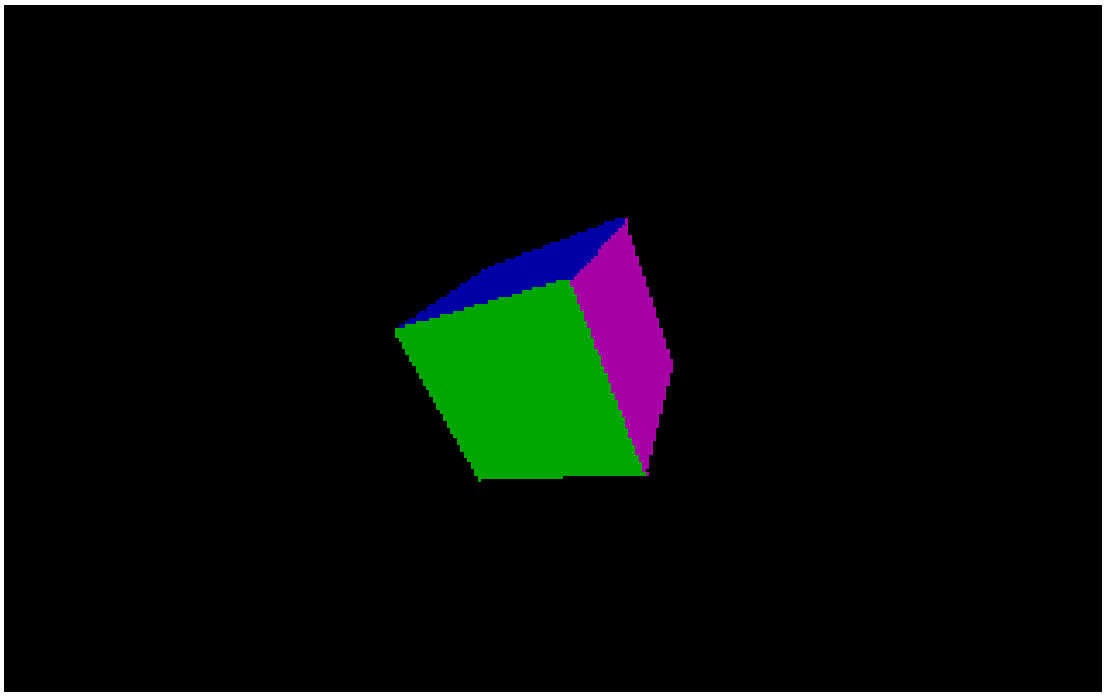
Inf - Spezialgebiet: **GEHRI Thomas**

3D - Grafik **8C^{Rg} 6 1996/97**



3D - Grafik

3D - Grafik

Grafik-Hardware:

Der erste IBM-PC wurde 1981 zusammen mit einem MDA (Monochrome Display Adapter) ausgeliefert. Diese Karte verfügte nur über einen 80 x 25 - Textmodus und sehr wenig Video-RAM. Grafikdarstellung wurde vom MDA noch nicht unterstützt.

Ebenfalls 1981 kam der CGA (Color Graphics Adapter) auf den Markt. Diese Karte konnte, wegen der niedrigen Frequenzen, noch über einen speziellen Ausgang an einen normalen Fernseher angeschlossen werden. Im Textmodus schaffte der CGA, genau wie MDA, 80 x 25 Zeichen, die aber auf einer kleineren Punktmatrix basierten. An Grafikmodi standen 320 x 200 (Punkte) x 4 (Farben) sowie 640 x 200 x 2 zur Verfügung. MDA und CGA basierten auf dem MC6845 von Motorola.

1982 erschien die weitestgehend zum MDA kompatible Hercules Graphics Card. Sie konnte zusätzlich noch zwei Grafikseiten bei einer Auflösung von 720 x 348 monochrom darstellen. Diese Karte war sehr einfach zu programmieren und verfügte auch über eine parallele Schnittstelle. Sie stellt heute den Monochrom-Standard dar.

1985 wurde der EGA (Enhanced Graphics Adapter) vorgestellt. Er ist voll kompatibel zu MDA und CGA und kann monochrome und farbige Grafiken darstellen. Im Grafikmodus schafft die EGA-Karte bei 640 x 350 Punkten 16 Farben gleichzeitig aus einer Palette von 64. Der Video-RAM ist standardmäßig 64 KB groß, die Karte kann aber mit bis zu 256 KB bestückt werden, um mehrere Grafikseiten im Speicher unterzubringen. Das Bild ist wesentlich schärfer als bei einer CGA-Karte, außerdem kann der EGA mit variablen Schriftsätzen arbeiten. Erstmals hat die EGA-Karte ein eigenes ROM-BIOS onboard, um den Zugriff auf die erweiterten Leistungsmerkmale zu erleichtern, auch wenn die Programmierung über ROM-Routinen langsam ist.

1987 wurde schließlich der VGA (Video Graphics Array) vorgestellt, der wiederum zu allen seinen Vorgängern kompatibel ist. Die VGA-Karte ist heute Standard und der kleinste

gemeinsame Nenner, zu dem alle SVGA-Karten kompatibel sein müssen. Der VGA sendet erstmals ananloge Signale an den Monitor, was eine große Farbenvielfalt ermöglicht. Die höchste Standardauflösung ist 640 x 480 x 16, im Mode 13h können bei 320 x 200 Bildpunkten 256 Farben gleichzeitig aus einer Palette von 262.144 dargestellt werden. Dieser Modus diente ursprünglich der Kompatibilität zu MCGA, wobei der VGA mit einem Trick durch Setzen von zwei Registern die tatsächliche Auflösung von 640 x 400 halbiert. Grafikprofis verschafften sich dadurch bald durch Löschen der beiden Register einen sehr einfachen hochauflösenden Modus. Ursprünglich wurden VGA-Karten mit 256 KB Video-RAM bestückt, heute sind aber auch 8 MB keine Seltenheit mehr.

Diese Karten sind sämtlich bereits SVGA (Super VGA) -Karten, die erweiterte Möglichkeiten bieten, die aber leider nicht genormt sind. Die Auflösungen reichen bis 1600 x 1200 Bildpunkte, bei bis zu 16,7 Millionen Farben gleichzeitig. Dabei sind bei Karten der Luxusklasse Bildwiederholraten von bis zu 200 Hz problemlos möglich, sofern dies der Monitor verkraftet.[1]

Außerdem verfügen heute viele Karten auch schon über intelligente Grafik-Prozessoren, die eine ganze Menge zusätzliche Möglichkeiten bieten. In Verbindung mit den neuen MMX-Prozessoren (mit erweitertem Befehlssatz, speziell für Multimedia-Anwendungen) sind deshalb in naher Zukunft noch weitere Leistungssteigerungen zu erwarten. Windows-Beschleuniger erledigen die typischen Windows-Funktionen nur mehr auf der Grafikkarte, ohne den Hauptprozessor zu belasten. AVI-Beschleiniger kümmern sich speziell um die “Briefmarken-Videos” und bieten häufig auch Hardware-Zoom. MPEG-Decoder spielen MPEG-Videos ab, ohne den Prozessor zu belasten. Dabei werden Videos manchmal, speziell in Verbindung mit einer Video-Grabber-Karte, auch als Hardware-Overlays über das normale Bild gelegt. 3D-Beschleuniger bieten heute auch schon hardwaremäßige 3D-Funktionen, die komplett auf der Grafikkarte Perspektive, Schatten, Texturen, Filter und vieles mehr erledigen.

Leider sind alle diese Funktionen nicht genormt. Deshalb wurde für DOS 1989 das VESA-Komitee (Video Electronics Standard Association) gegründet, das den VESA-Standard entwickelte. Dieser wird entweder über das Video-BIOS oder einen eigenen Treiber implementiert und bietet eine genormte Schnittstelle zum Zugriff auf SVGA-Karten und mit

[1] PC intern 4, Michael Tischer, S. 311-318

der Version 2.0 auch auf viele Zusatzfunktionen. Unter Windows stellen sich alle diese Probleme nicht, da hier einfach jeder Karten-Hersteller einen Treiber für seine Hardware mitliefert, der sich um die optimale Darstellung der Daten aus dem Windows-GDI kümmert.[2]

Speicher:

Für den Video-RAM gibt es mehrere Möglichkeiten. Ursprünglich wurden DRAMs verwendet, welche allerdings relativ langsam sind. Die nächste Generation bildeten VRAMs, die gleichzeitig ausgelesen und beschrieben werden konnten. Sie sind allerdings auch empfindlich teurer. Ab dann wird die Entwicklung chaotisch. Der Grafikkarten-Hersteller Matrox erfand WRAMs, die noch schneller und angeblich sogar billiger sind. Manche Hersteller verwenden auch EDO-RAM, das einen Cache gleich eingebaut hat. Heute gibt es weiters SGRAM und Multibank-RAM, und jeder meint den Stein der Weisen gefunden zu haben. Doch egal wie sie bestückt sind, sind die heutigen Karten generell um einiges schneller als noch vor einem Jahr, und die Entwicklung ist noch lange nicht abgeschlossen.
Fürs Programmieren werden bei Standard-VGA Teile des Video-RAMs, die über Register selektiert werden, in den konventionellen Speicher eingeblendet. Dabei liegt der Grafikbereich zwischen A000 und AFFF, der Monochrom-Textbereich (der häufig von manchen Memorymanagern verwendet wird) zwischen B000 und B7FF und der Farb-Textbereich zwischen B800 und BFFF. Auf der Grafikkarte wird die lineare Adressierung dann in eine planare umgewandelt und die Punkte abhängig von den Registereinstellungen gesetzt. Bei SVGA ist das teilweise etwas anders, zum Beispiel werden manchmal Selektoren verwendet, um Speicherbereiche zu adressieren.[3]

Windows:

Unter Windows ist die Grafikprogrammierung deutlich einfacher, da es eine einheitliche Schnittstelle bietet. Programme schreiben ihre Grafikdaten einfach ins GDI (Graphical Device Interface), einen reservierten Systembereich, und Windows kümmert sich dann um die korrekte Darstellung aller Fenster gemäß den aktuellen Desktop-Einstellungen. Die Ansteuerung der Grafikkarte obliegt dabei einem meist vom Hersteller mitgelieferten Treiber,

[2] PC intern 4, S. 621-630
[3] PC intern 4, S. 355-359

der die Karte optimal ausnutzt. Nur für Spezialfunktionen müssen immer noch zusätzliche Treiber herhalten.

Da diese Methode der Grafikausgabe auch relativ langsam ist, bietet Windows 95 mit DirectX eine eigene Funktionsbibliothek, die eine direkte Ansteuerung der Hardware ermöglicht. Auch die Bibliotheken WinG und Win32s bieten einige Möglichkeiten, und Video für Windows ist bei Windows 95 gleich fix eingebaut. Für 3D-Effekte besonders wichtig ist Direct3D, ein Bestandteil von DirectX, der die Möglichkeiten spezieller 3D-Karten ausnutzt.

VGA-Register:

Für professionelle Programmierung ist es unerläßlich, die Standard-Register einer VGA-Karte zwecks direkter Manipulation zu kennen. Zwar ist es sinnvoll, nur bei Bedarf die benötigten nachzuschlagen, doch sollen hier der Übersicht halber einmal alle kurz vorgestellt werden. Vorsicht ist bei Änderungen des VGA-Timings geboten, da hiermit der Monitor irreparabel, wenn auch manchmal spektakulär, zerstört werden kann. Aus diesem Grund sind die CRTC-Register 0 bis 7 über das Protection-Bit 7 des CRTC-Registers 11h geschützt.[4]

Einzelregister:	Lesen	Schreiben
Miscellaneous Output Register	3CCh	3C2h
Input Status Register 0	3C2h	
Input Status Register 1	3DAh	
Indizierte Register:	Index-Register	Daten-Register
Cathod Ray Tube Controller (CRTC)	3D4h	3D5h

CRTC-Register 0: Horizontal Total
CRTC-Register 1: Horizontal Display End
CRTC-Register 2: Horizontal Blank Start
CRTC-Register 3: Horizontal Blank End
CRTC-Register 4: Horizontal Sync Start
CRTC-Register 5: Horizontal Sync End
CRTC-Register 6: Vertical Total
CRTC-Register 7: Overflow
CRTC-Register 8: Initial Row Address
CRTC-Register 9: Maximum Row Address
CRTC-Register 0Ah: Cursor Start-Zeile
CRTC-Register 0Bh: Cursor End-Zeile
CRTC-Register 0Ch: Linear Starting Address High
CRTC-Register 0Dh: Linear Starting Address Low
CRTC-Register 0Eh: Cursor Address High
CRTC-Register 0Fh: Cursor Address Low
CRTC-Register 10h: Vertical Sync Start
CRTC-Register 11h: Vertical Sync End
CRTC-Register 12h: Vertical Display End
CRTC-Register 13h: Row Offset
CRTC-Register 14h: Underline Location
CRTC-Register 15h: Vertical Blank Start

[4] PC underground, Boris Bertelsons, Mathias Rasch, S. 89-114

CRTC-Register 16h: Vertical Blank End
CRTC-Register 17h: CRTC Mode
CRTC-Register 18h: Line Compare (Split Screen)

Timing Sequencer (TS)	3C4h	3C5h
TS-Register 0: Synchroner Reset		
TS-Register 1: TS Mode		
TS-Register 2: Write Plane Mask		
TS-Register 3: Font Select		
TS-Register 4: Memory Mode		
Graphics Data Controller (GDC)	3CEh	3CFh
GDC-Register 0: Set/Reset		
GDC-Register 1: Enable Set/Reset		
GDC-Register 2: Color Compare		
GDC-Register 3: Function Select		
GDC-Register 4: Read Plane Select		
GDC-Register 5: GDC Mode		
GDC-Register 6: Miscellaneous		
GDC-Register 7: Color Care		
GDC-Register 8: Bit Mask		

Attribute Controller (ATC)
Schreibzugriffe: 3C0h → *Index/Data-Flip-Flop*
Lesezugriff auf *Input Status Register 1* → Index-Mode
Schreibzugriff: Index auf 3C0h, anschließend Daten-Byte auf gleichen Port
Lesezugriff: Index geschrieben, 3C1h → Daten-Byte (3C0h → Index)
ATC-Register: Index/Data
ATC-Register 0 - F: Palette Ram
ATC-Register 10h: Mode Control
ATC-Register 11h: Overscan Color
ATC-Register 12h: Color Plane Enable
ATC-Register 13h: Horizontal Pixel Panning
ATC-Register 14h: Color Select

Digital to Analog Converter (DAC)	
Pixel Mask	3C6h
Pixel Write Address	3C8h
Pixel Read Address	3C7h
Pixel Color Value	3C9h
DAC State	3C7h

Mode 13h:

Dies ist der 256-Farbenmodus des Standard-VGA und voll kompatibel zu MCGA. Die Auflösung von 320 x 200 wird durch interne Halbierung von 640 x 400 erreicht. Dieser Modus macht sich das Chaining der Bit-Planes zu einem linearen Adreßraum zunutze, das heißt der VGA-Adreßraum wird ab Segment A000 im konventionellen Speicher eingeblendet. Ein Punkt läßt sich in diesem Modus durch folgende Formel adressieren:

Offset = Y * 320 + X

Am entsprechenden Offset steht der Farbwert des Punktes aus der Palette. Diese bietet Platz für 256 Farben und enthält für jede den entsprechenden Rot-, Grün- und Blauwert. Da an der Punktadresse nur ein Pointer auf den Paletteneintrag steht, spart dieses Modell bei 256-

Farbenmodi sehr viel Platz. VGA-intern wird die lineare Adressierung dann wieder in eine planare umgewandelt. Dazu werden Bit 0 und 1 des Offsets zur Selektion der Schreib- bzw. Lese-Plane verwendet, die restlichen sechs Bit (2 - 7) werden als physikalische Adresse innerhalb der Plane verwendet.
Ein ähnliches Verfahren (Odd/Even-Adressierung) verwenden übrigens auch sämtliche Textmodi. Dabei dient Bit 0 zur Selektion zwischen Plane 0 und 1, so daß sich aus Sicht der CPU Zeichen- und Attribut-Byte jeweils direkt hintereinander befinden, intern jedoch Zeichen in Plane 0 und Attribute in Plane 1 abgelegt werden, Plane 2 und 3 dienen als Zeichensatzspeicher.[5]

Mode X:

Mitunter ergibt sich die Notwendigkeit von mehreren Bildschirmseiten. Diese können einfach hintereinander im Bildschirmspeicher abgelegt werden und über Register 0Ch und 0Dh (Linear Starting Address) gewählt werden. Das Problem besteht beim Mode 13h jedoch darin, daß das ganze Bild bereits fast 64 KB (genau 64.000 Bytes) groß ist. Eine Bildschirmseite belegt also schon den gesamten im Hauptspeicher eingeblendeten Videospeicher (0A0000h - 0AFFFFh), was es der CPU unmöglich macht, die zweite Bildschirmseite anzusprechen. Modifikationen sind also nur über die Segmentselektoren des VGA möglich, die aber bei jedem Hersteller anders programmiert werden. Aus diesem Grund wurde der Mode X erdacht. Ursprünglich aus dem Hacker-Untergrund stammend, ist dieser Modus heute durchaus offiziell.
Im Mode X werden bei einem Byte-Zugriff vier Pixel auf einmal kopiert, außerdem werden Read-Mode 0 und Write-Mode 1 verwendet, die weder aufwendige interne Adreßumwandlungen erfordern noch Daten an die CPU schicken, was den Geschwindigkeitsvorteil gegenüber 32-Bit-Zugriffen der CPU ausmacht.
Zur Initialisierung wird der Chain-4-Mechanismus abgeschaltet, so daß wieder freier Zugriff auf einzelne Planes möglich ist, außerdem muß sichergestellt werden, daß der Odd/Even-Mode (Plane-Selektion durch unterstes Offset-Bit) ausgeschaltet ist, dazu muß nur im TS-Register 4 (Memory Mode) Bit 3 (Enable Chain4) gelöscht und Bit 2 (Odd/Even-Mode) gesetzt werden. Je nach Grafikkarte muß noch der Speicherzugriff auf Byte-Adressierung

[5] PC underground, S. 60-61

geschaltet werden, also zunächst Doubleword-Adressierung aus Bit 6 in CRTC-Register 14h (Underline Row Address) löschen und Bit 6 in CRTC-Register 17h (CRTC-Mode) setzen. Sinnvollerweise wird jetztnoch der Bildschirmspeicher gelöscht, weil an den vom Mode 13h unbenutzten Stellen, die jetzt sichtbar werden, noch Byte-Müll aus anderen Videomodi stehen kann. Am einfachsten erfolgt das über das Write Plane Mask Register 2 des Timing-Sequenzers, in dem alle Planes eingeschaltet werden, so daß 32.000 Word-Zugriffe oder 16.000 DWord-Zugriffe ausreichen, um alle vier Bildschirmseiten zu löschen. Ab nun muß man sich allerdings um alles selbst kümmern, da weder vom BIOS noch einer Hochsprache irgendwelche Unterstützung zu erwarten ist.

In sämtlichen Plane-basierten Grafikmodi verbergen sich hinter einer Speicheradresse gleich 4 Byte, jeweils eins pro Plane. Die 4 Bytes liegen quasi übereinander an einer Adresse. Sie lassen sich einzeln ansprechen, werden aber parallel verwendet. Die Adressierung erfolgt allerdings gänzlich anders als in den 16-Farbenmodi. Die Punktnummer läßt sich im Mode X errechnen wie im Mode 13h, Plane und Offset lassen sich nach folgenden Formeln berechnen:

Plane = X mod 4

Offset = Y * 80 + X div 4

Dieses Verfahren entspricht dem des Mode 13h, mit dem Unterschied, daß der Offset durch Shiften der Punktnummer um 2 Bit nach rechts statt durch Maskieren berechnet wird, so daß im Speicher keine Lücken zwischen den Punkten entstehen und somit vier Seiten in die 256 KB Video-RAM passen. Bei der Selektion der Plane im Mode X muß allerdings noch zwischen Schreib- und Lesezugriffen unterschieden werden: Beim Lesen wird die Plane-Nummer in Register 4 (Read Plane Select) des GDC geschrieben, beim Schreiben dagegen ist es möglich, mehrere Planes gleichzeitig anzusprechen. Daher wird eine Maske in Register 2 (Write Plane Mask) des TS gesetzt, die erst aus der Plane-Nummer erzeugt werden muß, nach folgender Formel:

Maske = 1 shl Plane-Nummer

Die byteweise Adressierung seitens der CPU ist notwendig aufgrund der vier Latches des VGA.[6]

Voxel-Spacing:

[6] PC underground, S. 115-119

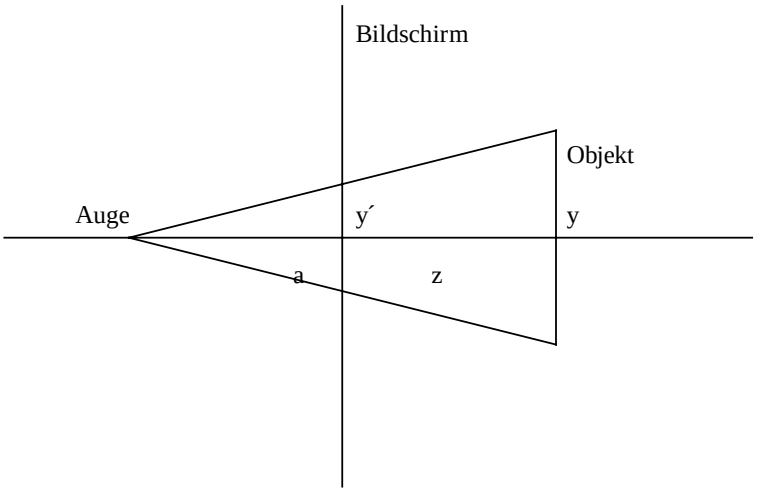
Für dieses Verfahren zur Landschaftsdarstellung existieren viele verschiedene Algorithmen. Prinzipiell wird einfach eine Landkarte in die Ebene gekippt, so daß der Betrachter von schräg oben darauf schaut. Die Höheninformation kann nun durch verschieden lange vertikale Linien dargestellt werden. Als Landkarte, auf der die Höhen und Tiefen der Landschaft verzeichnet sind, dient ein Bild mit möglichst fließenden Übergängen. Dabei entspricht Farbe 0 dem tiefsten Punkt und Farbe 255 dem höchsten. Die Projektion erfolgt in der Praxis zweidimensional nach einem relativ einfachen Prinzip: Bekanntlich erscheinen weiter entfernte Gegenstände kleiner als nähere. Wenn man nun durch einen Rahmen (und nichts anderes ist der Bildschirm) in eine Landschaft schaut, stellt man fest, daß in der Ferne mehr Gegenstände in diesen Rahmen passen als in der Nähe, weil sie eben kleiner sind. Man sieht also immer einen Ausschnitt. Der rechteckige Rahmen stellt die gesamte existierende Landschaft aus der Vogelperspektive dar und das Trapez den sichtbaren Teil derselben. Nun muß lediglich dafür gesorgt werden, daß dieses Trapez auf einen rechteckigen Bildschirmbereich abgebildet wird, dadurch wird der vordere Teil waagrecht gestreckt, so daß Gegenstände hier vergrößert erscheinen. Die trapezförmige Projektion erreicht man einfach dadurch, daß die Anzahl der in eine Zeile passenden Pixel ständig verringert wird. Anders ausgedrückt: Das Verhältnis von Landschafts- zu Bildschirmpixelgröße wird ständig dekrementiert. Der Startwert ist eine 1 : 1 - Abbildung, während des Zeichnens wird das Verhältnis immer weiter verringert, bis etwa eine 1 : 2 - Abbildung erreicht ist. Bei der zeilenweisen Darstellung ist allerdings auch darauf zu achten, daß die Abstände zwischen den Zeilen nach hinten immer enger werden. Beim Zeichnen von oben nach unten ist also der Zeilenabstand kontinuierlich zu erhöhen. Nun fehlt nur noch die Höheninformation im Bild. Dazu wird die Farbe jedes Punktes herangezogen, indem eine entsprechende Anzahl Pixel übereinander angeordnet werden. Durch diese vertikale Struktur werden auch Lücken vermieden, die durch den nach vorne wachsenden Zeilenabstand auftreten. Zur Darstellung von Wasserflächen wird einfach jede Farbe unterhalb eines bestimmten Wertes auf eben diesen gesetzt.[7]

[7] PC underground, S. 200-206

Vektorgrafik:

Für die zweidimensionale Darstellung dreidimensionaler Objekte gibt es viele Methoden, bis hin zu komplexen Raytracing-Algorithmen. Aufgrund des hohen Rechenaufwands sind diese allerdings den Renderern vorbehalten.

Die einfachste Methode, die Tiefeninformation umzurechnen, die der Monitor ja nicht darstellen kann, ist, sie komplett zu ignorieren. Dabei werden die zweidimensionalen Koordinaten der Eckpunkte jeder Fläche aus den x- und y-Koordinaten der dreidimensionalen Definition gewonnen. Bei dieser Methode erscheinen parallele Geraden des dreidimensionalen Raums als parallele Geraden auf dem Bildschirm - daher der Name Parallelprojektion. Das hat gewisse Vorteile (und wird z. B. in der Architektur benutzt), hat aber mit der Wirklichkeit wenig zu tun; schließlich erzeugen weiter entfernte Objekte ein kleineres Bild auf der Netzhaut des Auges. Diese Eigenschaft muß nun durch die Abbildung simuliert werden, indem die Tiefe eben nicht mehr ignoriert wird, sondern zur Verkleinerung des Bildes dient. Den benötigten Algorithmus kann man sich anhand des Strahlensatzes leicht veranschaulichen:



$\frac{a}{z} = \frac{y'}{y} \Rightarrow y' = \frac{a\,y}{z}$ entspr.: $x' = \frac{a\,x}{z}$

Hier betrachtet man die dreidimensionale Welt als hinter dem Bildschirm stehend. Von dieser Welt sieht man zunächst nichts, aber die Abbildung auf dem Bildschirm ist sichtbar. Alle Strahlen, die vom Objekt ausgehen, müssen dabei letztendlich beim Auge ankommen, so daß im Bild hier der Ursprung liegt. Jede Koordinate des Objekts (hier beispielhaft für die y-Koordinate der oberen und unteren Ecke) muß in eine zweidimensionale

Bildschirmkoordinate (hier y´) übergeführt werden. Dazu bedient man sich des Strahlensatzes, der eine eindeutige Beziehung zwischen y, y´, a (Abstand Auge - Bildschirm) und z (Tiefe des Objekts) herstellt. Es ist offensichtlich, daß bei diesem Algorithmus eine größer werdende Tiefe (z) den Winkel zwischen den Strahlen verkleinert und somit auch das Bild auf dem Schirm, so daß hier die gewünschte Fluchtpunktperspektive erreicht ist. Dieser Fluchtpunkt, an dem alle parallelen Linien in der Ferne zusammenlaufen, befindet sich bei dieser Methode allerdings immer auf der z-Achse, er ist nicht beliebig plazierbar, was für die meisten Anwendungen auch gar nicht nötig ist. Bei der Berechnung anhand dieser Formel kann man viel Rechenzeit sparen, wenn man für den (ohnehin willkürlichen) Abstand a eine Zweierpotenz einsetzt. In diesem Fall läßt sich die Multiplikation durch Shiften (SHL / SHR) erheblich beschleunigen.[8]

Transformationen:

In den seltensten Fällen wird man sich bei der Programmierung von dreidimensionalen Welten mit einem unveränderlichen Bild begnügen. Gerade die Bewegung macht ein realitätsnahes Bild erst aus. Eine Bewegung setzt sich im wesentlichen aus zwei Arten zusammen: Translation und Rotation. Auch die Skalierung kann man unter Umständen noch dazuzählen.

Eine Translation ist nichts weiter als eine Verschiebung in eine bestimmte Richtung, zum Beispiel eine Bewegung durch einen langen Gang. Mathematisch gesehen baut die Translation auf einer Vektoraddition auf: Die Verschiebung wird durch einen Vektor, den Translationsvektor, bestimmt. Dieser Vektor wird einfach auf alle Ortsvektoren (Punkt-Koordinaten) des zu verschiebenden Objekts addiert. Eine Bewegung des Betrachters wird dabei ganz genauso erreicht, die Sichtweise ist jedoch umgekehrt: Möchte man sich als Betrachter um eine Einheit in z-Richtung bewegen, verschiebt man einfach die gesamte 3D-Welt um eine Einheit in negative z-Richtung.

Die Rotation ist da schon etwas komplizierter, vor allem, wenn man um jeden Preis mit Matrizen rechnen will, wie das oft propagiert wird. Matrizen fassen in einer Art Tabelle die notwendigen Rechenschritte für eine Transformation zusammen; Translationen, Rotationen, Skalierungen, alles hat seine Matrize. Das hat den Vorteil, daß man mehrere Matrizen

[8] PC underground, S. 233-235

zusammenfassen und damit etwas Rechenzeit sparen kann - wenn von vornherein feststeht, welche Transformationen in welcher Reihenfolge durchgeführt werden sollen. Weil das jedoch selten der Fall ist, werden die Rotationsmatrizen hier nur der Vollständigkeit halber erwähnt. Bei der Rotation muß grundsätzlich unterschieden werden, um welche der drei Achsen rotiert werden soll, es werden jeweils andere Verknüpfungen durchgeführt.

Um die x-Achse:	Die entsprechende Matrix:
x´ = x	1 0 0
y´ = y * cos (a) - z * sin (a)	0 cos (a) - sin (a)
z´ = y * sin (a) + z * cos (a)	0 sin (a) cos (a)
Um die y-Achse:	Die entsprechende Matrix:
x´ = x * cos (a) + z * sin (a)	cos (a) 0 sin (a)
y´ = y	0 1 0
z´ = - x * sin (a) + z * cos (a)	- sin (a) 0 cos (a)
Um die z-Achse:	Die entsprechende Matrix:
x´ = x * cos (a) - y * sin (a)	cos (a) - sin (a) 0
y´ = x * sin (a) + y * cos (a)	sin (a) cos (a) 0
z´ = z	0 0 1

Diese Formeln betrachten zunächst einmal nur die Rotation um die Koordinatenachsen. Durch eine Kombination mehrerer Rotationen kann aber jede beliebige durch den Ursprung verlaufende Gerade eine Achse bilden. Möchte man auch noch die Einschränkung der Ursprungsgeraden umgehen, muß man die Rotation noch mit einer Translation verbinden. Dazu wird vor der Rotation die Welt so weit verschoben, daß der Punkt, um den gedreht werden soll, sich im Ursprung befindet. Nach der Rotation wird dann gegebenenfalls wieder zurückverschoben, wobei allerdings auch der Translationsvektor vorher rotiert werden muß. Bei der aufeinanderfolgenden Rotation um mehrere Achsen müssen selbstverständlich jeweils die errechneten Koordinaten der vorherigen Rotation als Quellkoordinaten in die folgende eingesetzt werden. Verwendet man immer wieder die eigentlichen Weltkoordinaten als Quelle, dürfte man recht ungewöhnliche Ergebnisse erhalten, die mit der eigentlich darzustellenden Welt nicht viel gemeinsam haben. Gerade die Rotation mit ihren vielen Sinus- und Cosinus-Berechnungen stellt einen besonders geeigneten Anwendungsfall der Tabellen-Rechnung dar. Wollte man all diese Berechnungen mit gewöhnlichen Pascal-

Funktionen programmieren, käme nie eine flüssige Bewegung zustande. Also werden alle Sinus- und Cosinus-Werte aus der (gleichen) Tabelle entnommen und mit den Koordinaten entsprechend der jeweiligen Rechenvorschrift multipliziert.

Außer den reinen Translationen sind die genannten Transformationen grundsätzlich nicht kommutativ, d. h. ihre Reihenfolge ist nicht gleichgültig. Rotiert man beispielsweise den auf der x-Achse liegenden Punkt (1/0/0) zunächst um 90 Grad um die x-Achse, dann um den gleichen Winkel um die z-Achse, liegt das Ergebnis auf der y-Achse. Bei umgekehrter Reihenfolge liegt es auf der z-Achse. Daher sollte man eine einheitliche Reihenfolge festlegen, nach der rotiert wird, zweckmäßigerweise erst um die x-, dann y- und schließlich z-Achse. Auch bei gemischten Transformationen muß die Reihenfolge beachtet werden. Bei einer Translation mit folgender Rotation kommt sicher ein anderer Punkt heraus als bei umgekehrter Reihenfolge, daher sollte man sich auch hier festlegen. Am sinnvollsten ist in diesem Fall die Folge Translation - Rotation, weil sich dann die Translations-Werte auf die bekannten Weltkoordinaten bezeiehen und nicht auf deren rotierte Abbildungen. Zusammenfassend kann man also folgende Reihenfolge als geeignetste ansehen:[9]

1. Translation
2. Rotation (x, y, dann z)
3. Projektion auf den Bildschirm

Drahtmodelle:

Die einfachste Möglichkeit, dreidimensionale Objekte auf den Bildschirm zu bekommen, stellen die Drahtmodelle dar. Hier werden nur die Kanten des jeweiligen Körpers gezeichnet, nicht seine Flächen. Dadurch erscheint der Körper durchsichtig, und mann muß sich nicht um eine eventuelle Unterdrückung verdeckter Flächen kümmern. Dieses Modell macht sich zunutze, daß sowohl bei der Parallelprojektion als auch bei der beschriebenen Fluchtpunktperspektive dreidimensionale Geraden als Geraden am Bildschirm erscheinen und nicht etwa als Kurven. Dank dieser Tatsache kann man sich nämlich auf die Transformation der Eckpunkte beschränken und muß nicht jeden Punkt der Kante verschieben, rotieren und abbilden. Verbindet man nun diese berechneten Eckpunkte, erhält man ein realistisches Bild des Körpers - als Drahtmodell. Der wichtigste Teil dieses Modells ist zweifellos der

[9] PC underground, S. 235-237

Linienalgorithmus. Von diesem hängt ein Großteil der späteren Darstellungsgeschwindigkeit ab, denn die Transformationen an sich benötigen kaum Rechenzeit. Eins der zur Zeit schnellsten Verfahren zum Zeichnen einer Linie ist der Bresenham-Algorithmus. Der Algorithmus beschränkt sich zunächst auf Steigungen zwischen 0 und 1 (0 bis 45 Grad). Während des Zeichnens der Linie muß jetzt nur noch für jeden Punkt entschieden werden, ob er sich genau rechts neben dem vorherigen oder rechts über diesem befindet, andere Punkte sind nicht möglich. Eine Variable (gespeichert in BP) wird abhängig vom letzten Schritt (rechts oder rechts oben) entweder um SI oder um DI erhöht und beim nächsten Punkt dann die Entscheidung davon abhängig gemacht, ob BP positiv oder negativ ist. Die Beschränkung auf Steigungen zwischen 0 und 1 läßt sich nun sehr einfach aufheben: Steigungen zwischen 1 und unendlich (45 bis 90 Grad) werden durch Vertauschen von x und y erreicht und negative Steigungen durch Umkehren der Bearbeitungsrichtung.[10]

Glaskörper:

Die bisherigen Objekte bestehen praktisch nur aus Kanten, sie haben mit realen Objekten daher sehr wenig zu tun. Der nächste Schritt muß also sein, den Körpern Seitenflächen und damit Masse zu verleihen. Dabei stößt man jedoch sehr bald auf eines der größten Probleme aller berechneten 3D-Welten: die verdeckten Flächen. Zeichnet man einfach alle Flächen hintereinander weg, wie sie definiert sind, erscheinen oft Flächen, die normalerweise überhaupt nicht sichtbar wären. Um die Unterdrückung dieses Flächen wird sich jedoch erst der folgende Abschnitt kümmern, hier soll zunächst ein anderer Weg beschritten werden. Statt die Abbildungen der Wirklichkeit anzupassen, soll erst einmal die Wirklichkeit der Abbildung angepaßt werden: Bei einem Glaskörper sind alle Seitenflächen immer sichtbar. Trotzdem können die Seiten natürlich auch eine Farbe haben, müssen sie sogar, um überhaupt ein Bild zu erzeugen. Liegen nun zwei Flächen hintereinander, so überlagern sich deren Farben, und insgesamt kommt eine etwas dunklere (zwei Flächen filtern mehr Licht aus als eine) Mischfarbe dabei heraus. Dabei muß prinzipiell jede mögliche Kombination von Flächen berücksichtigt werden, d. h. wenn Fläche A Fläche B unter irgendeinem Winkel überlagern kann, muß eine Mischfarbe dieser beiden Flächen existieren. Die einzige Möglichkeit, dies zu realisieren, besteht darin, für jede Fläche ein Bit in der Farbinformation zu reservieren. Dabei

[10] PC underground, S. 237-257

dürfen nur Flächen, die sich unter keinen Umständen überlagern können, das gleiche Bit benutzen, weil ansonsten eine Mischung nicht möglich ist. Wird nun bei der Darstellung der Fläche nicht die alte Farbe überschrieben, sondern beide Werte OR-verknüpft, so entsteht ein neuer Farbwert. Natürlich muß auch die Palette dieser besonderen Struktur folgen. Zum einen muß sie die reinen Farben enthalten, zum anderen jede Bit-Kombination mit einer Mischfarbe aus den entsprechenden Bits versehen. Die Palette muß nun zunächst beim Programmstart entsprechend vorbereitet werden, beim Füllen der Polygone macht man sich die eingebaute arithmetische Einheit des VGA zunutze. Dieser kann nämlich über GDC-Register 3 in den OR-Modus geschaltet werden, der die ankommenden CPU-Daten mit den in den Latches liegenden OR-Werten verknüpft, bevor sie in den Bildschirmspeicher geschrieben werden. Nun muß man lediglich vor dem Schreibzugriff die Latches mit den bereits im Bildschirmspeicher stehenden Werten laden, indem man einen Lesezugriff auf die gleiche Speicherstelle ausführt.

Bei den Füllalgorithmen gibt es zwei grundsätzliche Kategorien: zum einen die allgemeine Füllung, die beliebige, vorgezeichnete Flächen füllt und häufig in Malprogrammen zum Einsatz kommt, zum anderen aber die Füllung eines durch Koordinaten definierten Polygons. Letzterer Algorithmus ist für diese Zwecke eindeutig der bei weitem schnellere. Im wesentlichen baut die hier beschriebene Methode auf dem Zeichnen von Linien auf. Dazu wird, ausgehend vom Punkt mit der niedrigsten y-Koordinate, der linke und rechte Rand des Polygons abgetastet, bis der Punkt mit der größten y-Koordinate ereeicht ist. Dabei werden die Begrenzungslinien des Polygons nur berechnet und nicht gezeichnet. Ist man auf diese Weise auf beiden Seiten um eine Zeile weitergekommen, kann eine horizontale Linie zwischen dem linken und rechten errechneten Punkt gezogen werden. Dabei macht man sich zunutze, daß gerade im Mode X horizontale Linien mit sehr hoher Geschwindigkeit gezeichnet werden können. Eine Füllroutine muß also sowohl am linken als auch am rechten Polygonrand ständig Linien berechnen. Ist eine Linie fertig “gezeichnet”, wird die nächste, deren Startpunkt ja dem letzten Eckpunkt entspricht, begonnen.[11]

Hidden Lines:

[11] PC underground, S. 257-273

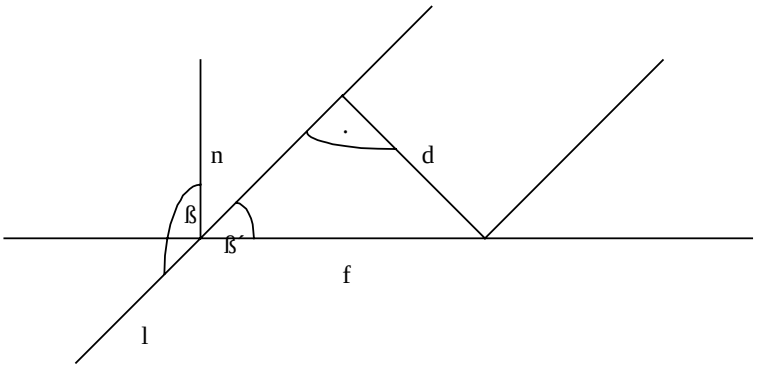
Glaskörper mögen ihren eigenen Reiz haben, aber für eine Darstellung realer Körper sind sie selten geeignet. Die meisten Körper sind nun einmal undurchsichtig, so daß man dafür sorgen muß, daß auch eine Computer-Abbildung derselben keine eigentlich unsichtbaren Rückenflächen zeigt. Das Problem der Flächenrücken-Unterdrückung ist eines der komplexesten Themen der dreidimensionalen Darstellung. Es geht ja nicht nur darum, bestimmte Flächen zu zeichnen und andere nicht. Teilweise überlagern sich Flächen, so daß beide gezeichnet werden müssen - aber in der richtigen Reihenfolge. Beide Methoden - Unterdrückung und Sortierung - werden hier vorgestellt. Zur Unterdrückung unsichtbarer Flächen gibt es eine Unzahl verschiedener Ansätze, die meist einen Winkel zwischen der Fläche und der Blickgeraden (Gerade vom Auge zur betreffenden Fläche) bilden. Anhand dieses Winkels läßt sich dann zeigen, ob der Betrachter auf die Vorder- oder die Rückseite der Fläche blickt. Letzterer Fall bedeutet, daß die Fläche unterdrückt werden muß. Die hier vorgestellte Methode geht von einem ähnlichen Ansatz aus, nimmt jedoch eine gewaltige Vereinfachung vor: Es wird davon ausgegangen, daß alle Flächen im Gegen-Uhrzeigersinn (mathematisch positiv) definiert sind. Dadurch wird die Fläche unabhängig von ihrer Lage im Raum auch immer “linksherum” gezeichnet. Hat sie sich jedoch so weit gedreht, daß der Betrachter auf ihre Rückseite blickt, erscheint sie spiegelverkehrt und wird “rechtsherum” gezeichnet. Nun wird einfach beim Zeichnen der horizontalen Linien geprüft, ob deren Endpunkt - der Definition des Flächenalgorithmus entsprechend - rechts vom Startpunkt liegt. Ist dies nicht der Fall, blickt man auf eine Rückseite, die unterdrückt werden muß. Diese Methode ist bereits vollkommen ausreichend für konvexe Körper, also Körper ohne “Vertiefungen”, z. B. Würfel. Was passiert aber bei konkaven Körpern wie zum Beispiel einem U-förmigen Objekt? Unsichtbare Flächen werden aussortiert, aber die Reihenfolge ist noch nicht korrekt, so daß teilweise Flächen voll sichtbar sind, die eigentlich von anderen verdeckt werden. Sortiert man nun die Flächen so, daß zuerst die Fläche mit der größten z-Koordinate, also die am weitesten hinten liegende, gezeichnet und dann immer weiter nach vorne gearbeitet wird, so verdecken die neuen, vorderen Flächen Teile der bereits gezeichneten Welt. Da diese Methode auch in der Malerei angewandt wird, spricht man dabei auch von “Painter´s Algorithm”. Flächen sortieren, gut und schön, aber wie? Die Ecken einer Fläche haben in den meisten Fällen völlig unterschiedliche z-Koordinaten. Komplexe und daher langsame Algorithmen versuchen, dem Rechnung zu tragen und Beziehungen zwischen

den Ecken zu finden, die eine eindeutige Zuordnung ermöglichen. Weitaus schneller ist dagegen die Sortierung nach mittleren Tiefen. Dazu wird der Mittelwert der Tiefeninformationen jeder Fläche gebildet und als Sortierkriterium verwandt. Diese Methode ist sehr ungenau, vor allem bei Flächen mit großer Ausdehnung in z-Richtung, erzielt jedoch in Verbindung mit einer Flächenrücken-Unterdrückung bereits sehr ansprechende und vor allem schnelle Ergebnisse.[12]

Lichtquellen-Schattierung:

Mittlerweile verfügen wir über feste, undurchsichtige Körper, die sich beliebig im Raum drehen können. Ein wichtiger Aspekt blieb jedoch bisher unbeachtet: die Beleuchtung. Durch Einbringen einer Lichtquelle lassen sich dreidimensionale Welten noch eindrucksvoller darstellen, als das mit den bisherigen Routinen möglich ist. Derzeit ist noch kein Echtzeit-Raytracing möglich (oder nur mit spezieller Hardware), derartige Bilder benötigen immer noch Minuten bis zur vollständigen Berechnung. Also braucht man eine schnellere Methode, die zwar nicht jeden Lichtstrahl verfolgt, aber mit gewissen Vereinfachungen bereits sehr schöne Effekte erzeugt. Geht man von einer unendlich weit entfernten Lichtquelle aus, so gelangen alle Lichtstrahlen parallel auf die Flächen der Objekte, daher kann man mit einem einzigen Lichtvektor rechnen, statt für jeden Punkt der Flächen einen eigenen auszurechnen. Durch diese homogene Beleuchtung reicht es aus, für jede Fläche eine Helligkeit zu berechnen, in der sie dann komplett eingefärbt wird. Wie berechnet man aber die Helligkeit dieser Fläche? Dazu bedient man sich eines einfachen Modells: Je flacher das Licht auf die Fläche trifft, desto dunkler wird diese, bei senkrechter Beleuchtung ist die Helligkeit dagegen maximal. Dies kommt daher, daß eine gleich große Energiemenge bei flacherem Winkel über eine größere Fläche verteilt wird und daher nicht mehr so dicht ist:

[12] PC underground, S. 273-276



$\frac{d}{f} = \sin ß´$ ß´ = ß - 90 $\frac{d}{f} = \sin (ß - 90) = - \cos ß$

Das Verhältnis d / f ist hier proportional zur Helligkeit der Fläche. Wie die Herleitung zeigt, ist dieses Verhältnis gleich dem negativen Cosinus des Winkels zwischen Lichtvektor und Normalvektor der Fläche. Der Normalvektor ist ein Vektor, der senkrecht auf der Fläche steht. Er läßt sich leicht durch ein Kreuzprodukt zweier in der Ebene liegender Vektoren bestimmen. Daß hier der Cosinus des Winkels benötigt wird und nicht der Winkel selbst, vereinfacht die Berechnung enorm, da als Ergebnis einer Winkelbestimmung (durch das Skalarprodukt) der Cosinus des Winkels herauskommt. Die Vorgehensweise zur Bestimmung der Helligkeit ist also wie folgt:

- Zwei Vektoren finden, die auf der Fläche liegen; am einfachsten zwei Randvektoren (vom ersten zum zweiten und zum letzten Punkt)
- Normalvektor bilden (Kreuzprodukt der Flächenvektoren)
- Durch Skalarprodukt Winkel zwischen (konstantem) Lichtvektor und Normalvektor bilden
- Ergebnis auf Farbe addieren

Das Ergebnis der Winkelberechnung ist im gezeichneten Fall negativ. Ist die Fläche aber dem Licht abgewandt (ß < 90 Grad), so ist das Ergebnis positiv, und es darf nur die Grundfarbe der Fläche benutzt werden, weil sie im Schatten liegt und daher nur Streulicht abbekommt.[13]

Texturen:

[13] PC underground, S. 277-283

Der letzte und zugleich größte Schritt, den unsere 3D-Routinen erfahren sollen, ist die Einbindung von Texturen, die den bisher glatten und einfarbigen Flächen eine Struktur verleihen sollen. Dazu werden Bitmap-Grafiken auf die Flächen der Objekte projiziert und bei jeder Rotation mitbewegt. Dadurch werden die Bitmaps praktisch auf die Flächen aufgeklebt und können eine bestimmte Oberfläche wie zum Beispiel Holz oder Metall simulieren. Überlegt man sich konkret ein Konzept zur Programmierung, wird man in den meisten Fällen zunächst die einfachste Möglichkeit ins Auge fassen: Man setze eine Fläche aus vielen kleinen Flächen zusammen, die jeweils mit einem Punkt der Bitmap korrespondieren. Diese Technik ist allerdings nicht die schnellste, abgesehen davon, daß sie in den wenigsten Fällen überhaupt ordnungsgemäß funktioniert. Wird eine solche Fläche etwas vergrößert oder gedreht, tauchen sofort Lücken auf, weil sich einige Punkte überlagern, die dann direkt nebenan fehlen. Die einzige praktikable Lösung dieses Problems besteht in der Umkehrung des Verfahrens: Bei der Darstellung der Fläche wird wie bekannt vorgegangen und somit jeder Punkt gesetzt. Dieser Punkt wird jedoch jedesmal auf die ursprüngliche Fläche zurückprojiziert, um seine Lage innerhalb dieser Fläche zu bestimmen. Anhand dieser Lage kann die Farbe des Punktes aus der Textur-Bitmap ausgelesen werden. Da es jedoch unmöglich ist, aus den zweidimensionalen Bildschirmkoordinaten auf die dreidimensionale Lage des Punktes zu schließen, werden die 3D-Koordinaten beim Füllen von vornherein immer mitgezählt, so daß man zu jedem Punkt seine Koordinaten kennt und somit seine Lage innerhalb der Fläche.

Gleich zu Beginn der Prozedur wird die Hauptdeterminante gebildet. Mit deren Hilfe wird später die relative Koordinate des Punktes innerhalb der Fläche bestimmt. Dazu ist es wichtig zu wissen, daß jeder Punkt auf einer Fläche eine Lösung des folgenden Gleichungssystems ist:

x1 = lambda1 * a1 + lambda2 * b1

x2 = lambda1 * a2 + lambda2 * b2

x3 = lambda1 * a3 + lambda3 * b3

Dabei sind x1 - x3 die Koordinaten des Punktes, a1 - a3 die Komponenten des ersten Flächenvektors und b1 - b3 die des zweiten. lambda1 und 2 geben die affinen Koordinaten relativ zu den beiden Flächenvektoren an und können direkt zum Zugriff auf die Textur benutzt werden. Um lambda1 und lambda2 auszurechnen, benötigt man lediglich zwei der Gleichungen, die dritte ist dann in jedem Fall erfüllt, weil der Punkt ja in der Ebene liegt.

Nimmt man zum Beispiel die ersten beiden Gleichungen, so kann man die Lösung sehr einfach durch Determinanten finden: Die Hauptdeterminante beträgt D = a1 * b2 - a2 * b1, die erste Nebendeterminante D1 = x1 * b2 - x2 * b1 und die zweite Nebendeterminante D2 = a1 * x2 - a2 * x1. Die beiden Unbekannten ergeben sich nun als lambda1 = D1 / D und lambda2 = D2 / D. Die Hauptdeterminante ist jetzt für die ganze Fläche gleich, so daß sie hier direkt ausgerechnet werden kann. Dabei tritt jedoch noch ein Problem auf: Unter Umständen kann die Wahl der ersten beiden Gleichungen ungünstig sein, was sich dadurch zeigt, daß die Hauptdeterminante dann 0 beträgt. In diesem Fall muß einfach eine andere Kombination benutzt werden.[14]

[14] PC underground, S. 283-294

Inhaltsverzeichnis:

Bibliographie:

PC intern 4, Michael Tischer, Data Becker, ISBN 3-8158-1094-9

PC underground, Boris Bertelsons, Mathias Rasch, Data Becker, ISBN 3-8158-1117-1

BYTE´s DOS Programmer´s Cookbook, Craig Menefee, Lenny Bailes, Nick Anis,
Osborne McGraw-Hill, ISBN 0-07-882048-0