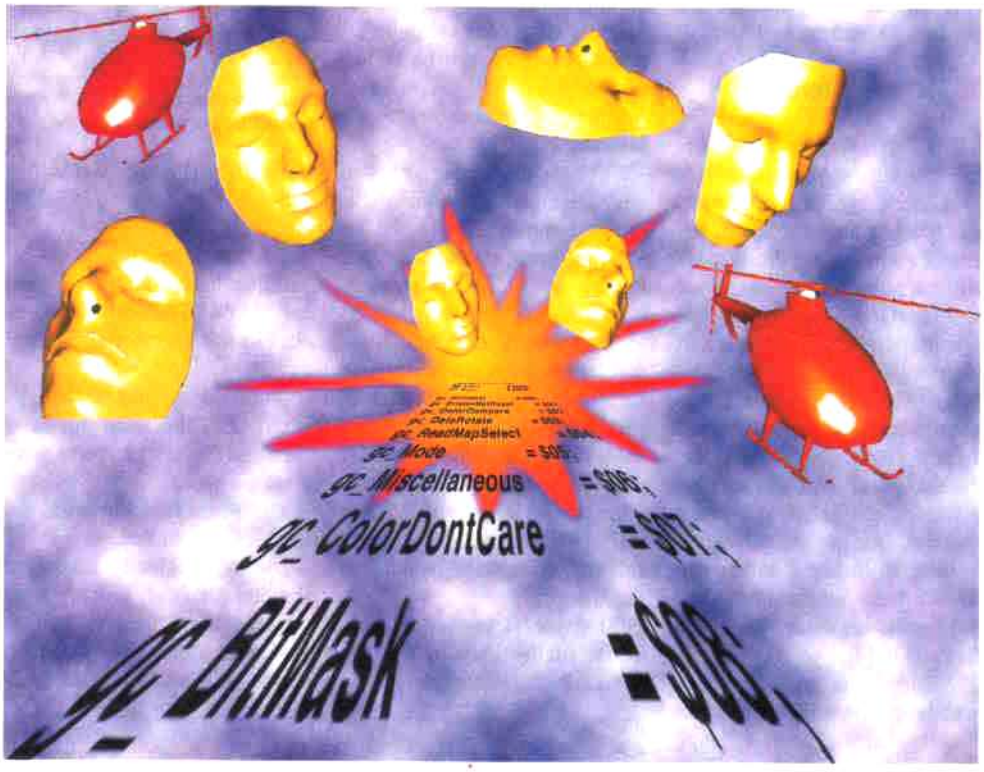
Hoffmann, J. E., “Mit Gouraud-Shading zur 3D Animation”,
MC extra (www.pc-magazin.de), 03/1995, 34-40.

Gouraud-Shading zur 3D-Animation

Täuschend echte Bewegungen

J.E. Hoffmann

Noch vor kurzem war die 3D-Computergrafik nur teuren Grafik-Workstations vorbehalten. Mittlerweile stellt die 3D-Vektorgrafik in Echtzeit auch für den PC kein Problem mehr dar. Schon mit einem 386DX-40 erzielen Sie beeindruckende Resultate. Allerdings nur, wenn Sie Ihren Computer bis zum letzten ausreizen.



Wem gefallen sie nicht, die tollen Computeranimationen in zahlreichen Spielen, wie beispielsweise in Kings Quest VII oder Wing Commander. Es ist dann bald wie Fernsehen auf dem Computer. In nicht mehr allzu langer Zeit werden bestimmt ganze Spielfilme auf dem PC produziert, wenn das Problem mit dem dafür notwendigen Speicher gelöst ist.

In diesem Beitrag soll Ihnen nun gezeigt werden, wie Sie selber tolle 3D-Animationen in Ihre Programme oder Präsentationen einbauen können. Dafür fallen natürlich sehr viele Berechnungen an. Deshalb stellt sich zuerst die Frage: „Benötigt man einen Coprozessor?“. Oder anders gefragt, benötigt man überhaupt Gleitkommaoperationen? Die Antwort lautet Nein.

Fixed Point Arithmetic

Bei der Fixed Point Arithmetic handelt es sich um einen ziemlich simplen Trick, der die Ausführungsgeschwindigkeit jedoch um ein vielfaches steigert. Sie verwenden eine Integer-Zahl, von der ein bestimmter Teil als Kommaanteil behandelt wird. Dazu wird die Gleitkommazahl mit einem konstanten Faktor multipliziert und gerundet. Aus der Gleitkommazahl ist nun eine Integerzahl geworden.

Wie Sie leicht nachprüfen können, sind Addition und Subtraktion ohne weiterers ausführbar. Bei Division und Multiplikation müssen Sie jedoch Korrekturen durchführen. Hierzu einige Beispiel-Rechnungen:

- Addition:
  1.2*100+1.5*100=2.7*100
- Subtraktion:
  1.2*100-1.5*100=0.3*100
- Multiplikation:
  1.2*100*1.5*100=1.8*100*100

Bei der Multiplikation müssen Sie also zum Schluß noch durch 100 dividieren!

- Division:
  1.2*100 / 1.5*100 = 0.8

Bei der Division müssen Sie noch mit 100 multiplizieren!

Da es nahezu egal ist, mit welchem Wert Sie multiplizieren/dividieren (nur die Genau-

igkeit hängt davon ab) liegt es nahe, keine 10er Potenz zu verwenden, sondern eine 2er Potenz. Das Multiplizieren/Dividieren können Sie so durch sehr viel schnellere Schiebeoperationen ersetzen.
256 (8 Bit Kommaanteil) würden sich hierfür anbieten. Im folgenden wird jedoch die Zahl 512 (9 Bit Kommaanteil) verwendet, um eine höhere Genauigkeit zu erhalten. Außerdem wird 32-Bit-Integer (LongInt Variablen) verwendet. Das gleiche Prinzip läßt sich auch mit 16 Bit verwirklichen (schneller aber kleinerer Zahlenbereich). Das Umrechnen einer Gleitkommazahl in eine Fixed-Point-Zahl erfolgt also nach der einfachen Formel:

```
fixed:=round(_real * $200)
```

und die Umkehrung entsprechend:

```
_real:=fixed / $200;
```

Da diese Umwandlungen sehr viel Zeit auf einem Rechner ohne Coprozessor benötigen, sollten Sie die Umwandlungen am besten vorberechnen, so daß sie zur Laufzeit bereits vorhanden ist.
In Assembler lassen sich Division und Multiplikation sehr leicht mit Hilfe der 32-Bit-Befehle des 386er koppeln.
Beide im folgenden vorgestellten Routinen sind für Turbo-Pascal beschrieben, lassen sich allerdings auch ohne Probleme unter jedem 16-Bit-C-Compiler nutzen. Als Assembler setzen Sie den TASM im Ideal-Modus-Syntax ein. (Alle hier auszugsweise vorgestellten Listings finden Sie vollständig auf der Databox der DOS International zu dieser Ausgabe.)

```
; function FixMul
   (A,B:longInt):LongInt;
         near; external;
  PROC FixMul NEAR
       pop    cx
       pop    ebx
       pop    eax
       imul   ebx
       shrd   eax,edx,9
       rol    eax,16
       mov    dx,ax
       rol    eax,16
       push   cx
       ret
  ENDP
```

Damit kein Stackframe eingerichtet werden muß, holt die Funktion zuerst die Rücksprungadresse vom Stack (Achtung: Prozedur ist Near und kann daher nicht in einer Unit deklariert werden!) und danach die beiden Faktoren. Anschließend wird die Multiplikation durchgeführt und das Ergebnis um 9 Bit nach rechts verschoben (= Division durch 512). Da Turbo-Pascal als 16-Bit-Compiler keine 32-Bit-Register unterstützt, müssen Sie den Rückgabewert in dx:ax übergeben.
Die Divisionsroutine sieht ähnlich aus:

```
; function FixDiv(A,B :
   LongInt):LongInt; near;
                external;
  PROC FixDiv NEAR
       pop    cx
       pop    ebx
       pop    eax

       cdq
       shld   edx,eax,9
       shl    eax,9
       idiv   ebx

       rol    eax,16
       mov    dx,ax
       rol    eax,16
       push   cx
       ret
  ENDP
```

Der Divident in *eax* wird nach edx:eax erweitert. Danach wird alles um 9 Bit nach links geschoben und die Division durchgeführt.
Außer der Division und der Multiplikation sind die trigonometrischen Funktionen wie Sinus und Cosinus von Bedeutung (sie werden für die Rotation von Punkten benötigt). Die Berechnung von Sinus- und Cosinuswerten anhand von Reihen ist sehr rechenzeitaufwendig, daher sollten Sie Tabellen benutzen. Dabei müssen Sie darauf achten, das 360° einer 2er Potenz entspricht. Dann kann eine AND-Verknüpfung verwendet werden (anstatt einer Division), um Mehrfachrotationen (0°=360°=720°) handhaben zu können.

```
; function FixSin(W :
  Integer) :LongInt; near;
                external;
  PROC FixSin NEAR
       pop    cx
       pop    bx
  $$Sin: and  bx,1FFh
       cmp    bx,256
       jge    @@21
       cmp    bx,128
       jng    @@11
       neg    bx
       add    bx,256
  @@11: shl   bx,1
    mov  ax,[bx+SinTable]
       cwd
       push   cx
       ret
  @@21: sub   bx,256
       cmp    bx,128
       jnge   @@22
       neg    bx
       add    bx,256
  @@22: shl   bx,1
   mov   ax,[bx+SinTable]
       neg    ax
       cwd
       push   cx
       ret
  ENDP
```

Damit die Sinustabelle nicht zu groß ist, enthält Sie nur Werte für 0°-90°. 360° entsprechen hier 512 „fixed". Außerdem wurden keine LongInts abgespeichert, sondern Words die mit *cwd* in einen LongInt in *dx:ax* umgewandelt werden.

```
  function FixCos(W:
   Integer) :LongInt; near;
                external;
PROC FixCos NEAR
       pop    cx
       pop    bx
       add    bx,128
       jmp    $$Sin
  ENDP
```

Die Cosinusroutine addiert einfach 128 (90) und springt dann in die Sinusrountine. Wie Sie sehen, sind die Berechnungen verhältnismäßig schnell ausgeführt.
Vektorobjekte werden in der Computergrafik durch Punkte beschrieben (als Punktemenge im dreidimensionalen Euklidischen Punktraum). Um ein Objekt zu bewegen, ist es so nur wichtig, diese Punkte zu verändern. Hier bieten sich Verfahren aus der Vektormathematik an.

Vektoren, Matrizen, Homogene Koordinaten

Mathematisch läßt sich die Bewegung eines Objektes durch lineare Transformationen beschreiben. Translation (Verschieben im Raum), Rotation (Drehung) und Skalierung.
Diese linearen Transformationen lassen sich sehr einfach durch geeignet definierte Matrizen beschreiben.
Die Punkte des Vektorobjektes müssen nur mit den entsprechenden Matrizen für Translation, Rotation und Skalierung multipliziert werden. Hierfür ist eine Matrixmultiplikation mit dem Punkt notwendig, da die Transformationsmatrizen zu einer Gesamtmatrix zusammengefaßt werden.
Für den dreidimensionalen Raum gibt es keine Transformationsmatrix der Größe 3*3. Durch die Einführung von sogenannten „Homogenen Koordinaten" können Sie alle drei linearen Transformationen durch eine einzige Matrix beschreiben. Hierbei wird ein Punkt nicht mehr durch einen dreispaltigen, sondern durch einen vierspaltigen Vektor beschrieben (Bild 1, Formel 1).
Die linearen Transformationen werden daher durch eine 4x4-Matrix beschrieben.
Die Komponente der vierten Dimension (w) wird auf 1 gesetzt, so daß unser Vektorobjekt einer Projektion auf die Ebene w=1 im vierdimensiona-

$$[1]\quad \vec{A} = \begin{bmatrix} a_x \\ a_y \\ a_z \\ w \end{bmatrix} \qquad [2]\quad \begin{bmatrix} a_x + t_x \\ a_y + t_y \\ a_z + t_z \\ w \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} a_x \\ a_y \\ a_z \\ 1 \end{bmatrix} = T \cdot \vec{A} \qquad [3]\quad \begin{bmatrix} a_x \cdot s_x \\ a_y \cdot s_y \\ a_z \cdot s_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} a_x \\ a_y \\ a_z \\ 1 \end{bmatrix} = S \cdot \vec{A}$$

$$[4]\quad R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad [5]\quad R_y = \begin{bmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad [6]\quad R_z = \begin{bmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$[7]\quad \vec{M} = \begin{bmatrix} \cos\beta \cdot \cos\gamma \cdot s_x & (\sin\alpha \cdot \sin\beta \cdot \cos\gamma + \cos\alpha \cdot \sin\gamma) \cdot s_x & (\cos\alpha \cdot \sin\beta \cdot \cos\gamma - \sin\alpha \cdot \sin\gamma) \cdot s_x & d_x \cdot s_x \\ \cos\beta \cdot \sin\gamma \cdot s_y & (\sin\alpha \cdot \sin\beta \cdot \sin\gamma + \cos\alpha - \cos\gamma) \cdot s_y & (\cos\alpha \cdot \sin\beta \cdot \sin\gamma - \cos\alpha \cdot \cos\gamma) \cdot s_y & d_y \cdot s_y \\ -\sin\beta \cdot s_z & \sin\alpha \cdot \cos\beta \cdot s_z & \cos\alpha \cdot \cos\beta \cdot s_z & d_z \cdot s_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Bild 1. Für die Transformation im dreidimensionalen Raum müssen Sie mit diesen Formeln umgehen können

len Raum entspricht. Eine Verschiebung des Vektors A kann nun durch eine Translationsmatrix beschrieben werden (Bild 1, Formel 2):
Die Skalierungsmatrix finden Sie in Bild 1, Formel 3.
Für die Rotation gibt es drei verschiedene Rotationsmatrizen – für jede Achse eine: x,y,z (Bild 1, Formeln 4 bis 6).
Wir haben Ihnen nun ein wenig Arbeit abgenommen und die Gesamtmatrix für Sie bestimmt. Wichtig ist die Reihenfolge mit der die Matrizen multipliziert werden, denn die einzelnen Transformationen werden genau in dieser Reihenfolge ausgeführt. Die Skalierung findet als letztes statt, um das Vektorobjekt einfach an verschiedene Bildschirmauflösungen anzupassen (Bild 1, Formel 7).
Eine weitere Berechnung die anfällt, ist die Umrechnung der 3D-Koordinaten in 2D-Koordinaten (Projektion). Dabei hilft der Strahlensatz der Mathematik (Bild 2, Formel 8).
Setzen Sie nun z1=1 (Blickwinkel=90°), so vereinfacht sich die Formel (Bild 2, Formel 9).

Die Pascal Version sieht folgendermaßen aus (Achtung! 9 Bit (=$200) Fixed Point Arithmetic):

```
xp:=160+(LongInt(V[x])*
$200) div V[z];
yp:=100-(LongInt(V[y])*
$200) div V[z];
```

Nachdem Sie nun wissen, wie Sie Objekte im Raum beschreiben, bewegen und das dreidimensionale Objekt in die zweidimensionale Punktkoordinaten des Bildschirm umrechnen, geht es weiter mit dem Füllen von Polygonen.
Hierbei sollen Sie die Polygone nicht mit einer einheitlichen Farbe füllen, weil dann ein Farbsprung zwischen den einzelnen Flächen entsteht, auch „Flat Shading" genannt. Es soll vielmehr eine weiche Farbverteilung erreicht werden. Eine weiche Farbverteilung ergibt sich dadurch, daß Sie jedem Punkt einen Normalvektor/ Farbwert zuweisen. Dann können Sie die Polygone mit Hilfe der „Phong"- oder „Gouraud"-Shading-Technik darstellen.
Beide Techniken sind Interpolationsverfahren. Beim Phong-Shading wird der Normalvektor des zu zeichnenden Punktes durch Interpolation bestimmt (und dann der Farbwert berechnet). Beim Gouraud-Shading werden nur die Farbwerte interpoliert.
Die Gouraud-Shading Methode braucht verhältnismäßig wenig Rechenaufwand. Ein großer Nachteil dieser Methode ist aber, daß Sie keine Schlaglichter im Inneren des Polygons darstellen können.
Für das Gouraud-Shading müssen Sie jedem Punkt einen Farbwert zuordnen. Dazu ist aber Voraussetzung, daß jeder Punkt einen Normalvektor besitzt (dieser Normalvektor stellt den Mittelwert der benachbarten Flächennormalvektoren dar). Anschließend werden zuerst die Farbwerte für die Kanten des Polygons interpoliert und dann jeweils die Farbwerte für jeden einzelnen Punkt. Der Vorgang wird an einem Beispiel demonstriert. Jetzt soll jedoch erstmal geklärt werden, inwiefern ein Polygon zweckmäßig gezeichnet wird. Hier die Definition einiger Strukturen:

```
STRUC TLine
  x     dw ?
  ddy   dw ?
  ddx   dw ?
  icx   dw ?
  error dw ?
  yTo   dw ?
  xTo   dw ?
  Col   dd ?
  icC   dd ?
ENDS
STRUC TEntry
  x     dw ?
  Col   dd ?
ENDS
```

_VirtualVSeg enthält die Segmentadresse des Videospeichers. Diese kann sowohl 0A000h betragen als auch die Segmentadresse eines eigenen Videobuffers enthalten. *xSize* gibt die x-Breite des Bildschirms in Pixeln an.

```
EXTRN _VirtualVSeg :Word
EXTRN xSize        :Word
  yi     dw ?
  LL     TLine ?
  RL     TLine ?
  yMin   dw ?
  yMax   dw ?
  min    dw ?
  L      dw ?
```

```
R       dw ?
K       dw ?
k1      TEntry ?
k2      TEntry ?
DeltaX  dw ?
Cic     dd ?
```

Der Routine werden in *P* die Bildschirmkoordinaten des Polygons übergeben – und zwar als „array [0..Count-1,x..y] of Integer". In *C* werden die Farbwerte als „array [0..Count-1] of Byte" übergeben. *Count* gibt die Anzahl der Eckpunkte des Polygons an.

```
Cx yShadedPoly(P:PPoints;
 C :PBytes; Count :Word);
  PROC CxyShadedPoly FAR
       ARG Count:Word,
    C:DWord,
    P:DWord = @@return
    enter  0,0
```

Zuerst wird der höchste und niedrigste y-Wert bestimmt und in *yMin* und *yMax* gespeichert. Gleichzeit wird die Position im Koordinaten-Array von *yMin* in *min* gemerkt.

```
        cld
        mov     [min],-1
        mov     [yMin],200
        mov     [yMax],-1
        xor     cx,cx
        les     si,[P]
        mov     bx,si
        inc     si
        inc     si
@@11:   mov     ax,[es:si]
        cmp     ax,[yMin]
        jnl     @@12
        mov     [yMin],ax
        mov     [min],cx
@@12:   mov     ax,[es:si]
        cmp     ax,[yMax]
        jnge    @@13
        mov     [yMax],ax
@@13:   add     si,4
        inc     cx
        cmp     cx,[Count]
        jb      @@11
```

Ist *yMin* größer als 199 oder *yMax* kleiner Null so ist das Polygon nicht auf dem Bildschirm sichtbar.

```
        cmp     [yMin],199
        jg      @@71
        cmp     [yMax],0
        jl      @@71
```

yi gibt die aktuelle y-Position der Polygonroutine an. Die Polygonroutine geht dann alle Zeilen von *yMin* bis *yMax* durch und zieht eine entsprechende horizontale Line. Dazu wird eine Rechte und eine Linke Linie berechnet, welche Start- und Endpunkt der horizontalen Linie bestimmen.
Die zu zeichnenden Polygone müssen konvex sein. Ein Polygon ist dann konvex, wenn Sie an jeder beliebigen Stelle eine Linie hindurchlegen können, ohne daß diese Linie das Polygon mehr als zweimal schneidet. Ein Dreieck ist immer konvex. Genauso ein Viereck. Bei der Definition der Polygone ihres Vektorobjektes müssen Sie auch beachten, daß zwei konvexe Polygone schneller zu zeichnen sind, als ein konkaves Polygon. Im folgenden Code-Abschnitt werden die Startwerte für die Polygonroutine geladen.

```
        mov     ax,[yMin]
        mov     [yi],ax
```

L gibt die Position der linken Linie im Koordinaten-Array und *R* die Position der rechten Linie. Beide werden mit dem Anfangspunkt geladen. LL enhält die Daten der „L"inken „L"inie und RL die der „R"echte „L"inie. Die x-Werte beider Linien werden jeweils mit dem x-Startwert geladen.

```
        mov     ax,[min]
        mov     [L],ax
        mov     [R],ax
        shl     ax,2
        add     bx,ax
        mov     ax,[es:bx]
        mov     [LL.x],ax
        mov     [RL.x],ax
```

InitLeft und *InitRight* laden *LL* beziehungsweise *RL* mit den Werten für den nächsten Linienabschnitt.
Zuerst wird der x-Startwert der Linie gelesen und *L* eine Position weiter gestellt. Dafür wird *L* um Eins dekrementiert. Ist *L* gleich Null, wird *L* Count zugewiesen und dann um einen dekrementiert.

```
        mov     bx,[L]
        mov     [K],bx
        shl     bx,2
        mov     ax,[es:si+bx]
        mov     [LL.x],ax
        mov     bx,[L]
        cmp     bx,0
        jne     @@12
        mov     bx,[Count]
@@12:   dec     bx
        mov     [L],bx
```

Ist *L* um Eins weitergestellt worden, wird der y-Endwert (yTo) der Linie geladen und die Differenz zwischen *yTo* und *yi* (aktuelle y-Position) in *ddy* gespeichert. Danach wird der x-Endwert geladen und in *xTo* gespeichert.

```
        shl     bx,2
        mov     ax,[es:si+bx+2]
        mov     [LL.yTo],ax
        sub     ax,[yi]
        jns     @@13
        neg     ax
@@13:   inc     ax
        mov     [LL.ddy],ax
        mov     ax,[es:si+bx]
        mov     [LL.xTo],ax
```

Nachdem der Start-Farbwert geladen ist, wird er in eine 16-Bit-Fixed-Point-Zahl umgerechnet und in *Col* gespeichert.

```
        les     si,[C]
        mov     bx,[k]
        movzx   ax,[BYTE PTR
                    es:si+bx]
        shl     eax,16
        mov     [LL.Col],eax
```

Nun wird hier die Differenz zwischen dem x-Start- und x-Endwert gebildet.
Die Variable *icx* wird entsprechend dem Vorzeichen der Differenz gesetzt. In *ddx* wird der Betrag der Differenz abgespeichert!

```
        mov     [LL.icx],1
        mov     ax,[LL.xTo]
        sub     ax,[LL.x]
        jns     @@14
        neg     [LL.icx]
        neg     ax
@@14:   mov     [LL.ddx],ax
        mov     [LL.error],0
```

An dieser Stelle erfolgt die Berechnung für die Interpolation der Kanten-Farbwerte. Start-Farbwert und Endfarbwert werden geladen und die Differenz zwischen ihnen gebildet. Dann wird das ganze durch *ddy* geteilt (Achtung: 16 Bit Fixed Point Arithmetic wird hier verwendet). Sie erhalten so in *Cic* den Wert, den Sie pro y-Zeile auf *Col* hinzuaddieren müssen, um in *Col* den Farbwert für den aktuellen Kantenpunkt zu erhalten.

```
        mov     bx,[L]
        movzx   ax,[BYTE PTR
es:si+bx]
        shl     eax,16
        sub     eax,[LL.Col]
        cdq
        movzx   ebx,[LL.ddy]
        idiv    ebx
        mov     [LL.icC],eax
@@15:   ret
  ENDP
```

Die InitRight-Routine hat einen sehr ähnlichen Aufbau. Einziger Unterschied ist, das *R* nicht dekrementiert, sondern inkrementiert wird. An dieser Stelle

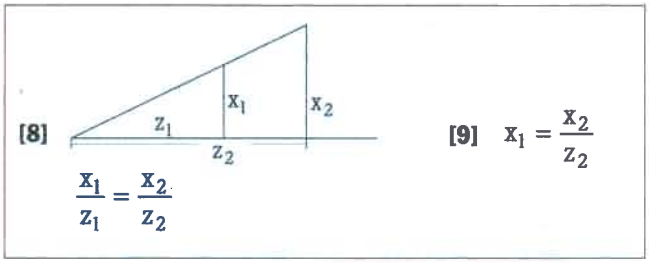


Bild 2. Zur Umrechnung der 3D-Koordinaten hilft Ihnen der Strahlensatz

wird nun mit der CxyShaded-Poly-Routine fortgefahren. Ist *yMax* größer als 199, wird *yMax* auf 199 gesetzt.

```
        mov     ax,199
        cmp     [yMax],ax
        jng     @@14
        mov     [yMax],ax
@@14:   mov     ax,[yMin]
        mov     [yi],ax
```

Hier beginnt die Hauptschleife, in der alle y-Zeilen durchlaufen werden. Zum Ziehen der Linken beziehungsweise Rechten Linie wird eine modifizierte Version des Bresam-Algorithmus eingesetzt.
Um das Ganze zu beschleunigen, werden die benötigten Werte teilweise in Register geladen. Ist *error* <= *ddx*, wird zum Label @@33 gesprungen. Ansonsten findet ein Vergleich statt, ob der x-Endwert erreicht wurde. Ist das nicht der Fall wird *error* um *ddy* erhöht und *icx* zu *x* addiert. Am Ende werden die veränderten Werte aus den Registern zurück in die Variable geschrieben.

```
        mov     ax,[LL.x]
        mov     dx,[LL.ddy]
        mov     bx,[LL.error]
        cmp     bx,[LL.ddx]
        jge     @@33
        cmp     ax,[LL.xTo]
        je      @@33
@@31:   add     bx,dx
        add     ax,[LL.icx]
        cmp     bx,[LL.ddx]
        jge     @@32
        cmp     ax,[LL.xTo]
        jne     @@31
@@32:   mov     [LL.x],ax
@@33:   sub     bx,[LL.ddx]
        mov     [LL.error],bx
```

Nun führen wir die Farbwert-Interpolation für die Linke Linie durch.

```
        mov     eax,[LL.icC]
        add     [LL.Col],eax
```

Gleiches passiert für die Rechte Linie.

```
        mov     ax,[RL.x]
        mov     dx,[RL.ddy]
        mov     bx,[RL.error]
        cmp     bx,[RL.ddx]
        jge     @@43
        cmp     ax,[RL.xTo]
        je      @@43
@@41:   add     bx,dx
        add     ax,[RL.icx]
        cmp     bx,[RL.ddx]
        jge     @@42
        cmp     ax,[RL.xTo]
        jne     @@41
@@42:   mov     [RL.x],ax
@@43:   sub     bx,[RL.ddx]
        mov     [RL.error],bx
        mov     eax,[RL.icC]
        add     [RL.Col],eax
```

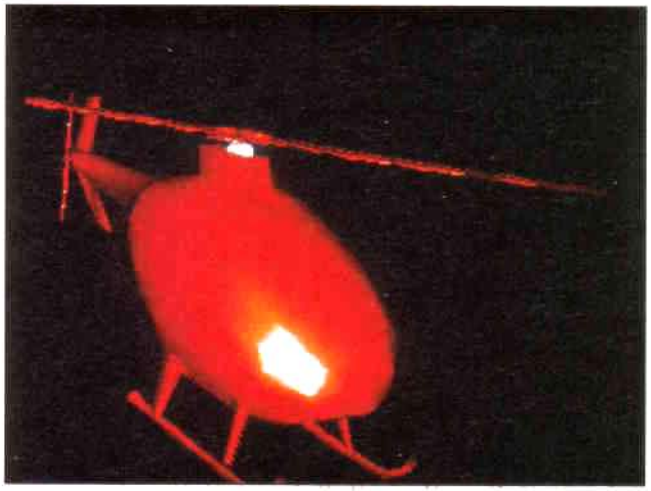


Bild 3. Hier fehlen nur noch die rotierenden Bewegungen der Propeller

Dann wird getestet, ob die zu ziehende Linie außerhalb des Bildschirm liegt. Liegt sie tatsächlich außerhalb, so wird die Linie nicht gezeichnet.

```
        cmp     [yi],0
        jl      @@60
```

Anschließend erzeugt das Programm Kopien der Variablen *L.x/R.x* und *L.Col/R.Col* und ruft *GouraudVLine* auf, die für das Zeichnen der horizontalen Linie in den Bildschirm/Videobuffer verantwortlich ist.

```
        mov     ax,[LL.x]
        mov     [k1.x],ax
        mov     eax,[LL.Col]
        mov     [k1.Col],eax
        mov     ax,[RL.x]
        mov     [k2.x],ax
        mov     eax,[RL.Col]
        mov     [k2.Col],eax
        call    GouraudVLine
```

Ist *yi* gleich *L.yTo* beziehungsweise *R.yTo* so wird *InitLeft* oder *InitRight* aufgerufen.

```
@@60:   mov     ax,[yi]
        cmp     [LL.yTo],ax
        jne     @@61
        call    InitLeft
        mov     ax,[yi]
@@61:   cmp     [RL.yTo],ax
        jne     @@62
        call    InitRight
```

Dann wird *yi* solange um Eins erhöht, wie *yi* < *yMax* ist, danach wird wieder an den Anfang (Label @@21) gesprungen.

```
        inc     [yi]
        mov     ax,[yMax]
        cmp     [yi],ax
        jle     @@21
@@71:   leave
        ret     @@return
  ENDP
```

Als nächstes folgt die Routine „PROC GouraudVLine NEAR" die das Zeichnen der horizontalen Linie übernimmt.
Als erstes wird die Differenz aus *k1.x* und *k2.x* gebildet und in *DeltaX* gespeichert.

```
        movzx   ebx,[k2.x]
        sub     bx,[k1.x]
        jns     @@01
        neg     bx
@@01:   inc     bx
        mov     [DeltaX],bx
```

Danach wird das Farb-Inkrement bestimmt.

```
        xor     edx,edx
        mov     eax,[k2.Col]
        sub     eax,[k1.Col]
        jns     @@02
        neg     eax
@@02:   div     ebx
        mov     [Cic],eax
```

Dann werden die x-Koordinaten geclipped. Dabei wird unterschieden ob *k1.x* > *k2.x* ist oder nicht.
Entsprechend müssen natürlich neben den Koordinaten auch die Farbwerte angepaßt werden.

```
        movzx   ebx,[k1.x]
        movzx   ecx,[k2.x]
        cmp     bx,cx
        jnl     @@13
        cmp     bx,0
        jnl     @@11
        neg     bx
        mov     eax,[Cic]
        imul    ebx
        add     [k1.Col],eax
        xor     bx,bx
@@11:   cmp     cx,319
        jng     @@12
        sub     cx,319
        mov     eax,[Cic]
        imul    ecx
        add     [k2.Col],eax
        mov     cx,319
@@12:   cmp     bx,cx
        jg      @@40
        jmp     @@16
@@13:   cmp     cx,0
        jnl     @@14
        neg     cx
        mov     eax,[Cic]
        imul    ecx
        add     [k1.Col],eax
        xor     cx,cx
@@14:   cmp     bx,319
        jng     @@15
        sub     bx,319
        mov     eax,[Cic]
        imul    ebx
        add     [k2.Col],eax
        mov     bx,319
@@15:   cmp     cx,bx
        jg      @@40
```

Ab dieser Stelle werden alle für den Mainloop benötigten Variablen berechnet und in Register

geladen. Da sich *DeltaX* aufgrund des Clippings geändert haben kann, wird es an dieser Stelle neu berechnet.

```
        mov     [k1.x],bx
        mov     [k2.x],cx
        mov     bx,[k2.x]
        sub     bx,[k1.x]
        jns     @@03
        neg     bx
@@03:   inc     bx
        mov     [DeltaX],bx
```

Hier wird das Low-Byte des Hi-Word des DWords *Cic* in das CL-Register und das Low-Word von Cic ins Hi-Word von ebx geladen. Warum, wird bei der Beschreibung des Mainloops erklärt.

```
        mov     eax,[Cic]
        mov     bx,ax
        rol     ebx,16
        rol     eax,16
        mov     cl,al
```

Das bx-Register wird mit dem Inkrement geladen um den der Offset in den Videobuffer erhöht wird.
Für den weiteren Verlauf ist wieder eine Unterscheidung wichtig, ob *k1.Col* > *k2.Col* oder nicht.

```
        mov     eax,[k1.Col]
        cmp     eax,[k2.Col]
```

Entsprechend wird k1.Col/k1.x beziehungsweise k2.Col/k2.x als Farb/x-Startwert genommen. Dabei wird das Low-Word von Col in das Hi-Word von edi geladen. Das Low-Byte des Hi-Words von *Col* wird in das AL-Register geladen. Der Offset in den Bildschirmspeicher/Videobuffer wird anschließend in das DI-Register geladen. Das Low-Byte des Hi-Words von *Cic* welches ins CL-Register kopiert wurde wird nun ins ah-Register übertragen.

```
        mov     ah,cl
```

Die Hauptschleife arbeiten wir zum besseren Verständnis Schritt für Schritt durch.

```
        mov     cx,[DeltaX]
        mov     ds,[_VirtualVSeg]
```

In al befindet sich der Farbwert des Bildschirmpunktes [ds:di].

```
        mov     [di],al
```

Während der folgenden 32-Bit-Addition geschehen nun zwei Dinge gleichzeitig. Zwei 16-Bit-Additionen sind hier zu einer zusammengefaßt worden. DI wird je nachdem um 1 oder -1 „erhöht" und gleichzeitig werden Kommaanteil von *Col* und *Cic* addiert (im Hi-Word). Kommt es zu einem Uberlauf, so wird das Carry-Flag gesetzt, sonst gelöscht.

```
        add     edi,ebx
```

Das Carry-Flag wird nun zusammen mit dem Vorkommaanteil von *Cic* zum Vorkommaanteil von *Col* addiert.
Anstatt „Loop" wird hier die Befehlsreihenfolge „dec/jnz" verwendet. Diese ist schneller auf 486er-PCs als der Loop-Befehl.
Das Programm, welches nun alle Funktionen zusammensetzt, ist vollständig in Turbo-Pascal geschrieben. Vielleicht werden Sie sich wundern, warum es nicht vollständig in Assembler geschrieben wurde. Die Antwort ist ganz einfach. Zum einen ist das Programm so übersichtlicher und zum anderen ist an dieser Stelle ein Assembler-Programm nicht notwendig, da die Pascal-Programmteile verhältnismäßig wenig Zeit verbrauchen.
Die folgenden Datenstrukturen werden dabei verwendet:

```
type
 P2D=^T2D;
 T2D=array[x..y] of
Integer;
 P3D=^T3D;
 T3D=array[x..z] of
LongInt;

 PPointList=^TPointList;
 TPointList=
     array[0..2447] of T3D;

 PProjectList=^TProject-
List;
 TProjectList=
     array[0..2447] of
T2D;
 PColorList=^TColorList;
 TColorList=
    array[0..2447] of
Byte;
```

Count gibt die Anzahl der Eckkoordinaten an. *List* gibt den Index an, unter dem der Eckpunkt in der Punktliste (TPoint-List) zu finden ist.

```
  PFace = ^TFace;
  TFace = RECORD
    Count :Integer;
    List  :array[0..255]
of Word;
  end;
```

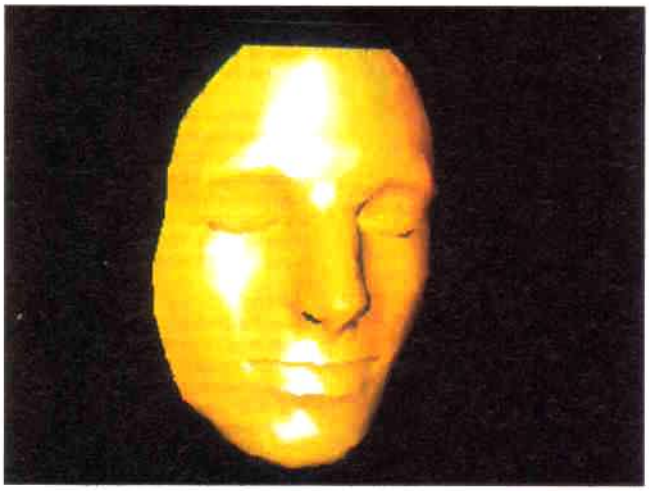


Bild 4. An der Maske können Sie die tollen Licht und Schatteneffekte betrachten.

```
  PFaceList = ^TFaceList;
  TFaceList =
array[0..2447] of PFace;
```

Hier kommt der Tiefensortier-Algorithmus (depth-sort) zum Einsatz. Bei diesem werden die Flächen nach ihrem Abstand vom Betrachter sortiert. *TZEntry* nimmt hierbei den Abstand und einen Zeiger auf die entsprechende Fläche auf.

```
  PZEntry = ^TZEntry;
  TZEntry = RECORD
    z  :LongInt;
    P  :PFace;
  end;
  PZList = ^TZList;
  TZList=
  array[0..2047] of
TZEntry;
```

Sie werden sich vielleicht wundern, warum die definierten Listen so groß sind? Die Typen wurden für den einfachen Zugriff definiert (Type-casting). Alle Listen werden mit *Getmem* und nicht mit *New* allokiert. So wird auch nur so viel Speicher verbraucht, wie benötigt.
PCount enthält die Anzahl der Punkte und *FCount* die Anzahl der Flächen.

```
  PCount: Integer;
  FCount: Integer;
```

VL nimmt die noch nicht transformierten Punkte auf. In *TL* werden die transformierten Punkte gespeichert (die für jeden Frame neu berechnet werden müssen). In *PL* werden letztendlich die projektierten Punkte, sprich die Bildschirmkoordinaten abgelegt.
NL nimmt zum Schluß noch die Normalvektoren der Punkte auf. Auch Sie müssen für jeden Frame aktualisiert werden.

```
  VL: PPointList;
  TL: PPointList;
  PL: PProjectList;
  NL: PPointList;
```

ZL stellt eine Liste dar, in der z-Wert und ein Pointer auf die

zugehörige Fläche gespeichert sind. Für jeden Frame werden die z-Werte neu berechnet. Anschließend wird die Liste mit einem Q-Sort sortiert. Nun können die Flächen durch einfaches Durchlaufen der Liste in richtiger Reihenfolge dargestellt werden.
Um ein mehrfaches Berechnen der Farbwerte zu verhindern, wird der Farbwert für jeden Punkt in *CL* abgelegt. *FL* enthält Pointer auf die Flächen.
Die Prozedur „ShowShadedObject" führt alle nötigen Operationen aus, um ein Objekt darzustellen.
Zuerst wird der Bildschirm gelöscht. Der Bildschirm ist hier nicht der wirkliche Bildschirm, sondern ein Buffer der Größe 64 KByte im Hauptspeicher. Speicherzugriffe im Hauptspeicher sind doppelt so schnell wie Zugriffe auf den VGA-RAM (zumindest beim ISA-Bus).
Mit Create wird die Transfomationsmatrix *M* erzeugt. Mit Rotate wird die Rotationsmatrix *R* aufgebaut. *M* ist für die Transformation der Punkte wichtig und *R* für die Drehung der Normalen (nicht jedesmal neu berechnen, sondern einfach nur entsprechend drehen).
Die Prozedur Project führt diese Operationen durch.
Danach wird *ZL* aktualisiert und sortiert. Entsprechendes geschieht mit *CL*. Danach sind alle Daten berechnet und die Flächen können dargestellt werden.
Anschließend wird der Videobuffer in den VGA-RAM kopiert (CopyVBuffer).

```
procedure ShowShadedOb-
ject;
var
  M :TMatrix;
  R :TMatrix;
  i :Integer;
begin
  ClearScreen(0);
  Create(tx,ty,tz, $200,
  $1B0,$200, rx,ry,rz, M);
  Rotate(rx,ry,rz, R);
  Project(M);
  UpdateZList;
  SortZList(0,FCount-1);
  UpdateColorList(R);
  for i:=FCount-1 downto 0
do
ShowShadedFace(ZL^[i].P);
  ShowVBuffer;
end;
```

Für die Berechnung der Farbwerte wird nicht der Cosinus zwischen Betrachtervektor und Normalvektor genutzt, sondern der Betrachtervektor wird als (0,0,1) angenommen. Dies ist zwar nicht korrekt sieht aber trotzdem sehr gut aus. Der Farbwert entspricht dann der z-Komponente des gedrehten Normalvektors. Da neun Bit Kommaanteil verwendet werden, muß die z-Komponente um 1 nach rechts geshiftet werden, um einen Farbwert zwischen 0..255 zu erhalten (procedure UpdateColorList(R :TMatrix);).
Ein sehr wichtiger Trick, der die Ausführungsgeschwindigkeit steigert, ist die Entfernung von nicht sichtbaren Rückseiten.
Kann man nicht in ein Vektorobjekt hineinsehen, so können die Flächen von denen die Rückseite sichbar ist, entfernt werden. Anhand der Bildschirmkoordinaten können Sie leicht bestimmen, ob die Vorderseite oder die Rückseite eines Objektes sichbar ist. Dazu ist es aber wichtig, daß von allen Flächen die Punkte immer in der richtigen Reihenfolge definiert sind (rechtsdrehend für die Vorderseite und linksdrehend für die Rückseite). Die Formel hierfür lautet:

```
S = (y1-y0)*(x2-x0)-(y2-
y0)*(x1-x0)
(z - Komponente des Vektor
Kreuz Produktes)
```

Bildschirmkoordinaten und Farbwerte werden in zwei Arrays kopiert und zusammen mit der Eckpunktanzahl *CxyShadedPoly* übergeben (siehe Procedur *ShowShadedFace*).
Obwohl an vielen Stellen Optimierungen nicht vorgenommen wurden, ist es trotzdem möglich zirka 800 Flächen auf einem 486SX-25 mit annehmbarer Geschwindigkeit zu drehen. *uk*