Kirill Kranz

Eine Untersuchung der Geschwindigkeit von zwei Algorithmen zur Füllung von Dreiecken, der Scan-Line und der Half-Space Füllmethode

IED911

SAE Hamburg

16.8.2012

Jan-Friedrich Conrad

Inhaltsverzeichnis

1.Vorwort

In dieser Abhandlung wird verglichen, wie die Geschwindigkeit zweier Simpler Polygon Schattierung Algorithmen im relativen Vergleich zueinander stehen. Es wird das Flatshading Model angewandt. Um Flatshading zu realisieren, zeichnet man ein gefülltes Polygon mit einer Grundfarbe. Es wird die einfachste Art vom Polygonen zugelassen, ein Dreieck. Die Bedingung eines dreidimensionalen Dreiecks ist gegeben, wenn alle 3 Vertexes[1] der Kanten auf einer Ebene liegen. Übergeben werden folgende Parameter: vertexA, vertexB, vertexC und die Farbe.
Auf eine beliebige Surface[2] wird ein Dreieck mit den Schnittpunkten, die in zweidimensionale Weltkoordinaten umgewandelt worden sind, mit der gegebenen Farbe gezeichnet.
Man unterscheidet hier zwei Arten der Darstellung:

- die Half-Space Methode.
- die Scan-Line Methode. (Rasterzeilen Algorithmus)

Die erste wird im Kapitel 2 ausführlich erläutert und in 4 Arten implementiert. Es werden folgende Ansätze vorgestellt: Der Basisalgorithmus, die optimierte Variante und die Fixedpoint[3] Arithmetik Übersetzung sowie die Assemblierung von jener.
Das 3. Kapitel widmet sich der zweiten Methode, und deren Implementation auf zwei Arten: der Basisalgorithmus und die Assemblierung dessen.
Im Kapitel 4 wird ein Test-Programm zum direkten Vergleich aufgesetzt, das die entwickelten Algorithmen auf bestimmte Zeit laufen lässt und auf einem System, die Frame Rate misst.
Ein kleines persönliches Resümee und ein Literaturverzeichnis sowie ein Abbildungsverzeichnis finden sich am Ende dieser Arbeit; außerdem sind ein Archiv mit den Quelltexten und kompilierten Programmen im Anhang zu finden.
In der folgenden Arbeit werden Quelltexte in der Sprache Pascal, die eine Hochsprache repräsentiert, mit Zeilennummer versehen. Eine Erklärung zu den Denkabsätzen wird gegeben.
Es ist auch notwendig, die Sprache Assembler zu kennen, da es einen entscheidenden Geschwindigkeits- Vorteil bringt, eine Hochsprache manuell zu assemblieren.
Meines Erachtens existieren viel zu wenig gute Quellen, die sich mit diesem Thema befassen, geschweige denn einen Vergleich zwischen den beiden Algorithmen anstellen. Diese Umstände haben mich dazu bewegt, diese Arbeit zu schreiben.

[1] siehe: Jacobs, 1999, S. 18

[2] siehe: Zerbst, 2000, S. 167

[3] Gordon, 1998, http://www.eetindia.co.in/ARTICLES/1998APR/PDF/EEIOL_1998APR03_EMS_TA.pdf

2. Der Half-Space Algorithmus in Theorie und Praxis

2.1 Der theoretische Aufbau

„Als Erstes sollte gesagt sein, dass eine Half-Space Funktion auf einer Seite einer Linie positiv und auf der anderen Seite negativ ist. So zerteilt sie den Raum in zwei Hälften, daher der Name. Hier ist ein kleine Illustration, in der demonstriert wird, wie die Seiten von einem Dreieck den Bildschirm in einen positiven und einen negativen Part teilen.

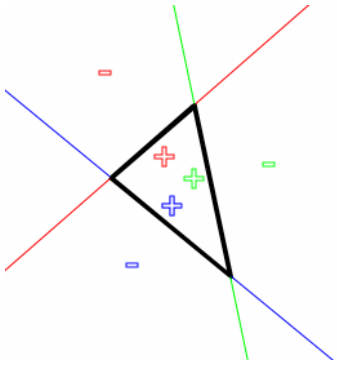

Bild 1.

Bereiche, wo alle drei Half-Space Funktionen positiv sind, befinden sich innerhalb des Dreiecks. Wenn die Half-Space Funkion Null ergibt, befinden wir uns auf der Seite des Dreiecks.

Falls eine von Ihnen negativ ist, befinden wir uns außerhalb des Dreiecks.
Diese Tatsache können wir nutzen, um ein Dreieck auf einer Surface, durch das Prüfen jedes Pixels, darzustellen.
Die Gleichung einer Linie ist genau das, was wir suchen. Für eine mit den Startpunkten X1 und Y1 und den Endpunkten X2 und Y2 lautet sie wie folgt:

$$(X2-X1) * (y-Y1) - (Y2-Y1) * (x-X1) = 0$$

Setzt man in diese Formel die zu prüfende Koordinaten x und y, erhält man Null, falls diese sich auf der Linie befinden, jedoch einen positiven Wert auf der einen Seite und einen negativen auf der anderen.
Da diese Gleichung für das Zeichnen eines Dreiecks eingesetzt wird, muss beachtet werden, dass die Vertexe, die die Eckpunkte des Dreiecks beschreiben, im Uhrzeigersinn angeordnet sind, somit bei jeder zu prüfenden Linie, relativ gesehen, die positiven Ergebnisse immer links zu finden sind und die negativen rechts.
Es ist natürlich möglich, den ganzen Bildschirm Pixel für Pixel zu prüfen, um herauszufinden, ob dieser Pixel sich innerhalb des Dreiecks befindet oder nicht, doch dieses Verfahren wäre reine Rechenzeitverschwendung. So wird ein Umgebungsrechteck um das zu zeichnen geltende Dreieck errechnet, und es wird der Half-Space Test nur in jenem durchgeführt.“ [1]

[1] **Capens, 2004,** http://devmaster.net/forums/topic/1145-advanced-rasterization/ **(Eigene Übersetzung)**

2.2 Implementation des Basisalgorithmus

Hier ist die Basisversion dieser Implementation gegeben. Es ist sinnvoll, sie in zwei Teile zu unterteilen: das Zeichnen des Dreiecks und das Agieren mit dem Z-Buffer.[1]
Jede Funktion bzw. Prozedur ist mit einem Header anzugeben, in dem festgelegt wird, wie sie heißt und mit welchen Parametern sie aufzurufen ist. An dieser Stelle übergeben wir alle Drei in 2D, durch eine perspektivische Projektion, umgewandelte Koordinaten zwecks sinngemäßer Variablen . Auch die Farbe und der Ort bzw. wo es hin zu zeichnen gilt(die Surface), wird angegeben.

```
001: procedure TriangleFLATZ(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3:single;color:dword;where:gfxImage);
```

Nun gilt es, die Variablen zu deklarieren und einige von ihnen zu initialisieren.
Zeile 003 beherbergt die notwendige Platzhalter für die planare Gleichung, die im Zusammenhang mit dem Ermitteln der Z Koordinate, die in der Variable in der nächsten Zeile gespeichert wird, benötigt wird. Danach folgen die Maße des Umgebungsrechteckes, die das zu zeichnen geltende Dreieck umgeben, gefolgt (Zeile 006) von den Variablen, die in der Hauptschleife inkrementiert werden. Zum Schluss sind drei Variablen, die die Ergebnisse der Half-Space Tests in sich beherbergen, angegeben(Zeile 007).

```
002: var
003:     a,b,c,D:single;
004:     z:single;
005:     minx,maxx,miny,maxy:word;
006:     x,y:word;
007:     f1,f2,f3:single;
```

Der Anfangspunkt der Routine, in Pascal, befindet sich hier.

```
008: begin
```

Da die planare Gleichung wie folgt aussieht: $ax+by+cz+d=0$ und aus dem Bild 2 leicht nachzuvollziehen ist, wieso es so ist, werden die Werte a,b,c und d wie folgt ermittelt:

$a = (By-Ay)(Cz-Az)-(Cy-Ay)(Bz-Az)$

$b = (Bz-Az)(Cx-Ax)-(Cz-Az)(Bx-Ax)$

$c = (Bx-Ax)(Cy-Ay)-(Cx-Ax)(By-Ay)$

$d = -(aAx+bAy+cAz).$

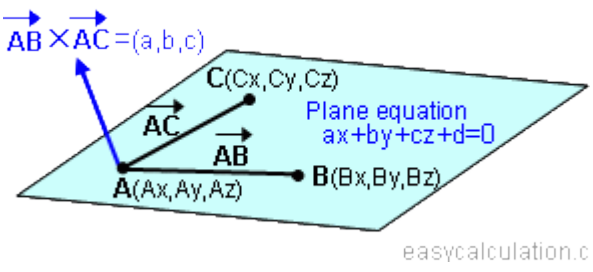

Bild 2.

[1] **siehe: Apetri, 2008, S. 373**

So wird die planare Gleichung nun implementiert

```
009:    a:= (y2 - y1)*(z3 - z1) - (y3 - y1)*(z2 - z1);
010:    b:= (x2 - x1)*(z3 - z1) - (x3 - x1)*(z2 - z1);
011:    c:= (x2 - x1)*(y3 - y1) - (x3 - x1)*(y2 - y1);
012:    D:= -x1*a + y1*b - z1*c;
```

In diesem Block wird das Umgebungsrechteck definiert. Es gilt die kleinstmögliche X und Y Koordinate, die an dem Dreieck oben links anliegt, zu finden. Sowohl auch die kleinstmögliche X und Y Koordinate, die an dem Dreieck rechts unten anliegt. Die Werte werden gerundet, da die beiden Hauptschleifen nur mit Ganzzahlwerten arbeiten können.

```
013:    minx:= round(min(x1,min(x2, x3)));
014:    maxx:= round(max(x1,max(x2, x3)));
015:    miny:= round(min(y1,min(y2, y3)));
016:    maxy:= round(max(y1,max(y2, y3)));
```

In den Hauptschleifen wird Pixel per Pixel das Umgebungsrechteck durchgegangen.

```
017:    for y:=miny to maxy do
018:        for x:=minx to maxx do begin
```

Nun wird der Half-Space Test für alle Seiten des Dreiecks durchgeführt. In f1, f2 und f3 werden die Ergebnisse eingetragen.

```
019:        f1:=((x1 - x2) * (y - y1)) - ((y1 - y2) * (x - x1));
020:        f2:=((x2 - x3) * (y - y2)) - ((y2 - y3) * (x - x2));
021:        f3:=((x3 - x1) * (y - y3)) - ((y3 - y1) * (x - x3));
```

Nun gilt es zu prüfen, ob sich die festgestellten Berechnungen als positiv erweisen.

```
022:        if ( f1 > 0) and ( f2 > 0) and ( f3 > 0) then begin
```

Falls der aktueller Pixel sich innerhalb des Dreiecks befindet, wird seine Z Position ermittelt. Ist die Fläche durch $Ax + By + Cz + D = 0$ gegeben, so ist die Tiefe des aktuellen Pixels Z.

```
023:            z:= (-D - a*x + b*y)/c;
```

Die letzte Prüfung gilt dem Z-Buffer, in dem der Z Wert des Pixels gespeichert ist. Falls der zum Zeichnen bereite Pixel sich näher an der Kamera befindet als der im Z-Buffer vorhandene (Prüfung in Zeile 024), wird er dargestellt (Zeile 025) und sein Z Wert wird in den Z-Buffer geschrieben(Zeile 026), sonst nicht.

```
024:                if z > zbuffer[x,y] then begin
025:                  PutPixel32(x, y, color, where);
026:                  zbuffer[x,y]:=z;
027:                end;
```

Ende der Schleifen.

```
028:          end;
029:      end;
```

Ende der Routine.

```
039:  end;
```

Leider hat dieser Ansatz pro Pixel sechs Multiplikationen und 15 Subtraktionen. Dazu kommen zwei Multiplikationen, eine Division, eine Addition und zwei Subtraktionen der Z-Buffer Algorithmen. Außerdem kommen noch drei Vergleiche hinzu.

Das dieser Code so langsam ist liegt zum Teil daran, dass hier mit Fließkommazahlen gearbeitet wird. Es ist besonders einfach, an Hand dieses Beispieles, zu verstehen wie der Half-Space-Algorithmus funktioniert.

Der oben gelistete Quellcode(Zeilen 001 bis 039) wurde aus dem Original der Publikation **Capens, 2004,** **http://devmaster.net/forums/topic/1145-advanced-rasterization/** in die Programmiersprache Pascal von mir übersetzt.

2.3 Praxisnähere Implementierung

Derselbe Algorithmus, doch stark optimiert, wird pro Pixel vier Subtraktionen,zwei Additionen und drei Vergleiche haben. Dies ist zu erreichen, da in jedem vertikalen Durchgang nur der X Wert sich verändert, so ist es möglich, viele Werte vor zu kalkulieren.

Für eine ausführliche Erklärung des Headers dieser Routine siehe Kapitel 2.1.

```
001: procedure TriangleFLATZ(X1,Y1,Z1,X2,Y2,Z2,X3,Y3,Z3:single;color:dword;where:gfxImage);
```

Auch die Koordinaten des Umgebungsrechteckes wurden in Kapitel 2.1 beschrieben.

```
002: var
003:     minx,maxx,miny,maxy:longint;
```

Neu sind die „Delta“ Variablen, die reine Seitenlängen enthalten.

```
004:     Dx12,Dx23,Dx31,Dy12,Dy23,Dy31:single;
```

Neu sind auch Variable, die die konstanten Werte der Gleichung enthalten.

```
005:     C1,C2,C3:single;
006:     Cx1,Cx2,Cx3,Cy1,Cy2,Cy3:single;
```

Die Zählvariablen der Hauptschleife sind unumgänglich.

```
007:     x,y:longint;
```

Um die Z Koordinate richtig zu ermitteln, brauchen wir wieder Variablen für die planare Gleichung und weitere für Konstante Werte.

```
008:     a,b,c,D:single;
009:     Z,initialZ,tempZ:single;
010:     dx,dy:single;
```

Fangen wir mit dem Ausrechnen der Seitenlängen an. Speichern wir diese ab, um zu einem späteren Zeitpunkt auf Subtraktionen zu verzichten.

```
011: begin
012:     Dx12:= x1 - x2;
013:     Dx23:= x2 - x3;
014:     Dx31:= x3 - x1;
015:     Dy12:= y1 - y2;
016:     Dy23:= y2 - y3;
017:     Dy31:= y3 - y1;
```

Der Code zum Herausfinden des Umgebungsrechtecks ist zu der Code im Kapitel 2.1 gleich geblieben.

```
018:  minx:=round(min(x1,min(x2, x3)));
019:  maxx:=round(max(x1,max(x2, x3)));
020:  miny:=round(min(y1,min(y2, y3)));
021:  maxy:=round(max(y1,max(y2, y3)));
```

Sowie auch die Planare Gleichung für die spätere Bestimmung des aktuellen Z-Wertes.

```
022:    a:= (y2 - y1)*(z3 - z1) - (y3 - y1)*(z2 - z1);
023:    b:= (x2 - x1)*(z3 - z1) - (x3 - x1)*(z2 - z1);
024:    c:= (x2 - x1)*(y3 - y1) - (x3 - x1)*(y2 - y1);
025:    D:= -x1*a + y1*b - z1*c;
```

Folgende drei Konstante Werte sind ein Teil der Half-Space Gleichung.

```
026:  C1:= Dy12 * x1 - Dx12 * y1;
027:  C2:= Dy23 * x2 - Dx23 * y2;
028:  C3:= Dy31 * x3 - Dx31 * y3;
```

In den folgenden Platzhaltern werden die Y-Startpunkte des Umgebungsrechteckes bzw. der oberen zu prüfenden Grenze gespeichert.

```
029:  Cy1:= C1 + Dx12 * miny - Dy12 * minx;
030:  Cy2:= C2 + Dx23 * miny - Dy23 * minx;
031:  Cy3:= C3 + Dx31 * miny - Dy31 * minx;
```

Es ist nicht möglich, durch Null zu teilen. Falls *C* doch Null beherbergt, wird die Routine abgebrochen.

```
032:  if c = 0 then exit;
```

Der Wert in *initialZ* ist der Startwert der Z Koordinate in der zu berechnen geltenden Rasterzeile.

```
033:  initialZ:= (A * -(minx) + B * -(miny) -D) / C;
```

Auf der Rasterzeile benachbarter X Punkt wird durch +dx bestimmt, sowie der benachbarte Y Punkt durch +dy. Hier sind die Konstanten, die in der Hauptschleife dafür benötigt werden.

```
034:  dx:= -A / C;
035:  dy:= -B / C;
```

Die Y-Schleife sorgt für pixelgenaueren Durchgang des Umgebungsrechteckes von „oben“ nach

„unten“.

```
036:   for y:= miny to maxy do begin
```

Hier werden Startwerte für den horizontalen Durchgang zugeordnet.

```
037:      Cx1:= Cy1;
038:      Cx2:= Cy2;
039:      Cx3:= Cy3;
040:      Z:=initialZ;
```

Die Hauptschleife, in der von links nach rechts in der Rasterzeile durchgegangen wird:

```
041:      for x:= minx to maxx do begin
```

Es ist an der Zeit festzustellen, ob der aktuelle Pixel sich tatsächlich in dem vorgegebenen Dreieck befindet.

```
042:         if (Cx1 > 0) and (Cx2 > 0) and (Cx3 > 0) then begin
```

Der Wert *Z* wird um die Konstante *dx* erhöht,

```
043:            Z:=Z+dx;
```

Von *Z* wird nun *zAddValue* abgezogen. Dies geschieht im Sinne des Clear-Reduction-Algorithmen.

```
044:            tempZ:=z-zAddValue;
```

In diesem Codesegment hat sich im Vergleich zu dem oben Beschriebenen nichts verändert.

```
045:            if tempZ > zbuffer[x,y] then begin
046:               PutPixel32(x, y, color, where);
047:               zbuffer[x,y]:=tempZ;
048:            end;
049:         end;
```

Es wird der *initialZ* Wert um eine Rasterzeile nach „unten“ bewegt.

```
050:         initialZ:=initialZ+dy;
```

In diesen drei Zeilen wird der nächste Pixel in seiner Half-Space Funktion berechnet. Dies ist nur mit dem Subtrahieren der festen Seitenlängen möglich.

```
051:         Cx1:=Cx1-Dy12;
052:         Cx2:=Cx2-Dy23;
053:         Cx3:=Cx3-Dy31;
054:      end;
```

Hier werden die aktuellen Werte eine Rasterzeile nach „unten" bewegt.

```
055:      Cy1:= Cy1+Dx12;
056:      Cy2:= Cy2+Dx23;
057:      Cy3:= Cy3+Dx31;

058:    end;
059:  end;
```

Diese Routine ist um einiges schneller als die Basisimplementation, obwohl sie immer noch mit Fließkommazahlen arbeitet.

Der oben gelistete Quellcode(Zeilen 001 bis 059) wurde aus dem Original der Publikation **Capens, 2004,** **http://devmaster.net/forums/topic/1145-advanced-rasterization/** in die Programmiersprache Pascal von mir übersetzt.

2.4 Fixedpoint implementation

Der Procedurheader ist unverändert

```
001: procedure TriangleFLATZ(theX1,theY1,theZ1,theX2,theY2,theZ2,theX3,theY3,theZ3:single;
                             color:dword;where:gfxImage);
```

In den folgenden Variablen werden die im Header übergebenen Koordinaten im Fixedpoint Format festgehalten.

```
002: var
003:     X1,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3:longint;
```

Die Seitenlängen und deren Fixedpoint Umsetzungen werden in den folgenden Variablen gespeichert.

```
004:     DX12,DX23,DX31,DY12,DY23,DY31:longint;
005:     FDX12,FDX23,FDX31,FDY12,FDY23,FDY31:longint;
```

Das Umgebungsrechteck wird durch folgende Variablen beschrieben.

```
006:     minx,maxx,miny,maxy:longint;
```

Wie in der vorherigen Routine werden konstante Werte der Half-Space Gleichung hier gespeichert

```
007:     C1,C2,C3:longint;
008:     CX1,CX2,CX3,CY1,CY2,CY3:longint;
```

Zähl-Variablen für die Hauptschleife werden deklariert.

```
009:     x,y:longint;
```

Folgende Variablen haben mit der Bestimmung der Z Koordinate zu tun.

```
010:     a,b,c,D:longint;
011:     Z:longint;
012:     initialZ:Longint;
013:     dx,dy:Longint;
014:     tempZ:Longint;
015: begin
```

Es findet die Konvertierung der Eckpunkt Koordinaten in den Fixedpoint Format 28:4 statt.
Doch was ist der Fixedpoint 28:4 Format? Dies ist eine Methode, um reale Zahlen als ganze Zahlen

zu beschreiben. Es gilt die reale Zahl so zu skalieren, dass die Nachkommazahl in die ganze Zahl reingeschrieben wird.
In unserem Fall ist der Skalar 16.0, dies erlaubt uns einen 28bit großen Ganzzahlwert und einen 4bit großen Nachkommastellen-Bereich.
Jetzt gilt es die konvertierte Zahlen zu runden und abzuspeichern.

```
016:   Y1:= round(16.0 * theY1);
017:   Y2:= round(16.0 * theY2);
018:   Y3:= round(16.0 * theY3);
019:   X1:= round(16.0 * theX1);
020:   X2:= round(16.0 * theX2);
021:   X3:= round(16.0 * theX3);
022:   Z1:= round(16.0 * theZ1);
023:   Z2:= round(16.0 * theZ2);
024:   Z3:= round(16.0 * theZ3);
```

Hier werden die Konstanten der planaren Gleichung ausgerechnet und in den Fixedpoint Format umgewandelt. Das Multiplizieren zweier Fixedpoint Zahlen erfolgt nach folgender Regel:

a * b / Scale-Faktor

In unserem Falle ist Scale-Faktor 16.0. Es wird dasselbe Ergebnis erzielt, wenn man mit einer binären Rechtsverschiebung um 4 anstatt einer Division durch 16 arbeitet. „Die Division mit Hilfe des SHR-Befehls ist etwa 40-mal schneller als eine Division mit dem DIV-Befehl.“[1]

```
025:       a:= ((y2 - y1)*(z3 - z1) shr 4) - ((y3 - y1)*(z2 - z1) shr 4);
026:       b:= ((x2 - x1)*(z3 - z1) shr 4) - ((x3 - x1)*(z2 - z1) shr 4);
027:       c:= ((x2 - x1)*(y3 - y1) shr 4) - ((x3 - x1)*(y2 - y1) shr 4);
028:       D:= ((-x1*a) + (y1*b) - (z1*c)) shr 4;
```

Die Seitenlängen werden ausgerechnet .

```
029:   DX12:= X1 - X2;
030:   DX23:= X2 - X3;
031:   DX31:= X3 - X1;
032:   DY12:= Y1 - Y2;
033:   DY23:= Y2 - Y3;
034:   DY31:= Y3 - Y1;
```

[1] siehe: Backer, 2003, S. 145

Seitenlängen werden in den Fixedpoint Format umgewandelt.

```
035:  FDX12:= DX12 shl 4;
036:  FDX23:= DX23 shl 4;
037:  FDX31:= DX31 shl 4;
038:  FDY12:= DY12 shl 4;
039:  FDY23:= DY23 shl 4;
040:  FDY31:= DY31 shl 4;
```

Das Umgebungsrechteck wird festgelegt. Es wird die kleinstmögliche X Koordinate festgestellt, mittels dem Vergleich zwischen X1,X2 und X3. Die größtmögliche, also die Rechte wird ermittelt durch den Vergleich von X1, X2 und X3. Dasselbe Verfahren wird für das Herausfinden der kleinstmöglichen und der größtmöglichen Y Koordinaten des Umgebungsrechteck verwendet.

```
041:      minx:= min(x1,min(x2, x3)) shr 4;
042:      maxx:=max(x1,max(x2, x3)) shr 4;
043:      miny:= min(y1,min(y2, y3)) shr 4;
044:      maxy:=max(y1,max(y2, y3)) shr 4;
```

Folgende drei konstante Werte sind ein Teil der Half-Space Gleichung. Die Addition sowie die Subtraktion von Fixedpoint Zahlen erfolgt durch ein normales arithmetisches Minus bzw. Plus.

```
045:  C1:= DY12 * X1 - DX12 * Y1;
046:  C2:= DY23 * X2 - DX23 * Y2;
047:  C3:= DY31 * X3 - DX31 * Y3;
```

Falls der Half-Space Test eine Null hervorbringt, liegt der getestete Pixel am Rand des Dreieckes. Nun gilt es nach folgender Regel dies zu behandeln: Alle Seiten auf der linken Seite oder auf der Horizontalen des oberen Dreiecks, werden als „in dem Dreieck“ behandelt. Hier erfolgt nun durch dir Sub-Pixel[1] accuracy Methode eine Korrektur der konstanten Werte.

```
048:  if (DY12 < 0) or ((DY12 = 0) and (DX12 > 0)) then inc(C1);
049:  if (DY23 < 0) or ((DY23 = 0) and (DX23 > 0)) then inc(C2);
050:  if (DY31 < 0) or ((DY31 = 0) and (DX31 > 0)) then inc(C3);
```

In den folgenden Platzhaltern werden die Y-Startpunkte des Umgebungsrechteckes bzw. der oberen zu prüfenden Grenze gespeichert.

[1] siehe: Nettle, http://www.cpp-home.com/tutorials/224_1.htm abgerufen August 2012

```
051:    CY1:= C1 + DX12 * (miny shl 4) - DY12 * (minx shl 4);
052:    CY2:= C2 + DX23 * (miny shl 4) - DY23 * (minx shl 4);
053:    CY3:= C3 + DX31 * (miny shl 4) - DY31 * (minx shl 4);
```

Es gilt nun zu prüfen, ob eine Division durch Null später zustande kommt.

```
054:    if (c shr 4) = 0 then exit;
```

Der Z-Startwert wird festgelegt.

```
055:    initialZ:= (((A * -(minx)) shr 4 + ((B * -(miny)) shr 4 -D)) div (C shr 4));
```

Sowie auch die Z-Wert Inkremente für den nächsten Pixel in der X Reihenfolge sowie in der Y Reihenfolge.

```
056:    dx:= ((-A) div (C shr 4));
057:    dy:= ((-B) div (C shr 4));
```

In der Y Schleife.

```
058:    for y:=miny to maxy do begin
```

werden erst einmal die Startpunkte festgelegt.

```
059:        CX1:=CY1;
060:        CX2:=CY2;
061:        CX3:=CY3;
```

Die Z Variable wird für die darauffolgende X-Schleife vorbereitet.

```
062:        Z:=initialZ;
063:        for x:=minx to maxx do begin
```

Jetzt wird getestet, ob sich der aktuelle Pixel innerhalb des Dreieckes befindet.

```
064:            if (CX1 > 0) and (CX2 > 0) and (CX3 > 0) then begin
```

Falls ja, wird der aktuelle Z-Wert ermittelt und nach dem Clear-Reduction-Algorithmus[1] der in den Z-Buffer zu speichernde Wert berechnet.

```
065:                    Z:=Z+dx;
```

[1] siehe: Apetri, 2008, S. 394

Der Hauptnachteil des Einsatzes des Z-Buffer liegt darin, dass man ihn am Anfang jedes Frames zurück auf einen Startwert setzen muss. Zum Glück gibt es den Clear-Reduction-Algorithmus, dessen Hauptidee so aussieht: Es gelingt, die Z Koordinate größer erscheinen zu lassen als die höchste Z-Koordinate des am weitesten hinten liegenden Polygons, so ist das Zurücksetzen des Z-Buffers in jedem Frame nicht mehr nötig. In der Frame-Aufruf-Hauptschleife wird nun der Wert der Variable zAddValue so aufbereitet, dass man sie einfach nur von der aktuellen Z-Koordinate abziehen muss. In jedem Frame Aufruf wird die clear_translation Variable um 1 erhöht. Der Wert der zAddValue Variable wird gewährleistet durch die Multiplikation der clear_translation Variable mit der maximalen Tiefe des Z-Buffers, wie folgt:

```
inc(clear_translation);
zAddValue:=clear_translation*z_max;
```

Die bitweise Verschiebung nach links stellt sicher, dass der Wert der zAddValue in Fixedpoint konvertiert wird.

```
066:        tempZ:=z-(zAddValue shl 4);
```

Nun der Z-Buffer Test(Zeile 067) und falls der Pixel zugelassen ist, wird er auf die Surface gezeichnet(Zeile 069) und sein Z-Wert in den Z-Bufer reingeschrieben(Zeile068).

```
067:        if tempZ > zBuffer[x,y] then begin
068:          zBuffer[x,y]:=tempZ;
069:          PutPixel32(x, y, color, where);
070:        end;
```

Ende von dem If-Satz der in der Zeile 064 angewendet worden war.

```
071:    end;
```

Weitere Konstanten, die den nächsten Pixel repräsentieren, werden berechnet.

```
072:    CX1:=CX1-FDY12;
073:    CX2:=CX2-FDY23;
074:    CX3:=CX3-FDY31;
075:  end;
```

Die Start-Z-Koordinate wird ermittelt.

```
076:  initialZ:=initialZ+dy;
```

Sowie auch die Start Konstanten, die die waagerechten Zeilenanfänge repräsentieren.

```
077:        CY1:=CY1+FDX12;
078:        CY2:=CY2+FDX23;
079:        CY3:=CY3+FDX31;
080:    end;
081: end;
```

Der Umstieg von der FPU zu die CPU bringt unheimlichen Fortschritt was die Schnelligkeit des Programms betrifft.

Der oben gelistete Quellcode(Zeilen 001 bis 081) wurde aus dem Original der Publikation **Capens, 2004,** **http://devmaster.net/forums/topic/1145-advanced-rasterization/** in die Programmiersprache Pascal von mir übersetzt.

2.5 x86 Assembler Version der Routine

In dem assemblierten Quelltext ist vor einem übersetzten Codesegment immer sein hochsprachengleicher Befehl in den auskommentierten Zeilen zu finden.
Da der zu assemblierende Inhalt klar in den vorherigen Kapiteln ausführlich erläutert worden ist, liegt der Schwerpunkt nun auf der Assemblierung selbst, so auch die gegebenen Erklärungen.

Der Procedurheader bleibt unverändert.

```
001: procedure TriangleFLATZ(theX1,theY1,theZ1,
                             theX2,theY2,theZ2,theX3,
                             theY3,theZ3:single;color:dword;where:gfxImage);
```

Es befindet sich nur eine neue Variable(Zeile 014), die den Wert 16 speichert. Sonst ist hier alles beim alten.

```
002: var
003:     X1,X2,X3,Y1,Y2,Y3,Z1,Z2,Z3:longint;
004:     DX12,DX23,DX31,DY12,DY23,DY31:longint;
005:     FDX12,FDX23,FDX31,FDY12,FDY23,FDY31:longint;
006:     minx,maxx,miny,maxy:longint;
007:     C1,C2,C3:longint;
008:     CX1,CX2,CX3,CY1,CY2,CY3:longint;
009:     a,b,c,D:longint;
010:     Z:longint;
011:     initialZ:Longint;
012:     ddx,ddy:Longint;
013:     x,y:longint;
014:     my16:Longint;
```

Angefangen wird mit der Umrechnung ganzer Zahlen in Fixedpoint-Zahlen. Es passiert Folgendes: In der Register EAX lade ich die Zahl 16(Zeile 017), schreibe in die Variable my16, das EAX Register(Zeile 018), sodass wir in der Variable my16 die Zahl 16 haben.

```
015: begin
016: asm
017:     mov eax,16
018:     mov my16,eax
```

Multipliziert(Zeile 022), durch den Befehl fmul[1], wird Konstante theY1 mit der my16-Variable(Zeile 022). Gespeichert wird das Ergebnis in die Variable Y1(Zeil 023). Als Erstes wird finit[2] ausgeführt, was das Zurücksetzen des FPU auf Standartwerte zufolge hat. Danach werden die Variablen my16 und theY1 dank des Befehls FLD[3] geladen. Gerundet und gespeichert wird mit dem Befehl FISTP[4].

```
        // Y1:= round(16.0 * v1.y);
019:   finit
020:   fild my16
021:    fld theY1
022:   fmul st,st(1)
023:  fistp Y1
```

Hier passiert nichts anderes als in dem idee-gleichen Block(Zeile 019 bis 023) davor.

```
        // Y2:= round(16.0 * v2.y);
024:    fld theY2
025:   fmul st,st(1)
026:  fistp Y2
        // Y3:= round(16.0 * v3.y);
027:    fld theY3
028:   fmul st,st(1)
029:  fistp Y3
        // X1:= round(16.0 * v1.x);
030:    fld theX1
031:   fmul st,st(1)
032:  fistp X1
        // X2:= round(16.0 * v2.x);
033:    fld theX2
034:   fmul st,st(1)
035:  fistp X2
```

[1] **siehe: Rohde, 2007, S. 293**

[2] **siehe: Rohde, 2007, S. 256**

[3] **siehe: Rohde, 2007, S. 273**

[4] **siehe: Rohde, 2007, S. 275**

```
        // X3:= round(16.0 * v3.x);
036:    fld theX3
037:    fmul st,st(1)
038:    fistp X3
        // Z1:= round(16.0 * v1.z);
039:    fld theZ1
040:    fmul st,st(1)
041:    fistp Z1
        //Z2:= round(16.0 * v2.z);
042:    fld theZ2
043:    fmul st,st(1)
044:    fistp Z2
        // Z3:= round(16.0 * v3.z);
045:    fld theZ3
046:    fmul st,st(1)
047:    fistp Z3
048:    fstp st
```

Es ist möglich, die folgende Gleichung in zwei Teile zu unterteilen. Zu beachten ist, dass die bitweise Verschiebung vor der Multiplikation/Division steht.
Für die rechte Seite: Z2-Z1, bitweise Verschiebung nach rechts um 4, Multiplikation mit (Y3-Y1).
Für die linke Seite: Z3-Z1, bitweise Verschiebung nach rechts um 4, Multiplikation mit (Y2-Y1).
Nun subtrahiert man von der linken Seite die rechte Seite der Gleichung.

```
        //      a:= ((y2 - y1)*(z3 - z1) shr 4) - ((y3 - y1)*(z2 - z1) shr 4);
```

Der Befehl MOV[1] lädt in ein Register[2] einen Wert, der entweder explizit angegeben ober durch eine Variable übergeben wird.

```
049:    mov esi,z1
050:    mov edi,y1
```

Zuerst wird die rechte Seite der Gleichung mit der Hilfe der Befehle SUB[1] und IMUL[2] berechnet,

```
051:    mov eax,z2
052:    sub eax,esi
```

[1] **siehe: Rohde, 2007, S. 293**

[2] **siehe: Rohde, 2007, S. 256**

053: mov ebx,y3

054: sub ebx,edi

055: imul eax,ebx

056: shr eax,4

dann die linke.

057: mov ebx,z3

058: sub ebx,esi

059: mov ecx,y2

060: sub ecx,edi

061: imul ebx,ecx

062: shr ebx,4

Und nun wird die rechte Seite von der linken subtrahiert.

063: sub ebx,eax

064: mov a,ebx

Wie im Verfahren davor werden auch hier die Konstanten B und C(ab Zeile 080) berechnet.

// b:= ((x2 - x1)(z3 - z1) shr 4) - ((x3 - x1)*(z2 - z1) shr 4);*

065: mov edi,x1

066: mov eax,z2

067: sub eax,esi

068: mov ebx,x3

069: sub ebx,edi

070: imul eax,ebx

071: shr eax,4

072: mov ebx,z2

073: sub ebx,esi

074: mov ecx,x2

075: sub ecx,edi

076: imul ebx,ecx

077: shr ebx,4

[1] siehe: Rohde, 2007, S. 273

[2] siehe: Rohde, 2007, S. 275

```
078:    sub ebx,eax
079:    mov b,ebx

        //  c:= ((x2 - x1)*(y3 - y1) shr 4) - ((x3 - x1)*(y2 - y1) shr 4);
080:    mov esi,y1
081:   mov eax,y2
082:    sub eax,esi
083:    mov ebx,x3
084:    sub ebx,edi
085:   imul eax,ebx
086:    shr eax,4

087:    mov ebx,y3
088:    sub ebx,esi
089:    mov ecx,x2
090:    sub ecx,edi
091:   imul ebx,ecx
092:    shr ebx,4
093:    sub ebx,eax
094:    mov c,ebx
```

Die Gleichung, in der die Konstante D beschrieben wird, wurde abgewandelt

D:=(-X1*A) shr 4 + (Y1*B) shr 4 -(Z1*C) shr 4

da man die bitweise Verschiebung nach rechts um 4 ausklammern kann.

```
// D:= ((-x1*a) + (y1*b) - (z1*c)) shr 4;
```

In ESI ist noch die Variable Y1 gespeichert. So wird ESI mit B multipliziert.

```
095:   imul esi,b
```

Z1 wird mit C multipliziert und von ESI, wo sich momentan das Ergebnis von (Z1*C) befindet subtrahiert.

```
096:    mov eax,z1
097:   imul eax,c
098:    sub esi,eax
```

Mit X1 und A wird gleich verfahren. Bloß wird X1 durch den Befehl NEG[1] negativiert vor der Multiplikation mit A.

```
099:    mov eax,x1
100:    neg eax
101:   imul eax,a
```

Nun addieren wir den Rest und führen die bitweise Verschiebung nach rechts um 4 durch.

```
102:    add eax,esi
103:    shr eax,4
104:    mov d,eax
```

Es ist an der Zeit, die Seitenlängen herauszurechnen(Zeile 105 bis 110) und sie in den Fixedpoint Format umzuwandeln(Zeile 111 bis 112).

```
        //   DX12:= X1 - X2;
        //   FDX12:= DX12 shl 4;
105:    mov esi,x1
106:    mov edi,x2
107:    mov edx,x3
108:    mov eax,esi
109:    sub eax,edi
110:    mov DX12,eax
111:    shl eax,4
112:    mov FDX12,eax
```

Das gleiche für alle anderen Seitenlängen.

```
        //   DX23:= X2 - X3;
        //   FDX23:= DX23 shl 4;
113:    mov eax,edi
114:    sub eax,edx

115:    mov DX23,eax
116:    shl eax,4
117:    mov FDX23,eax
```

[1] siehe: Rohde, 2007, S. 178

```
        //   DX31:= X3 - X1;
        //   FDX31:= DX31 shl 4;
118:    sub edx,esi
119:    mov DX31,edx
120:    shl edx,4
121:    mov FDX31,edx

        //   DY12:= Y1 - Y2;
        //   FDY12:= DY12 shl 4;
122:    mov esi,y1
123:    mov edi,y2
124:    mov edx,y3
125:    mov eax,esi
126:    sub eax,edi
127:    mov DY12,eax
128:    shl eax,4
129:    mov FDY12,eax

        //   DY23:= Y2 - Y3;
        //   FDY23:= DY23 shl 4;
130:    mov eax,edi
131:    sub eax,edx
132:    mov DY23,eax
133:    shl eax,4
134:    mov FDY23,eax

        //   DY31:= Y3 - Y1;
        //   FDY31:= DY31 shl 4;
135:    sub edx,esi
136:    mov DY31,edx
137:    shl edx,4
138:    mov FDY31,edx
```

Das Umgebungsrechteck wird angepasst. Hier seine linke Koordinate:

```
        // minx:= min(x1,min(x2, x3)) shr 4;
```

Die Variablen X1,X2 und X3 werden provisorisch in die Register ESI,EDI und EDX gespeichert,

um später einen schnelleren Zugang zu diesen Werten zu haben.

```
139:    mov esi,x1
140:    mov edi,x2
141:    mov edx,x3

142:    mov eax,esi
143:    mov ebx,edi
144:    mov ecx,edx
```

Es wird verglichen, ob X3 kleiner als X2 ist, falls dem nicht so ist, werden die beiden Werte vertauscht. CMP[1] und der Sprung zu Markierung[2] werden hier verwendet.

```
145:    cmp ebx,ecx
146:     jb @l1
147:    mov ebx,ecx
148:@l1:
```

Hier wird das Ergebnis des letzten Tests mit der Variable X1 verglichen, falls sie kleiner ist, werden die Werte vertauscht und in die Variable MINX gespeichert.

```
149:    cmp eax,ebx
150:     jb @l2
151:    mov eax,ebx
152:@l2:
153:    shr eax,4
154:    mov minx,eax
```

Hier analog dazu die rechte Koordinate des Umgebungsrechteckes:

```
        //      maxx:= max(x1,max(x2, x3)) shr 4;

155:    mov eax,esi
156:    mov ebx,edi
157:    mov ecx,edx
```

[1] siehe: Rohde, 2007, S. 179

[2] siehe: Swan, 1996, S. 83

```
158:    cmp ebx,ecx
159:     ja @l3
160:    mov ebx,ecx
161:@l3:
162:    cmp eax,ebx
163:     ja @l4
164:    mov eax,ebx
165:@l4:
166:    shr eax,4
167:    mov maxx,eax
168:    mov esi,y1
169:    mov edi,y2
170:    mov edx,y3
```

Für seine oberste Koordinate:

```
        //      miny:= min(y1,min(y2, y3)) shr 4;
```

```
171:    mov eax,esi
172:    mov ebx,edi
173:    mov ecx,edx

174:    cmp ebx,ecx
175:     jb @l5
176:    mov ebx,ecx
177:@l5:
178:    cmp eax,ebx
179:     jb @l6
180:    mov eax,ebx
181:@l6:
182:    shr eax,4
183:    mov miny,eax
```

und zuletzt seine untere:

```
        //      maxy:= max(y1,max(y2, y3)) shr 4;
```

```
184:    mov eax,esi
185:    mov ebx,edi
```

```
186:    mov ecx,edx

187:    cmp ebx,ecx
188:     ja @l7
189:    mov ebx,ecx
190:@l7:
191:    cmp eax,ebx
192:     ja @l8
193:    mov eax,ebx
194:@l8:
195:    shr eax,4
196:    mov maxy,eax
```

In diesem Teil der Routine wird ein Teil der Half-Space Gleichung berechnet. Wir berechnen die Konstanten C1, C2 und C3. Diese werden später zur Vervollständigung der Gleichung gebraucht. Es werden einfache arithmetische Operatoren benutzt.

```
        //   C1:= DY12 * X1 - DX12 * Y1;

197:    mov eax,DX12
198:   imul eax,esi
199:    mov ebx,DY12
200:   imul ebx,X1
201:    sub ebx,eax
202:    mov C1,ebx
        //    C2:= DY23 * X2 - DX23 * Y2;

203:    mov eax,DX23
204:   imul eax,edi
205:    mov ebx,DY23
206:   imul ebx,X2
207:    sub ebx,eax
208:    mov C2,ebx
        //   C3:= DY31 * X3 - DX31 * Y3;

209:    mov eax,DX31
210:   imul eax,edx
211:    mov ebx,DY31
```

```
212:   imul ebx,X3
213:    sub ebx,eax
214:    mov C3,ebx
```

Die Korrektur des Startwertes, um nicht zwei Pixel auf dieselbe Stelle zu zeichnen, erfolgt durch das Prüfen der Längenwerte und falls nötig durch die Inkrementierung durch den Befehl INC[1] der Konstante C1, C2 und C3.

```
        //    if (DY12 < 0) or ((DY12 = 0) and (DX12 > 0)) then inc(C1);

215:    mov eax,DY12
216:    cmp eax,0
217:     jb @l9
218:    mov eax,DY12
219:    cmp eax,0
220:    jne @l10
221:    mov eax,DX12
222:    cmp eax,0
223:     jb @l10
224:@l9:
225:    inc C1
226:@l10:

        //    if (DY23 < 0) or ((DY23 = 0) and (DX23 > 0)) then inc(C2);
227:    mov eax,DY23
228:    cmp eax,0
229:     jb @l12
230:    mov eax,DY23
231:    cmp eax,0
232:    jne @l11
233:    mov eax,DX23
234:    cmp eax,0
235:     jb @l12
236:@l11:
237:    inc C2
238:@l12:
```

[1] **siehe: Rohde, 2007, S. 177**

```
        //    if (DY31 < 0) or ((DY31 = 0) and (DX31 > 0)) then inc(C3);
239:    mov eax,DY31
240:    cmp eax,0
241:    jb @l14
242:    mov eax,DY31
243:    cmp eax,0
244:    jne @l13
245:    mov eax,DX31
246:    cmp eax,0
247:    jb @l14
248:@l13:
249:    inc C3
250:@l14:
```

Die Konstanten CY1,CY2 und CY3 repräsentieren die waagerechte Startwerte des Umgebungsrechteckes bzw. der oberen zu prüfenden Grenze.

```
        //    CY1:= C1 + DX12 * (miny shl 4) - DY12 * (minx shl 4);

251:    mov esi,miny
252:    shl esi,4
253:    mov edi,minx
254:    shl edi,4
255:    mov eax,DX12
256:   imul eax,esi
257:    mov ebx,DY12
258:   imul ebx,edi
259:    sub eax,ebx
260:    add eax,C1
261:    mov CY1,eax
        //    CY2:= C2 + DX23 * (miny shl 4) - DY23 * (minx shl 4);

262:    mov eax,DX23
263:    imul eax,esi
264:    mov ebx,DY23
265:    imul ebx,edi
266:    sub eax,ebx
267:    add eax,C2
```

```
268:     mov CY2,eax
         //   CY3:= C3 + DX31 * (miny shl 4) - DY31 * (minx shl 4);

269:     mov eax,DX31
270:    imul eax,esi
271:    mov ebx,DY31
272:   imul ebx,edi
273:    sub eax,ebx
274:    add eax,C3
275:    mov CY3,eax
```

Division durch Null ist unzulässig, so wird falls C Null beherbergt ans Ende der Routine gesprungen.

```
         //  if (c shr 4) = 0 then exit;

276:    mov eax,c
277:    shr eax,4
278:    cmp eax,0
279:    jne @ok
280:    jmp @q
281:   @ok:
```

In die folgende Variable wird der Z-Startwert eingetragen. Hier ist zu beachten, dass bei einer Fixedpoint Division der Divisor durch den Formatskalar dividiert werden muss. Da dieser Skalar hier 16 ist, kann man statt einer Division durch 16 eine bitweise Verschiebung nach rechts um den Wert 4 durchführen. Neu hier ist der Befehl CDQ[1].

```
         //  initialZ:= (((A * -(minx)) shr 4 + ((B * -(miny)) shr 4 -D)) div (C shr 4));

282:   mov eax,minx
283:   neg eax
284:  imul eax,A
285:   shr eax,4
286:   mov ebx,miny
287:   neg ebx
288:  imul ebx,B
```

[1] siehe: Rohde, 2007, S. 85

```
289:    shr ebx,4
290:    sub ebx,D
291:    add eax,ebx
292:    mov ebx,C
293:    shr ebx,4
294:    cdq
295:   idiv ebx
296:    mov initialZ,eax
```

Wie im oberen Kapitel beschrieben, sind die folgenden Konstanten für die planare Gleichung zur Bestimmung der Z Koordinate notwendig.

```
        //    ddx:= ((-A) div (C shr 4));
```

```
297:    mov eax,A
298:    neg eax
299:    cdq
300:   idiv ebx
301:    mov ddx,eax
```

```
        //    ddy:= ((-B) div (C shr 4));
302:    mov eax,B
303:    neg eax
304:    cdq
305:   idiv ebx
306:    mov ddy,eax
```

Nun die Y-Schleife.

```
        //    for y:= miny to maxy do begin
```

```
307:      mov eax,miny
308:      mov y,eax
309:@YLoop:
```

Die Festlegung der Startwerte:

```
      //  CX1:=CY1;
310:        mov eax,CY1
```

```
311:        mov CX1,eax
      // CX2:=CY2;
312:        mov eax,CY2
313:        mov CX2,eax
      //CX3:=CY3;
314:        mov eax,CY3
315:        mov CX3,eax

      //Z:=initialZ;
316:        mov eax,initialZ
317:        mov Z,eax
```

Die X-Schleife:

```
          //      for x:=minx to maxx do begin
318:        mov eax,minx
319:        mov x,eax
320:        @XLoop:
```

Die Prüfung, ob der Pixel innerhalb des Dreieckes liegt:

```
          //          if (CX1 > 0) and (CX2 > 0) and (CX3 > 0) then begin
312:                  mov eax,CX1
322:                  cmp eax,0
323:                  jle @SkipPixel
324:                  mov eax,CX2
325:                  cmp eax,0
326:                  jle @SkipPixel
327:                  mov eax,CX3
328:                  cmp eax,0
329:                  jle @SkipPixel
```

Falls dies der Fall ist, wird die Z Koordinate des Pixels ermittelt.

```
          //                Z:=Z+ddx;
330:                  mov edx,z
331:                  add edx,ddx
332:                  mov z,edx
```

Hier wird der bereits beschriebene Clear-Reduction-Algorithmus angewandt.

```
        //              tempZ:=z-(zAddValue shl 4);
333:              mov eax,zAddValue
334:              shl eax,4
335:              sub edx,eax
336:              mov esi,edx   // tempZ
```

Falls der aktuelle Z-Wert kleiner als der im Z-Buffer ist, wird der aktuelle Pixel gezeichnet.

```
                  // if tempZ < zBuffer[x,y] then begin
                  //     PutPixel32(x, y, color, where);

337:              mov eax,x
338:              mov edx,y
339:              mov ecx,color

340:              imul edx,where.width
341:              add edx,eax
342:              shl edx,2

343:              mov edi,edx
344:              add edi,zBuffer
345:              mov eax,esi   // tempZ
346:              cmp eax,[edi]
347:              jle @SkipPixel

348:              mov [edi],eax

349:              add edx,where.data
350:              mov [edx],ecx
351:        @SkipPixel:
```

Hier wird der direkte Zugriff zum Speicher angewandt, die Konstante xscale enthält die Länge der Rasterzeile des Z-Buffers.

```
                //   putL(zbuffer+(y*xscale + x) shl 2,tempZ);
              //end;
```

Weitere Konstanten, die nach der Half-Space Formel den nächsten Pixel repräsentieren, werden berechnet.

```
        //      CX1:=CX1-FDY12;

352:        mov eax,CX1
353:        sub eax,FDY12
354:        mov CX1,eax
        //      CX2:=CX2-FDY23;

355:        mov eax,CX2
356:        sub eax,FDY23
357:        mov CX2,eax
        //      CX3:=CX3-FDY31;

358:        mov eax,CX3
359:        sub eax,FDY31
360:        mov CX3,eax
```

Der Inkrementen(Befehl INC[1]) Vergleich der X-Schleife:

```
361:    mov eax,x
362:    inc eax
363:    mov x,eax
364:    cmp eax,maxx
365:    jbe @Xloop
```

Der nächster Pixel des Z-Anfangswertes wird gesetzt.

```
        //      initialZ:=initialZ+ddy;
366:      mov eax,initialZ
367:      add eax,ddy
368:      mov initialZ,eax
```

[1] **siehe: Rohde, 2007, S. 177**

Weitere Konstanten der Half-Space Formel:

```
        //      CY1:=CY1+FDX12;

369:        mov eax,CY1
370:        add eax,FDX12
371:        mov CY1,eax
        //      CY2:=CY2+FDX23;

372:        mov eax,CY2
373:        add eax,FDX23
374:        mov CY2,eax
        //      CY3:=CY3+FDX31;

375:        mov eax,CY3
376:        add eax,FDX31
377:        mov CY3,eax
```

Nun das Ausklinken der Y-Schleife:

```
378:        inc y
379:        mov eax,y
380:        cmp eax,maxy
381:        jbe @Yloop
382:  @Q:
```

Das Ende dieser Routine:

```
383:end;
384:end;
```

3. Der Scan-Line Algorithmus in Theorie und Praxis

3.1 Der theoretische Aufbau

In der Scan-Line[1] Methode wird Rasterzeile für Rasterzeile das zu zeichnen geltende Dreieck durchgegangen. Dabei werden die Start- und Endpunkte der jeweiligen Rasterzeilen interpoliert, sodass mit einer einfachen Konstanten, die man zur letzten Start-Position bzw. End-Position addiert, neue Eckpunktepunkte entstehen.

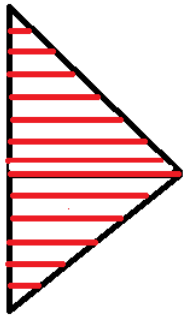

Bild 3.

Die Interpolation geschieht nach der folgenden Gleichung:

Schrittwerte = (Endgröße – Anfangsgröße) / Streckenlänge

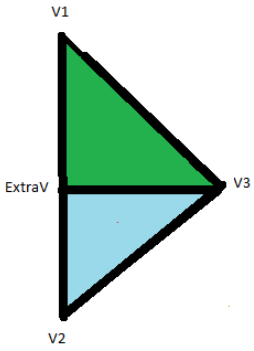

Bild 4.

Das zu zeichnen geltende Dreieck wird in zwei Dreiecke aufgeteilt. Es entsteht ein neuer Vertex. Es ist der Extravertex.

Als Beispiel betrachtet wird hier das obere nun neue entstandene grüne Dreieck.

Benutzt wird die interpolare Gleichung folgendermaßen:

stepXright = (V3.x-V1.x) / (V3.y-V1.y)

Dies bedeutet, dass man in (V3.y-V1.y) Schritten, falls man pro Schritt zu V1.x jedes mal stepXright hinzuaddiert, am Ende zu V3 kommt.

So wird die oben getroffene Aussage dafür verwendet, die Schrittwerte stepXleft und stepXright (auf dem Bild 5 blau markiert) zu dem Anfangsvertex (v1 im Bild 5) pro Rasterzeile (Im Bild 5 Gelb) zur jeweiligen Start- und End- Position (grün im Bild 5) zu addieren, um ihre neuen Werte zu berechnen.

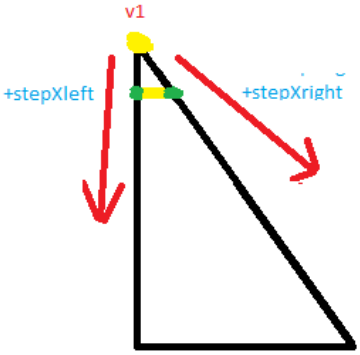

Bild 5.

[1] **siehe: LaMothe, 2004, S. 652**

3.2 Implementierung des Basisalgorithmus

In dem Prozedurheader hat sich nichts geändert.

```
001: Procedure TriangleFLATZ(theX1,theY1,theZ1,
                             theX2,theY2,theZ2,
                             theX3,theY3,theZ3:single;color:dword;where:gfxImage);
```

Die Variablen v1,v2,v3 speichern die Koordinate aller drei Vertexe. Der Typos tV3DVertex beherbergt die XYZ Koordinate des Vertex, außerdem seine Farbe im RGB Format. Der ExtraVertex „Vextra“ ist der neu gewonnene Vertex der das zu zeichnende Dreieck in der Mitte teilt.

```
002: var v1,v2,v3,Vextra:tV3DVertex;
```

Die Variable S ist temporär, die Variable DISTANCE speichert das Ergebnis der Prüfung, auf welcher Seite sich der ExtraVertex befindet.

```
003:    s,distance:single;
```

Folgende Variable ist notwendig, um Sub-Pixel-Accuracy durchzuführen.

```
004:    deltaY:single;
```

Der interpolierte Schritt nach rechts und nach links wird in folgenden Variablen gespeichert:

```
005:    stepXright,stepXleft:single;
```

sowie die momentane X Kursor Position links und rechts.

```
006:    Xright,Xleft:single;
```

Die X und Y Kursor Position, jedoch im Ganzzahlwert:

```
007:    XR,XL:longint;
```

Die Zähl-Variablen der Hauptschleifen:

```
008:    x,y,:longint;
```

Die aktuelle Z Koordinate für die rechte wie für die linke Seite des Dreieckes werden hier gespeichert(Zeile 009), sowie die interpolierten Schrittwerte(Zeil 010).

```
009:    Zright,Zleft:single;
010:    stepZright,stepZleft:single;
```

Der aktueller Z Wert und sein Inkrement:

```
011:     z,zAdd:single;
```

Es gilt, die Vertexe, die im Prozedurheader übermittelt wurden, in die dafür vorgesehenen Variablen zu speichern.

```
012: begin
013: v1.x:=theX1;
014: v1.y:=theY1;
015: v1.z:=theZ1;
016: v2.x:=theX2;
017: v2.y:=theY2;
018: v2.z:=theZ2;
019: v3.x:=theX3;
020: v3.y:=theY3;
021: v3.z:=theZ3;
```

Jetzt werden sie im Uhrzeigersinn sortiert. Die Prozedur swapV() vertauscht zwei tV3DVertexe.

```
022:   if (v2.y<v1.y) then swapV(v1,v2);
023:   if (v3.y<v1.y) then swapV(v1,v3);
024:   if (v2.y<v3.y) then swapV(v2,v3);
```

Falls die senkrechte Länge des Dreiecks Null ist, brechen wir das Zeichnen ab.

```
025: s:=v2.y-v1.y;
026: if s=0 then exit;
```

Der ExtraVertex wird berechnet.

```
027:  Vextra.x:=v1.x+(v2.x-v1.x)*(v3.y-v1.y)/s;
028:  Vextra.y:=v3.y;
029:  Vextra.z:=v1.z+(v2.z-v1.z)*(v3.y-v1.y)/s;
```

Und seine Distanz zum dritten Vertex.

```
030:  distance:=Vextra.x-v3.x;
```

Falls diese Distanz größer als Null ist, befindet sich der ExtraVertex auf der rechten Seite des Dreieckes.

```
031:  if (distance>0) then begin
```

Interpoliert werden nun die Schrittwerte des Inkrementen der rechten sowie der linken Seite auf der X-Axis.

```
032:    stepXright:=(Vextra.x-V1.x)/(Vextra.y-V1.y);
033:    stepXleft:=(V3.x-V1.x)/(V3.y-V1.y);
```

Der Sub-Pixel wird berechnet

```
034:    DeltaY:= ceil(v1.y) - v1.y;
```

Die Startwerte werden initialisiert. Hinzu addiert wird der Sub-Pixel-Wert.

```
035:    xLeft:=v1.x+stepXleft*DeltaY;
036:    xRight:=v1.x+stepXright*DeltaY;
```

Interpoliert werden nun die Schrittwerte des Inkrementen der rechten sowie der linken Seite auf der Z-Axis.

```
037:    stepZright:=(Vextra.z-V1.z)/(Vextra.y-V1.y);
038:    stepZleft:=(V3.z-V1.z)/(V3.y-V1.y);
```

Die Startwerte werden initialisiert. Hinzu addiert wird der Sub-Pixel-Wert.

```
039:    zLeft:=v1.z+stepZleft*DeltaY;
040:    zRight:=v1.z+stepZright*DeltaY;
```

Ein halbes Dreieck wird nun gezeichnet. Es wird von der oberen Kante des Dreiecks zu der Mitte pixelweise durchgegangen. Die Fließkomma-Werte werden aufgerundet.

```
041:    for y:=ceil(v1.y) to ceil(v3.y)-1 do begin
```

Die Fließkomma-Werte werden aufgerundet, da die Schleifen nur mit Ganzzahlwerten arbeiten können.

```
042:      XL:=ceil(Xleft);
043:      XR:=ceil(Xright);
```

Falls XR kleiner als XL ist, werden die beiden Werte miteinander vertauscht. Die Prozedur swapL vertauscht Zwei Longint Werte.

```
044:      if (XL>XR) then swapL(XL,XR);
```

Interpoliert wird nun das Z-Inkrement, das pro Pixel zu der aktuellen Z Koordinate addiert wird.

```
045:      zAdd:=(zright-zleft)/(xright-xleft+1);
```

Der Start-Z-Wert wird gesetzt.

```
046:    z:=zleft;
```

Nun die X-Schleife. Durchgegangen wird von links nach rechts.

```
047:    for x:=XL to XR-1 do begin
```

Z-Koordinate um den Schrittwert zAdd erhöhen:

```
048:            z:=z+zAdd;
```

Den Z-Buffer Test durchführen:

```
049:            if z > zBuffer[x,y] then begin
```

Falls der Z-Buffer Test bestanden worden ist, wird der Pixel mit der gegeben Farbe und den X und Y Koordinaten gezeichnet (Zeile 050). An diese Stelle im Z-Buffer wird der neue Z-Wert eingetragen.(Zeile 051).

```
050:                PutPixel32(x, y, color, where);
051:                zBuffer[x,y]:=z;
052:            end;
```

Ende der X-Schleife:

```
053:    end;
```

Die Startwerte werden um die vorher interpolierten Konstanten erhöht.

```
054:    Xleft:=Xleft+stepXleft;
055:    Xright:=Xright+stepXright;
056:    Zleft:=Zleft+stepZleft;
057:    Zright:=Zright+stepZright;
```

Ende der Y-Schleife:

```
058:  end;
```

Der zweite Teil des Dreieckes ist nun zu zeichnen.

Interpoliert werden nun die Schrittwerte der Inkrementen der rechten sowie der linken Seite auf der X-Axis.

```
059:    stepXright:=(V2.x-Vextra.x)/(V2.y-Vextra.y);
060:    stepXleft:=(V2.x-V3.x)/(V2.y-Vextra.y);
```

Der Sub-Pixel wird berechnet:

```
061:     DeltaY:= ceil(v3.y) - v3.y;
```

Die Startwerte werden initialisiert. Hinzu addiert wird der Sub-Pixel-Wert.

```
062:     xLeft:=v3.x+DeltaY*stepXleft;
063:     xRight:=vextra.x+DeltaY*stepXright;
```

Interpoliert werden nun die Schrittwerte der Inkrementen der rechten sowie der linken Seite auf der Z-Axis.

```
064:    stepZright:=(V2.z-Vextra.z)/(V2.y-Vextra.y);
065:    stepZleft:=(V2.z-V3.z)/(V2.y-Vextra.y);
```

Die Startwerte werden initialisiert. Hinzu addiert wird der Sub-Pixel-Wert.

```
066:    Zleft:=v3.z+stepZleft*DeltaY;
067:    Zright:=vextra.z+stepZright*DeltaY;
```

Der Rest des Dreiecks wird nun gezeichnet. Es wird von der Mitte des Dreiecks zu der unteren Kante pixelweise durchgegangen. Die Fließkomma-Werte werden aufgerundet.

```
068:    for y:=ceil(v3.y) to ceil(v2.y)-1 do begin
```

Die Fließkomma-Werte werden aufgerundet, da die Schleifen nur mit Ganzzahlwerten arbeiten können.

```
069:      XL:=ceil(Xleft);
070:      XR:=ceil(Xright);
```

Falls XR kleiner als XL ist, werden die beiden Werte miteinander vertauscht.

```
071:      if (XL>XR) then swapL(XL,XR);
```

Interpoliert wird nun das Z Inkrement, das pro Pixel zu der aktuellen Z Koordinate addiert wird.

```
072:      zAdd:=(zright-zleft)/(xright-xleft+1);
```

Der Start-Z-Wert wird gesetzt.

073: z:=zleft;

Nun die X-Schleife. Durchgegangen wird von links nach rechts.

074: for x:=XL to XR-1 do begin

Z-Koordinate um den Schrittwert zAdd erhöhen:

075: z:=z+zAdd;

Den Z-Buffer-Test durchführen:

076: if z > zBuffer[x,y] then begin

Falls der Z-Buffer Test bestanden worden ist, wird der Pixel mit der gegebenen Farbe (Zeile 077) an die Koordinaten X und Y gezeichnet. Auch wird der Wert bei denselben Koordinaten in den Z-Buffer gespeichert(Zeile 078).

077: PutPixel32(x, y, color, where);

078: zBuffer[x,y]:=z;

079: end;

Ende der X-Schleife:

080: end;

Die Startwerte werden um die vorher interpolierten Konstanten erhöht.

081: Xleft:=Xleft+stepXleft;

082: Xright:=Xright+stepXright;

083: Zleft:=Zleft+stepZleft;

084: Zright:=Zright+stepZright;

Ende der Y-Schleife:

085: end;

Falls der ExtraVertex sich auf der linken Seite des zu zeichnen geltenden Dreieckes befindet, tritt das folgender Codesegment in Kraft.

086: end else begin

Die gegebenen Eckpunkte des Dreiecks werden im Uhrzeigersinn sortiert.

087: if (v2.y<v1.y) then swapV(v1,v2);

088: if (v3.y<v1.y) then swapV(v1,v3);

089: if (v3.y<v2.y) then swapV(v2,v3);

Der ExtraVertex befindet sich auf der gleichen Höhe wie der Vertex V2

090: Vextra.y:=v2.y;

Interpoliert werden nun die Schrittwerte der Inkrementen der rechten sowie der linken Seite auf der X-Axis.

091: stepXright:=(V2.x-V1.x)/(V2.y-V1.y);

092: stepXleft:=(Vextra.x-V1.x)/(Vextra.y-V1.y);

Der Sub-Pixel wird berechnet

093: DeltaY:= ceil(v1.y) - v1.y;

Die Startwerte werden initialisiert. Hinzu addiert wird der Sub-Pixel-Wert.

*094: xLeft:=v1.x+DeltaY*stepXleft;*

*095: xRight:=v1.x+DeltaY*stepXright;*

Interpoliert werden nun die Schrittwerte des Inkrementen der rechten sowie der linken Seite auf der Z-Axis.

096: stepZright:=(V2.z-V1.z)/(V2.y-V1.y);

097: stepZleft:=(Vextra.z-V1.z)/(Vextra.y-V1.y);

Die Startwerte werden initialisiert. Hinzu addiert wird der Sub-Pixel-Wert.

*098: zLeft:=v1.z+stepZleft*DeltaY;*

*099: zRight:=v1.z+stepZright*DeltaY;*

Ein halbes Dreieck wird nun gezeichnet. Es wird von der oberen Kante des Dreiecks zu der Mitte pixelweise durchgegangen. Die Fließkomma-Werte werden aufgerundet.

100: for y:=ceil(v1.y) to ceil(v2.y)-1 do begin

Die Fließkomma-Werte werden aufgerundet, da die Schleifen nur mit Ganzzahlwerten arbeiten können.

101: XL:=ceil(Xleft);

102: XR:=ceil(Xright);

Falls XR kleiner als XL ist, werden die beiden Werte miteinander vertauscht.

103: if (XL>XR) then swapL(XL,XR);

Interpoliert wird nun das Z Inkrement, das pro Pixel zu der aktuellen Z Koordinate addiert wird.

104: zAdd:=(zright-zleft)/(xright-xleft+1);

Der Start-Z-Wert wird gesetzt.

105: z:=zleft;

Nun die X-Schleife. Durchgegangen wird von links nach rechts.

106: for x:=XL to XR-1 do begin

Z-Koordinate um den Schrittwert zAdd erhöhen.

107: z:=z+zAdd;

Den Z-Buffer Test durchführen:

108: if z > zBuffer[x,y] then begin

Falls der Z-Buffer-Test bestanden worden ist, wird der Pixel mit der gegeben Farbe(Zeile 109) an die Koordinaten X und Y gezeichnet. Auch wird der Wert bei denselben Koordinaten in den Z-Buffer gespeichert(Zeile 110).

109: PutPixel32(x, y, color, where);

110: zBuffer[x,y]:=z;

111: end;

Ende der X-Schleife:

112: end;

Die Startwerte werden um die vorher interpolierten Konstanten erhöht.

113: Xleft:=Xleft+stepXleft;

114: Xright:=Xright+stepXright;

115: Zleft:=Zleft+stepZleft;

116: Zright:=Zright+stepZright;

Ende der Y-Schleife:

117: end;

Die Vertexe werden erneut sortiert, falls sie nicht im Uhrzeigersinn angeordnet waren.

118: if (Vextra.y<v1.y) then swapV(v1,Vextra);

119: if (V3.y<Vextra.y) then swapV(v3,Vextra);

Die Höhe des Dreiecks darf nicht Null sein.

120: if (v3.y-v2.y)=0 then exit;

Interpoliert werden nun die Schrittwerte der Inkrementen der rechten sowie der linken Seite auf der X-Axis.

121: stepXright:=(V3.x-V2.x)/(V3.y-V2.y);

122: stepXleft:=(V3.x-Vextra.x)/(V3.y-V2.y);

Der Sub-Pixel wird berechnet.

123: DeltaY:= ceil(v2.y) - v2.y;

Die Startwerte werden initialisiert. Hinzu addiert wird der Sub-Pixel-Wert.

*124: xLeft:=vextra.x+DeltaY*stepXleft;*

*125: xRight:=v2.x+DeltaY*stepXright;*

Interpoliert werden nun die Schrittwerte der Inkrementen der rechten sowie der linken Seite auf der Z-Axis.

126: stepZright:=(V3.z-V2.z)/(V3.y-V2.y);

127: stepZleft:=(V3.z-Vextra.z)/(V3.y-V2.y);

Die Startwerte werden initialisiert. Hinzu addiert wird der Sub-Pixel-Wert.

*128: zLeft:=vextra.z+stepZleft*DeltaY;*

*129: zRight:=v2.z+stepZright*DeltaY;*

Der Rest von des Dreiecks wird nun gezeichnet. Es wird von der Mitte des Dreiecks zu der unteren Kante pixelweise durchgegangen. Die Fließkomma-Werte werden aufgerundet.

130: for y:=ceil(v2.y) to ceil(v3.y)-1 do begin

Die Fließkomma-Werte werden aufgerundet, da die Schleifen nur mit Ganzzahlwerten arbeiten können.

131: XL:=ceil(Xleft);

132: XR:=ceil(Xright);

Falls XR kleiner als XL ist, werden die beiden Werte miteinander vertauscht.

```
133:    if (XL>XR) then swapL(XL,XR);
```

Interpoliert wird nun das Z Inkrement, das pro Pixel zu der aktuellen Z Koordinate addiert wird.

```
134:    zAdd:=(zright-zleft)/(xright-xleft+1);
```

Der Start-Z-Wert wird gesetzt.

```
135:    z:=zleft;
```

Nun die X-Schleife. Durchgegangen wird von links nach rechts.

```
136:    for x:=XL to XR-1 do begin
```

Z-Koordinate um den Schrittwert zAdd erhöhen.

```
137:            z:=z+zAdd;
```

Den Z-Buffer Test durchführen:

```
138:            if z > zBuffer[x,y] then begin
```

Falls der Z-Buffer-Test bestanden worden ist, wird der Pixel mit der gegeben Farbe(Zeile 139) an die Koordinaten X und Y gezeichnet. Auch wird der Wert bei denselben Koordinaten in den Z-Buffer gespeichert(Zeile 140).

```
139:                PutPixel32(x, y, color, where);
140:                zBuffer[x,y]:=z;
141:            end;
```

Ende der X-Schleife:

```
142:    end;
```

Die Startwerte werden um die vorher interpolierten Konstanten erhöht.

```
143:    Xleft:=Xleft+stepXleft;
144:    Xright:=Xright+stepXright;
145:    Zleft:=Zleft+stepZleft;
146:    Zright:=Zright+stepZright;
```

Ende der Y-Schleife:

```
147:  end;
```

148: end;

Ende der Routine.

149: end;

3.3 x86 Assembler Version der Routine

An dem Prozedurheader hat sich nichts geändert. Es ist möglich, die in dieser Arbeit geschriebenen Prozeduren einfach auszutauschen.

```
0001: procedure TriangleFLATZ(theX1,theY1,theZ1,
                              theX2,theY2,theZ2,
                              theX3,theY3,theZ3:single;
                              color:dword;where:gfxImage);
```

Der folgende Variablenblock ist unverändert.

```
0002: var v1,v2,v3,Vextra:tV3DVertex;
0003:     s,distance:single;
0004:     deltaY:single;
0005:     stepXright,stepXleft:single;
0006:     Xright,Xleft:single;
0007:     XR,XL:longint;
```

Neu hinzugekommen ist die CWR Variable, in ihr wird die Control-Word-Register-Einstellung für die Rundungsart gespeichert(Zeile 0008).

```
0008:     cwr:word;
0009:     x,y:longint;
0010:     Ycounter:longint;
0011:     Zright,Zleft:single;
0012:     stepZright,stepZleft:single;
0013:     z,Zadd:single;
0014:     TempS:single;
```

Am Anfang dieser Routine wird mit der Zuordnung der Variablen begonnen.

```
0015: begin
0016: asm
```

Auf den Stack[1] werden die Register ESI und EDI gespeichert, denn sie werden in der Routine verändert. Am Ende werden sie wieder vom Stack wiederhergestellt.

```
0017:  push esi
0018:  push edi
```

[1] siehe: Backer, 2003, S. 82

Hier werden die Fließkomma-Werte als 32bit Zahlen behandelt, damit es schneller geht, sie durch den Speicher zu verschieben.

```
        // v1.x:=theX1;
        // v1.y:=theY1;
0019: mov eax,theX1
0020: mov ebx,theY1
0021: mov v1.x,eax
0022: mov v1.y,ebx

        // v1.z:=theZ1;
        // v2.x:=theX2;
0023: mov eax,theZ1
0024: mov ebx,theX2

0025: mov v1.z,eax
0026: mov v2.x,ebx

        // v2.y:=theY2;
        // v2.z:=theZ2;
0027: mov eax,theY2
0028: mov ebx,theZ2

0029: mov v2.y,eax
0030: mov v2.z,ebx

        // v3.x:=theX3;
        // v3.y:=theY3;
0031: mov eax,theX3
0032: mov ebx,theY3

0033: mov v3.x,eax
0034: mov v3.y,ebx

        // v3.z:=theZ3;
0035: mov eax,theZ3
0036: mov v3.z,eax
```

Die Eckpunkte des Dreiecks werden sortiert. Mit dem Befehl fcom wird verglichen, danach wird der Status-Wert-Register in AX gespeichert und dann wird der AH Register in den Flag-Register geladen, erst nach diesem Verfahren kann ein bedingter Sprung durchgeführt werden.

// if (v2.y<v1.y) then begin

0037: fld v2.y
0038: fld v1.y
0039: fcom st(1)
0040: fstsw ax
0041: sahf
0042: jc @Below1

Falls das Vertauschen notwendig ist, wird das folgende Codesegment ausgeführt:

0043: mov eax,v1.x
0044: mov ebx,v2.x
0045: mov v1.x,ebx
0046: mov v2.x,eax

0047: mov eax,v1.y
0048: mov ebx,v2.y
0049: mov v1.y,ebx
0050: mov v2.y,eax

0051: mov eax,v1.z
0052: mov ebx,v2.z
0053: mov v1.z,ebx
0054: mov v2.z,eax

Sonst wird zu der folgenden Markierung gesprungen.

// end;
0055: @Below1:

Sortieren und falls keine Vertauschung notwendig ist, nach unten springen.

// if (v3.y<v1.y) then swapV(v1,v3);

0056: fld v3.y

0057: fld v1.y
0058: fcom st(1)
0059: fstsw ax
0060: sahf
0061: jc @Below2

Werte vertauschen.
0062: mov eax,v1.x
0063: mov ebx,v3.x
0064: mov v1.x,ebx
0065: mov v3.x,eax

0066: mov eax,v1.y
0067: mov ebx,v3.y
0068: mov v1.y,ebx
0069: mov v3.y,eax

0070: mov eax,v1.z
0071: mov ebx,v3.z
0072: mov v1.z,ebx
0073: mov v3.z,eax

// end;
0074: @Below2:

Neu ist hier der Befehl WAIT [1].
0075: wait
0076: finit

Sortieren und falls keine Vertauschung notwendig ist, nach unten springen.
// if (v2.y<v3.y) then swapV(v2,v3);

0077: fld v2.y
0078: fld v3.y

[1] siehe: Rohde, 2007, S. 231

0079: fcom st(1)

0080: fstsw ax

0081: sahf

0082: jc @Below3

Werte vertauschen.

0083: mov eax,v2.x

0084: mov ebx,v3.x

0085: mov v2.x,ebx

0086: mov v3.x,eax

0087: mov eax,v2.y

0088: mov ebx,v3.y

0089: mov v2.y,ebx

0090: mov v3.y,eax

0091: mov eax,v2.z

0092: mov ebx,v3.z

0093: mov v2.z,ebx

0094: mov v3.z,eax

// end;

0095: @Below3:

0096: wait

0097: finit

Seitenlänge berechnen:

// s:=v2.y-v1.y;

0098: fld v2.y

0099: fld v1.y

0100: fsubp st(1),st

0101: fst s

Falls sie Null ist, die Routine unterbrechen. Dies geschieht mit dem Springen zu der @MainExit Sprungmarke.

// if s=0 then exit;

0102: fld s

0103: fldz

0104: fcom st(1)

0105: fstsw ax

0106: sahf

0107: jz @MainExit

Den Extra-Vertex berechnen:

// Vextra.x:=v1.x+(v2.x-v1.x)(v3.y-v1.y)/s;*

// Vextra.y:=v3.y;

0108: fld v2.x

0109: fld v1.x

0110: fsubp st(1),st

0111: fld v3.y

0112: fst Vextra.y

0113: fld v1.y

0114: fsubp st(1),st

0115: fmulp st(1),st

0116: fld s

0117: fst st(2)

0118: fdivp st(1),st

0119: fadd v1.x

0120: fst Vextra.x

Die Distanz herausrechnen:

// distance:=Vextra.x-v3.x;

0121: fld v3.x

0122: fsubp st(1),st

0123: fst distance

Es wurde die Distanz-Kalkulation oben eingeschoben, da das Subtrahieren der Variable V3.X von Vextra.X so geschrieben worden ist, dass man nicht doppelt die Variable Vextra.X neu in den FPU Stack laden muss.

Weiter mit der Bestimmung der Variable „Vextra“.

// Vextra.z:=v1.z+(v2.z-v1.z)(v3.y-v1.y)/s;*

0124: fld v2.z

```
0125:   fld v1.z
0126:  fsubp st(1),st
0127:   fld v3.y
0128:   fld v1.y
0129:  fsubp st(1),st
0130:   fmulp
0131:   fld st(2)
0132:   fdivp st(1),st
0133:   fadd v1.z
0134:   fst Vextra.z
```

Geprüft wird nun, ob der Vextra sich auf der linken oder rechten Seite des Dreiecks befindet.

```
    // if (distance>0) then begin
```

Falls V3 sich auf der rechten Seite befindet, wird zu der Markierung @distanceJMP gesprungen.

```
 0135: finit
0136:   fldz
0137:   fld distance
0138:   fcom st(1)
0139:  fstsw ax
0140:   sahf
0141:   jbe @distanceJMP
```

Interpoliert wird nun der Schrittwert, den man in jede Zeile zu dem aktuellen X-Wert auf der rechten Seite addiert.

```
    //   stepXright:=(Vextra.x-V1.x)/(Vextra.y-V1.y);
```

```
0142:   fld Vextra.x
0143:   fld V1.x
0144:  fsubp st(1),st

0145:   fld Vextra.y
0146:   fld V1.y
0147:  fsubp st(1),st

0148:  fdivp st(1),st
```

0149: fst stepXright

0150: finit

Interpolation des Schrittwertes für die linke Seite.

// stepXleft:=(V3.x-V1.x)/(V3.y-V1.y);

0151: fld V3.x

0152: fld V1.x

0153: fsubp st(1),st

0154: fld V3.y

0155: fld V1.y

0156: fsubp st(1),st

0157: fdivp st(1),st

0158: fst stepXleft

Sub-Pixel-Wert.

// DeltaY:= ceil(v1.y) - v1.y;

0159: fld v1.y

Da hier keine normale Rundung in Frage kommt, sondern immer aufgerundet werden muss, folgt nun die „Einstellung der Rundungsart im Control-Word-Register“[1].

0160: mov ax,037Fh

0161: or ax,0000100000000000b

0162: mov cwr,ax

0163: fclex

0164: fldcw cwr

Jetzt kann sicher mit dem Befehl FRNDINT[2] gerundet werden.

0165: frndint

0166: fld v1.y

0167: fsubp st(1),st

[1] siehe: Rohde, Roming, 2006, S. 156

[2] siehe: Rohde, 2007, S. 298

```
0168:   fst DeltaY
0169:   fst st(2)
```

Es wird zu dem Anfangswert der linken Rasterzeile-Grenze die Sub-Pixel-Accuracy-Wert hinzuaddiert.

```
        //   xLeft:=v1.x+stepXleft*DeltaY;
0170:   fld stepXleft
0171:  fmulp st(1),st
0172:   fld v1.x
0173:  faddp st(1),st
0174:   fst xLeft
```

Dies geschieht auch für den rechten Anfangspunkt.

```
        //   xRight:=v1.x+stepXright*DeltaY;
0175:   fld stepXright
0176:   fld st(3)
0177:  fmulp st(1),st
0178:   fld v1.x
0179:  faddp st(1),st
0180:   fst xRight
```

Der Z-Schritt, pro Rasterzeile, nach rechts wird berechnet.

```
//   stepZright:=(Vextra.z-V1.z)/(Vextra.y-V1.y);
0181:  finit
0182:   fld Vextra.y
0183:   fld V1.y
0184:  fsubp st(1),st
0185:   fst st(2)
0186:   fld Vextra.z
0187:   fld V1.z
 0188: fsubp st(1),st
0189:   fld st(1)
0190:  fdivp st(1),st
0191:   fst stepZright
```

Der Z-Schritt, pro Rasterzeile, nach links wird berechnet.

```
        //   stepZleft:=(V3.z-V1.z)/(V3.y-V1.y);
0192: finit
0193:    fld v3.y
0194:    fld v1.y
0195:  fsubp st(1),st
0196:    fst st(2)
0197:    fld v3.z
0198:    fld v1.z
0199:  fsubp st(1),st
0200:    fld st(1)
0201:  fdivp st(1),st
0202:    fst stepZleft
```

Die linke Start-Z-Koordinate wird gesetzt.

```
        //    zLeft:=v1.z+stepZleft*DeltaY;
0203: finit
0204:   fld stepZleft
0205:   fld DeltaY
0206: fmulp st(1),st
0207:   fld v1.z
0208: faddp st(1),st
0209:   fst zLeft
```

Die rechte Start-Z-Koordinate wird gesetzt.

```
        // zRight:=v1.z+stepZright*DeltaY;
0210: finit
0211:   fld stepZright
0212:   fld DeltaY
0213: fmulp st(1),st
0214:   fld v1.z
0215: faddp st(1),st
0216:   fst zRight
```

Nun die Y-Schleife:

```
        //    for y:=ceil(v1.y) to ceil(v3.y)-1 do begin
```

0217: finit
0218: fld v1.y
0219: fclex

Aufgerundet (Zeile 0223) wird durch das Setzen des Control-Word-Registers(Zeile 0220) auf „aufrunden“.
0220: fldcw cwr
0221: fist y
0222: fld v3.y
0223: fist Ycounter
0224: dec Ycounter

Falls die Höhe des zu zeichnen geltenden Dreieckes Null entspricht, wird die Y-Schleife übersprungen.
0225: mov edx,y
0226: cmp edx,Ycounter
0227: ja @Yequ4

Markierung für den Anfangspunkt der Y-Schleife:
0228: @Yloop4:

Aufgerundet wird der Xleft Wert und in die Integer-Variable XL gespeichert.
// XL:=ceil(Xleft);
0229: finit
0230: fld Xleft
0231: fclex
0232: fldcw cwr
0233: fist XL

Aufgerundet wird der Xright Wert und in die Integer-Variable XR gespeichert.
// XR:=ceil(Xright);

0234: fld Xright
0235: fist XR
0236: wait

Dies ist für den Zähler der X-Schleife notwendig.

Vertauscht werden die Werte, falls X-Links größer als X-Rechts ist.

```
        //   if (XL>XR) then swapL(XL,XR);

0237:  mov eax,XL
0238:  mov ebx,XR
0239:  cmp eax,ebx
0240:  jle @skipit1

0241:  mov XL,ebx
0242:  mov XR,eax

0243: @skipit1:
```

Interpoliert wird auch das Z-Inkrement.

```
        //   zAdd:=(zright-zleft)/(xright-xleft+1);

0244:    fld xright
0245:    fld xleft
0246:  fsubp st(1),st
0247:   fld1
0248:  faddp st(1),st
0249:    fst st(2)
0250:    fld zright
0251:    fld zleft
0252:  fsubp st(1),st
0253:    fld st(1)
0254:  fdivp st(1),st
0255:    fst zAdd
```

Z wird an den Anfangspunkt gesetzt.

```
        //      z:=zleft;
0256:   mov eax,zleft
0257:   mov z,eax
```

Nun die X-Schleife:

```
        // for x:=XL to XR-1 do begin
0258:  mov ebx,XL
0259:  cmp ebx,XR
0260:  je @XLRLequ4
0261: @Xloop4:
```

Z-Berechnen:

```
        //    z:=z+zAdd;
0262:   finit
0263:    fld z
0264:    fld zAdd
0265:  faddp st(1),st
0266:    fst st(1)
0267:   fstp z
0268:   wait
```

Z-Buffer mit der aktuellen Z-Koordinate prüfen. Falls sie größer ist als die bereits im Z-Buffer enthaltene, den Pixel auf die vorgegebene Surface zeichnen und den Z-Wert in den Z-Buffer eintragen.

```
        //    if z > zBuffer[x,y] then begin

0269:   mov eax,y
0270:   mul where.width
0271:   add eax,ebx
0272:   shl eax,2

0273:   mov esi,eax
0274:   mov edi,eax

0275:   add esi,zbuffer

0276:   fld dword ptr [esi]
0277:   fld st(1)
0278:  fcom st(1)

0279: fstsw ax
```

```
0280:  sahf
0281:    jc @Ztest4

         //                    PutPixel32(x, y, color, where);
0281:    add edi,where.data
0282:    mov eax,color
0283:    mov [edi],eax
         //                     putS(zbuffer+(y*xscale+x) shl 2,z);
0284:    mov eax, z
0285:    mov [esi],eax
         //     end;
0286: @Ztest4:
```

Den Zähler der Schleife gilt es zu inkrementieren und zu prüfen, ob er das Ende der Scan-Line erreicht hat.

```
0287: inc ebx
0288: mov edx,XR
0289: cmp ebx,edx
0290: jb @Xloop4
0291: @XLRLequ4:
```

Der linke X-Anfangspunkt der Rasterzeile wird mit dem dazugehörigen interpolierten Wert addiert.

```
0292: finit
         //     Xleft:=Xleft+stepXleft;
0293:   fld Xleft
0294:   fld stepXleft
0295: faddp st(1),st
0296:   fst Xleft
```

So auch der rechte.

```
         //    Xright:=Xright+stepXright;
0297:   fld Xright
0298:   fld stepXright
0299: faddp st(1),st
0300:   fst Xright
```

Der linke Z-Anfangspunkt der Rasterzeile wird mit dem dazugehörigen interpolierten Wert addiert.

```
        //   Zleft:=Zleft+stepZleft;
0301:   fld Zleft
0302:   fld stepZleft
0303: faddp st(1),st
0304:   fst Zleft
```

Der rechte Z-Anfangspunkt der Rasterzeile wird mit dem dazugehörigen interpolierten Wert addiert.

```
        //   Zright:=Zright+stepZright;
0305:   fld Zright
0306:   fld stepZright
0307: faddp st(1),st
0308:   fst Zright
```

Die Y-Schleife wird aktualisiert.

```
0309: inc y
0310: mov eax,y
0311: cmp eax,Ycounter
0312: jbe @Yloop4

0313: @Yequ4:
```

Jetzt ist es an der Zeit, die zweite Hälfte des Dreieckes zu zeichnen.

Interpoliert wird der rechte X-Schrittwert.

```
        //   stepXright:=(V2.x-Vextra.x)/(V2.y-Vextra.y);
0314: finit
0315:    fld V2.y
0316:    fld Vextra.y
0317:  fsubp st(1),st
0318:    fst st(1)
0319:    fld V2.x
0320:    fld Vextra.x
0321:  fsubp st(1),st
0322:   fld st(2)
0323:  fdivp st(1),st
```

```
0324:    fst stepXright
```

Interpoliert wird der linke X-Schrittwert.

```
         //   stepXleft:=(V2.x-V3.x)/(V2.y-Vextra.y);
0325:    finit
0326:   fld v2.y
0327:   fld Vextra.y
0328: fsubp st(1),st
0329:   fst st(1)
0330:   fld V2.x
0331:   fld V3.x
0332: fsubp st(1),st
0333:   fld st(2)
0334: fdivp st(1),st
0335:   fst stepXleft
```

Sub-Pixel-Wert wird berechnet.

```
         //    DeltaY:= ceil(v3.y) - v3.y;
0336:   finit
0337:     fld v3.y
0338:   fclex
0339:   fldcw cwr
0340: frndint
0341:     fld v3.y
0342:   fsubp st(1),st
0343:     fst DeltaY
```

Der linke Anfangspunkt wird um den Sub-Pixel-Wert korrigiert.

```
         //   xLeft:=v3.x+DeltaY*stepXleft;
0344:   fld stepXleft
0345: fmulp st(1),st
0346:   fld v3.x
0347: faddp st(1),st
0348:   fst xLeft
```

Der rechte Anfangspunkt wird um den Sub-Pixel-Wert korrigiert.

```
        //   xRight:=vextra.x+DeltaY*stepXright;
0349:   fld stepXright
0350:   fld DeltaY
0351: fmulp st(1),st
0352:   fld vextra.x
0353: faddp st(1),st
0354:   fst xRight
```

Der rechte Z-Schrittwert wird interpoliert.

```
        //   stepZright:=(V2.z-Vextra.z)/(V2.y-Vextra.y);
0355:   fld V2.y
0356:   fld Vextra.y
0357: fsubp st(1),st
0358:   fst st(1)
0359:   fld V2.z
0360:   fld Vextra.z
0361: fsubp st(1),st
0362:   fld st(1)
0363: fdivp st(1),st
0364:   fst stepZright
```

Der linke Z-Schrittwert wird interpoliert

```
        // stepZleft:=(V2.z-V3.z)/(V2.y-Vextra.y);
0365: finit
0366:   fld V2.y
0367:   fld Vextra.y
0368: fsubp st(1),st
0369:   fst st(1)
0370:   fld V2.z
0371:   fld v3.z
0372: fsubp st(1),st
0373:   fld st(1)
0374: fdivp st(1),st
0375:   fst stepZleft
```

Der linke Z-Anfangspunkt wird durch den Sub-Pixel-Wert korrigiert.

```
        //   Zleft:=v3.z+stepZleft*DeltaY;

0375:   fld stepZleft
0376:   fld DeltaY
0377: fmulp st(1),st
0378:   fld v3.z
0379: faddp st(1),st
0380:   fst Zleft
```

Der rechte Z-Anfangspunkt wird durch den Sub-Pixel-Wert korrigiert.

```
        // Zright:=vextra.z+stepZright*DeltaY;
0381: finit
0382:   fld stepZright
0383:   fld DeltaY
0384: fmulp st(1),st
0385:   fld Vextra.z
0386: faddp st(1),st
0387:   fst Zright
```

Die Y-Schleife:

```
        // for y:=ceil(v3.y) to ceil(v2.y)-1 do begin
0388:   finit
0389:    fld v3.y
0390:   fclex
0391:   fldcw cwr
0392:    fist y

0393:    fld v2.y
0394:    fist Ycounter
0395:    dec Ycounter

0396:   mov edx,y
0397:   cmp edx,Ycounter
0398:   ja @Yequ3

0399: @Yloop3:
```

Der Fließkomma-Wert Xleft wird aufgerundet und in XL gespeichert.

```
        // XL:=ceil(Xleft);
0401:  finit
0402:   fld Xleft
0403:  fclex
0404:  fldcw cwr
0405:   fist XL
```

Der Fließkomma-Wert Xright wird aufgerundet und in XR gespeichert.

```
        // XR:=ceil(Xright);
0406:   fld Xright
0407:  fist XR
0408:     wait
```

Es wird verglichen, ob XL größer als XR ist, falls so, werden diese Werte vertauscht.

```
        //   if (XL>XR) then swapL(XL,XR);
0409:  mov eax,XL
0410:  mov ebx,XR
0411:  cmp eax,ebx
0412:  jle @skipit2

0413:  mov XL,ebx
0414:  mov XR,eax

0415: @skipit2:
```

Der Z-Schrittwert, der innerhalb der X-Schleife gebraucht wird, wird interpoliert.

```
//   zAdd:=(zright-zleft)/(xright-xleft+1);

0416:    fld xright
0417:    fld xleft
0418:  fsubp st(1),st
0419:   fld1
0420:  faddp st(1),st
0421:    fst st(2)
0422:    fld zright
```

```
0423:   fld zleft
0424:  fsubp st(1),st
0425:   fld st(1)
0426:  fdivp st(1),st
0427:   fst zAdd
```

Z wird auf den Startwert gesetzt.

```
        //     z:=zleft;
0428:   mov eax,zleft
0429:   mov z,eax
```

Nun die X-Schleife:

```
        // for x:=XL to XR-1 do begin
0430:  mov ebx,XL
0431:  cmp ebx,XR
0432:  je @XLRLequ3
0433: @Xloop3:
```

Z um den Z-Schrittwert erhöhen.

```
        //    z:=z+zAdd;
0434:   finit
0435:    fld z
0436:    fld zAdd
0437:   faddp st(1),st
0438:    fst st(1)
0439:   fstp z
0440:    wait
```

Den Z-Buffer Test ausführen. Falls der aktuelle Z-Wert kleiner ist als der Wert, der sich bereits im Z-Buffer befindet, wird das folgende Codesegment übersprungen.

```
        //    if z > zBuffer[x,y] then begin

0441:   mov eax,y
0442:   mul where.width
0443:   add eax,ebx
0444:   shl eax,2
```

```
0445:   mov esi,eax
0446:   mov edi,eax
0447:   add esi,zbuffer

0448:   fld dword ptr [esi]
0449:   fld st(1)
0450:  fcom st(1)

0451: fstsw ax
0452:  sahf
0453:    jc @Ztest3
```

Den aktuellen Pixel zeichnen.

```
        //                      PutPixel32(x, y, color, where);

0454:    add edi,where.data
0455:    mov eax,color
0456:    mov [edi],eax
```

Sowie die aktuelle Z-Koordinate in den Z-Buffer eintragen.

```
        //                      zBuffer[x,y]:=z;

0457:    mov eax, z
0458:    mov [esi],eax
        //     end;

0459: @Ztest3:

0460: inc ebx
0461: mov edx,XR
0462: cmp ebx,edx
0463: jb @Xloop3
0464: @XLRLequ3:
```

Ende der X-Schleife.

Die linke X-Startkoordinate wird um einen Schritt erhöht.

0465: finit

// Xleft:=Xleft+stepXleft;

0466: fld Xleft

0467: fld stepXleft

0468: faddp st(1),st

0469: fst Xleft

Die rechte X-Startkoordinate wird um einen Schritt erhöht.

// Xright:=Xright+stepXright;

0470: fld Xright

0471: fld stepXright

0472: faddp st(1),st

0473: fst Xright

Die linke Z-Startkoordinate wird gesetzt.

// Zleft:=Zleft+stepZleft;

0474: fld Zleft

0475: fld stepZleft

0476: faddp st(1),st

0477: fst Zleft

Die rechte Z-Startkoordinate wird gesetzt.

// Zright:=Zright+stepZright;

0478: fld Zright

0479: fld stepZright

0480: faddp st(1),st

0481: fst Zright

Der Zähler der Y-Schleife wird erhöht.

0482: inc y

0483: mov eax,y

0484: cmp eax,Ycounter

0485: jbe @Yloop3

0486: @Yequ3:

Hier endet das Zeichnen des Dreiecks, dessen ExtraVertex auf der rechten Seite des Dreiecks war. Es wird nun zu dem Ende der Routine gesprungen.

0487: jmp @MainExit

Falls sich der ExtraVertex auf der linken Seite des Dreiecks befindet, ist hier der Startpunkt der Zeichen-Routine.

0488: @distanceJMP:

Die Ecke des Dreiecks werden so sortiert, dass sie danach im Uhrzeigersinn angeordnet sind.

0489: finit

// if (v2.y<v1.y) then begin

0490: fld v2.y

0491: fld v1.y

0492: fcom st(1)

0493: fstsw ax

0494: sahf

0495: jc @Below4

Falls nicht, werden die Werte vertauscht.

0496: mov eax,v1.x

0497: mov ebx,v2.x

0498: mov v1.x,ebx

0499: mov v2.x,eax

0500: mov eax,v1.y

0501: mov ebx,v2.y

0502: mov v1.y,ebx

0503: mov v2.y,eax

0504: mov eax,v1.z

0505: mov ebx,v2.z

0506: mov v1.z,ebx

0507: mov v2.z,eax

// end;

0508: @Below4:

Vertexe werden sortiert.

// if (v3.y<v1.y) then swapV(v1,v3);

0509: fld v3.y
0510: fld v1.y
0511: fcom st(1)
0512: fstsw ax
0513: sahf
0514: jc @Below5

Die Werte werden vertauscht.

0515: mov eax,v1.x
0516: mov ebx,v3.x
0517: mov v1.x,ebx
0518: mov v3.x,eax

0519: mov eax,v1.y
0520: mov ebx,v3.y
0521: mov v1.y,ebx
0522: mov v3.y,eax

0523: mov eax,v1.z
0524: mov ebx,v3.z
0525: mov v1.z,ebx
0526: mov v3.z,eax

// end;
0526: @Below5:
0527: wait
0528: finit

Alle drei Vertexe werden sortiert.

// if (v2.y<v3.y) then swapV(v2,v3);

0529: fld v3.y

0530: fld v2.y

0531: fcom st(1)

0532: fstsw ax

0533: sahf

0534: jc @Below6

Die Werte werden vertauscht.

0535: mov eax,v2.x

0536: mov ebx,v3.x

0537: mov v2.x,ebx

0538: mov v3.x,eax

0539: mov eax,v2.y

0540: mov ebx,v3.y

0541: mov v2.y,ebx

0542: mov v3.y,eax

0543: mov eax,v2.z

0544: mov ebx,v3.z

0545: mov v2.z,ebx

0546: mov v3.z,eax

// end;

0547: @Below6:

Der Variablen Vextra.y wird der Wert der Variable V2.y zugeordnet.

// Vextra.y:=v2.y;

0548: mov eax,v2.y

0549: mov Vextra.y,eax

Der rechte X-Schrittwert wird interpoliert

// stepXright:=(V2.x-V1.x)/(V2.y-V1.y);

0550: fld v2.y

0551: fld v1.y

0552: fsubp st(1),st

0553: fst st(1)

```
0554:   fld v2.x
0555:   fld v1.x
0556: fsubp st(1),st
0557:   fld st(1)
0558: fdivp st(1),st
0559:   fst stepXright
```

Der linke X-Schrittwert wird interpoliert

```
        //    stepXleft:=(Vextra.x-V1.x)/(Vextra.y-V1.y);
```

```
0560: finit
0561:   fld Vextra.y
0562:   fld V1.y
0563: fsubp st(1),st
0564:   fst st(1)
0565:   fld Vextra.x
0566:   fld V1.x
0567: fsubp st(1),st
0568:   fld st(1)
0569: fdivp st(1),st
0570:  fst stepXleft
```

Der Sub-Pixel-Wert wird ermittelt.

```
        //    DeltaY:= ceil(v1.y) – v1.y;
```

```
0571:    fld v1.y
0572:    fst st(1)
0573:    mov ax,037Fh
0574:     or ax,0000100000000000b
0575:    mov cwr,ax
0576:   fclex
0577:   fldcw cwr
0578: frndint
0579:    fld  st(1)
0580:   fsubp st(1),st
0581:    fst DeltaY
```

Der linke Anfangspunkt wird durch den Sub-Pixel-Wert korrigiert.

```
//   xLeft:=v1.x+DeltaY*stepXleft;

0582:    fld stepXleft
0583:  fmulp st(1),st
0584:    fld v1.x
0585:  faddp st(1),st
0586:    fst xLeft
```

Der rechte Anfangspunkt wird durch den Sub-Pixel-Wert korrigiert.

```
//    xRight:=v1.x+DeltaY*stepXright;

0587:    fld DeltaY
0588:    fld stepXright
0589:  fmulp st(1),st
0590:    fld v1.x
0591:  faddp st(1),st
0592:    fst xRight
```

Der rechte Z-Schrittwert wird interpoliert.

```
//   stepZright:=(V2.z-V1.z)/(V2.y-V1.y);

0593:    fld V2.y
0594:    fld V1.y
0595:  fsubp st(1),st
0596:    fst st(1)
0597:    fld V2.z
0598:    fld v1.z
0599:  fsubp st(1),st
0600:    fld st(1)
0601:  fdivp st(1),st
0602:    fst stepZright
```

Der linke Z-Schrittwert wird interpoliert.

```
//   stepZleft:=(Vextra.z-V1.z)/(Vextra.y-V1.y);
```

```
0603:   finit
0604:     fld Vextra.y
0605:     fld V1.y
0606:   fsubp st(1),st
0607:     fst st(1)
0608:     fld Vextra.z
0609:     fld V1.z
0610:   fsubp st(1),st
0611:     fld st(1)
0612:   fdivp st(1),st
0613:     fst stepZleft
```

Die linke Z-Anfangskoordinate wird durch den Sub-Pixel-Wert korrigiert.

```
        //    zLeft:=v1.z+stepZleft*DeltaY;
```

```
0614:    fld DeltaY
0615:    fst st(1)
0616:    fld stepZleft
0617:  fmulp st(1),st
0618:    fld v1.z
0619:  faddp st(1),st
0620:    fst zLeft
```

Die rechte Z-Anfangskoordinate wird durch den Sub-Pixel-Wert korrigiert.

```
        //    zRight:=v1.z+stepZright*DeltaY;
```

```
0621:  finit
0622:    fld DeltaY
0623:    fld stepZright
0624:  fmulp st(1),st
0625:    fld v1.z
0626:  faddp st(1),st
0627:    fst zRight
```

Die Y-Schleife.

```
        // for y:=ceil(v1.y) to ceil(v2.y)-1 do begin
```

Die Werte in V1.y und V2.y werden aufgerundet.

```
0628:  finit
0629:    fld v1.y
0630:  fclex
0631:  fldcw cwr
0632:   fist y

0633:    fld v2.y
0634:   fist Ycounter
0635:    dec Ycounter

0636:  mov edx,y
0637:  cmp edx,Ycounter
0638:  ja @Yequ2

0639: @Yloop2:
```

Der Wert Xleft wird aufgerundet und in die Variable XL gespeichert.

```
        // XL:=ceil(Xleft);

0640:  finit
0641:   fld Xleft
0642:  fclex
0643:  fldcw cwr
0644:   fist XL
```

Der Wert Xright wird aufgerundet und in die Variable XR gespeichert.

```
        // XR:=ceil(Xright);

0645:   fld Xright
0646:  fist XR
0647:    wait
```

Hier wird verglichen, ob XL größer als XR ist. Falls dem so ist, werden die beide Werte vertauscht.

```
        //   if (XL>XR) then swapL(XL,XR);
```

```
0648:  mov eax,XL
0649:  mov ebx,XR
0650:  cmp eax,ebx
0651:  jle @skipit3

0652:  mov XL,ebx
0653:  mov XR,eax

0654: @skipit3:
```

Der Wert, um den Z in der X-Schleife erhöht wird, wird interpoliert.

```
//   zAdd:=(zright-zleft)/(xright-xleft+1);

0655:    fld xright
0656:    fld xleft
0657:  fsubp st(1),st
0658:   fld1
0659:  faddp st(1),st
0660:    fst st(2)
0661:    fld zright
0662:    fld zleft
0663:  fsubp st(1),st
0664:    fld st(1)
0665:  fdivp st(1),st
0666:    fst zAdd
```

Die Variable, in der der Wert der Koordinate Z gespeichert ist, wird initialisiert.

```
        //      z:=zleft;

0667:   mov eax,zleft
0668:   mov z,eax
```

Die X-Schleife:

```
        // for x:=XL to XR-1 do begin

0669:  mov ebx,XL
0670:  cmp ebx,XR
```

```
0671:  je @XLRLequ2
0672: @Xloop2:
```

Pro Pixel wird auch der Z-Wert neu errechnet.

```
        //    z:=z+zAdd;
```

```
0673:  finit
0674:    fld z
0675:    fld zAdd
0676:  faddp st(1),st
0677:    fst st(1)
0678:   fstp z
0679:   wait
```

Der Z-Buffer Test wird durchgeführt. Falls der aktuelle Z-Wert kleiner ist als der Wert im Z-Buffer, wird das folgende Codesegment übersprungen.

```
        //    if z > zBuffer[x,y] then begin
```

```
0680:  mov eax,y
0681:  mul where.width
0682:  add eax,ebx
0683:  shl eax,2

0684:  mov esi,eax
0685:  mov edi,eax

0686:  add esi,zbuffer


0687:  fld dword ptr [esi]
0688:  fld st(1)
0689:  fcom st(1)

0690: fstsw ax
0691:  sahf
0692:   jc @Ztest2
```

Gezeichnet wird der aktuelle Pixel.

// PutPixel32(x, y, color, where);

0693: add edi,where.data
0694: mov eax,color
0695: mov [edi],eax

In den Z-Buffer wird der aktuelle Z-Wert eingetragen.

// zBuffer[x,y]:=z;

0696: mov eax, z
0697: mov [esi],eax
// end;

0698: @Ztest2:

Ende der X-Schleife:

0699: inc ebx
0700: mov edx,XR
0701: cmp ebx,edx
0702: jb @Xloop2
0703: @XLRLequ2:

Die linke X-Startkoordinate wird um einen Schritt erhöht.

0704: finit
// Xleft:=Xleft+stepXleft;

0705: fld Xleft
0706: fld stepXleft
0707: faddp st(1),st
0708: fst Xleft

Die rechte X-Startkoordinate wird um einen Schritt erhöht.

// Xright:=Xright+stepXright;

0709: fld Xright

0710: fld stepXright
0711: faddp st(1),st
0712: fst Xright

Die linke Z-Startkoordinate wird um einen Schritt erhöht.

// Zleft:=Zleft+stepZleft;

0713: fld Zleft
0714: fld stepZleft
0715: faddp st(1),st
0716: fst Zleft

Die rechte Z-Startkoordinate wird um einen Schritt erhöht.

// Zright:=Zright+stepZright;

0717: fld Zright
0718: fld stepZright
0719: faddp st(1),st
0720: fst Zright

Ende der Y-Schleife:

0721: inc y
0722: mov eax,y
0723: cmp eax,Ycounter
0724: jbe @Yloop2

0725: @Yequ2:

Falls die Länge des zweiten Teils des Dreieckes Null ist, wird das folgende Codesegment übersprungen.

// if (v3.y-v2.y)=0 then exit;

0726: fld v3.y
0727: fld v2.y
0728: fsubp st(1),st

```
0729:   fldz
0730:   fcom st(1)
0731:   fstsw ax
0732:   sahf
0733:   jz @MainExit
```

Der rechte X-Schrittwert wird interpoliert.

```
0734: finit
        //   stepXright:=(V3.x-V2.x)/(V3.y-V2.y);

0735:   fld v3.y
0736:   fld v2.y
0737:  fsubp st(1),st
0738:   fst st(1)
0739:   fld V3.x
0740:   fld v2.x
0741:  fsubp st(1),st
0742:   fld st(1)
0743:  fdivp st(1),st
0744:   fst stepXright
```

Der linke X-Schrittwert wird interpoliert.

```
        //   stepXleft:=(V3.x-Vextra.x)/(V3.y-V2.y);

0745:   finit
0746:   fld v3.y
0747:   fld v2.y
0748:  fsubp st(1),st
0749:   fst st(1)
0760:   fld v3.x
0761:   fld Vextra.x
0762:  fsubp st(1),st
0763:   fld st(1)
0764:  fdivp st(1),st
0765:   fst stepXleft
```

Der Sub-Pixel-Wert wird bestimmt.

// DeltaY:= ceil(v2.y) – v2.y;

```
0766:  finit
0767:   fld v2.y
0768:  fclex
0769:  fldcw cwr
0770: frndint
0771:   fld v2.y
0772:  fsubp st(1),st
0773:   fst DeltaY
```

Der linke X-Anfangspunkt wird durch den Sub-Pixel-Wert korrigiert.

*// xLeft:=vextra.x+DeltaY*stepXleft;*

```
0774:   fld stepXleft
0775:  fmulp st(1),st
0776:   fld vextra.x
0777:  faddp st(1),st
0778:   fst xLeft
```

Der rechte X-Anfangspunkt wird durch den Sub-Pixel-Wert korrigiert.

*// xRight:=v2.x+DeltaY*stepXright;*

```
0779:   fld DeltaY
0780:   fld stepXright
0781:  fmulp st(1),st
0782:   fld v2.x
0783:  faddp st(1),st
0784:   fst xRight
```

Der rechte Z-Schrittwert wird interpoliert.

// stepZright:=(V3.z-V2.z)/(V3.y-V2.y);

```
0785:   fld V3.y
```

```
0786:    fld V2.y
0787:  fsubp st(1),st
0788:    fst st(1)
0789:    fld v3.z
0800:    fld v2.z
0801:   fsubp st(1),st
0802:    fld st(1)
0803:   fdivp st(1),st
0804:    fst stepZright
```

Der linke Z-Schrittwert wird interpoliert.

```
0805: finit
        //   stepZleft:=(V3.z-Vextra.z)/(V3.y-V2.y);

0806:    fld V3.y
0807:    fld V2.y
0808:   fsubp st(1),st
0809:    fst st(1)
0810:    fld v3.z
0811:    fld Vextra.z
0812:   fsubp st(1),st
0813:    fld st(1)
0814:   fdivp st(1),st
0815:    fst stepZleft
```

Der linke Z-Anfangspunkt wird durch den Sub-Pixel-Wert korrigiert.

```
        //   zLeft:=vextra.z+stepZleft*DeltaY;

0816:    fld DeltaY
0817:   fmulp st(1),st
0818:    fld vextra.z
0819:   faddp st(1),st
0820:    fst zLeft
```

Der rechte Z-Anfangspunkt wird durch den Sub-Pixel-Wert korrigiert.

```
        //   zRight:=v2.z+stepZright*DeltaY;
```

0821: fld DeltaY

0822: fld stepZright

0823: fmulp st(1),st

0824: fld v2.z

0825: faddp st(1),st

0826: fst zRight

Die Y-Schleife:

// for y:=ceil(v2.y) to ceil(v3.y)-1 do begin

0827: finit

0828: fld v2.y

0829: fclex

0830: fldcw cwr

0831: fist y

0832: fld v3.y

0833: fist Ycounter

0834: dec Ycounter

0835: mov edx,y

0836: cmp edx,Ycounter

0837: ja @Yequ1

0838: @Yloop1:

Der Wert in der Variable Xleft wird aufgerundet und in die Variable XL gespeichert.

// XL:=ceil(Xleft);

0839: finit

0840: fld Xleft

0841: fclex

0842: fldcw cwr

0843: fist XL

Der Wert in der Variable Xright wird aufgerundet und in die Variable XR gespeichert.

// XR:=ceil(Xright);

```
0844:  fld Xright
0845:  fist XR
0846:    wait
```

Verglichen wird, ob der Wert in XL größer ist als der Wert in XR, falls dem so ist, werden die beiden Werte vertauscht.

```
        //  if (XL>XR) then swapL(XL,XR);
```

```
0847:  mov eax,XL
0848:  mov ebx,XR
0849:  cmp eax,ebx
0850:  jle @skipit4
0851:  mov XL,ebx
0852:  mov XR,eax
0853: @skipit4:
```

Der Wert, um den Z in der X-Schleife erhöht wird, wird interpoliert.

```
        //  zAdd:=(zright-zleft)/(xright-xleft+1);
```

```
0854:   fld xright
0855:   fld xleft
0856:  fsubp st(1),st
0857:   fld1
0858:  faddp st(1),st
0859:   fst st(2)
0860:   fld zright
0861:   fld zleft
0862:  fsubp st(1),st
0863:   fld st(1)
0864:  fdivp st(1),st
0865:   fst zAdd
```

Der Anfangswert der Z Variable wird festgelegt.

```
        //    z:=zleft;
```

```
0866:   mov eax,zleft
```

```
0867:   mov z,eax
```

Die X-Schleife:

```
        // for x:=XL to XR-1 do begin
0868:  mov ebx,XL
0869:  cmp ebx,XR
0870:  je @XLRLequ1
0871: @Xloop1:
```

Erhöht wird der Z Wert um den Z-Schrittwert.

```
        //    z:=z+zAdd;

0872:   finit
0873:    fld z
0874:    fld zAdd
0875:   faddp st(1),st
0876:    fst st(1)
0877:   fstp z
0878:   wait
```

Es ist an der Zeit, zu prüfen, ob der aktuelle Z-Wert kleiner als der im Z-Buffer, falls dem so ist, wird das folgende Codesegment übersprungen.

```
        //    if z > zBuffer[x,y] then begin

0879:   mov eax,y
0880:   mul where.width
0881:   add eax,ebx
0882:   shl eax,2

0883:   mov esi,eax
0884:   mov edi,eax

0885:   add esi,zbuffer

0886:   fld dword ptr [esi]
0887:   fld st(1)
```

```
0888:  fcom st(1)

0889: fstsw ax
0890:  sahf
0891:   jc @Ztest1
```

Ansonsten wird der aktuelle Pixel gezeichnet

```
        //                      PutPixel32(x, y, color, where);

0892:   add edi,where.data
0893:   mov eax,color
0894:   mov [edi],eax
```

und der aktuelle Z-Wert in den Z-Buffer gespeichert.

```
        //                      putS(zbuffer+(y*xscale+x) shl 2,z);

0895:   mov eax, z
0896:   mov [esi],eax

        //    end;
0897: @Ztest1:
```

Ende der X-Schleife:

```
0898: inc ebx
0899: mov edx,XR
0900: cmp ebx,edx
0901: jb @Xloop1
0902: @XLRLequ1:
```

Die linke X-Startkoordinate wird um einen Schritt erhöht.

```
0903: finit
        //    Xleft:=Xleft+stepXleft;

0904:   fld Xleft
0905:   fld stepXleft
0906: faddp st(1),st
```

0907: fst Xleft

Die rechte X-Startkoordinate wird um einen Schritt erhöht.

// Xright:=Xright+stepXright;

0908: fld Xright
0909: fld stepXright
0910: faddp st(1),st
0911: fst Xright

Die linke Z-Startkoordinate wird um einen Schritt erhöht.

// Zleft:=Zleft+stepZleft;

0912: fld Zleft
0913: fld stepZleft
0914: faddp st(1),st
0915: fst Zleft

Die rechte Z-Startkoordinate wird um einen Schritt erhöht.

// Zright:=Zright+stepZright;

0916: fld Zright
0917: fld stepZright
0918: faddp st(1),st
0919: fst Zright

Ende der Y-Schleife:

0920: inc y
0921: mov eax,y
0922: cmp eax,Ycounter
0923: jbe @Yloop1

0924: @Yequ1:

Ende der Routine:

0925: @MainExit:

0926: pop edi
0927: pop esi

0928: end{$IFDEF PAS_FPC}['EAX','EBX','ECX','EDX','ESI','EDI']{$ENDIF};
0929: end;

4. Ein einfacher Setup eines Tests

4.1 Ein Auszug aus einer Software 3D Engine

In diesem Kapitel wird vorausgesetzt, dass eine 2D-Engine folgende Dinge unterstützt:
eine Surface-Beschreibung mit einem Zeiger auf den Daten-Buffer, sowie die Maße der Surface, hier in Konstanten width und height. Es muss eine Routine für das Zeichen eines Pixels an angegebene Koordinaten geben. Hier PutPixel() . Und selbstverständlich eine Routine, die den Inhalt der Surface auf den Bildschirm bringt. Hier FlipScreen();

Vorausgesetzt wird auch, dass die 3D-Engine Folgendes unterstützt: eine ClearZBuffer() Routine, die den Z-Buffer löscht. MoveObject() sowie RotateObject() sorgen für die Translation des Objekts. Die Prozedur MakeProjection() sorgt dafür, dass aus den dreidimensionalen Koordinaten zweidimensionale werden, diese werden für das Zeichen der Dreiecks benötigt. Außerdem muss die Engine über einen Weg verfügen, geometrische Daten aus einer externen Datei zu laden. Selbstverständlich muss auch eine Struktur für das Beschreiben des Objektes selbst gegeben sein sowie seiner Poligonen.

Der nächste Code wird einmal pro Frame aufgerufen:

Das Löschen des Bildschirmes mit der Farbe Schwarz ist unumgänglich.

```
fastfill(vscreen.data,vscreen.wirdth * vscreen.height,0);
```

Falls nicht der Clear-Rederuction-Algorithm verwendet wird, wird der Inhalt des Z-Buffer durch den Aufruf von

```
ClearZBuffer;
```

geleert. Falls der Clear-Reduction-Algorithm angewandt wird, wird der Wert der Inkremente clear_translation um 1 erhöht. Die Variable zAddValue wird so gebildet, dass die Konstante z_max, die den maximalen Wert des Z-Buffer repräsentiert, mit dem Wert in der Variable clear_translation multipliziert wird. Somit ist das Löschen des Z-Buffers nicht nötig.

```
inc(clear_translation);
zAddValue:=clear_translation*z_max;
```

Im folgenden Codesegment wird das später zu zeichnende Objekt auf die Mitte des Bildschirmes verschoben, dann gedreht und wieder zurück auf seine Ausgangsposition zurück verschoben.

```
moveObject(myObject,0,0,-moveaway);
rotateObject(myObject,0.01,0.01,0.01);
moveObject(myObject,0,0,+moveaway);
```

Nun die Schleife, in der jedes Polygon des Objekts gezeichnet wird.

```
for i:=0 to myObject.numPolys-1 do begin
```

Extrahiert wird ein Polygon aus dem Objekt.

```
myPoly:=myObject.Poly[i];
```

Seine 3D Koordinate werden in Bildschirm-Koordinate umgewandelt.

```
MakeProjection(myPoly.A,v1.X,v1.Y,v1.Z);
MakeProjection(myPoly.B,v2.X,v2.Y,v2.Z);
MakeProjection(myPoly.C,v3.X,v3.Y,v3.Z);
```

Und schließlich wird das Dreieck gezeichnet.

```
TriangleFLATZ(v1.x,v1.y,v1.z,v2.x,v2.y,v2.z,v3.x,v3.y,v3.z,RGB32(v1.r,v1.g,v1.b),vscreen);
```

Ende der Schleife.

```
end;
```

Die Surface mit dem gerenderten Bild wird auf den Bildschirm gebracht.

```
FlipScreen(vscreen);
```

4.2 Direkter Vergleich zweier Methoden

Verglichen werden die Frame per Second Angaben unter verschiedenen Kompilatoren und unter verschiedenen Betriebssystemen. Diese Angaben sind relativ und werden auf jedem PC-System unterschiedlich ausfallen.

TMT_DOS steht für den TMT Pascal Compiler v.3.90 http://www.frameworkpascal.com/

FPC_DOS steht für den FreePascal Compiler für DOS v.2.6.0 http://www.freepascal.org/

FPC_WIN steht für den FreePascal Compiler für Windows v.2.6.0 http://www.freepascal.org/

Delphi_WIN steht für den Delphi 7 Compiler für Windows.

http://www.embarcadero.com/de/products/delphi/previous-versions

	TMT_DOS	FPC_DOS	FPC_WIN	Delphi_WIN
Half-Space Basisalgorithmus	5,29	7,23	247,9	318,82
Half-Space Optimiert	6,63	9,02	536,63	561,12
Half-Space Fixedpoint	7,45	41,34	717,34	722,12
Half-Space Fixedpoint ASM	43,6	80,94	803,77	845,89
Scan-Line Basisalgorithmus	8,27	23,31	629,54	694,76
Scan-Line FPU ASM	35,21	40,1	159,35	160,59

Diese Werte berücksichtigen nicht, dass die vertikale Synchronisation des Monitors die Frame Rate verfälschen kann.

5. Resümee

Die Half-Space-Methode hat insgesamt im praktischen Test besser abgeschnitten als die Scan-Line-Methode. Aber kein Wunder, die Scan-Line-Methode arbeitet mit Fließkomma-Zahlen, und da das Laden der Werte auf dem Speicher in die Register der FPU sehr viel mehr Zeit in Anspruch nimmt als das Laden der Werte aus dem Speicher in die CPU Register. Wunderlich ist jedoch, dass die von Hand erstellte ASM Version der Scan-Line-Routine unter Windows mit FPC oder Delphi langsamer ist als die vom Kompilator Assemblierte Version. Unter DOS ist es, wie erwartet andersrum.

6. Literaturverzeichnis

[1] Apetri, Marius: 3D-Grafik Programmierung. 2. Auflage. mitp-Verlag: Heidelberg, 2008

[2] Backer, Reiner: Assembler Maschinennahes Programmieren von Anfang an. Mit Windows- Programmierung. Überarbeitete und erweiterte Neuausgabe. Rowohlt Taschenbuch Verlag, 2003

[3] Capens, Nicolas: Advanced Rasterization. http://devmaster.net/forums/topic/1145-advanced-rasterization/ 2004, abgerufen Februar 2012

[4] Gordon, Robert: A calculated look at fixedpoint arithmetic http://www.eetindia.co.in/ARTICLES/1998APR/PDF/EEIOL_1998APR03_EMS_TA.pdf, abgerufen August 2012

[5] Jacobs, Jon Q.: Delphi Developer's Guide to OpenGL. Wordware Publishing, Inc. Plano, Texas 1999.

[6] Rohde, Joachim: Assembler GE-PACKT. 2. Auflage. mtp-Verlag: Heidelberg, 2007

[7] Rohde, Joachim aund Roming, Marcus: Assembler Grundlagen der Programmierung. 2. Auflage. mitp-Verlag: Heidelberg, 2006

[8] Swan, Tom: Mastering Turbo Assembler, 2. Auflage. Dialektika-Verlag: Kiew, 1996

[9] LaMothe, Andre: Tricks of the 3D Game programming Gurus advanced 3D graphics and rasterization. Sams Publishing: Moskau, 2004

[10] Zerbst, Stefan: 3D Spieleprogrammierung mit Direct X in C/C++. Libri Books on Demand, Braunschweig 2000.

[11] Nettle, Paul: Sub-Pixel Accuracy. http://www.cpp-home.com/tutorials/224_1.htm, abgerufen August 2012

7. Abbildungsverzeichnis

8. Anhang

Im Anhang befinden sich im Verzeichnis \Anhang\Code\ die Quellcodes der jeweiligen in dieser Abhandlung vorgestellten Routinen.

Im Verzeichnis \Anhang\TEST befinden sich die kompilierten Programme die im Kapitel 4.2 zum testen der FPS-Rate verwendet worden waren.