GRAFIK

[Inf-Referat, 27. 10. 1995, Stiedl Thomas & Gehri Thomas]

Historische Entwicklung der Grafik-Hardware:

MDA (Monochrome Display Adapter):
Diese Karte wurde 1981 zusammen mit dem ersten IBM-PC von IBM vorgestellt und war bis zum Erscheinen der Hercules-Karte Monochrom-Standard. Sie verfügt nur über einen Betriebsmodus zu 80 (Spalten) x 25 (Zeilen) Text und sehr wenig Video-RAM, so daß nur eine Bildschirmseite im Speicher Platz hat. Der MDA unterstützt keine Grafiken, weist aber eine deutlich höhere Bildschirmauflösung als der CGA auf.

CGA (Color Graphics Adapter):
Auch diese Karte kam 1981 auf den Markt und bot im Gegensatz zum MDA bereits Grafikdarstellung. Sie konnte über einen speziellen Ausgang an einen normalen Fernseher angeschlossen werden[1], oder über einen RGB-Ausgang an einen eigenen Monitor. Die Auflösung ist deutlich schlechter als beim MDA. Im Textmodus stellt die CGA-Karte, genau wie die MDA-Karte, 80 x 25 dar, wobei die einzelnen Zeichen aber auf einer kleineren Punktmatrix basieren. An Grafikmodi stehen 320 x 200 (Punkte) x 4 (Farben) sowie 640 x 200 x 2 zur Verfügung. CGA und MDA basierten beide auf dem MC6845 von Motorola.

Hercules Graphics Card:
Ein Jahr später kam die Hercules-Karte auf den Markt, die weitestgehend zum MDA kompatibel war. Darüber hinaus konnte sie jedoch zwei Grafikseiten mit einer Auflösung von 720 x 348 (monochrom) darstellen. Diese Karte war sehr einfach zu programmieren und bot auch eine parallele Schnittstelle. Sie stellt heute den Monochrom-Standard dar.

EGA (Enhanced Graphics Adapter):
Der EGA wurde 1985 vorgestellt und war eine späte Antwort auf die Hercules-Karte. Er ist voll kompatibel zu MDA und CGA und kann neben farbigen auch monochrome Grafiken darstellen. Die EGA-Karte war damit die erste, die mit einem Monochrom- oder einem Colormonitor eingesetzt werden konnte. Im Grafikmodus stellt sie bei 640 x 350 Punkten 16 Farben aus einer Palette von 64 gleichzeitig dar. 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[2].

VGA (Video Graphics Array):
Die VGA-Karte wurde 1987 vorgestellt und ist zu allen Vorgängern kompatibel[3]. Sie stellt heute den Standard der PC-Grafikkarten dar und ist dabei der kleinste gemeinsame Nenner, zu dem alle SVGA-Karten (siehe dort) kompatibel sind. Der VGA sendet erstmals analoge Signale an den Monitor, was die große Farbenvielfalt ermöglicht. Die höchste Auflösung ist 640 x 480, wobei maximal 16 Farben dargestellt werden können. Im Mode 13h können außerdem bei 320 x 200 Bildpunkten 256 Farben gleichzeitig aus einer Palette von 262.144 dargestellt werden. Dieser Modus diente dabei ursprünglich der Kompatibilität zu MCGA, wobei der VGA diese Auflösung eigentlich gar nicht darstellen kann. Durch einen Trick wird aber durch Setzen von zwei VGA-Registern die tatsächliche Auflösung von 640 x 400 einfach halbiert. Dem Profi-Programmierer bietet sich

---

[1] Dies war möglich, da die Horizontalfrequenz dieser Karte (und auch die Bildwiederholfrequenz) noch sehr gering war und sogar noch unter PAL-Standard lag. Bei heutigen SVGA-Karten sind beide um ein Vielfaches höher, genauso wie die Auflösung, so daß ein Anschluß nur über einen speziellen Adapter möglich ist, was allerdings die Bildqualität enorm verschlechtert.

[2] Hierzu ist zu bemerken, daß Grafikausgabe über die ROM-Routinen zwar sehr einfach und bequem, aber auch relativ langsam ist. Für professionelle Programmierung, insbesondere bei zeitkritischen Routinen, ist deshalb eine direkte Programmierung der Karte, zum Beispiel direkter Speicherzugriff, unerläßlich.

[3] Bei direkter Low-Level-Programmierung der Karte treten allerdings teilweise Probleme auf, so ist es zum Beispiel beim Grabben einer Grafik im Herculesmodus unmöglich, festzustellen, welche der beiden emulierten Hercules-Grafikseiten gerade aktiv ist.

dadurch die Möglichkeit, durch Löschen der beiden Register einen sehr einfach zu programmierenden hochauflösenden Modus zu erhalten, der von jeder VGA-Karte mit mindestens 512 KB Speicher dargestellt werden kann. Ursprünglich wurden VGA-Karten mit 256 KB Video-RAM bestückt, heute ist aber 1 MB bereits das Minimum, da jede Karte gleichzeitig auch eine SVGA-Karte ist, mit mehr oder weniger erweiterten Möglichkeiten.

SVGA (Super VGA):

SVGA-Karten sind VGA-Karten mit erweiterten Möglichkeiten. Sie sind mit mindestens 1 MB, meistens schon 2 MB, häufig 4 MB und inzwischen bis zu 8 MB Video-RAM bestückt. Der maximale Grafikmodus ist derzeit 1600 x 1200 Bildpunkte bei 16.7 Mio. Farben gleichzeitig (!), inzwischen bieten manche Karten sogar 32 Bit Farbtiefe, was die Brillianz des Bildes noch erhöhen soll. Bildwiederholraten von bis zu 150 oder gar 200 Hertz sind bei den Karten der Luxusklasse dabei schon keine Seltenheit mehr, erfordern aber auch einen entsprechend teuren und leistungsfähigen Monitor. Als weitere Zuckerln bieten viele Karten heute auch Zusatzfunktionen:

- Windows-Beschleuniger sollen dem Hauptprozessor unter Windows Arbeit abnehmen und sind speziell für die Zusammenarbeit mit dem GDI ausgelegt.
- AVI-Beschleuniger stellen AVIs unter Windows dar, ohne den Hauptprozessor zu belasten. Sie erlauben auch freies Skalieren und Zoomen und können das Video als hardwaremäßiges Overlay über das Bild legen.
- MPEG-Decoder spielen MPEG-Videos in Echtzeit ab, ohne den Prozessor zu belasten.

Der Nachteil des ganzen Zaubers ist allerdings, daß nichts (!) davon genormt ist und deswegen von Hersteller zu Hersteller unterschiedlich zu programmieren. Das soll sich zwar mit Windows ändern (siehe unten), für die Zeit bis dahin wurde allerdings 1989 das VESA-Komitee gegründet.

VESA (Video Electronics Standard Association):

Um das Chaos bei der Ansteuerung von SVGA-Karten zu beenden, haben sich 1989 die wichtigsten Anbieter an einen Tisch gesetzt und gemeinsam den VESA-Standard entwickelt. Zu diesen Firmen gehören unter anderen: ATI, Chips & Technologies, Everex, Genoa, Intel, Phoenix Technologies, Orchid, Paradise, Tseng, Video Seven und noch eine Reihe anderer, um nur die wichtigsten zu nennen. 1990 stellten sie den VESA-Standard vor, eine Video-BIOS-Erweiterung, die die Kartenansteuerung vereinheitlicht. Bei neueren Karten ist sie schon im ROM-BIOS eingebaut, für ältere Karten ist sie durch TSRs realisierbar. Die aktuelle Version ist 1.2. Dieser Standard ist ziemlich schnell und bietet sehr tiefgreifende Möglichkeiten, darüber hinaus ist er noch sehr einfach zu programmieren und stellt derzeit wohl die beste Möglichkeit dar, mit SVGA unter DOS zu arbeiten.

Windows:

Windows vereinfacht die Ansteuerung der Grafikkarte enorm, da es eine einheitliche Plattform für alle Programme bietet. Mit der zunehmenden Verbreitung von **Windows 95** als Betriebssystem sollte deshalb das Chaos der Programmierung bald der Vergangenheit angehören (Hörten wir das nicht von der Speicherverwaltung auch schon ?). In Zukunft liefert dann jeder Hersteller nur noch einen optimierten Windows-Treiber für seine Grafikkarte, der alle ihre Möglichkeiten bestmöglich ausnützt. Windows greift dann über diesen Treiber auf die Grafikhardware zu und stellt den Programmen eine einheitliche Schnittstelle zur Verfügung. Im konkreten sieht das so aus, daß der Programmierer seine Grafikdaten einfach ins **GDI**[4] schreibt, und Windows sorgt dann für die korrekte Darstellung. Allerdings hat das GDI nur eine beschränkte Größe, auch wenn es bei Windows 95 erweitert wurde, und Spezialfunktionen (wie z. B. das Abspielen von MPEG-Videos) bedürfen immer noch eines eigenen Treibers, der aber vom Hersteller mitgeliefert wird. Sollte hier aber nicht bald ein Machtwort gesprochen werden, könnte auf diesem Gebiet bald ein ähnliches Ansteuerungschaos drohen wie bei SVGA unter DOS. Ein klarer Vorteil des GDI ist aber sicherlich die extrem einfache Programmierung beliebiger Grafikmodi (die übrigens in C-Konvention definiert wurde). Die geringe Geschwindigkeit dieser Art der Grafikdarstellung ist zwar für Standardanwendungen (Textverarbeitung, ...) durchaus ausreichend, bei Spielen oder ähnlich grafikintensiven Anwendungen wird es aber problematisch. Zur Behebung dieser Schwierigkeiten wurde WIN-G definiert, das direkte Grafikprogrammierung unter Windows erlaubt, *ohne* irgendwelche Standard-Probleme! Zusammenfassend ist zu sagen, daß Windows 95 möglicherweise die Lösung aller Programmierprobleme darstellen könnte (Wer´s glaubt, hat Murphy´s Gesetze noch nicht gelesen ...) und man erstmals ohne Geschwindigkeitsverluste (!!!) aus einer Hochsprache ernsthaft Grafik programmieren könnte...

Speicher:

Beim Video-RAM unterscheidet man nicht nur nach der Größe, sondern auch nach der Art der Speicherchips. Ursprünglich gab es nur DRAMs, welche allerdings sehr langsam sind. Sie können nämlich nur entweder

[4] Das GDI (Graphical Device Interface) ist ein von Windows zur Verfügung gestellter Speicherbereich, in den alle Windows-Anwendungen ihre Grafikdaten schreiben. Windows kombiniert dann diese Fensterinhalte nach der derzeitigen Desktop-Gestaltung und stellt sie über den installierten Grafiktreiber dar.

gelesen oder beschrieben werden. Die nächste Generation bildeten dann die VRAMs, welche gleichzeitig ausgelesen und beschrieben werden konnten. Sie sind deshalb um mindestens die Hälfte schneller, allerdings auch empfindlich teurer. Heute kommen meistens VRAMs und nur noch selten DRAMs zum Einsatz. Der letzte Schrei sind WRAMs, welche noch einmal um cirka 50 % schneller und dabei sogar billiger zu produzieren sind. Derzeit sind zwar erst sehr wenige Karten mit ihnen bestückt, sie dürften in nächster Zeit aber sehr große Verbreitung erlangen. [5]

Speicherverwaltung:

Beim Programmieren sieht es normalerweise so aus, daß Teile des Video-RAMs (die über Register selektiert werden) in den konventionellen Speicherbereich eingeblendet werden (Zur leichteren Vorstellung: ähnlich wie bei EMS). Um genau zu sein, liegt der Grafikpuffer zwischen A000 und AFFF, der Monochrom-Textpuffer zwischen B000 und B7FF (der allerdings bei Farbsystemen häufig von diversen Memorymanagern verwendet wird) und der Farb-Textpuffer zwischen B800 und BFFF. Auf der Grafikkarte wird die lineare Adressierung dann in eine planare umgewandelt und die entsprechenden Punkte gesetzt, abhängig natürlich von den Registereinstellungen. Dies gilt aber alles nur für **VGA** und höher (und ist heute als einziges noch interessant), bei früheren Grafikstandards sah die Sache teilweise etwas anders aus. [6]

Mode 13h:

Dies ist der 256-Farbenmodus des Standard-VGA und wohl einer der am häufigsten verwendeten. Er ist kompatibel zu MCGA, bezieht daher aber auch eine seiner Schwächen, nämlich die interne Halbierung der Auflösung auf 320 x 200 durch Setzen der entsprechenden Register. Auf VGA-Karten mit mindestens 512 KB Video-RAM kann sich ein Profiprogrammierer durch Löschen dieser Register daher sehr einfach einen High-Res-256-Farbenmodus schaffen, muß dann allerdings auf BIOS-Unterstützung verzichten[7]. Kehren wir aber zurück zum Mode 13h. Hier macht man sich das **Chaining** der Bit-Planes zu einem linearen Adreßraum zunutze, das heißt der VGA-Adreßraum wird ab Segment A000 im konventionellen Speicher eingeblendet. Man kann dadurch einen Punkt sehr einfach durch folgende Formel adressieren:

Offset=Y x 320 + X

An die entsprechende Adresse schreibt man nun einfach den Farbwert aus der Palette, die man natürlich vorher in den VGA schreiben muß. Die Palette bietet hierbei 256 Farben von 262.144 und ist 768 Bytes groß. Für jede Farbe stehen dabei nacheinander die Rot-, Grün- und Blau-Werte. Da an der Punktadresse nur ein Pointer auf die entsprechende Farbe steht, spart dieser Modus sehr viel Speicher.[8] Intern wandelt der VGA die lineare dann wieder in eine planare Adressierung um. Dazu werden Bit 0 und 1 des Offsets zur Selektierung 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.

Grafikformate:

GIF:

GIF ist eines der am häufigsten verwendeten Grafikformate und das Standardformat von CompuServe. Es wurde 1987 entwickelt, ist systemunabhängig und bietet eine hervorragende Kompression. GIF erlaubt Bilder bis zu einer Auflösung von 16.000 x 16.000 Punkten bei einer Palette von 256 Farben aus 16.7 Mio. Intern baut GIF auf einer Blockstruktur auf, auf die hier aber nicht näher eingegangen werden soll (siehe Referenzen am

---

[5] Darüber hinaus werden in High-End-Maschinen (Silicon Graphics) auch noch SRAMs eingesetzt, welche zwar enorm schnell, aber leider auch schweineteuer sind.

[6] Wer sich heute noch für die Programmierung älterer Grafikkarten interessiert, sei an die entsprechende Fachliteratur verwiesen (siehe Textende). Wir wollen aus Gründen der Aktualität und Übersichtlichkeit nicht näher darauf eingehen.

[7] Profis adressieren die VGA-Karte aber meistens sowieso direkt, da das BIOS um ein Vielfaches langsamer ist und viele Spezialeffekte (SFX) überhaupt nur durch direkte Programmierung möglich sind.

[8] Bei mehr als 256 Farben geht dieser Vorteil natürlich verloren, weshalb man dann wieder für jeden Punkt seine RGB-Werte schreibt (Man stelle sich einmal eine Palette für 16.7 Mio. Farben vor...).

Textende). Als Packverfahren wird ein modifizierter LZW-Algorithmus (Lempel, Ziv, Welch) verwendet, der aber ebenfalls hier nicht erklärt werden kann[9].

BMP:

BMP steht für Bitmap, was auch gleich den internen Aufbau dieser Files sehr gut beschreibt. Es ist das Standardformat von Microsoft und wird unter Windows häufig verwendet. BMP ist an und für sich ein sehr einfaches Format und auch sehr schnell.

RLE:

RLE ist die komprimierte Form von BMP. Die Abkürzung RLE steht eigentlich für *Run Length Encoding* und bezeichnet das Packverfahren, das hier zum Einsatz kommt. Dabei wird bei sich wiederholenden Bytes einfach die Anzahl der Wiederholungen und dann das eigentliche Byte geschrieben. Es ist also offensichtlich ein sehr einfaches Pachverfahren, das schlechte Packraten durch hohe Geschwindigkeit und geringen Rechenaufwand kompensiert.

PCX:

Dieses Grafikformat wurde ursprünglich von ZSoft für ihr Programm Paintbrush entwickelt und ist heute ebenfalls sehr verbreitet. Auch PCX nutzt RLE zur Kompression der Bilddaten.

TIF:

TIF ist ein Grafikformat mit sehr ausgeprägter Blockstruktur und hervorragender Kompression, das aber leider selten unterstützt wird.

JPG:

Dieses Format nutzt intern das JPEG-Packverfahren, welches eine der besten Kompressionsraten überhaupt bietet, allerdings leider auch verbunden mit einem enormen Aufwand an Rechenzeit.

MPG:

Dies ist ein Format für Videos, das intern das MPEG-Packverfahren nutzt. MPEG ist eine Weiterentwicklung von Motion-JPEG, und es gilt das gleiche, das auch schon bei JPG gesagt wurde. Heute wird oft schon MPEG-2 verwendet, wobei Kompressionsrate und Geschwindigkeit noch einmal verbessert wurden. Der Rechenaufwand ist dabei aber so gigantisch, das spezielle Zusatzhardware (entweder in Form einer eigenen Steckkarte oder häufig auch schon auf Grafikkarten der Oberklasse integriert) benötigt wird, um das Ganze mit halbwegs annehmbarer Systembelastung abzuspielen.

AVI:

Auch AVI ist ein Video-Format, das von Microsoft entwickelt wurde. Der Rechenaufwand ist hier ebenfalls sehr hoch, allerdings bietet AVI leider nur Briefmarkenvideos. Diese können zwar auf Vollbild vergrößert werden, wirken dann aber arg pixelig. AVI wird unter Windows häufig verwendet.

VGA-Register:

Für professionelle Grafikprogrammierung ist es unerläßlich, die Register des VGA zu kennen. Niemand wird diese allerdings auswendig lernen, was auch sinnlos wäre. Wir wollen an dieser Stelle aus Platzgründen deshalb gar nicht näher auf dieses Thema eingehen und verweisen auf die vielfältige Sekundärliteratur (siehe Textende). Vorsicht ist allerdings bei Änderungen des VGA-Timings geboten. Extreme Werte können die Grafikhardware **irreparabel beschädigen** !!! Naiven Anfängern muß gesagt werden, daß Profis aus einer Standard(!)-VGA-Karte (allerdings ohne Bilddarstellung) Frequenzen von weit über 100 Hz herausholen können! Dieses Gebiet ist also für Experimente offensichtlich zu gefährlich und deshalb hoffentlich tabu! Aus eben diesen Gründen sind die entsprechenden Register auch extra geschützt (*Protection-Bit* 7 des CRTC-Registers 11), und wer dieses Bit löscht, sollte besser wissen, was er tut (!!!) !

Um aber trotzdem einen kleinen Einblick in das Innenleben einer VGA-Karte zu gewähren, hier nun eine kurze Vorstellung der einzelnen Register:

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		

[9] Genauere Ausführungen über diesen hochinteressanten und sehr leistungsfähigen Algorithmus würden ganze Bücher füllen. Wir möchten hier aber noch einmal auf unsere Referenzen verweisen, die auch diesem Thema einige Seiten widmen.

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
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 X:

Bei der Verwendung von **Sprites** (siehe unten) ergibt sich die Notwendigkeit von Bildschirmseiten. Diese Seiten können einfach hintereinander im Bildschirmspeicher abgelegt werden und über Register 0Ch und 0Dh (Linear Starting Address) gewählt werden. Das Problem besteht beim **Mode 13h** (siehe oben) 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[10]. Modifikationen sind also nur über die Segmentselektoren des VGA möglich, die aber bei jedem Hersteller anders programmiert werden. Die Lösung dieses Problems findet sich im Mode X, der auch eine hohe Zugriffsgeschwindigkeit bietet. 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.

Initialisierung:

Das Wichtigste ist die Abschaltung des *Chain-4*-Mechanismus, 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 (es lebe die Kompatibilität) muß noch der Speicherzugriff auf Byte-Adressierung 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 jetzt noch der Bildschirmspeicher gelöscht, weil an den vom Mode 13h unbenutzten Stellen des Bildschirmspeichers, die jetzt sichtbar werden, noch "Bit-Müll" aus anderen Videomodi stehen kann. Am einfachsten geht das über das *Write Plane Mask Register 2* des Timing-Sequenzers, in dem zum Löschen des Bildschirmspeichers alle Planes eingeschaltet werden, so daß 32.000 Word-Zugriffe oder 16.000 DWord-Zugriffe ausreichen, um alle vier Bildschirmseiten zu löschen. Die weitere Grafikprogrammierung muß nun aber direkt in Assembler erfolgen, da weder vom BIOS noch irgendeiner Hochsprache auch nur die geringste Unterstützung zu erwarten ist.

Aufbau:

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, daher auch der Begriff der Plane (Ebene). Die Planes sind quasi unabhängige Speicher, die sich einzeln ansprechen lassen, aber bei der Darstellung des Bildes auf dem Monitor parallel verwendet werden, d. h. die Daten aus allen vier Planes werden gleichzeitig gelesen. Im weiteren Aufbau gibt es jedoch gravierende Unterschiede zwischen den 16-Farben-Modi und Mode X. 16 Farben lassen sich durch 4 Bit darstellen, daher auch die 4 Planes. Hier wird nämlich Bit 0 eines Punktes im entsprechenden Bit der Plane 0 gespeichert, Nit 1 in Plane1 usw. Anders sieht es dagegen im Mode X aus, hier reichen 4 Bit für die Adressierung eines Punktes nicht mehr aus, so daß ein Punkt jetzt ein Byte einer bestimmten Plane verwendet. Dabei werden die Planes byteweise aufgefüllt, d. h. Punkt 0 befindet sich an Offset 0 in Plane 0, Punkt 1 am gleichen Offset in Plane 1, erst Punkt 4 steht an Offset 1, wobei wieder Plane 0 verwendet wird.

Die Punktnummer läßt sich im Mode X errechnen wie im Mode 13h (320 x Y + X). Plane und Offset lassen sich nach folgenden Formeln berechnen:

Plane = X mod 4

Offset = Y x 80 + X div 4

Dieses Verfahren entspricht exakt dem, das der Mode 13h standardmäßig verwendet, mit einem Unterschied: Der Offset wird durch Shiften der Punktnummer um 2 Bit nach rechts[11] gebildet, nicht durch Maskieren, so daß im Speicher keine Lücken zwischen den Punkten entstehen und somit vier Seiten in die 256 KB Bildschirmspeicher 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 (wie schon beim Bildschirmlöschen gezeigt). Daher wird eine Maske in Register 2 *(Write Plane Mask)* des TS gesetzt, die erst aus der Plane-Nummer erzeugt werden muß. Es gilt folgende Formel:

Maske = 1 shl Plane-Nummer

Aus Plane 2 wird somit die Maske 1 shl 2 = 4 = 0100.

---

[10] Im **Protected Mode** stellt sich dieses Problem natürlich nicht, meistens wird aber im **Real Mode** programmiert. Der Protected Mode ist wegen einiger Schwierigkeiten auch nur erfahrenen Programmierern zu empfehlen, weiters muß man auf die Kompatibilität (!) achten.

[11] Hier ist wohl eine Erklärung für Leute, die mit Mathematik auf Kriegsfuß stehen, angebracht: Die Gleichung Y x 320 + X wurde einfach durch 4 dividiert, also Y x (320/4) + X/4, gekürzt Y x 80 + X div 4. X div 4 kann man nun aber auch als X div 2^2 schreiben, was sich wiederum durch ein Shiften der Zahl um 2 Bit (Exponent!) nach rechts am schnellsten rechnen läßt. Hier kann man vor allem in Assembler sehr leicht Anfänger und Profis unterscheiden, denn Profis berechnen sämtliche Multiplikationen und Divisionen mit Zweier-Potenzen durch Shiften, was um ein Vielfaches schneller geht.

Ein Wort noch zur byteweisen Adressierung seitens der CPU: Es drängt sich förmlich die Versuchung auf, mittels 32-Bit-Zugriffen auf die Grafikdaten zuzugreifen (wozu hat man denn einen 386er?) oder wenigstens 16 Bit zu nutzen (das kann theoretisch schon der XT). Wer dies versucht, wird jedoch schon bald eines Besseren belehrt: Das Bild wird völlig verzerrt dargestellt, was sich durch die Vorgehensweise der CPU beim Speicherzugriff erklären läßt. Wenn die CPU ein Word (entsprechendes gilt für ein Doubleword) kopiert (per *movsw*), zerlegt sie diesen Kopiervorgang natürlich nicht in einzelne Byte-Bewegungen, sondern liest ein Wort komplett und schreibt es wieder komplett zurück. Der VGA kann jedoch in seinen vier Latches, die ja als Zwischenspeicher dienen, nur 4 Byte aufnehmen. Daher stehen nach dem Lesezugriff dort nur die 4 High-Bytes der gelesenen Words. Beim folgenden Schreibzugriff werden sowohl die 4 High- als auch die Low-Bytes auf den gleichen, den Latches entsprechenden Wert gesetzt. Am Bildschirm zeigt sich dies darin, daß immer zwei (bei 32-Bit-Zugriffen sogar vier) aufeinanderfolgende Viererblöcke den gleichen Inhalt haben, eine vernünftige Bilddarstellung also nicht mehr möglich ist.

Höhere Auflösungen im Mode X:

Der Mode X bietet den großen Vorzug, mit vier Bildschirmseiten arbeiten zu können, aber an der Auflösung hat er gegenüber seinem Vorgänger Mode 13h nichts geändert. Für bestimmte Anwendungen ist aber eine höhere Auflösung erforderlich. Dafür gibt es selbstverständlich die Super-VGA-Auflösungen, die zwar für jede Karte unterschiedlich angesprochen werden, was man aber zur Not über VESA-Treiber in den Griff bekommen kann. Das größere Problem taucht dann auf, wenn man auch in der hohen Auflösung mehrere Bildschirmseiten benutzen möchte. Zwar sind kaum noch VGA-Karten im Verkehr, die weniger als 1 MB Speicher besitzen, so daß selbst der 800 x 600-Punkte-Modus theoretisch zwei Bildschirmseiten im Speicher unterbringen könnte, aber eine Verwaltung von Bildschirmseiten ähnlich dem *Linear-Starting-Address*-Register ist unter VESA nicht vorgesehen; eine direkte Programmierung wirft wieder das bei Super-VGA unvermeidliche Kompatibilitätsproblem auf. Auf der Suche nach einem höherauflösenden Modus, der aber auch das Seitenkonzept unterstützt, fällt auf, daß der Mode X ganze vier Bildschirmseiten unterstützt, man aber in vielen Fällen nur zwei benötigt. Eine Verdoppelung der Auflösung auf 320 x 400 ist also ohne weiteres möglich. Daß 320 x 400 und nicht 640 x 200 verwendet wird, hängt mit dem eigentlichen Aufbau der 200-Zeilen-Modi auf VGA-Karten zusammen. 200 Zeilen können VGAs nämlich gar nicht explizit ansprechen (*Miscellaneous-Output*-Register, Bit 6-7 erlauben nur die Werte 350, 400, 480 und teilweise 768 Zeilen vertikale Auflösung). Daß sie es dennoch können, verdanken sie einer Fähigkeit, genannt *Double-Scan*. Das bedeutet nichts anderes, als daß bei einer physikalischen Vertikalauflösung von 400 Zeilen jede Zeile zweimal dargestellt und somit in y-Richtung verdoppelt wird, was zu einer Auflösungshalbierung führt. Dabei muß weder das horizontale noch das vertikale Timing verändert werden, weil physikalisch weiterhin 400 Zeilen zu je 320 Punkten dargestellt werden. Das macht es sehr einfach, diesen Mechanismus wieder auszuschalten: Im Register 9 des CRTC *(Maximum Row Address)* müssen 5 Bit gelöscht werden, Bit 7 und Bits 0 - 3. Je nach VGA-BIOS erfolgt die Verdoppelung über das eigentlich dafür vorgesehene Bit 7 *(Double Scan Enable)* oder Bits 0 - 3, die im Text

für Seitenprogrammierung maximalen Speicherverbrauch hinausschießen. Eine Seite läßt sich zwar auf jeder Standard-VGA darstellen, zwei Seiten überschreiten jedoch die 256 KB, die für die allgemeingültigen Adressierungsmethoden die oberste Grenze der Adressierbarkeit darstellen. Für höhere Adressen kommt das Problem der Inkompatibilität der Super-VGA-Karten zum Tragen, so daß man auch gleich einen Super-VGA-Modus verwenden kann, der dann wenigstens ein vernünftiges Seitenverhältnis aufweist.

Grabben:

Grabben (oder auch Capturen) bedeutet das “einfangen” des Bildschirminhalts (oder eines Teils davon). Dieser kann dann entweder in eine Datei gespeichert oder im Speicher verschoben werden, eventuell auch innerhalb des Video-RAMs oder wieder zurück in diesen. Ernsthaftes Grabben greift sehr weit in das System ein und ist deshalb auch nur in Assembler möglich[12]. Im allgemeinen arbeitet man Interrupt-gesteuert, das heißt man installiert seine Capture-Routine auf einen Interrupt (der natürlich frei sein muß, im Zweifelsfall vorher überprüfen !). Da meistens auf Tastendruck gegrabbt werden soll, hängt man sich auch in den Tastaturinterrupt ein, es sind aber auch andere Auslöser möglich. Wird die Routine nun aufgerufen, muß sie als

dargestellten Bereichs somit frei im RAM verschieben. Wird ein neues Bild durch den Kathodenstrahl aufgebaut, bezieht dieser die Bilddaten nicht mehr aus dem Anfang des Bildschirmspeichers, sondern vom Beginn der im Register *Linear Starting Address* festgelegten Stelle. Auf diese Weise läßt sich der physikalische Bildschirm, d. h. der Bereich, der letzten Endes auf dem Monitor erscheint, wie ein Fenster über das ganze VGA-RAM verschieben, so daß immer ein beliebiger Ausschnitt eines 256 KB großen Bildes angezeigt wird. Verschiebt man diesen Bildschirmstart innerhalb des RAM, bewegt sich das Fenster in diese Richtung, so daß die Grafik auf dem Monitor entgegengesetzt bewegt wird. Eine Erhöhung der *Linear Starting Address* um 80 bedeutet also eine Verschiebung des Bildinhalts um eine Zeile (320 Pixel / 4 Planes = 80 Byte lang) nach oben. Der Inhalt der Zeile 0 befindet sich außerhalb des Fensters, und Zeile 200 rückt von unten nach und erscheint als unterste Zeile. Jetzt sind also die Zeilen 1 - 200 sichtbar. Durch weiteres Erhöhen des Registers in Schritten von 80 Byte kann man unter minimalem Einsatz von Rechenleistung ein vertikales Scrolling des gesamten Bildschirminhalts erreichen. Oft möchte man nicht nur vertikal, sondern auch horizontal scrollen, man denke nur an ein vier Bildschirmseiten großes Logo, das immer nur ausschnittsweise sichtbar ist und sozusagen “unter” dem Bildschirm hin und her bewegt wird. Die Lösung scheint sehr einfach: Bisher wurde das Register *Linear Starting Address* immer noch in Schritten von 80 bewegt, warum also nicht einfach in Einerschritten zählen? In diesem Fall wird jedoch das, was am linken Bereich herausgescrollt wird, direkt am rechten Rand wieder angefügt. Dies resultiert einfach daraus, daß die Zeilen immer noch direkt hintereinander im Speicher liegen. Hat man nun den Bildschirmstart zum Beispiel um ein Byte verschoben (Linksverschiebung um 40 Pixel), so stellt der CRTC in der ersten Rasterzeile die Bytes 1 bis 80 (statt normalerweise 0 - 79) dar; das Byte 80 gehört jedoch eigentlich schon zur zweiten Zeile der Vorlage, dadurch wird jedes Byte, das am linken Rand herausgescrollt wird, eine Zeile höher am rechten Rand wieder angefügt. Die Lösung des Problems liegt darin, den Bildschirm virtuell auf 640 Pixel Breite zu vergrößern. Angezeigt werden dabei weiterhin nur 320 Punkte, aber rechts “neben” dem Monitor werden weitere theoretische Punkte angefügt, so daß die vier Bildschirmseiten nicht mehr übereinander liegen wie beim vertikalen Scrolling, sondern ein Quadrat bilden. Dann erscheinen links herausgescrollte Punkte ganz rechts außen (x-Koordinaten 636 - 639) im unsichtbaren Bereich, und neue Punkte werden vom unsichtbaren Bereich in den sichtbaren hineingescrollt. Anders gesagt kann man das Fenster, das der Bildschirm ja eigentlich im Mode X darstellt, auch horizontal bewegen, weil jetzt Platz dazu da ist (waagrecht nebeneinander liegende Bildschirmseiten). Doch wie erzeugt man diesen besonderen Modus? Der VGA besitzt natürlich auch hierfür ein Register. Es handelt sich dabei um das Register 13h des CRTC: *Row Offset*. Hinter diesem unscheinbaren Namen verbergen sich ungeahnte Nöglichkeiten. Hier wird nämlich die Sprungweite angegeben, um die beim Erreichen des rechten Rands durch den Kathodenstrahl der interne Zeiger auf die Daten *(Linear Counter)* weiterbewegt werden soll. Dies entspricht also dem Abstand der Zeilen innerhalb des Bildschirmspeichers, also deren Länge! Normalerweise enthält dieses Register sowohl im Mode 13h als auch im Mode X den Wert 40, was einer Breite von 80 Byte entspricht. Das Register zählt in Word-Schritten, d. h. 80 Byte Breite werden als 40 Word gezählt. Im Mode 13h sind die Zeilen zwar 320 Byte lang, aber auch die Berechnungsgrundlage, die der größten durch die CPU adressierbaren Einheit entspricht, wird vervierfacht von einem Byte auf ein Doubleword, wodurch als programmierter Wert wieder 320/8 = 40 herauskommt. Nun lassen sich natürlich auch größere Werte einsetzen. Schreibt man in dieses Register zum Beispiel den Wert 80, heißt das, daß die Zeilen einen Abstand, also eine Länge, von 160 Byte haben (entspricht 640 Pixel). Dadurch entstehen natürlich Lücken von 80 Byte Länge zwischen den einzelnen 80-Byte-Zeilen, die von den rechts überstehenden, unsichtbaren Zeilenhälften aufgefüllt werden.

Zusammenfahren eines Bildes:

Eine Anwendung der Kombination aus **Split Screen** und **Scrolling** besteht im Zusammenfahren eines Bildes aus zwei Hälften. Dabei wird die obere Hälfte, die von oben zur Mitte gefahren wird, durch die *Linear Starting Address* kontrolliert, sie wird also einfach nur vertikal (in diesem Fall nach unten) gescrollt. Dabei ist darauf zu achten, daß Bildschirmseite 1 leer oder mit einer bestimmten Farbe gefüllt ist, da sie zu Beginn zur Hälfte sichtbar ist. Die untere Hälfte wird durch das Splitting gesteuert: Durch Verringern der Zeilennummer wird der Beginn des Split-Screens nach oben geschoben, also zur Mitte hin.

Smooth-Scrolling im Textmodus:

Scrolling im Textmodus funktioniert eigentlich genauso wie im Grafikmodus: Über die Startadresse *Linear Starting Address)* wird der sichtbare Ausschnitt des Bildschirmspeichers verschoben. Die Sache hat jedoch einen Haken: Dieses Scrolling ruckelt stark, weil immer um ganze Zeichen verschoben wird, während im Grafikmodus um einzelne Punkte verschoben werden kann. Das liegt am Aufbau des Bildschirmspeichers im Textmodus: Informationen für einzelne Punkte liegen hier nicht mehr vor, so daß auch die *Linear Starting Address* sich immer auf ganze Zeichen bezieht und nur ein sehr grobes Scrolling erlaubt. Die Rettung kommt mal wieder von neuen Registern des VGA, die nur darauf warten, sich dieser Problematik anzunehmen. Es handelt sich dabei um das horizontale und das vertikale *Panning*-Register. Unter Panning versteht man generell die Verschiebung des Bildinhalts um einen Punkt, wobei diese beiden Register eben dies auch im Textmodus erlauben. Um nun sanft zu scrollen, verschiebt man einfach den Bildinhalt durch *Panning* in Einzelschritten in die gewünschte Richtung. Hat man dabei ein Zeichen weit grscrollt, setzt man das entsprechende *Panning*-Register wieder auf seinen Ursprungswert und modifiziert nun das Register *Linear Starting Address*. Dies ist

erforderlich, weil *Panning* nur bis zu einem Zeichen Breite bzw. Höhe möglich ist und somit nur zur Feinsteuerung dienen kann, während di

Video-Signal eine bestimmte Farbe anzeigt (meist Blau), ein Fernsehbild einblendet. Auch hier erfolgt die Mischung auf sehr niedriger Ebene und nicht im langsamen Bildschirmspeicher. Wie erzeugt man nun aber die verschiedenfarbigen Zeilen? Home-Computer besitzen meist einen sogenannten Rasterzeileninterrupt. Hier kann der Videocontroller programmiert werden, bei Erreichen einer bestimmten Rasterzeile einen Interrupt auszulösen, so daß sehr schnell auf dieses Ereignis reagiert und die Farbe neu gesetzt werden kann. Verfügen einige VGAs noch über einen Vertical-Retrace-Interrupt (oft über Dip-Schalter deaktiviert), so sieht die Sache beim Horizontal-Retrace wesentlich magerer aus. Keine uns bekannte Grafikkarte unterstützt diesen Interrupt. Also bleibt keine andere Wahl, als ständig den Zustand der VGA zu überwachen und die Rasterzeilen mitzuzählen, um in der gewünschten Zeile die Farbe zu ändern. Dazu muß zunächst ein definierter Ausgangszustand geschaffen werden, indem auf einen vertikalen Retrace gewartet wird. Wird das nächste Mal die *Display Enable*-Leitung aktiviert (abzulesen im *Input-Status*-Register, Bit 0 - *Display Enable Complement*), kann man sicher sein, daß man sich in Rasterzeile 0 befindet und kann durch permanentes Abfragen dieses Bits die aktuelle Zeile mitzählen. Weil die Darstellung einer Rasterzeile im Vergleich zur Darstellung des ganzen Bildes extrem kurz dauert (et

Ende des Programms wieder gesetzt werden. Schließlich muß es wohl einen Sinn ergeben, daß das *Protection*-Bit standardmäßig gesetzt ist.[14]

Paletten-Effekte:

Die Palette bietet hervorragende Möglichkeiten, durch wenige Befehle - und damit in kürzester Zeit - den gesamten Bildschirm zu verändern, weil alle Punkte eines Farbwertes auf einen Schlag auf eine neue Farbe gesetzt werden. In Nicht-Paletten-Modi müßte jeder einzelne Pixel neu gesetzt werden, was natürlich einen immensen Zeitaufwand bedeutet.

Ausblenden:

Der einfachste Effekt, der sich auf diese Weise erreichen läßt, ist das Ausblenden eines Bildes. Ähnlich wie beim Film wird dabei in relativ kurzer Zeit die Helligkeit des Bildes vom Normalwert auf Null reduziert. Dies läßt sich sehr einfach durch die Palette realisieren, indem in einer Schleife alle Farbwerte um 1 vermindert werden und die neu berechnete Palette dann gesetzt wird. Nun wartet man auf den nächsten Retrace und zählt wieder um 1 herunter, bis alles schwarz ist. An dieser Stelle zeigt sich auch ein großer Vorteil der direkten DAC-Programmierung gegenüber der Verwendung des BIOS: Ein Paletten-Zugriff im Text-Mode ist über das BIOS gar nicht möglich, während er im Grafikmodus lediglich unerträglich langsam ist. Daher bleibt in diesem Fall nur die direkte Registermanipulation.

Einblenden:

Die andere Richtung des Blendens ist das Einblenden, bei dem, von einem schwarzen Bild ausgehend, die Helligkeit bis zur eigentlichen Bildpalette hochgezogen wird. Im Prinzip funktioniert dies wie beim Ausblenden: Die Farbwerte werden bei jedem Durchlauf - natürlich synchronisiert mit dem Vertical-Retrace - immer um 1 oder mehr erhöht, bis der Zielwert (aus der Originalpalette des Bildes) erreicht ist. Als Abbruchbedingung kann natürlich nicht mehr das Erreichen von 0 herangezogen, sondern es muß ständig mit dem Zielwert verglichen werden.

Blenden von beliebiger Quelle auf Zielpalette:

Bis jetzt wurde immer entweder von einer Palette zu Schwarz oder umgekehrt geblendet. Was noch fehlt, ist ein Überblenden von einer konkreten Palette zu einer anderen. Dies scheint auf den ersten Blick keinen großen Sinn zu haben, weil sich außer einer schrittweisen Farbverfälschung des Bildes nicht viel erreichen läßt. Irrtum! Auf diese Weise läßt sich ein sehr schöner Effekt erzeugen. Wie wäre es zum Beispiel damit, daß ein Demoteil zu Ende geht und daraufhin das letzte Bild langsam auf Schwarzweiß-Darstellung heruntergerechnet wird? Im Vordergrund läßt sich dann zum Beispiel ein Text mit den **Credits**, also den Danksagungen, darstellen. Hier zeigt sich der Nutzen der neuen Prozedur: Nach Berechnung der Schwarzweiß-Palette muß nur noch auf diese herübergeblendet werden. Das erste Problem, das sich dabei stellt, ist: Wie erzeuge ich eine schwarzweiße Palette? Zum Glück besitzt das BIOS eine Funktion dafür, die hier zwar nicht benutzt werden soll (langsam, unflexibel), von der man aber die Funktionsweise sehr gut abgucken kann. Um eine Farbe, die sich aus Komponenten von Rot, Grün und Blau zusammensetzt, auf Schwarzweiß herunterzurechnen, muß man die drei Farbanteile zusammenzählen und mit diesem Wert in der neuen Palette alle drei Farben beschreiben. Werden alle drei Farben zu gleichen Teilen gemischt, kommt immer eine Abstufung zwischen Schwarz und Weiß heraus. Die Frage ist nur, wie die Summe zu bilden ist. Einfach alle drei Farben zu gleichen Teilen einfließen zu lassen, führt zu keinem befriedigenden Ergebnis. Das menschliche Auge empfindet die verschiedenen Farben nämlich nicht als gleich hell. Ein blauer Punkt von maximaler Helligkeit erscheint zum Beispiel wesentlich dunkler als ein grüner. Ebenfalls vom BIOS abgeguckt ist das optimale Mischverhältnis, das die naturgetreueste Darstellung gewährleistet: Man nehme 30 % des Rotanteils, 59 % des Grünanteils und 11 % des Blauanteils.

Überblenden von einem Bild zum nächsten:

Es ist schon wesentlich professioneller, beim Wechsel von einem Bild auf das nächste zunächst das erste auszublenden, um dann das nächste einzublenden, anstatt abrupt umzuschalten; dazu sind Sie mit den erarbeiteten Prozeduren bereits in der Lage. Für wirklich professionelle Bildwechsel kommt man allerdings nicht darum herum, einen fließenden Übergang von einem Bild zum anderen zu erzeugen, man muß also beide Bilder mischen. Zu diesem Zweck gibt es mittlerweile zahlreiche sogenannte **Morphing**-Programme[15], die allerdings den Nachteil vorberechneter Bilder haben: Die Bilder müssen erst einmal aufwendig berechnet werden, nehmen viel Platz auf der Platte und im Speicher ein und müssen umständlich (und langsam) in den Bildschirmspeicher kopiert werden. Dies ist also nur eine Lösung für Anfänger, der Profi berechnet seine Blenden in Echtzeit! Bei Palettenbildern - und die behandelten Grafikmodi sind alle palettenbasiert - läuft das Blenden, wie sich noch zeigen wird, im Endeffekt auf ein einfaches Paletten-Blenden hinaus. Erst bei

[14] Wenn man nicht gerade das Timing manipuliert, sollte das Protection-Bit unbedingt immer gesetzt sein! Ansonsten könnte durch einen “Unfall” (Programmfehler, ...) oder einen sonst harmlosen Virus großer Schaden angerichtet werden. Es wurden bereits viele Monitore durch sträflichen Leichtsinn oder gedankenlose Bequemlichkeit “geschossen” !

[15] Siehe hierzu auch letztjähriges Referat (Christof Zimmermann).

komplizierten Metamorphosen, die gleichzeitig die Bildteile durch Bewegung ineinander überführen, muß man also die Morphingsoftware zu Rate ziehen. Man blendet also nur eine Palette in eine andere über. Weil aber die beiden Bilder übereinanderliegen und eine Veränderung der Bilddaten (das ist der Weg des Morphing) aus Geschwindigkeitsgründen nicht in Frage kommt, müssen dieselben Bilddaten je nach Palette verschiedene Bilder darstellen. Zunächst ist die Quellpalette aktiv, wodurch die Daten das Quellbild darstellen. Am Schluß ist dann die Zielpalette aktiv, woraufhin dieselben Bilddaten, die vorher das Quellbild dargestellt haben, jetzt für das Zielbild stehen. Bei dieser Vorgehensweise stellen sich zwei Fragen:

1. Wie müssen die Bilddaten manipuliert werden, daß sie - je nach Palette - sowohl das ursprüngliche (Quell-) Bild oder aber das Zielbild darstellen?
2. Wie müssen die Paletten beschaffen sein, um aus den gleichen Bilddaten unterschiedliche Bilder zu erzeugen?

Zunächst zu Frage 1: Bei dieser Art von Blende haben wir es mit einem grundsätzlich anderen Problem als bisher zu tun. In den bisherigen Blenden war entweder die Zielfarbe für alle Punkte gleich, nämlich Schwarz (Ausblenden), oder die Ausgangsfarbe (Einblenden). Nun muß jede beliebige Farbe in jede beliebige andere übergeblendet werden. Es gibt rote Punkte, die im Zielbild grün sein müssen, aber auch rote, die später blaue Farbe annehmen sollen. Wer in Mathematik (Bereich der Stochastik, Thema Kombinatorik) gut aufgepaßt hat, weiß, was das für die Farbenpracht bedeutet: Bei der Kombination einer bestimmten Anzahl Farben mit einer gleichen Anzahl anderer Farben gibt es genausoviele Möglichkeiten, wie das Quadrat der Farbanzahl beträgt. Jede dieser Kombinationen muß beim Überblenden berücksichtigt werden, muß also einen Eintrag in der beim Blenden verwendeten Palette haben. Um ein Zweifarbbild in ein anderes zu überführen, werden vier Paletteneinträge benötigt (Farbe 0 nach Farbe 0, also gleichbleibend, Farbe 0 nach Farbe 1, 1 nach 0 und 1 nach 1). Bei zwei Vierfarbbildern werden bereits 16 Einträge belegt. Weil auf dem VGA nun einmal nur eine Palette von 256 Farben besteht, liegt also die größtmögliche Farbanzahl, die sich ohne Bilddaten-Veränderung überblenden läßt, bei 16 Farben. Beim mehrmaligen flimmerfreien (!) Überblenden von Bildern wird diese Zahl jedoch, wie weiter unten erklärt, noch weiter reduziert auf 15. Aus diesen kombinatorischen Überlegungen läßt sich auch bereits die Vorgehensweise beim Mischen der Bilder ableiten: Jede Kombination muß vertreten sein, dazu verwendet man N Blöcke mit jeweils N Einträgen, wobei N für die Anzahl Farben pro Bild steht. Die Blocknummer entspricht der Zielfarbe, während der Index innerhalb des Blocks der Quellfarbe entspricht. Andersherum ist es zwar auch denkbar, dies würde jedoch den später erläuterten Reset unnötig verkomplizieren. Um die Farbnummer zu ermitteln, geht man einfach nach der folgenden Formel vor:

Farbnummer = Zielfarbe x Anzahl Farben + Quellfarbe

Dieses Prinzip kommt manchem vielleicht bekannt vor: Auf die gleiche Weise rechnet man ein hexadezimales Byte in eine dezimale Zahl um: *Oberes Nibble (höherwertige Stelle) x 16 + unteres Nibble*. Das würde bei einer Farbanzahl von 16 die Sache sehr beschleunigen, weil einfach die Zielfarbe in das obere Nibble und die Quellfarbe in das untere Nibble geladen werden könnte, und man hätte den Farbwert, der benutzt werden muß. Die Sache hat nur den Haken, daß 16 Farben hier nicht verwendet werden können, sondern höchstens 15, also müssen wir beim langsamen Multiplizieren bleiben. Dies ist jedoch nicht weiter tragisch, da der Programmteil, der die Multiplikation beinhaltet, nicht zeitkritisch ist und höchstens vor der eigentlichen Blende eine kaum merkliche Zeitverzögerung bewirkt. Die folgende Skizze veranschaulicht noch einmal die Blockbildung für den Fall, daß zwei Vierfarbbilder ineinander übergeblendet werden sollen:

Farbwert 0	Farbwert 1	Farbwert 2	Farbwert 3	Farbwert 4	Farbwert 5	Farbwert 6	Farbwert 7	Farbwert 8	Farbwert 9	Farbwert 10	Farbwert 11	Farbwert 12	Farbwert 13	Farbwert 14	Farbwert 15
Quellfarbe 0	Quellfarbe 1	Quellfarbe 2	Quellfarbe 4	Quellfarbe 0	Quellfarbe 1	Quellfarbe 2	Quellfarbe 4	Quellfarbe 0	Quellfarbe 1	Quellfarbe 2	Quellfarbe 4	Quellfarbe 0	Quellfarbe 1	Quellfarbe 2	Quellfarbe 4
Zielfarbe 0 Block 0				Zielfarbe 1 Block 1				Zielfarbe 2 Block 2				Zielfarbe 3 Block 3			

Diese Skizze beantwortet auch die zweite, noch ausstehende Frage nach der Organisation der Paletten: Die Blockbildung entspricht exakt dem Palettenaufbau. Die Quellpalette (von der Größe 1 Block) wird sooft kopiert, wie es der Anzahl der Farben entspricht - dadurch entstehen die exakt gleichen Quellblöcke -, während die Zielpalette auf ihre volle Größe gespreizt wird. Dazu wird einfach jede Farbe einzeln verfielfacht, so daß die Blöcke in dieser Palette in sich homogen sind. Bei diesem Effekt ist es im übrigen nicht erforderlich, mehrere Bildschirmseiten zu verwenden oder auch vorberechnete Seiten zu erstellen und auf den Bildschirm zu kopieren, die Manipulation kann direkt am Bildschirm erfolgen. Dazu muß alleridings beachtet werden, daß sich das tatsächlich sichtbare Bild während der Modifikationen (also der Paletten-Generierung und des Mischens) nicht verändert, weil sonst Flimmern die Folge wäre. Wenn sich das Bild nicht verändern darf, sind zwei Punkte zu beachten: Zum einen darf die gerade aktive Palette nur in Bereichen verändert werden, deren Farben im Moment nicht auf dem Bildschirm sind, und zum zweiten muß bei Veränderung der Bilddaten die Palette bereits vorbereitet sein, damit die veränderten Punkte wieder auf ihrer ursprünglichen Farbe landen. Wenn also ein roter

Punkt, der z. B. die Farbe 3 hat, jetzt auf Farbe 23 gesetzt werden soll, muß sichergestellt sein, daß Farbe 0 bereits Rot enthält, damit sich am Bildschirm nichts verändert. Daraus folgt die Reihenfolge der Prozeduraufrufe, die beim Überblenden einzuhalten ist:

1. Paletten vorbereiten, dabei nur inaktive Farben verändern.
2. Bilddaten mischen, noch keine Veränderung sichtbar.
3. Paletten ineinander überblenden.

Aus der Forderung, daß nur inaktive Palettenbereiche zu verändern sind, folgt bereits die Einführung eines zusätzlichen Farbblocks. Dieser enthält zunächst die Quellfarben, die vom Quellbild verwandt werden. Da ein vierfarbiges Quellbild normalerweise die Farben 0 bis 3 belegt, befindet sich auch dieser neue Block an dieser Stelle. Um die Quellpalette zu generieren (Block in diesem Fall viermal kopieren), wird dieser Block als Ausgangspalette herangezogen und verfielfacht. Dabei wird der Block selbst nicht verändert, so daß sich am Bildschirm zunächst gar nichts tut. Dieser Block, er heißt übrigens **Reset-Block**, hat aber noch eine weitere wichtige Aufgabe zu erfüllen: Nach einer durchgeführten Blende hat das sichtbare Bild eine weit größere Palette als die Ursprungsbilder, nämlich genau deren Farbanzahl zum Quadrat. Möchte man nun in ein weiteres Bil

Farben auf Schwarz, dann ist nur der erste der zehn Pixel sichtbar, die übrigen haben Hintergrundfarbe. Verschiebt man die Palette jetzt um einen Punkt nach oben, so daß Farbe 1 Rot ist und Farbe 0 sowie Farben 2 bis 9 Schwarz sind, wird der zweite Pixel sichtbar, und der erste verschwindet. Verschiebt man die Palette nun immer weiter, wandert der Punkt auf dem Bildschirm von links nach rechts, ohne daß (nach der Initialisierung der zehn Punkte) auch nur ein Byte im Bildschirmspeicher verändert wurde. Wenn der rote Eintrag am oberen Ende (Farbe 9) angekommen ist, kann er natürlich wieder bei Farbe 0 angefügt werden, so daß eine zyklische Animation erfolgt, die den Punkt immer wieder von links nach rechts bewegt und wieder nach links springt. Genausogut ist es denkbar, bei Erreichen des Palettenendes die Bewegungsrichtung umzudrehen und den Punkt so hin- und herzubewegen. Für einen einzelnen Punkt ist diese Vorgehensweise offensichtlich noch nicht sehr sinnvoll, weil statt zwei Bytes im Bildschirmspeicher zu verändern (alten Punkt löschen, neuen setzen) gleich zehn Farbwerte neu gesetzt werden, und das auch noch über langsame Port-Zugriffe. Wesentlich effektiver ist die Palettenrotation bei großen und / oder kompliziert geformten Flächen. Sowohl eine hohe Anzahl Punkte als auch eine komplexe mathematische Beschreibung der zu bewegenden Punkte rechtfertigen die Zusammenfassung der Bewegungen in einer Palettenrotation. Möchte man nun größere Flächen bewegen, müssen diese natürlich entsprechen gestaltet werden. Die Flächen dürfen selbstverständlich nicht aus einer Palettenfarbe bestehen, sondern müssen in Bewegungsrichtung einen Farbverlauf aufweisen, damit praktisch zeilenweise bewegt werden kann. Anschaulich dargestellt sieht das dann aus wie in folgender Abbildung:

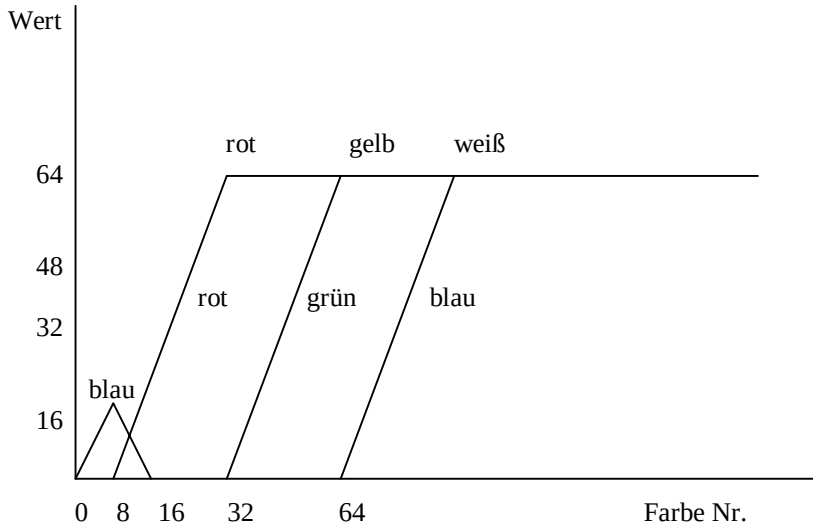
Farbe 0	rot	Farbe 0	rot
Farbe 1	rot	Farbe 1	blau
Farbe 2	blau	Farbe 2	blau
Farbe 3	blau	Farbe 3	rot
Farbe 0	rot	Farbe 0	rot

Hier werden Blöcke von der Höhe 2 Pixel nach oben durchgescrollt, wobei das Prinzip genauso auch für wesentlich höhere Blöcke und komplexere Strukturen gilt. Das Beispielbild enthält zweieinhalb Blöcke, die abwechselnd rot und blau gefärbt sind. Die Anzahl Blöcke spielt keine Rolle, das Bild kann also beliebig nach unten oder oben erweitert werden, solange die Reihenfolge der Farben eingehalten wird; auch Bruchteile von Blöcken sind möglich. Der Aufbau des Gesamtbildes ist periodisch, das heißt, nach zwei Blöcken wiederholt sich der Aufbau, daher sin zwei Blöcke mit einem zusammenhängenden (!) Farbverlauf von Farbe 0 bis 3 gefüllt. Anschließend werden in der Palette die Farben 0 und 1 auf Rot und die Farben 2 und 3 auf Blau gesetzt, so daß die Blöcke als solche zu erkennen sind. Diese Vorbereitungen erledigt man am besten mit einem Malprogramm[16], so daß sich das Programm nur noch um die Bewegung an sich kümmern muß. Diese Bewegung wird nun dadurch erzeugt, daß die (Teil-) Palette von Farbe 0 bis 3 um eine Position nach unten gescrollt wird. Farbe 0 erhält den Farbton aus Farbe 1, Farbe 1 den aus Farbe 2, Farbe 2 den aus Farbe 3 und Farbe 3 den aus Farbe 0, so daß eine Rotation entsteht. Wie in der Abbildung zu sehen, verschieben sich dadurch die roten und blauen Bereiche um eine Zeile nach oben, was ja auch Ziel der Aktion war. Nun ergibt auch dieses Scrolling noch keinen großen Sinn, weil es sich durch Verändern der Bildschirmstartadresse *(Linear Starting Address)* genausogut erreichen läßt. Interessanz wird das Gnaze, wenn die einfache Blockstruktur des Beispiels mit Effekten versehen wird. Baut man aus diesen Blöcken ein Schachbrett zusammen und kippt dieses mit einem guten Malprogramm (siehe Fußnote 16 !) nach hinten, erhält man mit der Palettenrotation eine nach hinten oder vorne scrollende Ebene. Dies konventionell über Pixelmanipulationen zu programmieren, scheitert unter Garantie am viel zu hohen Rechenaufwand. So überläßt man die Rechnerei dem Malprogramm und muß innerhalb des eigenen Programms nur noch einige Byte im Speicher verschieben und die Palette setzen. Ein weiterer Vorteil besteht darin, beliebige Objekte vor die Scrollebene zu plazieren, die normalerweise bei jedem Bildaufbau mit Hilfe einer (relativ langsamen) Sprite-Routine dorthin kopiert werden müßten, die bei dieser Technik jedoch statisch im Bild untergebracht sein können.

Feuer - Pyro-Effekt:

Ein loderndes Feuer auf dem Bildschirm zu sehen, ist spätestens nach den Unmengen an Rollenspielen nichts Besonderes mehr. Diese Flammen beruhen aber auf gezeichneten Bildern, die nacheinander abgespielt werden. Hier soll eine andere Möglichkeit aufgezeigt werden, die auch in vielen Demos Anwendung findet: Es wird direkt die Struktur der Flammen nachgeahmt. Dazu sind zwei wesentliche Punkte zu beachten: Ein Feuer befindet sich in einer ständigen Bewegung nach oben. Außerdem ist es im unteren Bereich weiß und verblaßt nach oben immer mehr zu roten Tönen. Es werden zwei Puffer benötigt, die den Bildschirmspeicher (bzw. seine untere Hälfte) im schnellen System-RAM wiederspiegeln. Das Hauptprogramm allokiert und löscht zunächst die beiden Puffer, schaltet den Mode 13h ein und bereitet die Palette auf die Flammen vor. Das Ergebnis veranschaulicht die Grafik:

[16] Professionelle Programmierer schreiben sich hier vielleicht vorher selbst ein solches, das dann optimal auf ihre Bedürfnisse abgestimmt ist. Eine nette Fingerübung, und unter Umständen durchaus eine reizvolle Aufgabe...



Pixel mit niedriger Farbnummer stellen später die kälteren Zonen der Flammen dar, sie befinden sich hier im linken Bereich. Liest man die Grafik von rechts nach links, also von hohen zu niedrigen Temperaturen, so sieht man folgendes: Zunächst stehen alle Farbkomponenten auf Maximum (= Weiß), ab Farbe 80 wird der Blauanteil langsam zurückgenommen, so daß die Farben etwas gelblicher erscheinen. Ab Farbe 56 wird zusätzlich der Grünanteil heruntergeregelt, so daß schließlich Rot übrigbleibt. Nachdem auch diese Farbe (ab Nummer 32) ausgefadet wurde, wäre normalerweise Schwarz das Ergebnis. In diesem Fall wird aber noch ein kleiner Schimmer Blau eingefügt (Farbe 16 bis 0), der über den roten Flammen steht. Wer´s nicht mag, entfernt diesen Teil einfach...

Voxel-Spacing:

Es ist ein beliebter Programmteil in vielen Demos und findet auch schon in einigen Spielen ausgiebige Anwendung: das Voxel Spacing. Dabei handelt es sich um eine imaginäre Landschaft mit Bergen und Tälern, über die sich der Betrachter bewegt. So allgemein diese Beschreibung ist, so viele Methoden existieren zur Darstellung dieses Effekts. Die Zahl der Algorithmen ist mindestens so hoch wie die Zahl der Programmierer, die sich bereits daran versuchten. Hier soll nun ein Verfahren gezeigt werden, das ohne komplexe dreidimensionale Abbildungen auskommt und sich statt dessen auf einfache zweidimensionale Zusammenhänge beschränkt. 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 einfaches 320 x 200-Bild mit möglichst fließenden Übergängen.[17] 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 - 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 angegebenen Koordinaten beziehen sich auf eine Projektion auf die untere Bildschirmhälfte des Mode X. 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, es passen 80 Pixel in eine Zeile (aus Geschwindigkeitsgründen diese Auflösung, siehe auch Fußnote 17), während des Zeichnens wird das Verhältnis immer weiter verringert, bis etwa eine 1 : 2-Abbildung erreicht ist und nur noch 40 Pixel in die vorderste Zeile passen. 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 wie

[17] Wer will, kann natürlich auch SVGA programmieren und verwendet dann am besten auch höhere Auflösungen. Profis können weiters eine Zoom-Möglichkeit einbauen, wobei aber spätestens dann **interpoliert** werden muß, sobald eine gewisse Nähe unterschritten wird. Experimentierfreudige können ihren Algorithmus dann soweit verfeinern, daß er für beliebige Bilder gute Resultate liefert. Zur Erinnerung: Voxel können hervorragend Rundungen darstellen, welche natürlich beliebiger (weiblicher ...) Art sein können...

gesagt die Farbe jedes Punktes herangezogen, indem eine entsprechende Anzahl Pixel übereinander angeordnet werden. Je größer der Farbwert, um so höher wächst also der Pixel zu einer senkrechten Linie. Durch diese vertikale Struktur werden außerdem Lücken vermieden, die durch den nach vorne wachsenden Zeilenabstand auftreten. Um die Szenerie möglichst real darzustellen, fehlen noch Wasserflächen. Dazu wird einfach jede Farbe unterhalb eines bestimmten Wertes auf eben diesen Wert gesetzt. Dadurch entstehen hier Ebenen, die keine vertikale Struktur aufweisen. Zur Erzeugung der Landschaften bedient man sich am einfachsten eines Fraktalgenerators, der Plasma-Wolken erzeugen kann.[18]

Sprites:

Ob als Raumgleiter im Action-Game oder Spielfigur im Jump-´n´-Run, überall begegnet man Sprites. Alles, was sich auf dem Bildschirm in irgendeiner Form anders bewegt als der Hintergrund, kann man im Prinzip als Sprite bezeichnen. Jeder weiß also, was damit gemeint ist. Es stellt sich allerdings die Frage: Wie programmiere ich so etwas?

Grundlagen:

Um selbst Sprites erzeugen zu können, muß man sich zunächst einmal etwas genauere Gedanken zu deren innerer Struktur machen. Sprites sind eigentlich nichts weiter als kleine (oder auch größere, je nach verlangter Rechenpower) Grafikausschnitte, die beliebig auf einem Hintergrundbild positioniert werden können. Sie müssen allerdings mindestens eine weitere Bedingung erfüllen: Es muß durchsichtige Stellen geben, an denen der Hintergrund durchscheint. Dies wird nicht nur bei “Löchern” mitten im Sprite benötigt, der weit häufigere Anwendungsfall sind die Randbereiche eines Sprites. Jedes Sprite, das nicht hundertprozentige Rechtecksstruktur aufweist, muß am Rand durchsichtige Bereiche besitzen. Bei der Darstellung kann auf Kreisformen oder ähnliches aus Geschwindigkeitsgründen keine Rücksicht genommen werden, es werden grundsätzlich rechteckige Bereiche auf den Bildschirm gebracht, daher muß der Rand aufgefüllt werden, sonst würden, unabhängig von der Erscheinungsform, immer gefüllte Rechtecke erscheinen. Während Home-Computer dabei dem Programmierer praktisch die gesamte Arbeit abnehmen und man nur noch die Position berechnen muß, ist die Aufgabe beim PC weit komplizierter. Hier spielt sich der gesamte Ablauf im Bildschirmspeicher des VGA ab, man muß also sehr optimiert programmieren. Die Softwarelösung des Pcs hat aber auch unschätzbare Vorteile: Die Gestaltung liegt völlig in der Hand des Programmierers, jede beliebige Sprite-Größe ist denkbar, wenn es auch irgendwann an der Rechenzeit scheitert. Auch Skalierungen oder Rotationen sind möglich, hier kann man sich also austoben - wenn man das kann. Zunächst ist die grundsätzliche Vorgehensweise bei der Sprite-Darstellung zu beachten:

1. Löschen der Sprites durch Kopieren des Hintergrundbildes auf die aktuelle Bildschirmseite
2. Bewegung und Darstellung der Sprites
3. Umschalten auf die fertige Seite

Bevor die Sprites in ihrer neuen Position dargestellt werden können, müssen die alten erst einmal vom Bildschirm verschwinden. Dazu eine ganze Bildschirmseite zu kopieren, mag überflüssig erscheinen. Könnte man nicht genausogut vor der Darstellung jedes Sprites dessen Hintergrund sichern und hinterher wieder zurückschreiben? Diese Vorgehensweise macht allerdings nur bei einem oder maximal zwei Sprites Sinn, weil der Verwaltungsaufwand enorm ist. Erstens kommt nur eine Hintergrundsicherung im VGA-RAM in Frage, weil allein hier ein schneller Zugriff möglich ist. Das RAM ist allerdings sehr begrenzt, irgendwann ist hier kein Platz mehr. Und zweitens übersteigt der nötige Verwaltungsaufwand bei weitem den Aufwand des Kopierens einer Bildschirmseite. Schließlich handelt es sich um Bereiche mitten im Bildschirm, die nur zeilenweise kopiert werden können, ein einf9Tmhirm

belassen und von dort immer auf die aktuelle Seite kopieren. Dies hat jedoch zwei entscheidende Nachteile. Der schnelle Write-Mode 1 basiert auf ganzen Viererblocks (ein Byte aus allen vier Planes), so daß es nicht möglich ist, ein Sprite z. B. von x-Koordinate 5 (Plane 1) nach 6 (Plane 2) zu kopieren, Daten können nur innerhalb der Planes kopiert werden. Die einzige Möglichkeit, diese Problem zu umgehen, ist, jedes Sprite viermal im Speicher zu halten, einmal an x-Position 0, einmal an x-Position 1 usw., was die Anzahl verfügbarer Sprites doch sehr einschränkt. Der zweite Nachteil dieses Konzepts besteht in der Ausmaskierung der durchsichtigen Punkte, was nur über separate, im Hauptspeicher liegende Masken möglich ist, die aber erst noch erzeugt werden müssen.

Sprites lesen und schreiben:

Die Alternative besteht im zweiten Konzept, das auch wir hier verwenden wollen: Die Sprite-Daten verbleiben im Hauptspeicher und werden Punkt für Punkt auf den Bildschirm gebracht. Der große Nachteil dieses Konzepts besteht in der relativ geringen Geschwindigkeit, so daß man sich schon einige Gedanken zur Optimierung seiner Routinen machen muß. Dies fängt bereits beim Format der Sprite-Daten im Hauptspeicher an. Die Grafikdaten werden beim Setzen eines Sprites auf die vier Planes verteilt, Punkte an durch vier teilbaren Adressen landen in Plane 0, mit Rest 1 teilbare in Plane 1 usw. Die einfachste Möglichkeit, die Daten auf den Bildschirm zu bringen, besteht daher darin, die Punkte in der "normalen" Reihenfolge zu setzen und ständig die Schreib-Plane umzuschalten. Daß dies nicht gerade die schnellste Lösung ist, wissen wir bereits. Deshalb beschreiten wir einen anderen Weg: zunächst alle Daten einer Plane kopieren, dann erst umschalten und die nächsten Daten kopieren. Daraus resultiert auch das Sprite-Format: Die Daten stehen hier nicht im Mode-13h-Format (lineare Aneinanderreihung der Pixel), sondern in einer dem Mode X angelehnten Reihenfolge. Zuerst kommen die kompletten Daten der ersten Plane, dann die der zweiten usw. Beim Schreiben der Sprites kann man dann ohne weiteren Verwaltungsaufwand auf diese Daten zugreifen, es muß lediglich die Startadresse in das SI-Register[19] geladen, dann die erste Plane kopiert und direkt im Anschluß die nächste Plane bearbeitet werden. Eine Neupositionierung des SI-Registers ist nicht erforderlich. Die Daten selbst enthalten direkt die Farbwerte, können also ohne Umwege in den Bildschirmspeicher kopiert werden. Lediglich die Farbe 0 spielt eine gesonderte Rolle: Sie steht für den Hintergrund, an diesen Stellen ist das Sprite also durchsichtig. Ein Problem tritt noch bei der Berechnung der Breite des Sprite auf, denn diese ist nicht in jeder Plane gleich. Man denke sich nur einmal ein fünf Pixel breites Sprite, das an x-Position 0 geschrieben werden soll. In diesem Fall müssen in Plane 0 2 Byte kopiert werden (Pixel 0 und 4), während die anderen nur ein Byte enthalten. Bei unseren Algorithmen ist diese Verteilung jedoch global, also nicht von der x-Koordinate der Zielposition abhängig. Wird das Sprite z. B. an die x-Position 1 kopiert, befindet sich der 2 Byte große Block zwar in der physikalischen Plane 1. Weil jedoch die Kopierroutine mit dieser Plane beginnt, handelt es sich aus Sicht der Sprite-Daten immer noch um deren Plane 0, die weiterhin zwei Byte breit ist. Mit anderen Worten: Allein aus der Breite des Sprites läßt sich ein Array mit den Breiten der einzelnen Planes erstellen, das in der Kopierschleife die Zählvariable versorgt. Die erste Plane der Sprite-Daten ist im genannten Beispiel immer 2 Byte breit, die anderen 1 Byte, unabhängig von der aktuellen Position. Beim Setzen muß die Plane, mit der begonnen wird, also nur zu dem Zweck berechnet werden, sie dem VGA mitzuteilen.

Clipping:

Neben der Berücksichtigung der 0 in den Sprite-Daten weist das Setzen von Sprites noch ein weiteres geschwindigkeitshemmendes Problem auf: das Clipping. Es läßt sich in kaum einem Programm vermeiden, daß ein Sprite den Bildschirmrand berührt und dahinter verschwindet. Kopiert man nach der bisherigen Strategie trotzdem blind die Daten in den Bildschirmspeicher, erhält man recht konfuse Ergebnisse: Beim Überschreiten des rechten oder unteren Bildschirmrands wird der Rest des Sprites in die nächste Zeile bzw. Bildschirmseite kopiert, am oberen und linken Rand wird entsprechend in die vorherige Zeile bzw. Seite geschrieben. Keiner dieser Fälle ist im Normalfall wünschenswert, daher muß ein Algorithmus gefunden werden, der das Clipping, das Abschneiden der "überstehenden" Daten erreicht. Die einfachste, aber bei weitem langsamste Möglichkeit besteht darin, vor dem Setzen jedes Punktes zunächst einmal zu überprüfen, ob der Punkt innerhalb des Bildschirm liegt. Weit geschickter ist es allerdings, noch vor der eigentlichen Kopierschleife die Rahmenbedingungen in der Weise zu verändern, daß eine Darstellung auf die gewohnte Weise möglich ist. Es wird also versucht, das Clipping weitgehend aus der Schleife herauszuhalten, denn je weiter außen ein Befehl innerhalb einer Struktur verschachtelter Schleifen steht, desto seltener wird er durchlaufen und desto weniger Rechenzeit benötigt er insgesamt. Im Prinzip ist das Clipping ganz einfach: Die Breite bzw. Höhe wird angeglichen und das Sprite dargestellt. Am rechten und unteren Rand muß jedoch noch eine Variable eingeführt werden, die dafür sorgt, daß der nicht dargestellte Bereich auch in den Quelldaten übersprungen wird. Wird eine Sprite-Zeile nur halb kopiert, muß dafür gesorgt werden, daß nach dieser Zeile der Quellzeiger dennoch auf den Anfang der nächsten Zeile zeigt. Dies gilt parallel dazu auch am unteren Rand, hier ist es der Abstand zu den Daten der nächsten Plane. Am oberen und linken Rand kommt noch ein Problem hinzu: Die Koordinate, an der mit der Darstellung begonnen wird, muß noch auf 0 gesetzt werden, damit innerhalb des tatsächlichen

[19] Zu den Prozessorregistern empfehlen wir PC intern (siehe auch Textende) als Nachschlagewerk.

Bildschirms gezeichnet wird. Dabei verändert sich am linken Rand allerdings auch die Start-Plane und der Start-Offset der ersten zu kopierenden Spalte. Bewegt sich ein Sprite z. B. um einen Punkt über den linken Rand, befindet sich die erste Spalte an der Koordinate -1, darf also nicht gezeichnet werden. Die nächste Spalte der ersten Sprite-Plane befindet sich nun an der x-Koordinate 3, so daß dieser Wert angeglichen werden muß. Allgemein muß beim Clipping am linken und rechten Rand auch wieder die Verteilung auf die verschiedenen Planes berücksichtigt werden, wobei ein ähnliches Prinzip wie bei der Breitenberechnung angewandt wird, jetzt allerdings von der Anzahl der überstehenden Pixel ausgehend. Um die Sache nicht unnötig zu verkomplizieren, kann ferner davon ausgegangen werden, daß ein Clipping am linken Bildrand ein Clipping am rechten ausschließt, dazu würde ein Sprite mit einer Breite größer 321 benötigt, das wohl in der Praxis sehr selten auftauchen wird, so daß man sich hier die Kombination linker Rand / rechter Rand sparen kann.

Skalieren:

Wenn eine Spielerfigur in einem Adventure nach hinten läuft, muß sie wegen des Fluchtpunktes kleiner werden, damit ein dreidimensionales Bild entsteht. Mit den bisherigen Mitteln gibt es dazu lediglich eine Möglichkeit: Für jede Zwischenstufe wird ein eigenes Sprite definiert, das mit einem Malprogramm auf die entsprechende Größe gebracht wurde. Dies ist die schnellste Methode, die allerdings auch bei weitem den meisten Speicher verbraucht. In vielen Situationen ist es daher sinnvoll, diese Skalierung automatisch durch das Programm zu erzeugen und nur mit einem Original-Sprite zu arbeiten. Zu dieser Problematik gibt es allgemeine mathematische Algorithmen, die jeden Punkt einzeln an seine neue Position projizieren. Daß dies sehr langsam ist, braucht wohl nicht weiter betont zu werden. Viel einfacher ist ein anderes Verfahren, dessen Ergebnis exakt dem mathematischen entspricht, das aber auf einem anderen Grundgedanken beruht. Exemplarisch wird hier die Skalierung in y-Richtung erläutert, die im Beispiel ein drehendes Logo erzeugt. Eine x-Skalierung funktioniert prinzipiell genauso, so daß der Einfachheit halber nicht näher darauf eingegangen wird. Möchte man ein Sprite um einen Stammbruch-Faktor skalieren, z. B. um ½, stellt man einfach von jeweils zwei Zeilen nur eine dar, überspringt also jede zweite. Bei einer Sprite-Drittelung wird analog von jeweils drei Zeilen nur eine verwendet. Dieses Verfahren läßt sich auf beliebige Brüche erweitern, wobei im folgenden wie auch im Programm selbst mit Prozent gerechnet wird. Es geht immer darum, festzustellen, nach wie vielen dargestellten Zeilen wieder eine Zeile ausgelassen werden muß. Dazu versieht man die y-Koordinate mit einem Nachkommateil, der bei der Darstellung auf eine ganzzahlige Koordinate gerundet wird. Dieses Verfahren erscheint logisch, wenn man den Bildschirmaufbau aus einem mathematischen Blickwinkel betrachtet, bei dem es nicht nur ganzzahlige Zeilen gibt, sondern beliebig viele Zwischenwerte, die bei der Skalierung verwendet werden. Weil der Bildschirm aber leider eine beschränkte Anzahl Zeilen hat, müssen diese Zwischenzeilen auf die tatsächlich vorhandenen abgebildet werden. Soweit zur Theorie, doch wie funktioniert das in der Praxis? Hier kommt die Festkommarechnung (Addition und Subtraktion) zum Einsatz, weil sie den einzig schenllen Ansatz bietet. Die relative y-Koordinate innerhalb des Sprites wird aufgeteilt in einen Vorkommateil, der alleridngs nicht explizit in einer Variablen geführt wird, und einen Nachkommateil. Nun wird nach jeder Zeile eine Konstante auf die Nachkommastelle addiert. Überschreitet diese den Wert 100, was einem Überlauf in die Vorkommastelle entspricht, wird die Konsequenz gezogen: Der Vorkommateil wird erhöht, was sich daran zeigt, daß die folgende Zeile übersprungen wird. Außerdem wird die Nachkommastelle wieder um 100 vermindert. Die Konstante folgt der Devise: Je größer der Skalierungsfaktor (je näher am Original), desto kleiner, weil dann seltener eine Zeile herausgenommen wird. Sie wird somit einfach als 100-Skalierungsfaktor berechnet. Ein Beispiel: Ein Sprite wird auf 60 % skaliert, die Additionskonstante beträgt also 40. Nach der ersten Zeile ist der Nachkommateil 40, nach der zweiten 80. In beiden Fällen geschieht nichts weiter, aber nach der dritten Zeile überschreitet der Nachkommateil mit 120 die Grenze von 100. Er wird auf 20 zurückgesetzt und die nächste Zeile ausgelassen, nach weiteren zwei Zeilen wiederholt sich die Auslassung mit einem Nachkommateil von 100, und das Spiel beginnt mit 0 von vorn. Das Auslassen selbst läßt sich auf zwei Arten gestalten, einmal kann man die Zeile in den Quelldaten überspringen, zum anderen kann man einfach den Zielzeiger stehenlassen und die nächste Zeile über die alte schreiben. Wir ziehen die letzte Möglichkeit vor: Bei dieser wird jede Zeile dargestellt, wenn auch nicht alle sichtbar sind. Daher ist die Aufbaugeschwindigkeit immer gleich, unabhängig vom Skalierungsgrad. Außerdem erleichtert diese Vorgehensweise die Erzeugung gespiegelter Sprites.

3D-Vektorgrafik:

Alle bisher gezeigten Effekte haben eines gemeinsam: Sie arbeiten in nur zwei Dimensionen. Möchte man wirklichkeitsnähere Bilder erzeugen, kommt man um die dritte Dimension - die Tiefe - nicht herum. Dabei stellt sich natürlich das Problem der Darstellung auf dem nach wie vor zweidimensionalen Monitor, der eine Tiefeninformation nur durch etliche Tricks wie Fluchtpunktperspektive und Lichtquellenschattierung glaubhaft machen kann. Diese Tricks erfordern ein nicht unerhebliches Maß an Mathematik, genauer gesagt an Geometrie. Transformationen, Abbildungen, Helligkeitsabstufungen etc. sind gerade in der Masse, die eine eindrucksvolle Präsentation erfordert, sehr komplexe Berechnungen. Diese müssen nach mathematischen und programmiertechnischen Aspekten optimiert werden, damit eine vernünftige Geschwindigkeit erreicht wird. Die

Sprache dieser Prozeduren ist daher Assembler, weil eine Hochsprache grundsätzlich zu komplizierte und damit zu langsame Strukturen erzeugt.[20]

3D-Körper in 2D darstellen:

Solange es noch keine erschwinglichen dreidimensionalen Ausgabegeräte gibt, bleibt für jegliche 3D-Berechnung die Hürde der zweidimensionalen Darstellung auf dem flachen Bildschirm. Für diese gibt es verschiedene Methoden, die von der einfachen Parallelprojektion bis zu komplexen Raytracing-Algorithmen reichen. Letztere Methode ist jedoch wegen des gewaltigen Rechenaufwands den Renderern[21] vorbehalten, die sich für ein Bild auch einmal ein paar Minuten Rechenzeit nehmen können. Die einfachste Methode, die Tiefeninformation umzurechnen, die der Monitor ja nicht darstellen kann, ist, sie komplett zu ignorieren. Dabei werden di

eindeutige Beziehung zwischen y, y´, a (Abstand Auge - Bildschirm) und z (Tiefe des Objekts) herstellt. Man kann sich leicht klarmachen, 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 (z. B. 128) einsetzt. In diesem Fall läßt sich die Multiplikation durch Shiften (SHL / SHR) erheblich beschleunigen.

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. Ein rotierender Würfel mit Bildern auf den Seitenflächen ist sicher interessanter als ein statisches Bild desselben, das man viel einfacher mit einem Malprogramm erzeugen kann. Eine Bewegung setzt sich im wesentlichen aus zwei Arten zusammen: Translation und Rotation. Die Skalierung kann man unter Umständen auch 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 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 ggf. 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:

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 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 auch in diesem Text zur Anwendung kommen wird. Ausführliche mathematische Herleitungen dieses Algorithmus findet man in jedem Grafikbuch, vielen allgemeinen PC-Büchern und sporadisch auch in Zeitschriften. Daher verzichten wir an dieser Stelle darauf und beschränken uns auf das grundsätzliche Funktionsprinzip. 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. Die Entscheidung darüber, welcher der beiden Punkte der nächste ist, ist dem bereits gezeigten Festkomma-Verfahren ähnlich. 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.

Glaskörper:

Die bisher gezeigten 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 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. Hat zum Beispiel Fläche A die Farbe 2 (Bit 1 gesetzt) und Fläche B die Farbe 16 (Bit 4), so kommt als Ergebnis der Verknüpfung die Farbe 18 (Bit 1 und 4 gesetzt) heraus. 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 genannte Farbe 18 muß in diesem Fall eine Mischung aus Farbe 2 und 16 sein. 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

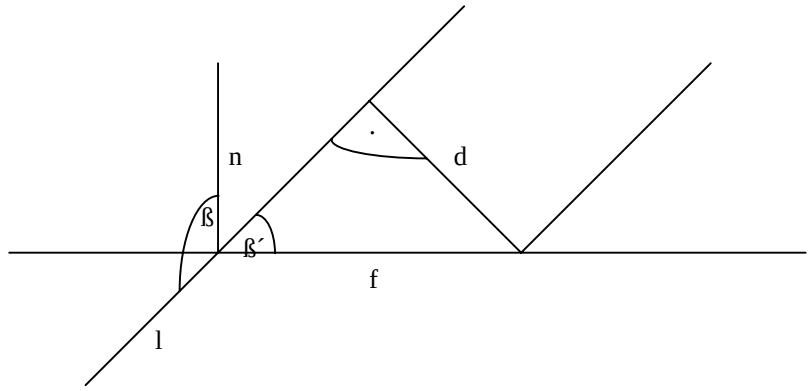
der bei weitem schnellere. Im wesentlichen baut die hier beschriebene Methode auf dem Zeichnen von Linien auf. Dazu wird, ausgehend vom Punkt mit der niedrigsten y-Koordinate, der linke und rechte Rand des Polygons abgetastet, bis der Punkt mit der größten y-Koordinate ereeicht ist. Dabei werden die 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.

Hidden Lines:

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 si 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 nch 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.

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. Natürlich ist auf heutigen PCs noch kein Echtzeit-**Raytracing** möglich, 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:



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

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.

Texturen:

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. Diese Technik findet zur Zeit viel Anwendung in Rollenspielen wie Ultima Underworld, wo die Wände mal aus Stein, mal aus Holz oder anderen Materialien bestehen. 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 Um