4. Zweidimensionale Zeichenalgorithmen

4.1. Grundlegende Rastergrafik-Algorithmen

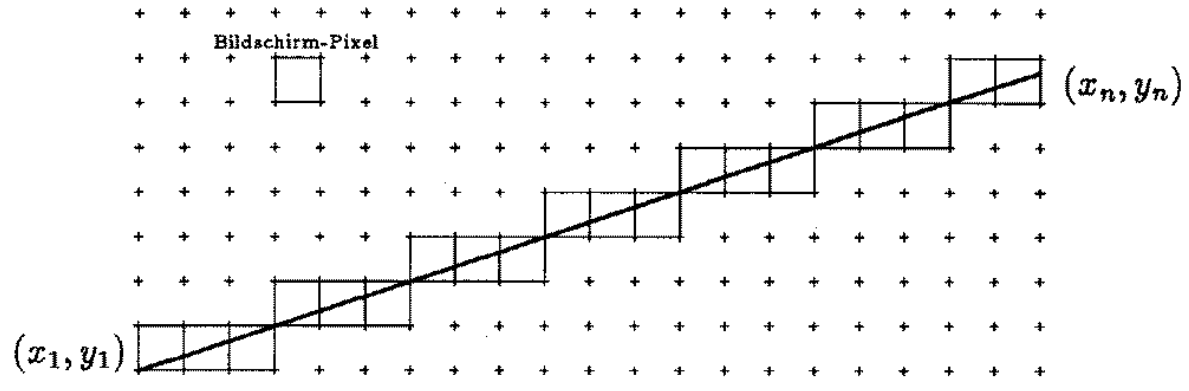
Problem:
virtuelles Zeichenblatt mit stetigen Koordinaten (float *x*, float *y*)
$\rightarrow$ reales Rasterdisplay mit diskreten Koordinaten (int *X*, int *Y*)

Nicht-ganzzahligen Punkten (*x*, *y*) werden ganzzahlige *Alias-Punkte* (*X*, *Y*) zugeordnet.
Dabei: Ungenauigkeiten (und im Extremfall sehr unschöne Effekte).

Ziel 1: Linien sollen ihrem idealen Verlauf möglichst nahe kommen
Ziel 2: Ausgleich möglicher, verbleibender Störeffekte.

Zunächst zu Ziel 1:
Raster-Konvertierungen für Geraden(-stücke), Kreise, Ellipsen...

Zunächst: Linien mit Dicke von 1 Pixel darstellen
$\Rightarrow$ für Gerade mit Steigung zwischen –1 und 1: setze in jeder Spalte genau 1 Pixel.
Für andere Geraden: Symmetrie ausnutzen (Koordinaten vertauschen)!



(aus Rauber 1993)

Abb.: Die gesetzten Pixel sind als Quadrate dargestellt.
Naiver Algorithmus:

aus *x*-Werten von x_1 bis x_n die *y*-Werte nach Geradengleichung $y = mx+b$ berechnen und auf nächstgelegene Integer-Zahl runden.

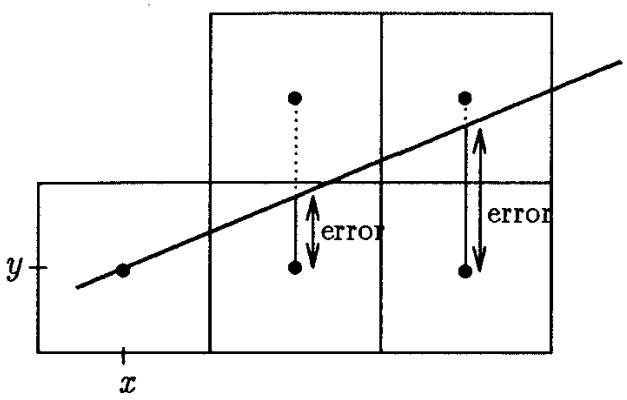
Nachteile:

- Floating-Point-Arithmetik (langsamer als Integer-Arithmetik)
- Multiplikationen
- Rundungsoperationen

möglichst einfache Operationen: wichtig auch für evtl. Hardware-Realisierung!

schrittweise Verbesserung:

- *y* inkrementieren statt durch Multiplikation berechnen
- Rundungsoperation durch Variable *error* und Vergleichsabfrage ersetzen (*error* liefert Abstand der idealen Geraden vom Pixel):



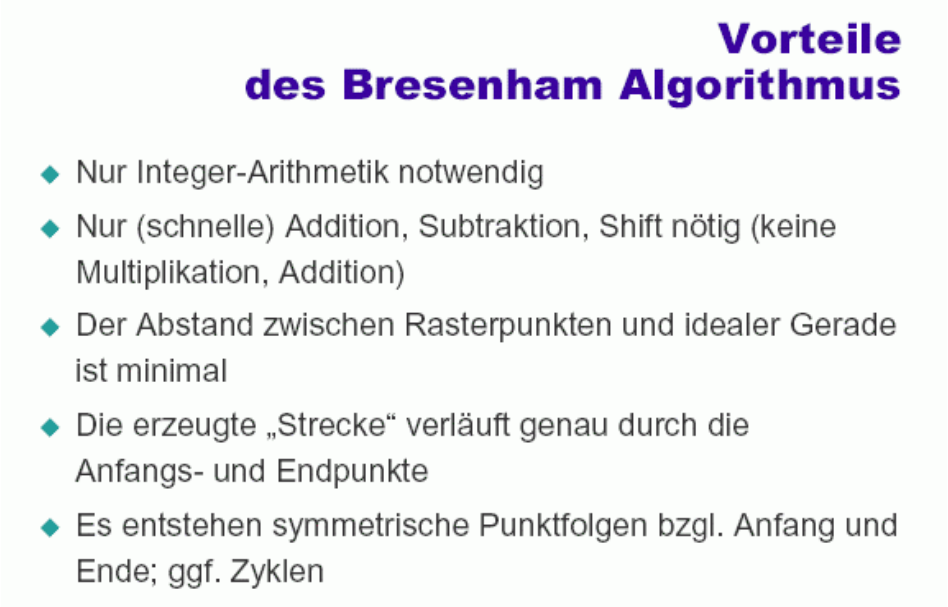
wenn *error* >= 0.5 ist, setze Pixel um 1 höher und dekrementiere *error* um 1. Damit optimale Nähe der gesetzen Pixel zur idealen Geraden gewährleistet.

Immer noch floating point-Operationen
$\Rightarrow$ weitere Verbesserung: alles mit $dx = x_n - x_1$ multiplizieren, dann Rechnung nur noch im Integer-Bereich!

Zusätzlich wird error mit $-dx/2$ (bzw. $-\text{int}(dx/2)$) statt mit 0 initialisiert ($\Rightarrow$ Vergleich nur noch mit 0 statt mit 0,5).

Ergebnis:
Bresenham-Algorithmus zur Linienrasterung (1965)

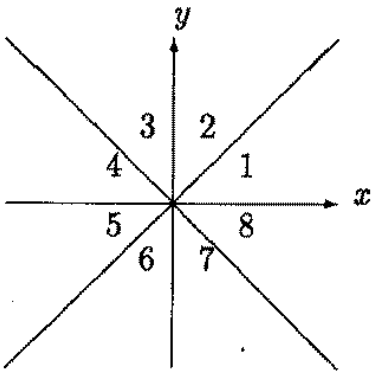
```
void Linie (int x1, int y1, int xn, int yn,
            int value)
/* Anfangspunkt (x1, y1), Endpunkt (xn, yn),
   Steigung zwischen 0 und 1 angenommen */
{
int error, x, y, dx, dy;
dx = xn – x1; dy = yn – y1;
error = –dx/2; y = y1;
for (x = x1; x <= xn; x++)
   {
   set_pixel(x, y, value);
   error += dy;
   if (error >= 0)
      {
      y++;
      error –= dx;
      }
   }
}
```



In obiger Form nur für 1. Oktanden.
Verallgemeinerung: Ausnutzung der Symmetrie.

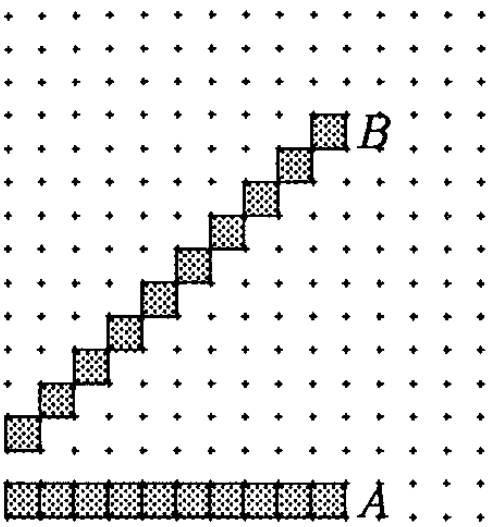
"*Treibende Achse*" = Achse, deren Werte mit fester Schrittweite 1 durchlaufen werden (im obigen Fall: *x*-Achse).

Fallunterscheidung nach dem Oktanden der Geraden:



Oktand	treibende Achse	andere Achse
1	x	inkrementiert
2	y	inkrementiert
3	y	dekrementiert
4	x	dekrementiert
5	x	inkrementiert
6	y	inkrementiert
7	y	dekrementiert
8	x	dekrementiert

Nachteil:
Linien unterschiedlicher Steigung erscheinen unterschiedlich hell.



Linie *B* ist $\sqrt{2}$ -mal länger als Linie *A*, hat aber dieselbe Zahl von Pixeln $\Rightarrow$ erscheint schwächer!

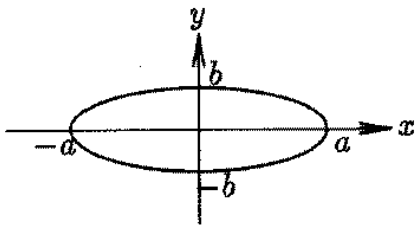
Abhilfe: Intensität (Parameter **`value`**) mit der Steigung der Linie erhöhen.

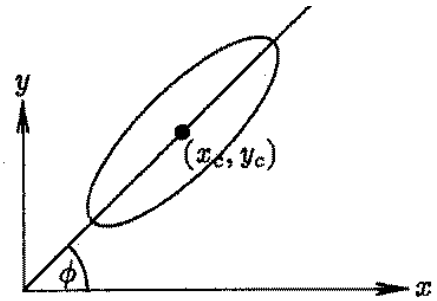
Rasterung von Ellipsen

Parametermethode (auch: trigonometrische Methode):
Ellipsengleichung in Parameterform (Parameter θ = Winkel im Mittelpunkt):

$$x = a \cos\theta, \quad y = b \sin\theta.$$

θ durchläuft die Werte von 0 bis 360° bzw. bis 2π.
a und *b* sind die Abschnitte auf den Hauptachsen. Hier Hauptachsen = Koordinatenachsen.





Um den Winkel ϕ gedrehte Ellipse mit Mittelpunkt (x_c, y_c):

$x = a \cos\phi \cos\theta - b \sin\phi \sin\theta + x_c = A \cos\theta - B \sin\theta + x_c$
$y = a \sin\phi \cos\theta + b \cos\phi \sin\theta + y_c = C \cos\theta + D \sin\theta + y_c$

mit $A = a \cos\phi$, $B = b \sin\phi$, $C = a \sin\phi$, $D = b \cos\phi$

(durch Anwendung einer Rotation und Translation auf die ungedrehte Ellipse, vgl. Transformationsmatrizen (später) bzw. Lineare Algebra).

- Durchlaufe äquidistante Parameterwerte
- verbinde die zugehörigen Ellipsenpunkte durch Linien (Geradensegmente).

n = Anzahl gewünschter Geradensegmente $\Rightarrow$
Schrittweite **`incr`** = $2\pi/n$.

Zu jedem $\theta_i = \theta_{i-1}$ + **`incr`** berechne zugehörigen Punkt (x_i, y_i), zeichne Linie von (x_{i-1}, y_{i-1}) nach (x_i, y_i).

Dazu: Merken des vorangegangenen Punktes.

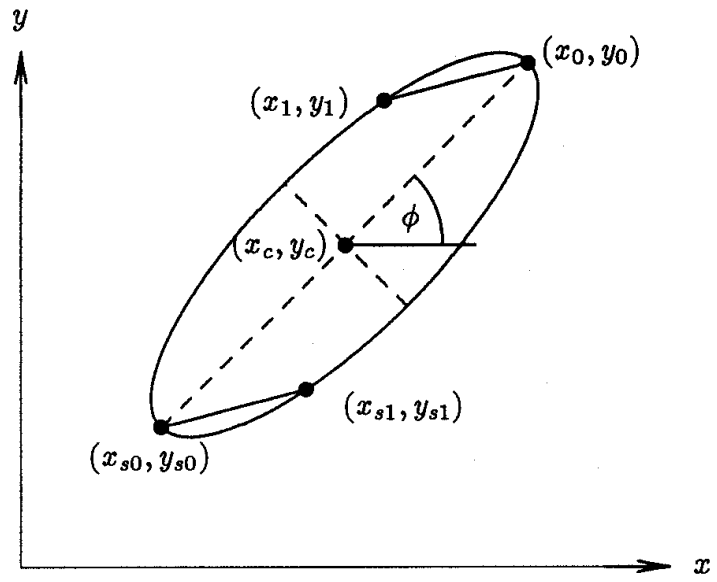
Implementation:

```
void ellipse(float a, float b, float phi,
         float xc, float yc, int n, int value)
{
float cosphi, sinphi, theta, pi, costheta,
   sintheta, A, B, C, D, x0, x1, y0, y1, incr;
int i;
pi = 4*arctan(1);
cosphi = sintab[pi/2 + phi];
sinphi = sintab[phi];
A = a*cosphi; B = b*sinphi;
C = a*sinphi; D = b*cosphi;
x0 = A + xc; y0 = C + yc;
theta = 0; incr = 2*pi/n;
for (j=1; j <= n; j++)
   {
   theta += incr;
   costheta = sintab[pi/2 + theta];
   sintheta = sintab[theta];
   x1 = A*costheta – B*sintheta + xc;
   y1 = C*costheta + D*sintheta + yc;
   Draw_line(x0, y0, x1, y1, value);
   x0 = x1; y0 = y1;
   }
}
```

Kosinusberechnung wird auf Sinus zurückgeführt.

Sinusberechnung wird hier durch Lookup-Table ersetzt.
Wegen sin x = sin(π–x) und sin x = –sin(x–π) braucht man nur die Werte zwischen 0 und π/2 zu speichern!

Verbesserung des Algorithmus (Halbierung der Laufzeit):
Ausnutzung der Symmetrie der Ellipse.



Nachteil der Parametermethode:
Feste Unterteilung der Parameterwerte ⇒ Ungenauigkeit an Stellen starker Krümmung.

Abhilfe: nichtlineare Einteilung der Parameterwerte (**`incr`** abhängig von der Krümmung).

anderer Ansatz:
Polynom-Methode

Ausgangspunkt jetzt:
kartesische Achsenabschnittsform der Ellipsengleichung (Polynom 2. Grades).
Ungedrehte Ellipse:

$$(x/a)^2 + (y/b)^2 = 1$$

Vorgehen:
Auflösen nach *y*, *x* durchläuft alle in Frage kommenden Werte in Einerschritten, 2 *y*-Werte werden jeweils berechnet.

Nachteil: hoher Rechenaufwand (Quadratwurzeln!).

alternativer Ansatz:
Scangeraden-Methode
(auch für andere Arten von Kurven)

Prinzip: horizontale Scangerade wird über die (ideale) Ellipse geschoben, für jede *y*-Position werden die Schnittpunkte bestimmt und im Raster approximiert.

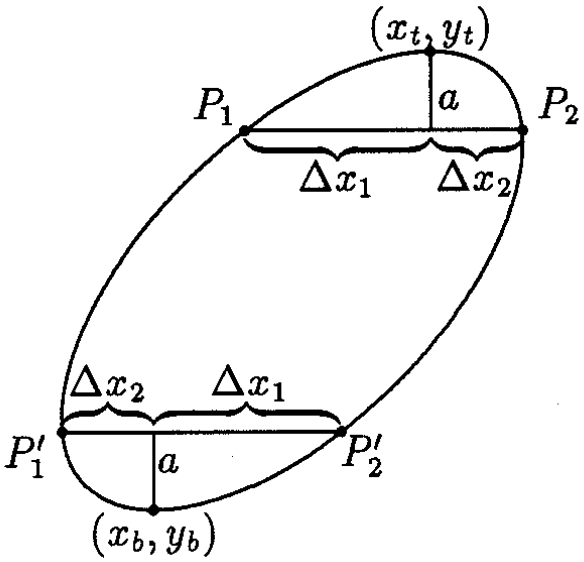
Rechnung geht wieder aus von der Achsenabschnittsform (s.o.)
Für gedrehte und verschobene Ellipsen: *x* und *y* entsprechend substituieren durch $x \cos \phi - y \sin \phi + x_c$
bzw. $x \sin \phi + y \cos \phi + y_c$.
Scangeraden-Gleichung: $y = y_i$ (= konst.)

Schnittpunkt-Berechnung: Einsetzen von $y = y_i$ in die Ellipsengleichung und Auflösen nach *x* (quadratische Gleichung; genaue Rechnung s. Rauber 1993, S. 31 ff.). Ergebnis: zwei *x*-Werte für die Schnittpunkte.

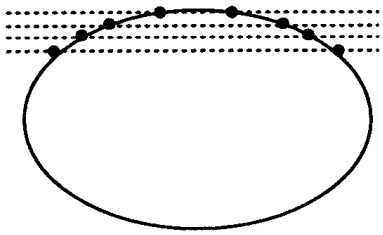
Berechnung nur nötig für solche Scangeraden, die die Ellipse schneiden:
Berechnung der oberen und unteren Ausdehnung y_t und y_b der Ellipse durch

$$y_{t,b} = \pm \sqrt{(b^2 \cos^2 \phi + a^2 \sin^2 \phi)}.$$

Ferner: Ausnutzung der Symmetrie:

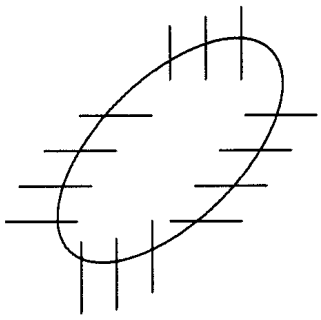


Nachteil der Methode:
In jeder Bildschirmzeile, die die Ellipse trifft (außer der oberen und unteren Tangente), werden genau 2 Pixel gesetzt
$\Rightarrow$ in Bereichen mit Steigung nahe 0 können "Lücken" auftreten



Abhilfe:

- überprüfe *x*-Abstand zum zuletzt gesetzten Pixel, wenn > 1: setze die dazwischenliegenden Pixel der Scangeraden
- oder: verwende vertikale Scangeraden in Bereichen mit Steigung zwischen –1 und 1:



Für letztere Vorgehensweise benötigt man die Parameterwerte mit Tangentensteigung +1 und –1:

bei $\theta_1 = \arctan(-b / (a \tan(\pi/4 - \phi)))$, $\theta_2 = \theta_1 + \pi$ ist die Steigung +1,
bei $\theta_3 = \arctan(-b / (a \tan(\pi/4 - \phi)))$, $\theta_4 = \theta_3 + \pi$ ist sie –1.
(Herleitung bei Rauber 1993.)

Vorteil der Scangeraden-Methode:
Direkt auch zum Füllen von Ellipsen geeignet (verbinde die Schnittpunkte jeweils durch horizontales Scangeraden-Stück).

Kreise:
Spezialfälle der Ellipsen.
Für Kreise können (theoretisch) weitere Symmetrien ausgenutzt werden.
Beachte aber:
Für viele Monitore sind Höhe und Breite der Pixel ungleich
$\Rightarrow$ Kreise müssen als "echte" Ellipsen (mit $a \neq b$) konstruiert werden.

Wegen seiner Einfachheit erwähnen wir den Kreisalgorithmus von Bresenham (Analogie zum Geraden-Algorithmus):

- es wird nur die Kreislinie im 2. Oktanden konstruiert (der Rest folgt aus Symmetrieoperationen)
- dort gibt es bei "x++" nur die beiden Fälle "y bleibt gleich" oder "y--"
- die Abstände der beiden Kandidaten-Punkte (wenn (x_i, y_i) gezeichnet ist) zum Mittelpunkt sind $d_s = (x_i + 1)^2 + y_i^2 - r^2$ und $d_t = (x_i + 1)^2 + (y_i - 1)^2 - r^2$
- d_s ist immer >0, d_t immer <0
- Entscheidungsvariable für die Auswahl des nächsten Pixels: $d = d_s + d_t$
- diese kann rekursiv berechnet werden:
  für $d > 0$: x++; d += $(4x + 6)$;
  für $d <= 0$: x++; y--; d += $(4(x-y)+10)$.

Beachte: Multiplikationen mit Zweierpotenzen lassen sich effizient als Shift-Operationen realisieren.
Damit kommt auch der Bresenham-Kreis-Algorithmus ohne "echte" Multiplikationen aus.

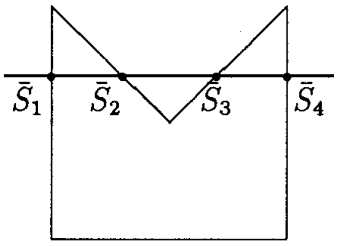
Pro Punkt: durchschnittl. $\sqrt{2}$ Additionen, $\sqrt{2}$ Shifts, 2 Inkremente und 1 Test.

Füllen von Polygonen und geschlossenen Kurven

Die Scangeraden-Methode

Voraussetzung: geschlossenes Polygon *P* liegt in geometrischer Beschreibung vor.

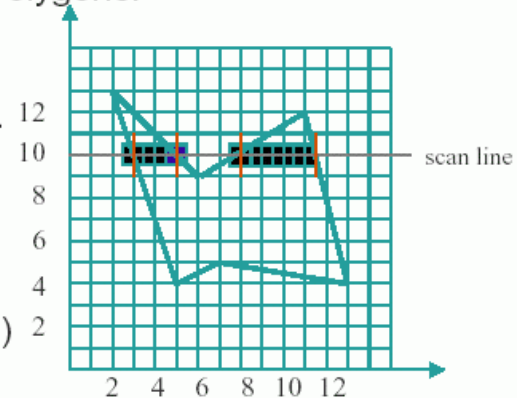
- Scangerade parallel zur *x*-Achse wird von unten nach oben über *P* geschoben,
- für jede Lage der Scangeraden werden die Schnittpunkte mit dem Polygon berechnet.
- Für konvexe Polygone gibt es genau 2 Schnittpunkte oder keinen,
- für konkave Polygone gibt es eine gerade Zahl von Schnittpunkten, die nach steigendem *x* geordnet werden.



1. **Finde Schnittpunkt der scan line mit allen Kanten des Polygons**
2. **Sortiere Schnittpunkte nach wachsender x-Koordinate**
3. **Fülle alle Pixel zwischen Paaren aufeinanderfolgender Schnittpunkte die im Inneren des Polygons liegen.**

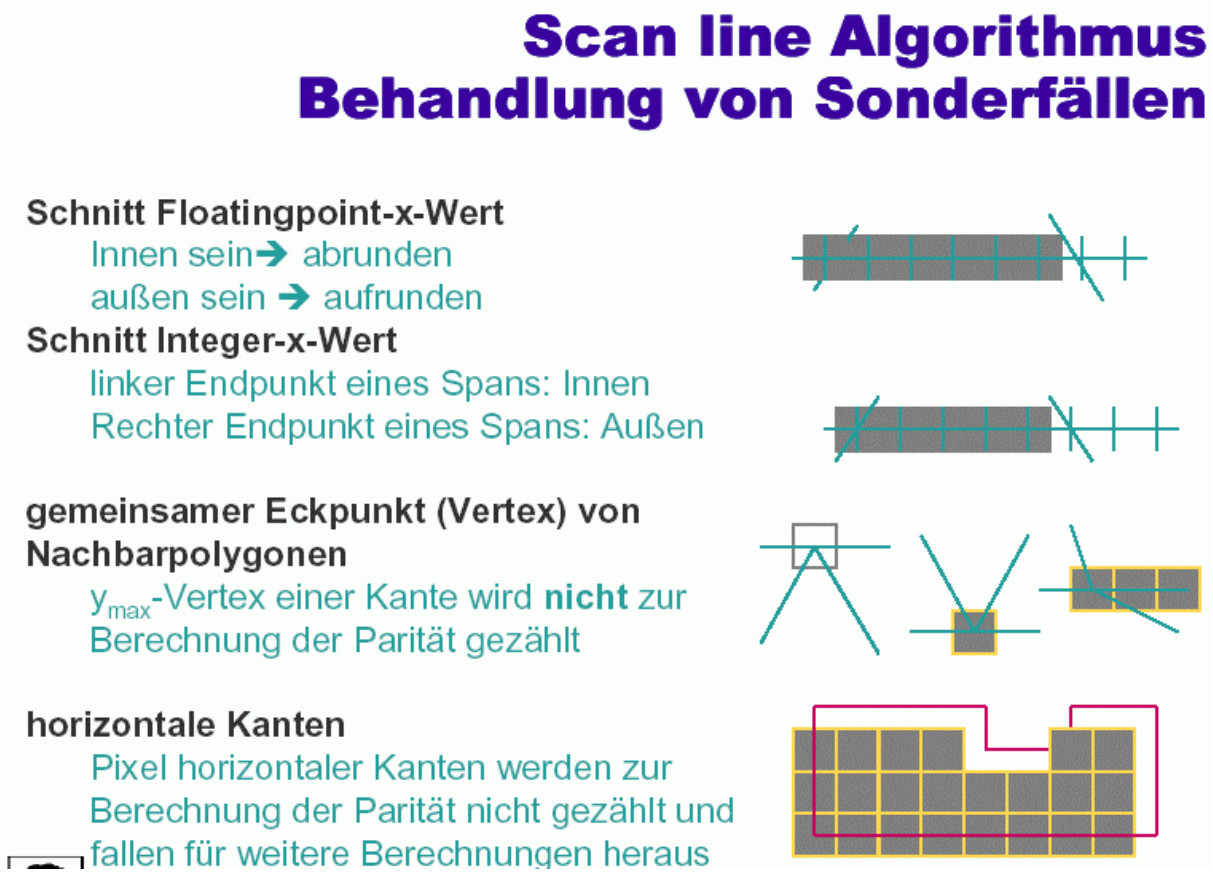
(Dieser Algorithmus behandelt beliebige Polygone:
Regel von der ungeraden Parität:
Parität ist am Anfang 0 und wird mit jedem Schnittpunkt um eins inkrementiert.
Pixel wird gesetzt falls Parität ungerade.)

Anm.: Alle Polygoneckpunkte sind auf Integerwerte gerundet.
Bei allen konvexen Polygonen (Dreiecken) ensteht nur ein Span



(zur Paritätsregel: vgl. "even-odd Filling rule" bei PostScript!)

Es sind Sonderfälle zu beachten:



(Krömker 2001)

Nach der Regel für den y_{max}-Knoten einer Kante werden in diesem Beisp. für die Scangerade y=0 die Kanten *a* und *b*, für die Scangerade y=2 die Kanten *c* und *d*, für die Scangerade y=4 aber keine Kante geschnitten:

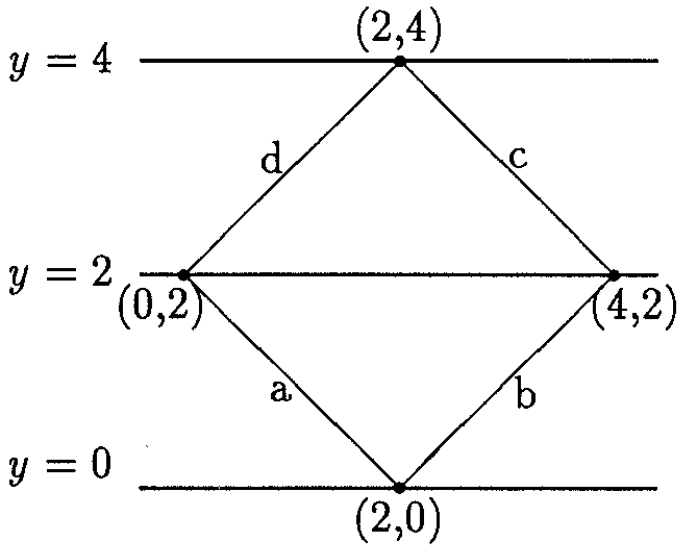
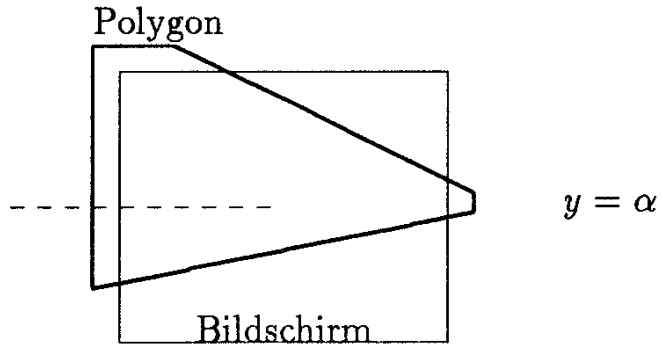


Abb. (*) (aus Rauber 1993)

Vorsicht beim Füllen von Polygonen, die sich über den Bildschirmrand erstrecken:



Wenn nur die sichtbaren Polygonkanten übergeben werden, kann z.B. für die Scangerade $y = \alpha$ nicht entschieden werden, ob die entspr. Bildschirmzeile innerhalb oder außerhalb des Polygons liegt.

Abhilfe: Beim Clipping (siehe später) müssen die Teile des Fensterrandes, die im Polygon liegen, als zusätzliche Kanten an den Scangeraden-Algorithmus übergeben werden.

Genaueres Vorgehen beim Einfärben:

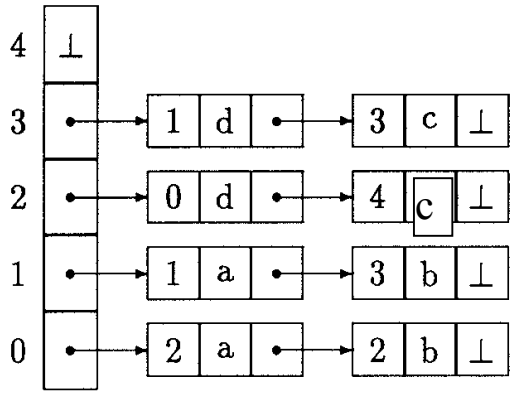
naiver Algorithmus:
äußere Schleife: Scangerade verschieben
innere Schleife: Schnittpunkte mit allen Polygonkanten berechnen

effizienter:
äußere Schleife über die Polygonkanten *e*
innere Schleife: berechne für e alle Schnittpunkte mit allen Scangeraden zwischen $y_{min}(e)$ und $y_{max}(e)$ und lege diese sortiert in einer (globalen) Datenstruktur ab.
Danach durchlaufe die Datenstruktur und färbe die entspr. Segmente für jede Scangerade.

Vorteil: Bei Berechnung der Schnittpunkte kann ein inkrementeller Algorithmus verwendet werden (Variante des Bresenham-Algorithmus).

In der Datenstruktur brauchen für jeden Schnittpunkt nur die *x*-Werte separat abgespeichert zu werden, der *y*-Wert gilt für die gesamte Scangerade.

Dieses Beispiel bezieht sich auf obige Abb. (*):



Der Übersichtlichkeit halber sind hier noch die Labels der Punkte mit angegeben worden (a,b,c,d) – diese braucht man in der Praxis nicht.

Diese Datenstruktur könnte in C etwa so realisiert werden:

```
struct scanline
   {
   struct xpixlist *list;
   struct scanline *next;
   int ypix;
   }
struct xpixlist
   {
   int xpix;
   struct xpixlist *next;
   }
```

Es gibt eine noch platzsparendere Variante, die einen *bucket sort*-Algorithmus zur Erstellung einer Kantenliste ausnutzt (s. Foley et al. 1990).

Für beliebige Kurven als Grenzen der Füllregion muss nur die Berechnung der Schnittpunkte geändert werden.

Die Saatfüll-Methode

wird benutzt, wo umgebende Polygone bereits auf dem Bildschirm dargestellt sind (interaktive Grafik)
keine geometrische Repräsentation des Polygons nötig!

stattdessen: Pixel der Grenzlinie müssen durch spezifische Grenzfarbe erkennbar sein.

Grundidee:
Setze ein Pixel im zu füllenden Bereich
setze dessen Nachbarpixel auf die Füllfarbe (sofern diese nicht die Grenzfarbe haben)
wiederhole dies, bis kein Pixel mehr neu gefärbt werden kann.

Beachte:
Die Nachbarschaft darf keine Diagonal-Pixel enthalten, da sonst vom Bresenham-Algorithmus gerasterte Geraden "undicht" sind:

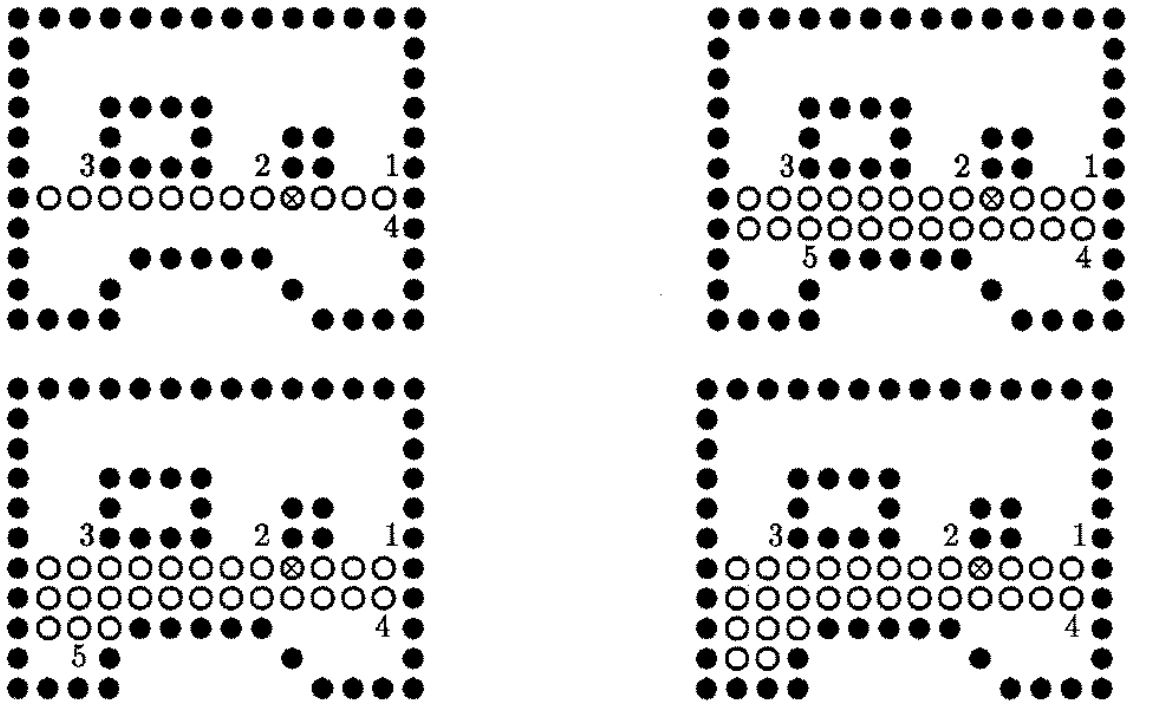
Implementierung:

naiver Ansatz: rekursiver Aufruf; ungünstig wegen Beanspruchung des Rekursionsstacks

effizienter:
"Horizontalisierung" des Füllens; 2 Schleifen: innere erzeugt "Pixelläufe" in horiz. Richtung, äußere iteriert dies nach oben und unten.

Beachte: Auch hier ist (außer bei konvexen Polygonen) ein Stack erforderlich, der das Auftreten zusätzlicher Pixelläufe verwaltet.

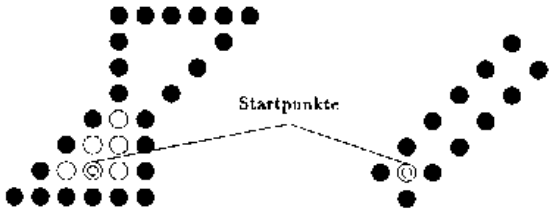
Beispiel:



(aus Rauber 1993)

Hier entstehen bei 1, 2, 3, 4 und 5 jeweils neue Pixelläufe.
Jeder Pixellauf wird durch sein rechtestes Pixel repräsentiert.
Der Algorithmus terminiert, wenn der Stack, der die Pixelläufe verwaltet, leer ist.

Nachteil:
Bei "engen" Polygonen kann der Saatfüll-Algorithmus vorzeitig stoppen. Abhilfe: Zweite Saat setzen.



(aus Fellner 1992)

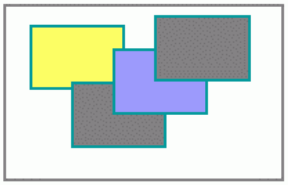
Ausschnittsbildung (Windowing und Klipping)

Um von der gesamten vorhandenen Bildinformation nur einen Ausschnitt darzustellen wird ein üblicherweise rechteckiges Fenster (Window) definiert, dessen Ränder den interessierenden Bildausschnitt begrenzen.

Um ein fehlerfreies Bild zu erhalten, muß die außerhalb des Fensters liegende Bildinformation vor der Bildausgabe abgeschnitten werden (Clipping = Klippen).

In der Regel wird das Fenster in Weltkoordinaten spezifiziert und mit "World-coordinate-window" oder einfach Window" bezeichnet.

Viewport



- Bildschirm in verschiedene Darstellungs-flächen (Viewports) aufteilen.
- Ohne Clipping würden die Inhalte der einzelnenViewports sich gegenseitig beeinflussen, also falsche Bilder erzeugen
- Auf dem Bildschirm wird als Koordinatensystem das Bildschirmkoordintensystem verwendet und das Ausgabefenster in diesem angegeben und Viewport genannt.
- Die Transformation zwischen dem Weltkoordinatensystem und dem Bildschirmkoordinatensystem heißt Window-Viewport-Transformation.

Clipping von Linien

häufige Aufgabe: Abschneiden von Geradensegmenten an einem Polygon (häufig an einem Rechteck). *Clipping*, *Clippen*. (Vgl. Clipping-Pfade in PostScript.)

einfachste Variante:
für jedes Pixel während der Bildschirm-Darstellung prüfen, ob es innerhalb des Clipping-Polygons liegt.

- sinnvoll, wenn Schnittpunktberechnung zu aufwändig
- oder wenn das zu clippende Objekt sehr klein ist (nur wenige Pixel zu prüfen).

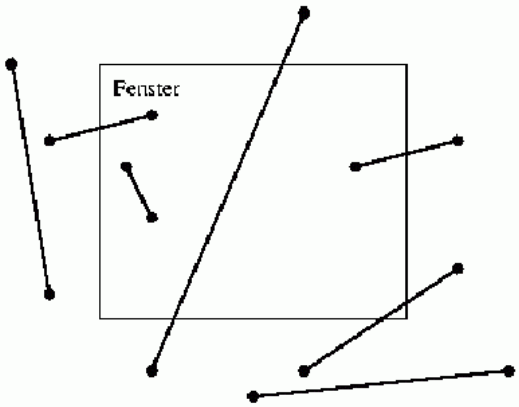
Analytische Methode:

Schnittpunkte berechnen
Fallunterscheidungen schon bei einfachsten Objekten erforderlich.

Clippen eines Geradensegments *s* an einem rechteckigen Fenster:

Clipping von Liniensegmenten

Beim Clipping von Liniensegmenten an einem rechteckigen Fenster (allgemeiner konvexen Objekt) entsteht entweder **kein** oder (bei Unterteilung einer Geraden in sichtbare und unsichtbare Teile) **ein** sichtbarer Teil. Vektoren, deren beide Endpunkte oberhalb, unterhalb, rechts oder links des Fensters liegen, sind völlig unsichtbar. Können diese Vektoren einfach aussortiert werden, so ist eine erhebliche Beschleunigung des Clipping zu erwarten. Der Cohen-Sutherland-Algorithmus (s.u.) nutzt diese Eigenschaft.



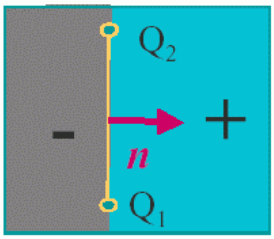
(Krömker 2001)

Clipping von Liniensegmenten am Halbraum

Lian-Barsky-Algorithmus:

Jede (Fenster)-Kante wird als implizite Gerade dargestellt:

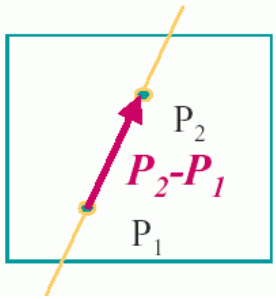
$$Q_1 = (x_1, y_1), Q_2 = (x_2, y_2),$$
$$n = (\Delta y, -\Delta x) = (y_2 - y_1, -(x_2 - x_1))$$
$$P = (x, y)$$
$$E(P) = n * P - n * Q_1$$



Konvention: Normale zeigt ins Innere!

Jedes Liniensegment wird parametrisch dargestellt:

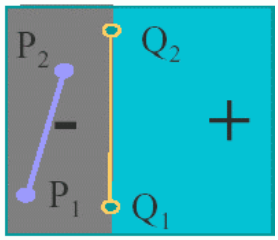
$$l(t) = P_1 + t(P_2 - P_1)$$



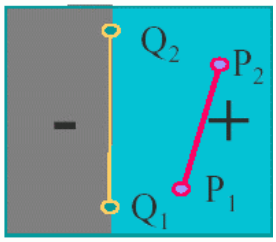
E(*P*) liefert den vorzeichenbehafteten Abstand des Punktes *P* von der Fensterkante (positiv = innen, negativ = außen).

Nun gibt es folgende 3 Fälle
(Sonderfälle horizontaler oder vertikaler Geraden sind einfach und sind gesondert zu betrachten

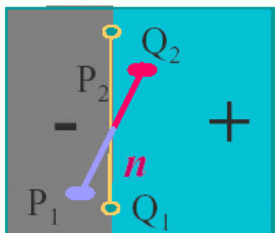
1.) P_1 *und* P_2 liegen außen:
$E(P_1) \le 0,\ E(P_2) \le 0$



2.) P_1 *und* P_2 liegen innen:
$E(P_1) \ge 0,\ E(P_2) \ge 0$

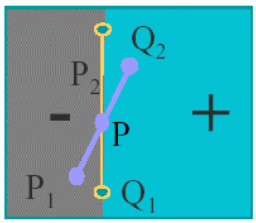


3.) P_1 *und* P_2 liegen auf verschiedenen Seiten
$E(P_1) < 0,\ E(P_2) > 0,$ bzw.
$E(P_1) > 0,\ E(P_2) < 0$



Im Fall 3 müssen wir den Schnittpunkt P berechnen:
Einsetzen der parametrischen in die implizite Gleichung liefert:

$$
\begin{aligned}
& E(P_1 + t(P_2 - P_1)) = 0 \\
\Leftrightarrow\ & (P_1 + t(P_1 - P_2))n - Q_1 n = 0 \\
\Leftrightarrow\ & t = \frac{Q_1 n - P_1 n}{(P_1 - P_2)n} \\
\Rightarrow\ & P = P_1 + \frac{Q_1 n - P_1 n}{(P_1 - P_2)n}(P_2 - P_1)
\end{aligned}
$$



Der Lian-Barsky-Algorithmus verallgemeinert auf 3D:

- Halbräume sind nun durch Ebenen definiert.
- Ebene wird auch in impliziter Form durch Punkt und Normale beschrieben.
- Parametrische Form für das Liniensegment bleibt unverändert.

ABER Algorithmus ist zu aufwendig!

Vorgehensweise, wenn zu clippende Linie und Fensterkante beide in Parameterform:

Endpunkte von *s*: P_1 und P_2
Parametergleichung der Geraden, auf der *s* liegt:
$g = P_1 + t(P_2 - P_1)$, *t* beliebig
Für *s* selbst: $t \in [0; 1]$.

Fenster-Eckpunkte: $A = (x_{min}, y_{min})$, $B = (x_{max}, y_{min})$,
$C = (x_{max}, y_{max})$, $D = (x_{min}, y_{max})$.
Gleichung der Geraden durch die Fenster-Eckpunkte *A*, *B*:
$f = A + u(B - A)$. (analog für *BC*, *CD*, *DA*.)
Schnittpunkt aus Gleichsetzen der Geradengleichungen:
$P_1 + t(P_2 - P_1) = A + u(B - A)$
(entspr. 2 Gleichungen mit 2 Unbek.) $\Rightarrow$ Lösungspaar *t*, *u*.

$0 \le t \le 1 \Leftrightarrow$ Schnittpunkt liegt auf dem gegebenen Segment
$0 \le u \le 1 \Leftrightarrow$ Schnittpunkt liegt auf der Fensterkante.

Fall A: beide Endpunkte von *s* liegen innerhalb des Fensters
⇒ *s* liegt ganz innerhalb des Fensters
Fall B: ein Endpkt. liegt innerhalb, einer außerhalb
⇒ (nur) ein Schnittpunkt zu bestimmen
Fall C: beide Entpkte liegen außerhalb des Fensters
⇒ *s* **kann** durch das Fenster verlaufen; beide Schnitt-
punkte zu bestimmen.

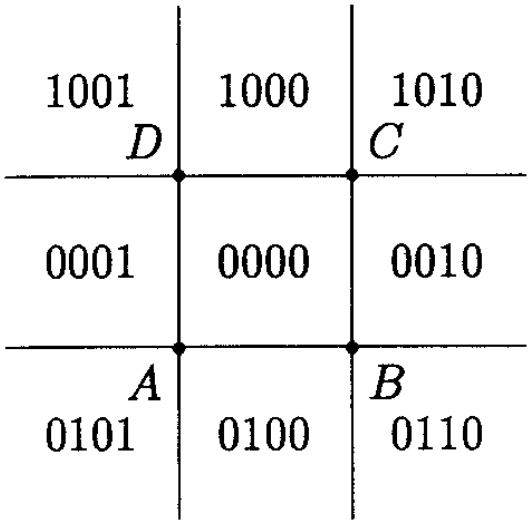
Ziel: Zahl der Schnittpunktberechnungen vermindern!

Beispiel: Wenn beide Endpunkte von *s* links des Fensters liegen, kann s nicht durch das Fenster verlaufen.
Ebenso: wenn beide rechts, beide oberhalb, beide unterhalb.

Systematisierung:

<u>Algorithmus von Cohen und Sutherland</u>

Kennzeichnung der 9 Bereiche in Bezug auf das Fenster durch 4 Bits. Bit 0 gesetzt für Bereiche links des Fensters, 1 für rechts, 2 für unten, 3 für oben: ("*outcodes*")



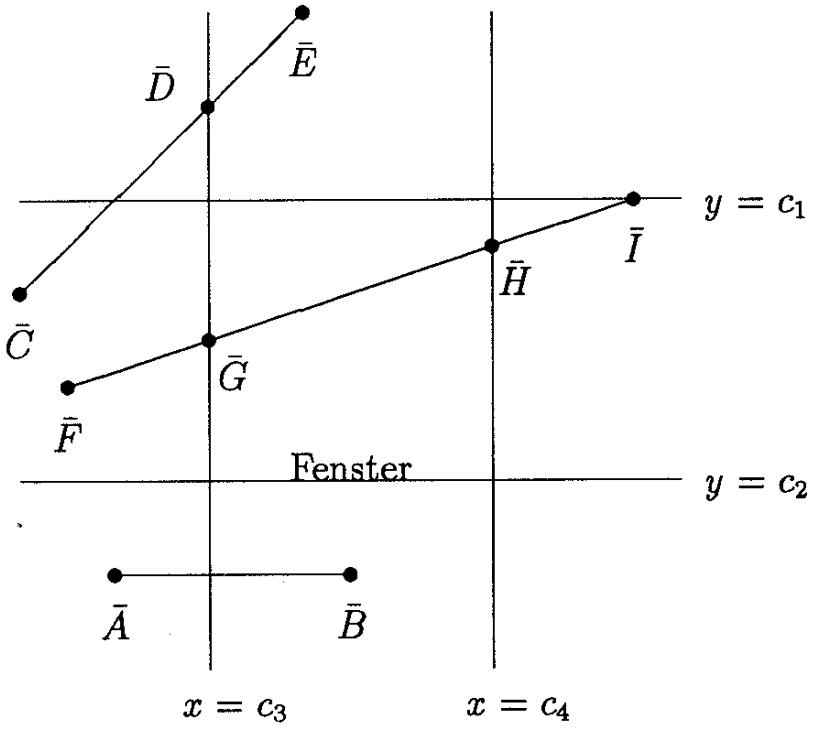
⇒ Bitmuster von horizontal oder vertikal benachbarten Bereichen unterscheiden sich in genau 1 Bitposition.

Ordne den Endpunkten von *s* ihre Bitcodes *c*1 und *c*2 zu.

$c1 \mid c2 = 0$ (bitweises Oder) $\Leftrightarrow$ $c1$ und $c2$ sind beide 0000 $\Leftrightarrow$ s liegt ganz im Fenster.

$c1 \,\&\, c2 \neq 0$ (bitweises Und) $\Leftrightarrow$ $c1$ und $c2$ haben in mindestens einer Stelle gemeinsame Bits $\Leftrightarrow$ s liegt auf einer Seite des Fensters, kann nicht durch das Fenster gehen!

in allen übrigen Fällen: Teile s mittels einer der Fenstergeraden in zwei Teile (hier Schnittpunktberechnung erforderlich!), wiederhole das Verfahren für beide Teile (einer muss ganz außen liegen!). Mit welcher Fenstergerade schneiden: entscheide mittels des Bitcodes z.B. von $c1$ (wenn $c1 \neq 0$, sonst $c2$).
Wenn Bit 3 gesetzt: entspr. Punkt liegt ganz oberhalb des Fensters; wähle die obere Begrenzungsgerade. Analog für die anderen Bits.



(aus Rauber 1993)

3 Beispielfälle für die Anwendung des Algorithmus

Programmskizze:

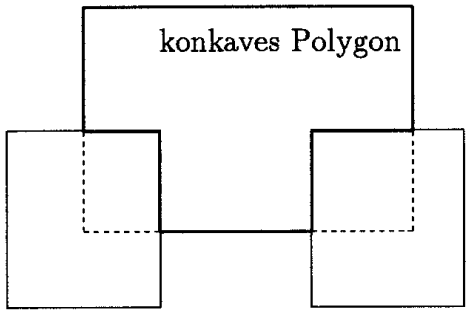
```
void CS_clip (float x1, float y1, float x2,
              float y2, float xmin, float xmax,
              float ymin, float ymax,
              int value)
{
int c1, c2; float xs, ys;
c1 = code(x1, y1);  c2 = code(x2, y2);
if (c1 | c2 == 0x0)
   Draw_Line(x1, y1, x2, y2, value);
else
   if (c1 & c2 != 0x0)
      return;
   else
      {
      intersect(xs, ys, x1, y1, x2, y2,
                xmin, xmax, ymin, ymax);
      if (is_outside(x1, y1))
         CS_clip(xs, ys, x2, y2, xmin, xmax,
                 ymin, ymax, value);
      else
         CS_clip(x1, y1, xs, ys, xmin, xmax,
                 ymin, ymax, value);
      }
}
```

`intersect` berechnet in den Rückgabeparametern **`xs`** und **`ys`** einen Schnittpunkt zwischen dem Geradensegment und einer der Fenstergeraden, die anhand der 4-Bit-Zahl der Endpunkte des Geradensegments ausgewählt wird.
`is_outside` berechnet, ob der Punkt (**`x1`**, **`y1`**) bzgl. derselben Fenstergeraden außerhalb des Fensters liegt.

(nach Rauber 1993)

Clipping bzgl. beliebiger (auch konkaver) Polygone:

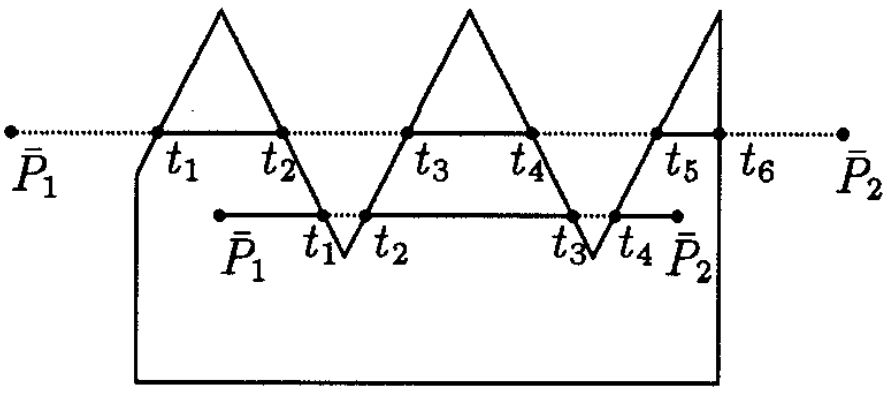
wichtig z.B. bei Systemen, die das Überlappen mehrerer Fenster auf einem Bildschirm erlauben!



Vorgehensweise:
Bestimme alle *n* Schnittpunkte des Geradensegments *s* mit dem Clipping-Polygon *P*.
Ordne diese nach aufsteigenden Parameterwerten.
Entscheide, welche von den *n*+1 Segmenten von *s* zwischen den Schnittpunkten im Inneren von *P* liegen ("sichtbar sind"):

Liegt der Endpunkt P_1 von *s* innerhalb von *P*?
Teststrahl von P_1 aus, zähle Anzahl der Schnittpunkte mit *P*, Paritätsprüfung (ungerade $\Rightarrow$ innerhalb).

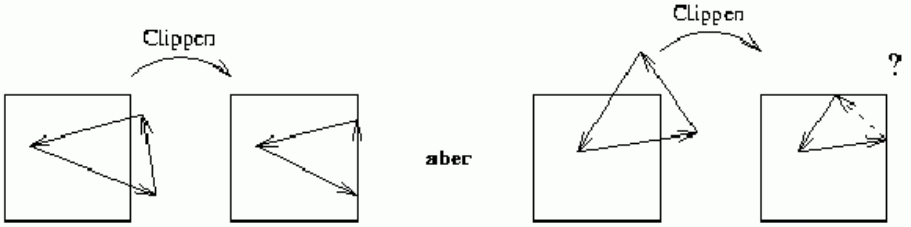
Wenn P_1 innerhalb von *P*: das bei P_1 beginnende und danach jedes zweite Parameterintervall liegen in *P*.
Sonst genau die hierzu komplementären.



(aus Rauber 1993).

Polygon-Clipping

Ein Klipping-Algorithmus für geschlossene Polygone muß als Ergebnis des Klippingvorgangs wieder geschlossene Polygone liefern. Dies ist nur durch richtige Einbeziehung von Teilen der Fensterbegrenzung in das ,,geklippte" Polygon möglich.



Probleme entstehen, wenn das zu ,,klippende" Polygon Ecken des Fensters umschließt.

Verschiedene Methoden wurden vorgeschlagen, um diese Sonderfälle behandeln zu können. Die einfachste Methode ist die Umkehrung der Schleifenschachtelung: Das gesamte Polygon wird zunächst an einer Fen ergrenze geklippt, anschließend wird an der nächsten Fenstergrenze geklippt etc Dies ist die wesentliche Idee des Sutherland-Hodgman-Algorithmus.

Sutherland-Hodgeman-Algorithmus