

# Mario Zechner



# Vorwort

Beinahe 8 Jahre ist es nun schon her, dass ich meinen damals von meinen Eltern erworbenen 486er mit der ersten BASIC-Programmzeile gequält habe. Das Ergebnis dieser Anstrengung war ein einfaches Hallo Welt -Programm, welches nichts anderes tat, als eben diesen Satz im Textmodus des DOS auszugeben.

Grund für diese Bemühungen war mein Interesse an Computerspielen. Es war für mich bis zu diesem Zeitpunkt ein Rätsel, wie diese Spiele und vor allem deren Grafiken entstehen. Auf die ersten textbasierenden Programme folgten nach monatelangem Selbststudium der Programmiersprache BASIC bald einfachste Grafikanwendungen, die wenig spektakulär und nützlich waren. Anhand von Quellcodes verschiedener Grafikdemos von Freunden aus dem Internet entwickelte ich allmählich ein Verständnis für die Prinzipien der Spielgrafikprogrammierung. Grundsätzlich war mein Lernprozess von der Trial and Error -Methode geprägt, was mein PC meist mit Abstürzen belohnte. Literatur in Form von Büchern war nicht erhältlich, darum musste ich auf verschiedene Tutorien im Internet zurückgreifen, die jedoch zum größten Teil auf Englisch und meist auch noch in einem so kryptischen Stil geschrieben waren, dass ich mich dazu entschloss, selbst herauszufinden was hinter all diesen Quellcodes verborgen liegt. An dieser Stelle möchte ich mich bei meinem Freund Victor bedanken, der mir all die Jahre zur Seite stand und mir nützliche Tipps und Tricks zu allerlei Problemen bot.

Mit dieser Fachbereichsarbeit möchte ich versuchen, mein angehäuften Wissen zu ordnen und dieses Einstiegern schriftlich zur Verfügung zu stellen. Alle Kapitel bauen aufeinander auf und sollten deshalb im Zusammenhang gelesen werden (was nicht bedeutet, dass Fortgeschrittene nicht auch nur einzelne Kapitel als Wissensquelle verwenden können). Als Programmiersprache habe ich BASIC gewählt. Das mag in Zeiten von Windows XP vielleicht etwas seltsam anmuten, aus eigener Erfahrung weiß ich aber, dass der Einstieg durch diese Wahl um einiges erleichtert wird. Alle hier vorgestellten Methoden haben bis heute Gültigkeit und können somit auf andere Systeme transponiert werden. Denn Beweis dafür liefert das Beispielprogramm VBRAY.VBP, welches die Ray-Casting-Methode mit Hilfe von Visual Basic und DirectX in einer Windowsumgebung umsetzt.

Das größte Problem, das sich mir während dieser Arbeit stellte, war die Verwendung von Quellen. Wie bereits erwähnt, ist es unglaublich schwierig diesbezüglich nützliches Material zu bekommen. In meinem Fall war das auch nicht nötig, da ich mir dieses Wissen ja bereits erworben hatte, und zwar nicht durch das Lesen von Büchern, sondern durch das Sezieren von Quellcodes anderer Personen sowie der Trial and Error -Methode. Hinzu kam noch die enge Verbundenheit dieses Gebietes mit der Mathematik, die ebenfalls, so glaube ich zumindest, eine meiner Stärken ist. Das Gebiet der Spielgrafikprogrammierung ist ein offenes Feld, wo jeder seine Gedanken und Ideen so umsetzen kann wie es ihm beliebt. Es gibt keine Grenzen und damit auch keine Konventionen. Trotzdem haben sich einige Prinzipien im Laufe der Jahre durchgesetzt, zumeist aus Gründen der Kompatibilität.

Als Beweis für die Richtigkeit des Inhalts dieser Fachbereichsarbeit habe ich aber trotzdem Belegstellen angeführt, sowie von mir entwickelte funktionsfähige Beispielprogramme beigelegt. Die Belegstellen beinhalten meistens auch weiterführende Gedanken, die ich im Rahmen dieser Fachbereichsarbeit aus Platzgründen nicht mehr aufgreifen konnte.

Da diese Fachbereichsarbeit gleichzeitig auch als Tutorium dienen soll, habe ich es mir erlaubt den Leser direkt anzusprechen, was sich hoffentlich nicht als störend auswirkt. Anstatt eines Anhangs habe ich eine CD-ROM beigelegt, die alle Beispielprogramme enthält. Diese können Sie über ein selbststartendes Menü bequem ausführen.

Damit möchte ich Sie nicht mehr länger aufhalten und hoffe, dass ich mit den folgenden Seiten Ihr Interesse wecken kann.

## **INHALTSVERZEICHNIS**

<b>1) DER BILDSCHIRM.....</b>	<b>2</b>
<b>2) DER VIDEOMODUS 13H .....</b>	<b>5</b>
2.1) SPEICHERORGANISATION DES MODUS 13H.....	5
2.2) DIE PALETTE – FARBEN IM MODUS 13H .....	6
2.3) AKTIVIEREN DES MODUS 13H.....	9
<b>3) FUNKTIONEN ZUM ÄNDERN DES BILDSCHIRMINHALTS.....</b>	<b>10</b>
3.1) DAS LÖSCHEN DES BILDSCHIRMS.....	10
3.2) DRAWING PRIMITIVES .....	10
3.2.1) Das Bildschirmkoordinatensystem .....	11
3.2.2) Das Zeichnen von Pixel .....	12
3.2.3) Das Zeichnen von Linien .....	14
3.2.4) Das Zeichnen von Dreiecken .....	18
3.2.5) Das Zeichnen ausgefüllter Dreiecke.....	19
3.4) BLITTING .....	25
3.4.1) Transparentes Blitting.....	30
3.4.2) Die Getpicture-Routine .....	31
3.4.3) Dimensionierung und Übergabe von Bild-Arrays an die Funktionen Blit und Getpicture in BASIC .....	33
3.5) DOUBLE BUFFERING.....	34
<b>4) GRAFIK-ENGINES .....</b>	<b>36</b>
4.1) 2D-ENGINES .....	36
4.1.1) Programmierung von Sprites .....	36
4.1.2) Scrolling .....	43
4.2) 3D-ENGINES .....	48
4.2.1) Ray-Casting .....	48
<b>ARBEITSPROTOKOLL .....</b>	<b>60</b>
<b>VERZEICHNIS DER BELEGSTELLEN:.....</b>	<b>62</b>

## 1) Der Bildschirm<sup>1</sup>

Der Bildschirm funktioniert im Grunde wie ein herkömmlicher Fernseher. Die für den Benutzer sichtbare Bildfläche besteht aus einer Vielzahl kleiner Punkte, die zeilenweise aneinandergereiht sind. Jeder dieser Punkte, auch Pixel (picture element = Bildelement) genannt, ist aus drei Phosphorteilchen mit den Farben Rot, Grün und Blau zusammengesetzt. Drei Elektronenstrahlen tasten den Bildschirm pixel- bzw. zeilenweise ab, wobei jeder Elektronenstrahl eines der Phosphorteilchen des abzutastenden Pixel zum Leuchten bringt. Indem man die Intensität jedes einzelnen Strahls variiert, erhält der Pixel die gewünschte Farbe. Dabei gelten die Regeln der additiven Farbmischung. So ergeben Grün und Blau Gelb und so weiter. Mehr über Farben erfahren sie im Kapitel „Die Palette – Farben im Videomodus 13h“. Die Signale für jeden Elektronenstrahl erhält der Monitor von einer speziellen Einheit der Grafikkarte, dem Catode-ray-tube-controller (CRTC). Dieser liest für jeden Pixel die Farbe aus dem Videospeicher und zerlegt sie in die einzelnen RGB-Werte und somit in die nötigen Intensitäten für die einzelnen Elektronenstrahlen. Der CRTC steuert auch das Timing der Elektronenstrahlen, ist also mitbestimmend bei der Bildwiederholfrequenz. Die Elektronenstrahlen tasten den Bildschirm zeilenweise von links nach rechts ab. Haben sie das Ende einer Zeile erreicht, werden sie an den Anfang der nächsten Zeile zurückgesetzt. Dieser Vorgang wird als horizontaler Rücklauf (horizontal Retrace) bezeichnet. Analog dazu gibt es den vertikalen Rücklauf. Dabei springen die Elektronenstrahlen vom letzten Pixel in der untersten Zeile rechts, zum ersten Pixel im linken oberen Eck.

Erst wenn die Elektronenstrahlen alle Zeilen durchlaufen haben, kann ein

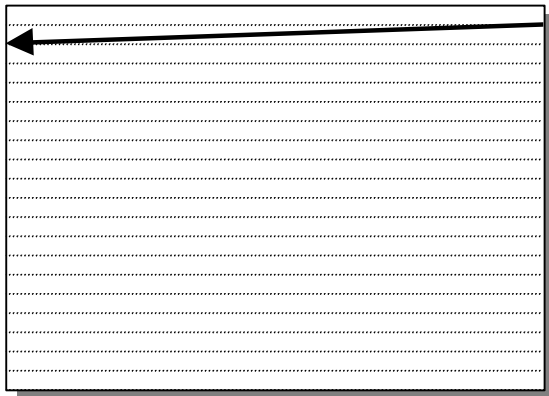


Abb.1 horizontaler Rücklauf

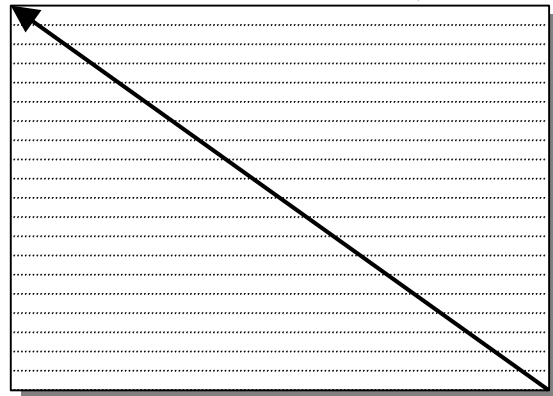


Abb.2 vertikaler Rücklauf

neues Bild aus dem Videospeicher gelesen werden. Diese Tatsache wird uns im Kapitel „Double Buffering“ noch einmal begegnen. Der Monitor hat nämlich im Durchschnitt eine Bildwiederholfrequenz von ca. 70 Herz. Der Bildschirm wird also 70 mal pro Sekunde neu aufgebaut. Die CPU bzw. die Grafikchips können aber pro Sekunde um einiges mehr an Bildern in den Videospeicher schreiben. Wird der Videospeicher schneller beschrieben, als der Bildschirm aufgebaut wird, kommt es zu unerwünschtem Flackern. Mit Hilfe des Double Buffering kann das jedoch verhindert werden.

---

<sup>1</sup> Vgl.: Michael Tischer, *PC intern* 4. Düsseldorf: DATA BECKER 1994.

Die heute erhältlichen Monitore sind im Vergleich zu ihren Urahnen wahnsinnig leistungsfähig. Dabei sollte aber nie außer acht gelassen werden, dass die Entwicklung der Bildschirme immer Hand in Hand mit der Entwicklung der Grafikkarten geht. Beide Komponenten müssen zwei Größen handhaben können: Die Auflösung und die Farbtiefe. Unter der Auflösung versteht man die Anzahl der am Bildschirm dargestellten Pixel. Dabei werden immer zwei Werte angegeben, die Anzahl der Pixel in einer Zeile sowie die Anzahl der Pixel in einer Spalte.

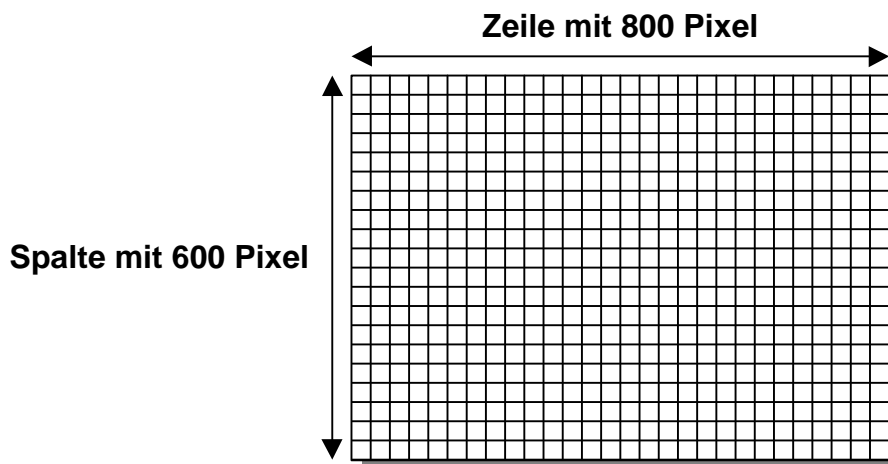


Abb.3 horizontale und vertikale Auflösung in Pixel

Multipliziert man diese beiden Werte, so erhält man die Gesamtanzahl der verwendeten Pixel. Die Farbtiefe gibt die höchstmögliche Anzahl der gleichzeitig darstellbaren Farben an. Die Angabe erfolgt meist in Bit. Der Grund dafür liegt in der Darstellung der Farbe eines Pixel im Videospeicher. Die gängigsten Farbtiefen sind 8-bit (256 Farben), 16-bit und 32-bit. Auflösung und Farbtiefe definieren zusammen einen Videomodus. Wer unter Windows die Auflösung ändert, wechselt gleichzeitig auch den Videomodus. Diese Fachbereichsarbeit behandelt im speziellen nur den Videomodus 13h (Modus 13h) der seit nunmehr 10 Jahren Verwendung findet. Dieser ist am leichtesten zu bearbeiten und funktioniert auf allen Grafikkarten. Außerdem kann er über einen Interruptaufruf initialisiert werden, was einen Teil der Programmierarbeit wegfallen lässt. Alle anderen Videomodi, die über den Modus 13h hinausgehen, sind sehr schwierig zu programmieren. Für sie gibt es eigene Code-Bibliotheken (DirectX, Allegro oder OpenGL), die Befehlssätze zum Arbeiten mit diesen bereitstellen. Hier noch eine kurze Liste der üblichen Grafikmodi:

- 320x200; 8-bit ( Modus 13h)
- 640x480; 8-bit
- 800x600; 8-bit
- 640x480; 16-bit
- 800x600; 16-bit
- 1024x748; 16-bit
- 640x480; 32-bit
- 800x600; 32-bit
- 1024x748; 32-bit

Diese Liste beinhaltet nur einen Bruchteil der möglichen Grafikmodi. Es wäre einfach unmöglich, auf alle diese Modi einzugehen, darum beschränke ich mich auf den Modus 13h, der zwar etwas antiquiert ist, zum Vermitteln der Grundlagen der Grafikprogrammierung aber aufgrund seiner Einfachheit ideal geeignet ist.

## 2) Der Videomodus 13h

### 2.1) Speicherorganisation des Modus 13h<sup>2</sup>

Wie bereits im vorhergehenden Kapitel erwähnt, bietet der Modus 13h eine Auflösung von 320x200 Pixel bei einer Farbtiefe von 8-bit. Das am Bildschirm dargestellte Bild wird aus dem Videospeicher der Grafikkarte gelesen. Für den Grafikprogrammierer ist es unerlässlich zu wissen, wie der Videospeicher im zu programmierenden Videomodus organisiert ist, denn er ändert das dargestellte Bild, indem er die im Videospeicher abgelegten Farbwerte der Pixel manipuliert. Die Organisation des Modus 13h ist sehr simpel. Beginnend beim linken oberen Pixel werden alle Farbwerte hintereinander im Videospeicher abgelegt. Bei der gegebenen Farbtiefe von 8-bit verbraucht jedes Pixel 1 Byte im Videospeicher. Die Adressen 0 bis 319 im Videospeicher werden von den Farbwerten der Pixel der ersten Zeile belegt, die nächste Zeile wird an den Adressen 320 bis 639 abgebildet und so weiter. Die einzelnen Zeilen werden also im Speicher aneinander gereiht:

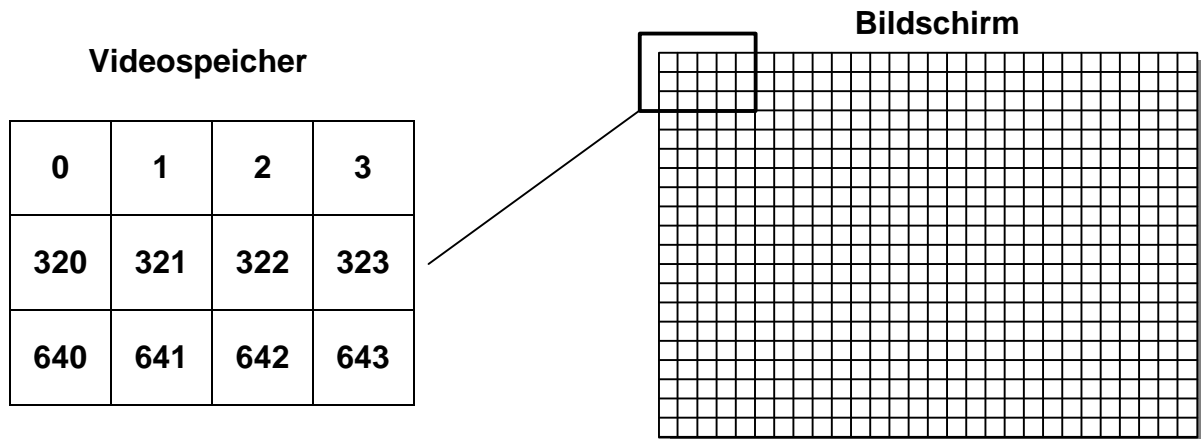


Abb.4 Die Zahlen in den Zellen des Videospeichers repräsentieren die Adressen der korrespondierenden Pixel am Bildschirm

Die Adresse des letzten Pixel im Videospeicher ergibt sich aus der Gesamtanzahl der dargestellten Pixel minus 1, da die Adressierung bei der Adresse 0 beginnt. Insgesamt werden für den Modus 13h 320x200 Byte = 64000 Byte benötigt. Da sich der Videospeicher auf der Grafikkarte befindet, stellt sich natürlich die Frage wie man auf diesen Speicher zugreifen kann, denn im Real-Mode ist nur das erste Megabyte des normalen RAM adressierbar. Dieses Problem wurde elegant gelöst: Anstatt über die Ports der Grafikkarte auf den Videospeicher zuzugreifen, wird der Videospeicher in das Segment mit der Adresse A000hex im konventionellen RAM gespiegelt. Dieses Segment dient als Fenster zum Videospeicher und kann gleich bearbeitet werden wie der restliche RAM. Alle Änderungen in diesem Segment haben einen unmittelbaren Effekt auf den tatsächlichen Videospeicher:

<sup>2</sup> Vgl.: <http://www.comprenica.com/atrevda/atrtut07.html>  
<http://www.maths.tcd.ie/~nryan/demos/vgaprog.html>  
<http://www.gamedev.net/reference/articles/article315.asp>



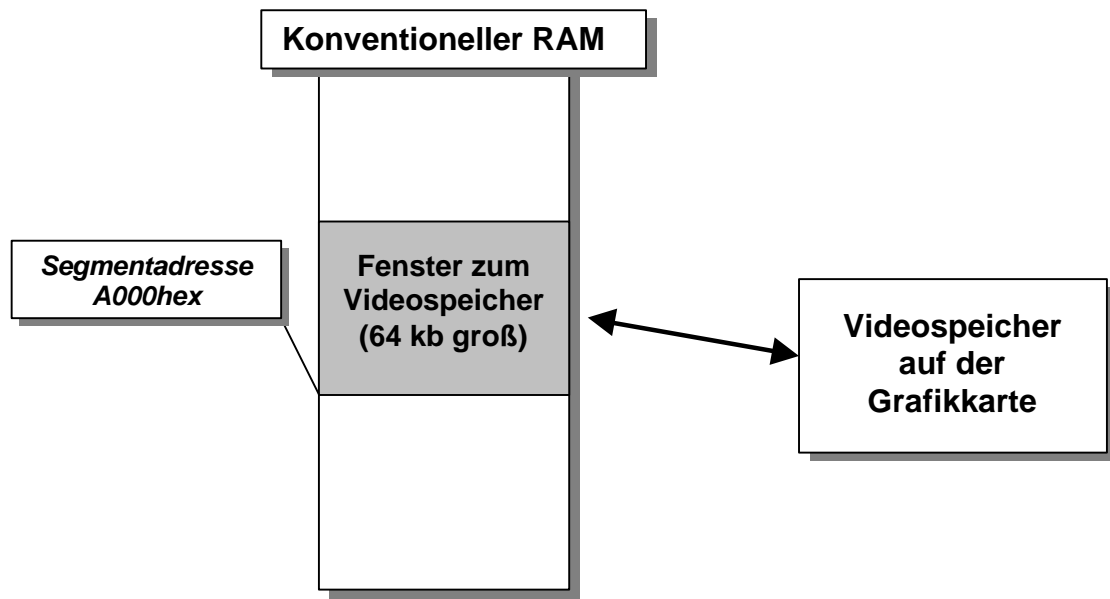


Abb.5 Wird ein Byte im Segment an der Adresse A000hex geändert, hat das auch einen Effekt auf den eigentlichen Videospeicher auf der Grafikkarte.

Es besteht ein Zusammenhang zwischen Auflösung, Farbtiefe und benötigtem Videospeicher. Im Modus 13h braucht man exakt 64000 Byte Speicher. Diese Zahl kann über folgende Formel ermittelt werden:

Benötigte Byte = Anzahl der Pixel \* Farbtiefe in Byte  
Benötigte Byte im Modus 13h = 320 \* 200 \* 1 = 64000 Byte

Im Modus 13h entspricht der benötigte Speicher für eine virtuelle Bildschirmseite (der Speicher, in dem der Bildschirminhalt abgebildet ist) relativ genau einem üblichem 64-Kilobyte-Segment im Real-Mode. Wäre die Auflösung oder die Farbtiefe des Modus 13h größer, so würde die Bildschirmseite nicht mehr in einem Real-Mode-Segment Platz finden. Man könnte aber z.B. durchaus eine Bildschirmseite für einen Modus mit 640x200 Pixel Auflösung bei einer Farbtiefe von 4-bit (16 Farben) in einem Segment unterbringen, nur sind die Abstriche bei der Farbtiefe ein ausreichendes Argument, um sich gegen diesen Modus zu entscheiden. Das beste Verhältnis zwischen Auflösung und Farbtiefe bietet im Real-Mode der Modus 13h, der die gegebenen Speicherbegrenzungen am besten ausnutzt.

### 2.2) Die Palette – Farben im Modus 13h<sup>3</sup>

Ich sprach vorher davon, dass die Farbwerte der Pixel im Videospeicher abgelegt werden. Das stimmt so nicht ganz. Vielmehr sind diese Werte Indizes auf Einträge in einer sogenannten Palette, die die eigentlichen Farbwerte enthält. Im ersten Kapitel haben wir erfahren, dass sich die

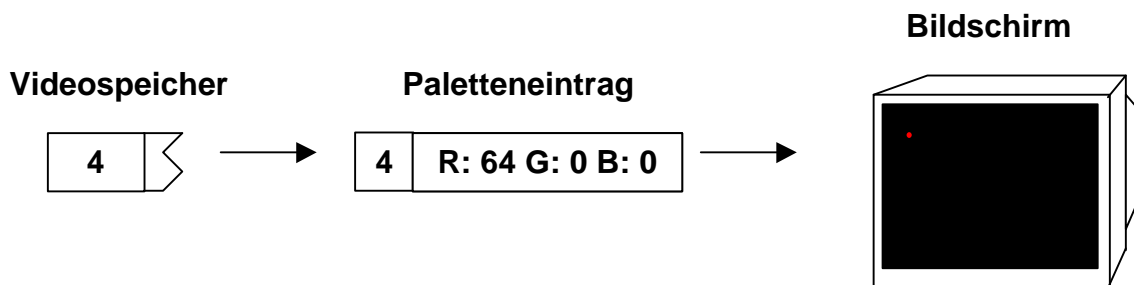
<sup>3</sup> Vgl.: <http://qbtuts.qb45.com/graphics/palette.txt>  
<http://www.gamedev.net/reference/articles/article317.asp>  
<http://www.gamedev.net/reference/articles/article348.asp>  
<http://www.qbtuts.qb45.com/graphics/qbtmpalette.htm>

Beispielprogramm: „PALFADE.BAS“



Farben eines Pixel aus der Intensität seiner einzelnen Phosphorteilchen ergibt. Die Intensität eines solchen Teilchens wird in der Palette als ganzzahliger Wert gespeichert. Der Wertebereich der Intensitäten hängt vom verwendeten Videomodus ab. Im Modus 13h erstreckt sich dieser von 0 bis 63 und zwar für alle drei Komponenten (Rot, Grün, Blau) der Farbe. Intern werden diese Farben als 16-bit Werte abgespeichert: Die ersten fünf Bit geben die Intensität der Blaukomponente an, die nächsten fünf Bit die Intensität der Grünkomponente und die darauffolgenden fünf Bit die Intensität der Rot-Komponente. Das sechzehnte Bit bleibt ungenützt. Haben alle drei Komponenten den Wert 0, so ergibt das die Farbe Schwarz. Haben alle drei Komponenten den Wert 63, ergibt das die Farbe Weiss.

Es gelten die Regeln der additiven Farbmischung. Insgesamt können über dieses System  $64^3 = 262144$  Farben erzeugt werden. Die Palette können Sie sich wie die Palette eines Malers vorstellen, die 256 Plätze für Farben zur Verfügung stellt. Der Maler kann die drei Farben Rot, Grün und Blau mischen wie er will, er kann aber nur 256 Farben gleichzeitig verwenden. Jeder Index, der in den Videospeicher geschrieben wird, zeigt auf einen der 256 Paletteneinträge. Dort liest die Grafikkarte dann die Intensitäten der 3 Komponenten aus und schickt sie an den Monitor, der das Pixel dann in der



*Abb.6 Die Farbindeindex des Pixel zeigt auf den Paletteneintrag für seine Farbe. Der Paletteneintrag wird ausgelesen und an den Monitor geschickt, der Pixel erscheint in der gewünschten Farbe.*

gewünschten Farbe zeichnet:

Nachdem der Modus 13h aktiviert worden ist, haben die Paletteneinträge bereits vordefinierte Werte. Diese Werte decken ein großes Spektrum an Farben ab. Es kann aber durchaus sein, dass Sie die Paletteneinträge ändern müssen. Wenn Sie mit einem Zeichenprogramm z.B. eine Bitmap-Datei mit einer 8-Bit Farbtiefe erzeugt haben, so entsprechen die voreingestellten Werte der Palette nicht den Werten die das Bild für die richtige Farbdarstellung benötigt. Zum Ändern oder Lesen der Paletteneinträge stellt uns die Grafikkarte zwei Ports zur Verfügung. Diese Ports können Sie in allen gängigen Programmiersprachen ansprechen. Ich beschränke mich auf die BASIC-Variante. BASIC bietet die Möglichkeit, Paletteneinträge über den Befehl PALETTE zu manipulieren. Es ist jedoch schneller und unkomplizierter, die Einträge durch direkten Zugriff auf die Hardware zu verändern. Über die Befehle INP und OUT können sie einem I/O Port einen Wert zuweisen bzw. diesen auslesen. Die beiden Befehle haben folgende Syntax:

```
OUT Port, Wert  
INP (Port)
```

Port steht für eine Adresse, auf die sowohl die Grafikkarte, als auch das Programm zugreifen kann. Ein Port ist also eine Schnittstelle zwischen der Hardware und den Programmen. Wollen Sie eine Farbe der Modus 13h Palette in ihren RGB-Werten ändern, müssen Sie die Ports 3C8hex und 3C9hex ansteuern. An die Adresse 3C8hex müssen Sie zuerst den Index der Farbe schreiben. Danach übergeben Sie die RGB-Werte hintereinander an die Adresse 3C9hex. Die folgenden BASIC Befehle weisen dem Paletteneintrag mit dem Index 43 die Farbe Weiss zu. Alle Pixel, die im Videospeicher mit dem Wert 43 dargestellt werden, erscheinen nach Aufruf der Befehle Weiss:

```
OUT &H3C8, 43          ;Paletteneintrag wählen
OUT &H3C9, 63          ;Rotwert übergeben
OUT &H3C9, 63          ;Grünwert übergeben
OUT &H3C9, 63          ;Blauwert übergeben
```

Möchten Sie wissen, welche RGB-Werte ein Paletteneintrag enthält, so ist dafür das Port 3C7hex zuständig. Sie gehen dabei ähnlich vor wie beim Ändern der RGB-Werte. Zuerst teilen Sie der Grafikkarte mit, welchen Paletteneintrag Sie auslesen wollen. Danach lesen Sie die einzelnen Werte nacheinander vom Port 3C9hex ein:

```
OUT &H3C7, 43          ;Paletteneintrag wählen
R = INP( &H3C9 )      ;Rotwert einlesen
G = INP( &H3C9 )      ;Grünwert einlesen
B = INP( &H3C9 )      ;Blauwert einlesen
```

Liest man den Rotanteil ein, so wird das der Grafikkarte mitgeteilt. Darauf stellt sie den Grünwert bereit. Dasselbe geschieht, nachdem man den Grünanteil ausgelesen hat. Wurde der Blauanteil vom Port geholt, ist der Prozess abgeschlossen und die Grafikkarte ist bereit für die nächste Palettenmanipulation.

Hier noch alle vordefinierten Farben mit ihren Indizes:



Abb.7 Die vordefinierten Farben der Palette und deren Indizes

Verwenden Sie die voreingestellten Farben, so schreiben Sie z.B. den Wert 4 an die Adresse eines Pixel, der die Farbe Rot haben soll, für die Farbe Blau den Wert 1 etc. Übrigens verwendet BASIC die selbe Palette in diesem Modus. Wollen Sie also einen gelben Pixel mit dem Befehl PSET erzeugen, so geben Sie den Wert 14 als Farbe an.

Wir werden vorrangig die vordefinierte Palette verwenden. Nötige Änderungen an dieser werden explizit angegeben.

### **2.3) Aktivieren des Modus 13h**

Um im Modus 13h arbeiten zu können, müssen Sie diesen erst aktivieren. Dafür bieten Ihnen sowohl BASIC als auch das BIOS geeignete Befehle. Programmieren Sie mit BASIC, können Sie den Befehl SCREEN verwenden. Dieser Befehl ruft dann seinerseits den BIOS Interrupt 10hex auf, der die notwendigen Änderungen an den Grafikkartenregistern vornimmt. Um den Modus 13h mit BASIC aufzurufen, verwenden Sie folgenden Befehl:

```
SCREEN 13
```

Nach Aufruf des Befehls befinden Sie sich im Modus 13h. Alle BASIC-Grafikbefehle wie PSET, LINE oder CLS werden nun auf diesen Modus angewendet. Sie können natürlich auch direkt den Interrupt 10hex aufrufen. Übergeben Sie nur dem Register AL den Wert 13hex und rufen dann den Interrupt auf:

```
mov al, 13h  
int 10h
```

Sämtliche Codebeispiele dieser Fachbereichsarbeit setzen voraus, dass zuvor der Modus 13h initialisiert wurde. Die Initialisierung wird nicht mit angegeben.

### 3) Funktionen zum Ändern des Bildschirminhalts

#### 3.1) Das Löschen des Bildschirms<sup>4</sup>

Beim Löschen des Bildschirms werden sämtliche Pixel auf ein und die selbe Farbe gesetzt. Dabei erhalten alle Byte im Videospeicher den selben Farbindex. BASIC bietet dafür die Funktion CLS ( clear screen = Bildschirm löschen ). Diese setzt alle Byte im Videospeicher auf den Farbindex 0, was bei der Standartpalette der Farbe Schwarz entspricht. Ein Manko besitzt diese Funktion jedoch: Wollen Sie den Bildschirm mit einer anderen Farbe löschen, müssen Sie den Paletteneintrag mit dem Index 0 ändern. Da dies zu umständlich ist, wollen wir eine allgemeinere Lösung dieses Problems erarbeiten. Im Grunde müssen wir den Befehl CLS nur um einen Parameter erweitern, über den die gewünschte Farbe angegeben wird. Der Programmcode hinter dieser Funktion gestaltet sich zwar äußerst simpel, der Vollständigkeit halber aber hier die BASIC-Variante:

```
SUB ClearScreen (farbe as Integer)
DEF SEG = &HA000
    FOR offset& = 0 TO 63999
        POKE offset&, farbe
    Next offset&
DEF SEG
END SUB
```

Zuerst wird über den Befehl DEF SEG das zu beschreibende Segment definiert. Es folgt eine Schleife, in der inkrementell jedes Byte im Videospeicher die an die Funktion ClearScreen übergebene Farbe erhält.

Eine 32-bit Maschinensprachenversion wäre natürlich um einiges schneller. Prinzipiell sollten die meisten Grafikroutinen auf Maschinensprachenebene geschrieben sein, da diese oft in zeitkritischen Situationen verwendet werden. Eine Ausnahme macht hier die Programmiersprache C, deren Compiler zum Großteil noch effizienteren Maschinensprachencode erzeugen als das per Hand möglich wäre.

#### 3.2) Drawing Primitives

Der Begriff „Drawing Primitives“ umfasst sämtliche Funktionen zum Zeichnen von geometrischen Grundformen wie Punkten, Linien, Kreisen oder auch Dreiecken. Alle Funktionen bauen auf dem Algorithmus zum Setzen eines Pixel auf. In diesem Zusammenhang von einem Algorithmus zu sprechen ist meines Erachtens nicht ganz gerechtfertigt, da es sich dabei nur um die Berechnung einer Adresse im Videospeicher handelt. Wir werden diese Formel im nächsten Kapitel herleiten. Zuvor müssen wir uns aber noch Gedanken um ein mögliches Koordinatensystem am Bildschirm machen.

---

<sup>4</sup> Beispielprogramm: „CLEARSCR.BAS“

### 3.2.1) Das Bildschirmkoordinatensystem

Um die Position von Figuren am Bildschirm angeben zu können, muss man sich zuerst ein Koordinatensystem zurechtlegen. Eine Möglichkeit wäre z.B. das Polarkoordinatensystem, aufgrund der spalten- und zeilenweisen Anordnung der Pixel am Bildschirm ist das kartesische Koordinatensystem aber weitaus naheliegender. Das von uns verwendete System unterliegt jedoch einigen Beschränkungen. Sehen wir uns die Eigenschaften unseres Koordinatensystems an.

- Den Ursprung des Koordinatensystems stellt der Pixel in der linken oberen Ecke des Bildschirm dar. Er besitzt die Koordinaten (0;0).
- Die positive x-Achse zeigt wie gewohnt nach rechts.
- Da der Ursprung sich im linken oberen Eck des Bildschirms befindet, verläuft die positive y-Achse nach unten.
- Es können nur ganzzahlige Koordinaten verwendet werden, da uns nur eine endliche Anzahl an Pixel zur Verfügung steht. (320 Pixel horizontal, 200 Pixel vertikal)
- Daraus ergibt sich wiederum, dass nur ganzzahlige Koordinaten im Intervall [0,319] auf der x-Achse bzw. im Intervall [0,199] auf der y-Achse Sinn ergeben.

Vor allem der dritte Punkt verwirrt die meisten Grafikprogrammierer. Es dauert aber nicht lange, bis man sich an diese Gegebenheit gewöhnt hat.

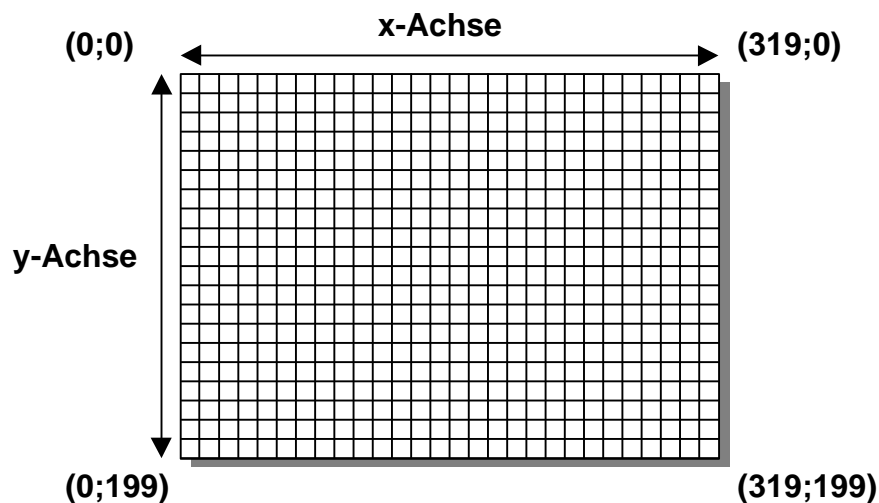


Abb.8 Das Bildschirmkoordinatensystem

Zur Veranschaulichung hier eine kleine Grafik:

Zahlen mit Nachkommastellen werden auf die nächste integere Zahl auf- oder abgerundet. Alle Pixel, die außerhalb der Intervalle liegen, werden nicht gezeichnet. Das Überprüfen, ob die Koordinaten eines zu zeichnenden Pixel innerhalb der definierten Grenzen des Bildschirms liegen, nennt man Clipping. Ich werde im nächsten Kapitel näher darauf eingehen.

### 3.2.2) Das Zeichnen von Pixel<sup>5</sup>

Die Funktion zum Zeichnen eines Pixel ist die rudimentärste unter allen Grafikroutinen. Sämtliche Funktionen, die über diese hinaus gehen, bauen auf ihr auf.

Unser Ziel ist es, die Koordinaten eines Pixel in seine Adresse im Videospeicher umzurechnen und an diese Adresse dann den gewünschten Farbwert zu schreiben. Um das Problem der Umrechnung lösen zu können, betrachten wir zuerst den in Zeilen und Spalten angeordneten Videospeicher mit darübergelegtem Koordinatensystem.

		x-Achse															
		0	1	2	3	...	...	...	...	...	...	...	316	317	318	319	
y-Achse	0	0	1	2	3	...	...	...	...	...	...	...	316	317	318	319	
	1	320	321	322	323	...	...	...	...	...	...	...	636	637	638	639	
	2	640	641	642	643	...	...	...	...	...	...	...	956	957	958	959	
	3	960	961	962	963	...	...	...	...	...	...	...	1276	1277	1278	1279	
	4	1280	1281	1282	1283	...	...	...	...	...	...	...	1596	1597	1598	1599	
	5	1600	1601	1602	1603	...	...	...	...	...	...	...	1916	1917	1918	1919	
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
	196	62720	62721	62722	62723	...	...	...	...	...	...	...	63036	63037	63038	63039	
	197	63040	63041	63042	63043	...	...	...	...	...	...	...	63356	63357	63358	63359	
	198	63360	63361	63362	63363	...	...	...	...	...	...	...	63676	63677	63678	63679	
	199	63680	63681	63682	63683	...	...	...	...	...	...	...	63996	63997	63998	63999	

Abb.9 Der Videospeicher im Bildschirmkoordinatensystem. Innerhalb der Speicherzellen stehen deren Adressen.

Wie wir bereits wissen, stellen die Adressen 0 bis 319 die erste Zeile am Bildschirm dar, d.h. alle Pixel mit den Koordinaten (0;0) bis (0;319). Die Adresse eines Pixel entspricht also seiner x-Koordinate. Für Pixel, deren y-Koordinate größer als null ist, gilt das nicht. Wie können deren Koordinaten berechnet werden? Betrachten wir das Pixel an den Koordinaten (2;1). Es hat die Adresse 322 im Videospeicher. Das Pixel (2;2) hat die Adresse 642. Man erkennt, dass man um von einer Zeile in die nächste zu wechseln, zur Adresse den Wert 320, also die Länge einer Zeile in Byte, addieren muss. Allgemein gilt also die Formel:

$$\text{Adresse im Videospeicher} = 320 * y + x$$

Wir errechnen zuerst die Adresse des ersten Byte einer Zeile (320\*y) und addieren dann die x-Koordinate des Pixel dazu. Dies gilt auch für die erste Zeile:

<sup>5</sup> Beispielprogramm: „DRAWPIX.BAS“

Adresse von Pixel(0;5) =  $320 * 0 + 5 = 5$

Das bedeutet, die x-Koordinate dient als Offset innerhalb einer Zeile. Mit Hilfe dieser Gleichung können wir nun ein Pixel am Bildschirm setzen. Hier die BASIC-Variante:

```
SUB DrawPixel (x, y, Farbe)
DEF SEG = &HA000
    POKE 320*y+x, Farbe
DEF SEG
END SUB
```

Diese Routine kann und sollte noch optimiert werden. Die zeitaufwendigste Operation innerhalb dieser Prozedur ist die Multiplikation mit 320. Diese kann man auch durch zwei Shifts nach links ersetzen ( $320 * y = 256 * y + 64 * y = y \text{ SHL } 8 + y \text{ SHL } 6$ ). Leider bietet uns BASIC diese Möglichkeit nicht. Darum hier die Assembler Variante:

```
public DrawPixel
DrawPixel proc x:word, y:word, farbe:byte
    Mov di, y                ;di mit y laden
    Mov ax, di               ;ax mit y laden
    Shl di, 6                ;di = 64 * y
    Shl ax, 8                ;ax = 256 * y
    Add di, ax               ;di = 320 * y
    Add di, x                ;di = 320 * y + x
    Mov ax, a000h            ;es auf Videospeicher-
    Mov es, ax               ;segment setzen
    Mov byte ptr es:[di], farbe ;Pixel setzen
    Ret
DrawPixel endp
```

Damit besitzen wir eine sehr leistungsfähige Routine. Wir müssen uns jetzt nur noch um das vorher angesprochene Clipping kümmern. Clipping soll verhindern, dass Pixel, die außerhalb des Bildschirms liegen, gezeichnet werden. Würde man dies unterlassen, gäbe es falsche Ergebnisse bei der Adressenberechnung, was sogar zu einem Absturz führen kann. Das kann aber leicht durch einfache If-Anweisungen verhindert werden. Diese werden am Anfang der Routine durchlaufen und brechen diese notfalls ab. Die BASIC-Version:

```
IF x > 319 OR x < 0 THEN EXIT SUB
IF y > 199 OR y < 0 THEN EXIT SUB
```

Haben Sie diese Anweisungen in Ihre Routine geschrieben, brauchen Sie sich bei anderen Grafikroutinen, die auf dieser aufbauen, nicht mehr um das Clipping kümmern.

Anstatt ein Pixel zu setzen, kann man über die Formel zur Berechnung seines Offsets auch dessen Farbe in Erfahrung bringen. Dazu wird der



Schreibzugriff auf das Byte einfach zu einem Lesezugriff geändert. In BASIC wird dafür der Befehl PEEK verwendet.

### 3.2.3) Das Zeichnen von Linien<sup>6</sup>

Mathematisch gesehen ist eine Linie ein Teil einer Geraden, der durch einen Start- und einen Endpunkt abgegrenzt wird. Alle Punkte zwischen diesen beiden Punkten sind sowohl Teil der Linie als auch Teil der Geraden. Eine Gerade wird mathematisch durch die Gleichung  $y = k \cdot x + d$  dargestellt. Mit ihrer Hilfe können alle Punkte einer Geraden errechnet werden. Da die Linie Teil der Geraden ist, erhält man auch alle Punkte der Linie über diese Gleichung. An den Algorithmus werden folgende Forderungen gestellt:

- Er muss alle ganzzahligen Koordinaten der Pixel innerhalb des Intervalls [Startpunkt; Endpunkt] berechnen können.
- Er muss diese Pixel in der gewünschten Farbe zeichnen können.
- Wenn nötig soll er diese auch clippen können.

Die letzten beiden Punkte stellen keine Herausforderung dar, da wir zu diesem Zweck die im letzten Kapitel vorgestellte DrawPixel-Routine verwenden können. Sie kann die errechneten Pixel zeichnen und nötigenfalls auch clippen. Den aufwendigsten Teil des Algorithmus macht also die Berechnung der Punkte der Linie aus. Wie können wir diese Problemstellung lösen?

Da wir nur ganzzahlige Koordinaten am Bildschirm darstellen können, müssen wir nur die y-Koordinaten aller ganzzahligen x-Werte im Intervall [Startpunkt<sub>x</sub>, Endpunkt<sub>x</sub>] unter Verwendung der Geradengleichung berechnen. Natürlich muss man zuvor noch die Werte k und d errechnen. Die Steigung k erhalten Sie über folgende Gleichung:

$$k = (\text{Endpunkt}_y - \text{Startpunkt}_y) / (\text{Endpunkt}_x - \text{Startpunkt}_x)$$

Die Variable d wird errechnet, indem wir die Geradengleichung umformen und die Steigung k sowie die Koordinaten eines Punktes (Start- oder Endpunkt) einsetzen:

$$d = y - k \cdot x = \text{Startpunkt}_y - k \cdot \text{Startpunkt}_x$$

Damit besitzen wir alle Voraussetzungen, um die Punkte der Linie berechnen zu können. Dies geschieht innerhalb einer Schleife. Wir beginnen damit, den Startpunkt zu zeichnen. Die Berechnung seiner Koordinaten ist überflüssig, da wir diese ja über die an die Funktion übergebenen Parameter bereits besitzen. Der nächste zu berechnende Punkt besitzt die x-Koordinate Startpunkt<sub>x</sub> + 1. Wir setzen diese in die Geradengleichung ein, erhalten die gesuchte y-Koordinate und zeichnen dieses Pixel. Da die y-Koordinate sicherlich kein ganzzahliger Wert ist, muss dieser noch gerundet werden. Wir müssen uns darum aber nicht kümmern, da dies bei der Übergabe der Koordinaten an die DrawPixel-Routine von BASIC übernommen wird. Nachdem der Bildpunkt gezeichnet wurde, wird das nächste Pixel errechnet.

---

<sup>6</sup> Vgl.: <http://www.comprenica.com/atrevida/atrtut08.html>  
Beispielprogramm: „DRAWLINE.BAS“

Dazu addieren wir 1 zur x-Koordinate des letzten Punktes und führen wieder die Berechnung der y-Koordinate, sowie das Zeichnen des neuen Punktes durch. Die Schleife wird solange durchlaufen, bis der Endpunkt der Linie erreicht ist. In BASIC könnte das so aussehen:

```
SUB DrawLine (Startx, Starty, Endx, Endy, Farbe)
k = (Endy-Starty)/(Endx-Startx)
d = Starty - k * Startx
x = Startx
y = Starty
While x <= Endx
    x = x + 1
    y = k * x + d
    DrawPixel(x ,y ,Farbe)
WEND
END SUB
```

Wir berechnen die x-Koordinate inkrementell, indem wir zur x-Koordinate des letzten Punktes 1 addieren. Kann man die y-Koordinate ebenfalls inkrementell errechnen, und wenn ja, wie erhält man die nötige Schrittweite?

Die Antwort auf diese Frage liefert uns die folgende Abbildung:

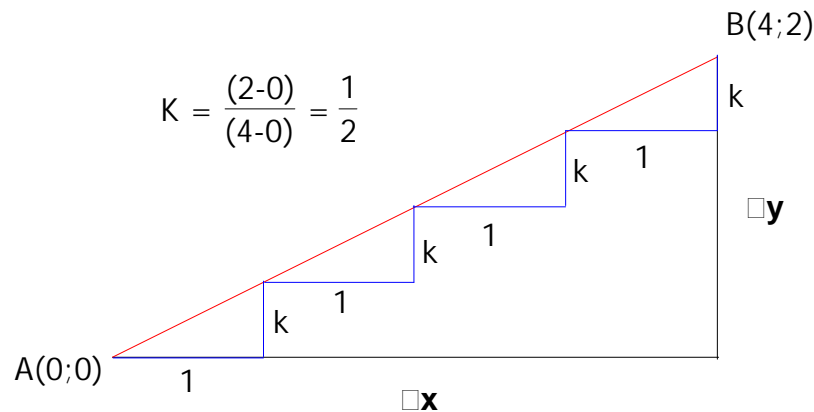


Abb.10 Linie mit Steigungsdreiecken

Durch die Steigungsdreiecke wird ersichtlich, dass die Steigung k das äquivalente Inkrement zur x-Schrittweite 1 ist. Das heißt, wir können auf die Berechnung von d verzichten und ersetzen die in der Schleife verwendete Geradengleichung durch eine Addition des letzten y-Werts mit der y-Schrittweite k. Unsere optimierte Version des Algorithmus lautet wie folgt:

```
SUB DrawLine (Startx, Starty, Endx, Endy, Farbe)
k = (Endy-Starty)/(Endx-Startx)
x = Startx
y = Starty
While x <= Endx
    x = x + 1
    y = y + k
    DrawPixel(x ,y ,Farbe)
WEND
END SUB
```

```
    DrawPixel(x ,y ,Farbe)
WEND
END SUB
```

Die Laufvariablen  $x$  und  $y$  für die Pixel-Koordinaten werden wie gehabt mit den Werten des Startpunktes initialisiert. Die  $y$ -Koordinate wird nun aber nicht mehr über die Geradengleichung sondern mit Hilfe des  $y$ -Inkrement  $k$  errechnet, d.h.:

Aktuelles Pixel  $X = (x + 1; y + k)$

Dadurch ersparen wir uns zwei Multiplikationen von  $k \cdot x$  außerhalb und noch wichtiger innerhalb der Schleife, was einen Geschwindigkeitszuwachs bringt.

Bisher wurde angenommen, dass wir die Linie von links nach rechts zeichnen, Startpunkt<sub>x</sub> also kleiner als Endpunkt<sub>x</sub> und damit das  $x$ -Inkrement 1 ist. Was aber, wenn an die Funktion Parameter für den umgekehrten Fall übergeben werden? Man löst dieses Problem, indem man am Anfang der Funktion die Parameter überprüft. Ist Startpunkt<sub>x</sub> größer als Endpunkt<sub>x</sub>, so werden die beiden Koordinatenpaare miteinander vertauscht. Dadurch gewährleisten wir, dass alle Linien von links nach rechts gezeichnet werden, das  $x$ -Inkrement also immer 1 ist:

```
IF startx > endx THEN SWAP startx, endx: SWAP starty, endy
```

Leider gibt es noch ein Problem um das man sich kümmern muss:

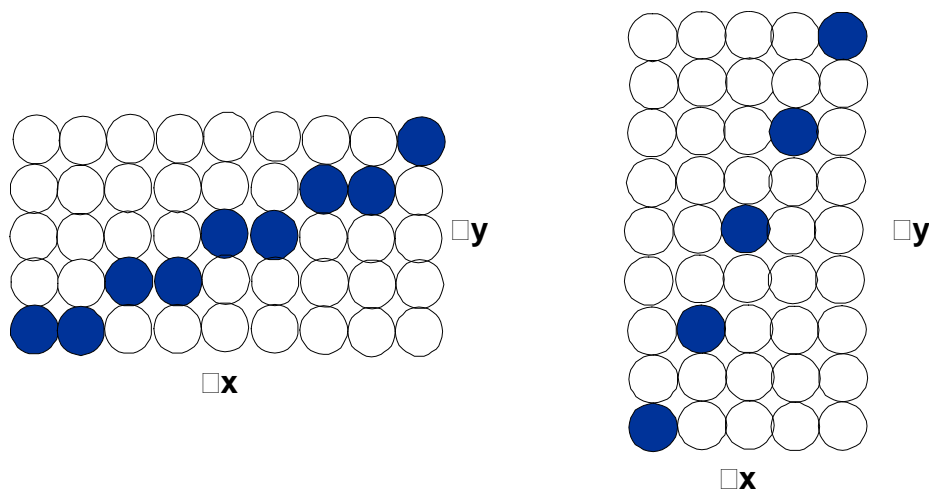


Abb.11 Zwei Linien am Bildschirm, gezeichnet mit dem bisherigen Linien-Algorithmus.

Ist  $\Delta x$  größer als  $\Delta y$ , so existiert für jedes ganzzahlige  $x$  genau eine  $y$ -Koordinate. Im umgekehrten Fall gibt es für jede  $x$  mehrere  $y$ -Koordinaten. Unser bisheriger Algorithmus berechnet aber immer nur eine  $y$ -Koordinate pro  $x$ -Koordinate, was dazu führt, dass die restlichen Pixel mit dieser  $x$ -Koordinate nicht gezeichnet werden.

Um nun auch den Fall  $\Delta y > \Delta x$  handhaben zu können, müssen wir unseren Algorithmus erweitern. Ist  $\Delta x > \Delta y$ , können wir den bisher verwendeten Algorithmus verwenden. Ist  $\Delta y > \Delta x$ , dann zeichnen wir die Linie nicht von

links nach rechts, sondern von unten nach oben (Vergessen sie bei dieser Gelegenheit nicht, dass die positive y-Achse am Bildschirm nach unten zeigt). Das heißt konkret, dass nun das y-Inkrement den Wert 1 und das x-Inkrement die Steigung k besitzt. Wir müssen diese Steigung jetzt aber anders berechnen:

$$K = \Delta x / \Delta y = (\text{Endpunkt}_x - \text{Startpunkt}_x) / (\text{Endpunkt}_y - \text{Startpunkt}_y)$$

Weiters wird nun die Schleife solange durchlaufen, bis die Variable y der Variablen Endy entspricht. Wir müssen auch wieder gewährleisten, dass die Linie von unten nach oben gezeichnet wird, das y-Inkrement also immer den Wert 1 besitzt. Dies erreichen wir wieder, indem wir die Koordinatenpaare falls nötig vertauschen. Somit können wir nun den endgültigen Algorithmus formulieren:

```
SUB DrawLine (Startx, Starty, Endx, Endy, Farbe)
IF ABS(endx - startx) > ABS(endy - starty) THEN
  IF startx > endx THEN SWAP startx, endx: SWAP starty, endy
  k = (endy - starty) / (endx - startx)
  x = startx
  y = starty
  DrawPixel (x, y, Farbe)
  WHILE x <= endx
    x = x + 1
    y = y + k
    DrawPixel (x, y, Farbe)
  WEND
ELSE
  IF starty > endy THEN SWAP startx, endx: SWAP starty, endy
  k = (endx - startx) / (endy - starty)
  x = startx
  y = starty
  DrawPixel (x, y, Farbe)
  WHILE y <= endy
    x = x + k
    y = y + 1
    DrawPixel (x, y, Farbe)
  WEND
END IF
END SUB
```

Um das Clipping brauchen wir uns in diesem Fall nicht zu kümmern, da diese Aufgabe die DrawPixel-Routine übernimmt.

### 3.2.4) Das Zeichnen von Dreiecken<sup>7</sup>

Dreiecke gehören zur Klasse der Polygone und sind deren einfachste Form. Ein Polygon ist eine Fläche, die über mindestens 3 Punkte definiert wird. Dabei grenzen Linien zwischen den einzelnen Punkten die Fläche ab.

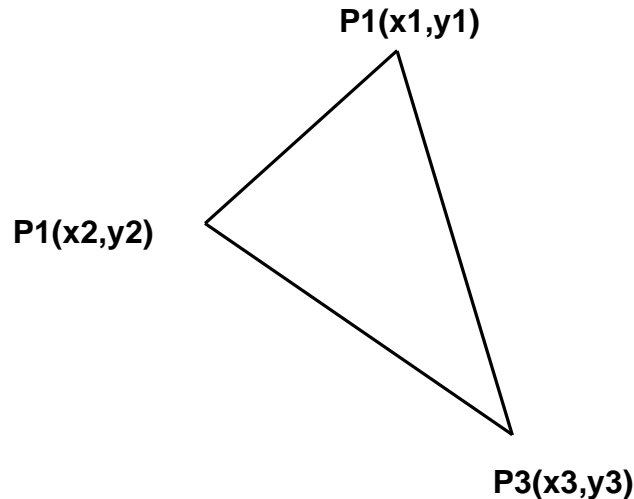


Abb.12 Ein allgemeines Dreieck

Anhand des oben dargestellten Dreiecks wird ersichtlich, dass dieses von 3 Linien begrenzt wird:

- 1. Linie: Von P1 nach P2 (kurz P1↯P2)
- 2. Linie: Von P2 nach P3 (kurz P2↯P3)
- 3. Linie: Von P3 nach P1 (kurz P3↯P1)

Zuerst wollen wir uns damit begnügen, die Umrisse des Dreiecks, d.h. seine Begrenzungslinien, zu zeichnen. Es liegt nahe, zu diesem Zweck die Drawline-Routine zu verwenden. Die Funktion zum Zeichnen des Umrisses eines Dreiecks sollte die 3 Koordinatenpaare der Punkte des Dreiecks, sowie die Farbe in der es gezeichnet werden soll, entgegennehmen und verarbeiten. Innerhalb der Funktion werden lediglich die drei Linien P1↯P2, P2↯P3 und P3↯P1 mit Hilfe der Drawline-Routine gezeichnet.

```
SUB DrawTriangle (x1, y1, x2, y2, x3, y3, Farbe)
    DrawLine(x1, y1, x2, y2, Farbe)
    DrawLine(x2, y2, x3, y3, Farbe)
    DrawLine(x3, y3, x1, y1, Farbe)
END SUB
```

Das Clipping wird wieder von der Routine DrawPixel innerhalb der DrawLine-Prozedur übernommen.

---

<sup>7</sup> Beispielprogramm: „DRAWTRI.BAS“

### 3.2.5) Das Zeichnen ausgefüllter Dreiecke<sup>8</sup>

Das Zeichnen ausgefüllter Dreiecke gestaltet sich weitaus komplexer, als das bloße Zeichnen ihrer Umrisse. Das Prinzip hinter diesem Algorithmus ist wesentlich schwieriger zu verstehen und herzuleiten. Beginnen wir mit einer Abbildung:

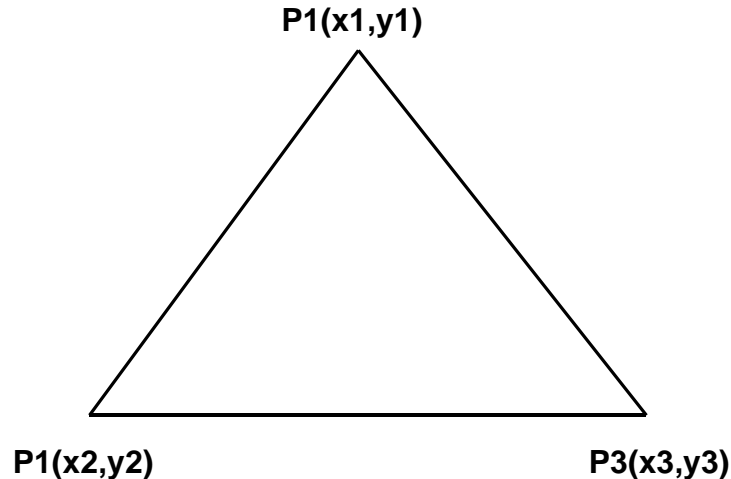


Abb.13 Ein spezielles Dreieck

Das obige Dreieck stellt einen Spezialfall dar. Die Punkte P2 und P3 liegen auf selber Höhe, sprich ihre y-Koordinaten sind ident. Weiters ist noch zu beachten, dass  $y1 < y2 = y3$ . Es wird also das Bildschirmkoordinatensystem mit positiver y-Achse nach unten verwendet. Um dieses Dreieck zu Zeichnen, bedient man sich des sogenannten Scan-Line-Verfahrens. Dabei wird das Dreieck in horizontale Linien aufgeteilt, deren Anfangs- und Endpunkte jeweils Teil einer das Dreieck begrenzenden Linie sind. Diese Anfangs- und Endpunkte können jeweils separat mit dem DrawLine-Algorithmus berechnet werden. Man geht dabei folgendermaßen vor:

P1 ist der Startpunkt sowohl für die Linie P1P2, als auch für die Linie P1P3. Zur Berechnung der Punkte beider Linien brauchen wir jeweils zwei Laufvariablen, die die Koordinaten der zu berechnenden Punkte beinhalten, sowie jeweils eine Variable für die Steigungen der Linien. Die beiden Laufvariablen für die Linie P1P2 nennen wir u1 und v1, die der Linie P1P3 u2 und v2, wobei der Buchstabe u der x-Koordinate und v der y-Koordinate eines Punktes entspricht. Alle vier Variablen werden mit den Koordinaten des Punktes P1 initialisiert, da beide Linien diesem Punkt entspringen. Für alle ganzzahligen y-Koordinaten zwischen y1 und y2 bzw. y3 werden die x-Koordinaten für die Punkte beider Linien berechnet. Das Dreieck wird von unten nach oben gezeichnet, daher ist das y-Inkrement beider Linien 1. Die Steigungen der beiden Linien werden so berechnet:

Steigung der Linie P1P2:  $k12 = (x2 - x1) / (y2 - y1)$   
Steigung der Linie P1P3:  $k13 = (x3 - x1) / (y3 - y1)$

Innerhalb einer Schleife werden nun immer der Anfangs- und Endpunkt einer horizontalen Linie des Dreiecks berechnet. Die beiden Punkte werden

---

<sup>8</sup> Beispielprogramm: „DRAWTRIF.BAS“

dann als Parameter an die DrawLine-Routine übergeben, die wiederum die horizontale Linie zwischen den beiden Punkten zeichnet. Die Schleife wird solange durchlaufen, bis die horizontale Linie zwischen P2 und P3 gezeichnet wurde, d.h.  $v1 = v2 = y3$ . Damit können wir bereits den ersten Spezialfall zeichnen.

```
SUB DrawFilledTriangle (x1, y1, x2, y2, x3, y3, Farbe)
  u1 = x1
  v1 = y1
  u2 = x1
  v2 = y1
  k12 = (x2-x1)/(y2-y1)
  k13 = (x3-x1)/(y3-y1)
  WHILE v1 <= y3
    DrawLine(u1, v1, u2, v2, Farbe)
    u1 = u1 + k12
    v1 = v1 + 1
    u2 = u2 + k13
    v2 = v2 + 1
  WEND
END SUB
```

Die Variablen  $v1$  und  $v2$  könnten natürlich zu einer Variablen zusammengefasst werden, denn sie besitzen immer den gleichen Wert. Ich verzichte jedoch der Verständlichkeit halber darauf. Da dieser Algorithmus vielleicht etwas schwierig zu verstehen ist, möchte ich einen Durchlauf

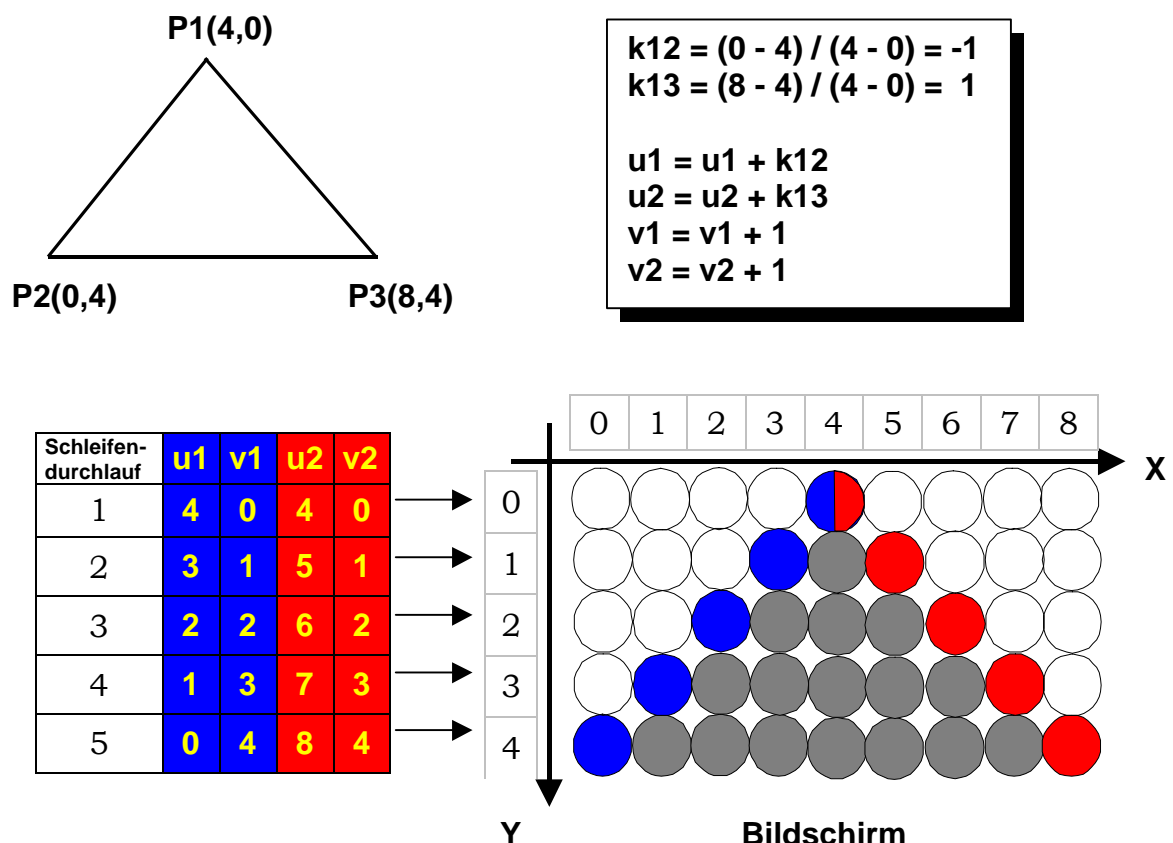


Abb.14 Simulation des bis jetzt erarbeiteten Algorithmus zum Zeichnen von Dreiecken. Die blauen und roten Pixel entsprechen den berechneten Anfangs- und Endpunkten der horizontalen Linien des Dreiecks.



simulieren.

Es gibt noch einen zweiten Spezialfall der dem ersten sehr ähnlich ist:

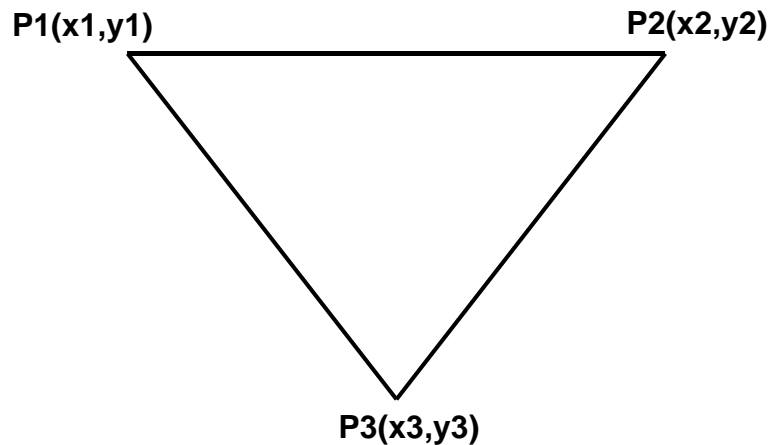


Abb. 15 Der zweite Spezialfall eines Dreiecks

Bis jetzt haben wir die Laufvariablen  $u1$ ,  $v1$ ,  $u2$  und  $v2$  mit den Werten des Punktes  $P1$  initialisiert, da die zwei zu berechnenden Linien diesem Punkt entspringen. Die Linie  $P2P3$  konnte vernachlässigt werden, da diese eine horizontale Linie ist und durch den Algorithmus berechnet wurde (vgl. Abb. 12 Schleifendurchlauf 5). Im obigen Fall liegen aber die Punkte  $P1$  und  $P2$  auf selber Höhe. Zwar bilden diese eine horizontale Linie, jedoch muss nun anstatt der Linie  $P1P2$  die Linie  $P2P3$  berechnet werden. Die beiden zu berechnenden Linien entspringen also nicht mehr dem selben Punkt, wenn wir das Dreieck von unten nach oben zeichnen. Aus diesem Grund müssen wir die Variable  $u1$  anders initialisieren. Für die Berechnung der Linie  $P2P3$  muss auch eine neue Variable für ihre Steigung eingeführt werden.

Am Anfang des Algorithmus sollte geprüft werden, um welchen der beiden Spezialfälle es sich handelt. Davon wird dann die Initialisierung der Variablen abhängig gemacht. Indem wir die  $y$ -Koordinaten der Punkte des Dreiecks miteinander vergleichen, können wir feststellen, mit welchem Fall wir es zu tun haben. Vom ersten Spezialfall wissen wir, dass  $y3 = y2$ , was gleichbedeutend mit  $y2 - y2 = 0$  ist. Wird diese Bedingung erfüllt, handelt es sich um den ersten Spezialfall. Für den zweiten Spezialfall gilt  $y1 = y2$ , d.h.  $y2 - y1 = 0$ .

Wie werden die Laufvariablen initialisiert und wie gestaltet sich deren Berechnung innerhalb der Schleife? Dafür existieren mehrere Lösungsmöglichkeiten. Ich werde aber nur eine besprechen, da dieses Kapitel sonst die Dimensionen dieser Fachbereichsarbeit sprengen würde.

$v1$  und  $v2$  werden unabhängig vom Fall immer mit dem  $y$ -Inkrement 1 kalkuliert. Auch  $u1$  wird immer um die Steigung  $k13$  erhöht. Die Variablen  $u2$ ,  $v1$  und  $v2$  können somit immer mit den Werten von  $P1$  initialisiert werden. Das Problem lässt sich auf die Initialisierung und Berechnung von  $u1$  reduzieren. Die Initialisierung wird wie bereits erwähnt von den Prüfungen auf die Spezialfälle abhängig gemacht. Tritt der zweite Spezialfall ein, wird die Variable  $u1$  mit der  $x$ -Koordinate des Punktes  $P2$  geladen. Bei der Berechnung von  $u1$  wird dann die Steigung  $k23$  verwendet.

Für den Code innerhalb der Schleife bedeutet das, dass abhängig vom Fall  $u1$  entweder mit  $k12$  oder  $k23$  berechnet werden muss. Aus diesem Grund

wird eine allgemeine Variable eingeführt, die entweder den Wert  $k_{12}$  oder  $k_{23}$  besitzt. Diese nennen wir  $k_{temp}$ . Stellt sich bei der Prüfung heraus, dass der erste Spezialfall vorliegt, wird diese Variable mit dem Wert von  $k_{12}$  geladen, für den zweiten mit  $k_{23}$ .  $u_1$  wird innerhalb der Schleife immer mit  $k_{temp}$  berechnet.

Letztendlich erhalten wir folgende Funktion, die beide Spezialfälle verarbeiten kann:

```
SUB DrawFilledTriangle (x1, y1, x2, y2, x3, y3, Farbe)
  IF (y3-y2) = 0 THEN
    k12 = (x2-x1) / (y2-y1)
    ktemp = k12
    u1 = x1
  END IF
  IF (y2-y1) = 0 then
    k23 = (x3-x2) / (y3-y2)
    ktemp = k23
    u1 = x2
  END IF

  v1 = y1
  u2 = x1
  v2 = y1

  WHILE v1 <= y3
    DrawLine (u1, v1, u2, v2, Farbe)
    u1 = u1 + ktemp
    v1 = v1 + 1
    u2 = u2 + k13
    v2 = v2 + 1
  WEND
END SUB
```

Nun können wir uns endlich dem Algorithmus zum Zeichnen von allgemeinen Dreiecken zuwenden. Was würde passieren, wenn wir den bisherigen Algorithmus auf ein allgemeines Dreieck anwenden würden?

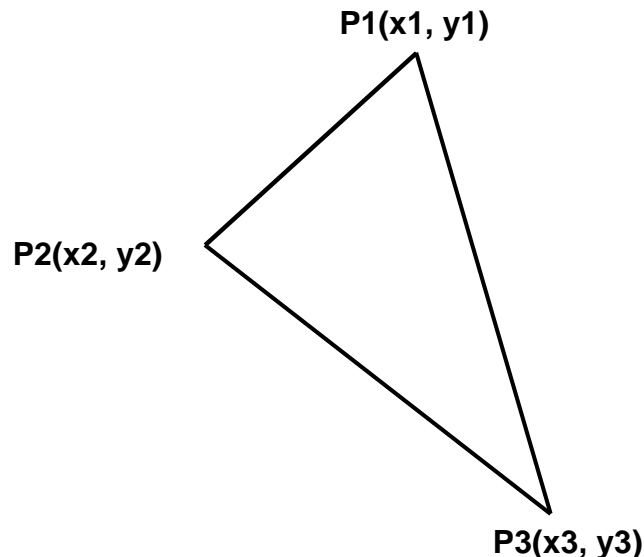


Abb. 16 Ein allgemeines Dreieck

Die Punkte des Dreiecks besitzen alle verschiedene y-Koordinaten. Für die Variablen  $v_1$ ,  $v_2$  und  $u_2$  hätte das keine Bedeutung, da die beiden Variablen  $v_1$  und  $v_2$  immer mit 1 und die Variable  $u_2$  immer mit  $k_{13}$  berechnet werden.  $u_1$  jedoch wurde immer nur entweder mit  $k_{12}$  oder  $k_{23}$  berechnet, d.h. eine der beiden Linien  $P_1P_3$  und  $P_2P_3$  wurde bisher immer vernachlässigt. Zum Zeichnen allgemeiner Dreiecke müssen aber alle Linien berechnet werden, zuerst die Linien  $P_1P_3$  und  $P_1P_2$  und dann die Linie  $P_2P_3$  sowie der Rest der Linie  $P_1P_3$ . Das Dreieck wird in zwei Teildreiecke

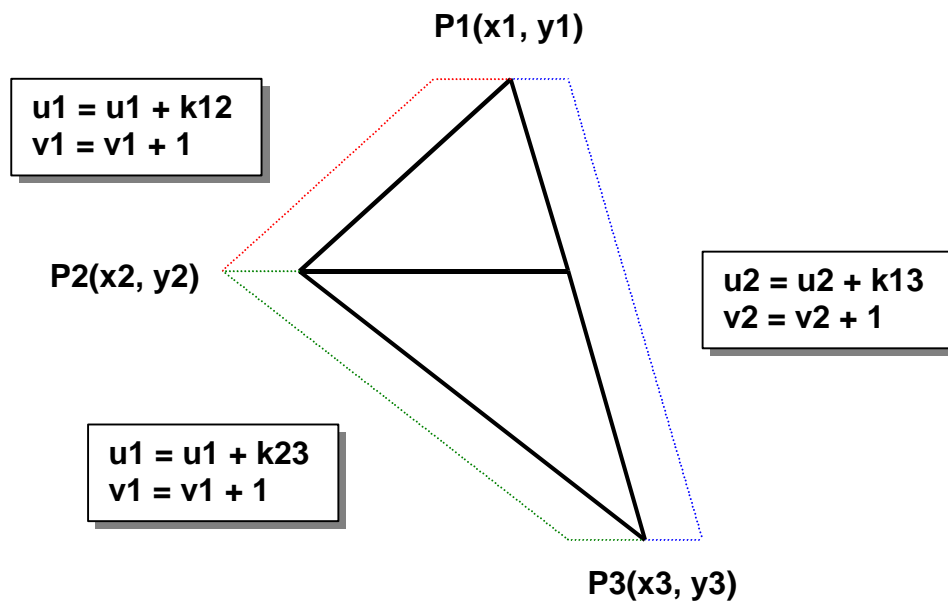


Abb.17 Ein allgemeines Dreieck wird in zwei Teile zerlegt.

zerlegt.

Wir benötigen zwei Schleifen. Die erste wird solange durchlaufen bis  $v_1 = v_2 = y_2$  und damit der erste Teil des Dreiecks gezeichnet wurde.  $u_1$  wird innerhalb dieser Schleife mit  $k_{12}$  und  $u_2$  mit  $k_{13}$  berechnet. Für die zweite Schleife behalten die Laufvariablen  $u_1$ ,  $v_1$ ,  $u_2$  und  $v_2$  ihre Werte, die sie zuletzt in der ersten Schleife erhalten haben, bei.  $u_2$  wird weiterhin mit  $k_{13}$  berechnet.  $u_1$  muss innerhalb der zweiten Schleife aber mit  $k_{23}$  kalkuliert werden. Diese Schleife wird solange ausgeführt, bis  $v_1 = v_2 = y_3$  und damit der untere Teil des Dreiecks gezeichnet wurde. Die Laufvariablen werden vor der Ausführung der ersten Schleife mit den Werten des Punktes  $P_1$  initialisiert.

Leider kann dieser Algorithmus die beiden Spezialfälle nicht verarbeiten. Dazu muss er noch leicht modifiziert werden. Die Schleife für den ersten Teil des Dreiecks entspricht der Schleife für den ersten Spezialfall ( $y_2 = y_3$ ). Werden die Koordinaten eines solchen Dreiecks an den Algorithmus übergeben, zeichnet die erste Schleife das gesamte Dreieck. Die zweite Schleife wird sofort abgebrochen, da  $v_1$  nach der ersten Schleife bereits gleich  $y_3$  ist. Wir müssen daher lediglich eine Division bei der Berechnung von  $k_{23}$  vermeiden. Für den zweiten Fall verwenden wir nur die zweite Schleife. Wir vermeiden über eine IF-Anweisung wieder eine Division durch null bei der Berechnung von  $k_{12}$ . Die erste Schleife wird bei der Ausführung gleich abgebrochen, da ja  $v_1 = v_2 = y_2$  ist. Bevor aber die zweite Schleife

ausgeführt wird, muss  $u1$  noch mit der x-Koordinate von P2 geladen werden, da ansonsten ein Fehler bei der Berechnung der Anfangs- und Endpunkte auftreten würde. Der Algorithmus wäre damit fertig. Er funktioniert aber nur, wenn die Punkte nach y aufsteigend übergeben werden, d.h.  $y1 < y2 < y3$ . Um das zu gewährleisten, werden die übergebenen Koordinatenpaare vor der Ausführung des eigentlichen Algorithmus noch sortiert. Ich verzichte auf die Erklärung des Sortieralgorithmus, da dieser normalerweise von jedem Programmierer beherrscht wird.

```
SUB DrawTriangle (x1, y1, x2, y2, x3, y3, Farbe)
  IF y3 < y2 THEN SWAP y3, y2: SWAP x3, x2
  IF y2 < y1 THEN SWAP y2, y1: SWAP x2, x1
  IF y3 < y2 THEN SWAP y3, y2: SWAP x3, x2

  IF (y2 - y1) <> 0 THEN
    k12 = (x2 - x1) / (y2 - y1)
  ELSE
    k12 = 0
  END IF
  IF (y3 - y1) <> 0 THEN
    k13 = (x3 - x1) / (y3 - y1)
  ELSE
    k13 = 0
  END IF
  IF (y3 - y2) <> 0 THEN
    k13 = (x3 - x2) / (y3 - y2)
  END IF

  u1 = x1
  v1 = y1
  u2 = x1
  v1 = y2

  WHILE v1 <= y2
    DrawLine (u1, v1, u2, v2, Farbe)
    u1 = u1 + k12
    v1 = v1 + 1
    u2 = u2 + k13
    v2 = v2 + 1
  WEND

  u1 = x2

  WHILE v1 <= y3
    DrawLine (u1, v1, u2, v2, Farbe)
    u1 = u1 + k23
    v1 = v1 + 1
    u2 = u2 + k13
    v2 = v2 + 1
  WEND
END SUB
```

### 3.4) Blitting<sup>9</sup>

Blitting wird meist dazu verwendet, ein Bild, das sich in einem eindimensionalen Array oder einer anderen Datenstruktur im Speicher befindet, auf den Bildschirm zu transferieren. Diese Methode wird eingesetzt, um Sprites oder Bildteile auf den Bildschirm zu zeichnen. Im Grunde handelt es sich dabei nur um einen Transfer von Speicherinhalten in den Video-RAM. Bevor man sich um die Ausarbeitung der eigentlichen Funktion kümmert, sollte man sich zuerst ein Format zurechtlegen, indem die Bilder im Speicher abgelegt werden. Für den Videomodus 13h werden wir mit einem eindimensionalen Array arbeiten, dessen Elemente Bytegröße besitzen. Dieser wird ähnlich dem Videospeicher organisiert. Das zu speichernde Bild besteht aus horizontalen Pixelreihen einer gewissen Länge. Innerhalb des Arrays werden diese Zeilen hintereinander abgelegt. Da die Dimensionen des Bildes Variable sein können, sollten vor der eigentlichen Bildinformation noch zwei Array-Elemente liegen, die die horizontale und

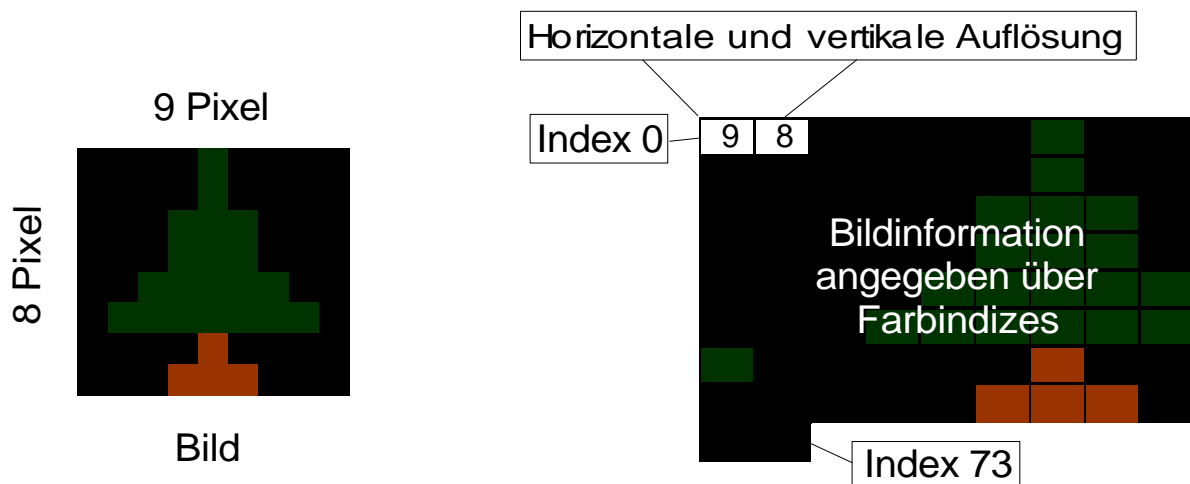


Abb.18 Bild mit seiner Abbildung in einem Array

vertikale Auflösung des Bildes in Byte speichern. Ein solcher Array könnte so aussehen:

Damit hätten wir bereits eine sehr flexible Datenstruktur. Es sei noch bemerkt, dass über diese Datenstruktur nur Bilder bis zu einer Auflösung von 255x255 Pixel definiert werden können, da die ersten beiden die Dimensionen des Bildes bestimmenden Array-Elemente nur Bytegröße besitzen.

Das zu „blittende“ Bild soll an eine beliebige Position am Bildschirm gezeichnet werden. Man gibt dafür die Koordinaten des linken oberen Ecks des Bildes am Bildschirm an.

Der Funktion zum Blitten müssen die Koordinaten, ab der das Bild gezeichnet werden soll, sowie die Adresse des Bild-Arrays übergeben werden. Wieder taucht ein Problem auf: Jede Programmiersprache handhabt Arrays anders, vor allem was deren Lokalisierung im Speicher angeht. BASIC z.B.

<sup>9</sup> Vgl.: <http://www.comprenica.com/atrevda/atrtut10.html>  
Beispielprogramm: „GETBLIT.BAS“

lokalisiert jeden Array am Anfang eines beliebigen Segments. Das erste Element eines solchen Arrays besitzt daher immer den Offset 0 innerhalb des Segments. C wiederum lagert Arrays willkürlich im Speicher aus. Wir sollten darum die Übergabe des Arrays an die Funktion so allgemein wie möglich halten. Zu diesem Zweck soll die Funktion mit der Segment-Offset-Adresse des Arrays arbeiten. Somit garantieren wir eine Kompatibilität zu den wichtigsten Real-Mode-Programmiersprachen, die alle Befehle zum direkten Arbeiten mit Adressen bieten. BASICs VARSEG und VARPTR Befehle liefern z.B. die Segment- und Offset-Adresse einer Variablen bzw. eines Array-Elements. Das heißt, wir können in BASIC einen Array definieren und übergeben dessen Segment-Adresse sowie den Offset des ersten Elements des Arrays an die Blitting-Funktion.

Fassen wir noch einmal zusammen: Als Argumente übergeben wir der Routine die Koordinaten der Position am Bildschirm, ab der das Bild gezeichnet werden soll, sowie die Segment und Offset-Adresse des Arrays. Die Offset-Adresse muss zu Beginn auf das erste Byte der Datenstruktur und damit auf die Angabe der Breite des Bildes zeigen. Diese und die darauffolgende Information über die Höhe des Bildes im nächsten Byte des Arrays lesen wir am Anfang in zwei Variablen ein, die wir widthp (Breite) und heightp (Höhe) nennen. Sie geben an, wie viele Pixel pro Zeile (widthp) bzw. wie viele Zeilen (heightp) gezeichnet werden müssen. Betrachten wir den Code zum Einlesen dieser beiden Werte:

```
SUB Blit (x1, y1, Segment, Offset)
  DEF SEG = Segment
    widthp = PEEK (Offset)
    Offset = Offset + 1
    heightp = PEEK (Offset)
    Offset = Offset + 1
  DEF SEG
  ...
```

Da wir direkt auf die Speicherzellen zugreifen, müssen wir über den Befehl DEF SEG das Segment definieren, auf das mit der Funktion PEEK zugegriffen werden soll. Wir setzen hier die übergebene Adresse des Bild-Arrays ein. Nun können wir über den PEEK-Befehl direkt auf das erste Byte des Arrays und damit auf den Wert für die Breite des Bildes zugreifen. Die Variable Offset besitzt dabei die übergebene Adresse für das erste Byte des Arrays innerhalb des Segments. Der Wert für die Höhe liegt im nächsten Byte. Das heißt, der Zeiger Offset wird um 1 erhöht. Wir lesen den Wert in die Variable heightp ein und erhöhen den Offset wieder um 1. Damit zeigt dieser auf den Farbindex des ersten Pixel der ersten Zeile des Bildes. Die beiden Koordinaten x1 und y1 werden in deren Adresse im Videospeicher umgerechnet. Den so erhaltenen Wert speichern wir in der Variable Pixeladresse, die uns fortan als Zeiger in den Videospeicher dienen soll. Damit haben wir schon fast alle nötigen Variablen für den Hauptteil des Algorithmus, der aus zwei verschachtelten Schleifen besteht. Die äußere Schleife zählt die bereits gezeichneten Zeilen. Die Variable heightp hält die Anzahl der Zeilen des Bildes bereit, über die wir die Durchlaufanzahl der ersten Schleife angeben:

```
FOR y = 1 TO heightp
```

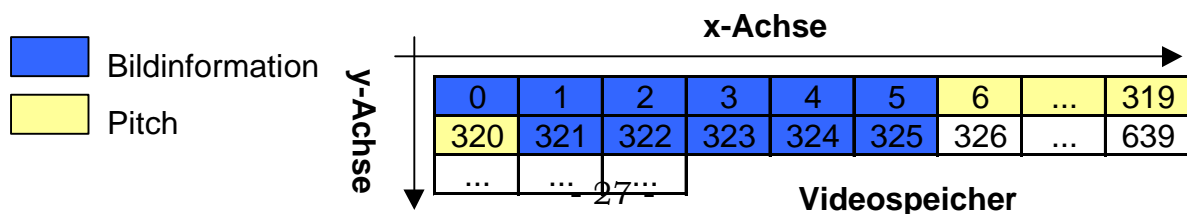
Diese Schleife wird solange ausgeführt, bis alle Zeilen des Bildes gezeichnet wurden. In die ersten Schleife wird eine weitere Schleife eingefügt, die für das Zeichnen der Zeilen verantwortlich ist. Sie zählt, wie viele Pixel der aktuellen Zeile bereits gezeichnet wurden. Hier verwenden wir analog zur ersten Schleife die Variable widthp, um die Durchlaufanzahl anzugeben.

```
FOR x = 1 to widthp
```

Die zweite Schleife wird solange durchlaufen, bis alle Pixel der aktuellen Zeile gezeichnet wurden. Danach erhöht sich die Laufvariable y der ersten Schleife um 1, die zweite Schleife wird wieder von vorne ausgeführt. Erreicht die Laufvariable y der ersten Schleife den Wert von heightp, so wurden alle Zeilen des Bildes gezeichnet und der Algorithmus ist beendet. Für uns ist nun nur noch interessant, wie der Transfer der Bildinformation aus dem Array in den Videospeicher erfolgt. Diese Aufgabe gestaltet sich relativ einfach. Innerhalb der zweiten Schleife lesen wir zuerst den Farbindex für das erste Pixel aus dem Array. Dazu muss wieder das Segment des Arrays über den Befehl DEF SEG als zu bearbeitendes Segment ausgewiesen werden. Wir lesen dann den Farbindex an der Adresse Offset aus und speichern ihn in der Variablen Pixelcolor. Dieser Wert wird dann nur noch an die zuvor berechnete Adresse des ersten Pixel im Videospeicher geschrieben, was wiederum über die Befehle DEF SEG und POKE geschieht. Sind diese Befehle ausgeführt, erhöhen wir die beiden Zeiger Offset und Pixeladresse um 1, um diese auf die Adresse des nächsten zu zeichnenden Pixel der Zeile im Videospeicher (Pixeladresse), sowie auf den dazugehörigen Farbindex im Array (Offset) zeigen zu lassen. Diese Prozedur wird solange wiederholt, bis alle Pixel der Zeile gezeichnet wurden

```
...
  FOR x = 1 TO widthp
    DEF SEG = Segment
    Pixelcolor = PEEK (Offset)
    DEF SEG = &HA000
    POKE Pixeladresse, Pixelcolor
    DEF SEG
    Offset = Offset + 1
    Pixeladresse = Pixeladresse + 1
  NEXT x
...
```

Wie wir wissen, sind innerhalb des Arrays alle Zeilen des Bildes unmittelbar hintereinander abgelegt. Das heißt, der Zeiger Offset braucht immer nur um 1 erhöht zu werden. Für den Zeiger Pixeladresse gilt das nicht. Er muss, nachdem eine Zeile gezeichnet wurde, um einen bestimmten Wert größer 1 erhöht werden. Diesen nennen wir Pitch.





Beim ersten Durchlauf der zweiten Schleife würde die Variable Pixeladresse die Werte 0 bis 5 annehmen. Ohne Addition dieser Variable mit der Pitch würde der Algorithmus die nächste Zeile gleich an die vorhergehende ab der Adresse 6 im Videospeicher anhängen. Wir müssen also den Zeiger Pixeladresse auf das erste Pixel der nächsten Zeile des Bildes am Bildschirm zeigen lassen. Das erreichen wir, indem wir die Pitch zu dieser Variablen addieren.

Betrachten wir die obige Abbildung, so ergibt sich folgende Formel für die Pitch:

$$\text{Pitch} = 320 - \text{widthp}$$

Nachdem eine Zeile gezeichnet wurde, wird diese zur Variablen Pixeladresse addiert. Die Funktion sieht bis jetzt so aus:

```
SUB Blit (x1, y1, Segment, Offset)
  DEF SEG = Segment
  Widthp = PEEK (Offset)
  Offset = Offset + 1
  Heightp = PEEK (Offset)
  Offset = Offset + 1
  DEF SEG

  Pixeladresse = y1 * 320 + x1
  Pitch = 320 - widthp

  FOR y = 1 TO heightp
    FOR x = 1 TO widthp
      DEF SEG = Segment
      Pixelcolor = PEEK (Offset)
      DEF SEG = &HA000
      POKE Pixeladresse, Pixelcolor
      DEF SEG
      Offset = Offset + 1
      Pixeladresse = Pixeladresse + 1
    NEXT x
    Pixeladresse = Pixeladresse + Pitch
  NEXT y
END SUB
```

Zu guter Letzt kommen wir wieder zum Clipping. Da wir die Funktion DrawPixel nicht verwenden, sondern direkt auf den Videospeicher zugreifen, müssen wir uns selbst um das Clipping kümmern.

Wir führen dazu zwei Variablen ein, die die Koordinaten des aktuellen Pixel besitzen, Clipx und Clipy. Diese werden mit den Werten x1 und y1 initialisiert, da wir das Zeichnen bei diesem Punkt beginnen. Innerhalb der zweiten Schleife überprüfen wir über If-Anweisungen, ob das aktuelle Pixel innerhalb des sichtbaren Bildschirmbereichs liegt. Ist dem nicht so, überspringen wir das Schreiben des Farbwerts des Pixel in den Videospeicher. Nach jedem Durchlauf der zweiten Schleife wird die Variable Clipx um 1 erhöht, da die Zeilen von links nach rechts gezeichnet werden.

Ist die zweite Schleife beendet, wird die Variable Clipy um 1 erhöht, da die Zeilen von unten nach oben gezeichnet werden. Die Variable Clipx muss um widthp vermindert werden, um sie wieder an den Anfang der nächsten Zeile zu setzen. Der endgültige Algorithmus sieht wie folgt aus.

```
SUB Blit (x1, y1, Segment, Offset)
  DEF SEG = Segment
  Widthp = PEEK (Offset)
  Offset = Offset + 1
  Heightp = PEEK (Offset)
  Offset = Offset + 1
  DEF SEG

  Pixeladresse = y1 * 320& + x1
  Pitch = 320 - widthp
  Clipx = x1
  Clipy = y1

  FOR y = 1 TO heightp
    FOR x = 1 to widthp
      IF Clipx < 0 OR Clipx > 319 THEN GOTO Skippixel
      IF Clipy < 0 OR Clipy > 199 THEN GOTO Skippixel
      DEF SEG = Segment
      Pixelcolor = PEEK (Offset)
      DEF SEG = &HA000
      POKE Pixeladresse, Pixelcolor
      DEF SEG
Skippixel:
      Offset = Offset + 1
      Pixeladresse = Pixeladresse + 1
    NEXT x
    Pixeladresse = Pixeladresse + Pitch
    Clipx = Clipx - widthp
    Clipy = Clipy + 1
  NEXT y
END SUB
```

### 3.4.1) Transparentes Blitting

Bei jedem Aufruf der Blitting-Methode wird ein rechteckiger Bereich am Bildschirm mit den Pixel des gezeichneten Bildes überschrieben. Die Bildinformation hinter dem Bild geht dabei verloren. Mit der Datenstruktur des Bild-Arrays ist es uns leider nur möglich, rechteckige Bilder zu speichern und zu zeichnen. Wie aber könnte z.B. ein Baum mit dem Hintergrund kombiniert werden ohne diesen zu zerstören?

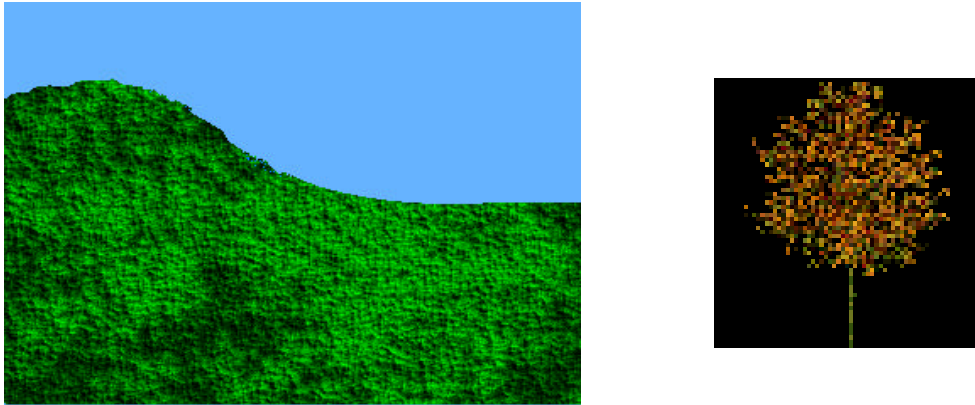


Abb.20 Hintergrund und Objekt einer Szene

Das Bild des Baumes wird wie gehabt im Bild-Array gespeichert. Die Blitting-Methode aber zeichnet nicht nur den Baum selbst, sondern auch den schwarzen Hintergrund der ihn umgibt. Wir müssen der Methode also irgendwie mitteilen, dass sie nur den Baum selbst zeichnen soll. Der Schlüssel dazu liegt in der Farbe des Hintergrunds. Nachdem der Farbwert eines Pixel aus dem Bild-Array gelesen wurde, wird geprüft, ob dieser mit der Farbe des Hintergrunds des Bildes übereinstimmt. Für die obere Abbildung wäre dies der Farbindex 0.

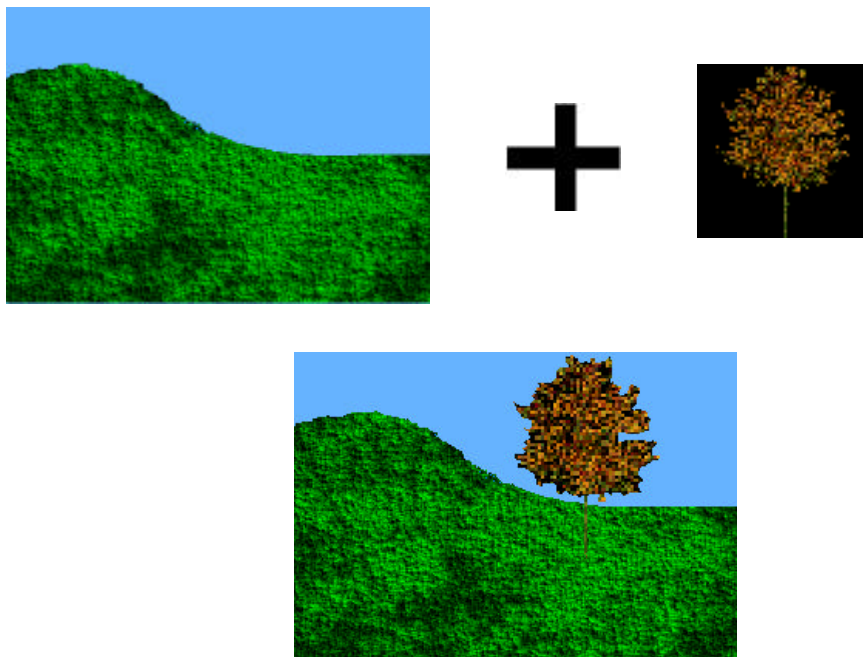


Abb.21 Kombination von Hintergrund und Objekt mit Hilfe der BlitTrans-Routine

Nachdem der Farbwert des aktuellen Pixel aus dem Bild-Array in die Variable Pixelcolor eingelesen wurde, prüfen wir einfach ob die Variable den Wert 0 besitzt. Trifft dies zu, wird das Schreiben des Farbwerts in den Videospeicher übersprungen. Damit erreichen wir einen Transparenz-Effekt. Man muss sich lediglich auf eine Farbe festlegen, die als transparent gelten soll. Die nachfolgende Routine sieht den Farbindex 0 als durchsichtig an. Wollen Sie, dass Teile eines Bildes nicht gezeichnet werden, so müssen diese den Farbindex 0 besitzen.

```
SUB BlitTrans (x1, y1, Segment, Offset)
  DEF SEG = Segment
  Widthp = PEEK (Offset)
  Offset = Offset + 1
  Heightp = PEEK (Offset)
  Offset = Offset + 1
  DEF SEG

  Pixeladresse = y1 * 320& + x1
  Pitch = 320 - widthp
  Clipx = x1
  Clipy = y1

  FOR y = 1 TO heightp
    FOR x = 1 TO widthp
      IF Clipx < 0 OR Clipx > 319 THEN GOTO Skippixel
      IF Clipy < 0 OR Clipy > 199 THEN GOTO Skippixel
      DEF SEG = Segment
      Pixelcolor = PEEK (Offset)
      IF Pixelcolor = 0 THEN GOTO Skippixel
      DEF SEG = &HA000
      POKE Pixeladresse, Pixelcolor

Skippixel:
      DEF SEG
      Offset = Offset + 1
      Pixeladresse = Pixeladresse + 1
    NEXT x
    Pixeladresse = Pixeladresse + Pitch
    Clipx = Clipx - widthp
    Clipy = Clipy + 1
  NEXT y
END SUB
```

### **3.4.2) Die Getpicture-Routine**

Wahrscheinlich haben Sie sich schon gefragt, wie Sie überhaupt ein Bild in einen Array laden können. Normalerweise erstellen Sie ein Bild mit einem herkömmlichen Zeichenprogramm und speichern es im gewünschten Datenformat ab. Die Datei können Sie dann mit einer geeigneten Prozedur in

den Videospeicher laden, wobei die Dimension des Bildes mit der Auflösung des Videomodus übereinstimmen sollte. Mit Hilfe der Getpicture-Routine können Sie dann jeden beliebigen Ausschnitt des Bildschirms in einen Array speichern und später mit der Blitting-Prozedur zeichnen.

Der Algorithmus hinter dieser Funktion ist beinahe deckungsgleich mit dem der Blitting-Routine. Anstatt das Bild in den Videospeicher zu schreiben, lesen wir es von dort in den Array ein. Dabei wird die selbe Datenstruktur erzeugt, wie sie von der Blitting-Methode verlangt wird. Der Funktion muss mitgeteilt werden, welcher Bereich des Bildschirms eingelesen und wohin die Information gespeichert werden soll. Da es sich um einen rechteckigen Bereich handelt, brauchen wir lediglich 2 Koordinatenpaare, die die linke obere Ecke (Startpunkt des Bildes) und die rechte untere Ecke des Bildes definieren. Aus diesen beiden Koordinatenpaaren können wir die Breite und Höhe des Bildes errechnen. Diese Werte werden an die ersten beiden Elemente des Arrays geschrieben, um die Konventionen der Datenstruktur zu erfüllen. Außerdem benötigen wir die beiden Werte für die Variablen widthp und heightp in weiterer Folge für die beiden Schleifen. Die Farbwerte der Pixel des Bildes werden aus dem Videospeicher eingelesen und im Bild-Array gespeichert. Die Berechnung der beiden Zeiger Offset und Pixeladresse bleibt dieselbe, genauso wie die Übergabe der Segment-Offset-Adresse des Arrays. Clipping braucht nicht durchgeführt werden, da der Zugriff auf einen nicht vorhandenen Bildbereich keinen Sinn ergäbe.

```
SUB GetPicture (x1, y1, x2, y2, Segment, Offset)
    heightp = y2 - y1 + 1
    widthp = x2 - y1 + 1

    DEF SEG = Segment
    POKE Offset, widthp
    Offset = Offset + 1
    POKE Offset, heightp
    Offset = Offset + 1
    DEF SEG

    Pixeladresse = y1 * 320& + x1
    Pitch = 320 - widthp

    FOR y = 1 TO heightp
        FOR x = 1 TO widthp
            DEF SEG = &HA000
            Pixelcolor = PEEK (Pixeladresse)
            DEF SEG = Segment
            POKE Offset, Pixelcolor
            DEF SEG
            Offset = Offset + 1
            Pixeladresse = Pixeladresse + 1
        NEXT x
        Pixeladresse = Pixeladresse + Pitch
    NEXT y
END SUB
```

### 3.4.3) Dimensionierung und Übergabe von Bild-Arrays an die Funktionen Blit und Getpicture in BASIC

Im Zusammenhang mit den beiden Prozeduren Blit und Getpicture muss natürlich auch die Dimensionierung der Bild-Arrays erörtert werden. Grundsätzlich muss bei dieser Frage auch die verwendete Programmiersprache einbezogen werden. BASIC z.B. bietet im Gegensatz zu C nur indirekt die Möglichkeit, einen aus Byte-Elementen bestehenden Array zu erzeugen. Um dies zu erreichen, muss der Array vom Typ String \* 1 sein. Mit den BASIC-Befehlen können sie dessen Elementen nur Zeichen zuweisen. Da die oben genannten Funktionen den Array aber direkt über die Befehle PEEK und POKE bearbeiten, fällt dieses Problem weg.

Es stellt sich die Frage wie groß, d.h. aus wie vielen Elementen ein Array bestehen muss, um ein bestimmtes Bild in der verwendeten Datenstruktur aufnehmen zu können. Wie wir wissen sind die ersten beiden Byte für die Werte der Breite und Höhe reserviert. Auf diese folgt die eigentliche Bildinformation Pixel für Pixel. Jedes Pixel nimmt ein Byte ein. Die benötigte Anzahl der Bytes für ein Bild setzt sich also aus der Anzahl der Pixel des Bildes (Höhe mal Breite) plus zwei weiteren Byte für die ersten beiden Elemente, die die Breite und Höhe des Bildes angeben zusammen. Dimensionieren wir in BASIC einen Array mit dem so erhaltenen Wert, sollten wir noch 1 abziehen, da der Array beim Index null beginnen soll.

```
DIM Bildarray ((Breite * Höhe + 2) -1) AS STRING * 1
```

Die Übergabe der Adresse des Arrays erfolgt wie bereits erwähnt über die beiden Befehle VARSEG und VARPTR. VARSEG liefert das Segment zurück in dem sich der Array im Speicher befindet, VARPTR den Offset des übergebenen Array-Elements. Arbeiten sie mit den Befehlen Blit oder Getpicture, müssen sie den Offset des ersten Elements der Bild-Datenstruktur übergeben. Im folgenden Beispiel wird vorausgesetzt, dass die Bild-Datenstruktur ab dem Element 0 vorliegt.

```
Blit 160, 100, VARSEG(Bildarray()), VARPTR(Bildarray(0))
GetPicture 160, 100, 180, 120, VARSEG(Bildarray()), _
    _VARPTR(Bildarray(0))
```

Sie haben auch die Möglichkeit mehrere Bilder in einem Array zu speichern. Diese Methode wird vor allem bei der Programmierung von Sprites eingesetzt. Die Bilder sollten nach Möglichkeit alle die selbe Größe besitzen, um deren Organisation zu erleichtern.

Nehmen wir an, wir wollen zwei Bilder mit der Auflösung von 16 \* 16 Pixel im Array Bilder ablegen. Das erste Bild befindet sich an Position (0;0) das zweite an der Position (16;0). Jedes der beiden Bilder verbraucht 258 Byte im Array, d.h. unser Array muss 516 Elemente groß sein.

```
DIM Bilder (516-1) AS String * 1
```

Das erste Bild wird ab dem Index 0 abgelegt, die Elemente 0 bis 257 werden von ihm eingenommen. Das zweite Bild speichern wir direkt dahinter ab dem Index 258.

```
GetPicture 0, 0, 15, 15, VARSEG(Bilder()), VARPTR(Bilder(0))  
GetPicture 16, 0, 31, 15, VARSEG(Bilder()), VARPTR(Bilder(258))
```

Zeichnen wir das erste Bild, übergeben wir den Offset des Elements 0 an die Blit-Funktion, für das zweite Bild den Offset des Elements 258. Das Ansteuern der einzelnen Bilder kann noch einfacher gestaltet werden, unter der Voraussetzung, dass alle Bilder gleich groß sind. Die Bilder in einem Array werden durchnummeriert, und zwar von null aufsteigend. Brauchen sie nun den Index des Elements, ab dem ein Bild beginnt, multiplizieren sie die Nummer des Bildes mit der Anzahl der benötigten Byte pro Bild. Für das obige Beispiel bedeutet das folgendes:

```
Benötigte Byte pro Bild: 16*16 + 2 = 258  
Start-Index des ersten Bildes: 0*(258) = 0  
Blit 160, 100, VARSEG(Bildarray()), VARPTR(Bildarray(0*258))  
Start-Index des zweiten Bildes: 1*(258) = 258  
Blit 160, 100, VARSEG(Bildarray()), VARPTR(Bildarray(1*258))
```

Speziell bei einer größeren Anzahl von Bildern in einem Array erleichtert diese Methode den Zugriff auf die Bilder um einiges. Sie wird uns im Verlauf der nächsten Kapitel öfter begegnen.

### 3.5) Double Buffering<sup>10</sup>

Das Thema Double Buffering wurde von mir bereits im ersten Kapitel angesprochen. Es dient dazu, auftretendes Flackern zu verhindern. Das Flackern entsteht primär dadurch, dass der Beobachter den Aufbau der verschiedenen geometrischen Figuren bzw. Bilder mitbekommt. Vor allem das Zeichnen von Sprites, das wir im nächsten Kapitel besprechen werden, sowie das Restaurieren des Hintergrundes würde ohne Double Buffering ein sehr unruhiges Bild erzeugen.

Es geht also darum, den Aufbau der aktuellen Szene vor dem Betrachter zu verbergen. Das direkte Schreiben in den Videospeicher bewirkt gleichzeitig auch, dass die Veränderung sofort am Bildschirm sichtbar wird, also genau den Effekt, den wir vermeiden wollen. Aus diesem Grund legt man einen Buffer an, der den Dimensionen des Videospeicher entspricht. Bevor die Szene am Bildschirm zu sehen ist, wird sie im Buffer komplett aufgebaut. Sämtliche Funktionen zum Zeichnen werden also nicht mehr auf den Videospeicher, sondern auf das Segment des Buffers angewendet.

Ist die Szene fertig gezeichnet, wird sie in den Videospeicher kopiert. Damit vermeidet man, dass der Aufbau der einzelnen Bildelemente sichtbar wird und reduziert damit mögliches Flackern.

Als Buffer wird ein Array verwendet, der die selbe Größe wie der Videospeicher besitzt (64000 Byte). So große Arrays werden ausnahmslos am Anfang eines Segments angelegt, was bedeutet, dass das erste Element des Buffers immer den Offset 0 besitzt. Der Buffer ist ein hundertprozentiges Duplikat des Videospeichers und unterscheidet sich von diesem nur durch seine Segment-Adresse. Darum müssen die einzelnen Zeichenfunktionen

---

<sup>10</sup> Vgl.: <http://www.gamedev.net/reference/articles/article350.asp>  
Beispielprogramme: „DOUBLEBF.BAS“, „DBASM.BAS“



lediglich so modifiziert werden, dass sie in ein beliebiges Segment schreiben. Zu diesem Zweck werden sie um einen Parameter erweitert, an den die Adresse des zu bearbeitenden Segments übergeben werden soll. Das hat den Vorteil, dass man sowohl direkt in den Videospeicher, als auch in ein anderes beliebiges Segment zeichnen kann. Die modifizierte DrawPixel-Routine würde daher wie folgt aussehen:

```
SUB DrawPixel ( ZielSegment, x, y, Farbe)
    DEF SEG = ZielSegment
        POKE y*320& + y, Farbe
    DEF SEG
END SUB
```

Wollen Sie in den Buffer schreiben, übergeben Sie einfach die Segment-Adresse des Arrays an die gewünschte Zeichenroutine (z.B. DrawPixel VARSEG(Buffer()) ...). Neben der Modifikation der Routinen benötigen wir auch eine Funktion, die ein Segment in ein anderes kopiert. Wurde die Szene im Buffer fertiggestellt, muss der Buffer-Inhalt in den Videospeicher kopiert werden, um das Bild für den Betrachter sichtbar zu machen.

```
CopySegment ( QuellSegment, ZielSegment)
    FOR Offset& = 0 to 63999
        DEF SEG = QuellSegment
        PixelColor = PEEK (Offset&)
        DEF SEG = ZielSegment
        Poke Offset&, PixelColor
    NEXT Offset&
    DEF SEG
END SUB
```

Ein simpler Speichertransfer: Farbindex aus dem aktuellen Offset im Quellsegment einlesen und an den selben Offset im Zielsegment schreiben. Das Kopieren des Inhalts des Buffers in den Videospeicher würde so aussehen:

```
CopySegment VARSEG(Buffer()), &HA000
```

Sie finden alle abgeänderten Routinen, sowie die Beispielprogramme „DOUBLEBF.BAS“ und „DBASM.BAS“ auf der Anhang-CD.

## **4) Grafik-Engines**

Grafik-Engines fassen alle bisher besprochenen Konzepte zu einem Komplex zusammen, der verschiedene Aufgabenstellungen lösen soll. Diese Aufgaben können vom simplen Zeichnen eines animierten Sprites bis hin zum Darstellen dreidimensionaler Welten reichen. Die gestellte Aufgabe wird in Teilaufgaben zerlegt, für die bestimmte Teile der Engine verantwortlich sind. Man unterscheidet zwischen 2D- und 3D-Engines. 2D-Engines sind meist ohne großen Aufwand zu programmieren und befassen sich primär mit der Organisation und Animation von Bildern. Der mathematische Aufwand hinter solchen Engines ist nicht mit dem von 3D-Engines zu vergleichen. Diese müssen neben dem eigentlichen Zeichenvorgang auch mathematische Operationen, wie z.B. das Rotieren von Punkten im Raum, oder die Projektion dieser auf eine Ebene, durchführen.

Der eigentliche Sinn hinter solchen Engines ist es, diese so allgemein wie möglich zu gestalten, um einen möglichst großen Aufgabenbereich meistern zu können. Ihre Entwicklung erfolgt modular, was eventuelle Wartungsarbeiten bzw. Erweiterungen erleichtert.

Eine komplette Engine zu erarbeiten braucht viel Zeit, manchmal sogar Jahre. Ich hoffe daher auf ihr Verständnis dafür, dass hier lediglich die Grundstrukturen bzw. einzelne Module besprochen werden können. Es dürfte jedoch kein Problem sein, mit dem hier vermittelten Wissen diese Grundstrukturen zu einer vollständigen Engine zu erweitern.

### **4.1) 2D-Engines**

#### *4.1.1) Programmierung von Sprites <sup>11</sup>*

Sprites bilden die Grundlage der zweidimensionalen Computergrafik. Sie werden primär dazu eingesetzt, Spielfiguren bzw. animierte Gegenstände darzustellen.

Wir wollen mit nicht animierten Sprites beginnen.

Jedes Sprite wird am Bildschirm durch ein Bild, wie z.B. dem Baum aus Kapitel 3.4.1 repräsentiert. Dieses Bild wird wie gehabt in einem Bild-Array abgelegt, auf den die Funktionen Blit, Blittrans und Getpicture zugreifen können. Da Sprites in 99 Prozent aller Fälle eine unregelmäßige, d.h. nicht rechteckige Form besitzen, werden wir zum Zeichnen die Funktion Blittrans verwenden. Voraussetzung dafür ist, dass die Farbe des Hintergrunds des Sprites als transparent definiert wird. Das Bild eines Sprites wird auch Frame genannt.

Um Sprites leichter organisieren zu können, legen wir uns eine Klasse zurecht, die sämtliche Eigenschaften eines Sprites beinhaltet. Die erste Eigenschaft eines Sprites ist seine Position am Bildschirm. Außerdem benötigt man Informationen darüber, wo sich das Frame des Sprites befindet. Da Frames in einem Bild-Array abgelegt werden und deren Adressen an die Zeichen-Funktionen übergeben werden, legen wir einfach zwei Variablen an, die das Segment und den Offset des Frames speichern.

---

<sup>11</sup> Vgl.: <http://www.gamedev.net/reference/articles/article353.asp>  
Beispielprogramm: „SPRITES.BAS“, „SPRITANI.BAS“

Das hat auch den Vorteil, dass mehrere Sprites das selbe Frame verwenden können, da dieses ja separat von der Datenstruktur der Sprites im Speicher vorliegt. In BASIC deklarieren wir den Sprite-Datentyp so:

```
TYPE Sprite
  X AS INTEGER
  Y AS INTEGER
  FrSegment AS INTEGER
  FrOffset AS INTEGER
END TYPE
```

Die ersten beiden Variablen geben die Position des Sprites an. Die Variablen FrSegment und FrOffset zeigen auf die Adresse des ersten Byte des Frames im Bild-Array. Soll das Sprite gezeichnet werden, übergibt man einfach diese vier Variablen an die Funktion Blit bzw. Blittrans. Um das Sprite zu bewegen, wird seine Position geändert und sein Frame an den neuen Koordinaten gezeichnet. Wollen Sie mehrere Sprites mit dem selben Frame zeichnen, legen Sie einfach mehrere Variablen dieses Datentyps an, und lassen deren Adressenzeiger auf das selbe Frame im Bild-Array zeigen.

Die Kombination eines Sprites mit dem Hintergrund gestaltet sich leider etwas schwierig. Bei jedem Zeichenvorgang wird ein Teil des Hintergrundes vom Frame des Sprites überzeichnet und dadurch zerstört. Werden bei jedem Aufbau einer Szene sowohl die Sprites, als auch der Hintergrund neu gezeichnet, ergibt sich dieses Problem nicht. Ist der Hintergrund jedoch statisch, d.h. er wird nur einmal in den Videospeicher oder Buffer gezeichnet, muss der vom Frame zerstörte Hintergrundausschnitt restauriert werden. Der einfachste Weg wäre es, noch ein Duplikat des Hintergrundes im Speicher abzulegen und dieses jedes Mal von neuem zusammen mit den Sprites in das Zielsegment zu zeichnen. Der Buffer würde dabei immer mit dem originalen Hintergrund gelöscht werden, was ein Restaurieren unnötig

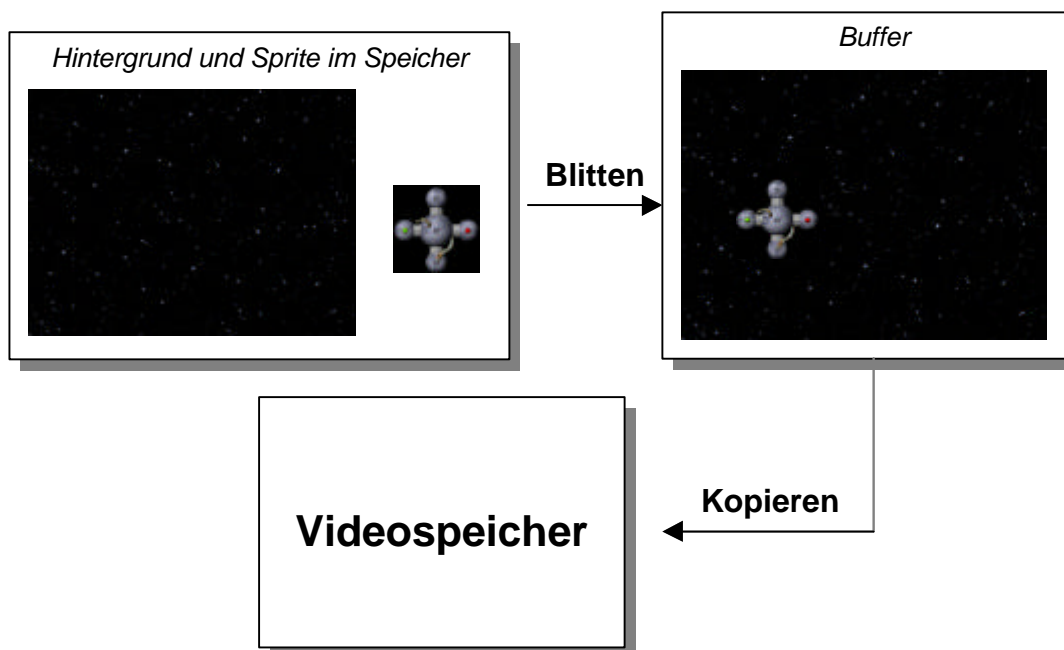


Abb.22 Sprite und Hintergrund werden bei jedem Szenenaufbau in den Buffer geblittet und der Buffer-Inhalt dann in den Videospeicher kopiert

macht.

Der Nachteil dieser Methode liegt auf der Hand: neben dem 64kb großen Buffer brauchen wir noch einmal soviele Speicher für den Hintergrund. Außerdem ist das permanente Kopieren des gesamten Hintergrundes in den Buffer sehr zeitaufwendig. Prinzipiell ist diese Variante aber realisierbar.

Die zweite Möglichkeit, die sich uns bietet, geht von einem anderen Ansatz aus. Jedem Sprite wird dabei noch ein Bild-Array zugewiesen, was sich in der Sprite-Datenstruktur durch zwei weitere Zeigervariablen ausdrückt. Bevor das Sprite über den Hintergrund gezeichnet wird, wird der Hintergrundausschnitt, der überzeichnet werden soll, in diesem zweiten Bild-Array des Sprites gespeichert. Dies geschieht mit Hilfe der GetPicture-Routine. Der benötigte Speicherplatz für diesen Bild-Array entspricht dem eines Frames des Sprites.

Soll das Sprite an eine neue Position gezeichnet werden, wird vorher der gesicherte Hintergrundausschnitt an die alte Stelle kopiert, womit der Hintergrund restauriert wäre. Vor dem Zeichnen an die neue Position wird wieder der Hintergrundausschnitt an dieser neuen Stelle gesichert und das Prozedere beginnt von vorne.

Das Problem bei dieser Methode ist wieder der Speicherbedarf. Bei entsprechend hoher Sprite-Anzahl kann sich der Compiler schon einmal weigern, das Programm zu kompilieren. Trotzdem ist diese Variante weitaus sparsamer als die erste. Das folgende Programm soll Ihnen dieses Prinzip verdeutlichen. Dabei wird das Einlesen des Frames ausgelassen. Wir nehmen einfach an, dass sich bereits ein Bild mit 16x16 Pixel Auflösung im Bild-Array Frame befindet. Als transparente Hintergrundfarbe wird der Farbindex 0 definiert.

```
DIM BUFFER (31999) AS INTEGER
DIM Frame ( 16*16 + 2 - 1) AS STRING * 1
DIM Back ((16*16 + 2) * 2 - 1) AS STRING * 1
```

<< Hier würde das Frame eingelesen werden >>

```
DIM BufSeg, VidSeg AS INTEGER
BufSeg = VARSEG(BUFFER())
VidSeg = &HA000
```

```
TYPE Sprite
  X AS INTEGER
  Y AS INTEGER
  FrSegment AS INTEGER
  FrOffset AS INTEGER
  BaSegment AS INTEGER
  BaOffset AS INTEGER
END TYPE
```

```
DIM Spr1, Spr2 AS SPRITE
```

```
Spr1.x = RND * 319: Spr1.y = RND * 199
Spr1.FrSegment = VARSEG(Frame())
Spr1.FrOffset = VARSEG(Frame(0))
```

```
Spr1.BaSegment = VARSEG(Back())
Spr1.BaOffset = VARSEG(Back(0))

Spr2.x = RND * 319: Spr2.y = RND * 199
Spr2.FrSegment = VARSEG(Frame())
Spr2.FrOffset = VARSEG(Frame(0))
Spr2.BaSegment = VARSEG(Back())
Spr2.BaOffset = VARSEG(Back(16*16+2*1))
```

Bis hierher müsste alles noch relativ klar sein. Wir dimensionieren zuerst drei Arrays, wovon der erste (BUFFER()) für das Double Buffering verwendet wird. Die anderen beiden werden für das Frame (Frame()) und zur Sicherung der Hintergrundausschnitte für beide Sprites verwendet. Es folgt die bereits erklärte Datenstruktur der Sprites, die um weitere zwei Adressvariablen (BaSegment, BaOffset) erweitert wurde. Zwei Variablen, Spr1 und Spr2, werden mit der Sprite-Datenstruktur dimensioniert. Vor Beginn des Hauptteils des Programms müssen die Eigenschaften beider Sprites initialisiert werden. Zuerst weisen wir den beiden Sprites irgendeine Position am Bildschirm zu. Da beide Sprites das selbe Frame besitzen sollen, wird beiden die Adresse des Frames im Array Frame() zugewiesen. Bei den Variablen BaSegment und BaOffset ist das anders. Jedes der beiden Sprites benötigt einen eigenen Speicherbereich, wo der von ihnen überzeichnete Hintergrund gesichert werden kann, jeweils 258 Byte groß (16\*16 Byte Bildinformation + 2 Byte für die beiden Elemente der Breite & Höhe). Diese zwei Speicherbereiche befinden sich im Array Back(). Dem ersten Sprite stehen die Elemente mit den Indizes von 0 bis 257 zur Verfügung, weshalb der Variable BaOffset des ersten Sprites der Offset des Elements 0 übergeben wird. Sichern wir den Hintergrundausschnitt des ersten Sprites, wird dieser nach Übergabe der Variablen BaSegment und BaOffset ab dem ersten Element des Arrays abgelegt. Dem zweiten Sprite stehen die restlichen 258 Byte ab dem Index 258 zur Verfügung. Die Variable BaOffset des zweiten Sprites erhält deshalb den Offset von Back(258).

```
ClearScreen BufSeg, 1
GetPicture BufSeg, Spr1.x, Spr1.y, Spr1.x + 15, Spr1.y + 15, _
    _Spr1.BaSegment, Spr1.BaOffset
GetPicture BufSeg, Spr2.x, Spr2.y, Spr2.x + 15, Spr2.y + 15, _
    _Spr1.BaSegment, Spr2.BaOffset
```

Die ClearScreen-Methode löscht den Buffer mit der Farbe Blau. Selbstverständlich könnte man hier ein Bild in den Array einlesen, ein einfarbiger Hintergrund sollte aber für unsere Zwecke reichen. Bevor die Sprites gezeichnet werden können, müssen die Hintergrundausschnitte, die sie überschreiben werden im Array Back() gespeichert werden. Diese Aufgabe übernehmen die beiden GetPicture-Anweisungen. Jetzt können wir uns endlich der Hauptprogrammschleife widmen.

Die beiden Sprites sollen sich solange von links nach rechts bewegen, bis sie den sichtbaren Bildschirmbereich verlassen haben. Ihre x-Position wird dafür bei jedem Schleifendurchlauf um 1 erhöht. Hat ein Sprite den Bildschirm verlassen, wird ihm eine neue Position zugewiesen.

Do

```
BlitTrans BufSeg, Spr1.x, Spr1.y, Spr1.FrSegment, _  
    _Spr1.FrOffset  
BlitTrans BufSeg, Spr2.x, Spr2.y, Spr2.FrSegment, _  
    _Spr2.FrOffset  
  
CopySegment Bufseg, Vidseg  
  
Blit BufSeg, Spr1.x, Spr2.y, Spr2.BaSegment, Spr2.BaOffset  
Blit BufSeg, Spr2.x, Spr2.y, Spr2.BaSegment, Spr2.BaOffset  
  
Spr1.x = Spr1.x + 1: Spr2.x = Spr2.x + 1  
IF Spr1.x > 319 THEN Spr1.x = RND * 319: Spr1.y = RND * 199  
IF Spr2.x > 319 THEN Spr2.x = RND * 319: Spr2.y = RND * 199  
  
GetPicture BufSeg, Spr1.x, Spr1.y, Spr1.x + 15, Spr1.y + _  
    _15, Spr1.BaSegment, Spr1.BaOffset  
GetPicture BufSeg, Spr2.x, Spr2.y, Spr2.x + 15, Spr2.y + _  
    _15, Spr1.BaSegment, Spr2.BaOffset  
LOOP UNTIL INKEY$ <> ""
```

Gleich zu Beginn der Schleife werden die beiden Sprites in den Buffer gezeichnet. Da die aktuelle Szene damit bereits fertig ist, kopieren wir den Buffer-Inhalt in den Videospeicher. Die Szene wird zum erstenmal sichtbar. Während der Betrachter gebannt auf den Bildschirm starrt, nutzen wir die Zeit gleich und restaurieren den Hintergrund im Buffer. Die Hintergrundausschnitte werden an die Stelle gezeichnet, wo der Hintergrund von den Sprites überschrieben wurde. Jetzt können die Positionen der Sprites verändert werden, was über die nächsten beiden Befehle geschieht. Die darauffolgenden If-Anweisungen prüfen, ob eines der Sprites den sichtbaren Bildschirmbereich verlassen hat und weisen diesem wenn nötig eine neue Position zu. Die beiden Sprites sind nun bereit, an die nächste Position gezeichnet zu werden. Zuvor müssen wir nur noch die beiden Hintergrundausschnitte ab den neuen Koordinaten der Sprites sichern, was die letzten beiden GetPicture-Befehle bewerkstelligen. Wurde keine Taste gedrückt, beginnt die Schleife von vorne.

Sprites sollten nach Möglichkeit in einem Array zusammengefasst werden z.B.

```
DIM Sprites(4) AS Sprites
```

Dadurch erleichtern Sie sich Ihre Arbeit erheblich. Anstatt für jedes einzelne Sprite einen eigenen Blit- bzw. Getpicture-Befehl zu schreiben, können Sie z.B. das Zeichnen der Sprites bequem mit einer Schleife erledigen

```
FOR t = 0 to 4  
    Blittrans BufSeg, Sprites(t).x, Sprites(t).y, _  
        _Sprites(t).FrSegment, Sprites(t).FrOffset  
NEXT t
```

Im hier gezeigten Fall ersparen Sie sich 5 Blit-Befehle hintereinander aufzulisten. Prinzipiell können und sollen Sie die Variante über eine Schleife für jede Operation mit Sprites verwenden. Stellen Sie sich den Sourcecode für 100 Sprites vor, der ohne diese Methode entstehen würde.

Zum Abschluß des Kapitels möchte ich noch auf die Technik zur Animation von Sprites eingehen. Um Animation zu erzielen, werden, wie bei einem Trickfilm, die einzelnen Animationsphasen nacheinander gezeichnet. Einem Sprite werden mehrere Frames zugeschrieben.

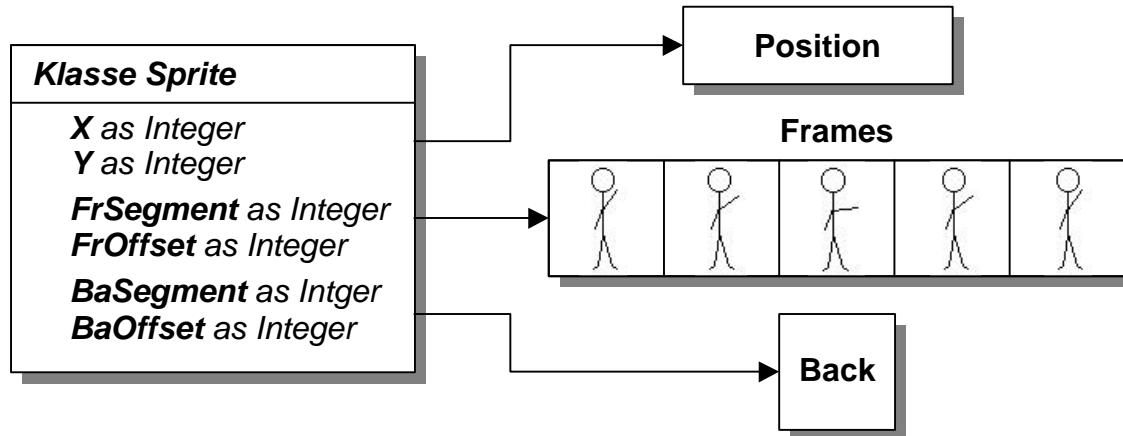


Abb.23 Die Eigenschaft eines Sprites

Im Array `Frame()` werden die einzelnen Frames hintereinander abgelegt. Wie in Kapitel 3.4.3 gezeigt, werden die Frames von Null aufwärts durchnummeriert. Die Eigenschaften des Sprites werden zu Beginn wie gehabt initialisiert. Die Variablen `FrSegment` und `FrOffset` zeigen auf das erste Frame der Animationsphasen. Der Sprite-Klasse wird aber noch eine Variable hinzugefügt, die die Nummer des aktuellen Frames speichert. Wir nennen sie `CurFrame`. Sie wird zu Beginn mit dem Wert Null initialisiert.

Zwischen den einzelnen Animationsphasen sollte eine bestimmte Zeit verstreichen, da der Benutzer die Animation sonst nicht erkennen kann. Diese Zeitspanne ist abhängig von der Anzahl der Frames, dem gewünschten Animationseffekt und so weiter. Wollen Sie Ihre eigene Animation erzeugen, müssen Sie etwas experimentieren, bis Sie den richtigen Wert gefunden haben. Wurde die erste Animationsphase lange genug gezeichnet, wird der Zeiger `FrOffset` auf das nächste Frame gesetzt. Dazu benötigen wir den Offset des ersten Elements des nächsten Frames. Um den Index dieses Elements zu erhalten, bedienen wir uns dem Wert von `CurFrame`. Dieser muss vor der Berechnung um eins erhöht werden, damit er auf das aktuelle Frame verweist. Außerdem sollte überprüft werden, ob er auf ein gültiges Frame zeigt, d.h. bei 8 Frames darf er nie den Wert 8 annehmen. Passiert dies, wird er auf den Wert Null für das erste Frame gesetzt und die Animation beginnt von vorne. Die Berechnung des Offset des nächsten Frames mit Hilfe der Variable `Curframe`:

```
benötigter Platz pro Frame = Höhe * Breite + 2
Sprite.FrOffset = VARPTR(Frames((Höhe*Breite + 2) * Curframe))
```

Um die Zeitabstände zwischen dem Umschalten auf das nächste Frame messen zu können, verwenden wir in BASIC die Funktion TIMER. Sie liefert uns die seit Mitternacht vergangenen Sekunden bis auf eine hundertstel Sekunde genau zurück.

Beim folgenden Programmcode nehmen wir an, dass 4 Frames einer Animationsphase im Array Frames() abgelegt wurden. Ein Sprite soll mit diesen Frames animiert werden. Die Initialisierung der Eigenschaften der Sprites, sowie die Dimensionierung der verschiedenen Variablen wird von mir ausgelassen, damit wir uns auf den Hauptteil des Programms konzentrieren können. Zwischen den Animationsphasen sollen 0.25 Sekunden vergehen, jedes Frame benötigt 258 Byte im Array ( $16 * 16 + 2$ ).

```
Spr.CurFrame = 0
Zeitspanne = TIMER
```

```
Do
    BlitTrans BufSeg, Spr.x, Spr.y, Spr.FrSegment, _
        _Spr1.FrOffset
    CopySegment Bufseg, Vidseg
    Blit BufSeg, Spr.x, Spr.y, Spr.BaSegment, Spr.BaOffset

    IF TIMER - Zeitspanne > 0.25 THEN
        Spr.CurFrame = Spr.CurFrame + 1
        IF Spr.CurFrame > 3 THEN Spr.CurFrame = 0
        Spr.FrOffset = VARPTR(Frames(Spr.CurFrame*258))
        Zeitspanne = TIMER
    END IF

LOOP UNTIL INKEY$ <> ""
```

Die ersten drei Befehle zeichnen das Sprite, kopieren den Buffer-Inhalt in den Videospeicher und restaurieren den Hintergrund. Da sich das Sprite nicht bewegt, muss auch kein neuer Hintergrundausschnitt eingelesen werden.

Der interessante Teil ist die If-Anweisung, die die vergangene Zeit misst. Sind 0.25 Sekunden vergangen, wird der Rumpf der Anweisung ausgeführt. Zuerst wird die Variable CurFrame auf das nächste Frame gesetzt. Die Frames besitzen die Nummern 0, 1, 2 und 3. Ist der Wert von CurFrame größer als drei, verweist er auf ein nicht gültiges Frame und wird daher auf das erste Frame der Animation zurückgesetzt. Es folgt die Berechnung des Offset des aktuellen Frames, wobei wir die Nummer des Frames mit der benötigten Anzahl an Bytes pro Frame multiplizieren. Zuletzt wird die Variable Zeitspanne auf die aktuelle Zeit gesetzt und die Schleife beginnt von vorne. Solange die Zeitspanne nicht vergangen ist, wird das bisher verwendete Frame gezeichnet.



### **4.1.2) Scrolling** <sup>12</sup>

Spiele, vor allem Strategiespiele wie Westwood's Command & Conquer, erfordern es oft, große Areale darzustellen, die im Ganzen nicht am Bildschirm Platz finden. Der Hintergrund der Spielwelt besteht aus einer Karte die sich wiederum aus mehreren bildschirmfüllenden Bildern zusammensetzt. Am Bildschirm selbst wird immer nur ein Teil der Karte gezeigt. Der Bildschirm kann als Fenster zu dieser Karte angesehen werden. Das Bewegen über die Karte, d.h. das sichtbar machen verschiedener Kartenausschnitte, wird allgemein als Scrolling bezeichnet.

---

<sup>12</sup> Vgl.: <http://www.geocities.com/SiliconValley/Pines/1732/tut.htm>  
Beispielprogramme: „TBTSCR.BAS“, „PBPSR.BAS“

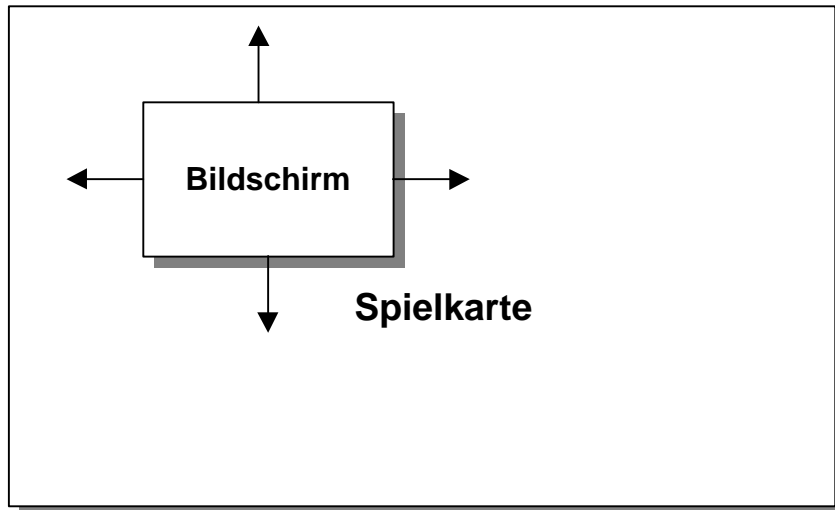


Abb.24 Der Bildschirm dient als Fenster zur Spielkarte und kann über diese bewegt werden.

Je nachdem, wie diese Karte im Speicher vorliegt, bzw. welche Möglichkeiten dem Betrachter zum Scrollen zur Verfügung stehen, unterscheidet man zwischen verschiedenen Scrolling-Engines. Die Karte kann z.B. als komplettes Bild im Speicher vorliegen, wobei gleich vorweggenommen werden soll, dass diese Methode im Real-Mode nicht möglich ist. Denken Sie daran, dass bereits eine Bildschirmseite ein ganzes Segment okkupiert.

Aus diesem Grund muss eine andere Methode gefunden werden, die weitaus Speicher schonender ist. Tile-Scrolling ist die wohl beste Lösung dieses Problems. Die gesamte Spielkarte wird aus einer begrenzten Anzahl an Kacheln (engl. Tile) aufgebaut, was natürlich die Vielfältigkeit der Spielwelt etwas einschränkt. Alle Kacheln besitzen die selbe Größe, meist zwischen 16x16 und 32x32 Pixel. Die Kachel werden, wie könnte es auch anders sein, in einem Bild-Array abgelegt, durchnummeriert und später mit der Blit-Funktion gezeichnet. Die eigentliche Spielkarte befindet sich in einem zweidimensionalen Array, wobei die Elemente jeweils die Nummer der ihnen zugewiesenen Kachel speichern.

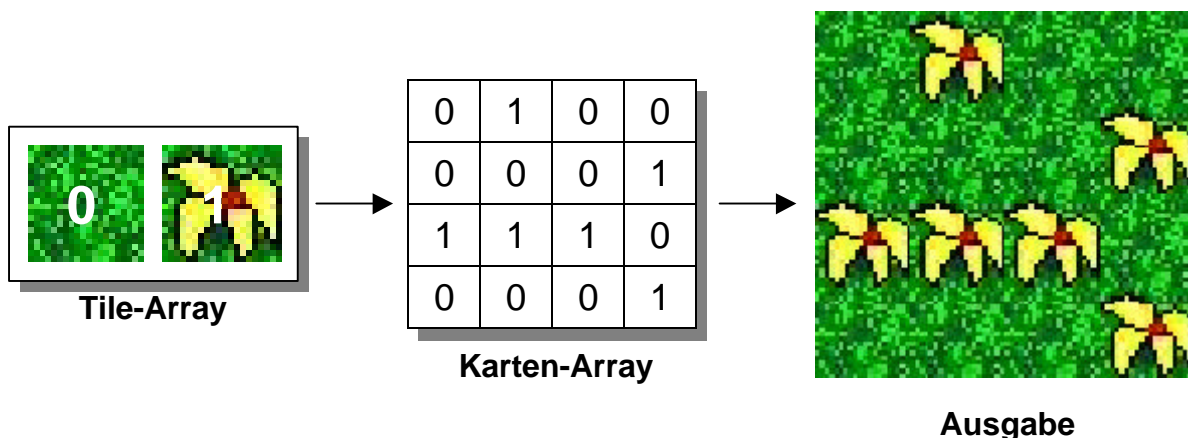


Abb.25 In der Karte werden die Tiles durch ihre Nummern repräsentiert. Bei der Ausgabe werden über diese Nummern die Adressen der Tiles für die Blit-Routine errechnet.

Grundsätzlich sollte man darauf achten, die Größe der Kacheln so zu wählen, dass deren Anzahl in einer Reihe bzw. in einer Spalte am Bildschirm einen integeren Wert ergibt. Bei einer Auflösung von 20x20 Pixel pro Kachel

ergibt das z.B. 16 Kacheln pro Reihe ( $320 \setminus 20 = 16$ ) bzw. 10 Kacheln pro Spalte ( $200 \setminus 20 = 10$ ). Bei diesen Größen wird der Bildschirminhalt daher aus 160 Kacheln aufgebaut. Bleiben wir gleich bei diesem Beispiel.

Wir wollen uns vorerst nur mit dem Zeichnen eines Kartenausschnitts ab den Indizes (0,0) im Karten-Array begnügen und lassen das Scrolling einstweilen beiseite. Dazu dimensionieren wir einen Karten-Array, der aus 160 Kacheln besteht, d.h. gezeichnet lediglich eine Bildschirmseite einnimmt.

```
DIM Karte (0 TO 15, 0 TO 9) AS INTEGER
```

Darüber hinaus benötigen wir einen Bild-Array, der die Kacheln speichert. Bei einer Größe von 20x20 Pixel pro Tile ergibt sich folgende Dimensionierung des Bild-Arrays:

```
DIM Tiles (( 20 * 20 + 2) * 2 - 1) AS STRING * 1
```

Die benötigte Anzahl an Bytes pro Kachel speichern wir in der Variable BytesProTile.

```
BytesProTile = 20 * 20 + 2
```

Wie Sie bereits aus den vorhergehenden Kapiteln wissen, erhalten wir bei hintereinander liegenden Bildern durch diesen Wert, multipliziert mit der Nummer des Bildes, den gesuchten Index zur Offset-Berechnung. Den Offset benötigen wir für den Blit-Befehl, mit dem die Kacheln gezeichnet werden.

Da unsere Karte aus zwei Kacheln zusammengesetzt wird, können die Elemente des Arrays Karte () nur die Werte 0 und 1 annehmen (erste Kachel: 0, zweite Kachel: 1). Wir nehmen an, dass sowohl die Karte, als auch der Tile-Array mit Werten versehen worden ist.

Der Kartenausschnitt wird am Bildschirm zeilenweise, beginnend im linken oberen Eck des Bildschirms, von unten nach oben, gezeichnet. Die Kacheln einer Zeile werden unmittelbar hintereinander gezeichnet, wobei sich die x-Koordinate der nächsten Kachel aus der der vorhergehenden Kachel plus der Kachelbreite ergibt. In unserem Fall hat die erste Kachel die Koordinaten (0;0) und eine Breite von 20 Pixel. Daraus folgt, dass die nächste Kachel der ersten Zeile die Koordinaten (20;0) und die dritte Kachel die Koordinaten (40;0) am Bildschirm besitzt. Die letzte Kachel einer Zeile wird an die x-Koordinate 300 gezeichnet. Aus diesen Informationen können wir eine Schleife konstruieren, deren Laufvariable die x-Koordinate der aktuellen Kachel einer Zeile angibt.

```
FOR Kachelx = 0 TO 300 STEP 20  
...  
NEXT Kachelx
```

Neben der x-Koordinate der aktuellen Kachel benötigen wir auch deren y-Koordinate. Diese ist von der Zeile, in der sich die Kachel befindet, abhängig. Alle Kacheln einer Zeile besitzen die selbe y-Koordinate am Bildschirm. Hier kann analog zur Berechnung der x-Koordinate vorgegangen werden. Die

erste Zeile besitzt die y-Koordinate 0. Da die Höhe der Kacheln wie die Breite 20 Pixel beträgt, befindet sich die nächste Zeile an der y-Koordinate 20, die dritte an der y-Koordinate 40 u.s.w.. Die letzte Zeile besitzt die y-Koordinate 180 woraus sich kombiniert mit der ersten Schleife letztendlich diese Konstruktion ergibt.

```
FOR Kachely = 0 TO 180 STEP 20
  FOR Kachelx = 0 TO 300 STEP 20
    ...
  NEXT Kachelx
NEXT Kachely
```

Die Nummer der ersten zu zeichnenden Kachel befindet sich in diesem Beispiel an den Indizes (0,0) im Karten-Array. Diese lesen wir aus und verwenden sie innerhalb der Blit-Anweisung zur Berechnung des ersten Index der Kachel im Tile-Array. Die beiden Laufvariablen Kachelx und Kachely besitzen zu diesem Zeitpunkt beide den Wert 0. Sie geben gleichzeitig die Position dieser Kachel am Bildschirm an. Für die nächste Kachel lesen wir die Nummer an den Indizes (1,0) des Karten-Arrays ein und zeichnen sie wieder an den Koordinaten (Kachelx, Kachely), die nun die Werte (20;0) angenommen haben. Irgendwie müssen wir die Indizes des auszulesenden Elements mit den Laufvariablen Kachelx und Kachely in Verbindung bringen können. Hier können wir uns eines kleinen Tricks bedienen. Beginnt das Auslesen der Karte an den Indizes (0,0), erhalten wir die gesuchten Indizes der aktuellen Kachel indem wir Kachelx und Kachely durch die Kachelbreite bzw. Kachelhöhe dividieren.

```
FOR Kachely = 0 TO 180 STEP 20
  FOR Kachelx = 0 TO 300 STEP 20
    Kachelnummer = Karte(Kachelx\20, Kachely\20)
    Blit BufSeg, Kachelx, Kachely, VARSEG(Tiles()), _
      _VARPTR(Tiles(BytesProTile*Kachelnummer))
  NEXT Kachelx
NEXT Kachely
```

Achten Sie darauf, dass die Indizes über eine Integer-Division errechnet werden, da diese prinzipiell nur ganzzahlige Werte annehmen.

Mit dieser Methode können wir den ersten Ausschnitt ab den Indizes (0,0) zeichnen.

Der Kartenausschnitt, der am Bildschirm sichtbar sein soll, wird lediglich über die Indizes seiner ersten Kachel definiert. Im letzten Beispiel entsprach das dem Element mit den Indizes (0,0). Um nun ein flexibles Zeichnen der Karte und im weiteren Sinn Scrolling zu ermöglichen, führen wir zwei Variablen ein, die die Indizes der ersten Kachel des zu zeichnenden Kartenausschnitts angeben. Wir nennen sie Mapx und Mapy. Diese werden zu den errechneten Indizes eines Kartenausschnitts beginnend bei den Indizes (0,0) hinzuaddiert. Es werden weiterhin 16 Element pro Zeile ausgelesen, nun aber ab den Indizes (Mapx, Mapy). Das Scrolling erreicht man, indem man einfach diese beiden Variablen auf andere Indizes setzt. Für ein Scrolling um eine Kachel nach oben subtrahiert man z.B. den Wert 1 von Mapy u.s.w.

```
FOR Kachely = 0 TO 180 STEP 20
  FOR Kachelx = 0 TO 300 STEP 20
    Kachelnummer = Karte(Kachelx\20 + Mapx, Kachely\20 + _
                        _Mapy)
    Blit BufSeg, Kachelx, Kachely, VARSEG(Tiles(), _
        _VARPTR(Tiles(BytesProTile*Kachelnummer)))
  NEXT Kachelx
NEXT Kachely
```

Durch diese Methode wird kachelweise gescrollt. Aus Platzgründen kann ich hier auf das pixelweise Scrolling leider nicht eingehen, Sie finden jedoch zu dieser Methode auf der Anhang-CD das Beispielsprogramm „PBPSR.BAS“.

## 4.2) 3D-Engines

### 4.2.1) Ray-Casting<sup>13</sup>

Ray-Casting ist eine Technik, bei der Information z.B. in Form einer zweidimensionalen Karte einer Welt, in eine dreidimensionale Projektion übersetzt wird. Dies geschieht, indem unsere Art des Sehens imitiert wird. Im Gegensatz zum realen Leben, sendet jedoch der virtuelle Betrachter (Licht-)Strahlen aus, bis diese auf ein Objekt stoßen.

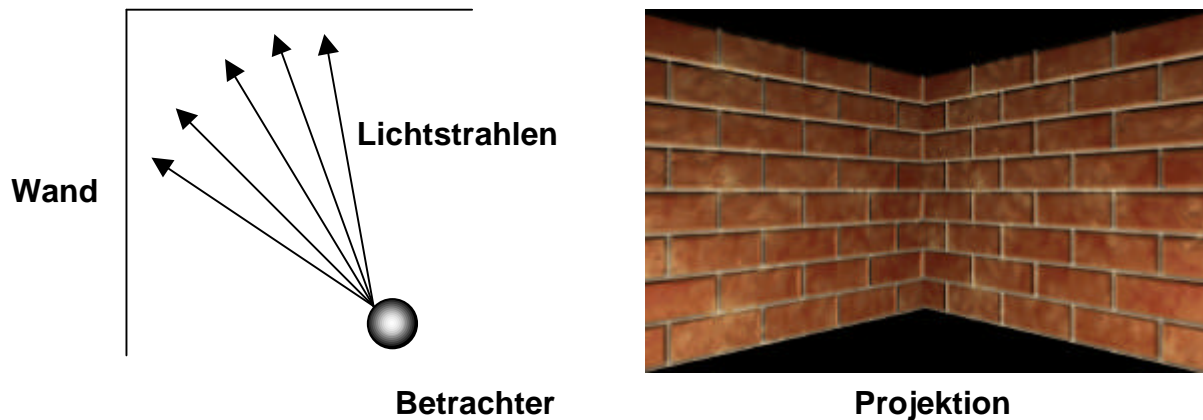


Abb.26 Die zweidimensionale Karte wird in eine dreidimensionale Projektion umgewandelt.

Ray-Casting ist der kleine Bruder der Ray-Tracing-Technik, die die Strahlen in einen dreidimensionalen Raum aussendet. Es fand vor allem deshalb in vielen Spielen Verwendung, da es im Gegensatz zum Ray-Tracing die Daten sehr schnell in eine dreidimensionale Abbildung projizieren kann. Der Grund dafür liegt in der Tatsache, dass bei der verwendeten Geometrie einige Abstriche gemacht werden. So stehen die Wände z.B. normal zum Boden, d.h. schiefe Formen wie Dachgiebel oder ähnliches sind nicht möglich. Auch kann der Betrachter nicht um die Z-Achse rotieren, da sonst der Geschwindigkeitszuwachs, der durch das Zeichnen der Wände in vertikalen Linien erreicht wird, verloren ginge.

Prinzipiell gelten folgende Restriktionen:

- Die Wände schließen mit dem Boden einen Winkel von 90° ein.
- Die Wände können aufeinander nur normal stehen (Jede Ecke 90°).
- Die Wände bestehen aus Würfeln, die alle die selbe Größe besitzen, was zur Folge hat, dass alle Wände die selbe Höhe haben.

Durch verschiedene Kunstgriffe können diese oben genannten Beschränkungen jedoch zum Teil aufgehoben werden.

Die Würfel, aus denen die Wände bestehen, werden in diesem Kapitel durchgehend die Größe von 64x64x64 Einheiten besitzen. Programmcodes wie in den vorhergehenden Kapiteln werden Sie nicht finden, dafür sollten verschiedene Pseudocodes für Klarheit sorgen.

---

<sup>13</sup> Vgl.: <http://www.programmersheaven.com/zone10/cat183/6287.htm>  
Beispielprogramm: „RAYCAST.BAS“, „VBRAY.BAS“

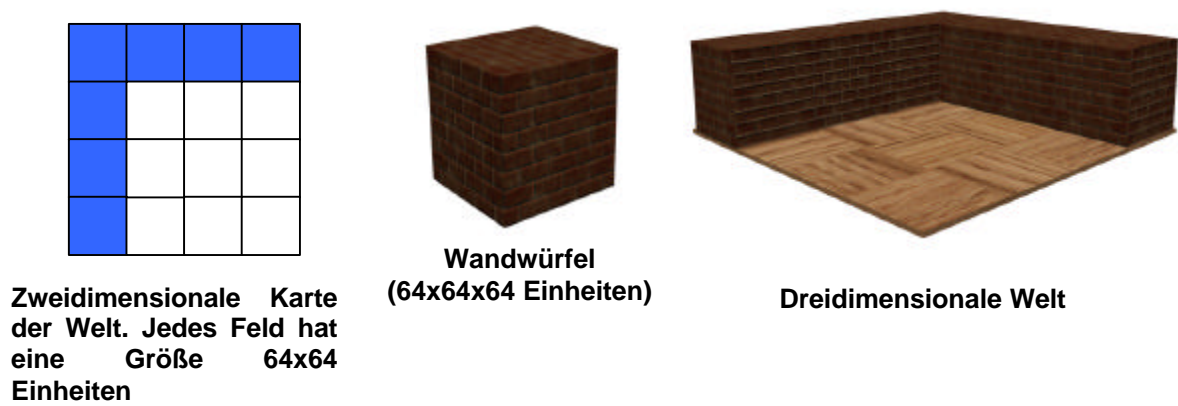
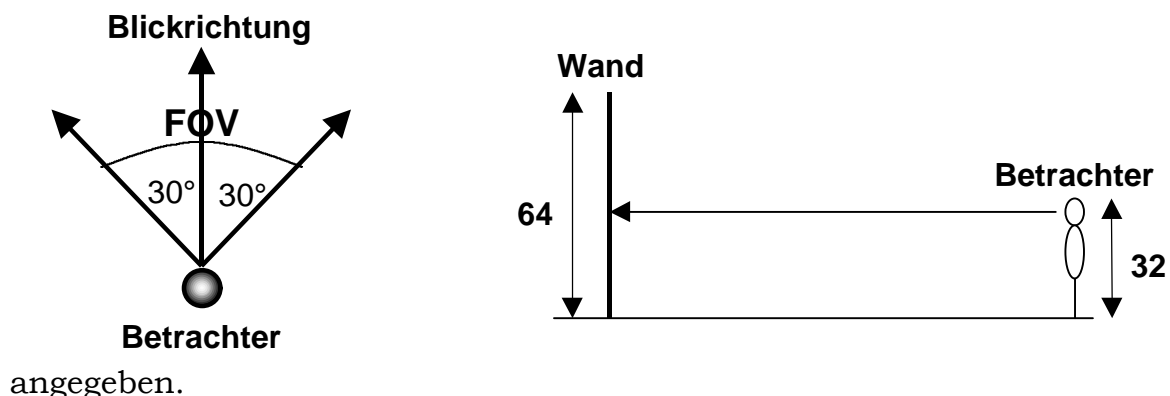


Abb.27 Sowohl die zweidimensionale als auch die dreidimensionale Welt ist aus Würfeln aufgebaut

Da die Gestaltung der Welt damit behandelt wäre, wenden wir uns der Projektion zu. Bevor wir diese anwenden können, müssen verschiedene Faktoren bekannt sein:

- Höhe des Betrachters
- Sichtfeld des Betrachters (engl. „Field of View“, kurz FOV)
- Position und Blickrichtung des Betrachters innerhalb der virtuellen Welt
- Dimensionen der Projektionsebene
- Abstand zwischen Betrachter und Projektionsebene

Das FOV des Betrachters definiert den Bereich der Welt, der für den Betrachter sichtbar ist. Die Strahlen werden innerhalb dieses Sichtkegels ausgesendet. Ein durchschnittlicher Mensch besitzt ein Sichtfeld von ca. 90°. Würden wir diesen Wert unserem virtuellen Betrachter zuweisen, wären die errechneten Szenen etwas unrealistisch. Ein Sichtfeld von 60° liefert die besten Ergebnisse. Dieser Wert ist willkürlich gewählt, aus eigener Erfahrung weiß ich aber, dass von dieser Größe stark abweichende Werte sich als relativ ungünstig erweisen. Als Höhe des Betrachters verwenden wir die halbe Höhe der Wände, d.h. 32 Einheiten. Dadurch zentrieren wir die Welt am Bildschirm. Die Position des Betrachters wird über seine x- und y-Koordinate innerhalb der zweidimensionalen Welt angegeben. Da er sich nicht nach oben oder unten bewegen kann, wird die dritte Koordinate vernachlässigt. Diese entspricht der Höhe des Betrachters und sollte einen fixen Wert besitzen. Die Blickrichtung wird gleich wie das FOV in Grad



Über die zweidimensionale Weltkarte werden zwei Koordinatensysteme gelegt. Das erste nennen wir Kartenkoordinatensystem. Hier besitzt jedes Feld ein integeres Koordinatenpaar und keine Ausdehnung. Das zweite nennen wir Einheitenkoordinatensystem, wobei jedes Feld jeweils die Abmessungen 64x64 Einheiten besitzt.

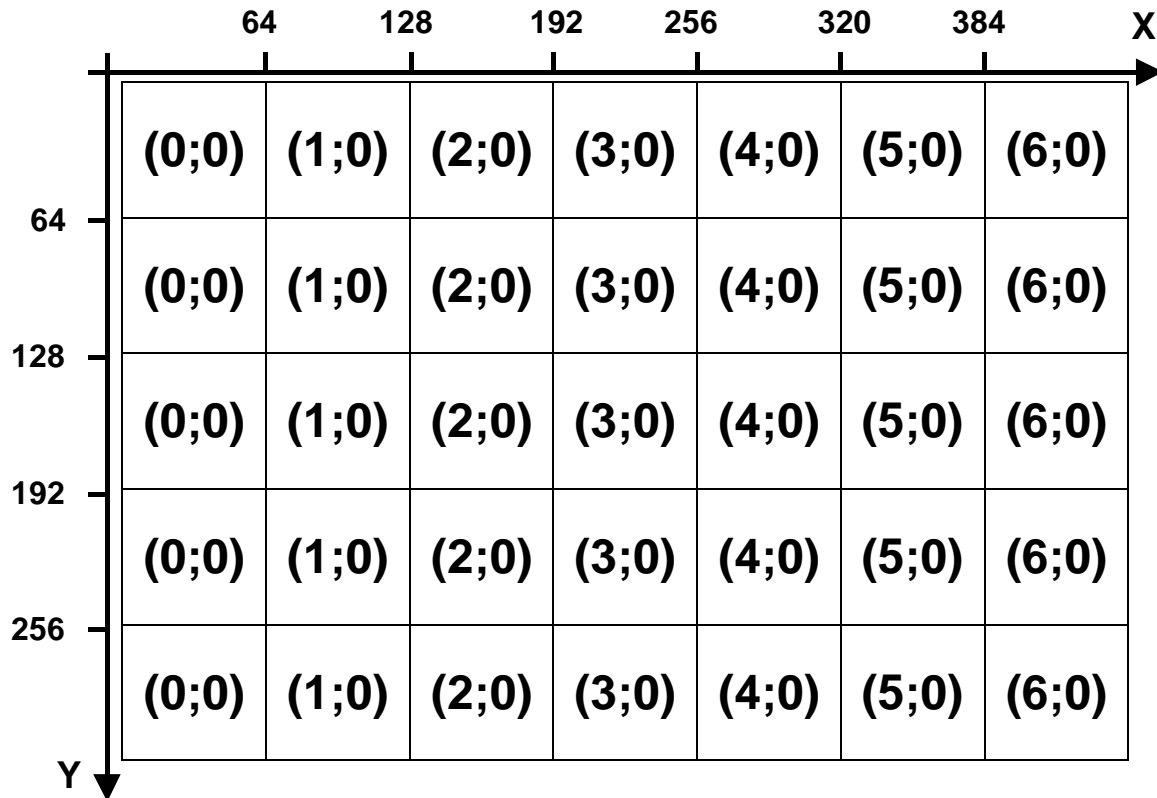


Abb.29 Innerhalb der Felder stehen deren Koordinaten im Kartenkoordinatensystem. Die Werte auf den beiden Achsen geben die Einheitenkoordinaten der jeweiligen Feldgrenzlinie an

Die Umrechnung von Einheitenkoordinaten in Kartenkoordinaten gestaltet sich folgendermaßen:

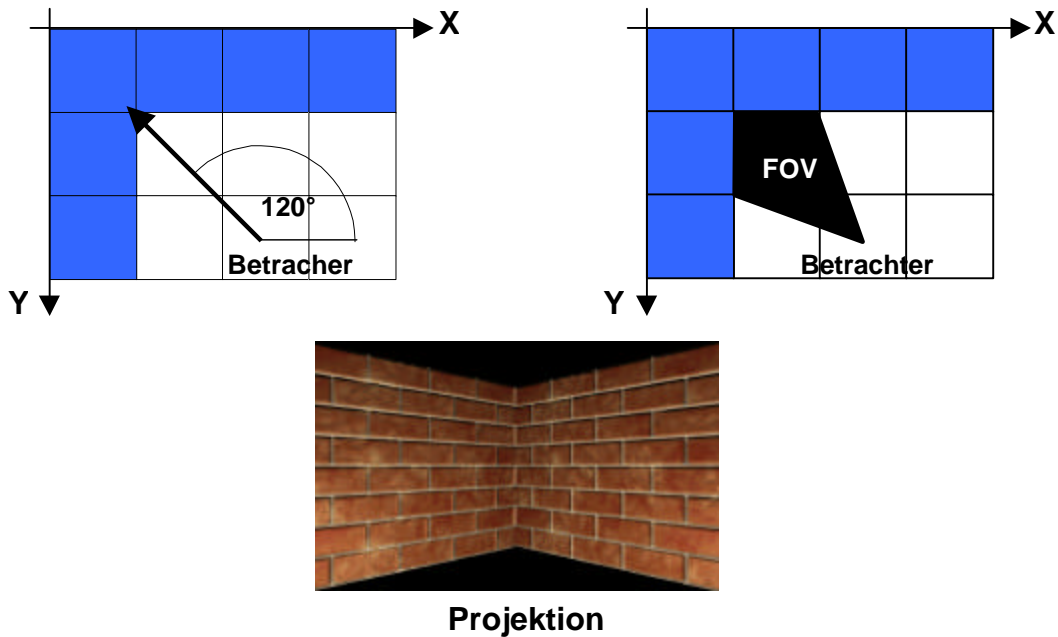
$$\begin{aligned}\text{Kartex} &= \text{Einheitx} \setminus 64 \\ \text{KarteY} &= \text{Einheity} \setminus 64\end{aligned}$$

Wie beim Tile-Scrolling führen wir eine Integer-Division durch, da beide Koordinatensysteme nur mit ganzzahligen Werten arbeiten. Formen wir die obigen Gleichungen um, erhalten wir für die inverse Berechnung diese Formeln:

$$\begin{aligned}\text{Einheitx} &= \text{Kartex} * 64 \\ \text{Einheity} &= \text{KarteY} * 64\end{aligned}$$

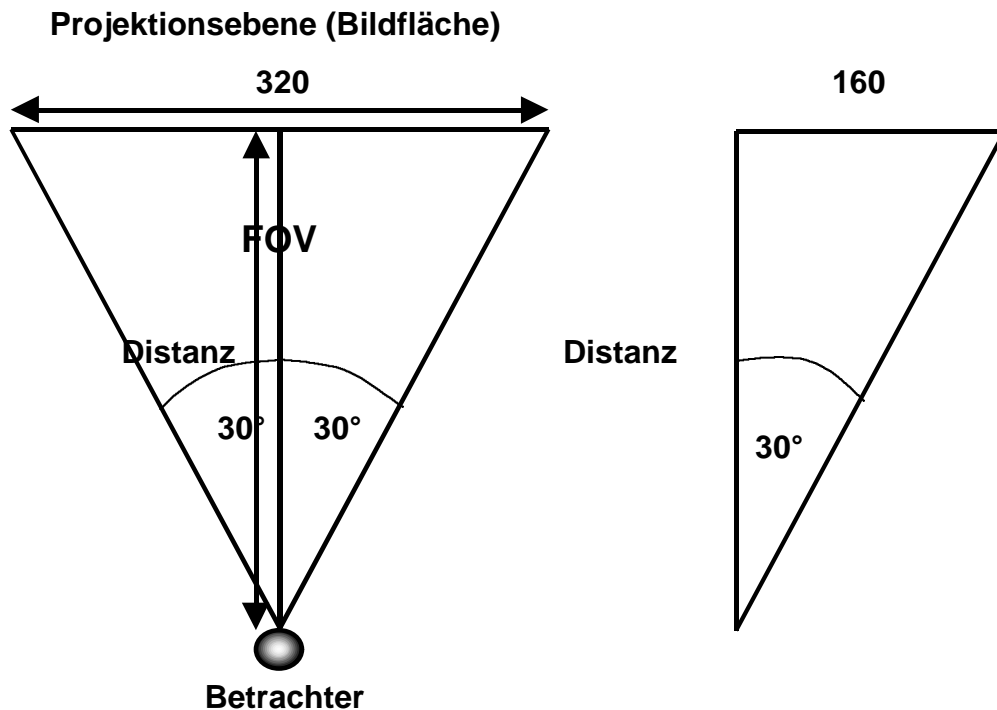
Im unten dargestellten Beispiel positionieren wir den Betrachter an den Einheitskoordinaten (160,160) bzw. an den Kartenkoordinaten (2,2). Der Blickwinkel beträgt 120°, der Winkel des FOV 60°.





*Abb.30 Konkretes Beispiel einer Projektion des Sichtfeldes*

Als Projektionsebene wird der Bildschirm herangezogen. Auf ihm wird das vom Betrachter wahrgenommene Bild der Welt projiziert. Die Dimensionen der Projektionsebene entsprechen der Auflösung des gewählten Bildschirmmodus, im Fall des Modus 13h 320x200 Pixel. Aus den Dimensionen der Projektionsebene, sowie dem Wert des FOV können wir die Distanz des Betrachters zur Ebene errechnen.



*Abb.31 Berechnung der Distanz vom Betrachter zur Projektionsebene*

In der oben dargestellten Abbildung können wir ein rechteckiges Dreieck erkennen. Mit Hilfe der Trigonometrie lässt sich so die gesuchte Größe Distanz errechnen.

$$\text{Distanz} = 160 / \tan(30^\circ) = 277$$

Unsere Projektionsebene, d.h. der Bildschirm, setzt sich aus 320 Spalten zusammen. Für jede dieser Spalten wird ein Strahl vom Betrachter aus über die Karte geschickt. Trifft ein Strahl auf eine Wand, können wir aus der Entfernung vom Betrachter zur Wand und den restlichen Projektionsfaktoren die projizierte Höhe des getroffenen Teils der Wand am Bildschirm errechnen. Dieser Teil der Wand wird dann in der Spalte des ausgesendeten Strahls als eine vertikale Linie gezeichnet. Der Vorgang wird für alle 320 Spalten des Bildschirms wiederholt, was bedeutet, dass die Szene aus 320 vertikalen Linien aufgebaut wird, die in ihrer Länge je nach Entfernung zum getroffenen Wandteil variieren. Je weiter entfernt der getroffene Wandteil ist, desto kleiner ist seine projizierte Höhe am Bildschirm.

Die Strahlen haben einen bestimmten Winkelabstand zueinander, der über die folgende Gleichung berechnet wird:

$$\text{Winkelabstand} = \text{FOV} / \text{Anzahl der Strahlen} = (60 / 320)^\circ$$

Diesen Wert benötigen wir später um die Winkel der ausgesendeten Strahlen inkrementell errechnen zu können.

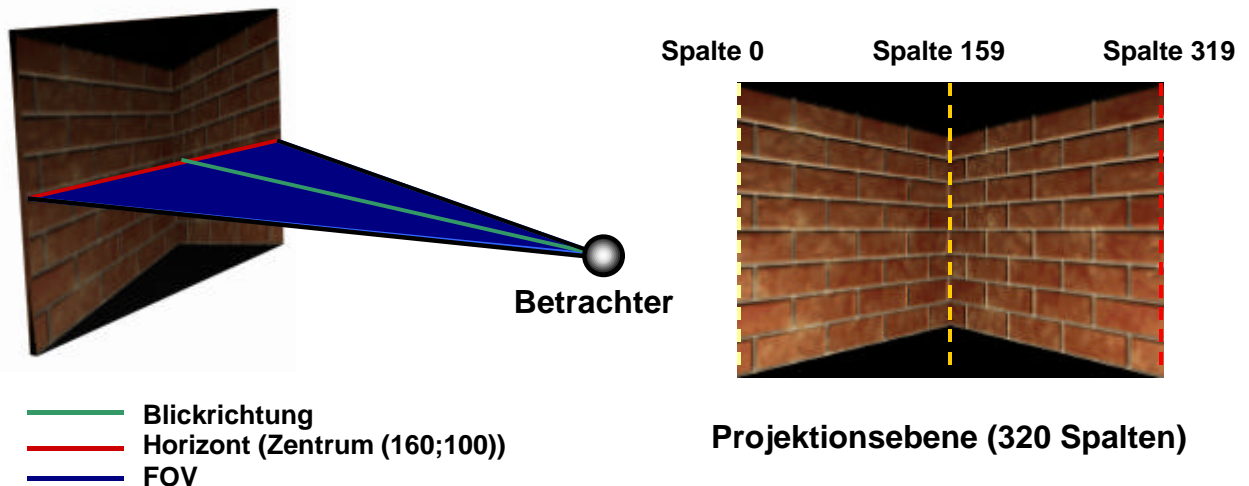


Abb.32 Der Bildschirm als Projektionsebene, in Spalten aufgeteilt. Für jede Spalte wird ein Strahl ausgesendet

Damit haben wir, abgesehen von den Distanzen zu den getroffenen Wandteilen, alle Faktoren beisammen.

- Dimension der Projektionsebene = 320x200 Einheiten
- Zentrum der Projektionsebene (160;100)
- Abstand des Betrachters zur Projektionsebene = 277 Einheiten
- Winkelabstand zwischen den einzelnen Strahlen =  $(60 / 320)^\circ$

Die Eigenschaften des Beobachters (Position, Blickrichtung), die vorher als Projektionsfaktoren genannt wurden, werden zur Berechnung der Distanzen verwendet, was der Grund dafür ist, dass sie hier nicht mehr angeführt wurden.

Um die Szene zu zeichnen, senden wir 320 Spalten aus, berechnen die Distanzen zu den von ihnen getroffenen Wandteilen und projizieren die Höhen dieser Teile spaltenweise in Form von vertikalen Linien auf den Bildschirm. Dieser Vorgang geschieht innerhalb einer Schleife, was der folgende Pseudocode illustrieren soll:

1. Subtrahiere vom Blickwinkel 30 Grad um den Winkel des ersten Strahls durch die Spalte 0 zu erlangen.
2. Beginne bei der Spalte 0:
  - A) Sende den Strahl aus
  - B) Verfolge den Strahl bis er eine Wand trifft
  - C) Berechne die Distanz zum Wandteil, und zeichne die über die Projektion erhaltene Höhe dieses Teils in der aktuellen Spalte durch eine vertikale Linie ein
3. Errechne den Winkel des nächsten Strahls in dem zum Winkel des vorhergehenden Strahls der Winkelabstand addiert wird.
4. Wiederhole Schritt zwei und drei für alle Spalten.

Anstatt bei Schritt 2A jedes Einheitenkoordinatenpaar zu prüfen, werden nur die Schnittpunkte des aktuellen Strahls mit den Grenzen der Felder der Karte auf einen Treffer mit einer Wand geprüft. Betrachten sie dazu das folgende Beispiel:

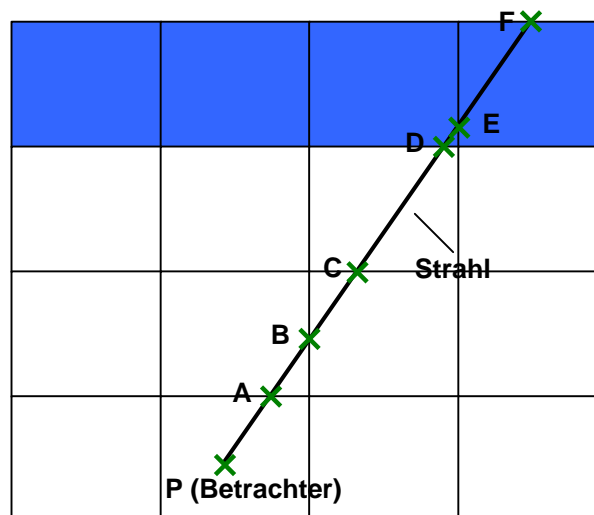


Abb.33 Der vom Betrachter ausgesandte Strahl schneidet die Feldgrenzen in den Punkten A, B, C, D, E und F.

Es reicht vollkommen aus, nur die Punkte A, B, C, D, E und F auf einen Treffer zu prüfen, da die Grenzen der Felder gleichzeitig die sichtbaren Flächen der Wände darstellen.

Die Suche nach den Schnittpunkten wird in zwei Prozesse aufgeteilt, der eine für die horizontalen Schnittpunkte (A, C, E), der andere für die

vertikalen ( B, D, F). Wird während der Ausführung dieser beiden Teilaufgaben festgestellt, dass der Strahl eine Wand getroffen hat, wird die Suche abgebrochen und die Distanz zum Schnittpunkt errechnet. Treffen beide Suchen auf einen Schnittpunkt, wird die kürzere Distanz zwischen dem Betrachter und den Schnittpunkten gewählt.

Widmen wir uns zuerst der Suche nach horizontalen Schnittpunkten.

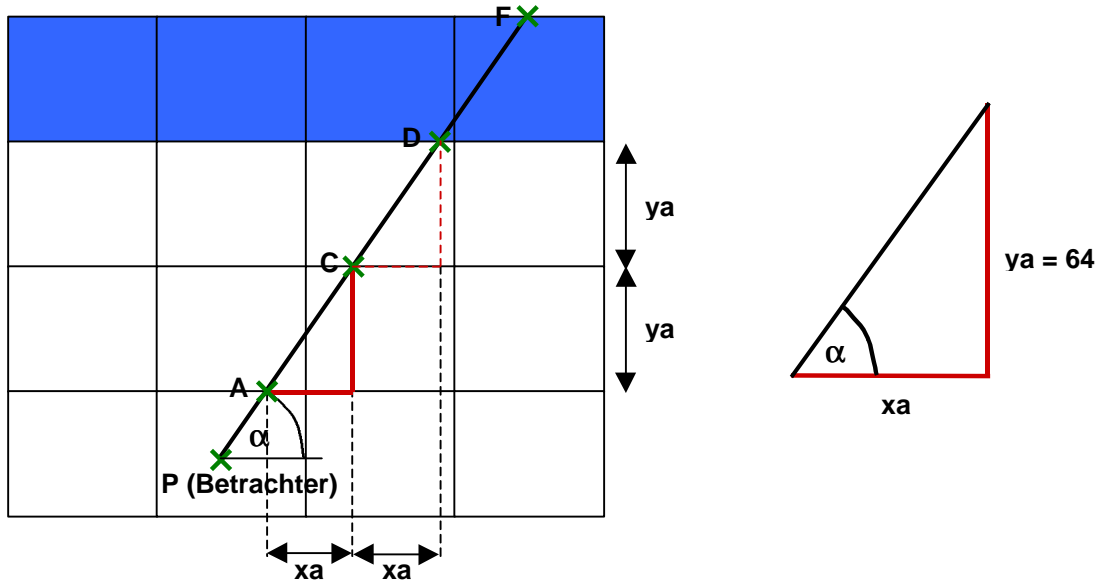
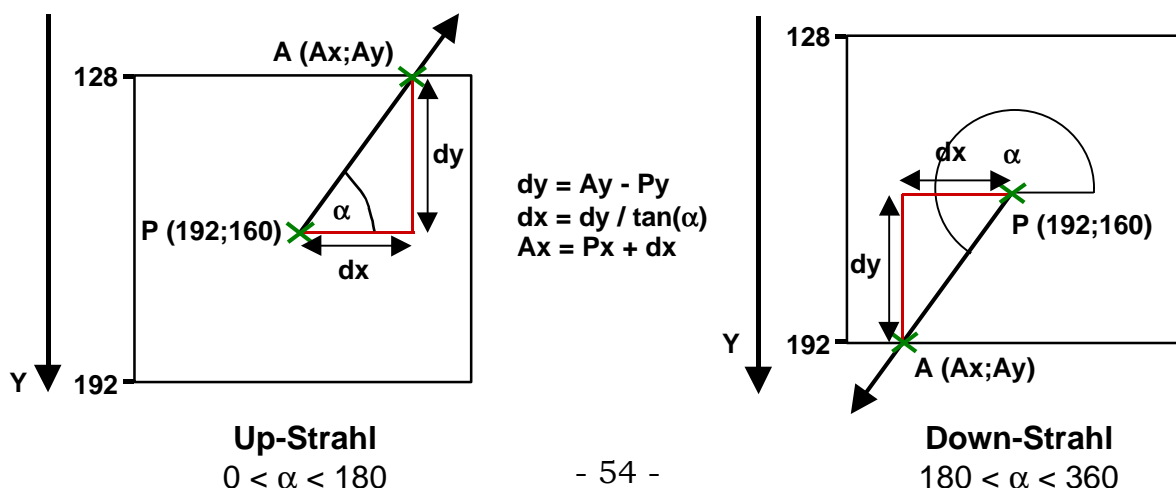


Abb.34 Die Schnittpunkte eines Strahls mit den horizontalen Grenzen der Felder.

Aus der Abbildung geht hervor, dass ab dem ersten Schnittpunkt A alle weiteren Schnittpunkte inkrementell errechnet werden können. Zu den Koordinaten des vorhergehenden Punktes werden einfach die beiden Größen xa und ya des Steigungsdreiecks des Strahl addiert. Zuvor müssen aber die Koordinaten des Punktes A, sowie der Wert xa berechnet werden. xa kann wieder über die Trigonometrie kalkuliert werden.

$$xa = ya / \tan(\alpha) = \pm 64 / \tan(\alpha)$$

Wurde der Strahl in einem Winkel zwischen 0 und 180 Grad ausgesendet, ist ya negativ. Für einen Winkel zwischen 180 und 360 Grad besitzt ya ein positives Vorzeichen. Damit fehlen nur mehr die Koordinaten des Punktes A, um die Suche zu beginnen. Die Berechnung seiner Koordinaten ist wieder abhängig davon, in welcher Richtung der Strahl ausgesendet wird.



Zuerst wird die y-Koordinate des Punktes A berechnet. Im Fall eines Up-Strahls entspricht diese der y-Koordinate der horizontalen Feldgrenze unterhalb des Punktes P. Über eine Integer-Division und eine darauffolgende Multiplikation jeweils mit dem Wert 64, können wir die y-Koordinate des Punktes P „abrunden“ und erhalten so die y-Koordinate des Punktes A.

$$A_y = P_y \setminus 64 * 64 - 1$$

Eins wird subtrahiert, um A als Teil des unteren Feldes auszuweisen. Darauf basierend können wir die Gleichung für die y-Koordinaten des Punktes A eines Down-Strahls aufstellen. Wir addieren einfach die Länge eines Feldes (64 Einheiten) zur abgerundeten y-Koordinate des Punktes P. Damit springen wir von der unterhalb von P liegenden horizontalen Feldgrenze auf die über P liegende.

$$A_y = P_y \setminus 64 * 64 + 64$$

Die x-Koordinate erhalten wir wieder über einfache Trigonometrie:

$$A_x = P_x + dy / \tan(\alpha) = P_x + (P_y - A_y) / \tan(\alpha)$$

Hier der Pseudocode zur Suche nach den horizontalen Schnittpunkten:

1. Punkt A berechnen
  - Up-Strahl:  $A_y = P_y \setminus 64 * 64 - 1$
  - Down-Strahl:  $A_y = P_y \setminus 64 * 64 + 64$
  - $A_x = P_x + (P_y - A_y) / \tan(\alpha)$
2. ya initialisieren
  - Up-Strahl:  $ya = -64$
  - Down-Strahl:  $ya = 64$
3. xa berechnen
  - $xa = ya / \tan(\alpha)$
4. Prüfen ob sich an den Kartenkoordinaten des errechneten Schnittpunktes eine Mauer befindet. Trifft das zu stoppe die Suche und errechne die Distanz zum Schnittpunkt. Sonst bei Punkt 5 fortfahren.
  - Kartenkoordinatenx = Schnittpunktx \ 64
  - Kartenkoordinateny = Schnittpunkty \ 64
5. Nächsten Schnittpunkt inkrementel berechnen
  - Schnittpunktx = letzter Schnittpunktx + xa
  - Schnittpunkty = letzter Schnittpunkty + ya
6. Schritt 4 und 5 solange wiederholen bis bei Schritt vier abgebrochen wird oder der Strahl die Karte verlässt, ohne eine Wand zu treffen.

Die Suche nach vertikalen Schnittpunkten funktioniert ganz gleich, jedoch ist die Reihenfolge der Berechnung anders.

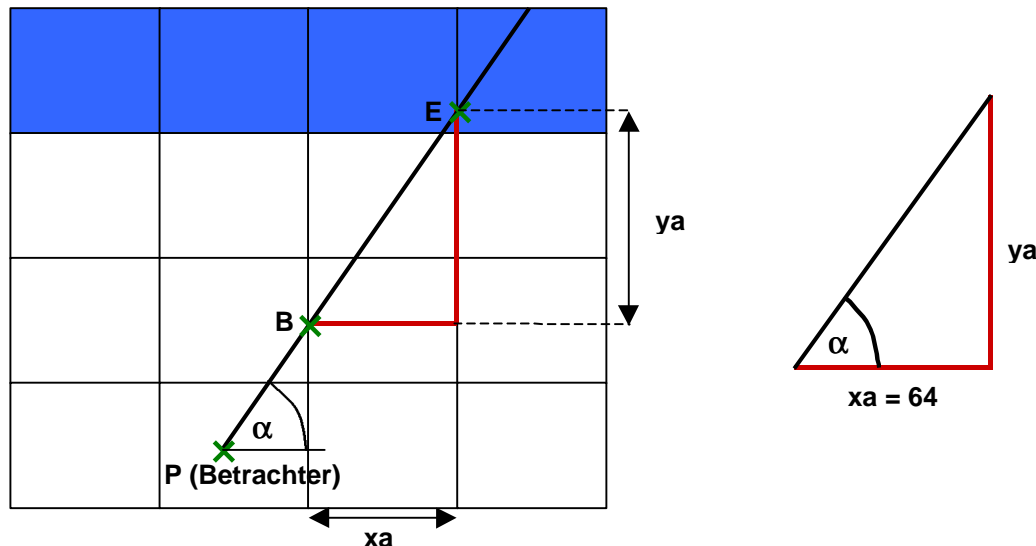


Abb.36 Die Schnittpunkte eines Strahls mit den vertikalen Grenzen der Felder.

Wieder können alle Schnittpunkte inkrementell errechnet werden. Im Gegensatz zu vorher muss nun das Inkrement  $ya$  kalkuliert werden:

$$ya = xa * \tan(\alpha) = \pm 64 * \tan(\alpha)$$

Abhängig davon, ob der Strahl nach rechts, d.h. sein Winkel zwischen 90 und 270 Grad liegt, oder nach links ( $0 < \alpha < 90$ , bzw.  $270 < \alpha < 360$ ) zeigt, besitzt  $xa$  ein positives (Right-Strahl) oder negatives (Left-Strahl) Vorzeichen. Der erste vertikale Schnittpunkt B muss wie der erste horizontale Schnittpunkt A per Hand berechnet werden. Abhängig von der Richtung des Strahls ergeben sich folgende Gleichungen für die x-Koordinate des ersten Schnittpunktes:

$$\begin{aligned} \text{Left-Strahl: } Bx &= Px \setminus 64 * 64 - 1 \\ \text{Right-Strahl: } Bx &= Px \setminus 64 * 64 + 64 \end{aligned}$$

Wir runden also die x-Koordinate des Punktes P auf die nächstgelegenen vertikalen Feldgrenzen auf, bzw. ab. Die y-Koordinate erhalten wir über diese Gleichung:

$$By = Py + (Px - Bx) * \tan(\alpha)$$

Die Pseudocodes der beiden Suchalgorithmen unterscheiden sich nur in der Berechnung der Inkremente  $xa$  und  $ya$ , sowie der Koordinaten des ersten Schnittpunktes. Hier der Pseudocode für die vertikale Suche:

1. Punkt B errechnen
  - Left-Strahl:  $Bx = Px \setminus 64 * 64 - 1$
  - Right-Strahl:  $Bx = Px \setminus 64 * 64 + 64$
  - $By = Py + (Px - Bx) * \tan(\alpha)$
2.  $xa$  initialisieren
  - Left-Strahl:  $xa = -64$
  - Right-Strahl:  $ya = 64$
3.  $ya$  berechnen

- ```
ya = xa * TAN(ALPHA)
```
4. Prüfen, ob sich an den Kartenkoordinaten des errechneten Schnittpunktes eine Mauer befindet. Trifft das zu, stoppe die Suche und errechne die Distanz zum Schnittpunkt. Sonst bei Punkt 5 fortfahren.  
 Kartenkoordinatenx = Schnittpunktx \ 64  
 Kartenkoordinateny = Schnittpunkty \ 64
  5. Nächsten Schnittpunkt inkrementel berechnen  
 Schnittpunktx = letzter Schnittpunktx + xa  
 Schnittpunkty = letzter Schnittpunkty + ya
  6. Schritt 4 und 5 solange wiederholen bis bei Schritt vier abgebrochen wird oder der Strahl die Karte verläßt, ohne eine Wand zu treffen.

Zur Berechnung der Distanz vom Betrachter zum kalkulierten Schnittpunkt stehen mehrere Methoden zur Verfügung:

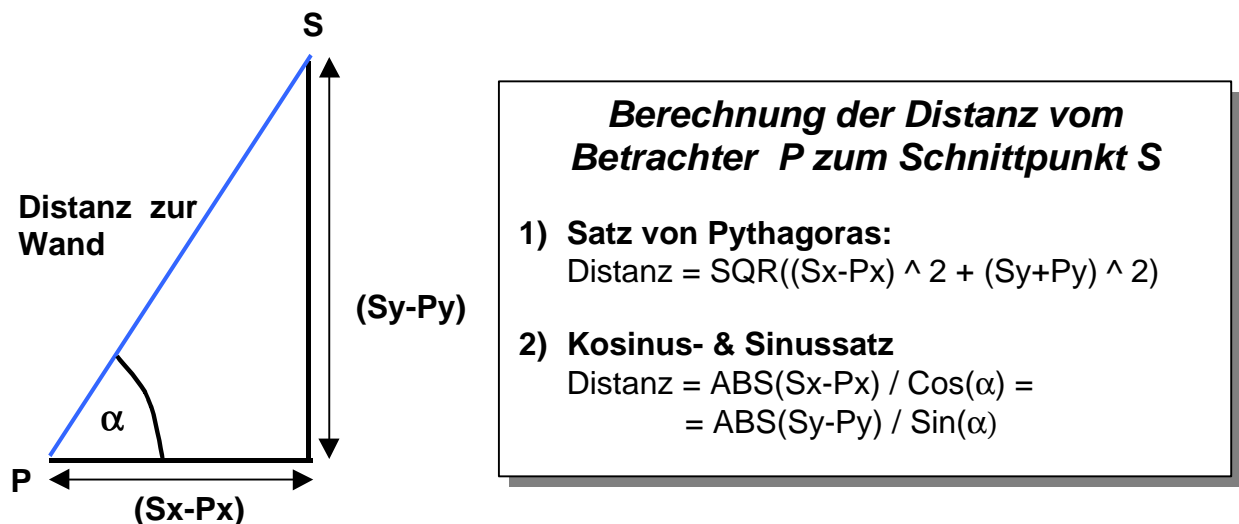


Abb.37 Berechnung der Distanz vom Betrachter zum Schnittpunkt des Strahls mit der Wand.

Die trigonometrische Methode ist weniger rechenaufwendig und sollte aus diesem Grund bevorzugt werden.

Widmen wir uns der letzten Hürde, dem Projizieren und Zeichnen der getroffenen Wandteile. Da wir nun auch die Distanzen errechnen können, ist es uns jetzt möglich, den Projektionsvorgang zu komplettieren.

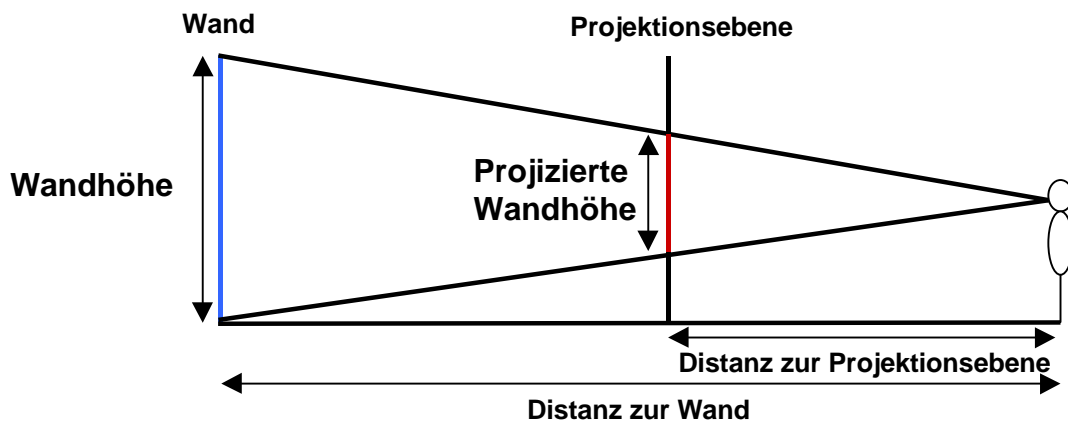


Abb.38 Projektion der Wandhöhe

Da wir es hier mit ähnlichen Dreiecken zu tun haben, können wir den Strahlensatz anwenden.

$$\frac{\text{Projizierte Wandhöhe}}{\text{Distanz zur Projektionsebene}} = \frac{\text{Wandhöhe}}{\text{Distanz zur Wand}}$$

$$\text{Projizierte Wandhöhe} = \frac{\text{Wandhöhe}}{\text{Distanz zur Wand}} * \text{Distanz zur Projektionsebene}$$

$$\text{Projizierte Wandhöhe} = \frac{64 * 277}{\text{Distanz zur Wand}}$$

Nehmen wir an, der Strahl der Spalte 200 trifft eine Wand 330 Einheiten vom Betrachter entfernt. Die projizierte Höhe des getroffenen Wandteiles beträgt nach der obigen Formel  $(64 * 277) / 330 = 54$  Pixel. Da sich das Zentrum der Projektionsebene an der y-Koordinate 100 befindet, sollte dort der Mittelpunkt der gezeichneten vertikalen Linie sein. Das heißt wir zeichnen die Linie von der y-Koordinate  $100 - 27 = 73$  ( $54 / 2 = 27$ ) zur y-Koordinate  $100 + 27 = 127$  an der x-Koordinate 200. So gehen Sie für alle 320 Spalten am Bildschirm vor.

Zum Schluß dieses Kapitels möchte ich Sie noch auf den Fish-Bowl-Effekt aufmerksam machen. Dieser resultiert aus der Tatsache, dass die Berechnung der Distanzen zwar richtig, aber nicht deckungsgleich mit unserer Wahrnehmung ist.

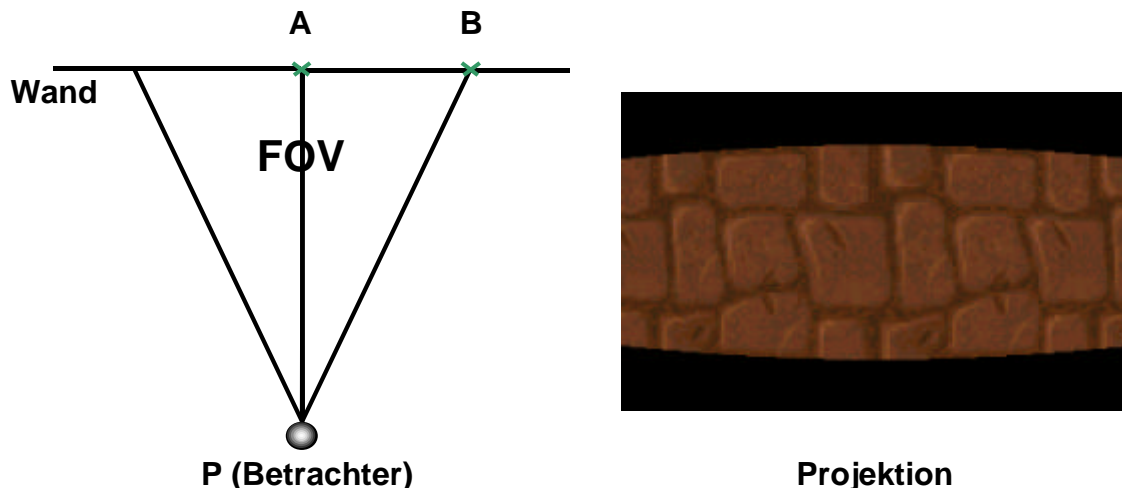


Abb.39 Der Fish-Bowl-Effekt

Die Distanz zwischen den Punkten P und B ist größer als die zwischen P und A, was zur Folge hat, dass die Höhe der Mauer in der äußeren rechten Spalte kleiner ist als die in der Mitte des Bildschirms. Aus dem täglichen Leben wissen wir aber, dass eine Mauer, die wir frontal betrachten, an allen Stellen die selbe Höhe besitzt. Prinzipiell würde unsere Wahrnehmung uns das oben dargestellte Bild liefern, wäre da nicht die Krümmung unseres Auges, sowie



unser Gehirn, das eventuelle Ungereimtheiten ausgleicht. Da der Bildschirm aber flach ist, müssen wir diesen Effekt mathematisch kompensieren. Korrigieren Sie die errechneten Distanzen einfach mit Hilfe dieser Formel:

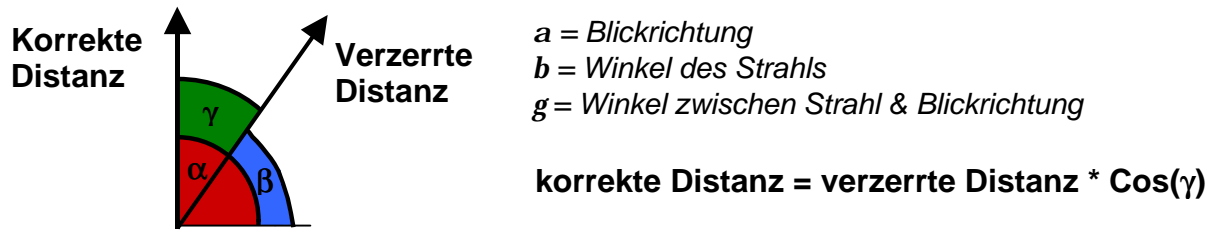


Abb.40 Korrektur des Fish-Bowl-Effekt

Auf der Anhang-CD befinden sich zwei Ray-Casting-Engines. Um die BASIC-Version so verständlich wie möglich zu halten, wurde sie nicht optimiert, was natürlich bedeutet, dass sie relativ langsam arbeitet. Die Visual-Basic-Ausführung hingegen arbeitet sehr schnell, wird aber für einen Einsteiger schwer zu durchschauen sein. Außerdem bietet sie neben einer höheren Geschwindigkeit auch Texture-Mapping. Aus Platzgründen muss ich leider darauf verzichten, dieses eigentlich sehr wichtige Gebiet der 3d-Computergrafik zu erklären. Grob gesagt tapeziert man damit die Oberfläche eines dreidimensionalen Objekts, in unserem Fall die der Wand, mit einem Muster oder besser gesagt mit einer Grafik, um diese realistischer erscheinen zu lassen. Die oben dargestellten Projektionen verwenden diese Technik ohne die sämtliche Wände einfärbig wären.

**Arbeitsprotokoll**  
**von**  
**Mario Zechner, 8.B BG Rein, 2001/2002**  
**für**  
**Fachbereichsarbeit aus Informatik**  
**Mag. Beate Peichler**

- Herbst 2000:** **Kontaktaufnahme mit dem Fachprofessor.**  
**Grobes Be**  
**-sprechen der Möglichkeiten einer**  
**Fachbereichsarbeit**
- Jänner 2001:** Abgrenzung des Themengebietes auf den Bereich der  
Spielgrafikprogrammierung
- März 2001:** Vorstellen einzelner Algorithmen (Scrolling, Ray-  
Casting)
- 2. Mai 2001:** Vorbesprechung über das Themengebiet sowie  
dessen Umfang.
- 27. Oktober 2001:** Besprechung der Disposition
- 3. Dezember 2001:** Vorlage und Besprechung der bisherigen Ergebnisse.  
Fixierung der verwendeten Programmiersprachen.
- 13. Dezember 2001:** Suchen und Vorlegen von Belegstellen
- 14. Dezember 2001:** Suchen von Belegstellen
- 14. Dezember 2001:** Besprechung der weiteren Vorgehensweise
- 17. Dezember 2001:** Suchen von Belegstellen
- 20. Dezember 2001:** Erstellen der Beispielprogramme „DRAWPIX.BAS“,  
„DRAWTRI.BAS“ und „DRAWTRIF.BAS“
- 27. Dezember 2001:** Erstellen des Beispielprogramms „PALFADE.BAS“
- 1. Jänner 2002:** Erstellen des Beispielprogramms „DRAWLINE.BAS“
- 5. Jänner 2002:** Auszugsweise Vorlage der ausgearbeiteten Kapitel.  
Erstellen des Beispielprogramms „GETBLIT.BAS“
- 6. Jänner 2002:** Erstellen der Beispielprogramme „CLEARSCR.BAS“  
und „GETPIXEL.BAS“.

- 9. Jänner 2002:** Erstellen der Beispielprogramme „DOUBLEBF.BAS“ und „DB.ASM“ sowie der Bibliothek „DB.ASM“
- 14. Jänner 2002:** Suche von Belegstellen
- 16. Jänner 2002:** Erstellen der Beispielprogramme „TBTSCR.BAS“ und „PBPSR.BAS“
- 18. Jänner 2002:** Erstellen der Beispielprogramme „SPRITES.BAS“ und „SPRANIM.BAS“
- 23. Jänner 2002:** Eingrenzen des Kapitels 4.2. Texture-Mapping, Z-Buffering und Binary Space Portioning werden aufgrund von Platzmangel vernachlässigt.
- 2. Februar 2002:** Erstellen der Beispielprogramme „RAYCAST.BAS“ und „VBRAY.VBP“.
- 8. Februar 2002:** Demonstration der bisher ausgearbeiteten Beispielprogramme sowie der CD - Menüführung. Prüfung auf Verständlichkeit der Programmcodes. Behebung von Formatierungsproblemen.
- 25. Februar 2002:** Einreichung der Fachbereichsarbeit

**Unterschrift des Kandidaten:**

## Verzeichnis der Belegstellen:

- Autor unbekannt: **"VGA Programming"**  
<http://www.maths.tcd.ie/~nryan/demos/vgaprog.html>
- Brian Bartels: **"The VGA palette, QBasic, and you"**  
<http://qbtuts.qb45.com/graphics/palette.txt>
- DarkDread: **"RPGs in Qbasic Tutorial"**  
<http://www.geocities.com/SiliconValley/Pines/1732/tut.htm>
- Keith Dowd: **"Playing with the Palette"**  
<http://www.gamedev.net/reference/articles/article317.asp>
- Kasper Faubery: **"Raycasting - Wolfenstein Style"**  
<http://www.programmersheaven.com/zone10/cat183/6287.htm>
- Kevin Matz: **"Introduction to VGA Graphics Programming using Mode 13h"**  
<http://www.comprenica.com/atrevida/atrtut07.html>
- Kevin Matz : **"VGA Mode 13h Graphics Primitives, Part 1"**  
<http://www.comprenica.com/atrevida/atrtut08.html>
- Kevin Matz: **"Basic Animation Techniques for Mode 13h"**  
<http://www.comprenica.com/atrevida/atrtut10.html>
- Grant Smith: **"Palette Changing"**  
<http://www.gamedev.net/reference/articles/article348.asp>
- Grant Smith: **"Virtual Screens",**  
<http://www.gamedev.net/reference/articles/article350.asp>
- Grant Smith: **"Animation"**  
<http://www.gamedev.net/reference/articles/article353.asp>
- Michael Tischer: **„PC intern 4“**. Düsseldorf: Data Becker 1994
- Nathan Vegdahl: **"The Video Mode 13h"**  
<http://www.gamedev.net/reference/articles/article315.asp>
- Zkman: **"Palette Tricks"**  
<http://qbtuts.qb45.com/graphics/qbtmpalette.htm>