] + b[0]
c[1] = a[1] + b[1]
c[2] = a[2] + b[2]
```

Entspricht also:

$$\begin{pmatrix} c_x \\ c_y \\ c_z \end{pmatrix} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} + \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix}$$

Entsprechendes gilt für alle anderen Vektoroperationen. Die verschiedenen Makros (für die Addition das Makro **VEKADD(c,a,b))** führen nun genau diese Operationen aus. Dabei müssen später im Programm als Makroparameter nur die Namen der Arrays angegeben werden, z.B.: VEKADD(ergebnis, vektor1, vektor2) addiert die Elemente der Vektoren vektor1[ ] und vektor2[ ] und speichert das Resultat in ergebnis[ ]. Achten Sie darauf, daß nur einige Makros in größere Ausdrücke eingebaut werden können. Die meisten haben sozusagen den »Rückgabewert« VOID.

Der nächste wichtige Schritt ist die Deklaration aller im Programm verwendeten Strukturen. Sie sind praktisch Dreh- und Angelpunkt im gesamten System. Grundsätzlich werden (fast) alle Daten in Strukturen gehalten, die als Kommunikationsparameter von Funktion zu Funktion übergeben werden. Übergeben werden dabei nie die Strukturen selbst, sondern lediglich ihre Zeiger. Auf diese Weise können jeder Funktion sehr schnell sehr viele lokale Daten zugänglich gemacht werden. Gehen wir die Strukturen doch einmal der Reihe nach durch:

struct punkt:

Dies ist eigentlich eine reine Kommunikationsstruktur. Sie enthält Daten, die im Laufe des Programmes einmal errechnet und später vielleicht noch einmal benötigt werden. So ist z.B. die Errechnung eines Schnittpunktes nicht in den verschiedenen Routinen separat notwendig. Wenn er einmal ermittelt wurde, steht er ab sofort allen anderen Funktionen zur Verfügung. Auch andere Strukturen enthalten solche »Zwischenwertspeicher«, die erst im Laufe des Programmes aufgefüllt werden. Hier allerdings haben wir es mit all den Daten zu tun, die einen Punkt im Raum beschreiben, seine Position, seine Farbe, sein s-Wert für die vektorielle Geradengleichung des Sichtstrahles usw. Einige Speicher (user0-2) sind sogar teilweise noch ungenutzt und für spätere Erweiterungen gedacht. Hier wird später letztendlich die Farbe des Bildschirmpunktes stehen.

struct gerade:

Eine vektorielle Geradendefinition besteht aus einem festen Punkt P0 und einem sogenannten Richtungsvektor $\vec{a}$ (s. vektorielle Geradengleichung). Jede Gerade im Programm wird auf diese Weise und mit dieser Struktur festgelegt.

struct ebene:

Gemäß der vektoriellen k-m-Gleichung (oder s-t-Gleichung) ist eine Ebene durch einen Fixpunkt P0 und zwei Richtungsvektoren $\vec{a}$ und $\vec{b}$ bestimmt. Eventuell gibt es die Möglichkeit, auch den Normalenvektor der Ebene hinzuzuziehen (s. Kapitel 4.3), was bestimmte Berechnungen vereinfacht. Diese Parameter werden in der Ebenenstruktur festgehalten. Da der Normalenvektor nicht immer notwendig ist, bestimmt die BOOL-Variable n_valid, ob er in der Struktur zur Zeit angegeben ist oder nicht.

struct material:

Schon kommen wir zu den interessanten Teilen des Programmes. Die besagte Struktur legt die spezifischen Eigenschaften eines Objektes im Raum fest. Sie ist natürlich nie eigenständig vorhanden, sondern stets Teil einer anderen, einer Objektstruktur (z.B. für eine Kugel oder eine Fläche). Hier erfährt das Programm, welche Farbe(n), welche Reflexionseigenschaften das Objekt besitzt, ob es verspiegelt (die Option »Durchsichtig« ist in dieser Version des Programmes nicht implementiert) oder ob seine Oberfläche besonders strukturiert ist. Allgemein ist hier später der Ort für die meisten Veränderungen an einem Objekt (sieht es poliert, matt oder metallisch aus? Ist es hell oder dunkel, rot, grün oder weiß?).

Diese Struktur wurde möglichst offen konzipiert. Sollten Sie irgendwann einmal Erweiterungen an diesem Programm vornehmen wollen, so bietet sie Ihnen Platz für viele Parameter, die das Objekt näher beschreiben können. Bereits implementiert sind als Oberflächenstrukturen für die Fläche Karos und beliebige Muster. Die Musterdefinition befindet sich dabei innerhalb der Struktur-Union in der Struktur »muster« (s.u.). Etwas weiter unten werden wir noch näher auf diese Strukturen eingehen.

struct hintergrund:

Da sind wir doch bereits bei der ersten Objektstruktur angelangt: dem Hintergrund. Normalerweise hat eine Objektstruktur den folgenden groben Aufbau:

Typenkennzeichnung des Objektes (z.B. »0« für Hintergrund)
Position des Objektes
sonstige beschreibenden Parameter
Materialeigenschaften (meist struct material)
An den Stellen, an denen kein Objekt zu sehen ist, erscheint der Hintergrund. Als einziger veränderbarer Parameter ist hier zur Zeit die Farbe angegeben. Platz ist für eine zweite Farbe (z.B. für Farbverläufe vom Horizont zum Himmel etc.), die in diesem Programm allerdings nicht genutzt wird.

struct lichtquelle:

Auch dies ist eine besondere Objektstruktur. Sie definiert die Position, Helligkeit und Farbe einer Lichtquelle. Sollte eine Lichtquelle einmal sichtbar sein, dann wird sie genauso gehandhabt wie eine einfarbige Kugel, deshalb die Radiusangabe. Die Distanzkonstante dist_konst ist die gleiche Konstante Kd, wie wir sie bereits bei den Abhandlungen über diffuse und spiegelnde Reflexionen kennengelernt haben. Sie dient dazu, sehr weit entfernte Lichtquellen trotzdem nicht all zu dunkel erscheinen zu lassen (Kd wird mit der Entfernung d der Lichtquelle zum jeweiligen Objekt multipliziert). Je kleiner also Kd, desto heller die Lichtquelle.

struct kugel:

Da ist sie! Die Definition einer Kugel mit Typ, Position, Radius und den diversen Materialeigenschaften.

struct flaeche:

Ja, und auch sie ist hier zu finden: ein Parallelogramm, definiert durch drei Eckpunkte und die Materialkonstanten. Die Ebenenstruktur innerhalb dieser Struktur wird erst später initialisiert. Die Ebenenformel berechnet das Programm dann aus den drei Eckpunkten.

struct beobachter:

Damit lassen wir die Objekte hinter uns und wenden uns dem Beobachter zu, dessen Position, Blickrichtung usw. ja ebenfalls bekannt sein müssen. Angegeben werden müssen:

- Position
- Blickpunkt, also der Punkt, auf den der Beobachter gerade schaut. Damit können Sie ein Objekt direkt anvisieren.
- Oder Sie geben den Blickvektor an, also die Blickrichtung.
- Der Abstand zur Mattscheibe (normalerweise gleich 1, kann aber auch andere Werte annehmen, wodurch der Sichtwinkel zusätzlich verkleinert oder vergrößert wird. Er bestimmt sozusagen die Brennweite des Objektives.

- Die Punktbreite und -höhe eines Punktes auf dem Bildschirm oder dem Gerät, auf dem die Grafik ausgegeben werden soll, in beliebigen Einheiten (xpg und ypg).
- Die Anzahl der zu zeichnenden Punkte in x- und in y-Richtung
- Der horizontale Sichtwinkel. Mit ihm bestimmen Sie den Bildausschnitt, den Sie sehen wollen (ebenfalls eine Art, die Brennweite anzugeben).

Die anderen Parameter errechnet sich das Programm später aus den angegebenen Werten. Ein Wort zum Blickpunkt bzw. Blickvektor: Auf irgendeine Weise müssen Sie dem Computer klarmachen, in welche Richtung der Beobachter schaut (nach oben, unten, rechts oder links etc.). In unseren mathematischen Ableitungen sind wir dabei von einem bestimmten Punkt ausgegangen, den er anvisiert. Damit sind wir in der Lage, zu verhindern, daß wir an einem bestimmten Objekt »vorbeisehen«. Da sich allerdings bei dieser Angabeart die Blickrichtung und damit die Perspektive ändert, sobald der Beobachter verschoben wird, ist es oft günstiger, die Blickrichtung als Sichtvektor anzugeben (er berechnet sich aus der Differenz des Blickpunktes und der Beobachterposition, s.o.). Sie haben also die Wahl: Entweder Sie verwenden den Blickpunkt als Richtungsangabe, dann setzen Sie den Blickvektor auf Null, oder Sie geben den Blickvektor an. Das Programm versteht beides.

Mit der Angabe der Auflösung (bzw. der Anzahl der zu zeichnenden Punkte in x- und y-Richtung) bestimmen Sie, wie groß Ihr Bild auf dem Bildschirm wird. Ein kleines Bild ist natürlich erheblich schneller berechnet und eignet sich also für Testzwecke besser als die volle Bildschirmgröße, an der der Computer eventuell mehrere Stunden rechnet.

struct welt:

Die letzte Struktur hält die ganze Welt zusammen. Hier liest der Rechner nach, wie viele Objekte, wie viele Lichtquellen, Kugeln und Flächen sich in der Welt befinden und wo er ihre Definitionen findet. Alle Kugeln (ebenso alle Lichtquellen und alle Flächen) sind beispielsweise in einem Array gespeichert, das aus lauter Kugelstrukturen besteht. Die Startadresse dieses Arrays steht hier. Hier geben Sie auch an, wie hell das Streulicht, also die Hintergrundbeleuchtung sein soll, d.h. wie hell ein Objekt noch sein soll, das sich im Schatten befindet.

Puh, nun langt es aber! Immerhin haben Sie nun einen grundlegenden Einblick in die Struktur dieser synthetischen Welt erhalten. Jetzt geht es ans Eingemachte.

Dreh- und Angelpunkt aller Rechnungen und Mißrechnungen des Computers, aller Flüche und Schimpfkaskaden des Programmierers ist die Hauptroutine **do_program()**. Hier befinden sich die zwei ineinandergeschachtelten FOR-Schleifen, die jeden einzelnen Punkt des Bildschirms »in Auftrag geben«, die von jedem Punkt die Farbe berechnen und sich dann nicht einmal das Vergnügen nehmen lassen, den Punkt dann eigenhändig auf den Monitor zu bringen.

Doch vor dem hat der Programmierer die Initialisierungen gesetzt! Sie manifestieren sich in den Routinen init_welt( ), init_farben( ) und get_konst( ), die wir wohl oder übel über uns ergehen lassen müssen. Also dann...

init_welt( ) richtet alle Strukturen so ein, wie sie das Programm gerne vorfinden möchte. Im wesentlichen klinkt das Unterprogramm die Daten ein, die Sie für die einzelnen Objekte, Lichtquellen, den Beobachter etc. am Programmende eingegeben haben. Lassen Sie uns darauf später zurückkommen.

Machen wir also direkt weiter bei **init_farben( )**. Aber auch die Routine lassen wir erst einmal links liegen. Sie bereitet die Farbpalette für die Punkt-Setz-Funktion vor und ist zunächst nebensächlich.

Aber jetzt! **get_konst( ):** Mittelpunkt sind zwei Pflichtübungen, die mit der Beobachterstruktur zu tun haben: die Berechnung des Sichtvektors $\vec{s}$ vom Beobachter zur Mattscheibe, Sie erinnern sich? Schauen Sie dazu doch noch einmal ein paar Seiten weiter zurück. Am Anfang des Kapitels mußten Sie die mathematische Ableitung über sich ergehen lassen. Er berechnet sich hier entweder aus dem Blickpunkt oder – sofern angegeben – aus dem Blickvektor. Hier sehen Sie zum ersten Mal die Anwendung der #define-Makros: VEKLET( ), VEKSUB( ), VEK_LAENGE( ) und MULVEK( ). Machen Sie sich die Funktionsweise an dieser Stelle beispielhaft noch einmal genau klar, sie sind in Zukunft unser wichtigstes Arbeitszeug. Das Ergebnis, den normierten Sichtvektor, speichert die Routine dann in die noch nicht belegten Plätze der Beobachterstruktur (beo ist hier ein Pointer auf die Beobachterstruktur). Auf ein Element muß dann also entweder so:

```
(*beo).blickv[0]
```

oder kürzer so:

```
beo->blickv[0]
```

zugegriffen werden.

Die Berechnung der Schrittvektoren $\vec{xe}$ und $\vec{ye}$ hat Sie bereits oben in Atem gehalten, ersparen Sie uns also die Einzelheiten, da Sie sie einfach nachlesen können. Die Vektoren werden gleich mit der jeweiligen Punktgröße (xpg, ypg) multipliziert, so daß sich ein Punkt Pm auf der Mattscheibe mit den Bildschirmkoordinaten x,y dann einfach durch die bekannte Formel ermittelt:

$$P_m = B + \vec{s} + x*(xpg*\vec{xe}) + y*(xpg*\vec{ye})$$

Und das geschieht dann auch in der Funktion get_punkt( ).

Damit sind Sie bereits mitten in den beiden FOR-Schleifen von **do_program( )**. Nachdem nämlich die Raumkoordinaten des Mattscheibenpunktes bekannt sind, sollte es kein Problem sein, den Sichtstrahl zu berechnen. Den Strahl (bzw. die Gerade) also, der in die Weite der Welt hineinragt und der darauf getestet werden muß, ob er ein Objekt schneidet. Hier geht uns ein weiteres Makro GET_GERADE( ) zur Hand, das die Geradengleichung aus zwei Punkten ermittelt und direkt in die Geradenstruktur des Sichtstrahles einschreibt.

Hernach erfolgen die Initialisierung der sogenannten Intensitätsbeizahl, die lediglich für die Begrenzung von Mehrfachspiegelungen herangezogen wird, und des Punkt-Farbwertes, der ja erst noch ermittelt werden soll.

Und dann geht es schon los mit der Strahlverfolgung: **ray_trace( )**. Diese Routine ermittelt auf kurz oder (sehr) lang die exakte Farbe des Mattscheibenpunktes auf sieben Stellen hinter dem Komma genau im Bereich von null bis eins. Die Farbe setzt sich aus den Intensitätswerten für jede der drei Grundfarben Rot, Grün und Blau zusammen. Diesen ermittelten Farbwert (er befindet sich in der Punktstruktur) übergibt do_program( ) der Routine **plot( )**, die sich alle Mühe gibt, einen Punkt in etwa dieser Farbe auf den Bildschirm zu bringen (bei maximal 64 Farben ist das schon ein Problem). Haben wir diese Aktion auch hier erfolgreich abgeschlossen, dann testet die Routine noch kurz auf einen Tastendruck (Sie können ja stets per <Esc> unterbrechen) und nimmt sich dann den nächsten Punkt vor.

ray_trace( ):

ray_trace( ) ist wohl eine der wichtigsten Funktionen im ganzen Programm überhaupt, da sie einen Strahl »verfolgt« und alle Schnittpunkte dieses Strahles mit allen Objekten der Welt berechnet. Sie wählt sich den nächsten Schnittpunkt (zum Fixpunkt des Strahles) aus und ermittelt dann die Farbe dieses Objektpunktes. Sie ist auch deshalb sehr wichtig, da sie bei verspiegelten Objekten rekursiv aufgerufen werden kann.

Wie viele zentrale Funktionen ist auch **ray_trace( )** recht klein und überschaubar, da sie meist nur als »Schaltstelle« fungiert, die eigentlichen Aufgaben und die »Details« also anderen Routinen überläßt, die sie aufruft. So muß ja zunächst einmal der nächste Schnittpunkt ermittelt werden. Diese Aufgabe übernimmt die Funktion **schnitt_objs( )**, die den Objekttypen des nächsten Objektes übergibt und den s-Wert der Geradengleichung, an der der Schnittpunkt liegt. Berechnet wird also noch nicht unbedingt der Schnittpunkt selbst in Koordinatenform, sondern nur als s-Parameter (Verlängerungsfaktor) für die vektorielle Geradengleichung (aus dem der Punkt natürlich jederzeit ermittelt werden kann). Mit diesen Informationen kann ray_trace( ) nun gezielt die Routinen anspringen, die die Objektfarben an diesem Punkt ermitteln: hintergrundfarbe( ), lichtfarbe( ), kugelfarbe( ), flaechenfarbe( ).

schnitt_objs( ):

Die Zentrale der Schnittpunktberechnungen schnitt_objs( ) ist selbstverständlich wieder nur eine Schaltstelle für die einzelnen, sehr unterschiedlichen Routinen, die jeweils den Schnittpunkt einer Kugel oder einer Fläche etc. mit dem Sichtstrahl ermitteln: schnitt_kugel_a( ), schnitt_flaech( ). Für jeden Objekttyp ist hier eine FOR-Schleife eingerichtet, die nach und nach alle im Objektarray gespeicherten Objekte abklappert. Falls es einen Schnittpunkt mit dem jeweiligen Objekt gibt, dann wird geprüft, ob er (repräsentiert durch den s-Wert) näher am Beobachter liegt als der bisher nächste Schnittpunkt. Ist das der Fall, dann speichert die Routine das s und die Adresse auf die jeweilige Objektstruktur (eigentlich der Pointer auf das erste WORD der Objektstruktur, den Objekttyp) und fährt mit dem nächsten Objekt fort. Bei den Flächen werden zusätzlich noch die dort bereits berechnet vorliegenden Schnittpunktkoordinaten sowie die m- und k-Werte der Ebenengleichung (also die Position des Schnittpunktes auf der Ebene) registriert, die später noch benötigt werden (es soll ja nichts doppelt berechnet werden).

Sind auf diese Weise alle Objekte der Welt überprüft und hat sich kein einziger Schnittpunkt ergeben, dann muß wohl oder übel der Hintergrund einspringen. Die Routine gibt mit der etwas monströs aussehenden Anweisung

```
return (**objekt);
```

den nächsten Objekttypen (der ja in jeder Objektstruktur steht) zurück, speichert noch schnell s in der Punktstruktur und verabschiedet sich.

Die Schnittpunktberechnungen:

schnitt_kugel_a( ):

Viel brauchen wir hierzu nicht zu sagen, da bereits alles in den theoretischen Abhandlungen erläutert wurde. Die Routine berechnet den (die) Schnittpunkt(e) zwischen dem übergebenen Strahl und einer ebenfalls angegebenen Kugel, überprüft jedoch gleichzeitig, ob das gefundene kleinere s (es existieren ja immer zwei Schnittpunkte – wenn sie existieren – den Fall einer Berührung einmal beiseite gelassen) größer Null ist (besser: größer fast-null wegen der Rechenungenauigkeit). Ist das nicht der Fall, so berechnet sie das größere s als Schnittpunkt (man befindet sich dann also als Beobachter in der Kugel).

schnitt_flaech( ):

Auch hierzu ist nicht viel zu erzählen. Die mathematischen Grundlagen lesen Sie am besten weiter oben nach. Bemerkenswert ist vielleicht noch eine Stelle ganz zu Anfang der Routine. Hier überprüft das Programm, ob der Normalenvektor der Ebene, der hier ja benötigt wird, bereits in der Ebenenstruktur der Fläche eingetragen ist. Ist das nicht der Fall, berechnet das Programm ihn noch schnell durch ein Vektorprodukt, speichert ihn in der Ebenenstruktur und gibt ihn als gültig für alle Zukunft frei (n_valid). Für diese Fläche braucht also nie wieder der Normalenvektor ermittelt werden!

Gehen Sie jetzt bitte zurück zur Funktion **ray_trace( )**. Die Schnittpunktberechnungen haben wir hinter uns gelassen. Das naheliegendste Objekt ist ermittelt und steht mittlerweile in **obj_typ**. Nun gilt es, die Farbe des Schnittpunktes mit dem Objekt zu bestimmen. Je nachdem, um welches Objekt es sich handelt, springt die Routine nun per SWITCH-Anweisung zu den Farbberechnungsroutinen der einzelnen Objekttypen:

Farbberechnungen:

Bei den Farbberechnungen der einzelnen Objekte dürfen Sie nun voll Ihre Phantasie spielen lassen. Was Ihnen gerade einfällt an Oberflächenstrukturen, Farbverläufen oder sonstigen »Spielereien«.

hintergrundfarbe( ):

Na, endlich mal wieder etwas Einfaches. Die Hintergrundfarbe braucht nämlich nur aus der Hintergrundstruktur übernommen werden. Sie könnten an dieser Stelle vielleicht einen kleinen Farbverlauf o.ä. programmieren. Da die Berechnung der Hintergrundfarbe so ausgesprochen

schnell vonstatten geht, werden diejenigen Passagen, auf denen nichts zu sehen ist, ebenfalls besonders schnell gezeichnet (Sie können das leicht am Bildschirm nachverfolgen).

lichtfarbe( ):

Auch das ist kein Problem. Im Licht kann sich nichts spiegeln, verzerren oder sonstiges.

kugelfarbe( ):

Hat der Strahl eine Kugel getroffen, dann wird es schon etwas kostspieliger, ein schönes Bild zu schaffen. Dann nämlich werden Sie sehen, wie zum ersten Mal die Formeln für die diffuse und spiegelnde Reflexion eingesetzt werden. Aber dazu benötigt man zum einen die Koordinaten des Schnittpunktes, zum anderen den Normalenvektor, also den Vektor senkrecht zur Kugeloberfläche, wenn Sie sich erinnern. Also berechnet Ihr Amiga folgerichtig die notwendigen Teile: den Schnittpunkt aus der Geradengleichung des Strahles und dem Verlängerungsfaktor s (aus den Schnittpunktberechnungen), die Normale als Vektor vom Kugelmittelpunkt zum Schnittpunkt.

Ferner ermittelt er die Grundfarbe des Objektes, die ja durch die Reflexionsformeln zu verändern ist. Mit diesem Gepäck startet die Routine kugelfarbe() die zentrale Funktion zur Berechnung der Punktintensität farbintens( ). Da diese Routine nicht von objektspezifischen Eigenarten abhängig ist (Bezugsgröße sind ja nur Schnittpunkt, Normale, Punktfarbe und die neutralen Materialkonstanten selbstverständlich), wird sie von allen Objekten (außer Lichtquellen und Hintergrund natürlich) angesteuert. Hier »treffen« sie sich also wieder.

flaechenfarbe( ):

Natürlich ist das Ziel dieser Routine identisch mit dem der soeben besprochenen: Schnittpunkt, Normale, Punktfarbe. Trotzdem ist sie um einiges länger. Das liegt an den vielen zusätzlichen Oberflächeneffekten, die bei diesen Flächen angewählt werden dürfen: Normal einfarbig, zweifarbig kariert und zweifarbig mit einem beliebigen Muster versehen. Das hört sich doch schon einmal gut an, oder?

Der Schnittpunkt des Strahles mit der Fläche ist bereits bekannt! Weiter oben bei der Schnittpunktberechnung wurde er bereits einmal ermittelt und steht uns seitdem zur Verfügung – eine Arbeit weniger. Auch der Normalenvekor stellt nun kein Problem mehr dar, auch er wurde dort bereits berechnet (er steht nun in der Ebenenstruktur der Fläche).

Bleibt nur noch die Bestimmung der Grundfarbe des Objektes. Dafür aber wurden vom Programmierer drei Fälle per SWITCH unterschieden. In der Materialstruktur gibt es nämlich ein Byte mit dem Namen oberf_typ. Hier legen Sie eine einfache Zahl ab für die Art und Weise der Oberflächengestaltung:

oberf_typ = 0:

Das ist der einfache, der bekannte Fall: Die Routine nimmt sich die Grundfarbe des Objektes direkt aus der Farbanweisung der Materialstruktur, das Objekt ist einfarbig.

oberf_typ = 1:

Eine Eins in besagtem Byte gibt an, daß das Objekt (hier also die Fläche) kariert sein soll. Die zwei Farben für die Karos stammen dann aus den ersten beiden Farbregistern der Materialstruktur. Zusätzlich geben Sie in der Musterstruktur innerhalb der Materialstruktur noch zwei Werte an: die Anzahl der Karos in x- und die in y-Richtung bezüglich des Flächennullpunktes (P0), mit anderen Worten in m- und k-Richtung, wenn wir einmal von der vektoriellen Ebenengleichung ausgehen.

Jetzt braucht nur noch ermittelt zu werden, in welchem Karo (Farbe 1 oder Farbe 2) sich der Punkt gerade befindet, und die Farbe ist gewonnen.

oberf_typ = 2:

Das ist eine ganz trickreiche Sache! Bei dieser Einstellung können Sie ein 16x16-Punkte-Muster selbst definieren und Ihr Objekt bekommt auf einmal Streifen, Punkte, oder auch kleine Beschriftungen! Gleichzeitig stellen Sie ein, wie oft sich das Muster auf der Fläche (wieder in m- und k-Richtung) befinden soll. Sie dürfen sich also entscheiden, ob Ihre Amiga-Aaas richtig groß und protzig oder nur klein und dürr auf Ihrer Tapete erscheinen sollen. Auch hier stammen die beiden zur Verfügung stehenden Farben aus den beiden ersten Farbregistern.

farbintens( ):

Wie bereits erwähnt, treffen sich hier alle Routinen wieder, die die Farbe eines Objektes bzw. eines Objektpunktes berechnen sollen. farbintens( ) verlangt die Angabe des Normalenvektors, des Sichtstrahles, der Punktstruktur (mit den Koordinaten des Punktes und dem Intensitätsbeiwert), der Materialstruktur des Objektes und natürlich der Weltstruktur (alles in Pointern).

Die erste Entscheidung, die das Programm zu fällen hat, ist die Frage, ob das Objekt verspiegelt ist oder nicht. Ist das tatsächlich der Fall (die Variable **licht_typ** in der Materialstruktur entscheidet das), dann wird die Spiegelroutine eingeleitet. Im Endeffekt besteht sie darin, den Reflexionsstrahl zu berechnen (Einfallswinkel gleich Ausfallswinkel, Sie kennen das aus den mathematischen Ableitungen) und dann eine Rekursion einzuleiten, indem die Routine **ray_trace( )** mit dem Reflexionsstrahl als Sichtstrahl erneut aufgerufen wird. Das heißt also, es wird das nun naheliegendste Objekt gesucht, das natürlich wieder verspiegelt sein kann usw. Das darf natürlich nicht unendlich so weitergehen. Stellen Sie sich einfach einmal zwei verspiegelte Flächen vor, die parallel zueinander verlaufen. Jeder Spiegel würde sich unendlich oft in dem anderen spiegeln. Unser Programm »hängt sich auf«.

Benötigt wird also ein Abbruchkriterium für diese Rekursion. Und dazu muß der sogenannte Intensitätsbeiwert herhalten. Für jedes verspiegelte Objekt müssen Sie eine Spiegel-Intensitätszahl (spiegel_int[ ], von 0 bis 1) in der Materialstruktur angeben. Sie gibt an, wieviel Prozent des Lichtes von der verspiegelten Fläche tatsächlich gespiegelt wird (kein Spiegel reflektiert nämlich das gesamte einfallende Licht!). Diese Zahl (jeweils für RGB – es können also auch »farbige« Spiegel programmiert werden) nimmt das Programm dann mit dem aktuellen Intensitätsbeiwert mal, der damit also von Spiegelung zu Spiegelung kleiner wird

(es sei denn, Ihre Spiegel reflektieren hundertprozentig oder gar mehr). Sobald dieser Wert unter eine bestimmte Grenze fällt **(schwarz),** endet die Rekursion, da nicht mehr genügend Licht zum Spiegeln vorhanden ist.

Um die Zahl der Rekursionen aber in jedem Fall unter einem bestimmten Wert zu halten (sonst läuft ganz sicher irgendwann der Stack über und Ihr Amiga stürzt ab), zählt das Programm gleichzeitig die Zahl der Rekursionen in der globalen Variable rekur_zaehl. Ist rekur_zaehl größer als REKUR_MAX, die maximal erlaubte Anzahl an Rekursionen (hier 7), dann wird ebenfalls abgebrochen.

Beachten Sie, daß zur **Rekursion** in der Routine **farbintens( )** eine nagelneue lokale Punktstruktur und eine ebenso nagelneue und lokale Geradenstruktur eingerichtet wird! Schließlich müssen die alten Strukturen auch noch über die Rekursion hinweg erhalten bleiben.

Angenommen, die rekursiv aufgerufene Routine **ray_trace( )** kehrt ordnungsgemäß zurück (was ja hoffentlich immer passiert!). Sie hat dann die Farbe des Punktes auf dem nächsten Objekt errechnet und in der Punktstruktur abgelegt. Diese (neue) Punktstruktur wird nun nicht mehr benötigt. Lediglich der errechnete Farbwert muß in die alte übertragen werden, da mit ihm ja noch weitergerechnet werden soll. Das geschieht mit dem COLLET-Makro.

An dieser Stelle treffen sich die beiden Fälle (verspiegelt oder nicht verspiegelt) wieder. Jetzt beginnt nämlich die Berechnung der spiegelnden und diffusen Reflexion. Dabei berücksichtigt das Programm in einer großen Schleife das Licht, das von allen Lichtquellen eventuell auf das Objekt fällt. Zunächst aber muß getestet werden, ob der zu berechnende Punkt überhaupt von der jeweiligen Lichtquelle angeschienen wird. Es kann ja sein, daß sich irgend ein anderes Objekt zwischen Punkt und Lichtquelle geschoben hat und den armen Punkt im wahrsten Sinne des Wortes ein Schattendasein fristen läßt.

Wieder ist ein neuer Strahl zu berechnen, diesmal vom Punkt zur Lichtquelle (die Lichtquelle wird hier der Einfachheit halber als punktförmig idealisiert, obwohl sie ja einen Radius besitzt). Was ist nun zu tun? Die Frage lautet: Gibt es ein Objekt, das zwischen Punkt und Lichtquelle steht? Oder für unsere Zwecke ausgedrückt: Gibt es ein Objekt, das von dem Strahl zur Lichtquelle geschnitten wird?

Die Antwort auf diese Frage liefert uns wieder die globale Routine zur Berechnung des nächsten Schnittpunktes **schnitt_objs( )**, die bereits oben beschrieben wurde. Das Ergebnis dieser Funktion werten wir diesmal allerdings ein wenig anders aus. Die Routine wird nämlich in jedem Fall mindestens einen Schnittpunkt ermitteln, nämlich den mit der angepeilten Lichtquelle (so wurde der Strahl schließlich konstruiert). Findet die Routine genau diese Lichtquelle als naheliegendstes Schnittobjekt, dann ist die Sache geritzt, der Punkt liegt also nicht im Schatten, die Berechnung der Reflexionsformel kann beginnen. Findet schnitt_objs( ) allerdings ein anderes Objekt (festzustellen an der ermittelten Objektadresse), dann wird der Punkt »beschattet«, erhält von der momentan zur Debatte stehenden Lichtquelle also kein Licht.

Die Formel zur Berechnung der beiden Reflexionsarten kennen Sie bereits. Sie muß selbstverständlich schön separat und getrennt für jede der drei Grundfarben Rot, Grün und Blau ermittelt werden. Halten wir uns also nicht länger damit auf.

Das Programm arbeitet auf diese Weise alle Lichtquellen ab. Bleibt nur noch die Hintergrundhelligkeit (Streulicht) Ih*Koh. Sie wirkt ja auf jeden Punkt ein und muß damit am Ende noch hinzuaddiert werden. Die Farbe des Punktes ist perfekt und die Routine hat ein Ende.

plot( ):

Tja, da sind wir nun wieder in der guten alten Hauptschleife von **do_program( )**. **ray_trace( )** haben Sie genauso wie der Autor hinter sich gelassen. Vor uns liegt die Krönung aller komplizierten Rechnungen: plot( ), die Punktausgabe. Sie hat dem Programmierer noch das meiste Kopfzerbrechen beschert, da er mit den wenigen zur Verfügung stehenden Farbregistern auskommen mußte, obwohl die Farbe des Punktes tatsächlich auf sieben Stellen hinter dem Komma genau berechnet vorliegt, zahlreiche Schattierungen ein und derselben Farbe berücksichtigt werden müssen und trotzdem noch ein möglichst wirklichkeitsnahes Bild erzeugt werden sollte. Ich bin mir darüber im klaren, daß die hier gefundene Lösung nicht unbedingt die beste sein muß. Vielleicht haben Sie ja die Hyper-Idee und schreiben mir mal? Diese Routine ist aber auch die einzige, die bei einem Wechsel auf einen anderen Computer verändert werden müßte (evtl. noch init_farben( )). Alle anderen sind praktisch für einen idealen Computer mit mehreren Millionen Farben (gleichzeitig) und fast beliebig großer Grafikauflösung konzipiert, für einen Computer vom Schlage einer Cray II vielleicht?

Genau in dieser Routine sehen wir uns durch die Fähigkeiten des Rechners stark eingeschränkt, aber machen wir das Beste daraus!

An dieser Stelle muß wohl oder übel ein Wort über die Speicherung der verschiedenen Farben fallengelassen werden, wovor wir uns eben bei der Vorstellung von **init_farben( )** gedrückt haben. Im Datenteil des Programmes, also ziemlich am Ende, haben Sie die Möglichkeit, die Farbzuteilung der Palettenregister vorzunehmen. Hier geben Sie ganz normal für jede Grundfarbe Werte von 0 – 15 an. In init_farben( ) speichert das Programm diese Werte dann in die Palettenregister des Rechners und multipliziert die Werte dann aber mit $2^{16} = 65536$. Das hat den Zweck, aus Geschwindigkeitsgründen später auch Nachkommastellen der Farbwerte in einer Integer-Variablen zu speichern. Gleichzeitig werden in init_farben( ), für den Fall, daß Sie den Modus EXTRA_HALFBRITE gewählt haben, die entsprechenden Farbwerte für die Farben 32 – 63 errechnet. Dieses Format ist natürlich nur das programminterne, das für den Amiga später erst wieder rückgerechnet werden muß. Das klingt alles zwar etwas kompliziert und umständlich, Versuche mit Floatingpointzahlen allerdings brachten erhebliche Geschwindigkeitseinbußen.

Schauen Sie jetzt nach **plot( )**. Hier kommt in x,y die Bildschirmkoordinate an, an der der Punkt gesetzt werden muß (mit dem Nullpunkt links oben in der Ecke). Ferner wird der Funktion die Punktstruktur übermittelt, aus der sie die exakte Farbe des Punktes »extrahiert«. Sie muß in einer ersten Phase erst einmal für R, G und B zwischen Null und Eins gebracht werden, da es durchaus einmal passieren kann, daß hier vor allem größere Werte übermittelt werden. Gleichzeitig rechnet die Routine die Floatingpointwerte in das gerade beschriebene Farbformat um. Jetzt sehen Sie, daß auch die Stellen hinter dem Komma nicht vergessen werden!

Der folgende Schritt ist nun ein wenig kompliziert. Gesucht ist das Palettenregister, das der Farbe des Punktes am nächsten kommt. Dazu durchsucht der Rechner alle zur Verfügung

stehenden Farben (32 oder 64) und ermittelt die beiden Register, bei denen die Differenz aus Registerfarbe und tatsächlicher Punktfarbe am niedrigsten (**reg_min**) bzw. am zweitniedrigsten (**zweit_reg_min**) ist.

Normalerweise würde man nun sagen: reg_min ist das Register mit der Farbe, die der Punktfarbe am ähnlichsten ist, also sollte der zu zeichnende Punkt auch die Farbe aus reg_min erhalten. Im Idealfall haben Sie recht. Sie werden allerdings feststellen – falls Sie sich die Mühe machen und das einmal ausprobieren –, daß jedes Objekt dadurch äußerst unangenehme Ringe erhält, die sich dort ausbilden, wo gerade ein Farbregister gewechselt wurde. Das liegt daran, daß wir den Unterschied zwischen z.B. Rot-Intensität 13 und Rot-Intensität 12 durchaus noch deutlich erkennen.

Die Lösung: Die Grenzbereiche zwischen den Bereichen mit Rot 12 und Rot 13 müssen etwas »ausgefranst«, der Übergang also etwas »milder« gestaltet werden. Das erreichen wir mit einer kleinen Zufallsfunktion zufall( ). Sie produziert bei jedem erneuten Aufruf eine (scheinbar) zufällige Zahl zwischen 0 und 256 ($00-$FF). Die Größe dieser Zahl bestimmt nun, ob der Punkt mit Rot 12 oder Rot 13 geplottet werden soll. Die Wahrscheinlichkeit für Rot 12 nimmt aber stetig ab, je weiter wir in den Bereich von Rot 13 hineingeraten und umgekehrt. Das wird mit der kleinen Formel in dem IF-Statement erreicht:

$$gz = z + \frac{zs * mz}{(zs-es)} * wk$$

es = der Farbunterschied zwischen der erstbesten Farbe und der exakten Farbe des Punktes
zs = der Farbunterschied zwischen der zweitbesten Farbe und der exakten Farbe des Punktes
mz = maximale Größe der Zufallszahl (hier 256)
wk = Wichtungskorrektur
z = Zufallszahl (von 0 bis mz)
gz = gewichtete Zufallszahl (von 0 bis 2 * mz)

»es« und »zs« hat die Routine bereits mit dem besten bzw. zweitbesten Register ermittelt. mz ist hier gleich 256 (eigentlich gleich 255, aber wer wollte es beim Zufall schon so genaunehmen?), die größte mögliche Zufallszahl. z ist die Zufallszahl selbst, sie wird bekanntlich von **zufall( )** geliefert. Die »Wichtungskorrektur« wk wurde mit eingeführt, um die Wahrscheinlichkeit ein wenig in Richtung Erstfarbe zu verschieben. wk bestimmt also die Breite des ausgefransten Bereiches. gz schließlich ist die gesuchte, durch die Formel gewichtete Zufallszahl, die irgendwo zwischen null und 512 liegt.

Ist gz nun größer als 255, dann wird in der besten Farbe gezeichnet, bei gz $<=$ 255, tritt die zweitbeste Farbe in Aktion. Das Ergebnis ist dann zwar ein gepunktetes Bild, das läßt sich allerdings kaum vermeiden (höchstens durch die Erhöhung der Auflösung, z.B. per Interlace).

In Abbildung 6.9 können Sie sich noch einmal den groben Ablaufplan des Ray-Tracing-Programmes anschauen.

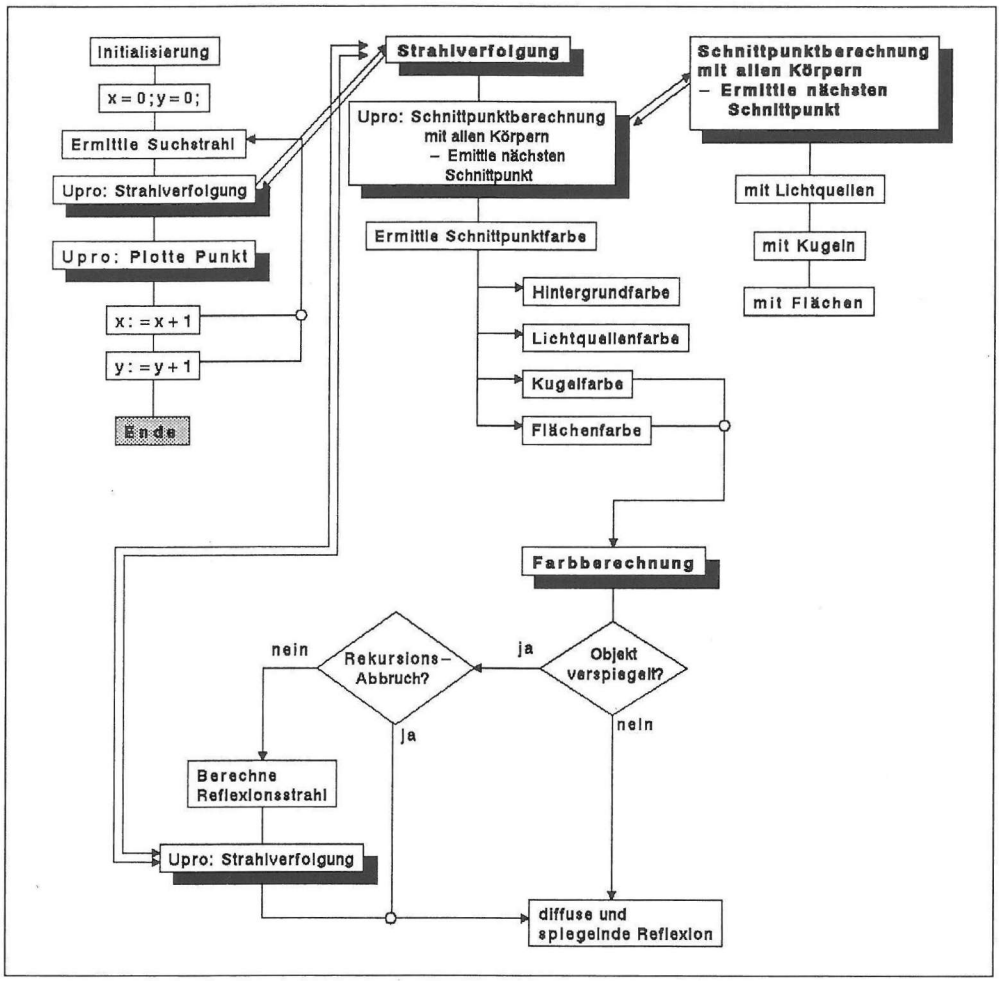


Bild 6.9: Ablaufplan Ray-Tracing-Programm

Was bleibt? Das Ändern der Welt natürlich, was auch sonst! Das geht hier allerdings viel einfacher als in der Realität. Schauen Sie sich jetzt einmal die Datenbereiche am Ende des Programmes an. Dort finden Sie alle notwendigen Informationen für die gesamte Szenerie, angefangen bei den Palettenfarben über die Beobachterwerte, bis hin zu den Lichtquellen, Kugeln und Flächen. Die Funktion der verschiedenen Parameter sollte Ihnen vom Theoretischen her nun bereits bekannt sein. Welche Auswirkungen und Effekte in der Praxis nachher bei den unterschiedlichsten Konstellationen resultieren, das ist reine Probiersache. Das Gefühl dafür bekommen Sie erst im Laufe der Zeit. Beginnen Sie einfach bei simplen und überschaubaren

Objekten. Sollten Sie eine ganze Szenerie im Auge haben, dann ist eine Planung auf dem Papier anzuraten. Ein Tip zum Ausprobieren: Ändern Sie die Farbe 1 der gelb und schwarz gestreiften Tischplatte in dem Beispiel einmal in Weiß und stellen Sie gleichzeitig »VERSPIEGELT« ein mit z.B. 60–70 Prozent (0.6–0.7) Spiegelintensität. Der Effekt gleicht einer glatt polierten Tischoberfläche, in der sich die Dinge spiegeln, in der aber auch das Streifenmuster noch erkennbar ist! Bei jeder Veränderung der Welt sollten Sie aber im Auge behalten, welche Farben mit der momentan eingestellten Farbpalette überhaupt möglich sind. Wenn Sie beispielsweise ein grünes Objekt implementieren wollen und es ist überhaupt kein Grün in Ihrer Palette, dann wählt die Routine plot( ) irgendeine andere Farbe, die ihrer Ansicht nach paßt. Das gleiche gilt bei Farbschattierungen etc. Sollte Ihr Bild einmal farblich nicht nach Ihrer Zufriedenheit entstehen, dann machen Sie sich also zunächst Gedanken über die Farbpalette. Dieser Punkt spielt auch dann eine Rolle, wenn Sie in HAM arbeiten.

Was bei diesem Programm eigentlich noch fehlt, ist eine Funktion, die eine ASCII-Definitionsdatei in die entsprechenden Programmstrukturen übersetzt. Im Moment müssen Sie bei jeder Weltveränderung neu kompilieren und linken. Im anderen Fall brauchen Sie z.B. mit dem guten alten ED nur eine Änderung in Ihrer ASCII-Datei vorzunehmen und dem Programm übergeben. Das wäre doch einmal eine interessante Sache!

Denkbar wäre auch ein kleiner Grafikeditor, mit dem Sie die Objekte als Drahtmodelle direkt auf dem Bildschirm positionieren können. Erst später könnte dann das eigentliche Ray-Tracing-Programm aufgerufen werden.

Oder fügen Sie neue Objekte hinzu, neue Oberflächenstrukturen, vielleicht schieben Sie ja mal eine Routine in farbintens( ) ein, die den Normalenvektor nach bestimmten Kriterien ändert. Damit könnten Sie eine rauhe oder buckelige Oberfläche simulieren etc. Ihrer Phantasie sind da überhaupt keine Grenzen gesetzt!

Und nun ans Werk! Ich bin gespannt, was demnächst an Demo-Ray-Tracing-Bildern über die Public-Domain-Kanäle fließen wird. Na, denn viel Spaß! – Und Schluß.

6.5 Durchsichtige Körper oder: Bekanntschaft mit dem Brechungsgesetz

Mit einer möglichen Eigenschaft von Körpern haben wir uns in all den Ausführungen noch nicht beschäftigt: Transparenz. Objekte aus Glas oder Acryl etc. können durchsichtig sein. Man kann Dinge sehen, die sich hinter diesem Objekt befinden. Je nach Grad der Durchsichtigkeit und der Dicke des durchsichtigen Objektes erscheinen sie milchig oder klar, original hell oder dunkler. Ist ein Objekt nur für bestimmte Farben durchsichtig (z.B. blaues Glas), dann erscheinen die dahinterliegenden Objekte nur in dem durchgelassenen Farbanteil (Blau).

Selbst wenn ein Ding hundertprozentig lichtdurchlässig ist, erkennen wir es dennoch. Das liegt an der sogenannten Lichtbrechung oder Refraktion. Sobald ein Lichtstrahl von einem Medium (Luft) in ein anderes Medium (Glas oder Wasser etc.) eintritt, verändert er seine Richtung. Die quantenmechanische Begründung sparen wir uns hier. Wir nehmen es als Tatsache hin.

Durch diese Lichtbrechung kann es passieren, daß wir hinter einem durchsichtigen Gegenstand an einer Stelle Dinge sehen, die sich eigentlich an einer ganz anderen Position befinden. Halten Sie einmal einen geraden Stab zur Hälfte ins Wasser. Sie werden meinen, der Stab sei an der Stelle, an der er ins Wasser taucht, leicht umgebogen. Linsen, wie sie in Fernrohren oder Foto-Objektiven Verwendung finden, arbeiten nach dem gleichen Prinzip.

Licht verändert also seine Richtung, wenn es in einen durchsichtigen Gegenstand eintritt. Es verändert aber ebenso seine Richtung, wenn es wieder hinaustritt. Das physikalische Gesetz, das dieses Phänomen beschreibt, nennt sich Brechungsgesetz und lautet (s. Bild 6.10):

$$n_1 * \sin(a_1) = n_2 * \sin(a_2)$$

n_1, n_2 Brechungsindizes der beiden Medien (Materialkonstanten) $n_1, n_2 \geqq 1$
a_1 Eintrittswinkel zwischen der Normalen $\vec{n}$ und dem eintretenden Strahl $\vec{S}$
a_2 Brechungswinkel zwischen der Normalen $\vec{n}$ und dem gebrochenen Strahl $\vec{Sb}$

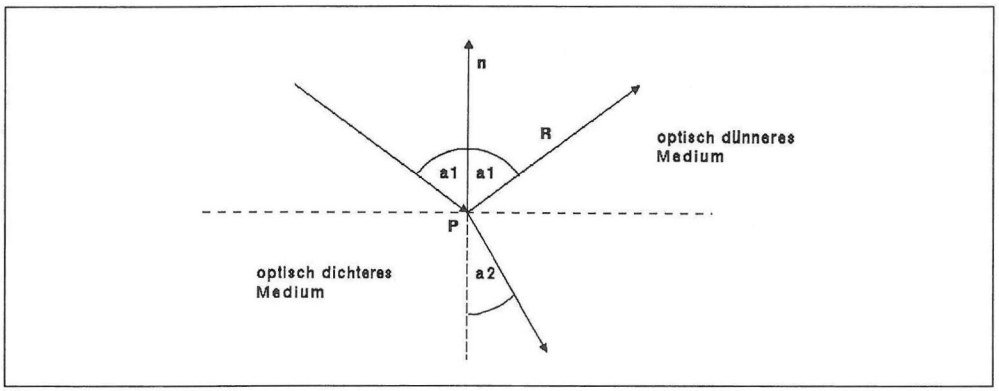


Bild 6.10: Das Brechungsgesetz

Tritt ein Lichtstrahl von einem optisch dünneren (z.B. Luft, kleinerer Brechungsindex, nl ist ungefähr gleich 1) in ein optisch dichteres Medium (z.B. Glas, größerer Brechungsindex), dann verläuft der gebrochene Strahl im zweiten Medium näher zur Senkrechten ($n_1 < n_2 \Rightarrow a_2 < a_1$). Im umgekehrten Fall, beim Übergang vom optisch dichteren zum optisch dünneren Medium dagegen, liegt er weiter vom Lot entfernt ($n_1 > n_2 \Rightarrow a_2 > a_1$). Im letzten Fall kann es dazu kommen, daß a_2 gleich 90 Grad wird ($\sin(a_2) = 1$). Da der Sinus nicht größer 1 werden kann, kommt es bei $a_2 > 90$ zu einer Totalreflexion (Spiegelung) des Strahles zurück ins dichtere Medium (also keine Brechung!)

Noch eine Sache ist wichtig: Nicht das gesamte Licht, das auf die Grenze zweier Medien fällt, tritt auch in das Medium ein (und wird dadurch gebrochen). Ein Teil wird ganz normal reflektiert, wie wir das bereits bei normalen Körpern kennengelernt haben. Je größer der Eintrittswinkel a_1 ist, desto mehr Licht wird auch reflektiert. Bei $a_1 = 0$ (senkrechter Eintritt) ist die Reflexion also gleich Null. Den Effekt können Sie auch in Ihrer Umgebung beobachten:

Normalerweise spiegelt sich in einer Fensterscheibe alles mögliche, so daß wir nur schwer hineinschauen können. Je senkrechter wir allerdings auf die Scheibe schauen, desto geringer werden die störenden Spiegelungen.

Ein durchsichtiger Körper weist also wie jeder andere Körper auch spiegelnde und diffuse Reflexionen auf. Das Maß der Reflexion hängt dann allerdings vom Eintrittswinkel des Lichtes ab.

Wir können unser Beleuchtungsmodell also noch um die Lichtbrechung bei durchsichtigen Körpern erweitern. Der Sichtstrahl muß nach den Regeln des Brechungsgesetzes (eventuell unter Berücksichtigung von Reflexion und Totalreflexion sowohl für Sichtstrahl als auch für eventuelle Lichtquellen) abgelenkt in den Körper eindringen. Beim Austritt sind die gleichen Gesetze zu berücksichtigen. Der so unter Umständen mehrmals abgelenkte Strahl kann dann wieder als ganz normaler Sichtstrahl behandelt werden. Auf diese Weise sind beispielsweise Vergrößerungs- oder Verkleinerungslinsen zu simulieren o.ä., wodurch besondere Effekte erzielt werden können.

KAPITEL 7

Noch eine Anwendung: Rotationskörper

Dieses Kapitel soll für eine kleine, aber interessante Anwendung zur dreidimensionalen Grafik Platz bieten. Einige Ideen kennen Sie bereits aus den vergangenen Kapiteln. Hier finden Sie zusätzliche Anregungen und Augenweiden.

Wir beschäftigen uns mit sogenannten Rotationskörpern. Dies sind teilweise komplizierte Gebilde, die aber relativ einfach zu konstruieren sind und deswegen in vielen 3-D-Systemen Verwendung finden.

7.1 Was sind Rotationskörper?

Bisher haben wir uns bei dreidimensionalen Darstellungen mit relativ einfachen Körpern beschäftigt. Das lag vor allem an dem Aufwand, den die Eingabe von vielen Eckpunkten und entsprechend vielen Linien macht. Die Darstellungen blieben oft recht eckig und kantig. Die Implementierung von runden Körpern (abgesehen einmal von der Ray-Tracing-Technik) macht normalerweise gehörige Probleme ob der Vielzahl von Ecken und Kanten.

An dieser Stelle möchten wir eine Technik einführen, die es auf einfachste Weise erlaubt, solche runden Körper zu produzieren. Beschränkt bleiben wir allerdings auf achsensymmetrische, sogenannte Rotationskörper, was allerdings keine besondere Einschränkung darstellt, da wir diese runden Körper ja mit eckigen Körpern kombinieren können.

Der Trick: Eingegeben werden nur wenige, ganz bestimmte Punkte, der Rest wird vom Programm aufgrund einer Rechenvorschrift für Rotationskörper errechnet.

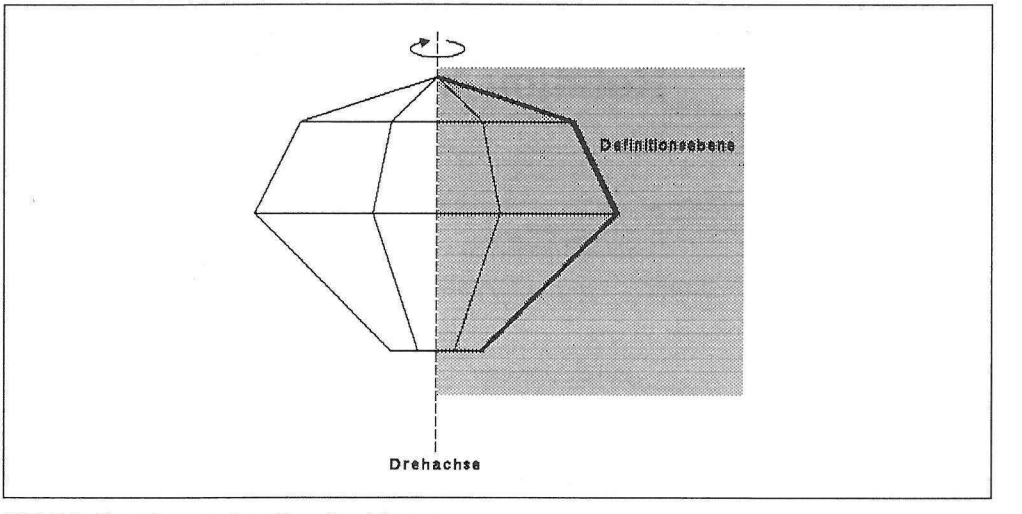


Bild 7.1: Generierung eines Rotationskörpers

Die Eingabe der notwendigen Definitionspunkte erfolgt in einem zwei(!)-dimensionalen Koordinatensystem, also in der x-y-Ebene mit z = 0. Hier können Sie nun ein beliebiges Gebilde mit beliebig vielen Punkten an ebenso beliebigen Stellen erschaffen. Jeder Punkt ist mit seinem Vorgänger durch eine Linie verbunden (s. Bild 7.1). Im einfachsten Fall verbinden Sie einfach zwei Punkte miteinander.

Um nun die Dimensionen der Ebene zu überwinden, rotieren wir die gesamte Zeichenebene mit einem beliebigen nicht zu großen und nicht zu kleinen Winkel um die y-Achse – jeder einzelne Punkt, jede Linie der Ebene wird mit rotiert. Die neuen Positionen der Punkte werden sich wie die alten in einer Punkteliste gemerkt. Auch die Linien müssen gespeichert werden. Im nächsten Schritt wird diese letzte Ebene um den gleichen Winkel noch einmal rotiert. Wieder speichert man die Punkte und Linien usw. Die ganze Operation wird so lange wiederholt, bis eine vollständige Umdrehung von 360 Grad durchgeführt wurde. Das Ergebnis ist der Eindruck eines runden Körpers, angenähert durch die vielen Drehungen.

Wenn nun noch jeder gedrehte Punkt mit seinem Ursprungspunkt verbunden wird, dann setzt sich der Körper aus vielen kleinen viereckigen Polygonen zusammen. Je kleiner die Drehschritte, desto runder erscheint das Gebilde.

Aus der Liste der Punkte und Linien generieren wir noch eine Flächenliste. Alle Listen werden zusammengelinkt zu einem Objekt, und schon können wir es haargenau so handhaben wie ein ganz normales dreidimensionales Objekt, wie wir es bereits kennengelernt haben: Das Objekt seinerseits kann skaliert, verschoben, rotiert, projiziert werden. Flächen können Flächen verdecken usw.

Das einzige Problem ist die Reservierung von genügend Speicherplatz, da sich die Anzahl der Punkte, Linien und Flächen ja summiert. Wenn w der Schrittwinkel zur Erzeugung des Rotationskörpers aus einer Ebene ist, P die Anzahl der Punkte in der Ebene, dann gilt für die Anzahl der zu speichernden Punkte, Linien und Flächen:

Punkte: Pges = 360/w * P
Linien: Lges = 360/w * (2*P–1)
Flächen: Fges = 360/w * (P–1)

w sollte immer ein Teiler von 360 Grad sein, ansonsten muß der Quotient 360/w aufgerundet werden. Bei einem Rotationsschrittwinkel von 10 Grad mit nur 6 Definitionspunkten in der Ebene hieße das:

Anzahl der Punkte: Pges = 216
Anzahl der Linien: Lges = 396
Anzahl der Flächen: Fges = 180

7.2 Die Anwendung für 3-D-Grafiken

Das folgende Basic-Programm erzeugt einen von Ihnen zu editierenden Rotationskörper und stellt ihn vollständig transformiert, projiziert mit verdeckten Flächen, auf dem Bildschirm dar. Dabei wird sogar eine Lichtquelle berücksichtigt, deren Position frei wählbar ist. Sie werden Ihre helle Freude haben:

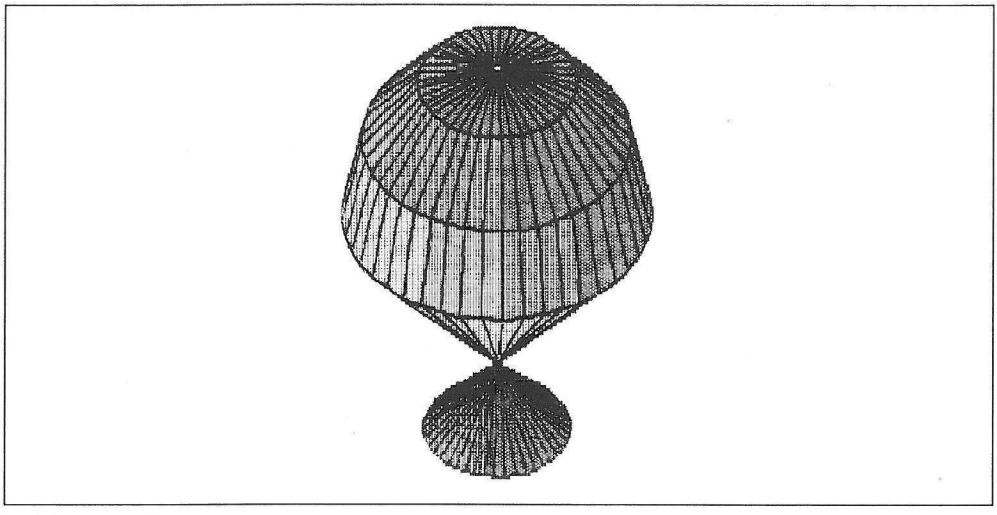


Bild 7.2: Rotationskörper 1

```
'************************
'**                    **
'**  Rotationskörper   **
'**                    **
'************************

'Speicherbedarfsrechnung:

'Maximalwerte je nach verfügbarem Speicherplatz einstellen:
DATA 15, 45

  READ MAX.ECKP%    'Maximale Zahl der Eckpunkte
READ MAX.ROTS%    'Maximale Zahl der Rotationen

'Speicherbedarf der Flächenarrays:
max.flaech& = MAX.ROTS% * (MAX.ECKP%-1) * 26
'Speicherbedarf der Punktearrays:
max.punkte& = MAX.ROTS% * MAX.ECKP% * 24
```

```
'Gesamter Speicherbedarf:
speicher& = max.flaech& + max.punkte& + 30000

  IF FRE(-1)+FRE(0) <= speicher& + 64000& THEN
  PRINT "Nicht genügend Speicherplatz!!!"
  PRINT : PRINT "zur Zeit notwendiger Platz:"
  PRINT "für Punkte:  ";max.punkte%
  PRINT "für Flächen: ";max.flaech%
  PRINT "gesamt:      ";speicher% : PRINT
  PRINT "Verringern Sie im Programm die maximalen Werte"
  PRINT "für die Zahl der Eckpunkte und/oder"
  PRINT "für die Zahl der Rotationen!"

  PRINT : PRINT "zur Zeit maximale Werte für:"
  PRINT "Eckpunkte:  ";MAX.ECKP%
  PRINT "Rotationen: ";MAX.ROTS%
  END
END IF

' Speicher reservieren:
CLEAR ,speicher&
RESTORE

READ MAX.ECKP%   'Maximale Zahl der Eckpunkte
READ MAX.ROTS%   'Maximale Zahl der Rotationen

PI = 3.141593

'Einen neuen Bildschirm mit Fenster öffnen:
anz.farben% = 16
SCREEN 2,640,200,4,2
wx% = 631
wy% = 186
WINDOW 2,"Rotationskörper",(0,0)-(wx%,wy%),4+8,2

blauf = 0     'Blaufaktor
gruenf = 0    'Grünfaktor
rotf = 1      'Rotfaktor
blauadd = .1  'Blau-Zusatz
gruenadd = .1 'Grün-Zusatz
rotadd = 0    'Rot-Zusatz

'Farben belegen:
FOR i=2 TO anz.farben%-1
```

```
    farbe = INT(i*100/15 + .5)/100
    rot   = farbe*rotf+rotadd
    gruen = farbe*gruenf+gruenadd
    blau  = farbe*blauf+blauadd
    IF rot>1 THEN rot=1
    IF gruen>1 THEN gruen=1
    IF blau>1 THEN blau=1

    PALETTE i,rot,gruen,blau
NEXT i

PALETTE 0,0,0,0      'Hintergrund: schwarz
PALETTE 1,1,.8,.13   'Rahmenfarbe

' Start-Transformationsparameter:
' Skalierung:
sx = 1 : sy = 1 : sz = 1

' Translation:
tx = 0 : ty = 0 : tz = 0

' Rotation:
rx = 10 : ry = 0 : rz = 0

' Beobachter:
bx = 0 : by = 0 : bz = 5000

' Lichtquelle:
lq.x& = 900 : lq.y& = 1000 : lq.z& = -500

' Lichtintensität/Flächenintensität:
li.int = .7 : fl.int = 1 : hin.int = .3

' Ebenentranslation:
tex = INT(wx%/2) : tey = INT(wy%/2)

' Ebenenskalierung:
sex = 1 : sey = 1

' Start-Anzahl der Rotationen für Körper:
anz.rots% = 8

' Umrechnung der Drehwinkel in RAD:
rx = rx*PI/180 : ry = ry*PI/180 : rz = rz*PI/180
```

```
'Drehinkrementwert (Grad):
dig = 10

di = PI * dig/180              'Umrechnung nach RAD

'Beobachterinkrementwert:
binc = 100

POKE WINDOW(8)+27,1            'Farbe des Area-Outline-Pen
flags = WINDOW(8)+32
POKEW flags,PEEKW(flags) OR 8 'Area-Outline-Flag setzen

'Rotationskörper erstellen, Objekt-Daten
'in Arrays vorbereiten und
'folgende Arrays definieren:
'xr(), yr(), zr()     - Koordinaten der Raumpunkte
'xe(),ye(),ze()       - Transformierte Koordinaten
'fld%(,)              - Flächendefinitionen
'flz%(,)              - zu zeichnende Flächen
'sort.f%()            - Sortierte Indizes der Flächen
'mit.z()              - Array gemittelter z-Werte der Flächen
'farben()             - Farbintensitäten aller zu
'                       zeichnender Flächen

get.rotkoerper(-1)


'Hauptschleife:
'***************

flag%=0
WHILE flag%<>1
  IF flag%<>-999 THEN

    transform       'Alle Punkte transformieren
    verdecke        'Verdeckte Flächen ausfiltern
                    '(und Farben ermitteln)
    projektion      'Alle Punkte projizieren

  END IF

  CLS               'Fenster löschen

  PATTERN &HFFFF    'Linienmuster = durchgezogen
```

```
'Objekt zeichnen:
FOR i=0 TO afz%-1                    'Alle Flächen
  flaech.nr% = sort.f%(i)            'zu zeichnende Flächennr.
  FOR k=0 TO 3                       'Alle Eckpunkte der Fläche
    punkt.nr% = flz%(flaech.nr%,k)   'Punktnummer
    x% = tex + sex*xe(punkt.nr%)     'Punktkoordinaten
    y% = tey - sey*ye(punkt.nr%)

    IF x%<0    THEN x% = 0
    IF x%>=wx% THEN x% = wx%-1
    IF y%<0    THEN y% = 0
    IF y%>=wy% THEN y% = wy%-1

    AREA (x%,y%)

  NEXT k

  COLOR INT(farbe(flaech.nr%)*14)+2 'Farbwert einstellen
  AREAFILL 0                         'Fläche zeichnen
NEXT i

maus% = 0 : flag%=0
WHILE maus%<>1 AND flag%=0
  SLEEP            'Auf Ereignis warten

  'Mauskoordinaten als neue Nullpunkt-
  'Koordinaten übernehmen:
  maus%=MOUSE(0)
  IF maus%=1 THEN
    tex = MOUSE(3)
    tey = MOUSE(4)
    flag% = -999  'Flag für nur Zeichnen
  END IF

  ch% = ASC(INKEY$+CHR$(0))
  IF ch%=31 THEN  'Cursor links =>
    rz=rz-di      'z-Achsendrehwinkel erniedrigen
      flag% = -1    'Flag für Neuzeichnen
  END IF
  IF ch%=30 THEN  'Cursor rechts =>
    rz=rz+di      'z-Achsendrehwinkel erhöhen
    flag% = -1    'Flag für Neuzeichnen
  END IF
```

```
IF ch%=28 THEN  'Cursor hoch =>
  rx=rx-di      'x-Achsendrehwinkel erniedrigen
  flag% = -1    'Flag für Neuzeichnen
END IF
IF ch%=29 THEN  'Cursor runter =>
  rx=rx+di      'x-Achsendrehwinkel erhöhen
  flag% = -1    'Flag für Neuzeichnen
END IF
IF ch%=127 THEN 'Del => Flächenbegrenzungen ein/aus
  POKEW flags,PEEKW(flags) XOR 8
  flag% = -999  'Flag für nur Zeichnen
END IF
IF ch%=43 THEN  '+ => Skalierung auf
  sx = sx+.1
  sy = sy+.1
  sz = sz+.1
  flag% = -1
END IF
IF ch%=45 THEN  '- => Skalierung ab
  sx = sx-.1
  sy = sy-.1
  sz = sz-.1
  flag% = -1
END IF
IF ch%=56 THEN  '8 => Beobachter entfernen
  bz = bz-binc
  flag% = -1
END IF
IF ch%=50 THEN  '2 => Beobachter annähern
  bz = bz+binc
  flag% = -1
END IF
IF ch%=139 THEN 'Help => Rotation = 0
  rx = 0
  ry = 0
  rz = 0
  flag% = -1
END IF
IF ch%=27 THEN  'Esc => Objekt neu erstellen
  get.rotkoerper(0)
  flag% = -1
END IF
```

```
    IF ch%=129 THEN 'F1 => Rotationszahl ändern
      LOCATE 1,1 : PRINT "Neueingabe der Rotationszahl:"
      PRINT "Momentane Anzahl der Rotationen: ";anz.rots%
      PRINT "Neue Anzahl der Rotationen:      ";: INPUT anz.rots%
      IF anz.rots% > MAX.ROTS% THEN
        anz.rots% = MAX.ROTS%
        PRINT
        PRINT "Zahl war zu hoch! Es wurde der maximal"
        PRINT "zulässige Wert ";MAX.ROTS%;" gewählt."
      END IF
      get.rotkoerper(1)
      flag% = -1
    END IF
    IF ch%=8 THEN   'Backstep => Farbwechsel
      'Farbparameter rotieren:
      zwis     = rotf
      rotf     = gruenf
      gruenf   = blauf
      blauf    = zwis
      zwis     = rotadd
      rotadd   = gruenadd
      gruenadd = blauadd
      blauadd  = zwis

      'Farben belegen:
      FOR i=2 TO anz.farben%-1
        farbe = INT(i*100/15 + .5)/100
        rot   = farbe*rotf+rotadd
        gruen = farbe*gruenf+gruenadd
        blau  = farbe*blauf+blauadd
        IF rot>1 THEN rot=1
        IF gruen>1 THEN gruen=1
        IF blau>1 THEN blau=1

        PALETTE i,rot,gruen,blau
      NEXT i
    END IF

    'Falls Fenster geschlossen -> Ende
    IF WINDOW(7)=0 THEN
      flag%=1
    END IF

  WEND
WEND
```

```
WINDOW OUTPUT 1
SCREEN CLOSE 2

END


'Rotationskörper berechnen:

SUB get.rotkoerper(flag%) STATIC
  ' flag% = -1 : Erstaufruf
  ' flag% =  0 : Völlige Neueingabe des Körpers
  ' flag% =  1 : Körper nur neu berechnen

  SHARED anz.rots%, MAX.ROTS%, MAX.ECKP%
  SHARED wx%, wy%, PI

  IF flag% = 0 OR flag% = -1 THEN
    GOSUB edit.koerper          'Körper editieren
  END IF

  PRINT : PRINT "Bitte warten, ich rechne!"

  ' Dimensionierungen und Deklarationen:

  IF flag%<>-1 THEN
    ERASE xr, yr, zr, xe, ye, ze
    ERASE fld%, flz%, sort.f%, mit.z, farbe
  END IF

  'Punktekoordinaten:
  SHARED ap%

  ap% = anz.rots% * ecken%      'Anzahl aller Punkte

  DIM SHARED xr(ap%-1),yr(ap%-1),zr(ap%-1)
  DIM SHARED xe(ap%-1),ye(ap%-1),ze(ap%-1)

  'Flächendefinitionen:
  SHARED afd%, afz%

  afd% = anz.rots% * (ecken%-1) 'Anzahl aller Flächen
  afz% = afd%          'Anzahl zu zeichnende Flächen

  DIM SHARED fld%(afd%-1,3),flz%(afz%-1,3)
  DIM SHARED sort.f%(afz%-1),mit.z(afz%-1)
  DIM SHARED farbe(afz%-1)
```

```
'Alle anderen Punkte und Flächen errechnen:

FOR i=0 TO ecken%-1
  xr(i) = xrot%(i)-INT(wx%/2) 'Eckkoordinaten ins
  yr(i) = INT(wy%/2)-yrot%(i) 'Raumpunkt-Array übertragen
  zr(i) = 0
NEXT i

ap.% = ecken% : alt% = 0
afd.% = 0
inc.w = 2*PI/anz.rots%          'Winkelinkrement berechnen

'Rotationskörper berechnen:
FOR w=inc.w TO 2*PI+inc.w/5 STEP inc.w
  IF ap.% = ap% THEN
    ap.% = 0                    'Startpunkte = Endpunkte
  ELSE
    si = SIN(w) : co = COS(w)

    'Definitionsebene drehen:
    FOR i=0 TO ecken%-1
      x = xr(i)                 'Koordinaten erste Reihe
      z = zr(i)
      xr(ap.%+i) = x*co + z*si'um die y-Achse drehen
      yr(ap.%+i) = yr(i)
      zr(ap.%+i) = -x*si + z*co
    NEXT i
  END IF

  'Flächen konstruieren:
  FOR i=0 TO ecken%-2
    fld%(afd.%+i,0) = alt%+i  'Punktnummern speichern
    fld%(afd.%+i,1) = alt%+i+1
    fld%(afd.%+i,2) = ap.%+i+1
    fld%(afd.%+i,3) = ap.%+i
  NEXT i

  alt% = ap.%
  ap.% = ap.% + ecken%
  afd.% = afd.% + ecken%-1

NEXT w

EXIT SUB
```

```
' Umrisse des Rotationskörpers editieren:
edit.koerper:
  IF flag%<>-1 THEN
    ERASE xrot%, yrot%
  END IF
  DIM xrot%(MAX.ECKP%-1), yrot%(MAX.ECKP%-1)

  ecken% = -1

  GOSUB zeichne

  WHILE (ch%<>27 OR ecken%<=0)     'Ende bei ESC
    SLEEP                          'Auf Ereignis warten

    maus% = MOUSE(0)
    IF maus%=1 THEN                'Maustaste gedrückt

      ecken% = ecken%+1
      IF ecken% >= MAX.ECKP% GOTO rotraus

      xrot%(ecken%) = MOUSE(3)     'Mauskoordinaten merken
      yrot%(ecken%) = MOUSE(4)     'als Rotationseckpunkt

      IF ecken%=0 THEN
        PSET (xrot%(0), yrot%(0)) 'Grafikcursor setzen
      ELSE                         'oder: Linie zeichnen:
          LINE -(xrot%(ecken%), yrot%(ecken%))
        END IF
      END IF

    ch% = ASC(INKEY$+CHR$(0))      'Taste holen
    IF ch%=127 AND ecken%>=0 THEN 'Del=>letzten Punkt löschen
      ecken% = ecken%-1
      GOSUB zeichne
      IF ecken% > 0 THEN
        FOR i=0 TO ecken%-1        'neu zeichnen:
          LINE (xrot%(i), yrot%(i))-(xrot%(i+1),yrot%(i+1))
        NEXT i
      ELSE
        IF ecken% = 0 THEN PSET (xrot%(0), yrot%(0))
      END IF
    END IF
```

```
      IF WINDOW(7)=0 THEN            'Fenster geschlossen!
        WINDOW OUTPUT 1
        SCREEN CLOSE 2
        END
      END IF
    WEND

    rotraus:
    ecken% = ecken% + 1
  RETURN

  zeichne:
    CLS

    COLOR 10
    LOCATE 1,22 : PRINT "Umrißeingabe eines Rotationskörpers"

    'Koordinatenkreuz zeichnen:
    PATTERN &HF0F0                   'gestrichelt
    COLOR 15
    LINE (0,wy%/2)-(wx%,wy%/2)       'Rotationsachse
    LINE (wx%/2,0)-(wx%/2,wy%)

    PATTERN &HFFFF
    COLOR 1
  RETURN

END SUB


'Transformation aller Raum-Punkte:
SUB transform STATIC
  SHARED ap%
  SHARED sx,sy,sz,tx,ty,tz,rx,ry,rz

  'Konstanten für die Drehung berechnen:
  si.x = SIN(rx) : co.x = COS(rx)
  si.y = SIN(ry) : co.y = COS(ry)
  si.z = SIN(rz) : co.z = COS(rz)

  A = co.y * co.z
  B = co.y * si.z
  C = -si.y
  D = si.x*si.y*co.z - co.x*si.z
```

```
  E = si.x*si.y*si.z + co.x*co.z
  F = si.x*co.y
  G = co.x*si.y*co.z + si.x*si.z
  H = co.x*si.y*si.z - si.x*co.z
  J = co.x*co.y

  FOR i=0 TO ap%-1
    ' Transformationen:
    xt = sx*xr(i) + tx          'Skalierung
    yt = sy*yr(i) + ty          'und Translation
    zt = sz*zr(i) + tz

    xe(i) = xt*A + yt*B + zt*C  'Rotationen
    ye(i) = xt*D + yt*E + zt*F
    ze(i) = xt*G + yt*H + zt*J
  NEXT i

END SUB


'Projektion aller Raum-Koordinaten in die Ebene:
SUB projektion STATIC
  SHARED ap%
  SHARED bx,by,bz

  FOR i=0 TO ap%-1
      ' Zentralprojektion:
      zwis = ze(i) - bz
      xe(i) = bx - bz * (xe(i)-bx)/zwis
      ye(i) = by - bz * (ye(i)-by)/zwis
  NEXT i
END SUB


'Ausfiltern aller verdeckten Flächen
'und Sortieren der mittleren z-Werte:

SUB verdecke STATIC
  SHARED afd%, afz%
  SHARED bx,by,bz

  afz% = 0          'Anzahl zu zeichnender Flächen

  FOR i=0 TO afd%-1
```

```
'Phase 1: Flächenrückenuntersuchung:
'**********************************

'Zwei (hoffentlich) linear unabhängige
'Vektoren der Fläche ermitteln:
'v = P2-P1 // w = P3-P1:
'mit: P1,P2,P3 = Eckpunkte der Fläche
P1% = fld%(i,0)
P2% = fld%(i,1)
P3% = fld%(i,2)
P1.x = xe(P1%)
P1.y = ye(P1%)
P1.z = ze(P1%)
v.x = xe(P2%) - P1.x
v.y = ye(P2%) - P1.y
v.z = ze(P2%) - P1.z
w.x = xe(P3%) - P1.x
w.y = ye(P3%) - P1.y
w.z = ze(P3%) - P1.z

'Vektorprodukt p = v x w ermitteln:
p.x = v.y*w.z - v.z*w.y
p.y = v.z*w.x - v.x*w.z
p.z = v.x*w.y - v.y*w.x

'Sichtvektor s vom Beobachter zu P1 berechnen:
s.x = P1.x - bx
s.y = P1.y - by
s.z = P1.z - bz

'Skalarprodukt q = p * s berechnen:
q = p.x*s.x + p.y*s.y + p.z*s.z

'Vorzeichen von q testen:
IF q>0 THEN

  'Fläche als sichtbar kennzeichnen:
  sum.z = 0
  FOR k=0 TO 3
    flz%(afz%,k) = fld%(i,k) 'Flächendef. übertragen
    sum.z = ze(fld%(i,k)) + sum.z 'z-Summe bilden
  NEXT k


  GOSUB farbe           'Farbwert der Fläche errechnen
```

```
    'Phase 2: z-Mittelwert einsortieren:
    '**********************************

    'z-Mittelwert in z-Array speichern:
    mittel  = sum.z / 4
    mit.z(afz%) = mittel

    'Nach z-Mittelwert den Flächenindex in Sort-Array einorden:
    k = 0
    'Einsortierstelle suchen:
    WHILE (mittel <= mit.z(sort.f%(k)) AND k < afz%)
      k=k+1
    WEND
    IF k <= afz% THEN
      'Ab Einsortierstelle verschieben:
      FOR m=afz% TO k STEP -1
        sort.f%(m+1) = sort.f%(m)
      NEXT m
    END IF
    sort.f%(k) = afz%  'Index der Fläche merken

    afz% = afz%+1      'Anzahl sichtbare Flächen erhöhen

  END IF

NEXT i                 'Nächste Fläche

EXIT SUB

'Flächen-Farbintensitäten errechnen:
farbe:
  SHARED lq.x&, lq.y&, lq.z&
  SHARED li.int, fl.int, hin.int

  ' Vektor vom Flächenpunkt zur Lichtquelle:
  L.x = P1.x - lq.x&
  L.y = P1.y - lq.y&
  L.z = P1.z - lq.z&

  ' Intensität des einfallendes Lichtes:
  cos.a = (p.x*p.x + p.y*p.y + p.z*p.z)
  cos.a = cos.a * (L.x*L.x + L.y*L.y + L.z*L.z)
  cos.a = (p.x*L.x + p.y*L.y + p.z*L.z)/SQR(cos.a)
```

```
    farb = hin.int*fl.int           'Hintergrundintensität
    IF cos.a < 0 THEN               'Fläche dem Licht zugewandt?
      farb = farb - li.int*fl.int * cos.a ' JA! Lichteinfall
    END IF                                ' ermitteln
    farbe(afz%) = farb - INT(farb) 'nur zwischen 0 und 1!
  RETURN

END SUB
```

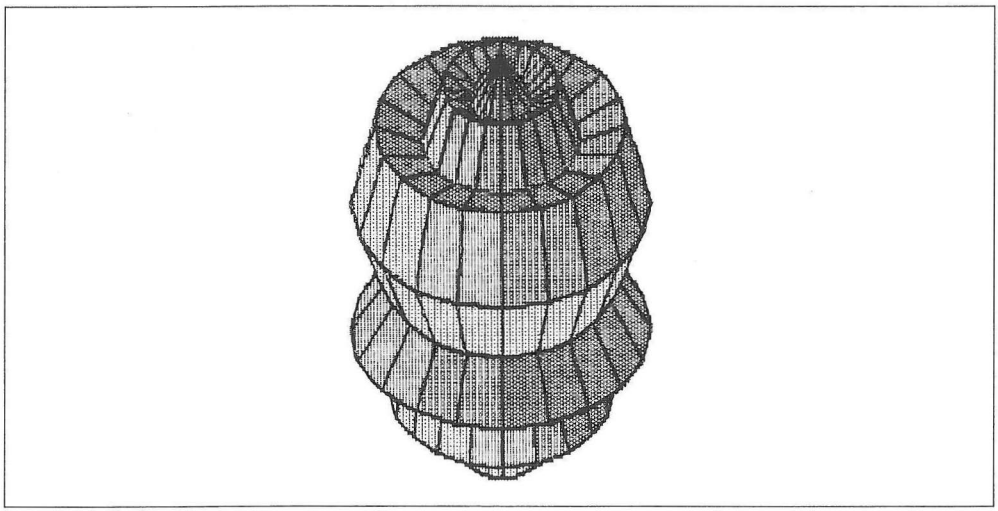


Bild 7.3: Rotationskörper 2

Gehen wir frisch ans Werk. Das ist wieder einmal ein Programm, mit dem Sie richtig kreativ arbeiten können. Hier brauchen Sie nicht nur zuschauen, hier erleben Sie mit! Starten Sie das Programm also einfach einmal. Es meldet sich mit einem fast leeren Bildschirm, in den nur ein zweidimensionales Koordinatenkreuz mit x-/y-Achsen und eine Überschrift: »Umrißeingabe eines Rotationskörpers« eingelassen ist. Hier ist die Stelle, an der Sie die Umrißebene des Rotationskörpers mit Ihrer Maus definieren können. Die senkrechte Linie markiert gleichzeitig die Position der Drehachse, um die alle von Ihnen eingegebenen Punkte später Schritt für Schritt gedreht werden, um den Rotationskörper zu erzeugen (s.o.).

Fahren Sie nun mit der Maus auf einen beliebigen Punkt des Bildschirms (möglichst nicht zu weit von der Drehachse entfernt) und betätigen den linken Mausknopf. Es erscheint ein Punkt. Bewegen Sie die Maus nun ein wenig weiter und betätigen den Mausknopf erneut. Sofort zieht das Programm eine Linie vom ersten zum zweiten Punkt. Fahren Sie so weiter fort, bis Sie etwa fünf bis sechs Punkte markiert und so verbunden haben. Jetzt haben Sie den Umriß Ihres Rotationskörpers stehen. Sollte Ihnen etwas nicht gefallen, dann drücken Sie einfach die Taste <Del> – der jeweils letzte Punkt (und mit ihm der verbindende Strich) verschwindet. Auf diese Weise gestalten Sie Ihren Rotationskörper nach Ihren Vorstellungen. Stimmt alles, dann ein Tip auf die Esc-Taste und... Die Kommandos noch einmal zusammengefaßt:

<Maustaste li>	Definieren eines Eckpunktes
<Del>	Löschen eines Eckpunktes
<Esc>	Berechnen und Zeichnen des Rotationskörpers
<Window close>	Programm beenden

...und nach wenigen Augenblicken erscheint das Corpus delicti auf dem Bildschirm. Haben Sie viele Eckpunkte eingegeben, dann kann schon einmal ein wenig mehr Zeit verstreichen (Amiga-Basic!). Ersten und letzten Punkt sollten Sie stets ziemlich nahe an der Rotationsachse positionieren, wenn nicht gar auf ihr.

Jetzt jedenfalls stehen Ihnen eine ganze Reihe von Kommandos zur Verfügung, um die Erscheinung Ihres Rotationskörpers zu variieren:

<Maustaste li>	Verschieben des Rotationskörpers
<Cursor li>	Drehung um die z-Achse links
<Cursor re>	Drehung um die z-Achse rechts
<Cursor auf>	Drehung um die x-Achse links
<Cursor ab>	Drehung um die x-Achse rechts
<Del>	Flächenbegrenzung ein/aus
< + >	Objekt vergrößern
<->	Objekt verkleinern
<8>	Beobachter entfernen
<2>	Beobachter annähern
<Help>	Alle Rotationswinkel gleich Null
<Backstep>	Farben wechseln
<F1>	Änderung der Rotationszahl
<Esc>	Neuen Rotationskörper entwerfen
<Window close>	Programm beenden

Sie haben also eine ganze Menge Auswahl. Mit F1 ändern Sie die Rotationszahl, ohne jedoch die Umrisse des Körpers zu verlieren. Sie geben nach F1 einfach die Zahl der gewünschten Rotationen ein (je höher, desto genauer) und Amiga-Basic berechnet den Rotationskörper völlig neu. Die Position der Lichtquelle und noch weitere Parameter verändern Sie bitte im Programm selbst.

Je mehr Eckpunkte und je größer die Rotationszahl, desto länger dauert es, bis ein neuer Rotationskörper berechnet, transformiert und projiziert ist, desto länger dauert die Flächen-Sichtbarkeitsüberprüfung usw. Bei ganz extremen Werten kann die Berechnung auch einige Minuten dauern (überlegen Sie einmal, wieviele Punkte, wieviele Flächen behandelt werden müssen). Ein Programm in C oder gar Assembler könnte hier in Sekundenschnelle Abhilfe schaffen! Dann wären sogar Animationen möglich.

Schreiten wir zur Programmbeschreibung. Sehr speicheraufwendig gestaltet sich die ganze Rechenarbeit. Zu Programmanfang sollte also geprüft werden, ob überhaupt genügend Speicher vorhanden ist und gegebenenfalls reserviert werden. Die Größe des zu reservierenden Speichers hängt fast nur von den Maximalwerten für Ecken- und Rotationszahlen ab. Die sind in den ersten Zeilen in einem DATA-Statement abgelegt (hier 15 und 45). Sollte Ihr Speicher

nicht reichen, dann verändern Sie diese Werte. Haben Sie mehr Speicher zur Verfügung, dann dürfen Sie sie ruhig erhöhen. Normalerweise warnt Sie das Programm, wenn Sie zuviel Speicher belegen möchten. Es kann allerdings auch passieren, daß es mit einem »OUT OF MEMORY« endet. Das liegt dann daran, daß zwar genügend Speicher vorhanden wäre, aber nicht genügend zusammenhängender Speicher zur Verfügung steht.

Die folgenden Initialisierungen kennen Sie bereits aus anderen Kapiteln. Auch die grundsätzliche Struktur und viele Routinen sollten Ihnen aus dem Kapitel zu den verdeckten Linien und Flächen bekannt sein. Es wurden nur geringfügige Änderungen (vor allem für die Rechengenauigkeit) vorgenommen. Lassen Sie uns also auf das Wesentlichste zu sprechen kommen, die Routine **get.rotkoerper( )**. Sie läßt Sie einen neuen Rotationskörper editieren und/oder berechnet alle Punkte und Flächen dieses Körpers – je nach dem Inhalt von flag%.

Die Editierung des Körpers geschieht in dem Unterprogramm-internen Sub-Programm edit.koerper. Hier wartet das Programm einfach auf Ihre Mauseingaben, merkt sich die Koordinaten in den Arrays xr( ), yr( ) und zr( ), zeichnet eine Linie vom letzten Punkt und gibt Ihnen die Möglichkeit, die Linien wieder zu löschen. Nichts von besonderer Schwierigkeit also.

Dann aber geht es los. In einer großen FOR...NEXT-Schleife wird der Winkel w langsam, Schritt für Schritt (die Anzahl der Schritte können Sie ja festlegen) hochgezählt. In der Schleife dreht das Programm alle Ihre eingegebenen Punkte mit dem Winkel w um die y-Achse und kennt damit die Koordinaten der Punkte der nächsten Drehebene. Dann konstruiert es – jeweils von vier Punkten eingerahmt – die Flächen des Körpers. Das ist eigentlich alles. Die Arrays werden dann den dafür zuständigen Routinen zur Ausgabe etc. übergeben.

KAPITEL 8

Anhang

8.1 Mathematische Grundlagen zur Computergrafik

Auf den folgenden Seiten finden Sie noch einmal eine kurze Zusammenfassung der wichtigsten mathematischen Prinzipien, Gesetze und Verfahren, die in diesem Buch zum Verständnis der dreidimensionalen Computergrafik notwendig sind. Teilweise werden hier Dinge wiederholt, die auch in den entsprechenden Kapiteln schon einmal erläutert wurden. Leser, die bei mathematischen Problemen in diesem Buch Schwierigkeiten haben, können kurz hier in den Anhang blättern und sich die notwendigen Zusammenhänge noch einmal vor Augen führen. Für die Vektorrechnung empfehle ich Ihnen auch das Kapitel 4.3, für die Matrizenrechnung Kapitel 3.2.1.

8.1.1 Trigonometrische Funktionen

Die sogenannten trigonometrischen Funktionen (Sinus, Cosinus, Tangens etc.) kennen Sie vielleicht noch aus der Schule. Gerade deshalb aber wollen wir hier kurz noch einmal zusammenfassen, was darunter zu verstehen ist und welche Gesetze es gibt.

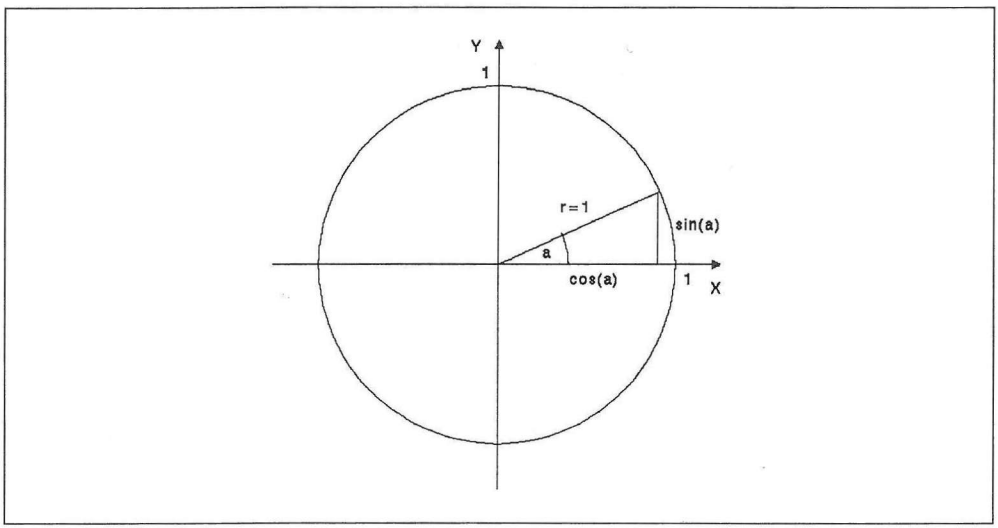


Bild 8.1: Die Definition der trigonometrischen Funktionen

Definiert werden die trigonometrischen Funktionen im sogenannten Einheitskreis (s. Bild 8.1), ein Kreis mit dem Radius 1 und dem Mittelpunkt im Punkt (0,0). Fällt man ein Lot vom Radius c auf die x-Achse, dann entsteht ein rechtwinkliges Dreieck mit den Seiten c, a und b und dem rechten Winkel zwischen den Seiten a und b. Die Länge dieser beiden Seiten hängt nun aber von dem Winkel w ab, der vom Radius c und der x-Achse gebildet wird. Man definiert die Längen a und b durch zwei neue Funktionen des Winkels w:

```
a = sin(w)      (sprich: »Sinus von w«)
b = cos(w)      (sprich: »Cosinus von w«)
```

Sie sehen, je größer w ist, desto größer wird a (also sin(w)) und desto kleiner wird b (cos(w)). Ist w größer als 90 Grad, so nimmt b negative Werte an, a wird dagegen immer kleiner usw. Nach 360 Grad beginnt der Ablauf von neuem. Die Funktion ist also periodisch mit einer Periode von 360 Grad. Trägt man die Funktionen in ein Koordinatensystem ein (auf der x-Achse die Werte für w, auf der y-Achse die für sin(w) bzw. cos(w)), resultiert die bekannte wellenförmige Sinusfunktion. Die Cosinusfunktion sieht genauso aus, ist nur um 90 Grad verschoben (s. Bild 8.2). Der größte Wert für die Sinusfunktionen beträgt 1, der niedrigste –1. Die Funktion pendelt stets zwischen diesen beiden Werten hin und her.

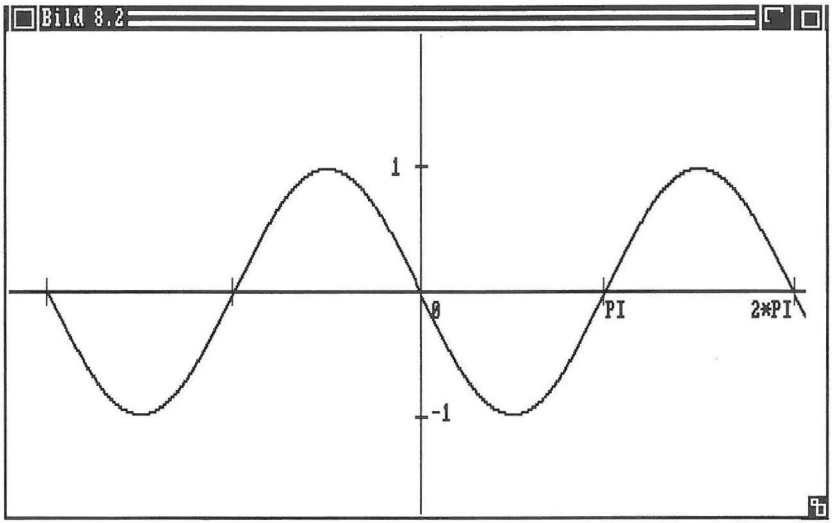


Bild 8.2: Die Sinusfunktion

Die dritte und vierte Winkelfunktion, die ebenfalls oft verwendet werden, sind wie folgt definiert:

```
tan(w) = sin(w)/cos(w)
cot(w) = cos(w)/sin(w) = 1/tan(w)
```

Die erste, tan(w), nennt sich Tangens, die zweite Cotangens. Im Einheitskreis ist der Tangens immer gleich 1.

Winkel werden oft statt wie hier in Altgrad auch in Radiant angegeben, das ist die Länge des Kreisbogens innerhalb eines Winkels w im Kreis mit dem Radius 1. Ein voller Kreis (360 Grad) hat dann die Bogenlänge von 2*PI. PI ist die bekannte Kreiskonstante mit

PI = 3.14159265...

Die Formeln für die Umrechnung eines Winkels von Grad (g) in Radiant (r) und umgekehrt lauten:

```
r = 2*PI*g/360      = PI*g/180
g = 360*r/(2*PI)    = 180*r/PI
```

Einige wichtige Winkel in Grad und Radiant:

```
  0 Grad = 0      Rad
  1 Grad = PI/180 Rad
 10 Grad = PI/9   Rad
 30 Grad = PI/6   Rad
 45 Grad = PI/4   Rad
 60 Grad = PI/3   Rad
 90 Grad = PI/2   Rad
180 Grad = PI     Rad
360 Grad = 2 * PI Rad
```

Aber zurück zu den Winkelfunktionen. Lassen wir den Radius des Kreises größer oder kleiner 1 werden, dann müssen die obigen Gleichungen erweitert werden. Mit normalen geometrischen Methoden ergibt sich dann:

```
sin(w) = a/c
cos(w) = b/c
tan(w) = a/b
cot(w) = b/a
```

mit c für den Kreisradius, w für den Winkel zwischen c und b und a bzw. b für die beiden Seiten am rechten Winkel des Dreiecks. Losgelöst von unserem Kreis besitzen wir damit Formeln für die Berechnung einer Seite eines rechtwinkligen Dreiecks aus einem Winkel und einer Seite. Gleichzeitig ist es möglich, jeden Winkel eines rechtwinkligen Dreiecks aus zwei bekannten Seiten zu bestimmen. c ist dabei stets die längste Seite im Dreieck, also die dem rechten Winkel gegenüberliegende Seite, auch Hypotenuse genannt. a und b sind dann die beiden Katheten.

Kennen Sie in einem rechtwinkligen Dreieck also einen Winkel w, dann ist der Sinus dieses Winkels immer das Verhältnis der gegenüberliegenden Seite zur Hypotenuse, der Cosinus entsprechend das Verhältnis der am Winkel liegenden Seite zur Hypotenuse, und der Tangens schließlich das Verhältnis der gegenüberliegenden zur am Winkel liegenden Seite.

Kennen Sie die Winkelfunktion eines Winkels, dann können Sie den Winkel selbst durch die sogenannten Umkehr-(Arcus-)Funktionen ermitteln. Es gilt:

```
w = arcsin(sin(w))
w = arccos(cos(w))
w = arctan(tan(w))
w = arccot(cot(w))
```

Man spricht von Arcussinus, Arcuscosinus, Arcustangens und Arcuscotangens (Ihr AmigaBasic kennt nur die Arcustangens-Funktion ATN(x)).

Hier einige wichtige Beziehungen zwischen den Winkelfunktionen:

$$\sin(x) = \cos(PI/2-x) \qquad \cos(x) = \sin(PI/2-x)$$
$$\tan(x) = \cot(PI/2-x) \qquad \cot(x) = \tan(PI/2-x)$$

$$\sin(PI+x) = -\sin(x) \qquad \cos(PI+x) = -\cos(x)$$
$$\tan(PI+x) = \tan(x) \qquad \cot(PI+x) = \cot(x)$$

$$\sin(x \pm y) = \sin(x)*\cos(y) \pm \cos(x)*\sin(y)$$
$$\cos(x \pm y) = \cos(x)*\cos(y) \mp \sin(x)*\sin(y)$$

$$\sin(2*x) = 2*\sin(x)\cos(x)$$
$$\cos(2*x) = \cos^2(x)-\sin^2(x)$$

Im rechtwinkligen Dreieck gilt auch der folgende sehr wichtige Satz (Satz des Pythagoras):

$$c^2 = a^2 + b^2$$

mit c als Hypotenuse, a und b als die beiden Katheten.

8.1.2 Vektorrechnung

Unter einem Vektor versteht man eine »gerichtete Strecke« im Raum oder in der Ebene (Pfeil) (Schreibweise: $\vec{v}$). Ein Vektor wird definiert durch Fuß- und Spitzenpunkt. Die Länge eines Vektors $\vec{a}$ nennt man auch Betrag des Vektors $|\vec{a}|$. Zwei Vektoren $\vec{a}$ und $\vec{b}$ sind gleich ($\vec{a} = \vec{b}$), wenn ihre Länge und ihre Richtung gleich sind (die Position im Raum oder in der Ebene ist gleichgültig!). Verschiebt man einen Vektor parallel, dann handelt es sich trotzdem um ein und denselben Vektor.

Hat ein Vektor die Länge eins, dann nennt man ihn den Einheitsvektor. Ein Einheitsvektor berechnet sich aus $\vec{v}' = \vec{v}/|\vec{v}|$. Der Nullvektor hat die Länge null, seine Richtung ist unbestimmt.

Man kann einen Vektor $\vec{v}$ auch alternativ durch einfache Koordinaten angeben. Dabei wird angenommen, daß sich der Fußpunkt des Vektors im Koordinaten-Nullpunkt befindet (wir können ihn ja beliebig verschieben). Als Koordinaten des Vektors werden dann nur die Koordinaten der Spitze angeführt (wir kommen also auch hier mit zwei Parametern für die Ebene und dreien für den Raum aus):

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix}$$

oder im Raum:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix}$$

Für die Länge (Betrag) eines Ebenenvektors gilt:

$$|\vec{v}|^2 = v_x^2 + v_y^2$$

Bei räumlichen Vektoren sieht das Ganze so aus:

$$|\vec{v}|^2 = v_x^2 + v_y^2 + v_z^2$$

Man kann mit Vektoren auch rechnen. Dabei gelten besondere Rechengesetze:

Multiplikation mit einer Zahl:

So ist es möglich, einen Vektor mit einer einfachen Zahl a zu multiplizieren. Das darf nicht mit dem weiter unten vorgestellten Skalarprodukt verwechselt werden! Grafisch ist darunter die Verlängerung oder Verkürzung des betreffenden Vektors $\vec{v}$ zu verstehen. Das Ergebnis ist also ebenfalls ein Vektor. Er besitzt die Länge $a * |\vec{v}|$. Ist a negativ, so kehrt sich gleichzeitig noch die Richtung des Vektors um. Bei $a = 0$ ist das Ergebnis der Nullvektor:

$$a*\vec{v} = a * \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} a*v_x \\ a*v_y \end{pmatrix}$$

im Raum:

$$a*\vec{v} = a * \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} = \begin{pmatrix} a*v_x \\ a*v_y \\ a*v_z \end{pmatrix}$$

In diesem Zusammenhang ist ein weiterer Begriff von Wichtigkeit: die lineare Unabhängigkeit von Vektoren. Zwei Vektoren sind genau dann linear unabhängig, wenn man den einen nicht durch eine Multiplikation mit einer einfachen Zahl in den anderen umwandeln kann. Mit anderen Worten: wenn sie nicht parallel oder antiparallel liegen. Linear abhängig oder besser: kolinear sind demnach zwei Vektoren, die die gleiche Richtung (oder entgegengesetzte Richtungen), aber nicht unbedingt die gleiche Länge besitzen.

Vektoraddition/-subtraktion:

Wir können zwei Vektoren $\vec{v}$ und $\vec{w}$ auch addieren und subtrahieren. Das Ergebnis ist ebenfalls ein Vektor. Wollen wir eine solche Addition grafisch darstellen, dann brauchen wir nur den Fuß des Vektors $\vec{w}$ an die Spitze des Vektors $\vec{v}$ zu verschieben. Der resultierende Vektor geht dann einfach vom Fuß des Vektors $\vec{v}$ bis zur Spitze von $\vec{w}$:

$$\vec{v}+\vec{w} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} = \begin{pmatrix} w_x \\ w_y \end{pmatrix} = \begin{pmatrix} v_x+w_x \\ v_y+w_y \end{pmatrix}$$

Entsprechendes gilt im Raum:

$$\vec{v}+\vec{w} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} + \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} v_x+w_x \\ v_y+w_y \\ v_z+w_z \end{pmatrix}$$

Die Subtraktion zweier Vektoren geht völlig analog vonstatten:

$$\vec{v}+\vec{w} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} - \begin{pmatrix} w_x \\ w_y \end{pmatrix} = \begin{pmatrix} v_x-w_x \\ v_y-w_y \end{pmatrix}$$

Im Raum:

$$\vec{v}+\vec{w} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} - \begin{pmatrix} w_x \\ w_y \\ w_z \end{pmatrix} = \begin{pmatrix} v_x-w_x \\ v_y-w_y \\ v_z-w_z \end{pmatrix}$$

Rechengesetze:

$$\begin{aligned} \vec{v}+\vec{w} &= \vec{w}+\vec{v} & \qquad \vec{v}+(\vec{w}+\vec{u}) &= (\vec{v}+\vec{w})+\vec{u} \\ a*(b*\vec{v}) &= (a*b)*\vec{v} & \qquad (a+b)*\vec{v} &= a*\vec{v} + b*\vec{v} \\ a*(\vec{v}+\vec{w}) &= a*\vec{v} + a*\vec{w} \\ |a*\vec{v}| &= |a|*|\vec{v}| \end{aligned}$$

Skalarprodukt:

Wir können zwei Vektoren $\vec{v}$ und $\vec{w}$ auch miteinander multiplizieren. Dabei gibt es allerdings zwei Möglichkeiten. Beim sogenannten Skalarprodukt $\vec{v}*\vec{w}$ (sprich: »$\vec{v}$ mal $\vec{w}$«) ist das Ergebnis eine einfache Zahl (Skalar), beim Vektorprodukt $\vec{v}\times\vec{w}$ (sprich: »$\vec{v}$ kreuz $\vec{w}$«) dagegen wieder ein Vektor.

Das Skalarprodukt ist wie folgt definiert:

$$\vec{v}*\vec{w} = |\vec{v}|*|\vec{w}|*\cos(a)$$

Dabei ist a der Winkel zwischen den beiden zu multiplizierenden Vektoren. Die Beträge von $\vec{v}$ und $\vec{w}$ sind die Längen der beiden Vektoren. Für die Koordinatenschreibweise gilt:

$$\vec{v}*\vec{w} = v_x*w_x + v_y*w_y$$

bzw. für räumliche Vektoren:

$$\vec{v}*\vec{w} = v_x*w_x + v_y*w_y + v_z*w_z$$

Stehen die beiden Vektoren aufeinander senkrecht, dann gilt:

$$\vec{v}*\vec{w} = 0 \qquad \text{für: } \vec{v} \text{ steht senkrecht auf } \vec{w}$$

Rechengesetze:

$$\begin{aligned} \vec{v}*\vec{w} &= \vec{w}*\vec{v} \\ (a*\vec{v})*\vec{w} &= a*(\vec{v}*\vec{w}) \\ \vec{v}*(\vec{w}+\vec{u}) &= \vec{v}*\vec{w} + \vec{v}*\vec{u} \\ \vec{v}*\vec{v} &= \vec{v}^2 \\ &= |\vec{v}|^2 \end{aligned}$$

Es gilt aber nicht unbedingt:

$$\vec{v}*(\vec{w}*\vec{u}) = (\vec{v}*\vec{w})*\vec{u}$$

Mit anderen Worten: Sie müssen darauf achten, die Skalarmultiplikationen stets in der richtigen Reihenfolge auszuführen!

Vektorprodukt:

Das Vektorprodukt haben wir bereits erwähnt. Es die Art der Multiplikation zweier Vektoren, die wieder einen Vektor zum Ergebnis hat. Dieser neue Vektor $\vec{p} = \vec{v} \times \vec{w}$ hat eine Länge von:

$$|\vec{p}| = |\vec{v} \times \vec{w}| = |\vec{v}| * |\vec{w}| * \sin(a)$$

Dies ist übrigens auch der Flächeninhalt des von $\vec{v}$ und $\vec{w}$ aufgespannten Parallelogramms. $\vec{p}$ steht auf den beiden Vektoren $\vec{v}$ und $\vec{w}$ senkrecht, ragt also in den Raum hinein ($\vec{v}$, $\vec{w}$ und $\vec{p}$ ergeben ein rechtshändiges System). Einen senkrechten Vektor nennt man auch Normalvektor.

Mathematisch ist das Vektorprodukt wie folgt definiert:

$$\vec{p} = \vec{v} \times \vec{w} = \begin{pmatrix} v_y * w_z - v_z * w_y \\ v_z * w_x - v_x * w_z \\ v_x * w_y - v_y * w_x \end{pmatrix}$$

Rechengesetze:

$$\begin{aligned} \vec{v} \times \vec{w} &= -(\vec{w} \times \vec{v}) \qquad !!! \\ (a * \vec{v}) \times \vec{w} &= a * (\vec{v} \times \vec{w}) \\ \vec{v} \times (\vec{w} + \vec{u}) &= \vec{v} \times \vec{w} + \vec{v} \times \vec{u} \\ \vec{v} \times \vec{w} &= \text{Nullvektor (falls } \vec{v} \text{ zu } \vec{w} \text{ parallel bzw. antiparallel ist)} \\ \vec{v} \times \vec{v} &= \text{Nullvektor} \end{aligned}$$

Punktdarstellung durch Vektoren:

Ein Punkt kann ebenfalls als Vektor interpretiert werden. In diesem Fall ist der Vektor vom Nullpunkt zum Punkt:

$$P = \vec{v} = \begin{pmatrix} x \\ y \end{pmatrix} \qquad \text{bzw. } P = \vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

x, y und z sind dabei die Koordinaten des Punktes. Mit P – steht ja eigentlich für den Punkt P – rechnen wir genauso, wie mit normalen Vektoren.

8.1.3 Geraden und Ebenen

Geraden:

Im folgenden werden einige Gleichungen aufgezeigt, mit denen Sie die Punkte einer Geraden berechnen können:

Ganz grundlegend dabei ist natürlich die Geradengleichung in der Ebene, die wohl jeder schon einmal irgendwo gesehen hat:

$$a * x + b * y + c = 0 \qquad \text{(allgemeine Form)}$$

oder:

$$y = m * x + n \qquad \text{(normierte Form)}$$

wobei m die Steigung und n den Schnittpunkt mit der y-Achse angeben, es gilt:

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

Für den Raum ist eine Gerade als lineares Gleichungssystem darstellbar (Schnittgerade zweier Ebenen):

$$a_1 * x + b_1 * y + c_1 * z + d_1 = 0 \quad \text{und}$$
$$a_2 * x + b_2 * y + c_2 * z + d_2 = 0$$

Auch diese Form ist möglich:

$$y = m * (x - x_1) + y_1 \qquad \text{(Punkt-Steigungs-Form)}$$

wobei:

x_1, y_1	ein Punkt auf der Geraden
m	Steigung (s.o.)

Daraus leitet sich die nächste Form ab:

$$\frac{y - y_1}{y_2 - y_1} = \frac{x - x_1}{x_2 - x_1} \qquad \text{(Zweipunktegleichung)}$$

Eine letzte, nicht so alltägliche Form lautet:

$$\frac{x}{a} + \frac{y}{b} = 1 \qquad \text{(Achsenabschnittsform)}$$

Diese Gerade schneidet die x-Achse im Punkt P(a,0) und die y-Achse im Punkt Q(0,b).

Vektorielle Geradengleichung (für die Ebene und für den Raum):

$$g: \vec{p} = \vec{p}_0 + t * \vec{a}$$

und in Parameterschreibweise:

$$g: \begin{aligned} p_x &= p_{0x} + t * a_x \\ p_y &= p_{0y} + t * a_y \\ p_z &= p_{0z} + t * a_z \end{aligned}$$

oder als Gleichungssystem:

$$\frac{x - p_{0x}}{a_x} = \frac{y - p_{0y}}{a_y} = \frac{z - p_{0z}}{a_z}$$

dabei bedeuten:

- $\vec{p}$ Vektor vom Koordinatenursprung zu dem zu berechnenden Punkt P der Geraden (oder einfach: Punkt P)
- $\vec{p}_0$ Vektor vom Koordinatenursprung zu einem beliebigen Punkt P_0 der Geraden (oder einfach: Punkt P_0)
- $\vec{a}$ »Richtungsvektor«, beliebiger Vektor parallel zur Geraden
- t Faktor zur »Verlängerung« (oder Verkürzung) des Vektors $\vec{a}$, um Punkt P zu erreichen

Um eine ganz bestimmte Gerade zu bestimmen, brauchen wir lediglich die Parameter $\vec{p}_0$ und $\vec{a}$ anzugeben. Aus beliebigen Werten für t errechnen sich dann alle Punkte P (bzw. Vektoren $\vec{p}$) der Gerade.

Ebenen:

Im folgenden werden einige Gleichungen aufgezeigt, mit denen Sie die Punkte einer Ebene im Raum berechnen können:

Hier die allgemeine Ebenengleichung:

$$a*x + b*y + c*z + d = 0$$

Die Achsenabschnittsform lautet:

$$\frac{x}{a} + \frac{y}{b} + \frac{z}{c} = 1$$

Diese Ebene schneidet die x-Achse im Punkt P(a,0,0), die y-Achse im Punkt Q(0,b,0) und die z-Achse im Punkt R(0,0,c).

Vektorielle Ebenengleichung:

$$e: \vec{p} = \vec{p}_0 + s*\vec{a} + t*\vec{b}$$

oder in Parameterform:

$$\begin{aligned} e: \; p_x &= p_{0x} + s*a_x + t*b_x \\ p_y &= p_{0y} + s*a_y + t*b_y \\ p_z &= p_{0z} + s*a_z + t*b_z \end{aligned}$$

dabei bedeuten:

- $\vec{p}$ Vektor vom Koordinatenursprung zu dem zu berechnenden Punkt P der Ebene (oder einfach: Punkt P)
- $\vec{p}_0$ Vektor vom Koordinatenursprung zu einem beliebigen Punkt P_0 der Ebene (oder einfach: Punkt P_0)
- $\vec{a}$,$\vec{b}$ Beliebige, linear unabhängige Vektoren auf der Ebene
- s,t Faktoren zur »Verlängerung« (oder Verkürzung) der Vektoren $\vec{a}$ und $\vec{b}$, um Punkt P zu erreichen

Um eine ganz bestimmte Ebene zu zeichnen, brauchen wir lediglich die Parameter $\vec{p}_0$, $\vec{a}$ und $\vec{b}$ anzugeben. Aus beliebigen Werten für s und t errechnen sich dann alle Punkte P (bzw. Vektoren $\vec{p}$) der Ebene.

Eine andere Schreibweise der vektoriellen Ebenengleichung:

$$\vec{n} * \vec{p} + d = 0$$

oder:

$$(\vec{p} - \vec{p}_0) * \vec{n} = 0$$

$\vec{n}$ ein auf der Ebene senkrechter Vektor (meist mit der Länge 1)

$\vec{p}$ Vektor vom Koordinatenursprung zu dem zu berechnenden Punkt P der Ebene (oder einfach: Punkt P)

$\vec{p}_0$ Vektor vom Koordinatenursprung zu einem beliebigen Punkt P_0 der Ebene (oder einfach: Punkt P_0)

d Höhenkoeffizient (bei $d = 0$ geht die Ebene durch den Nullpunkt)

Zur Umformung der beiden vektoriellen Ebenengleichungen in die jeweils andere schauen Sie bitte in Kapitel 4.3.

8.1.4 Matrizen

Unter einer Matrix stellen wir uns einfach eine rechteckige Anordnung von verschiedenen Zahlen vor:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & \dots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & \dots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & \dots & a_{3n} \\ a_{41} & a_{42} & . & . & . & \dots & . \\ . & . & . & . & . & \dots & . \\ . & . & . & . & . & \dots & . \\ a_{m1} & a_{m2} & a_{m3} & a_{m4} & a_{m5} & \dots & a_{mn} \end{pmatrix}$$

Jede Zahl a_{ik} (z.B. a_{11} oder a_{34}) stellt ein Element der Matrix dar (die Indizes an den Elementen oben sollen nur zur Unterscheidung dienen, die Namen für Matrizen sind vereinbarungsgemäß immer Großbuchstaben, hier A). Die Matrix setzt sich zusammen aus Zeilen (waagerechte Reihe von Elementen) und Spalten (senkrechte Reihe von Elementen). Sie ist, so sagt man, vom Typ (m,n), da sie m Zeilen und n Spalten besitzt. Man spricht auch von einer (m,n)-Matrix. Die Elemente a_{11}, a_{12}, a_{13}, ... (oder kurz: a_{1k}) stellen die erste Zeile dar, alle Elemente a_{2k} die zweite Zeile etc. Analog verhält es sich mit den Spalten.

Ein Element aik steht also in der Zeile i und in der Reihe k. Wir haben es hier also sozusagen mit einem zweidimensionalen Array zu tun.

Die Anzahl von Elementen in einer Zeile kann (muß aber nicht) gleich der Anzahl der Elemente in einer Spalte sein. In diesem Fall sprechen wir von einer n-reihigen quadratischen Matrix, z.B.:

$$A = \begin{pmatrix} 1 & 5 & -6 \\ -7 & 13 & 8 \\ -1 & 2 & 22 \end{pmatrix}$$

Diese Matrix ist dreireihig und quadratisch.

Ein weiterer Sonderfall von Matrizen sind solche, die nur eine Zeile oder nur eine Spalte besitzen. Sie sind daher vom Typ (1,n) bzw. (m,1) (s.o.), z.B.:

$$A = (1\ 5\ 3\ -4)$$

Hier haben wir es mit einer Matrix zu tun, die nur eine einzige Zeile besitzt (hier mit 4 »Spalten«). Sie wird auch »Zeilenvektor« genannt. Der andere Fall ist ein sogenannter »Spaltenvektor«, z.B.:

$$A = \begin{pmatrix} -4 \\ 9 \\ 6 \end{pmatrix}$$

Addition/Subtraktion zweier Matrizen:

Eine Addition bzw. Subtraktion kann nur mit typengleichen Matrizen (gleiche Anzahl Spalten und Zeilen) stattfinden.

Die Addition zweier Matrizen $C = A + B$ findet elementweise statt. Jedes Element aik der einen Matrix wird mit dem korrespondierenden Element bik der anderen Matrix addiert zum entsprechenden Element cik der Ergebnismatrix. Es gilt:

$$c_{ik} = a_{ik} + b_{ik} \qquad \text{für alle i,k}$$

Analoges gilt für die Subtraktion:

$$c_{ik} = a_{ik} - b_{ik} \qquad \text{für alle i,k}$$

Multiplikation einer Matrix mit einer Zahl:

Eine Multiplikation $B = n * A$ einer beliebigen Matrix mit einer reellen Zahl findet ebenfalls elementweise statt. Jedes Element aik der Matrix wird mit der zahl n multipliziert. Es gilt:

$$b_{ik} = n * a_{ik} \qquad \text{für alle i,k}$$

Addition, Subtraktion und Multiplikation mit einer Zahl gehorchen denselben Gesetzen wie die normalen Zahlenoperationen.

Multiplikation verketteter Matrizen:

Die Multiplikation zweier Matrizen ist nur für verkettete Matrizen definiert. Unter verketteten Matrizen verstehen wir zwei Matrizen A und B, von denen die erste genausoviele Spalten besitzt wie die anderen Zeilen. Ein Beispiel einer erlaubten Multiplikation wäre dann:

$$A * B = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

A besitzt drei Spalten, B drei Zeilen, und damit wäre die Bedingung erfüllt. Wir sehen aber auch, daß die Multiplikation B * A nicht erlaubt ist. Auch wenn B * A erlaubt wäre, hieße das nicht, daß das Ergebnis das gleiche wäre wie A * B. Die Reihenfolge der Faktoren ist also stets wichtig! Beachten Sie, daß eine Matrixmultiplikation zwischen zwei quadratischen Matrizen (s.o.) immer erlaubt ist.

Das Ergebnis einer Matrixmultiplikation ist wieder eine Matrix. Sie besitzt die gleiche Anzahl von Zeilen wie die erste (A) und die gleiche Anzahl von Spalten wie die zweite Matrix (B). Das Ergebnis der obigen Multiplikation ((3,3)-Matrix mal (3,2)-Matrix) wäre also eine Matrix vom Typ (3,2) (3 Zeilen, 2 Spalten).

Die Berechnung dieser Ergebnismatrix C ist ein wenig kompliziert. Für die kundigen Leser schreiben wir die Berechnung eines einzigen Elementes in mathematischer Kurzschreibweise:

$$c_{ik} = \sum_{j=1}^{n} a_{ij} * b_{jk} = a_{i1} * b_{1k} + a_{i2} * b_{2k} + \ldots + a_{in} * b_{nk}$$

was soviel heißt wie: Das Element cik berechnet sich aus der Summe aller $a_{ij} * b_{jk}$ für j = 1 bis j = n.

wobei:

c_{ik}	ein Element der Ergebnismatrix C
a_{ij}	ein Element der Matrix A
b_{jk}	ein Element der Matrix B
n	Anzahl der Spalten von A, also auch: Anzahl der Zeilen von B

Wir berechnen, so sagt die Formel, ein Element cik der Ergebnismatrix, indem wir alle Elemente der i-ten Zeile(!) von A mit den jeweils korrespondierenden Elementen der k-ten Spalte(!) von B multiplizieren und von den Ergebnissen die Summe bilden (also addieren):

$$C = A * B$$

$$= \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} * \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

$$= \begin{pmatrix} 1*1 + 2*3 + 3*5 & 1*2 + 2*4 + 3*6 \\ 4*1 + 5*3 + 6*5 & 4*2 + 5*4 + 6*6 \\ 7*1 + 8*3 + 9*5 & 7*2 + 8*4 + 9*6 \end{pmatrix}$$

$$= \begin{pmatrix} 22 & 28 \\ 49 & 64 \\ 76 & 100 \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{pmatrix}$$

Rechenregeln:

A*(B*C) = (A*B)*C

Falsch wäre allerdings: A*(B*C)=(A*C)*B

Wie wir oben bereits festgehalten haben, gilt nicht – und das ist sehr wichtig: A*B=B*A

Es existieren eine Reihe von besonderen Matrizen, mit denen man eine andere Matrix multiplizieren kann, ohne daß sich etwas ändert. Es sind dies quadratische Matrizen, in denen alle Elemente gleich Null sind. Lediglich die Elemente der sogenannten Hauptdiagonale müssen den Wert Eins besitzen, man nennt sie auch quadratische Einheitsmatrizen E:

(1) oder $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ oder $\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Es gilt:

A*E = A

8.1.5 Transformationsmatrizen

Im folgenden seien noch einmal alle im Buch besprochenen Transformationsmatrizen übersichtlich zusammengefaßt. Alle Matrizen werden hier in homogenen Koordinaten angegeben.

8.1.5.1 2-D-Matrizen

Skalierung (Vergrößerung/Verkleinerung):

Skalierungsmatrix:

$$S(S_x, S_y) = \begin{pmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Dabei bedeuten:

S_x Skalierungsfaktor in x-Richtung
S_y Skalierungsfaktor in y-Richtung

Je nach Wert für S_x oder S_y verändert sich das Bild in folgender Weise:

	$S_{x/y}$	> 1	Vergrößerung
	$S_{x/y}$	$= 1$	Keine Veränderung
$0 <$	$S_{x/y}$	< 1	Verkleinerung
	$S_{x/y}$	< 0	gleichzeitig Spiegelung um x/y-Achse

Skalierung eines Punktes:

$$P' = P(x,y) * S(S_x,S_y)$$
$$= (x\ y\ n) * S(S_x,S_y)$$

In Parameterschreibweise:

$$x' = S_x * x$$
$$y' = S_y * y$$
$$(n' = n)$$

Translation (Verschiebung):

Translationsmatrix:

$$T(T_x, T_y) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{pmatrix}$$

Dabei bedeuten:

T_x Verschiebung in x-Richtung
T_y Verschiebung in y-Richtung

Translation eines Punktes:

$$P' = P(x,y) * T(T_x,T_y)$$
$$= (x\ y\ n) * T(T_x,T_y)$$

In Parameterschreibweise:

$$x' = T_x * x$$
$$y' = T_y * y$$
$$(n' = n)$$

Spiegelungen:

Matrizen:

Spiegelung um die x-Achse:

$$M_x = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Spiegelung um die y-Achse:

$$M_y = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Parameterschreibweisen:
Spiegelung um die x-Achse:

$x' = x$
$y' = -y$

Spiegelung um die y-Achse:

$x' = -x$
$y' = y$

Rotation (Drehung) um den Nullpunkt:

Rotationsmatrix:

$$R(w) = \begin{pmatrix} \cos(w) & \sin(w) & 0 \\ -\sin(w) & \cos(w) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

wobei:

w – Rotationswinkel

Rotation eines Punktes:

$P' = P(x,y) * R(w)$
$= (x\ y\ n) * R(w)$

Parameterschreibweise:

$x' = x*\cos(w) - y*\sin(w)$
$y' = x*\sin(w) + y*\cos(w)$
$(n' = n)$

Rotation um einen beliebigen Punkt:

Rotationsmatrix:

$$R'(w,x_z,y_z) = T_1(-x_z,-y_z) * R(w) * T_2(x_z,y_z) =$$

$$\begin{pmatrix} \cos(w) & \sin(w) & 0 \\ -\sin(w) & \cos(w) & 0 \\ -x_z*\cos(w)+y_z*\sin(w)+x_z & -x_z*\sin(w)-y_z*\cos(w)+y_z & 1 \end{pmatrix}$$

wobei:

w Rotationswinkel
x_z x-Koordinate Rotationszentrum
y_z y-Koordinate Rotationszentrum

Rotation eines Punktes:

$$P' = P(x,y) * R'(w,x_z,y_z)$$
$$= (x\ y\ n) * R'(w,x_z,y_z)$$

Parameterschreibweise:

$$x' = x*\cos(w) - y*\sin(w) - x_z*\cos(w) + y_z*\sin(w) + x_z$$
$$y' = x*\sin(w) + y*\cos(w) - x_z*\sin(w) - y_z*\cos(w) + y_z$$
$$(n' = n)$$

8.1.5.2 3-D-Matrizen

Skalierung (Vergrößerung/Verkleinerung):

Skalierungsmatrix:

$$S(S_x, S_y, S_z) = \begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dabei bedeuten:

S_x Skalierung in x-Richtung
S_y Skalierung in y-Richtung
S_z Skalierung in z-Richtung

Skalierung eines Punktes:

$$P' = P(x,y,z) * S(S_x,S_y,S_z)$$
$$= (x\ y\ z\ n) * S(S_x,S_y,S_z)$$

In Parameterschreibweise:

$$x' = S_x*x$$
$$y' = S_y*y$$
$$z' = S_z*z$$
$$(n' = n)$$

Translation (Verschiebung):

Translationsmatrix:

$$T(T_x,T_y,T_z) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{pmatrix}$$

Dabei bedeuten:

T_x Verschiebung in x-Richtung
T_y Verschiebung in y-Richtung

Translation eines Punktes:

$$\begin{aligned} P' &= P(x,y,z) * T(T_x,T_y,T_z) \\ &= (x\ y\ z\ n) * T(T_x,T_y,T_z) \end{aligned}$$

In Parameterschreibweise:

$$\begin{aligned} x' &= y+T_x \\ y' &= y+T_y \\ z' &= z+T_z \\ (n' &= n) \end{aligned}$$

Rotationen um Koordinatenachsen:
Rotation um die x-Achse:

Rotationsmatrix:

$$R_x(a) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(a) & \sin(a) & 0 \\ 0 & -\sin(a) & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dabei bedeutet:

a Rotationswinkel um die x-Achse

Rotation eines Punktes:

$$\begin{aligned} P' &= P(x,y,z) * R_x(a) \\ &= (x\ y\ z\ n) * R_x(a) \end{aligned}$$

In Parameterschreibweise:

$$\begin{aligned} x' &= x \\ y' &= y*\cos(a) - z*\sin(a) \\ z' &= y*\sin(a) + z*\cos(a) \\ (n' &= n) \end{aligned}$$

Rotation um die y-Achse:

Rotationsmatrix:

$$R_y(a) = \begin{pmatrix} \cos(a) & 0 & -\sin(a) & 0 \\ 0 & 1 & 0 & 0 \\ \sin(a) & 0 & \cos(a) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dabei bedeutet:

a Rotationswinkel um die y-Achse

Rotation eines Punktes:

$$P' = P(x,y,z) * R_y(a)$$
$$= (x\ y\ z\ n) * R_y(a)$$

In Parameterschreibweise:

$$x' = x*\cos(a) + z*\sin(a)$$
$$y' = y$$
$$z' = -x*\sin(a) + z*\cos(a)$$
$$(n' = n)$$

Rotation um die z-Achse:

Rotationsmatrix:

$$R_z(a) = \begin{pmatrix} \cos(a) & \sin(a) & 0 & 0 \\ -\sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dabei bedeutet:

a Rotationswinkel um die z-Achse

Rotation eines Punktes:

$$P' = P(x,y,z) * R_z(a)$$
$$= (x\ y\ z\ n) * R_z(a)$$

In Parameterschreibweise:

$$x' = x*\cos(a) - y*\sin(a)$$
$$y' = x*\sin(a) + y*\cos(a)$$
$$z' = z$$
$$(n' = n)$$

Rotation um alle drei Achsen in der Reihenfolge x-, y-, z-Achse:

Rotationsmatrix:

$$R_x(a)_y(b)_z(c) = R_x(a) * R_y(b) * R_z(c) =$$

$$\begin{pmatrix} \cos(b)\cos(c) & \cos(b)\sin(c) & -\sin(b) & 0 \\ \sin(a)\sin(b)\cos(c) - \cos(a)\sin(c) & \sin(a)\sin(b)\sin(c) + \cos(a)\cos(c) & \sin(a)\cos(b) & 0 \\ \cos(a)\sin(b)\cos(c) + \sin(a)\sin(c) & \cos(a)\sin(b)\sin(c) - \sin(a)\cos(c) & \cos(a)\cos(b) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Dabei bedeutet:

a Rotationswinkel um die x-Achse
b Rotationswinkel um die y-Achse
c Rotationswinkel um die z-Achse

Rotation eines Punktes:

$$P' = P(x,y,z) * R_x(a)_y(b)_z(c)$$
$$= (x\ y\ z\ n) * R_x(a)_y(b)_z(c)$$

In Parameterschreibweise:

$$x' = x*A + y*B + z*C$$
$$y' = x*D + y*E + z*F$$
$$z' = x*G + y*H + z*I$$
$$(n' = n)$$

wobei:

$$A = \cos(b)*\cos(c)$$
$$B = \cos(b)*\sin(c)$$
$$C = -\sin(b)$$
$$D = \sin(a)*\sin(b)*\cos(c)-\cos(a)*\sin(c)$$
$$E = \sin(a)*\sin(b)*\sin(c)+\cos(a)*\cos(c)$$
$$F = \sin(a)*\cos(b)$$
$$G = \cos(a)*\sin(b)*\cos(c)+\sin(a)*\sin(c)$$
$$H = \cos(a)*\sin(b)*\sin(c)-\sin(a)*\cos(c)$$
$$I = \cos(a)*\cos(b)$$

Rotationen um eine beliebige Achse:

Rotationsmatrix:

$$R_{achse}(a) =$$
$$T(-x_0,-y_0,-z_0) * R_x(w_1) * R_y(w_2) * R_z(a) *$$
$$R_y(-w_2) * R_x(-w_1) * T(x_0,y_0,z_0)$$

mit:

$$T(-x_0,-y_0,-z_0) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_0 & -y_0 & -z_0 & 1 \end{pmatrix}$$

$$T(x_0,y_0,z_0) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ x_0 & y_0 & z_0 & 1 \end{pmatrix}$$

$$R_x(w_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & z/c & y/c & 0 \\ 0 & -y/c & z/c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_x(-w_1) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & z/c & -y/c & 0 \\ 0 & y/c & z/c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(w_2) = \begin{pmatrix} c/d & 0 & -x/d & 0 \\ 0 & 1 & 0 & 0 \\ x/d & 0 & c/d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y(-w_2) = \begin{pmatrix} c/d & 0 & x/d & 0 \\ 0 & 1 & 0 & 0 \\ -x/d & 0 & c/d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

wobei:

a	Rotationswinkel um die Achse
x_0,y_0,z_0	Koordinaten eines Punktes auf der Achse
x,y,z	Koordinaten eines zweiten Punktes der Achse

und:

$c^2 = y^2 + z^2$
$d^2 = x^2 + c^2$

Rotation eines Punktes:

$P' = P(x,y,z) * R_{achse}(a)$
$= (x\ y\ z\ n) * R_{achse}(a)$

8.2 Im Buch verwendete Library-Funktionen

In den verschiedenen Programmen dieses Buches wurde häufig Gebrauch von den vielen Library-Funktionen gemacht, die uns das Betriebssystem des Amiga zur Verfügung stellt. In den jeweiligen Kapiteln wurden sie nur kurz und meist nicht umfassend erklärt. Aus diesem Grunde finden Sie sie hier im Anhang noch einmal kurz und prägnant zusammengefaßt.

Selbstverständlich kann das hier nur ein winziger Teil der tatsächlich vorhandenen Library-Funktionen sein. Für weitergehende Informationen empfehle ich die umfassende Literatur zum Amiga-Betriebssystem.

Exec/DOS-Funktionen:
CloseLibrary(Library)

Die zuvor mit OpenLibrary( ) geöffnete Bibliothek wird wieder geschlossen.

Eingaben:
Library Adresse der Library-Knotenstruktur

Ausgaben:

C-Datentypen:
```
void CloseLibrary( );
z.B.: struct GfxBase *Library;
oder: struct IntuitionBase *Library;
```

Beschreibung:
Benötigen Sie die Funktionen einer Library für Ihr Programm nicht mehr, dann schließen Sie sie mit diesem Befehl. Danach dürfen Sie keine Bibliotheksfunktionen aus dieser Bibliothek mehr ansteuern.

Exit(Fehlercode)

Beenden eines Programmes (DOS-Funktion).

Eingaben:
Fehlercode Fehlercode für aufrufendes Programm

Ausgaben:

C-Datentypen:
```
void Exit( );
```

Beschreibung:
Exit( ) beendet Ihr Programm. Die Funktion kehrt also nicht wieder in Ihr Programm zurück. Wurde das Programm unter CLI gestartet, dann wird »Fehlercode« als ein Return-Code interpretiert.

Message = GetMsg(Port)

Holt die nächste Message (»Botschaft«) vom Message-Port.

Eingaben:
Port Adresse des Empfänger-Message-Ports

Ausgaben:
Message Adresse der empfangenen Message oder Null

C-Datentypen:

```
struct Message *GetMsg( );
z.B.: struct IntuiMessage *Message;
```

Beschreibung:
Mit dieser Funktion (sozusagen eine INPUT-Funktion) empfangen Sie Daten von irgendeinem Gerät (z.B. Tastatur oder Maus oder Disk oder...). Die Funktion wartet jedoch nicht auf eine Message, falls keine vorliegt. Warten können Sie beispielsweise mit Wait( ). Lag eine Information (z.B. über einen Tastendruck) vor, dann gibt der Befehl die Adresse der entsprechenden Message-Struktur zurück, andernfalls Null.

Die Message-Struktur in einem Intuition-Fenster sieht z.B. so aus:

```
struct IntuiMessage
{
   struct   Message ExecMessage;
   ULONG    Class;          /* IDCMP-Flag des Ereignisses */
   USHORT   Code;           /* Menünummer, Taste etc.     */
   USHORT   Qualifier;      /* Zusatzinf. SHIFT, Alt etc. */
   APTR     IAdress;        /* Adresse v. Gadget, Screen..*/
   SHORT    MouseX, MouseY; /* Mauskoordinaten            */
   ULONG    Seconds, Micros; /* Systemzeit d. Ereignisses*/
   struct   Window *IDCMPWindow; /* Fensteradresse        */
   struct   IntuiMessage *SpecialLink;
};
```

Nicht bei jeder Art von Meldung werden auch alle Strukturelemente verwendet. Aus dieser Struktur können Sie sich also die notwendigen Informationen über die Art und Inhalt der Botschaft holen. Wenn Sie die Message nicht mehr benötigen, dann müssen Sie sie mit **ReplyMsg( )** wieder quittieren.

Library = OpenLibrary(LibName, Version)

Schaffe den Zugriff auf eine Library.

Eingaben:

LibName	Library-Name
Version	Versionsnummer der Library

Ausgaben:

Library	Adresse der Library-Knotenstruktur oder Null

C-Datentypen:

```
struct Library *OpenLibrary( );
char *LibName;
long Version;
```

Beschreibung:
Die Funktion OpenLibrary( ) eröffnet Ihnen den Zugriff auf eine Library. Eventuell muß eine nicht ROM-residente Library von Disk zugeladen werden. Vorher dürfen Sie keine Funktionen ansteuern, die zu dieser Library gehören. Sollte ein Zugriff nicht möglich sein, liefert die Funktion eine Null. Vor dem Programmende sollten Sie die Library per CloseLibrary( ) wieder schließen.

ReplyMsg(Message)

Gibt eine Message zurück.

Eingaben:
Message Adresse der Message

Ausgaben:

C-Datentypen:
```
void ReplyMsg( );
struct Message Message;
```

Beschreibung:
Mit dieser Funktion geben Sie die empfangene Message (evtl. mit Rückmeldungen) zurück.

Signal = Wait(Signalmaske)

Wartet auf ein oder mehrere Signale.

Eingaben:
Signalmaske Bitmaske der Signale, auf die gewartet (Bit = 1) bzw. nicht gewartet (Bit = 0) werden soll.

Ausgaben:
Signal Nummer des eingetretenen Signals (Bitnummer)

C-Datentypen:
```
long Wait( );
long Signalmaske;
```

Beschreibung:
Wait( ) wartet auf ein bestimmtes oder mehrere Signale. Diese Signale reservieren Sie z.B. mit den IDCMP-Flags. Die Anwendung können Sie am besten den Demo-Programmen entnehmen.

Intuition-Funktionen:
CloseScreen(Screen)

Schließen eines zuvor mit OpenScreen( ) geöffneten Intuition-Screens.

Eingaben:
Screen Adresse der Screen-Struktur

Ausgaben:

C-Datentypen:
```
void CloseScreen( );
struct Screen *Screen
```

Beschreibung:
Mit dieser Funktion schließen Sie einen Screen unter Intuition. In dem Screen dürfen keine Fenster mehr offen sein. Ist der letzte Screen geschlossen, dann versucht diese Funktion, den Workbench-Screen zu öffnen.

CloseWindow(Window)

Schließen eines zuvor mit OpenWindow( ) geöffneten Intuition-Windows.

Eingaben:
Window Adresse der Window-Struktur

Ausgaben:

C-Datentypen:
```
void CloseWindow( );
struct Window *Window;
```

Beschreibung:
Mit dieser Funktion schließen Sie ein Fenster (Window) unter Intuition. In dem Fenster darf keine Menüzeile mehr definiert sein! Ist das letzte Fenster in einem System-Screen (nicht zu verwechseln mit einem Custom-Screen) geschlossen, dann schließt diese Funktion gleichzeitig den Screen.

Screen = OpenScreen(NewScreen)

Öffnen eines Intuition-Screens.

Eingaben:
NewScreen Adresse der NewScreen-Struktur

Ausgaben:
Screen Adresse der Screen-Struktur

C-Datentypen:
```
struct Screen *OpenScreen( );
struct NewScreen  * NewScreen;
```

Beschreibung:
Dies ist die zentrale Funktion zum Öffnen eines neuen Screens unter Intuition. Bevor Sie jedoch diese Funktion aufrufen können, müssen Sie eine Struktur mit dem Namen NewScreen initialisieren (in Basic entsprechenden Speicherplatz reservieren und belegen. Diese Struktur enthält

wichtige Daten über den neu einzurichtenden Screen (z.B. die Anzahl der Farben, die Auflösung etc.). Als Rückgabewert erhalten Sie die Adresse auf die eigentliche Screen-Struktur, in der Sie die Daten der NewScreen-Struktur und noch einige weitere Daten finden können. Den Aufbau der NewScreen-Struktur entnehmen Sie bitte den Beispielprogrammen.

Window = OpenWindow(NewWindow)

Öffnen eines Intuition-Windows.

Eingaben:
NewWindow Adresse der NewWindow-Struktur

Ausgaben:
Window Adresse der Window-Struktur

C-Datentypen:
```
struct Window *OpenWindow();
struct NewWindow *NewWindow;
```

Beschreibung:
Dieses Kommando öffnet ein neues Intuition-Fenster. Falls Sie in der NewWindow-Struktur den Screen-Typ CUSTOMSCREEN angeben, dann müssen Sie bereits einen solchen Screen per OpenScreen( ) geöffnet haben. Soll das Fenster in einem Standard-Screen geöffnet werden, dann wird dieser Screen automatisch geöffnet, falls er noch nicht existiert.

Bevor Sie jedoch diese Funktion aufrufen können, müssen Sie eine Struktur mit dem Namen NewWindow initialisieren (in Basic entsprechenden Speicherplatz reservieren und belegen. Diese Struktur enthält wichtige Daten über das neu einzurichtende Fenster (z.B. Gadgets, Größe, Position etc.). Als Rückgabewert erhalten Sie die Adresse auf die eigentliche Window-Struktur, in der Sie die Daten der NewWindow-Struktur und noch einige weitere Daten finden können. Den Aufbau der NewWindow-Struktur entnehmen Sie bitte den Beispielprogrammen.

ScreenToBack(Screen)

Der angegebene Screen wird hinter alle anderen Screens manövriert.

Eingaben:
Screen Adresse der Screen-Struktur

Ausgaben:

C-Datentypen:
```
void ScreenToBack();
struct Screen *Screen;
```

Beschreibung:
Mit diesem Kommando erreichen Sie genau dasselbe, als wenn Sie mit der Maus das Tiefen-Arrangement-Gadget eines Screens anklicken: Der betroffene Screen wird hinter allen anderen Screens versteckt.

ScreenToFront(Screen)

Der angegebene Screen wird vor alle anderen Screens manövriert.

Eingaben:
Screen Adresse der Screen-Struktur

Ausgaben:

C-Datentypen:
```
void ScreenToFront();
struct Screen *Screen;
```

Beschreibung:
Mit diesem Kommando erreichen Sie genau dasselbe, als wenn Sie mit der Maus das Tiefen-Arrangement-Gadget eines Screens anklicken: Der betroffene Screen wird vor alle anderen Screens gelegt.

Graphics-Funktionen:
ClearScreen(RastPort)

Löschen des gesamten Rasterports ab der aktuellen Grafikcursorposition.

Eingaben:
RastPort Adresse des angesprochenen Rasterports

Ausgaben:

C-Datentypen:
```
void ClearScreen();
struct RastPort *RastPort;
```

Beschreibung:
Diese Funktion löscht zunächst ab der momentanen Cursorposition (einstellbar mit Move( )) den Rest der Zeile bis zum rechten Bildschirmrand. Dann löscht sie ebenso den gesamten Rest des Rasters bis zum unteren Rand. Gelöscht wird mit 0. Im Zeichenmodus 2 mit der Farbe, die Sie durch SetBPen( ) setzen können.

Draw(RastPort, x, y)

Zeichnen einer Linie von der aktuellen Cursorposition nach x,y.

Eingaben:
RastPort Adresse des angesprochenen Rasterports
x, y Koordinaten des Endpunktes

Ausgaben:

C-Datentypen:

```
void Draw();
struct RastPort *RastPort;
long x;
long y;
```

Beschreibung:
Die vorgestellte Funktion zeichnet eine Linie (Farbe aus SetAPen) vom aktuellen Grafikcursor (mit Move() einstellbar, ansonsten der zuletzt gezeichnete Punkt) zum Punkt mit den Koordinaten x,y. Dieser Endpunkt der Linie wird gleichzeitig zum aktuellen Grafikcursor.

DrawCircle(RastPort, mx, my, radius)

Zeichnen eines Kreises mit dem Radius radius und dem Mittelpunkt bei mx,my.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
mx, my	Koordinaten des Kreismittelpunktes
radius	Größe des Radius in Pixel

Ausgaben:

C-Datentypen:

```
void DrawCircle();
struct RastPort *RastPort;
long mx;
long my;
long radius;
```

Beschreibung:
Die vorgestellte Funktion zeichnet einen Kreis (Farbe aus SetAPen) mit dem angegebenen Radius und den Mittelpunktkoordinaten mx, my.

DrawEllipse(RastPort, mx, my, x_radius, y_radius)

Zeichnen einer Ellipse mit den Radien x_radius, y_radius und dem Mittelpunkt bei mx,my.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
mx, my	Koordinaten des Kreismittelpunktes
x_radius	Größe des Radius in x-Richtung (in Pixel)
y_radius	Größe des Radius in y-Richtung (in Pixel)

Ausgaben:

C-Datentypen:

```
void DrawEllipse();
struct RastPort *RastPort;
long mx;
long my;
long x_radius;
long y_radius;
```

Beschreibung:
Die vorgestellte Funktion zeichnet eine Ellipse (Farbe aus SetAPen) mit den angegebenen Radien und den Mittelpunktkoordinaten mx, my.

Flood(RastPort, modus, x, y)

Ausfüllen einer beliebigen Fläche.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
x, y	Koordinaten eines Punktes in der Fläche
modus	Füllmodus:
	= 0: Ausfüllen der zusammenhängenden Fläche, die die Farbe des Punktes bei x,y besitzt.
	= 1: Ausfüllen der zusammenhängenden Fläche, die von Punkten umrahmt ist, die mit SetOPen() gesetzt wurden.

Ausgaben:

C-Datentypen:

```
void Flood();
struct RastPort *RastPort;
long x;
long y;
long modus;
```

Beschreibung:
Mit dieser Funktion können Sie schnell beliebige Flächen ausfüllen. Dazu geben Sie mit x,y einen Punkt innerhalb dieser Fläche an. Wie der Rechner die Grenzen einer Fläche erkennt, das bestimmen Sie mit »modus«. Entweder Sie möchten nur die Fläche ausfüllen, die zur Zeit eine bestimmte Farbe hat (modus = 0), nämlich die des Punktes mit den Koordinaten x,y. In diesem Fall zeichnet die Funktion bis dorthin, wo sie eine andere Farbe erkennt.

Im anderen Modus wird so lange ausgefüllt, bis die Funktion an einen Randpunkt gelangt, der die mit SetOPen gesetzte OUTLINE-Farbe besitzt. Das ist wohl der am meisten verwendete Modus.

Move(RastPort, x, y)

Verschieben des Grafikcursors nach x,y (kein Zeichnen)

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
x, y	Koordinaten des neuen Grafikcursors

Ausgaben:

C-Datentypen:

```
void Move();
struct RastPort *RastPort;
long x;
long y;
```

Beschreibung:

Versetzt den internen unsichtbaren Grafikcursor (normalerweise der zuletzt gezeichnete Punkt) auf den Punkt mit den Koordinaten x,y.

RectFill(RastPort, x1, y1, x2, y2)

Zeichnen eines ausgefüllten Rechtecks.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
x1, y1	Koordinaten des obersten linken Eckpunktes
x2, y2	Koordinaten des untersten rechten Eckpunktes

Ausgaben:

C-Datentypen:

```
void RectFill();
struct RastPort *RastPort;
long x1;
long y1;
long x2;
long y2
```

Beschreibung:

Die genannte Funktion zeichnet ein mit der durch SetAPen( ) gesetzten Farbe ausgefülltes Rechteck. Die Größe und die Position des Rechtecks definieren Sie durch die Angabe zweier gegenüberstehender Eckpunkte x1,y1 und x2,y2.

SetAPen(RastPort, pal_reg)

Setzen der aktuellen Zeichenfarbe auf die Farbe im Palettenregister pal_reg.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
pal_reg	Palettenregisternummer

Ausgaben:

C-Datentypen:

```
void SetAPen();
struct RastPort *RastPort;
long pal_reg;
```

Beschreibung:
SetAPen( ) definiert ein neues Palettenregister als Quelle der momentanen Zeichenfarbe, mit der ja alle normalen Zeichenoperationen ausgeführt werden.

SetBPen(RastPort, pal_reg)

Setzen der aktuellen Hintergrundfarbe auf die Farbe im Palettenregister pal_reg.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
pal_reg	Palettenregisternummer

Ausgaben:

C-Datentypen:

```
void SetAPen();
struct RastPort *RastPort;
long pal_reg;
```

Beschreibung:
SetBPen( ) definiert ein neues Palettenregister als Quelle der momentanen Hintergrundfarbe, die von verschiedenen Zeichenbefehlen verwendet wird.

SetOPen(RastPort, pal_reg)

Setzen der aktuellen OUTLINE-Farbe auf die Farbe im Palettenregister pal_reg.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
pal_reg	Palettenregisternummer

Ausgaben:

C-Datentypen:

```
void SetOPen();
struct RastPort *RastPort;
long pal_reg;
```

Beschreibung:

SetOPen( ) definiert ein neues Palettenregister als Quelle der momentanen OUTLINE-Farbe, mit der manche Zeichenbefehle arbeiten. Gleichzeitig schaltet es die Umrahmung für Flächenfülloperationen ein. SetOPen( ) ist keine eigentliche Funktion, sondern nur ein Makro (mit #define unter C im Include-File graphics/gfxmakros.h definiert). In Basic müssen Sie deshalb die folgende Befehlssequenz statt dessen verwenden:

```
POKE WINDOW(8)+27,pal_reg 'Farbe setzen
flags = WINDOW(8)+32
POKEW flags, PEEKW(flags) OR 8 'OUTLINE-Flag ein
```

SetDrMd(RastPort, modus)

Setzen des aktuellen Zeichenmodus.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
modus	Zeichenmodus: Die folgenden Werte bestimmen den Zeichenmodus und können durch Addition (Oder-Verkn.) kombiniert werden:

= 0: (JAM1)	Es wird nur mit der Zeichenfarbe (SetAPen( )) gezeichnet. Soll ein Punkt in der Hintergrundfarbe gesetzt werden, dann wird er ignoriert. Ein Füllmuster z.B. erscheint dann transparent an den Stellen, an denen das Muster Null-Bits enthält.
= 1: (JAM2)	Es werden sowohl Zeichenfarbe (SetAPen( )) als auch Hintergrundfarbe (SetBPen( )) gesetzt. Ein Füllmuster beispielsweise verdeckt alle unter ihm liegende Grafik auf diese Weise vollständig.
=2: (COMPLEMENT)	Führt eine Exklusiv-Oder-Verknüpfung (XOR) der Palettenregisternummer mit den Farben aller angesprochenen Punkten durch. Das können z.B. alle Punkte sein, die nur durch die Vordergrundfarbe angesprochen werden sollten (Kombination JAM1+COMPLEMENT).
=4: (INVERSIVD)	Hintergrund- und Vordergrundfarbe werden miteinander vertauscht.

Ausgaben:

C-Datentypen:

```
void SetDrMd();
struct RastPort  * RastPort;
long modus;
```

Beschreibung:
Der Befehl stellt den gewünschten Zeichenmodus ein, der von allen Zeichenfunktionen berücksichtigt wird.

SetRGB4(ViewPort, pal_reg, rot, gruen, blau)

Weist einem Palettenregister des ViewPort (Screen) eine Farbkombination zu.

Eingaben:

ViewPort	Adresse des angesprochenen Viewports. Unter Intuition steht der Viewport für den Screen in der Screenstruktur. Sie können also in C angeben: &Screen- > ViewPort
pal_reg	Palettenregisternummer
rot	Intensitätswert für den Rotanteil
gruen	Intensitätswert für den Grünanteil
blau	Intensitätswert für den Blauanteil

Ausgaben:

C-Datentypen:

```
void SetRGB4();
struct ViewPort *ViewPort;
long pal_reg;
long rot;
long gruen;
long blau;
```

Beschreibung:
Mit dieser Funktion können Sie einem Palettenregister eine der insgesamt 4096 möglichen Farbkombinationen zuweisen.

error = Text(RastPort, string, anz_zei)

Ausgabe eines Textstrings im aktuellen Zeichensatz.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
string	Adresse des auszugebenden Textstrings
anz_zei	Anzahl der Zeichen im String

Ausgaben:

error	= 0:	kein Fehler
	< >0:	Fehler

C-Datentypen:
```
BOOL Text();
struct RastPort *RastPort;
char *string;
long anz_zei;
```

Beschreibung:
Text_ gibt einen Text an der aktuellen Grafikcursorposition aus.

WaitTOF_

Warten, bis der Elektronenstrahl den nächsten View (Screen) erreicht hat.

Eingaben:

Ausgaben:

C-Datentypen:
```
void WaitTOF_;
```

Beschreibung:
Mit dieser Funktion haben Sie die Möglichkeit, Ihre Grafikausgaben mit dem Elektronenstrahl des Monitors zu synchronisieren. Sie wartet, bis dieser das Ende Ihres Screens erreicht hat. Auf diese Weise können Sie in der Zeit, die der Strahl bis zum Beginn Ihres Screens braucht, Grafiken flackerfrei zeichnen.

WritePixel(RastPort, x, y)

Setzen eines Punktes bei x,y.

Eingaben:

RastPort	Adresse des angesprochenen Rasterports
x, y	Koordinaten des zu setzenden Punktes

Ausgaben:

C-Datentypen:
```
void WritePixel();
struct RastPort *RastPort;
long x;
long y;
```

Beschreibung:
Mit dieser Funktion setzen Sie einen einzigen Punkt auf dem Bildschirm. Er nimmt die aktuelle Zeichenfarbe an. Gleichzeitig wird dieser Punkt zum aktuellen Grafikcursor.

Literaturhinweise

Grafik und 3-D:

David F. Rogers
Procedural Elements for
Computer Graphics
Mc Graw-Hill Book Company
ISBN 0-07-053534-5

Roy A. Plastock/Gordon Kalley
Schaum's Outline Series
Computer Graphics
Mc Graw-Hill Book Company
ISBN 0-07-050326-5

Steven Harrington
Computer Graphics
A Programming Approach
Mc Graw-Hill Book Company
ISBN 0-07-026751-0

Günter Spur/Frank-Lothar Krause
CAD-Technik
Carl Hanser Verlag München Wien
ISBN 3-446-13897-8

William M. Newman/Robert F. Sproull
Grundzüge der interaktiven Computergrafik
Mc Graw-Hill Book Company
ISBN 3-89028-015-3

Roy A. Plastock/Gordon Kalley
Theory and problems of
Computer Graphics
Mc Graw-Hill Book Company
ISBN 0-07-050326-5

Braun
Atari ST
3D Grafik Programmierung
DATA BECKER GmbH
ISBN 3-89011-130-0

Faux/Pratt
Computational Geometry for Design and Manufacture
Ellis Horwood Limited
ISBN 0-85312-114-1

Encarnacao
Computer Aided Design-Modelling, Systems Engineering,
CAD-Systems
Springer-Verlag
ISBN 0-387-10242-6

Encarnacao
Computer-Graphics – Programmierung und Anwendung von
grafischen Systemen
R. Oldenbourg Verlag
ISBN 3-486-34651-2

Brodlie
Mathematical Methods in Computer Graphics and Design
Academic Press
ISBN 0-12-134880-6

Barnhill/Boehm
Surfaces in Computer Aided Geometric Design
North-Holland
ISBN 0-444-86550-0

Myers
Microcomputer Graphics
Addison-Wesley Publishing Company
ISBN 0-201-05092-7

Günter Pomaska
Computergrafik 2D- und 3D-Programmierung
Vogel-Buchverlag
ISBN 3-8023-0759-3

Amiga-(Grafik-)Programmierung:

Amiga ROM Kernel Reference Manual:
Libraries and Devices
Amiga Technical Reference Series
Addison-Wesley Publishing Company
ISBN 0-201-11078-4

Amiga ROM Kernel Reference Manual: Exec
Amiga Technical Reference Series
Addison-Wesley Publishing Company
ISBN 0-201-11099-7

Amiga Intuition Reference Manual
Amiga Technical Reference Series
Addison-Wesley Publishing Company
ISBN 0-201-11076-8

Amiga Hardware Reference Manual
Amiga Technical Reference Series
Addison-Wesley Publishing Company
ISBN 0-201-11077-6

Frank Kremser/Jörg Koch
Amiga Programmierhandbuch
Markt & Technik Verlag AG
ISBN 3-89090-491-2

Sheldon Leemon
Inside Amiga Graphics
Compute! Publications
ISBN 0-87455-040-8

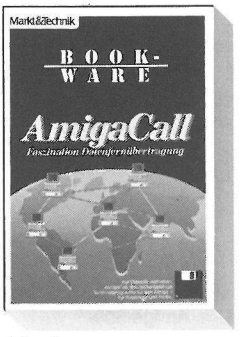
Programmiersprachen:

H. Herold/W. Unger
Das C-Buch
te-wi Verlag
ISBN 3-921803-62-4

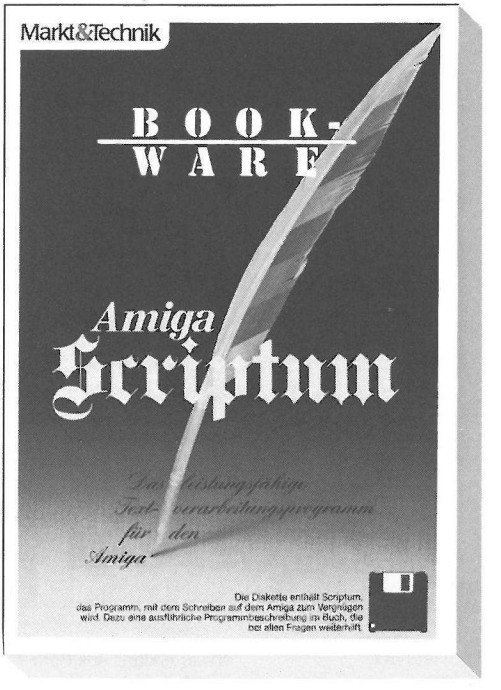
Werner Hilf/Anton Nausch
M68000 Familie
te-wi Verlag
ISBN 3-921803-16-0

Stichwortverzeichnis

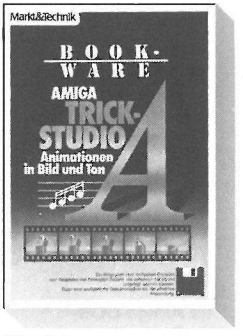
Bücher zum Amiga



Atlantis
AmigaCall
Treten Sie ein in die faszinierende Welt der Datenfernübertragung. Kommunizieren Sie über Mailboxen mit erfahrenen Computer-Anwendern, die Ihnen bei Ihren Problemen weiterhelfen können, oder Sie erhalten auf diesem Wege leistungsfähige Public-Domain-Software. AmigaCall nimmt Ihnen die meiste Arbeit ab. Schließen Sie Ihr Modem oder Ihren Akustikkoppler an, starten Sie AmigaCall - und auf geht's.
1988, 133 Seiten, inkl. 3½"-Programmdiskette
Bestell-Nr. 90716
ISBN 3-89090-716-4
DM 99,–*
(sFr 91,–*/öS 842,–*)



R. Arbinger/I. Krüger
Scriptum
Scriptum - das schnelle, leistungsfähige Textverarbeitungssystem für den Amiga. Ausführliche Bedienungsanleitung im Buch. Für alle, die auf dem Amiga Texte verarbeiten wollen.
1989, ca. 200 Seiten, inkl. 3½"-Programmdiskette
Bestell-Nr. 90650
ISBN 3-89090-650-8
DM 79,–*
(sFr 72,70*/öS 672,–*)
* Unverbindliche Preisempfehlung



Atlantis
Trickstudio A
Ob Sie Computerfilm-Pionier sind oder Trickprofi, ob Sie von Walt Disney inspiriert sind oder einfach nur einen guten Lehrfilm für technische Abläufe benötigen: Mit Trickstudio A können Sie Ihre eigenen Trickfilme erstellen und diese mit Sound und Geräuschen untermalen.
Wie wäre es also mit einem Stummfilm-Slapstick, einem Krimi oder einem Werbefilm für Ihr Schaufenster? Dazu Ihre Lieblingsmusik oder digitalisierte Stimmen? Entwerfen Sie die Einzelbilder, z. B. mit Deluxe Paint, erstellen Sie eine Sounddatei und dann: Klappe - Film; die erste.
1988, 86 Seiten, inkl. 3½"-Programmdiskette
Bestell-Nr. 90715
ISBN 3-89090-715-6
DM 99,–*
(sFr 91,–*/öS 842,–*)

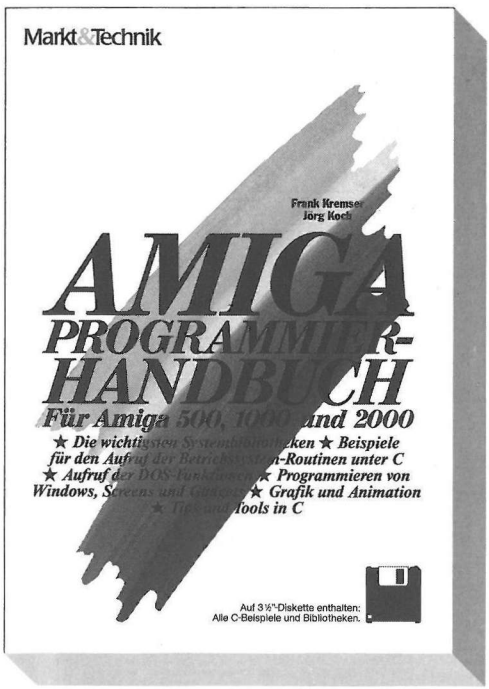
Markt&Technik
Zeitschriften · Bücher
Software · Schulung

427/901

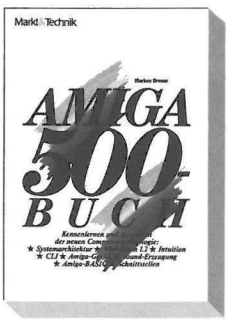
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bücher zum Amiga

Commodore-Amiga Inc.
Das Amiga-DOS-Handbuch für Amiga 500, 1000 und 2000
1988, 342 Seiten
Die Pflichtlektüre für jeden Commodore-Amiga-Anwender und Programmierer: eine Entwickler-Dokumentation zum Amiga-DOS-Betriebssystem, Version 1.2. Programmierung, interne Datenstruktur und Diskettenhandling. Mit diesem Buch lernen Sie das mächtige Amiga DOS schnell und sicher zu beherrschen. Alle Möglichkeiten des Systems, bis hin zum »Multi-Tasking« werden ausführlich und anschaulich beschrieben.
Best.-Nr. 90465
ISBN 3-89090-465-3
DM 59,–



Kremser/Koch
Amiga Programmierhandbuch
1987, 387 Seiten,
inkl. Diskette
Eine tolle Einführung in die »Interna« des Amiga: Die wichtigsten Systembibliotheken, die das Betriebssystem zur Verfügung stellt, werden anhand vieler Beispiele erklärt. Aus dem Inhalt: Aufruf der Betriebssystem-Routinen unter C, Aufruf der DOS-Funktionen, Programmieren von Windows, Screens und Gadgets, Grafik und Animation, Tips und Tools in C.
Best.-Nr. 90491
ISBN 3-89090-491-2
DM 69,–



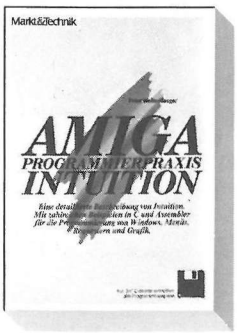
M. Breuer
Das Amiga-500-Handbuch
1987, 489 Seiten
Eine ausführliche Einführung in die Bedienung des Amiga 500. Kennenlernen und Anwenden der neuen Computer-Technologie: Systemarchitektur, Workbench 1.2, Intuition, CLI, Amiga-Grafik, Sound-Erzeugung, Amiga-BASIC und Schnittstellen. Neben dem Handbuchteil mit vielen Bildschirmfotos und Übersichtstabellen, die Ihnen beim täglichen Einsatz helfen, schnell und reibungslos zu arbeiten, enthält das Buch eine ausführliche Beschreibung des Amiga 500 und seines Zubehörs.
Best.-Nr. 90522
ISBN 3-89090-522-6
DM 49,–

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

712328

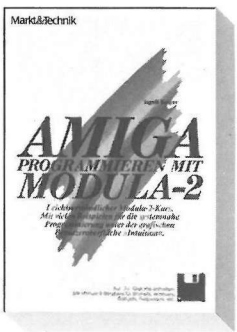
Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bücher zum Amiga



P. Wollschlaeger
Amiga: Programmierpraxis Intuition
Eine detaillierte Beschreibung von Intuition! Neben der Programmierung von Fenstern, Menüs und Grafiken behandelt der Autor auch wichtige Randgebiete, wie die Ein- und Ausgabe von Texten oder Zugriff auf die Diskette.
Sie erfahren, wie ein Programm zu gestalten ist, damit es sowohl unter CLI als auch unter Intuition läuft und Multitasking-fähig ist.
Mit allen Beispielen für die Programmierung von Windows, Menüs, Requestern und Grafik auf Diskette.
1988, 330 Seiten, inkl. Diskette
Bestell-Nr. 90593
ISBN 3-89090-593-5
DM 69,–
(sFr 63,50/öS 538,–)

P. Wollschlaeger
Amiga-Assembler-Buch
Nach einem Minimum an Theorie geht dieses Buch sofort in die Praxis. Aus dem Inhalt: Grundlagen des 68000er, Systemprogrammierung, Programmierung von Intuition, schnelle Grafik in Farbe, alle Systemroutinen mit Parametern.
1987, 329 Seiten, inkl. Diskette
Bestell-Nr. 90525
ISBN 3-89090-525-0
DM 59,–
(sFr 54,30/öS 460,–)



I. Krüger
Amiga: Programmieren mit Modula 2
Leichtverständlicher Modula-2-Kurs! Mit vielen Beispielen für die systemnahe Programmierung unter der grafischen Benutzeroberfläche »Intuition«. Aus dem Inhalt: Programm-Module, Variablendeklaration, Strukturanweisungen, Prozeduren, lokale und externe Module, Verwendung von Zeigern, systemnahe Programmierung, Coroutinen (Verarbeitung von parallelen Prozessen), Programmierung unter Intuition (Screens, Windows, Gadgets, Requester).
1988, 350 Seiten, inkl. Disk.
Bestell-Nr. 90554
ISBN 3-89090-554-4
DM 69,–
(sFr 63,50/öS 538,–)

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

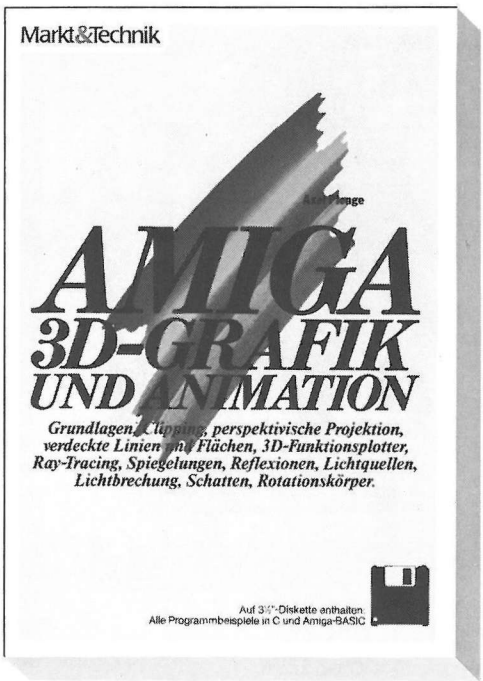
Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

428/901

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Bücher zum Amiga

H. Knappe
Fraktale Grafik auf dem Amiga
Das Thema dieses Buches wird den meisten als ungewöhnlich erscheinen, denn es führt in die Grenzbereiche des heutigen Wissens der Mathematik und Technik.
Grundlegende Kenntnisse der Programmiersprache C und ihrer Anwendung auf dem Amiga werden vorausgesetzt. Für die nachträgliche Veränderung berechneter Bilder ist es sinnvoll, ein Malprogramm (z. B. Deluxe Paint) zu besitzen.
1988, 278 Seiten,
inkl. Diskette
Bestell-Nr. 90600
ISBN 3-89090-600-1
DM 79,–
(sFr 72,70/öS 616,-)



A. Plenge
Amiga 3-D-Grafik und Animation
Angefangen bei den einfachsten Problemstellungen lernen Sie, professionelle 3-D-Grafiken auf Ihrem Commodore Amiga zu planen, zu programmieren und darzustellen.
1988, 376 Seiten,
inkl. Diskette
Bestell-Nr. 90526
ISBN 3-89090-526-9
DM 69,–
(sFr 63,50/öS 538,-)

H. R. Henning
Grafik mit Amiga-Basic
Dieses Buch ist speziell der Grafik-Programmierung auf dem Amiga gewidmet. Der erste Teil stellt für den Anfänger alle bekannten Grafik-Befehle des Amiga-Basic vor. Mit Beginn des zweiten Teiles werden die Routinen des Betriebssystems zur Grafik-Programmierung herangezogen. Damit werden die Möglichkeiten des Basic um ein Vielfaches erweitert, und es sind Geschwindigkeiten möglich, die kaum vermuten lassen, daß dabei ein Basic-Programm abläuft.
1989, ca. 300 Seiten,
inkl. Diskette
Bestell-Nr. 90669
ISBN 3-89090-669-9
DM 59,–
(sFr 54,30/öS 460,-)

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

429/901

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Amiga-Software

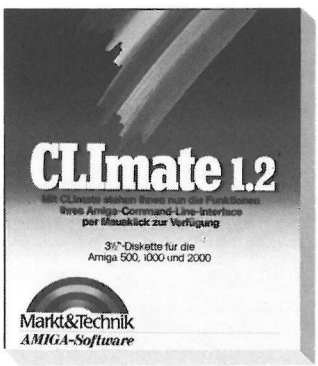
CLImate 1.2

Jetzt stehen Ihnen die Funktionen Ihres Amiga-Command-Line-Interface per Mausklick zur Verfügung!

Mit diesem Programm können Sie die Befehle des Command-Line-Interface (CLI) benutzerfreundlich und schnell per Mausklick verwenden!

Ihre Super-Vorteile mit CLImate 1.2:

- sehr große Übersichtlichkeit der Bildschirmdarstellung (Sie haben alle Funktionen auf einen Blick)
- leichte Bedienung aller Befehle mit der Maus
- drei externe Laufwerke (3½" oder 5¼"), zwei Festplatten, RAM-Disk unterstützen Sie
- schnelle Directory-Anzeige
- Sie können Disketten leicht nach Texten, Bildern u.ä. durchsuchen
- Dateien lassen sich mit Pause/Continue-Möglichkeit betrachten
- Ausdrucken von Dateien auf Drucker
- Informationen über die Disketten (Programmlänge und ähnliches)
- Betrachten von Bildern im IFF-Format (inklusive HAM)



Bestell-Nr. 51653

DM 79,-*
(sFr 72,-*/öS 990,-*)
*Unverbindliche Preisempfehlung

- Sie können Dateien aus beliebigen Verzeichnissen in andere Verzeichnisse kopieren
- Bildschirmausgabe von Dateien in ASCII und in hexadezimaler Form
- Unterstützung von Jokerzeichen bei Disketten- und Dateioperationen

CLImate 1.2 - das unentbehrliche Programm für den Amiga-500-, Amiga-1000- und Amiga-2000-Besitzer.

Am besten gleich bestellen!

Hardware-Anforderungen: Amiga 500, 1000 oder 2000 mit mindestens 512 Kbyte Hauptspeicher. Empfohlene Hardware: Farbmonitor. Software-Anforderungen: Kickstart 1.2 (oder ROM bei Amiga 500 und 2000), Workbench 1.2. Eine 3½"-Diskette für den Amiga 500, 1000 und 2000.

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

801271

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0

Amiga-Software

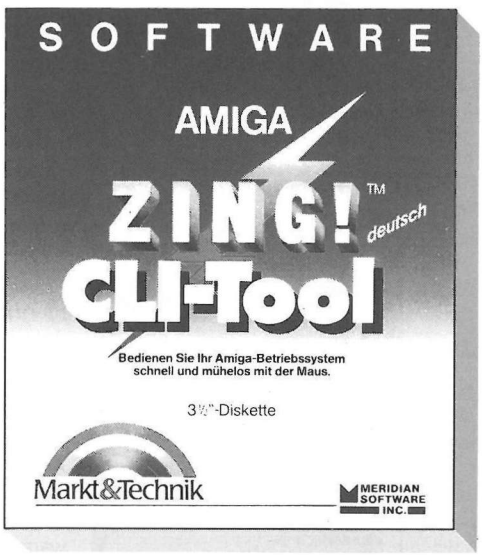
ZING!

Das mächtige CLI-Werkzeug

Haben Sie das Eintippen satt? Zing! ermöglicht Ihnen den mausgestützten Zugriff auf Ihr Amiga-Betriebssystem. Dieses Programm übernimmt die lästige und fehleranfällige Tipparbeit beim Arbeiten mit dem Betriebssystem Ihres Amiga. Zing! befindet sich nach dem erstmaligen Abrufen im Hintergrund und kann mit Hilfe von sogenannten »Hotkeys« jederzeit in Aktion treten. Volle Multitasking-Fähigkeit ist selbstverständlich. Wahlweise über Maus oder Funktionstasten stehen Ihnen speicherresident unter anderem folgende Funktionen zur Verfügung:

- Verzeichnis wechseln - Anzeigen eines Dateibaums - Dateien kopieren - Dateien umbenennen - Dateien schreibschützen - Restspeicheranzeige - Dateien löschen - Dateien zusammenführen - Dateien verlagern - Verzeichnisse erstellen - Dateikommentar erstellen - Systemstatusanzeige - automatische Bildschirmabschaltung (Screen Saver) ... und vieles mehr! Die Auswahl der Dateien kann mit der Maus vorgenommen werden, mögliche Kriterien sind zum Beispiel auf Dateinamen basierende Sortiermuster oder der Zeitpunkt der Dateierstellung. Verzeichnisanzeige mit Schnellsortierdurchlauf ist bei Zing! genauso selbstverständlich wie die Möglichkeit, sowohl ganze Dateibäume als auch Teile von ihnen zu kopieren. Zusätzlich enthält das Programm viele nützliche Dienstprogramme, zum Beispiel:

- Druckerspooler - Bildschirmausdruck - Speichern eines Bildschirms als IFF-Grafik - Überwachung von anderen Programmen
- Umbelegung der Funktionstasten - interne Symbolzuweisung
- Diskcopy-Funktion - Disketten installieren - Disketten umbenennen - Disketten formatieren - direkter Aufruf von Programmen



Bestell-Nr. 51670

DM 189,–*

(sFr 169,–*/öS 2290,–*)

* Unverbindliche Preisempfehlung

Lieferumfang:

- deutsche Programmversion auf 3½"-Diskette
- Handbuch deutsch

Hardware-Anforderungen:

- Amiga 500, 1000 oder 2000

Software-Anforderung
(speziell für Amiga 1000)

- Kickstart 33.180 (Version 1.2) oder höher

Markt&Technik
Zeitschriften · Bücher
Software · Schulung

Markt&Technik-Produkte erhalten Sie bei Ihrem Buchhändler, in Computer-Fachgeschäften oder in den Fachabteilungen der Warenhäuser.

802322

Markt&Technik Verlag AG, Buchverlag, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 4613-0