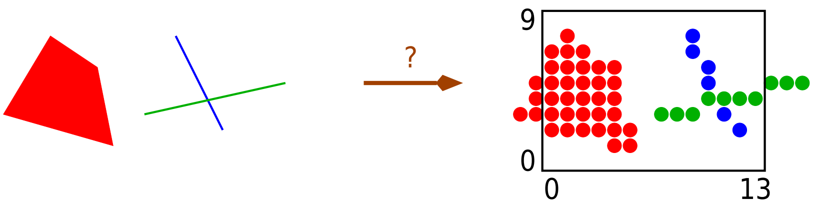
3 Scan Conversion

Inhalt

Problem: Gegeben ist eine Menge von „Objekten“.

- Linie von $(7,3)$ nach $(16,5)$
- Linie von $(9,8)$ nach $(12,2)$
- ausgefülltes Polygon mit den Eckpunkten $(5,1)$, $(4,6)$, $(1,8)$, $(-2,3)$
- ...

Gesucht sind die zu jedem Objekt gehörenden Rasterpunkte (picture elements, **„Pixels“**).



„logischer Bildwiederholspeicher“,
Pixel Map,
Pixmap

Bemerkung 3.1: Die Pixmap muss nicht mit dem aktuellen (dem Video Controller zugänglichen) Bildwiederholspeicher übereinstimmen!

Anwendung 1: „Doppelpufferung“

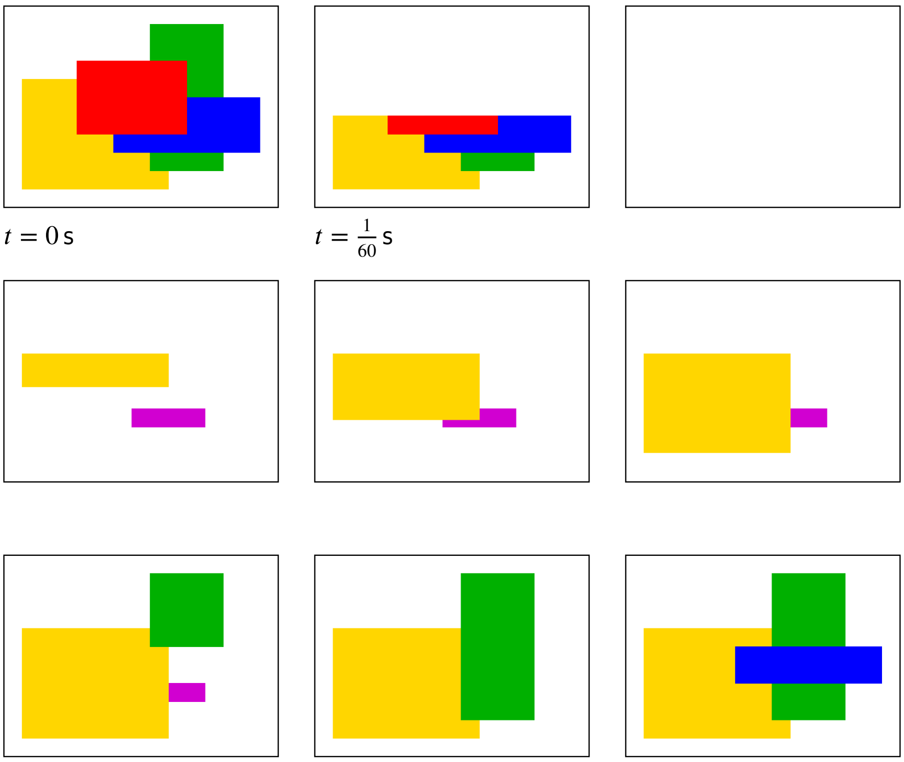
Problem: Gleichzeitiges Verändern und Auslesen des Bildwiederholspeichers führt zu unerwünschten Effekten.

(meist: Verzerrungen, Flackern)

Beispiel: Aus einem (relativ komplexen) Bild soll ein Objekt gelöscht werden.

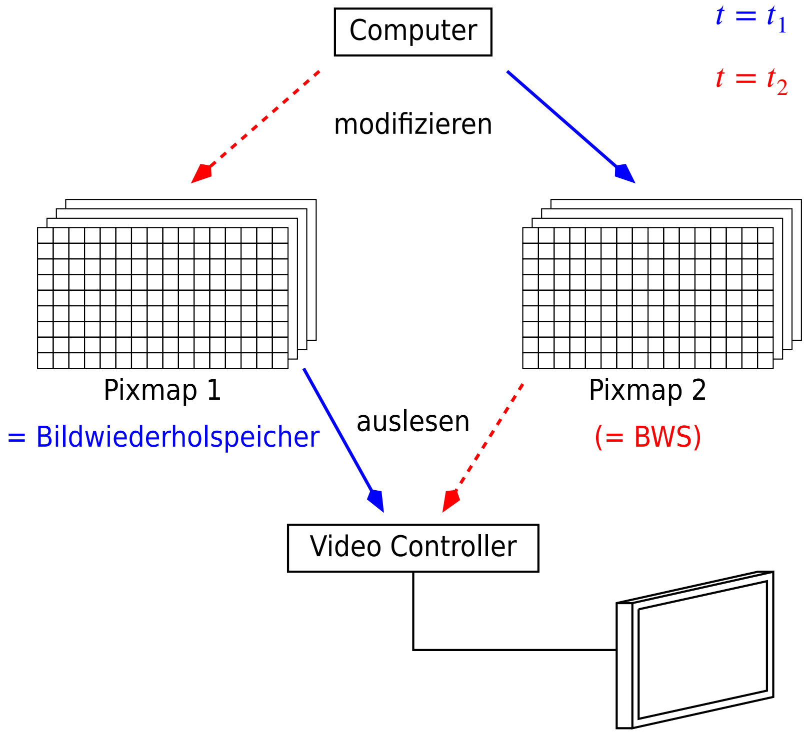
möglicher Ablauf ohne Doppelpufferung:

(Painter's Algorithm ⇝ Abschnitt 6.3.1)

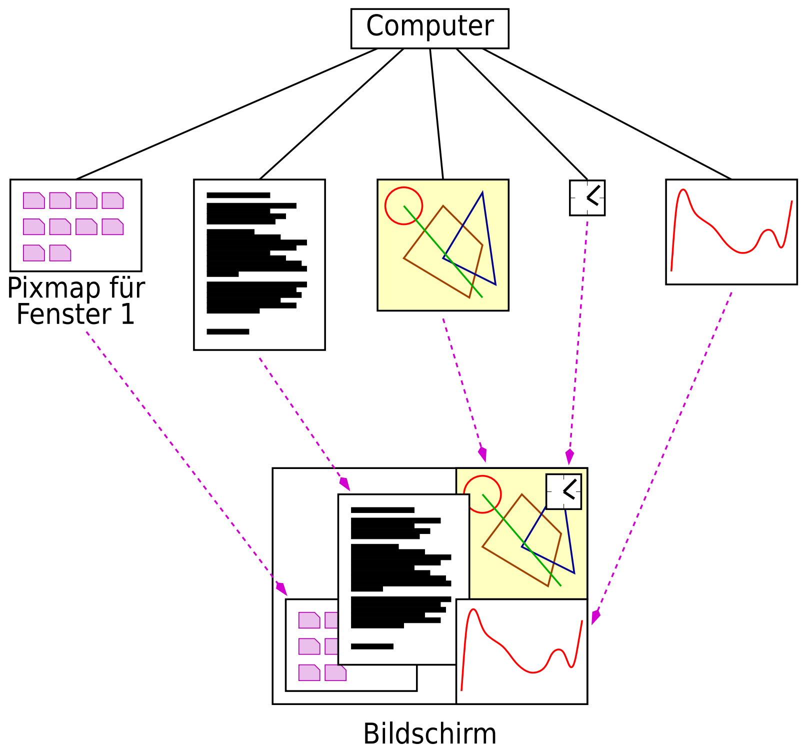


Abhilfe: Verwende zwei Pixmaps.

Es wird immer nur die Pixmap modifiziert, die gerade nicht vom Video Controller ausgelesen wird.



Anwendung 2: Verwaltung einer Fensteroberfläche



Anwendung 3: Rasterung für ein anderes Gerät

(z. B. Drucker)

Bemerkung 3.2: Für eine Pixmap der Größe $n_x \times n_y$ wird ein Speicherbereich passender Größe reserviert.

⇒ Pixel außerhalb der Pixmap dürfen nicht modifiziert werden.

⇒ Die Objekte werden an den Grenzen der Pixmap „abgeschnitten“.

⇝ „Clipping“ (Kapitel 5)

3.1 Scan Conversion für Strecken

Annahmen:

- Die Endpunkte der Strecken haben ganzzahlige Koordinaten:

$$\left(x_1, y_1\right) \in \mathbb{Z}^2, \quad \left(x_2, y_2\right) \in \mathbb{Z}^2$$

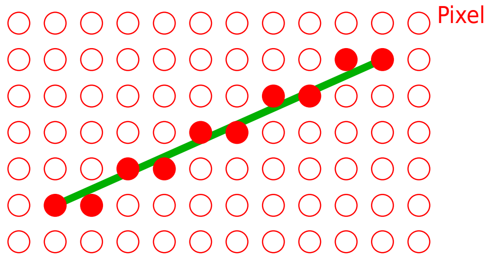
- $x_1 < x_2$
- Für die Steigung m der Strecke gilt:

$$|m| \leq 1$$

(Sonst vertauscht man die Rollen von x und y.)

Ziel: Für jeden x-Wert soll genau ein Pixel gesetzt werden, das möglichst nahe an der Strecke liegt.

Beispiel 3.3:



3.1.1 Das naive Verfahren

Algorithmus 3.4:

Algorithmus Scan Conversion für Strecke, naiv

$m \;:=\; \frac{y_2 - y_1}{x_2 - x_1}$
$b \;:=\; y_1 - m \cdot x_1$
// $y = mx + b$ ist die Steigungsform der Geraden durch (x_1, y_1) und (x_2, y_2)
für $x = x_1, x_1 + 1, \ldots, x_2$
  $y \;:=\; mx + b$
  modifiziere Pixel $(x, \mathrm{round}(y))$

Problem: viele Operationen mit reellen Zahlen sowie

Rundung reell ⇝ ganzzahlig

⇒ relativ langsam,

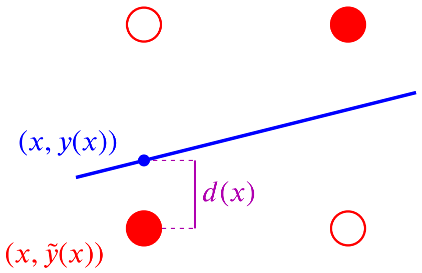
nicht für Hardware-Realisierung geeignet

Frage: Gibt es auch einen Algorithmus ohne reelle Operationen?

3.1.2 Inkrementell mit reeller Rechnung

Für jedes $x \in \{x_1, x_1+1, \ldots, x_2\}$ sei:

$y(x)$	$:= mx + b$	der zu x gehörige y-Wert auf der (exakten) Geraden
$\tilde{y}(x)$	$:= \mathrm{round}(y(x))$	der y-Wert des in Spalte x gewählten Pixels
$d(x)$	$:= y(x) - \tilde{y}(x)$	„um wie viel liegt die Gerade an der Stelle x oberhalb des Pixels?“



Idee: Berechne $y(x)$ und $\tilde{y}(x)$ nicht für jeden x-Wert gemäß der obigen Formeln, sondern bestimme, um wie viel sie sich beim Übergang von x zu $x+1$ ändern **(„Inkrement“)**:

$$y(x+1) = y(x) + \Delta y(x)$$
$$\tilde{y}(x+1) = \tilde{y}(x) + \Delta \tilde{y}(x)$$

Annahme: Im Folgenden sei $m \geq 0$ (also $m \in [0;1]$).

Es ist

$$\begin{aligned}\Delta y(x) &= y(x+1) - y(x) \\ &= m \cdot (x+1) + b - (m \cdot x + b) \\ &= m \qquad \text{(unabhängig von } x\text{)}\end{aligned}$$

und

$$\begin{aligned}y(x+1) - \tilde{y}(x) &= y(x) + m - \tilde{y}(x) \\ &= d(x) + m \\ &\in \left[-\frac{1}{2}; \frac{3}{2}\right] \qquad \text{(da } d(x) \in \left[-\frac{1}{2}; \frac{1}{2}\right] \text{ und } m \in [0;1]\text{)}.\end{aligned}$$

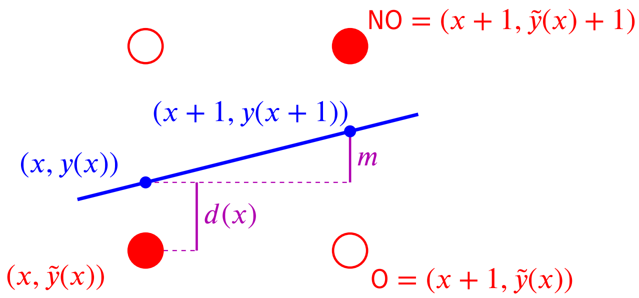
⇒ $y(x+1) \in \left[\tilde{y}(x) - \frac{1}{2}; \tilde{y}(x) + \frac{3}{2}\right]$

⇒ $\tilde{y}(x+1) = \mathrm{round}(y(x+1)) \in \{\tilde{y}(x), \tilde{y}(x)+1\}$

⇒ In Spalte $x+1$ wird eines der beiden Pixel

$$(x+1, \tilde{y}(x)) =: \textbf{O} \quad \text{(„östlicher Nachbar“)}$$
$$(x+1, \tilde{y}(x)+1) =: \textbf{NO} \quad \text{(„nordöstlicher Nachbar“)}$$

ausgewählt, d. h. $\Delta \tilde{y}(x) \in \{0, 1\}$.



Entscheidung für „O“

⇔ O liegt näher am Geradenpunkt $(x+1, y(x+1))$ als NO

⇔ $\underbrace{d(x)+m}_{=:\ \tilde{d}(x+1)} \le \frac{1}{2}$

Algorithmus 3.5:

Algorithmus **Scan Conversion für Strecke, inkrementell, reelle Rechnung**

$m \ := \frac{y_2-y_1}{x_2-x_1}$
$(x, \tilde{y}(x)) \ := \ (x_1, y_1)$
$d(x) \ := \ 0$ *// Startpixel liegt auf der Geraden*
modifiziere Pixel $(x, \tilde{y}(x))$
für $x = x_1+1, \ldots, x_2$
  $\tilde{d}(x) \ := \ d(x-1)+m$
  wenn $\tilde{d}(x) \le \frac{1}{2}$
    $\tilde{y}(x) \ := \ \tilde{y}(x-1)$ *// Entscheidung für „O“*
    $d(x) \ := \ \tilde{d}(x)$
  sonst
    $\tilde{y}(x) \ := \ \tilde{y}(x-1)+1$ *// „NO“*
    $d(x) \ := \ \tilde{d}(x)-1$
  modifiziere Pixel $(x, \tilde{y}(x))$

Bemerkungen 3.6:

1. Im Fall $m < 0$ kann auch $\widetilde{d}(x) < -\frac{1}{2}$ vorkommen. Dann ist $\Delta\tilde{y}(x) = -1$, also
$$(x+1, \tilde{y}(x)-1) =: \textbf{SO} \quad (\text{„südöstlicher Nachbar“})$$
zu wählen.
2. $y(x)$ tritt nicht mehr explizit auf.
3. keine reelle Multiplikation und keine Rundung reell ⇝ ganzzahlig mehr, aber noch reelle Addition und reeller Vergleich

3.1.3 Inkrementell mit ganzzahliger Rechnung

Idee: Die Größen m, $\widetilde{d}$, $\frac{1}{2}$ und d in Algorithmus 3.5 sind **rational**; Multiplikation mit dem **Hauptnenner**
$$H := 2 \cdot (x_2 - x_1)$$
macht sie ganzzahlig.

Algorithmus 3.7:

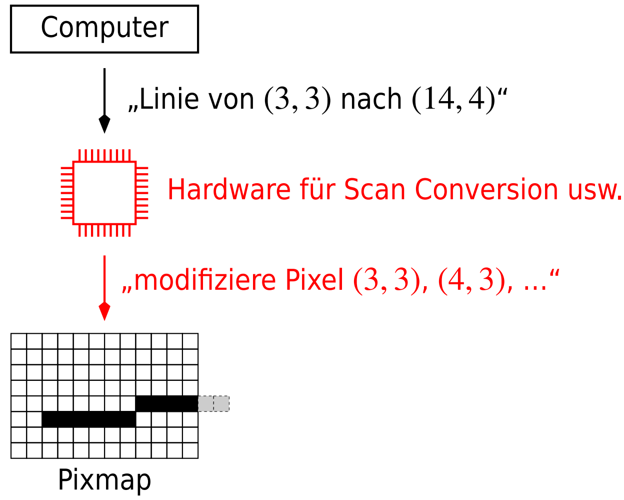
Algorithmus `Scan Conversion für Strecke, inkrementell, ganzzahlige Rechnung`
// *Erinnerung:* $m \in [0;1]$

$M := 2 \cdot (y_2 - y_1)$ // $= H \cdot m$
$Halb := x_2 - x_1$ // $= H \cdot \frac{1}{2}$
$Eins := 2 \cdot Halb$ // $= H \cdot 1$
$x := x_1$ // *Startpixel* $= (x_1, y_1)$
$\tilde{y} := y_1$
$D := 0$ // $= H \cdot d(x)$
`modifiziere Pixel` $(x, \tilde{y})$
für $x = x_1 + 1, \ldots, x_2$
  $D := D + M$ // D *enthält jetzt* $H \cdot \widetilde{d}(x)$
  wenn $D > Halb$
    $\tilde{y} := \tilde{y} + 1$ // *Entscheidung für „NO“*
    $D := D - Eins$
  `modifiziere Pixel` $(x, \tilde{y})$

Bemerkung 3.8: keine reellen Operationen mehr, nur noch ganzzahlige Additionen und Vergleiche

⇒ sehr schnell,

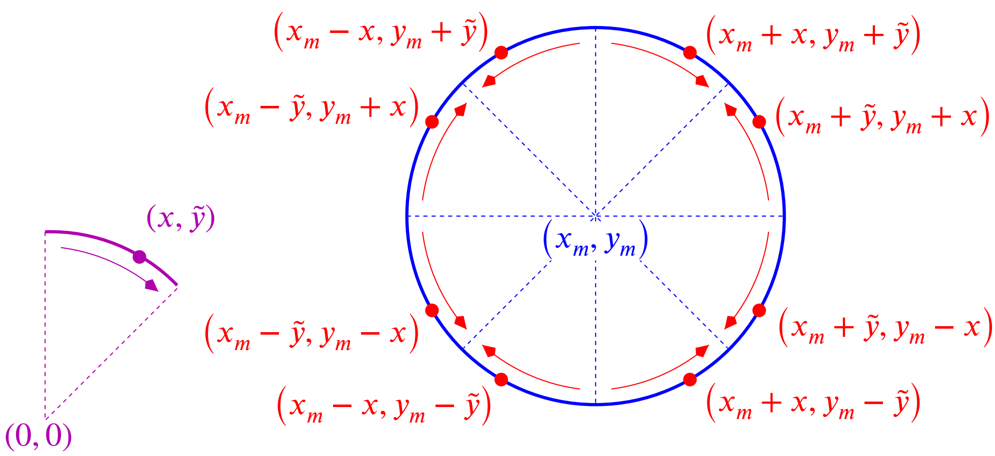
gut für Hardware-Realisierung geeignet



3.2 Scan Conversion für Kreislinien

Ziel: Modifikation der Pixel in der Pixmap, die dem Kreis um (x_m, y_m) mit Radius r am nächsten liegen

Bemerkung 3.9: Es genügt, die Pixel $(x, \tilde{y})$ im **„NNO-Achtel“** eines Kreises mit Radius r um den **Ursprung** zu erzeugen, denn



durchläuft $(x, \tilde{y})$ die Pixel des NNO-Achtels, so liefern die Pixel $(x_m \pm x, y_m \pm \tilde{y})$ und $(x_m \pm \tilde{y}, y_m \pm x)$ die gesuchte Kreislinie.

Frage: Wie erzeugt man die zum NNO-Achtel gehörigen Pixel?

3.2.1 Zwei naive Verfahren

Algorithmus 3.10:

```
Algorithmus Scan Conversion für Kreislinie, naiv via Parametrisierung

// basiert auf der Parametrisierung (x(t), y(t)) = (r · cos t, r · sin t), t ∈ [π/4; π/2], des NNO-Achtels
wähle Δt so klein, dass keine „Lücken“ entstehen
für t = π/2 mit Schrittweite −Δt bis π/4
   (x̃(t), ỹ(t))  :=  (round(r · cos t), round(r · sin t))
   modifiziere die acht zu (x̃(t), ỹ(t)) gehörigen Pixel
```

Bemerkung 3.11: reelle Rechnung, zwei Rundungen reell ⇝ ganzzahlig, **sin**, **cos**

⇒ langsam,

nicht für Hardware-Realisierung geeignet

Algorithmus 3.12:

```
Algorithmus Scan Conversion für Kreislinie, naiv via Funktionsdarstellung

// basiert auf der Darstellung y = √(r² − x²), x ∈ [−r; r], des oberen Halbkreises
x  :=  0                    // Startpunkt (0, r)
ỹ  :=  r
solange ỹ ≥ x ist          // noch im NNO-Achtel
   modifiziere die acht zu (x, ỹ) gehörigen Pixel
   x  :=  x + 1
   ỹ  :=  round(√(r² − x²))
```

Bemerkung 3.13: reelle Rechnung, Rundung reell ⇝ ganzzahlig, **Wurzel**

⇒ relativ langsam (aber schneller als Algorithmus 3.10),

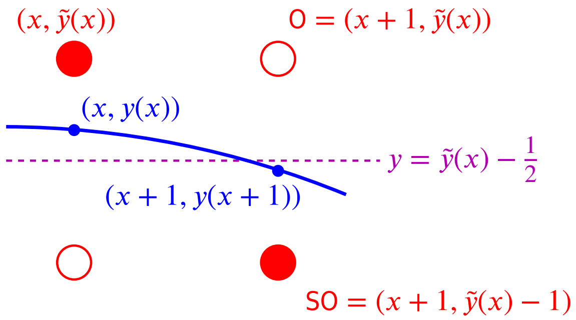
nicht für Hardware-Realisierung geeignet

Frage: Gibt es auch für Kreislinien einen inkrementellen Ansatz (wenn möglich ganzzahlig)?

3.2.2 Ein inkrementeller Ansatz

Für alle x-Werte $x = 0, 1, \ldots$ sei wieder

$(x, y(x))$ der zugehörige Punkt auf der Kreislinie und
$(x, \tilde{y}(x))$ das entsprechende Pixel.



Ausgehend von Pixel $(x, \tilde{y}(x))$ wird von den beiden Pixeln

$$(x + 1, \tilde{y}(x)) = \textbf{O} \quad \text{(östlicher Nachbar)}$$
$$(x + 1, \tilde{y}(x) - 1) = \textbf{SO} \quad \text{(südöstlicher Nachbar)}$$

das gewählt, welches näher an dem Punkt $(x + 1, y(x + 1))$ auf der Kreislinie liegt.

$(x + 1, y(x + 1))$ liegt näher bei SO als bei O

$$\Leftrightarrow y(x + 1) < \tilde{y}(x) - \frac{1}{2}$$
$$\Leftrightarrow y^2(x + 1) < \left(\tilde{y}(x) - \frac{1}{2}\right)^2 \qquad \text{(NNO-Achtel!)}$$
$$\Leftrightarrow r^2 - (x + 1)^2 < \left(\tilde{y}(x) - \frac{1}{2}\right)^2$$
$$\Leftrightarrow \underbrace{r^2 - x^2} - 2x - 1 < \tilde{y}^2(x) - \tilde{y}(x) + \frac{1}{4}$$
$$\Leftrightarrow y^2(x) - \tilde{y}^2(x) + \tilde{y}(x) - 2x - \frac{5}{4} < 0$$

Also: Setze

$$d(x) := y^2(x) - \tilde{y}^2(x) + \tilde{y}(x) - 2x - \frac{5}{4}$$

und wähle als nächstes Pixel

O, falls $d(x) \geq 0$
SO, falls $d(x) < 0$.

Frage: Kann auch d **inkrementell** berechnet werden, also $d(x + 1)$ aus $d(x)$?

Fall 1: Entscheidung für O, d. h. $\tilde{y}(x+1) = \tilde{y}(x)$

Dann ist

$$\begin{aligned}
d(x+1) &= \underbrace{y^2(x+1)} - \underbrace{\tilde{y}^2(x+1)} + \underbrace{\tilde{y}(x+1)} - 2(x+1) - \frac{5}{4} \\
&= r^2 - (x+1)^2 - \tilde{y}^2(x) + \tilde{y}(x) - 2(x+1) - \frac{5}{4} \\
&= \underbrace{r^2 - x^2} - 2x - 1 - \tilde{y}^2(x) + \tilde{y}(x) - 2x - 2 - \frac{5}{4} \\
&= \underbrace{y^2(x) - \tilde{y}^2(x) + \tilde{y}(x) - 2x - \frac{5}{4}} - 2x - 3 \\
&= d(x) - (2(x+1) + 1)\,.
\end{aligned}$$

Fall 2: Entscheidung für SO, d. h. $\tilde{y}(x+1) = \tilde{y}(x) - 1$

Dann ist

$$\begin{aligned}
d(x+1) &= \underbrace{y^2(x+1)} - \underbrace{\tilde{y}^2(x+1)} + \underbrace{\tilde{y}(x+1)} - 2(x+1) - \frac{5}{4} \\
&= r^2 - (x+1)^2 - (\tilde{y}(x) - 1)^2 + \tilde{y}(x) - 1 - 2(x+1) - \frac{5}{4} \\
&= \underbrace{r^2 - x^2} - 2x - 1 - \tilde{y}^2(x) + 2\tilde{y}(x) - 1 + \tilde{y}(x) - 1 - 2x - 2 - \frac{5}{4} \\
&= \underbrace{y^2(x) - \tilde{y}^2(x) + \tilde{y}(x) - 2x - \frac{5}{4}} + 2\tilde{y}(x) - 2x - 5 \\
&= d(x) - \left(2\left((x+1) - \underbrace{(\tilde{y}(x) - 1)}\right) + 1\right) \\
&= d(x) - (2\,((x+1) - \tilde{y}(x+1)) + 1)\ .
\end{aligned}$$

Algorithmus 3.14:

Algorithmus Scan Conversion für Kreislinie, inkrementell, reelle Rechnung

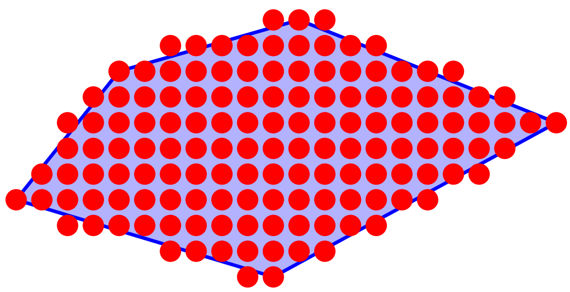
// basiert auf der Kreisgleichung $y^2 = r^2 - x^2$
$x \;:=\; 0$ // Startpunkt $(0, r)$
$\tilde{y} \;:=\; r$
$d \;:=\; r - \frac{5}{4}$ // $= d(0)$
solange $\tilde{y} \geq x$ **ist** // noch im NNO-Achtel
  modifiziere die acht zu $(x, \tilde{y})$ gehörigen Pixel
  $x \;:=\; x + 1$
  wenn $d \geq 0$
    $d \;:=\; d - (2x + 1)$ // Entscheidung für O
  sonst
    $\tilde{y} \;:=\; \tilde{y} - 1$ // Entscheidung für SO
    $d \;:=\; d - (2\,(x - \tilde{y}) + 1)$

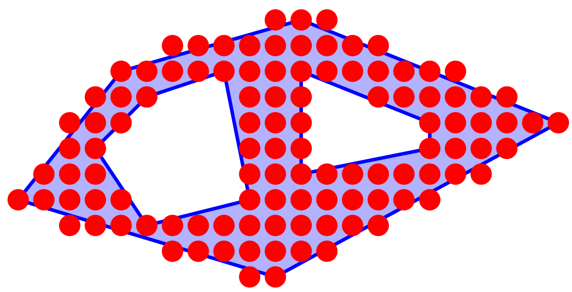
Bemerkung 3.15: Algorithmus 3.14 kann (für ganzzahlige r, x_m und y_m) in einen **inkrementellen ganzzahligen** Algorithmus überführt werden.

3.3 Scan Conversion für Polygone

gegeben: ein Polygon P in der Ebene

gesucht: die dem Polygon entsprechenden Pixel



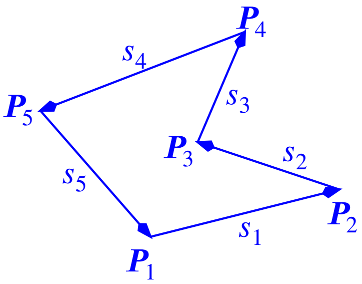


3.3.1 Polygone

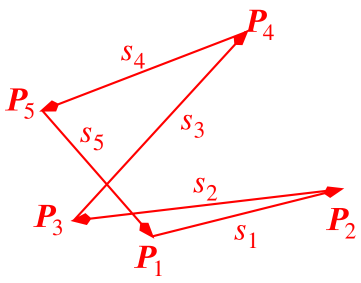
Ein **geschlossener einfacher Polygonzug** $Z = (s_1, s_2, \ldots, s_l)$ ist eine Folge von **Kanten** (gerichteten Strecken) $s_i = \overrightarrow{\boldsymbol{P}_{i-1}\boldsymbol{P}_i}$, so dass

- $\boldsymbol{P}_0 = \boldsymbol{P}_l$
- s_i hat mit s_{i+1} den Punkt $\boldsymbol{P}_i$ gemeinsam (sowie s_l mit s_1 den Punkt $\boldsymbol{P}_l$);

  weitere Schnittpunkte der Strecken gibt es nicht.



geschlossener einfacher Polygonzug



kein solcher

Ein **Polygon** ist definiert durch $m \geq 1$ geschlossene einfache Polygonzüge $Z_1, \ldots, Z_m$, die untereinander (als Punktmengen in der Ebene) disjunkt sind. Wird der Kantenzug $Z_j = \left(s_1^{(j)}, \ldots, s_{l_j}^{(j)}\right)$ in dieser Reihenfolge durchlaufen, so liegt das **Innere** des Polygons **links** jeder der gerichteten Strecken $s_i^{(j)}$.

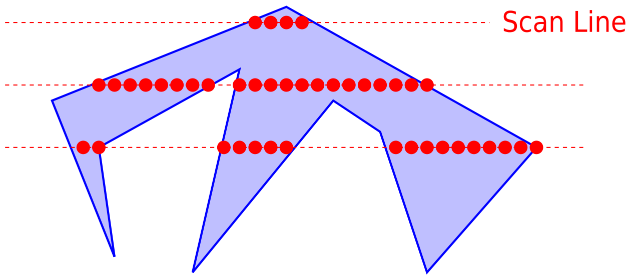
Die Punkte $\boldsymbol{P}_i^{(j)}$ heißen **Ecken** des Polygons.

Ein Polygon heißt **einfach**, wenn es durch einen einzigen Polygonzug definiert wird.

Beispiel 3.16:

3.3.2 Der Grundalgorithmus

Beobachtung: Die Pixel auf jeder horizontalen (oder vertikalen) Linie **(„Scan Line")** bilden eine oder mehrere Folgen aufeinander folgender Pixel **(„Spans")**.



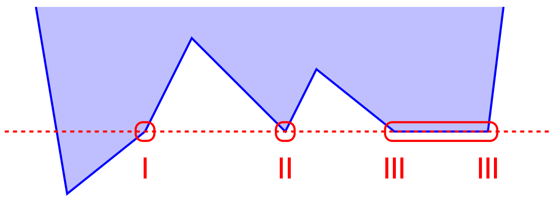
Jede solche Span wird durch zwei Schnittpunkte der Scan Line mit dem **Rand** des Polygons begrenzt.

Läuft man auf der Scan Line von links ($x = -\infty$) nach rechts ($x = +\infty$), so entspricht jeder Schnittpunkt mit einer Polygonkante einem Wechsel vom Äußeren ins Innere des Polygons und umgekehrt.

⇒ Ordnet man die Schnittpunkte $\boldsymbol{S}_j$ der Polygonkanten mit der Scan Line nach aufsteigenden x-Werten an, so folgt:

- Span 1 liegt zwischen $\boldsymbol{S}_1$ und $\boldsymbol{S}_2$.
- Span 2 liegt zwischen $\boldsymbol{S}_3$ und $\boldsymbol{S}_4$.
- ...

Bemerkung 3.17: Die Ecken des Polygons und seine horizontalen Kanten müssen gesondert behandelt werden:



I unterer Endpunkt der einen, oberer Endpunkt der anderen Kante

⇒ nur einmal zählen

II unterer (oder oberer) Endpunkt beider Kanten

⇒ zweimal (oder gar nicht) zählen

III gemeinsame Ecke einer horizontalen mit einer nicht horizontalen Kante

⇒ nur einmal zählen

Algorithmus 3.18:

Algorithmus **Scan Conversion für Polygone, Grundalgorithmus**

// bestimme den Laufbereich für die Scan Line
bestimme unter den Ecken von P die minimale und maximale y-Koordinate $y_{\min}$ und $y_{\max}$
// laufe mit der Scan Line über das Polygon
für $y = y_{\min}, \dots, y_{\max}$
  für **alle Seiten des Polygons**
    prüfe, ob die Scan Line die Seite schneidet und berechne ggf. den Schnittpunkt
  sortiere die Schnittpunkte mit der Scan Line nach aufsteigenden x-Werten
  entferne einige der doppelten Schnittpunkte gemäß der obigen Bemerkung
  // nun seien noch $S_1, \dots, S_k$ mit $x_1 \le \dots \le x_k$ übrig
  für $i = 1, \dots, \frac{k}{2}$
    modifiziere die Pixel der Span $\left(\mathrm{round}\left(x_{2i-1}\right), y\right)$, …, $\left(\mathrm{round}\left(x_{2i}\right), y\right)$

Bemerkung 3.19: Für die Modifikation der Pixel einer Span gibt es i. Allg. effizientere Methoden als Algorithmus 3.7.

Aufwand: Sei

n	die Anzahl der Ecken bzw. Kanten von P und
$l = y_{\max} - y_{\min} + 1$	die Anzahl der über das Polygon gelegten Scan Lines.

Dann beträgt der „Verwaltungsaufwand“ des Algorithmus **(ohne die eigentliche Pixelmodifikation)**:

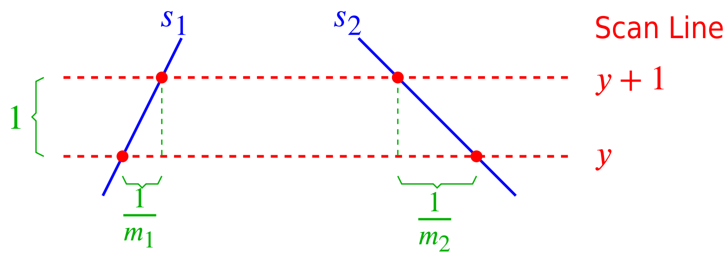
$\mathcal{O}(n)$ Operationen zur Bestimmung von $y_{\min}$, $y_{\max}$
l-mal:
  n Schnitttests bzw. Schnittpunktberechnungen
  $\mathcal{O}(n \log n)$ Operationen für das Sortieren der $\mathcal{O}(n)$ Schnittpunkte

---

$\sum$: $l \cdot n$ Schnitttests/Schnittpunktberechnungen
  $\mathcal{O}(l \cdot n \log n)$ sonstige Operationen

3.3.3 Ein inkrementeller Ansatz

Idee: nicht die einzelnen Pixel inkrementell berechnen, sondern die Ausdehnung der **Spans**



Beobachtung: Beim Übergang von einer Scan Line zur nächsten kann der **Schnittpunkt** mit jeder Polygonseite **inkrementell** berechnet werden: Ist

m die Steigung der Strecke und
$(x(y), y)$ ihr Schnittpunkt mit der Scan Line y,

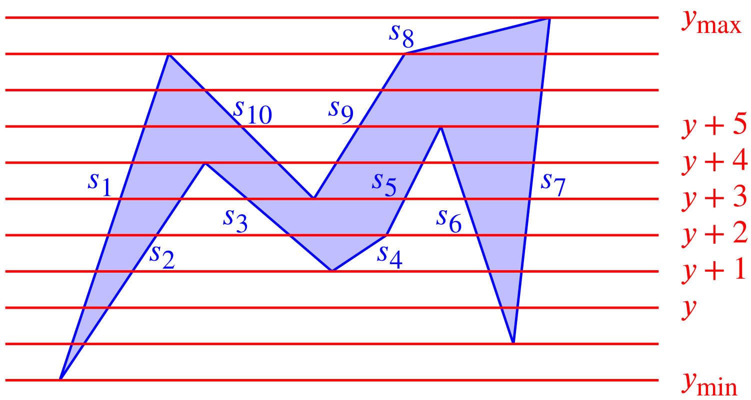
so ist

$$(x(y+1), y+1) = \left(x(y) + \frac{1}{m}, y+1\right)$$

ihr Schnittpunkt mit der nächsten Scan Line $y+1$.

Bemerkung 3.20: Dies funktioniert natürlich nur, wenn die Strecke bereits die vorige Scan Line geschnitten hat.

Strecke s heißt **aktiv** bei Scan Line y, wenn sie die Scan Line schneidet.



aktive Strecken bei Scan Line

$y+5$:	s_1			s_{10}	s_9		s_5	s_6	s_7
$y+4$:	s_1	s_2	s_3	s_{10}	s_9		s_5	s_6	s_7
$y+3$:	s_1	s_2	s_3	s_{10}	s_9		s_5	s_6	s_7
$y+2$:	s_1	s_2	s_3			s_4	s_5	s_6	s_7
$y+1$:	s_1	s_2	s_3			s_4		s_6	s_7
y:	s_1	s_2						s_6	s_7

Beobachtungen:

- Eine Strecke wird aktiv, sobald die Scan Line ihren **unteren** Endpunkt trifft;

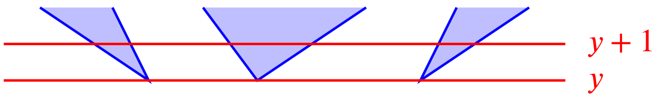
  sie wird inaktiv, wenn die Scan Line ihren **oberen** Endpunkt passiert hat.
- **Ordnet** man die aktiven Strecken bzgl. ihres Schnittpunkts mit der Scan Line, so bleibt diese Ordnung beim Übergang zur nächsten Scan Line erhalten.

Verwaltung der aktiven Strecken:

$$\mathit{Aktiv} = \left((\underbrace{i_1}_{\text{Nummer der Strecke}}, \underbrace{x_1}_{x\text{-Koordinate des Schnittpunkts der Scan Line mit Strecke } i_1}), (i_2, x_2), \ldots, (i_k, x_k) \right),$$

sortiert nach aufsteigenden x-Werten

Bemerkung 3.21: Werden gleichzeitig zwei Strecken mit gleichem unterem Endpunkt aktiv, so müssen sie nach $\Delta x := \frac{1}{m}$ sortiert in Aktiv eingefügt werden, da die Strecke mit kleinerem Δx auf der **nächsten** Scan Line zuerst kommt:

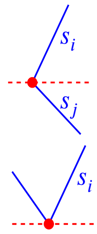


Algorithmus 3.22:

Algorithmus Scan Conversion für Polygone, inkrementell

// *Strecke s_i habe die Endpunkte $(x_{i,1}, y_{i,1})$ und $(x_{i,2}, y_{i,2})$;*
// *dabei sei o. B. d. A. $y_{i,1} \le y_{i,2}$*
// ——— *Vorverarbeitung* ———
für $i = 1, \ldots, n$
  berechne $\Delta x_i := \frac{x_{i,2} - x_{i,1}}{y_{i,2} - y_{i,1}} \left(= \frac{1}{m_i}\right)$
// *Laufbereich der Scan Line*
bestimme unter den Ecken von P die minimale und die maximale y-Koordinate $y_{\min}$ bzw. $y_{\max}$
// *bestimme zu jeder Scan Line y die Mengen*
// *$\mathit{wird\_aktiv}(y)$ bzw. $\mathit{wird\_inaktiv}(y)$ der dort aktiv bzw. inaktiv werdenden Strecken*
für $y = y_{\min}, \ldots, y_{\max}$
  $\mathit{wird\_aktiv}(y) := \emptyset$
  $\mathit{wird\_inaktiv}(y) := \emptyset$
für $i = 1, \ldots, n$
  füge i in $\mathit{wird\_aktiv}(y_{i,1})$ und $\mathit{wird\_inaktiv}(y_{i,2})$ ein

// ———— **laufe nun mit der Scan Line über** P ————
$Aktiv := \emptyset$
für $y := y_{\min}, \dots, y_{\max}$
  // **zuerst die aktiv gewordenen Strecken einordnen**
  für alle $i \in wird\_aktiv(y)$
    falls es in $wird\_inaktiv(y)$ **ein Element** j **gibt mit** $x_{i,1} = x_{j,2}$
      ersetze $(j, x_{j,2})$ **in** $Aktiv$ **durch** $(i, x_{i,1})$
      entferne j **aus** $wird\_inaktiv(y)$
    sonst
      füge $(i, x_{i,1})$ **bzgl.** x **(und ggf.** Δx**) sortiert in** $Aktiv$ **ein**



  // **die Spans zwischen den aktiven Strecken zeichnen;**
  // **sei** $Aktiv = \left(\left(i_1, x_{i_1}\right), \left(i_2, x_{i_2}\right), \dots, \left(i_k, x_{i_k}\right)\right)$
  modifiziere die Pixel der Spans $\left(\mathrm{round}\left(x_{i_{2j-1}}\right), y\right)$, …, $\left(\mathrm{round}\left(x_{i_{2j}}\right), y\right)$, $j = 1, \dots, \frac{k}{2}$
  // **entferne die inaktiv gewordenen Strecken**
  für alle $j \in wird\_inaktiv(y)$
    entferne $(j, x_{j,2})$ **aus** $Aktiv$
  // **berechne für die übrigen inkrementell den Schnittpunkt mit der nächsten Scan Line**
  für alle $(i, x_i) \in Aktiv$
    $x_i := x_i + \Delta x_i$

Aufwand: Sei wieder

n	die Anzahl der Ecken bzw. Kanten von P und
$l = y_{\max} - y_{\min} + 1$	die Anzahl der Scan Lines.

Δx_i, $y_{\min}$, $y_{\max}$ berechnen:	$\mathcal{O}(n)$
$wird\_aktiv$, $wird\_inaktiv$ belegen:	$\mathcal{O}(n + l)$
l-mal:	
$\frac{k}{2}$ Spans zeichnen:	—
n-mal:	
Eintrag in $wird\_inaktiv$ suchen, ggf. löschen:	je $\mathcal{O}(n)$
Eintrag in $Aktiv$ einfügen bzw. einen alten Eintrag ersetzen:	je $\mathcal{O}(n)$
Eintrag aus $Aktiv$ löschen:	je $\mathcal{O}(n)$
$\mathcal{O}(n \cdot l)$-mal:	
x-Wert eines $Aktiv$-Eintrags ändern:	je $\mathcal{O}(1)$

gesamter Verwaltungsaufwand: $\mathcal{O}(n \cdot (n + l))$

(**keine Schnittpunktberechnungen** mehr!)

Bemerkungen 3.23:

1. Für die Aufwandsanalyse wurde angenommen, dass $Aktiv$, $wird\_aktiv$ und $wird\_inaktiv$ mit Feldern implementiert sind.

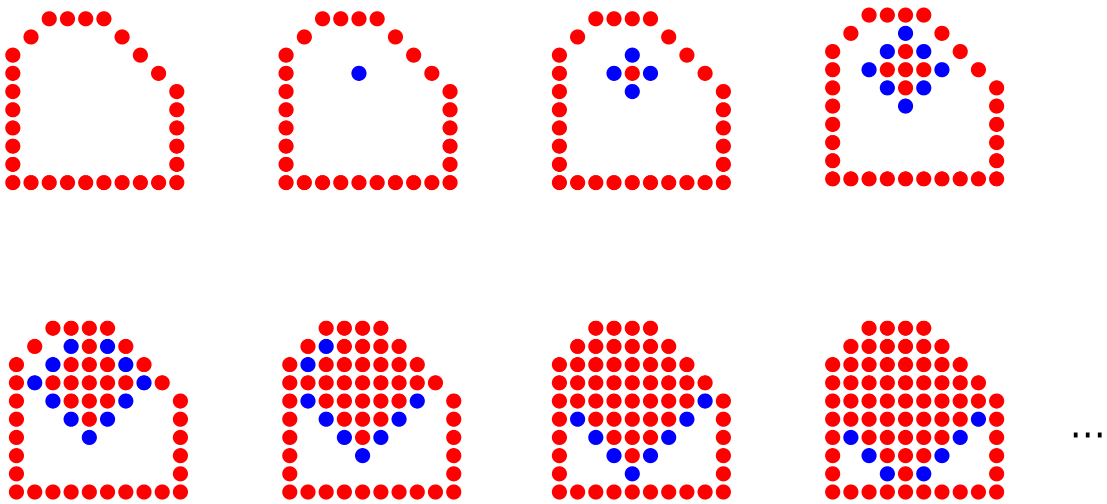
   Bei Verwendung geeigneter Datenstrukturen kann der Verwaltungsaufwand auf $\mathcal{O}(n \cdot (\log n + l))$ reduziert werden.

2. Im Algorithmus fehlt die Behandlung horizontaler Kanten.
   - einfachste Lösung: gar nicht berücksichtigen
   - Folge: „Obere“ horizontale Kanten werden nicht gemalt.

3. Für **konvexe** Polygone wird der Algorithmus wesentlich einfacher.

3.3.4 Der Flood Fill Algorithmus

Idee: Zeichne zuerst den Rand des Objekts und fülle dann das Innere aus:



Algorithmus 3.24:

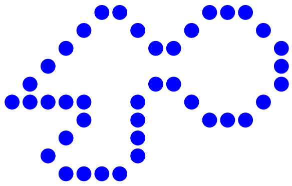
```
Algorithmus Flood Fill

zeichne den Rand des Objekts
wähle ein Pixel (x, y) im Innern des Objekts
fill(x, y)
```

Algorithmus 3.25:

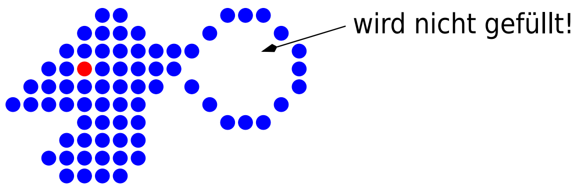
```
Algorithmus fill(x, y)
// rekursive Füll-Funktion

wenn Pixel (x, y) noch nicht den gewünschten Wert hat
   modifiziere Pixel (x, y)
   fill(x + 1, y)              // Nachbarpunkte
   fill(x, y + 1)
   fill(x − 1, y)
   fill(x, y − 1)
```


Bemerkungen 3.26:

- ⊕ schnell
- ⊕ sehr einfach, auch für Hardware-Realisierung geeignet
- ⊕ erlaubt das Füllen nahezu beliebiger Konturen



- ⊖ Objekte der gleichen Farbe können nicht „übermalt“ werden
- ⊖ Der Rand darf keine Lücken enthalten (z. B. nicht „gestrichelt“ gezeichnet).
- ⊖ arbeitet (zumindest in der angegebenen – vereinfachten – Version) nicht immer korrekt:



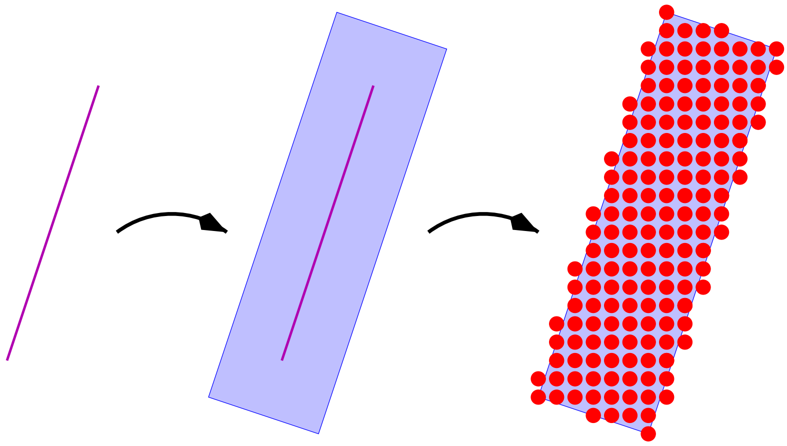
3.4 Nochmals: Strecken und Kreise

3.4.1 Linien vorgegebener „Strichbreite“

Problem: Wie zeichnet man eine Strecke/Kreislinie, die d Pixel breit ist?

Ansatz 1: Eine d Pixel breite Strecke ist ein Rechteck, also ein Polygon.

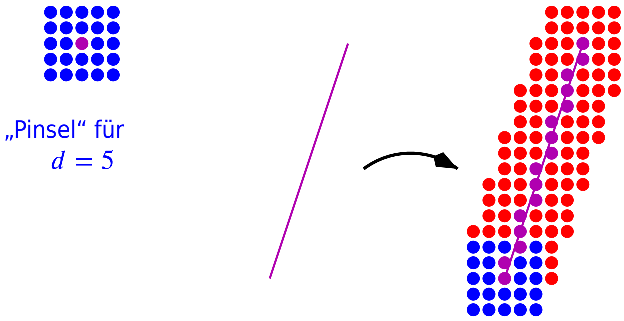
⇒ Berechne die vier Eckpunkte des Polygons und führe dann Scan Conversion für das Polygon durch.



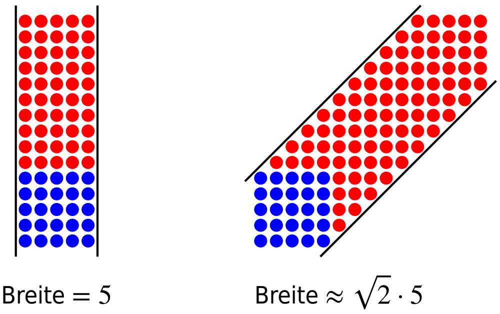
Ansatz 2: Erzeuge inkrementell die der Strecke am nächsten liegenden Pixel.

Modifiziere aber nicht nur diese, sondern einen ganzen Bereich um sie herum **(„Malen mit einem dickeren Pinsel“)**.

- quadratische Pinselform:



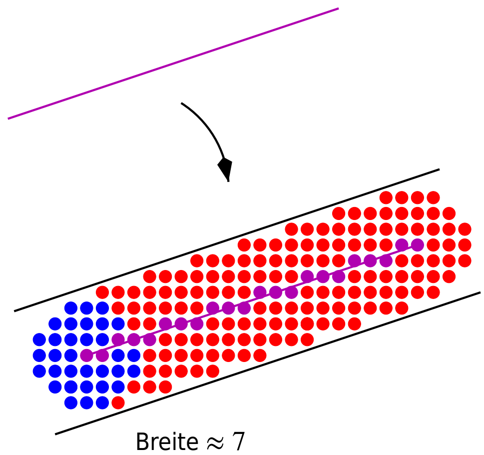
Problem: Die tatsächliche Strichbreite und die Form der Strecken-Enden hängen von der Steigung der Strecke ab:



Gegenmaßnahme: Korrektur der Pinselbreite in Abhängigkeit von der Steigung

oder

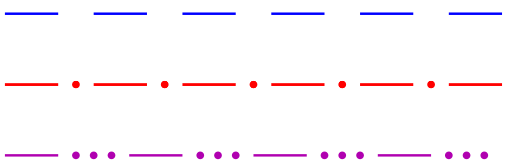
- kreisförmiger Pinsel:



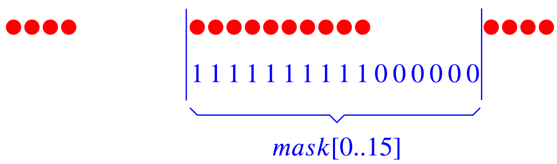
Tatsächliche Strichbreite und Form der Strecken-Enden sind (nahezu) **unabhängig von der Steigung**.

3.4.2 Unterbrochene Linien

Problem: Wie zeichnet man eine „gestrichelte“ oder „gestrichpunktete“ Linie?



Ansatz 1: Erzeuge eine **Maske** für die Periode des Musters:

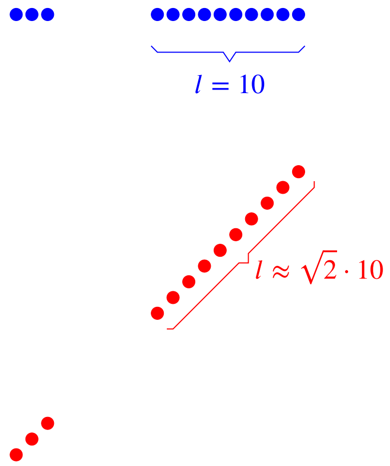


Verwende einen inkrementellen Algorithmus zur Scan Conversion, oder **zähle** die davor generierten Pixel mit (beginnend mit Nummer $i = 0$).

Modifiziere das i-te Pixel nur dann, wenn

$$mask[i \bmod \text{Maskenlänge}] = 1\,.$$

Problem: Die Länge der einzelnen Striche hängt von der Steigung der Strecke ab:



Abhilfe: steigungsabhängige Länge der Maske

oder

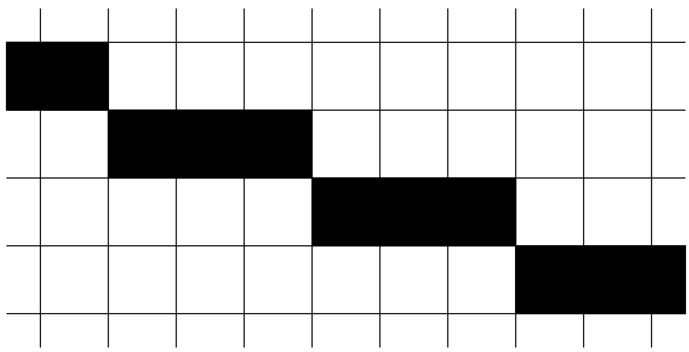
Ansatz 2: Berechne die Länge und die Lage der **einzelnen Striche** und führe für jeden einzelnen Strich die Scan Conversion durch.

Bemerkung 3.27: Beide Verfahren können auch für Strichbreite $d > 1$ verwendet werden.

3.5 Räumliche und farbliche Auflösung

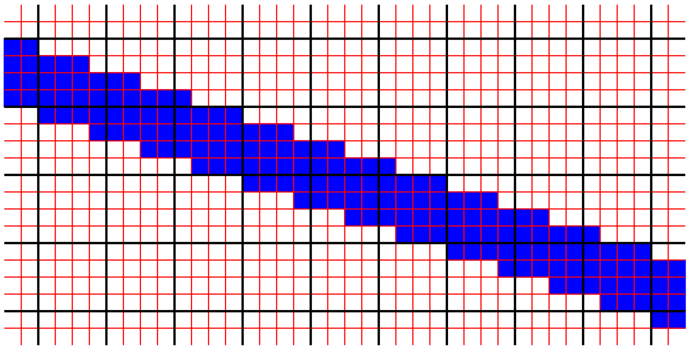
3.5.1 Räumliche Auflösung statt farblicher Auflösung

Problem: Wie reduziert man die **„Stufen"** bei (gekrümmten oder geraden) Linien?



Idee: Die Stufen wären sicher weniger auffallend, wenn die Pixmap eine höhere räumliche Auflösung (d. h. mehr Pixel pro Flächeninhalt) hätte.

⇒ Unterteile jedes vorhandene Pixel in $n_x \times n_y$ **„virtuelle Pixel"** und führe die Scan Conversion mit dieser **virtuellen Pixmap** durch.



Bestimme, welcher Anteil jedes (großen) Pixels von der Linie überdeckt wird:

$\frac{1}{16}\cdot$

11	6	1	0	0	0	0	0	0
5	10	15	11	6	1	0	0	0
0	0	0	5	10	15	11	6	1
0	0	0	0	0	0	5	10	15

Belege jedes Pixel mit einer Helligkeitsstufe, die seinem „Überdeckungsanteil“ entspricht.

Bemerkung 3.28: Dieses Verfahren gewichtet alle virtuellen Pixel innerhalb eines Pixels gleich. Meist ist eine **abstandsabhängige Gewichtung** noch besser:

$$\frac{1}{16}\cdot\begin{array}{|c|c|c|c|}\hline 1&1&1&1\\\hline 1&1&1&1\\\hline 1&1&1&1\\\hline 1&1&1&1\\\hline\end{array} \qquad \frac{1}{36}\cdot\begin{array}{|c|c|c|c|}\hline 1&2&2&1\\\hline 2&4&4&2\\\hline 2&4&4&2\\\hline 1&2&2&1\\\hline\end{array}$$

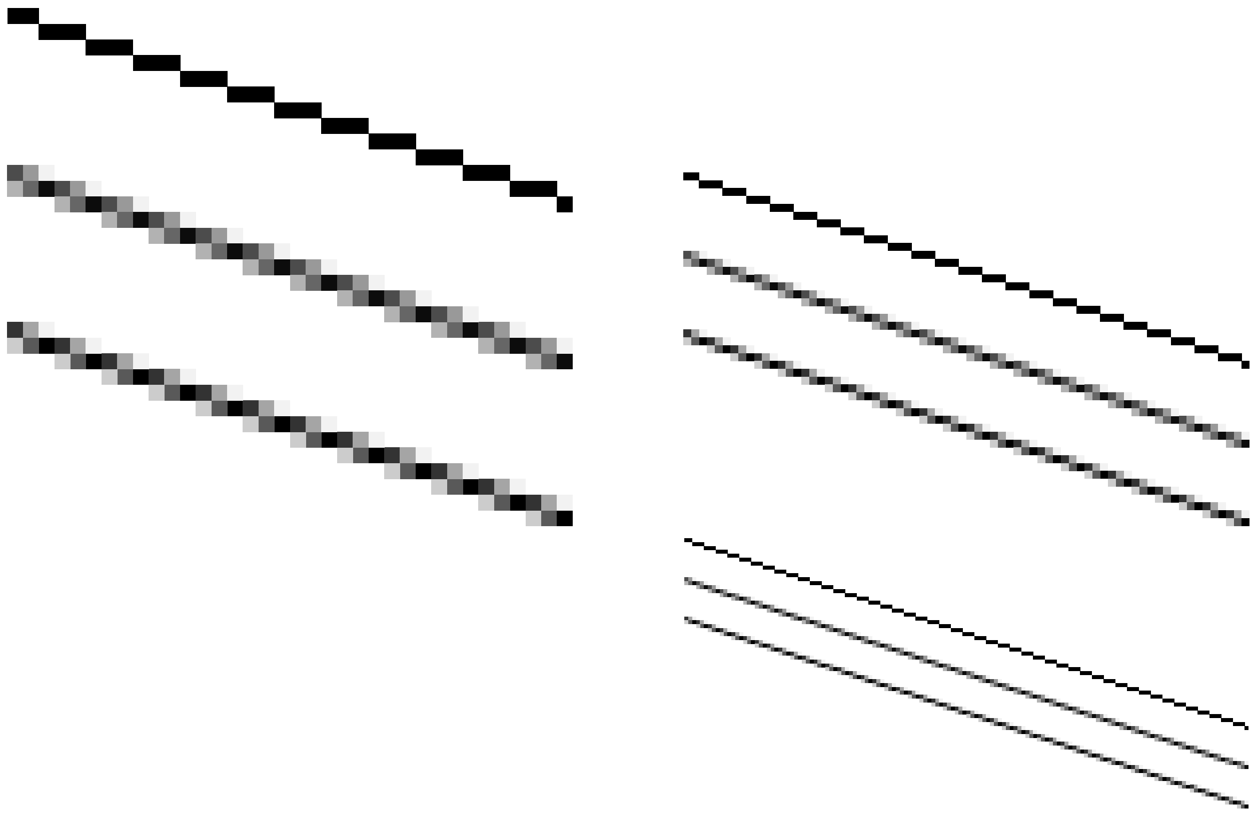
„Maske“ für konstante Gewichtung

abstandsabhängige Gewichtung

$\frac{1}{36}\cdot$

28	12	1	0	0	0	0	0	0
8	24	35	28	12	1	0	0	0
0	0	0	8	24	35	28	12	1
0	0	0	0	0	0	8	24	35

Manchmal werden auch überlappende Masken verwendet.



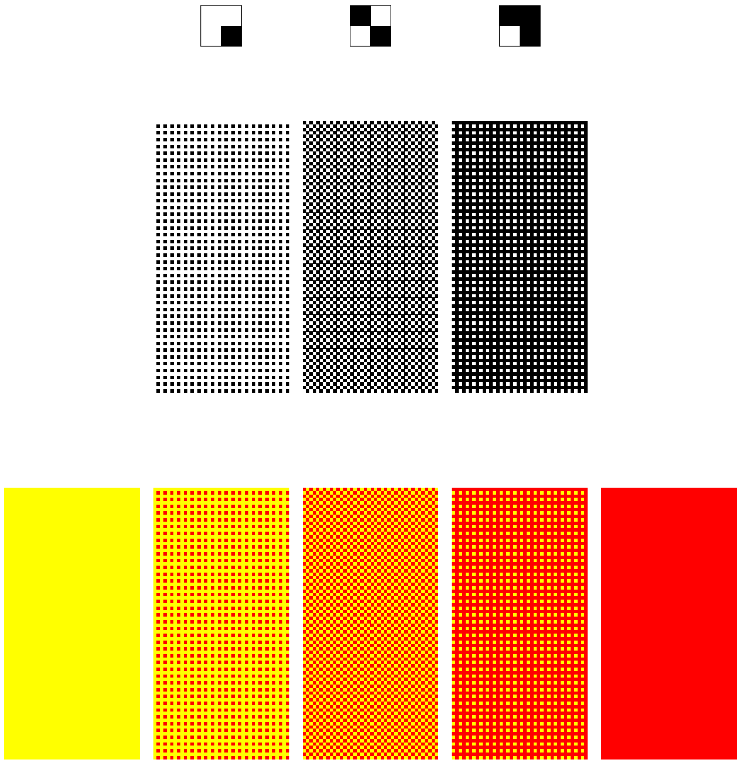
3.5.2 Farbliche Auflösung statt räumlicher Auflösung

Problem: Wie stellt man auf einem Schwarz/Weiß-Display (d. h. jedes Pixel kann nur „an“ oder „aus“ sein) **Graustufen** dar?

Wie kann man auf einem „k-Farben-Display“ (z. B. $k = 8$ mit den Farben Rot, Grün, Blau, Gelb, Zyan, Magenta, Weiß, Schwarz) wesentlich mehr als k **Farbtöne** gleichzeitig darstellen?

Idee: Graustufen lassen sich durch **„Mischen“** von Weiß und Schwarz, Farbtöne durch Mischen der vorhandenen „Grundfarben“ erzeugen.

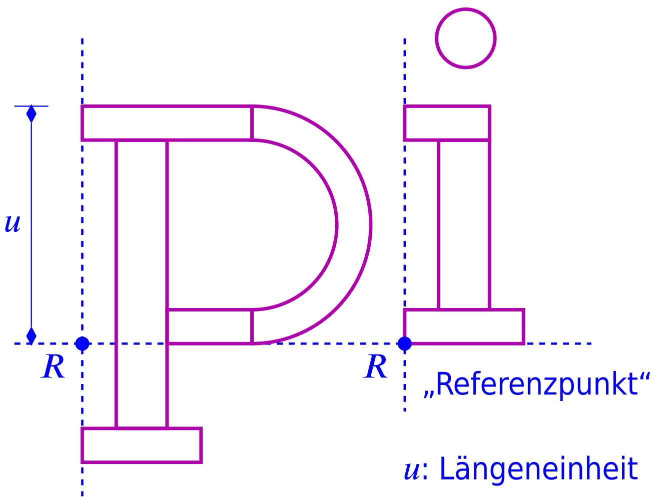
Fasse jeweils $n_x \times n_y$ Pixel zu einem **virtuellen Pixel** zusammen und belege die Pixel jedes virtuellen Pixels – dem Mischverhältnis entsprechend – mit Weiß/Schwarz bzw. mit den vorhandenen Grundfarben.



3.6 Text

Problem: Welche Pixel gehören zu einem bestimmten Zeichen eines **Zeichensatzes**?

„analytischer Ansatz“: Zerlege die Zeichen in einfachere Objekte, die gezeichnet werden können.



„p“: Linie der Breite ... von ... nach ... (relativ zu R)
Halbkreis der Breite ... mit Radius ... um ...
⋮

Übertragung des Zeichens in die Pixmap durch **Scan Conversion seiner Teilobjekte**.

⊖ relativ langsam

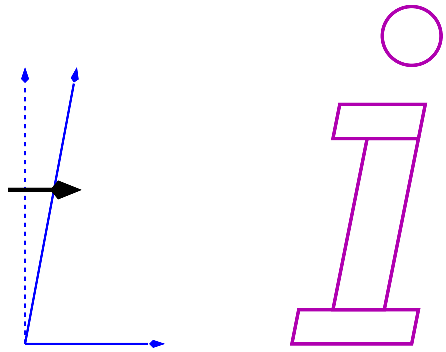
⊕ sehr flexibel bzgl. des Schriftbilds:

Zeichengröße: geeignete Wahl von u

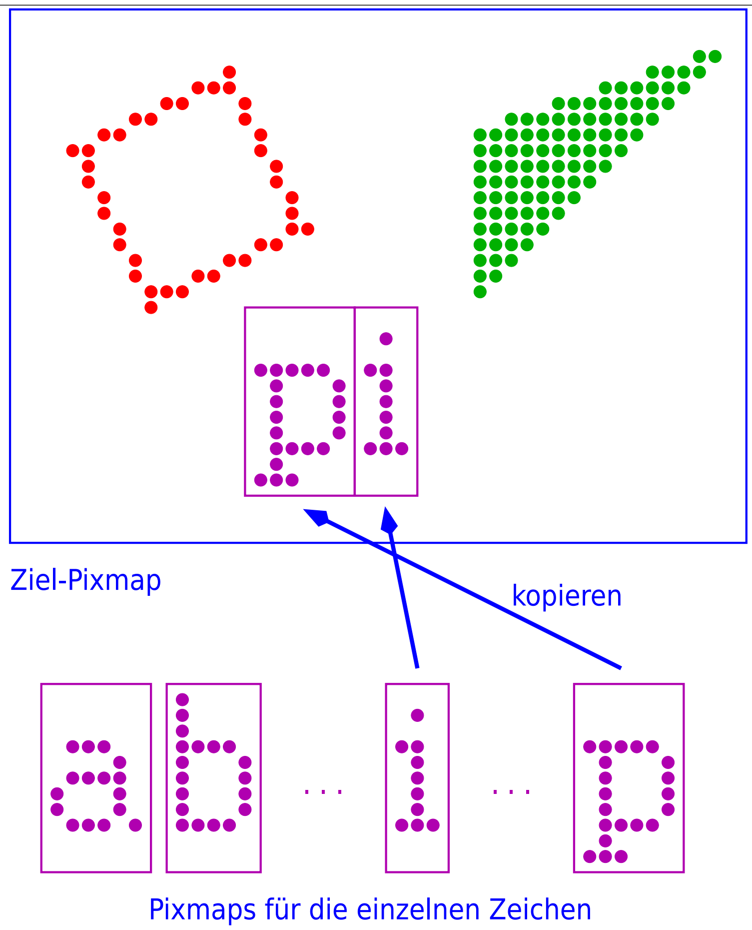
Fettdruck: Linienbreite geeignet wählen

Schrägschrift: Wende eine horizontale Scherung auf die Beschreibung des Zeichens an.

$$\begin{pmatrix} x \\ y \end{pmatrix} \rightsquigarrow \begin{pmatrix} 1 & s \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$



„Pre-converting“: Führe **vorab** für jedes Zeichen die Scan Conversion in eine eigene Pixmap durch und **kopiere** später deren Inhalt in die „Ziel-Pixmap“.



- ⊕ sehr schnell
- ⊖ benötigt viel Speicherplatz: Für jede Erscheinungsform eines Zeichens
  - Zeichengröße
  - normal/fett
  - normal/schräg

  muss eine eigene Pixmap gespeichert werden!

In der Praxis wird oft eine **gemischte Strategie** verwendet:

- Alle Zeichensätze liegen in einer analytischen Beschreibung vor;
- die am häufigsten verwendeten Zeichensätze werden zusätzlich pre-converted,
- seltener benutzte Zeichen(sätze) werden erst bei Bedarf in Pixel umgesetzt.