VERTEXWAHN - Der Bresenham Algorithmus
Nach einer wahren Begebenheit

Motivation

Dieser Artikel soll zeigen wie man Geradenstücke/Strecken effizient mithilfe des Bresenham Algorithmus rasterisiert (Scan Convention).

Das Geschichtliche

Jack Bresenham ist ein US-amerikanischer Informatiker. Derzeit ist Bresenham Professor an der Universität Dayton in Ohio.

Bresenham schrieb im November 2001:
"I was working in the computation lab at IBM's San Jose development lab. A Calcomp plotter had been attached to an IBM 1401 via the 1407 typewriter console. [The algorithm] was in production use by summer 1962, possibly a month or so earlier. Programs in those days were freely exchanged among corporations so Calcomp (Jim Newland and Calvin Hefte) had copies. When I returned to Stanford in Fall 1962, I put a copy in the Stanford comp center library.

A description of the line drawing routine was accepted for presentation at the 1963 ACM [Association for Computing Machinery] national convention in Denver, Colorado. It was a year in which no proceedings were published, only the agenda of speakers and topics in an issue of Communications of the ACM. A person from the IBM Systems Journal asked me after I made my presentation if they could publish the paper. I happily agreed, and they printed it in 1965."

Der Artikel „Algorithm for computer control of a digital plotter“ von J. E. Bresenham erschien in der IBM Systems Journal Vol.4 No. 1 1965 und kann von diversen Quellen kostenfrei runtergeladen werden.

Die Erkenntnisse von Bresenham lassen sich auf die Rasterisierung von Strecken übertragen.

Der Algorithmus von Bresenham zeichnet sich aus durch effiziente Rasterkonvertierung gerader Linien, da er nur ganzzahlige Addition und Subtraktion sowie die Multiplikation mit 2 beinhaltet. Diese Operationen können durch den Computer sehr schnell ausgeführt werden. Die Multiplikation mit 2 kann beispielsweise mit Hilfe einer Shiftoperation durchgeführt werden, so dass der Algorithmus ganz ohne Verwendung der Multiplikation auskommt.

Seine Algorithmen zählen zu den gängigen Standards und finden zum Beispiel in den meisten Zeichenprogrammen Verwendung.

Hier die Orginal IBM 1401 Implementierung (Autocoder) von 1962:

```
     000        JOB  ENTER PLOT POINTS FROM 1407
                CTL  4411
                ORG  501
     *
     *                           TRAVEL SUBROUTINE
     *
     TRAVEL     SBR  LEAVE&3
                ZA   ZEROA,ALPHA
```

```
            ZA   XTRGT,DELX
            ZA                  YTRGT,DELY
            S    XPR,DELX
            S                   YPR,DELY
            MCW  XTRGT,XPR
            MCW                 YTRGT,YPR
            ZA   DELX,DELA
            ZA                  DELY,DELB
            MZ   *&8,DELA                NEED MAGNITUDE SO  A B BITS
            MZ   *&1,DELB
*                              DETERMINE OCTANT
            C    DELA,DELB
            BM   DELXN,DELX
            BM   DELYN,DELY
            MN   PXPY,D45
            BH   D00PY
D00PX       MN   PX,D00
SETDEL      ZS   DELA,DEL
            A    DELA           DELA NOW DOUBLED-SEE DRIVING LOOP
            A                   DELB NOW DOUBLED-SEE DRIVING LOOP
            B    TALPHA
DELXN       BM   DELXYN,DELY
            MN   NXPY,D45
            BH   D00PY
D00NX       MN   NX,D00
            B    SETDEL
DELYN       MN   PXNY,D45
            BL   D00PX
D00NY       MN   NY,D00
            B    DELAY
D00PY       MN   PY,D00
DELAY       MN   DELY,DELA               DELA S/B DELY  SAVE PLUS
            MCW
            MN   DELX,DELB               DELB S/B DELX
            MCW
            B    SETDEL
DELXYN      MN   NXNY,D45
            BL   D00NX
            B    D00NY
*                              PLOTTER DRIVING LOOP
D00GO       MCW  %T0,D00,S               MOVE RELATIVE ZERO DEGREES
TALPHA      C    ALPHA,DELA
LEAVE       BE   0                       0 DUMMY
            A    TWOA,ALPHA        NOTE   DELA HAS BEEN DOUBLED
            A    DELB,DEL          NOTE   DELB PREVIOUSLY DOUBLED
            BM   D00GO,DEL
            MCW  %T0,D45,S
            S    DELA,DEL          NOTE   DELA PREVIOUSLY DOUBLED
            B    TALPHA
*                              CONSTANTS
ALPHA       DCW  #6
YTRGT       DCW  #6                   DESIRED Y COORDINATE
XTRGT       DCW  #6                   DESIRED X COORDINATE
DELY        DCW  #6                   Y2 MINUS Y1
DELX        DCW  #6                   X2 MINUS X1
YPR         DCW  #6                   PRESENT Y COORDINATE
XPR         DCW  #6                   PRESENT X COORDINATE
DELB        DCW  #6                   DEPENDENT VARIABLE DIFF
DELA        DCW  #6                   INDEPENDENT VARIABLE DIFF
DEL         DCW  #6                   DECISION DIFFERENCE
D00         DA   1X1,G                RELATIVE 0 DEGREE MOVE
D45         DA   1X1,G                RELATIVE 45 DEGREE MOVE
```

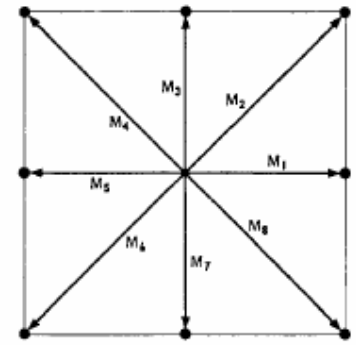
```
PX        DCW  @3@                    0  DEGREE CODE
PXPY      DCW  @2@                   45  DEGREE CODE
PY        DCW  @1@                   90  DEGREE CODE
NXPY      DCW  @8@                  135  DEGREE CODE
NX        DCW  @7@                  180  DEGREE CODE
NXNY      DCW  @6@                  225  DEGREE CODE
NY        DCW  @5@                  270  DEGREE CODE
PXNY      DCW  @4@                  315  DEGREE CODE
ZEROA     DCW  &0
TWOA      DCW  &2
*
*                          END OF TRAVEL SUBROUTINE
*
*                                 --CALLING SEQUENCE--
*                               1.   ZA   DESIRED X COORDINATE, XTRGT
*                               2.   ZA   DESIRED Y COORDINATE, YTRGT
*                               3.   B    TRAVEL
*
*                          XPR,YPR ARE UPDATED
*                          TRAVEL EXECUTED
*                          CONTROL RETURNED TO INSTR AFTER *B TRAVEL*
BEGINX    LCA  MXGX,47
          SW   27,MSGX-1
          MCE  XPR,33
          MCW  &XTRGT,PLACE&6
TYPE      LCA  @}@,48
          MCW  %T0,21,W
          CS   8
          LCA  @}@,8
          BIN  VIN,Q
          B    *-8
VIN       MCW  %T0,1,R
          SBR  PLACE&3
          BIN  TYPE,*
          BCE  PLACE&7,1,}
          MA   @I9H@,PLACE&3
          BM   VMINUS,1
          MCW  @?@,PLACE
          BWZ  VPLUS,1,B
          SW   1
PLACE     ZA   0,0
          BW   BEGINY,MSGX-1
          B    TRAVEL
          B    BEGINX
VMINUS    MCW  @!@,PLACE
VPLUS     SW   2
          B    PLACE
BEGINY    LCA  MSGY,47
          SW   27
          CW   MXGX-1
          MCE  YPR,33
          MCW  &YTRGT,PLACE&6
          B    TYPE
MSGX      DCW  @X NOW     0 -, ENTER NEXT X@
MSGY      DCW  @Y NOW     0 -, ENTER NEXT Y@
          END  BEGINX
```

Bresenham entwickelte also ein Programm für einen Computer der auf einen Plotter Graphen zeichnen sollte.

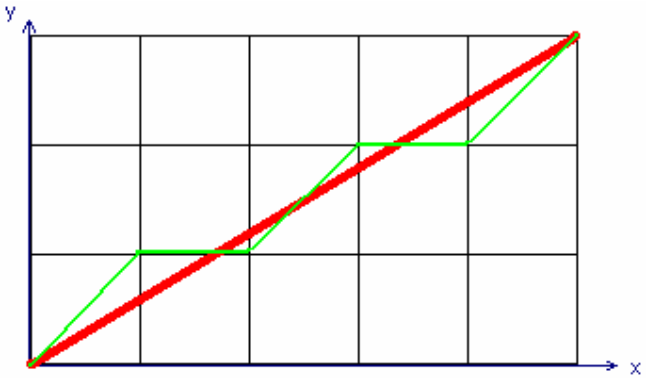
Zu diesem Zweck sollten bestimmte Punkte, beispielsweise einer Kurve, berechnet werden und diese dann mithilfe von Geradenstücken verbunden werden, wie folgendes Bild zeigt:

Das Vorwissen

Wie bereits erwähnt ist der Algorithmus für einen Plotter gedacht. Dabei gibt es gewisse Einschränkungen. Der Plotter kann sich nur in acht verschiedene Richtungen Bewegen. Gleichzeitig kann nur immer eine Bewegung durchgeführt werden:



Der Algorithmus von Bresenham versucht so präzise wie möglich mithilfe der Bewegungen M1 und M2 sich dem Geradenstück anzunähern.



Der Spannende Teil

Eine Strecke wird durch den Anfangs- und den Endpunkt definiert. Der Anfangspunkt der Strecke ist zugleich der Ursprung des Koordinatensystems. Zunächst sollen strecken betrachtet werden für deren Steigung m gilt: $0 < m < 1$.

Um eine solche Strecke zu rastern gibt Bresenham folgenden Algorithmus an:

$$\nabla_1 = 2\Delta b - \Delta a$$

$$\nabla_{i+1} = \begin{cases} \nabla_i + 2\Delta b - 2\Delta a & falls\ \nabla_i \geq 0 \\ \nabla_i + 2\Delta b & falls\ \nabla_i < 0 \end{cases}$$

Anfangspunkt: $D_1(x_1|y_1)$

Endpunkt: $D_2(x_2|y_2)$

$$x_2 > x_1$$
$$\Delta a = x_2 - x_1$$
$$\Delta b = y_2 - y_1$$

$$falls \begin{cases} \nabla_i < 0 & \rightarrow M_1 \\ \nabla_i \geq 0 & \rightarrow M_2 \end{cases}, \qquad i = 1,...,\Delta a$$

Anmerkungen:

Falls $\nabla_i < 0$ soll der Plotter die Bewegung M1 ausführen. Ansonsten die Bewegunsrichtung M2.

Beispiel:

Es soll eine Strecke von $D_1(0|0)$ nach $D_2(5|3)$ gezeichnet werden.

$$\Delta a = x_2 - x_1 = 5 - 0 = 5$$
$$\Delta b = y_2 - y = 3 - 0 = 3$$

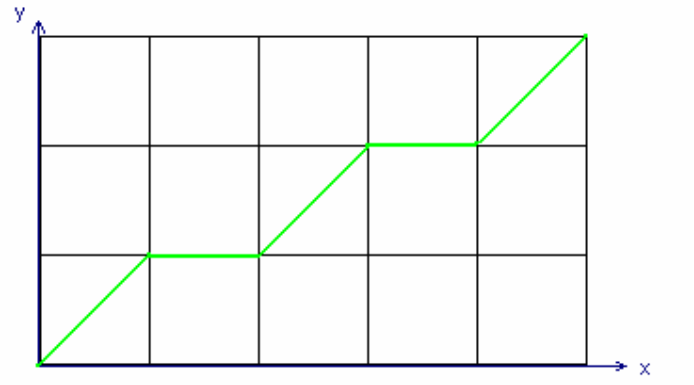
Die Steigung liegt zwischen 0 und 1

$$x_2 > x_1$$

$$\nabla_1 = 2 * 3 - 5 = 1$$
$$\nabla_2 = 1 + 2 * 3 - 2 * 5 = -3$$
$$\nabla_3 = -3 + 2 * 3 = 3$$
$$\nabla_4 = 3 + 2 * 3 - 2 * 5 = -1$$
$$\nabla_5 = -1 + 2 * 3 = 5$$

Folgende Bewegungen sind hintereinander durchzuführen: M2, M1, M2, M1, M2

Der Plotter würde folgendes Bild erstellen:

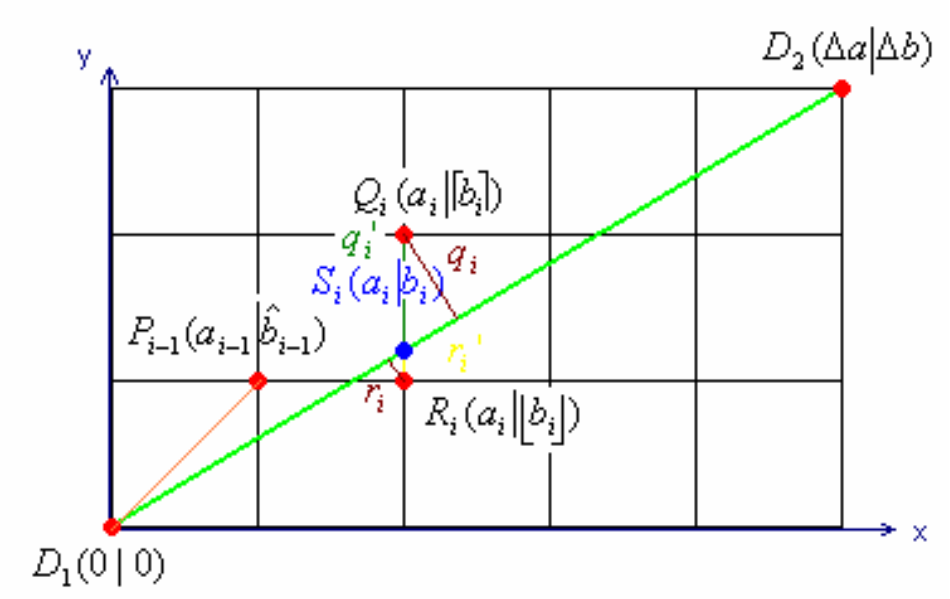


Etwas zur Mathematischen Notation

Mit $\lfloor x \rfloor$ ist der größte ganzzahlige Wert gemeint, der x nicht überschreitet
Mit $\lceil x \rceil$ ist der kleinste ganzzahlige Wert gemeint, der x überschreitet

Alles nur Mathematik

Testet man das Verfahren für andere Strecken mit der Einschränkung 0 < m < 1 funktioniert es genau so. Warum?



Angenommen der Algorithmus ist beim Punkt P angekommen. Nun muss er entscheiden, ob als nächster Referenzpunkt R oder Q zu wählen ist. Dies wird anhand des Abstandes des Punktes R bzw. des Punktes Q zur „idealen" Gerade ermittelt. Ist der Abstand von R zur Geraden geringer als der Abstand von Q zur Geraden, dann ist der nächste Referenzpunkt R – ansonsten Q.

Anders Ausgedrückt könnte man schreiben:

Falls: $r_i - q_i < 0$, dann bewege den Plotter von P nach R (Bewegungsrichtung M1)

Sonst: (also $r_i - q_i \geq 0$), dann bewege den Plotter von P nach Q (Bewegungsrichtung M2)

Wie man sieht spielt nur das Vorzeichen des Ausdrucks $r_i - q_i$ eine Rolle.

Aus der Theorie von ähnliche Dreiecken geht hervor das der Ausdruck $r'_i - q'_i$ das gleiche Vorzeichen wie der Ausdruck $r_i - q_i$ besitzt. Weiter ist bekannt, dass Δa für Geradenstücke, die unseren Einschränkungen unterliegen, immer positiv ist. Somit hat auch der Ausdruck $(r'_i - q'_i)\Delta a$ das gleiche Vorzeichen wie der Ausdruck $r_i - q_i$.

$$\nabla_i = (r'_i - q'_i)\Delta a = [(b_i - \lfloor b_i \rfloor) - (\lceil b_i \rceil - b_i)]\Delta a$$

Für die Koordinate b_i gilt: $b_i = \frac{\Delta b}{\Delta a} * a_i$ (geht aus der allgemeinen Geradengleichung y=mx+t hervor - t=0)

$$\nabla_i = (2b_i - \lfloor b_i \rfloor - \lceil b_i \rceil)\Delta a = (2\frac{\Delta b}{\Delta a}a_i - \lfloor b_i \rfloor - \lceil b_i \rceil)\Delta a = 2a_i\Delta b - \lfloor b_i \rfloor \Delta a - \lceil b_i \rceil \Delta a =$$
$$2a_i\Delta b - \lfloor b_i \rfloor \Delta a - \lceil b_i \rceil \Delta a = 2a_i\Delta b - \Delta a(\lfloor b_i \rfloor + \lceil b_i \rceil)$$

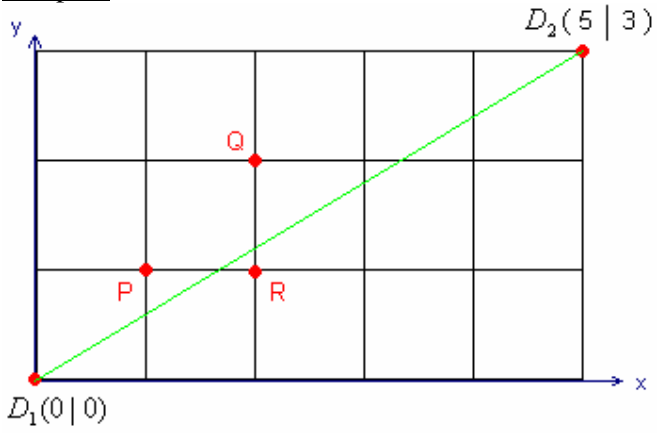
Für a_i gilt: $a_i = a_{i-1} + 1$
Für $\lceil b_i \rceil$ gilt: $\lceil b_i \rceil = \hat{b}_{i-1} + 1$
Für $\lfloor b_i \rfloor$ gilt: $\lfloor b_i \rfloor = \hat{b}_{i-1}$

Somit:

$$2(a_{i-1} + 1)\Delta b - \Delta a(\hat{b}_{i-1} + \hat{b}_{i-1} + 1) = (2a_{i-1} + 2)\Delta b - (2\hat{b}_{i-1}\Delta a + \Delta a) =$$
$$\nabla_i = 2a_{i-1}\Delta b + 2\Delta b - 2\hat{b}_{i-1}\Delta a - \Delta a$$

Mittelhilfe des Vorzeichens von ∇_i kann man nun entscheiden ob die Bewegung M1 bzw. die Bewegung M2 auszuführen ist.

<u>Beispiel:</u>



$P\left(a_1 \middle| \hat{b}_1\right) = P(1|1)$

$Q\left(a_2 \middle| \lceil b_2 \rceil\right) = Q(2|2)$

$R\left(a_2 \middle| \lfloor b_2 \rfloor\right) = R(2|1)$

$\nabla_2 = ?$

Anhand des Vorzeichens von ∇_2 kann man entscheiden, ob der nächste Referenzpunkt Q oder R ist.

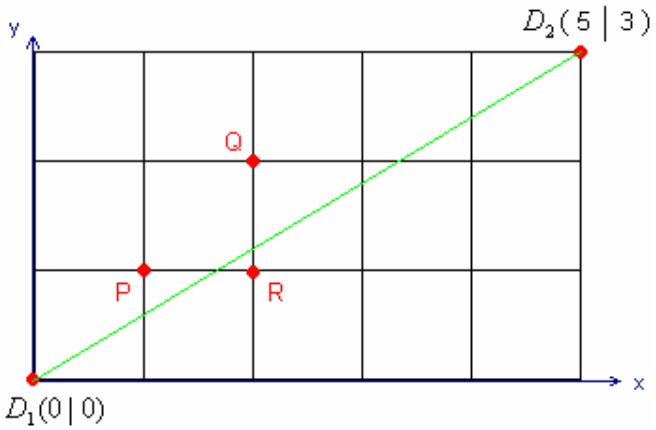
$$\nabla_2 = 2a_1\Delta b + 2\Delta b - 2\hat{b}_1\Delta a - \Delta a = 2*1*3 + 2*3 - 2*1*5 - 5 = -6$$

$\nabla_2 < 0$, d. h. der nächste Referenzpunkt ist R.

(Beispiel Ende)

Der Algorithmus erfüllt jetzt zwar seinen Zweck, jedoch kommen noch Multiplikationen mit Faktoren ungleich zwei vor.

Berechnen wir zunächst allgemein den Wert von ∇_1:



$P_0\left(a_0 \middle| \hat{b}_0\right) = (0|0)$

$\nabla_i = 2a_{i-1}\Delta b + 2\Delta b - 2\hat{b}_{i-1}\Delta a - \Delta a$

$\nabla_1 = 2a_0\Delta b + 2\Delta b - 2\hat{b}_0\Delta a - \Delta a = 2\Delta b - \Delta a$

Falls $\nabla_i \geq 0$:

$\hat{b}_i = \hat{b}_{i-1} + 1$

$\nabla_{i+1} = 2(a_{i-1} + 1)\Delta b - 2(\hat{b}_{i-1} + 1)\Delta a + 2\Delta b - \Delta a = 2a_{i-1}\Delta b + 2\Delta b - 2\hat{b}_{i-1}\Delta a - 2\Delta a + 2\Delta b - \Delta a$

$= \nabla_i - 2\Delta a + 2\Delta b$

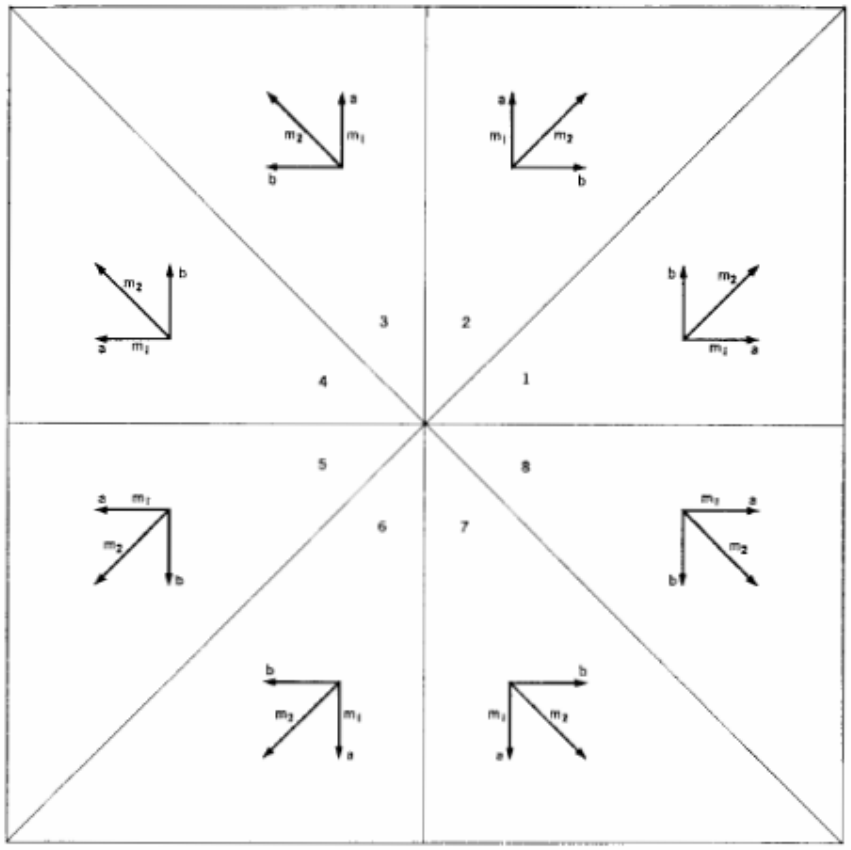
Falls $\nabla_i < 0$:

$\hat{b}_i = \hat{b}_{i-1}$

$\nabla_{i+1} = 2(a_{i-1} + 1)\Delta b + 2\Delta b - 2\hat{b}_i\Delta a - \Delta a = 2a_{i-1}\Delta b + 2\Delta b - 2\hat{b}_i\Delta a + 2\Delta b - \Delta a = \nabla_i + 2\Delta b$

Damit ist die „Formel“ von Bresenham beweisen.

Andere Oktanten
Natürlich funktioniert dieses Verfahren bisher nur für den ersten Oktanten. Aber auch hier hat sich Bresenham Gedanken gemacht.



Für andere Oktanten müssen einfach Δa und Δb nach einem anderen Schema berechnet werden und die Bewegungsrichtungen anders interpretiert werden. Wie kommt Bresenham auf dieses Ergebnis?

Zunächst soll der zweite Oktant untersucht werden:
Um eine Gerade mit der Steigung $1 < m < \infty$ zeichnen zu können werden nur Bewegungen vom Typ M2 und vom Typ M3 benötigt – dies leuchtet ein.

Für die Steigung $m = \frac{\Delta y}{\Delta x}$ gilt: $\Delta y > 0$ und $\Delta x > 0$

Für den Term $|\Delta x| - |\Delta y|$ gilt: $|\Delta x| - |\Delta y| < 0$

Weiter behauptet Bresenham das man den Algorithmus nur wie folgt abändern muss damit er wieder funktioniert:

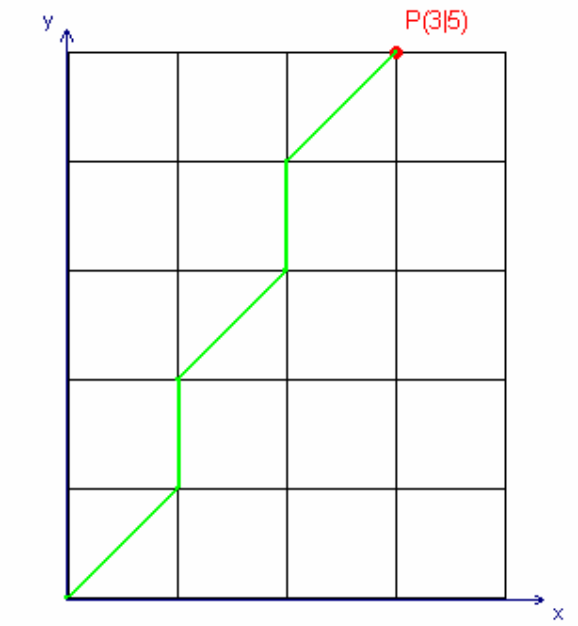
$\Delta a = \Delta y$
$\Delta b = \Delta x$

Zusätzlich ist die Bewegungsrichtung M1 durch die Bewegungsrichtung M3 zu ersetzen.

Beispiel:
Die Strecke von Q(0,0) zu P(3,5) ist zu zeichnen:

$$\Delta a = \Delta y = 5$$
$$\Delta b = \Delta x = 3$$
$$\nabla_1 = 2*3-5 = 1$$
$$\nabla_2 = \nabla_1 + 2\Delta b - 2\Delta a = 1+2*3-2*5 = -3$$
$$\nabla_3 = \nabla_2 + 2\Delta b = -3+2*3 = 3$$
$$\nabla_4 = \nabla_3 = +2\Delta b - 2\Delta a = 3+2*3-2*5 = -1$$
$$\nabla_5 = \nabla_4 + 2\Delta b = -1+2*3 = 5$$

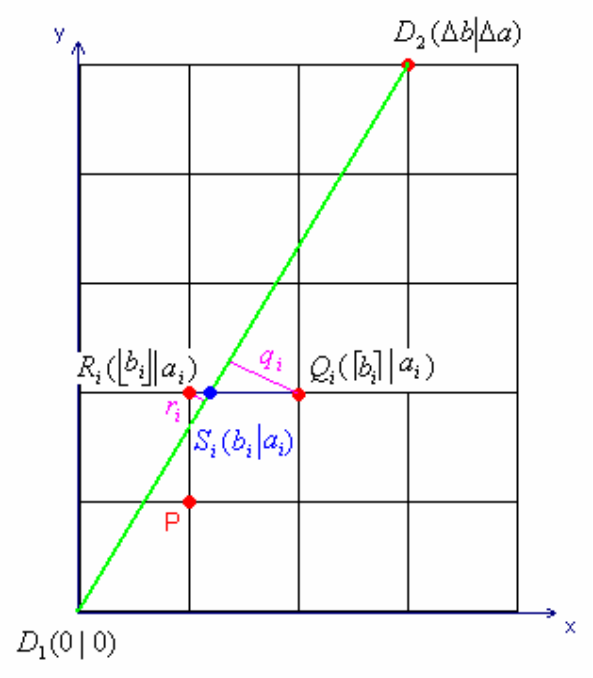
Nacheinander sind also folgende Bewegungsrichtungen auszuführen: M2, M3, M2, M3, M2



(Beispiel Ende)

Wie man sieht hat das Verfahren für diesen Fall geklappt. Wie ist Bresenham aber darauf gekommen? Genau weiß ich es auch nicht, aber ich vermute, dass er einfach für jeden Oktanten eine Skizze angefertigt hat und Gemeinsamkeiten untersucht hat und dabei auf diese Form gestoßen ist.

Genauso möchte ich hier auch vorgehen:



Δa ist immer positiv

$$\nabla_i = (r'_i - q'_i)\Delta a$$

$$falls\begin{Bmatrix}\nabla_i < 0 & \rightarrow M_3 \\ \nabla_i \geq 0 & \rightarrow M_2\end{Bmatrix}, \qquad i = 1,\ldots,\Delta a$$

$$a_i = \frac{\Delta a}{\Delta b} * b_i \Leftrightarrow b_i = \frac{\Delta b}{\Delta a} * a_i$$
$$\nabla_i = (2b_i - \lfloor b_i \rfloor - \lceil b_i \rceil)\Delta a = 2a_i\Delta b - \Delta a(\lfloor b_i \rfloor + \lceil b_i \rceil)$$

Hier haben wir wieder die gleiche Form wie für Geradenstücke mit der Steigung 0 < m < 1

Für a_i gilt: $a_i = a_{i-1} + 1$
Für $\lceil b_i \rceil$ gilt: $\lceil b_i \rceil = \hat{b}_{i-1} + 1$
Für $\lfloor b_i \rfloor$ gilt: $\lfloor b_i \rfloor = \breve{b}_{i-1}$

Auch hier ändert sich nichts. Somit ergibt sich folgende Abänderung vor den Algorithmus:

$\Delta a = \Delta y$
$\Delta b = \Delta x$

Zusätzlich ist die Bewegungsrichtung M1 durch die Bewegungsrichtung M3 zu ersetzen.

Für die anderen Oktanten sparre ich mir jetzt die Nachprüfung und gebe einfach die Tabelle von Bresenham an:

OCT	Δa	Δb	m_1	m_2
1	$\lvert \Delta x \rvert$	$\lvert \Delta y \rvert$	M_1	M_2
2	$\lvert \Delta y \rvert$	$\lvert \Delta x \rvert$	M_3	M_2
8	$\lvert \Delta x \rvert$	$\lvert \Delta y \rvert$	M_1	M_8
7	$\lvert \Delta y \rvert$	$\lvert \Delta x \rvert$	M_7	M_8
4	$\lvert \Delta x \rvert$	$\lvert \Delta y \rvert$	M_5	M_4
3	$\lvert \Delta y \rvert$	$\lvert \Delta x \rvert$	M_3	M_4
5	$\lvert \Delta x \rvert$	$\lvert \Delta y \rvert$	M_5	M_6
6	$\lvert \Delta y \rvert$	$\lvert \Delta x \rvert$	M_7	M_6

OCT steht für Oktant. In der Tabelle ist angegeben für welchen Oktanten welche Werte für Δa und Δb zu wählen sind. Außerdem ist angegeben welche Bewegungsrichtungen zu wählen sind.

$$falls\begin{cases}\nabla_i < 0 & \rightarrow m_1\\ \nabla_i \geq 0 & \rightarrow m_2\end{cases}, \qquad i = 1,...,\Delta a$$

„Einfache“ Bestimmung der Bewegungsrichtung

Es gibt insgesamt 8 verschiedene Bewegungsrichtungen für den Plotter.

Bresenham führt drei Boolsche Variablen ein: X, Y und Z. Diese stehen für die Vorzeichen von Δx, Δy bzw. $|\Delta x| - |\Delta y|$.

Eine 1 bedeutet ein positives Vorzeichen – eine 0 ein negatives Vorzeichen.

Δx	Δy	$\lvert \Delta x \rvert - \lvert \Delta y \rvert$	OCT	Δa	Δb	X	Y	Z	m_1	F	m_2	G
≥0	≥0	≥0	1	$\lvert \Delta x \rvert$	$\lvert \Delta y \rvert$	1	1	1	M_1	(1, 0, 0, 0)	M_2	(1, 0, 0, 0)
≥0	≥0	<0	2	$\lvert \Delta y \rvert$	$\lvert \Delta x \rvert$	1	1	0	M_3	(0, 1, 0, 0)	M_2	(1, 0, 0, 0)
≥0	<0	≥0	8	$\lvert \Delta x \rvert$	$\lvert \Delta y \rvert$	1	0	1	M_1	(1, 0, 0, 0)	M_8	(0, 0, 0, 1)
≥0	<0	<0	7	$\lvert \Delta y \rvert$	$\lvert \Delta x \rvert$	1	0	0	M_7	(0, 0, 0, 1)	M_8	(0, 0, 0, 1)
<0	≥0	≥0	4	$\lvert \Delta x \rvert$	$\lvert \Delta y \rvert$	0	1	1	M_5	(0, 0, 1, 0)	M_4	(0, 1, 0, 0)
<0	≥0	<0	3	$\lvert \Delta y \rvert$	$\lvert \Delta x \rvert$	0	1	0	M_3	(0, 1, 0, 0)	M_4	(0, 1, 0, 0)
<0	<0	≥0	5	$\lvert \Delta x \rvert$	$\lvert \Delta y \rvert$	0	0	1	M_5	(0, 0, 1, 0)	M_6	(0, 0, 1, 0)
<0	<0	<0	6	$\lvert \Delta y \rvert$	$\lvert \Delta x \rvert$	0	0	0	M_7	(0, 0, 0, 1)	M_6	(0, 0, 1, 0)

Des Weiteren gibt Bresenham folgende zwei Funktionen an:

$$F(X,Y,Z) = (XZ, Y\overline{Z}, \overline{X}Z, \overline{Y}\overline{Z})$$
$$G(X,Y) = (XY, \overline{X}Y, \overline{X}\overline{Y}, X\overline{Y})$$

Anmekrung: $\overline{A} = 1 - A$, ist die Negation. AB ist die Multiplikation, die man aber im Booleschen Sinne auch als Und-Verknüpfung auffassen kann.

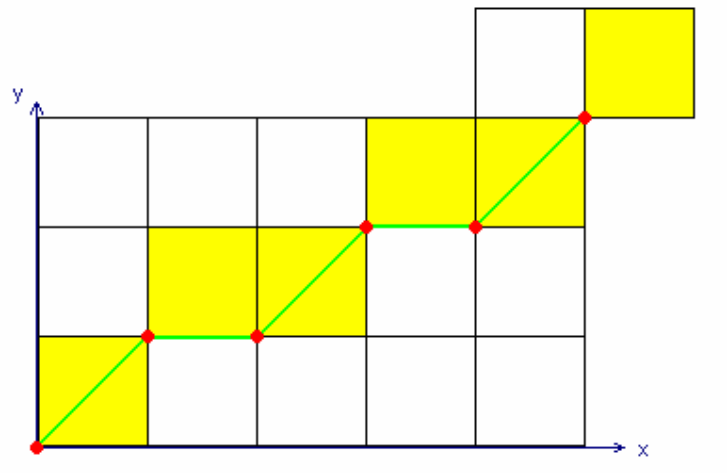
Bresenham schreibt, dass mithilfe dieser Formeln die Bewegungsrichtungen für den Plotter einfach bestimmt werden können. Der Nutzen von F und G wird nicht weiter ausgeführt.

Mir persönlich ist unklar was Bresenham mit diesem Formeln bewirken wollte, jedoch habe ich folgende Vermutungen:

Es ist auffallend, dass Bresenham nicht einfach ein Tabelle mit 8 Einträgen wählt anhand derer er mit den drei Booleschen Variablen (X,Y,Z = 3 Bit = 8 verschiedene Zustände) einfach die verschiednen Bewegungsrichtungen bestimmt. Stattdessen ordnet er einem dreier, bzw. zweier Tuppel einem vierer Tuppel zu, in dem nur eine einzige eins vorkommt. Vielleicht handelt es sich hier um eine spezifische Hardware-Implementation (Interne Codierung der Bewegungen für den Plotter). Wie gesagt das ist nur eine Spekulation.

Probleme mit Rastergrafiken

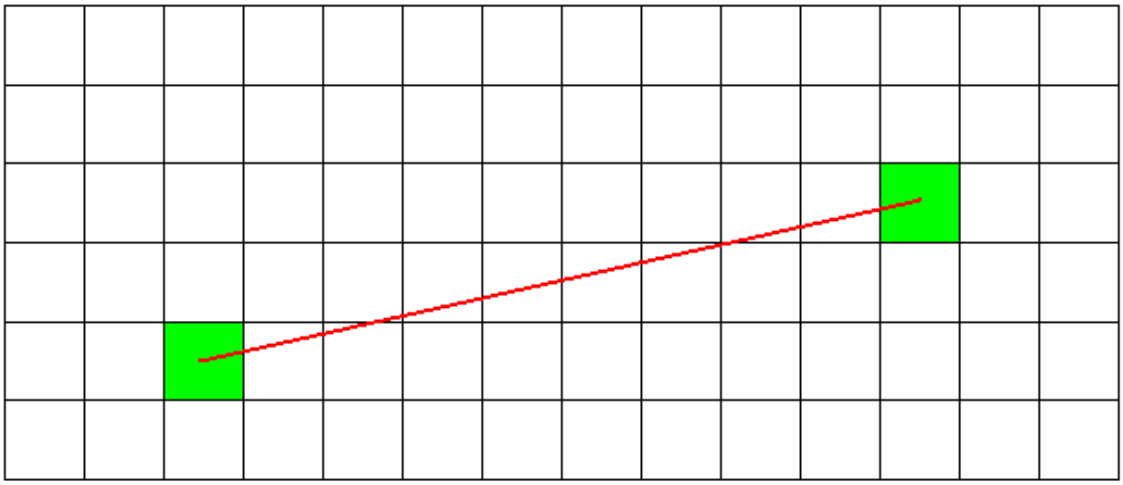
Die Grüne Line Zeigt den gedachten Bresenham Algorithmus – die gelben Pixel veranschaulichen das Ergebnis auf einer Rastergrafik.



Bresenham wie es Informatikstudenten in der Vorlesung Computergrafik lernen

In den meisten Büchern über Computergrafik wird man unter dem Thema Bresenham nichts über einen Plotter finden, sondern einen Algorithmus der mit ominösen Fehlerwerten arbeitet.

Es werden wieder zunächst Geradenstücke mit der Steigung 0<m<1 betrachtet:



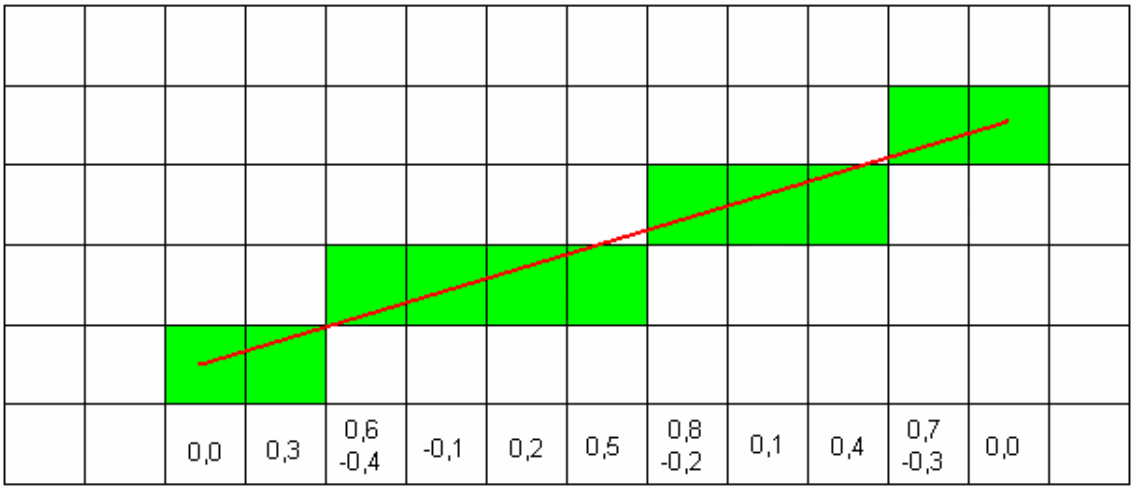
Steigung: $\frac{\Delta y}{\Delta x} = \frac{3}{10} = 0{,}3$

Die gezeichnete Linie weicht für gewöhnlich von der idealen Linie ab. Die Größe dieser Abweichung lässt sich ausnutzen, um zu entscheiden, ob man für die nächste x-Koordinate die aktuelle y-Koordinate beibehalten kann oder ob man die y-Koordinate um 1 erhöhen muss.

Hierbei führt man einen so genannten Fehlerwert ein, der die Abweichung vom idealen y Wert beschreibt:

$$error = y_{ideal} - y_{real}$$

Immer wenn der Fehler den Grenzwert 0,5 (ab 0,5 wird ja für gewöhnlich auf den nächst größeren Wert aufgerundet) überschreitet, wird der y Wert um 1 erhöht und der Fehler um 1 vermindert.



Pseudocode:

```
s = (double)(y2-y1)/ (double)(x2-x1);
error = 0.0;
x = x1;
```

```
y = y1;
while (x <= x2) {
      setPixel(x, y);
      x++;
      error = error + s;
      if (error > 0.5) {
            y++;
            error = error - 1.0;
      }
}
```

Man kann die verbleibende Gleitkommaarithmetik vermeiden, indem man die Steigung und den Fehler ganzzahlig macht.

0,5*2=1

$$\frac{y_2 - y_1}{x_2 - x_1} * x_2 - x_1 = y_2 - y_1$$

$$s_{neu} = s_{alt} \cdot 2dx = \frac{dy}{dx} \cdot 2dx = 2dy$$

Pseudocode:

```
delta = 2*(y2-y1);
error = 0.0;
x = x1;
y = y1;
while (x <= x2) {
      setPixel(x, y);
      x++;
      error = error + delta;
      if (error > x2-x1) {
            y++;
            error = error - 2*(x2-x1);
      }
}
```

Mit Trickreicher Codeersetzung wurde die Gleitkommarithmetik beseitigt.

Geraden in den anderen 7 Oktanten können durch Spiegelung und/oder Vertauschen von x und y auf den 1. Oktanten zurückgeführt werden.

Beispielsweise kann man jede Gerade im 2ten Oktanten durch eine Spiegelung einer Geraden im 1ten Oktanten an der Achse y = x erzeugen. Mann muss also nur alle x und y-Werte vertauschen.

Dies geht bei einem Plotter natürlich nicht – für einen Plotter ist eine Strecke vom Punkt (0,0) zum Punkt (10,10) etwas anderes wie eine Strecke vom Punkt (10,10) zum Punkt (0,0) (einmal wird der Bewegungstyp M2 und einmal der Bewegungstyp M6 benötigt). In der Computergrafik bietet es sich an hier vom Original Bresenham Algorithmus abzuweichen.

Bresenham hat mit rastern von Strecken nicht viel gemeinsam - die Ideen von Bresenham lassen sich aber sehr schön auf Rastergrafiken übertragen.

Die Implementierung

Für diese Implementierung verwende ich die Programmiersprache C++. Ich setzte ein kartesisches Rechsthändiges Koordinatensystem und eine SetPixel Funktion voraus.

```
void Bresenham(int d1x, int d1y, int d2x, int d2y)
{
	bool X, Y, Z;		// zur Bestimmung der Bewegunsrichtung
	int da, db;			// delta_a und delta_b
	int m1[2], m2[2];	// Bewegungsrichtung/Vektoren

	// delta a, delta b und Bewegungsrichtung bestimmen
	int dx = abs(d2x) - abs(d1x);
	int dy = abs(d2y) - abs(d1y);

	if(abs(dx)-abs(dy)>=0)
	{
		da = abs(dx);
		db = abs(dy);
		Z = true;
	}
	else
	{
		da = abs(dy);
		db = abs(dx);
		Z = false;
	}

	if(dx >= 0)
		X = true;
	else
		X = false;

	if(dy >= 0)
		Y = true;
	else
		Y = false;

	if(X == true && Y == true && Z == true)
	{
		// m1 = M1
		// m2 = M2
		m1[0] = 1;
		m1[1] = 0;
		m2[0] = 1;
		m2[1] = 1;
	}
	else
	{
		if(X == true && Y == true && Z == false)
		{
			// m1 = M3
			// m2 = M2
			m1[0] = 0;
			m1[1] = 1;
			m2[0] = 1;
			m2[1] = 1;
		}
		else
		{
			if(X == true && Y == false && Z == true)
			{
				// m1 = M1
				// m2 = M8
				m1[0] = 1;
				m1[1] = 0;
				m2[0] = 1;
				m2[1] = -1;
			}
			else
			{
				if(X == true && Y == false && Z == false)
				{
					// m1 = M7
					// m2 = M8
					m1[0] = 0;
					m1[1] = -1;
					m2[0] = 1;
					m2[1] = -1;
```

```
                                }
                                else
                                {
                                        if(X == false && Y == true && Z == true)
                                        {
                                                // m1 = M5
                                                // m2 = M4
                                                m1[0] = -1;
                                                m1[1] = 0;
                                                m2[0] = -1;
                                                m2[1] = 1;
                                        }
                                        else
                                        {
                                                if(X == false && Y == true && Z == false)
                                                {
                                                        // m1 = M3
                                                        // m2 = M4
                                                        m1[0] = 0;
                                                        m1[1] = 1;
                                                        m2[0] = -1;
                                                        m2[1] = 1;
                                                }
                                                else
                                                {
                                                        if(X == false && Y == false && Z == true)
                                                        {
                                                                // m1 = M5
                                                                // m2 = M6
                                                                m1[0] = -1;
                                                                m1[1] = 0;
                                                                m2[0] = -1;
                                                                m2[1] = -1;
                                                        }
                                                        else
                                                        {
                                                                // m1 = M7
                                                                // m2 = M6
                                                                m1[0] = 0;
                                                                m1[1] = -1;
                                                                m2[0] = -1;
                                                                m2[1] = -1;
                                                        }
                                                }
                                        }
                                }
                        }
                }
        }

        int gradient = 2 * db - da;
        int x = d1x;
        int y = d1y;

        SetPixel(x, y);

        for(int i = 0; i < da; i++)
        {
                if(gradient >= 0)
                {
                        x = x + m2[0];
                        y = y + m2[1];
                        gradient = gradient + 2 * db - 2 * da;
                }
                else
                {
                        x = x + m1[0];
                        y = y + m1[1];
                        gradient = gradient + 2 * db;
                }
                SetPixel(x, y);
        }
}
```

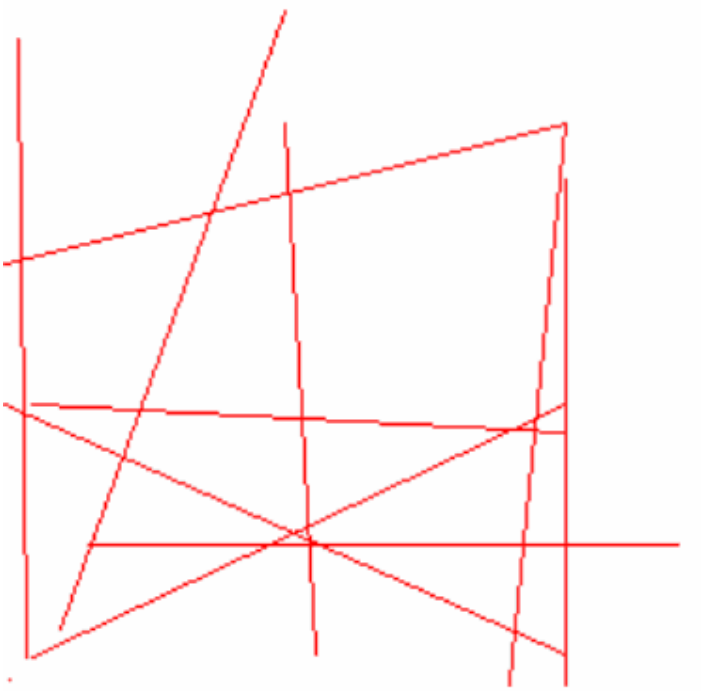
Ziel der Anstrengungen von Bresenham war es einen möglichst schnellen Algorithmus zu finden – es war nicht Ziel sauber zu programmieren ;). Obiger Code hat aber in Hinsicht auf

Schnelligkeit (und Sauberkeit) noch viele Schwächen. Die Schranke bei der Geschwindigkeitsoptimierung von Programmen wird durch das wissen des Programmieres über die Hardware festgelegt. Im obigen Quellcode wird z. B. mit der Zahl zwei multipliziert. Das ist totaler Unsinn, da der Computer mit dem Binären System arbeitet. Mit einem Bitshift geht es wesentlich schneller (die meisten Compiler optimieren dies jedoch automatisch). Weiteres Problem des Codes ist es, das zum Inkrementieren bzw. Dekrementieren addiert bzw. subtrahiert wird. Prozessoren haben für das Dekrementieren bzw. Inkrementieren einen eigenen Befehl (INC) der wesentlich schneller ist wie die Subtraktion bzw. Addition. In einer Echtzeitanwendung kann es dazu noch vorkommen, dass der Bresenham Algorithmus in jedem Frame (d. h. beispielsweise 70mal in der Sekunde) aufgerufen werden kann. C++ Programmierer wissen, das immer wenn eine Funktion aufgerufen wird alle Variablen der Funktion auf dem Stack abgelegt werden und am Ende wieder vom Stack gelöscht werden. Dies kostet wieder unnötig Zeit. Hier sollte man sich für globale Variablen entscheiden. Die hälfte der Fallunterscheidungen könnte man sich sparen, wenn man einfach den Endpunkt immer rechts vom Anfangspunkt setzt. Es gibt noch zahlreiche Optimierungen. Das Optimieren des Codes überlasse ich dem Leser. Mein Ziel war es nur dem Leser die Idee hinter dem Bresenham Algorithmus zu vermitteln.

Zum testen aller Oktanten kann folgender Code verwendet werden:

```
// 1. Oktant
Bresenham(10, 10, 200, 100);

// 8. Oktant
Bresenham(10, 100, 200, 90);

// 2. Oktant
Bresenham(20, 20, 100, 240);

// 7. Oktant
Bresenham(5, 230, 8, 10);

// Punkt
Bresenham(2, 2, 2, 2);

// Waagrechte\Horizontal
Bresenham(30, 50, 240, 50);

// Vertikale
Bresenham(200, 180, 200, 0);

// 3. Oktant
Bresenham(111, 11, 100, 200);

// 4. Oktant
Bresenham(200, 11, 0, 100);

// 5. Oktant
Bresenham(200, 200, 0, 150);

// 6. Oktant
Bresenham(200, 200, 180, 0);
```

Dieser Code erzeugt bei mir folgendes Bild:



Hier noch ein leicht überarbeite Version des Programmcodes. Die Fallunterscheidungen wurden geschickt durch den Einsatz eines Arrays beseitigt. Auffällig ist, dass die Multiplikation nicht durch ein Shiftoperation ersetzt wurde. Grund hierfür ist, dass der C++ Standard das Verhalten von signed Typen bei einer Linksverschiebung bzw. bei einer Rechtsverschiebung mit den Operatoren << bzw >> nicht definiert. Die Bitverschiebung ist nur für unsigned int definiert.

Früher war es in Programmiererkreisen üblich, Ganzzahlmultiplikationen mit 2 bzw. Ganzzahldivisionen durch 2 mit Hilfe der entsprechenden Shift-Operationen durchzuführen, da dies einen kleinen Performancegewinn im Vergleich zur „echten" mathematischen Operation darstellte. Heutzutage ist dieser Performancegewinn nicht mehr vorhanden, da alle derzeitigen Architekturen auf solche Operationen optimiert sind. Aus Gründen der Leserlichkeit sollte man also heutzutage nicht mehr auf diese Art der trickreichen Programmierung zurückgreifen. Darüber hinaus optimiert der Compiler in den meisten Fällen solche Dinge automatisch.

```
/*
      Beschreibung zum Algorithmus:
      http://turing.fh-landshut.de/~jamann/Bresenham.doc

      Autor: Julian Amann
      Kontakt: vertexwahn@gmx.de
      Erstellt am: 19.02.2005

      Ich setzte ein kartesisches Rechsthaendiges Koordinatensystem und
      eine SetPixel Funktion voraus.
*/

void Bresenham(int d1x, int d1y, int d2x, int d2y)
{
      bool Z = true;      // zur Bestimmung der Bewegunsrichtung
      int *m;             // Bewegungsrichtung/Vektoren

      // delta a, delta b und Bewegungsrichtung bestimmen
```

```
      int dx = abs(d2x) - abs(d1x);
      int dy = abs(d2y) - abs(d1y);

      int da = abs(dx), db = abs(dy);

      if(da-db<0)
      {
            int tmp = da;
            da = db;
            db = tmp;
            Z = false;
      }

      bool X = (dx >= 0);
      bool Y = (dy >= 0);

    int array[8][4] =
    {
        { 0,-1,-1,-1 },
        { 0,-1,1,-1 },
        { 0,1,-1,1 },
        { 0,1,1,1 },
        { -1,0,-1,-1 },
        { 1,0,1,-1 },
        { -1,0,-1,1 },
        { 1,0,1,1 },
    };

    m = array[(X?1:0) + (Y?2:0) + (Z?4:0)];

    int gradient = 2 * db - da;
    int x = d1x;
    int y = d1y;

    SetPixel(x, y);

    for(int i = 0; i < da; i++)
    {
            if(gradient >= 0)
        {
                  x = x + m[2];
            y = y + m[3];
            gradient = gradient + 2 * db - 2 * da;
        }
        else
        {
            x = x + m[0];
            y = y + m[1];
            gradient = gradient + 2 * db;
        }

        SetPixel(x, y);
    }
}
```

Es wirkt sich bestimmt positiv auf die Laufzeit aus, wenn man das Array global anlegt und nicht bei jedem Funktionsaufruf neu erschafft – man könnte hier das int Array auch alternativ als static const int Array definieren.

Es gibt bestimmt noch weitere Verbesserungen für obigen Programmcode. Jedoch sollte das Prinzip, das hinter dem Bresenham Algorithmus liegt verstanden worden sein – wie schon

erwähnt – „Das Optimieren des Codes überlasse ich dem Leser. Mein Ziel war es nur dem Leser die Idee hinter dem Bresenham Algorithmus zu vermitteln.“

The End

Fragen, Anregungen, Kritik, Beschwerden? Einfach mailen: vertexwahn@gmx.de