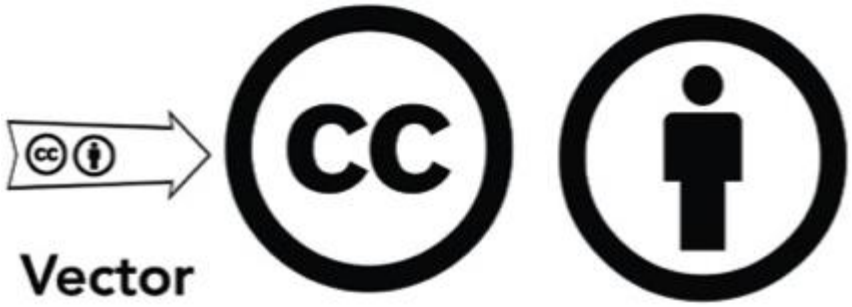
Rasterisierung

vom Polygon zum Pixel



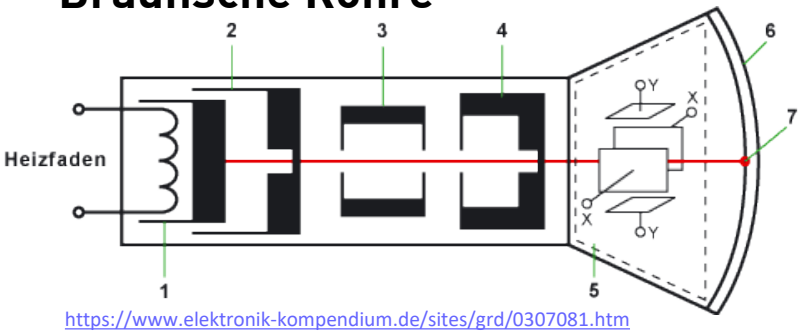
Inhalt

- Einführung
- Rasterisieren von Linien
- Füllalgorithmen
- Rasterisieren von Dreiecken
- Rasterisieren von Polygonen

EINFÜHRUNG

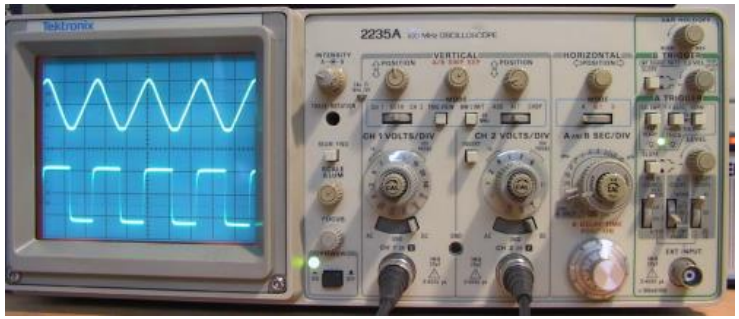
Einführung
Vektor-Displays

Braunsche Röhre



https://www.elektronik-kompendium.de/sites/grd/0307081.htm

1. Kathode
2. Wehneltzylinder
3. Elektronenoptik
4. Anode
5. Ablenkplatten
6. und 7. Leuchtschicht & Leuchtpunkt

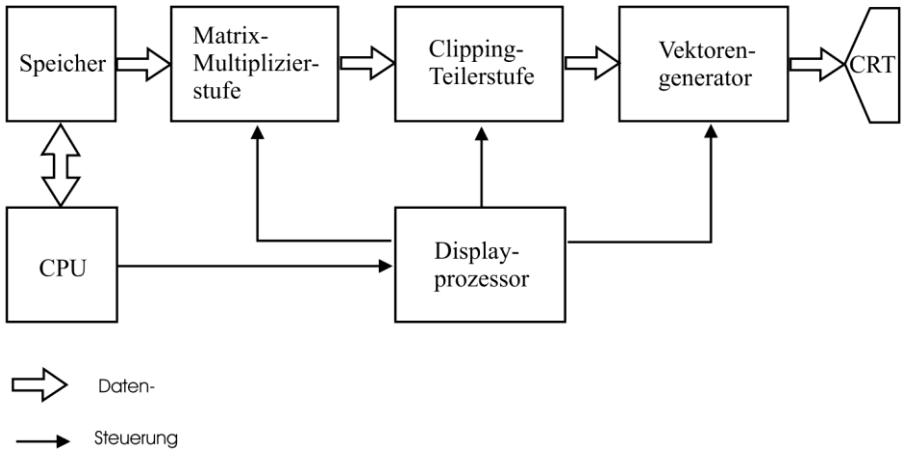


https://de.wikipedia.org/wiki/Oszilloskop#/media/Datei:Oscilloscope_sine_square.jpg

Vektordisplay

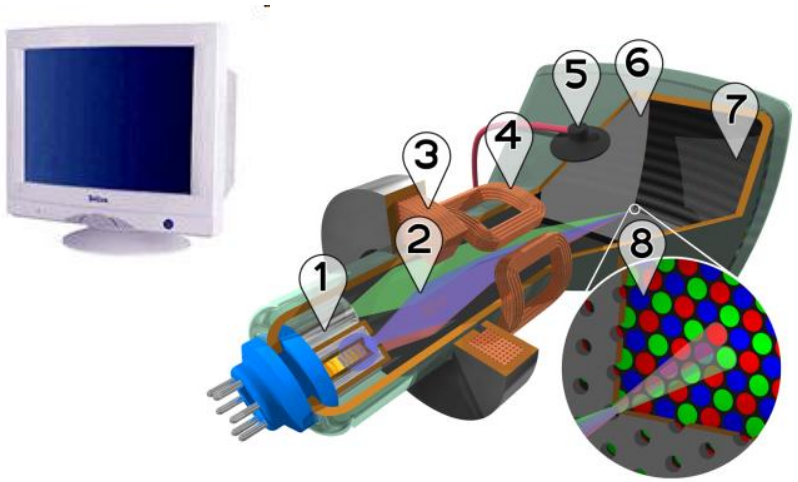
Für die Ansteuerung wurde eine vereinfachte Grafikpipeline mit Transformations- und Clipping-Stufen genutzt:

https://de.wikipedia.org/wiki/Vectrex#/media/Datei:Computerspielemuseum_13_(6702032489).jpg



Einführung
Raster-Displays

Color Cathode-Ray Tube (CRT)

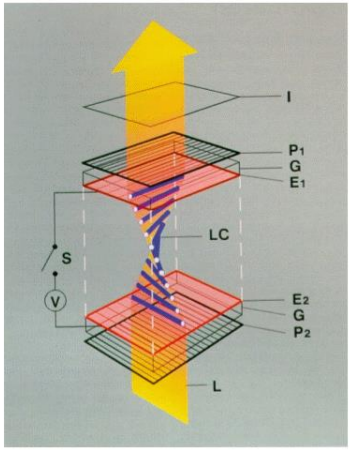


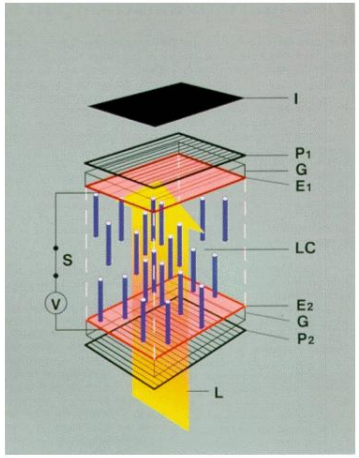
https://en.wikipedia.org/wiki/Cathode-ray_tube#/media/File:CRT_color_enhanced.png

1. Three electron emitters (for red, green, and blue phosphor dots)
2. Electron beams
3. Focusing coils
4. Deflection coils
5. Connection for final anodes
6. Mask for separating beams for red, green, and blue part of the displayed image
7. Phosphor layer (screen) with red, green, and blue zones
8. Close-up of the phosphor-coated inner side of the screen

Twisted Nematic Liquid Crystal Displays (TN-LCDs)

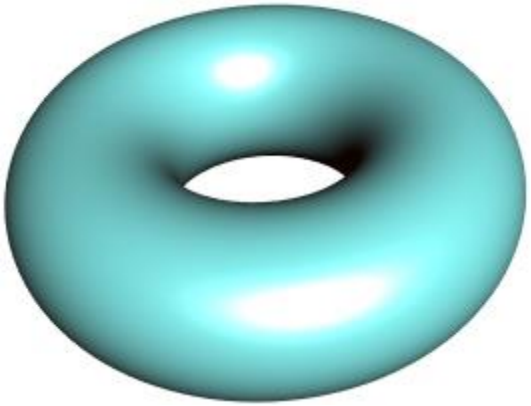
pro Pixel eine Schadt-Helfrich-Zelle:





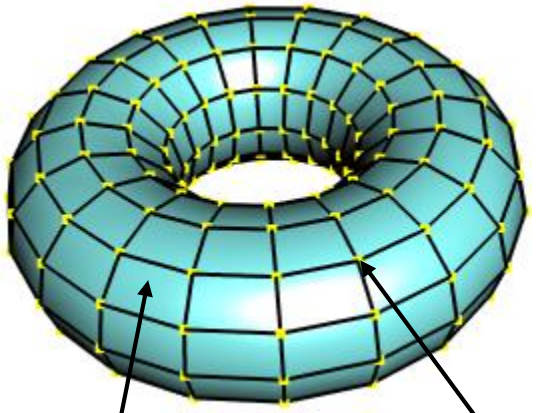
Pixelmatrix als active-matrix angesteuert, z.B. mittels Thin-Film-Transistor (TFT)

Einführung
Übersicht zur Echtzeitgraphik



Szene

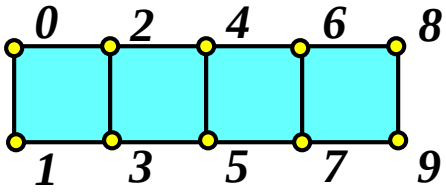
Tesselierung



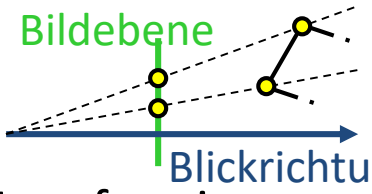
ebene Drei- oder Vierecke

Knoten mit Position, Normale, Farbe, …

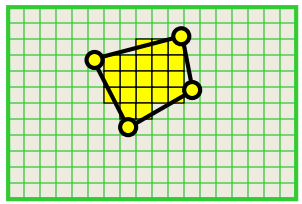
sende an GPU



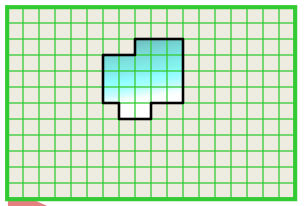
Drei- und Vierecke implizit in Knoten-reihenfolge



Transformiere Knoten in Bildkoordinaten



Zerlege Polygone in Fragmente



Berechne Fragment-farbe und Tiefe

GPU

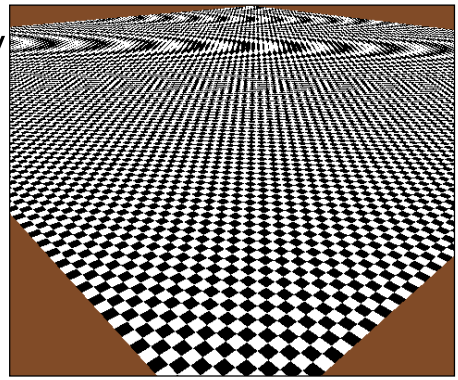
Einführung
Vergleich Vektor- vs Rasterdisplays

Vektordisplays

- ⊞ geringer Speicherbedarf für Strichzeichnungen
- ⊞ schnelles Umschalten für einfache Animation
- ⊟ Bildkomplexität begrenzt (Füllen ist z.B. aufwendig)
- Beispielgeräte
  - Vektorbildschirm
  - Plotter
  - Laser-Cutter

Rasterdisplays

- ⊞ hohe Parallelisierbarkeit
- ⊞ Bildkomplexität nur durch Auflösung beschränkt
- ⊞ unterschiedliche Displaytechnologien
- ⊟ Abtastprobleme
- Beispiele
  - Rasterdisplay
  - Drucker
  - Braille Display

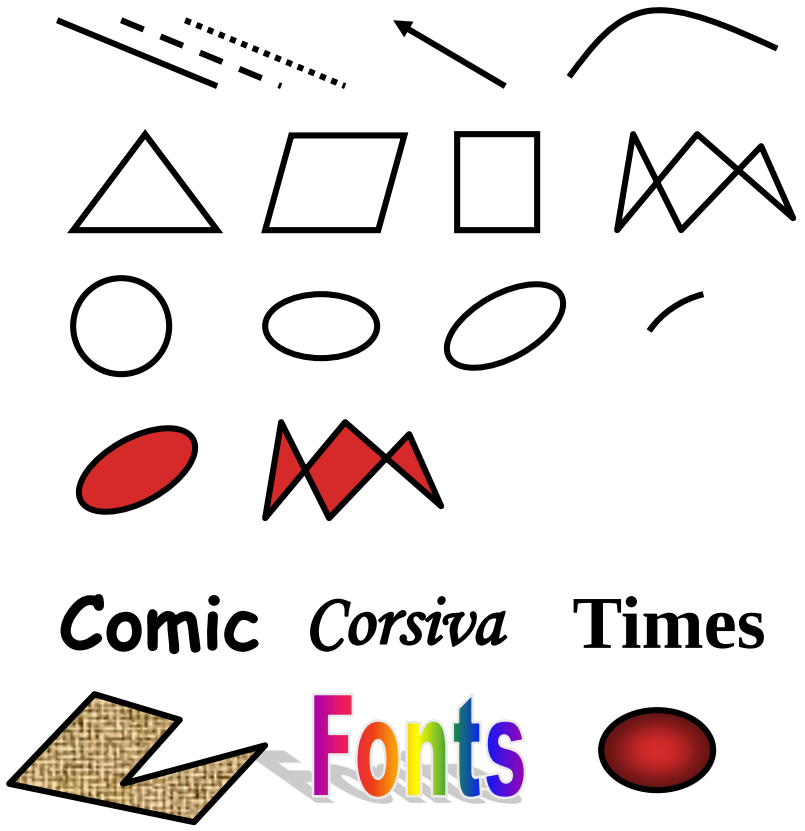


Beispiel für Aliasing

Einführung
Primitive zum Rasterisieren

- Welche Primitive werden benötigt?

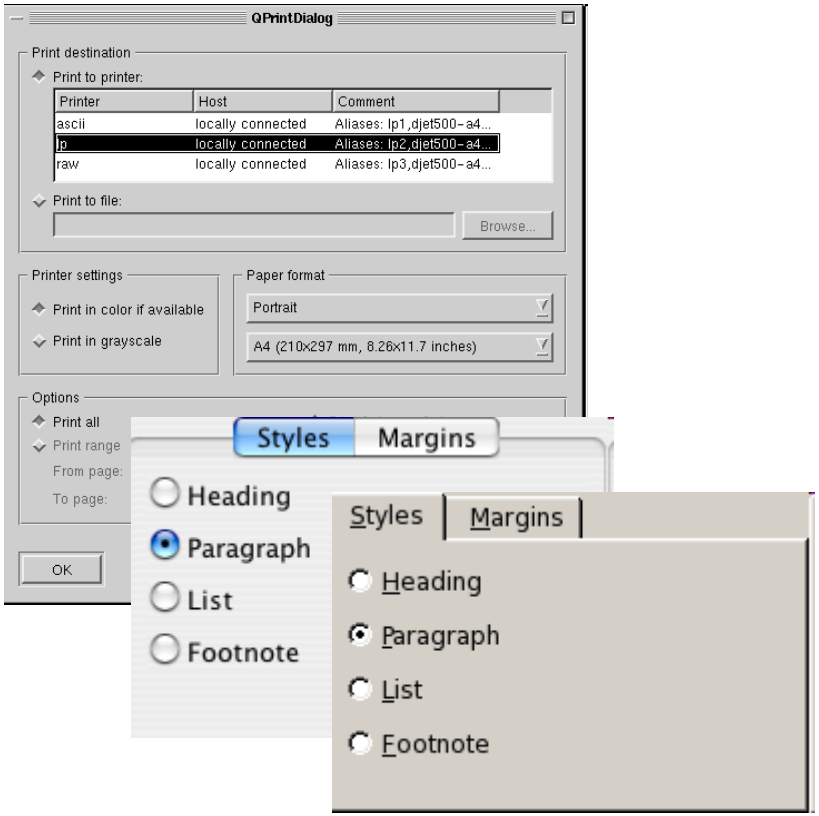
- Stilisierte Kurven …….
- Polygone ……
- Ellipsensegmente ……
- Füllalgorithmen ……
- Schriften ……...
- Verläufe, Muster …….



Plattformunabhängige Rastergraphik APIs

- Java AWT, Java 2D
- Qt, Gtk+, Fltk, … (C++)
- SVG

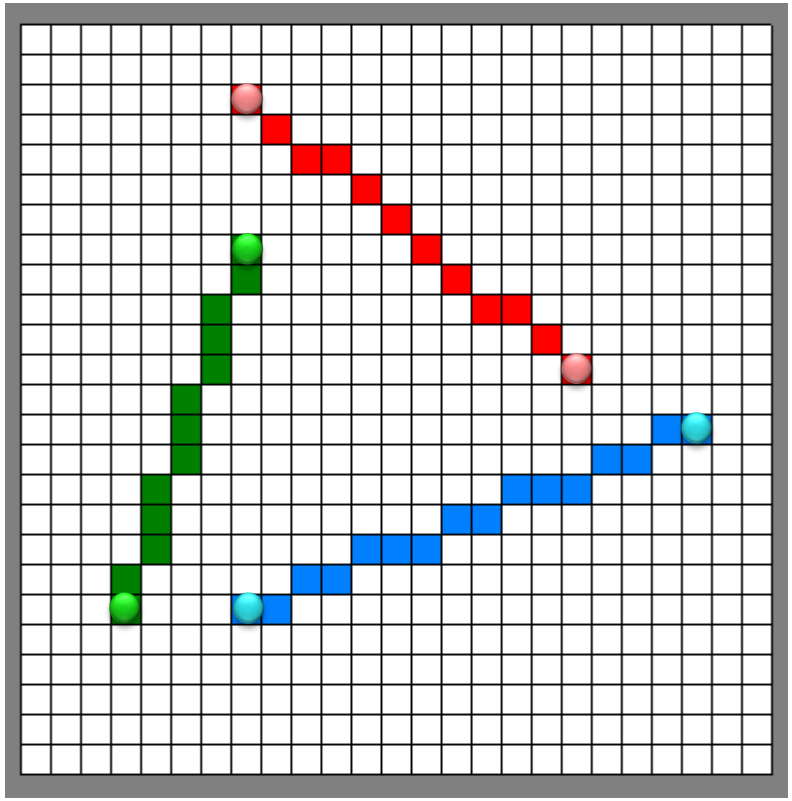
example SVG (link)



RASTERISIEREN VON LINIEN

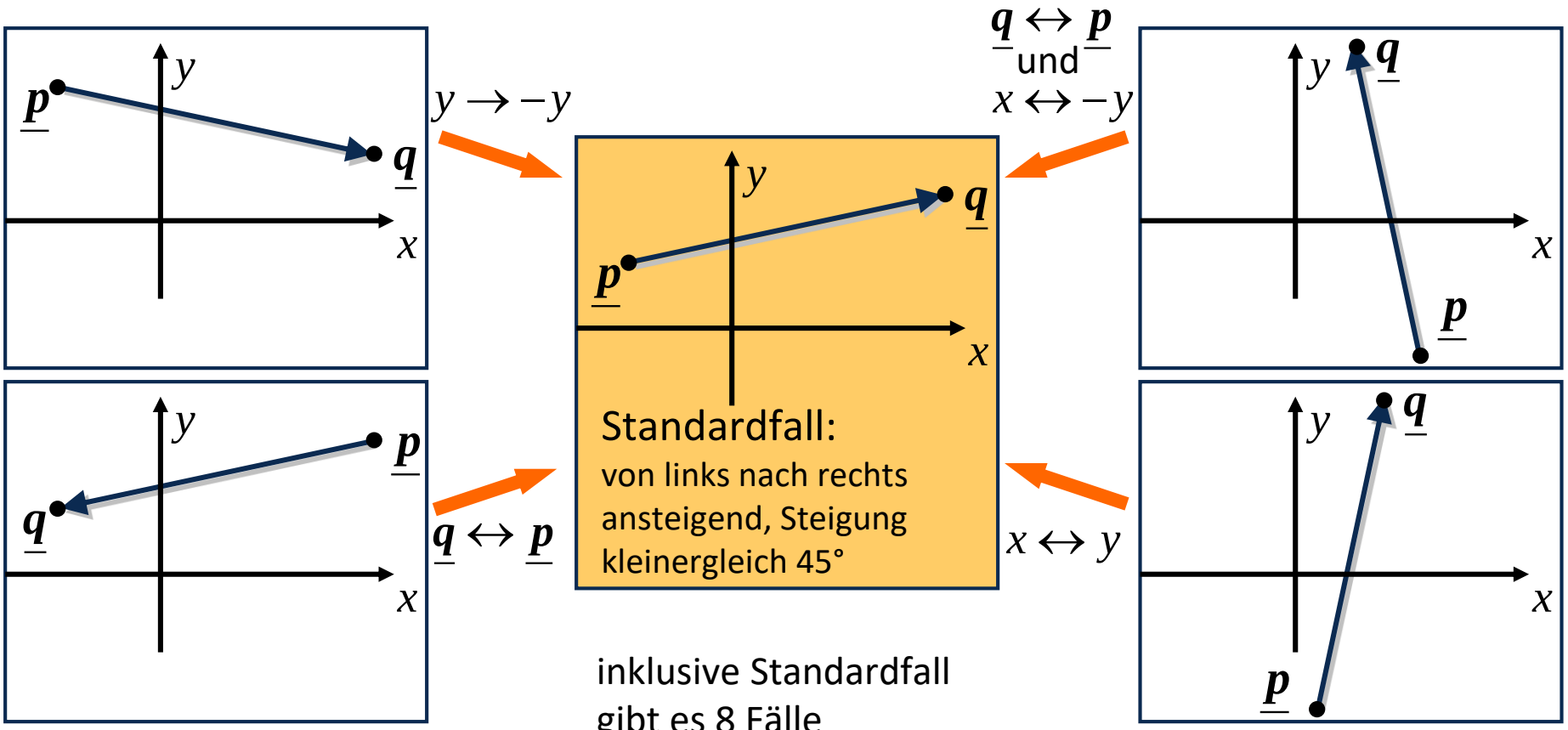
Rasterisieren von Linien
Übersicht

- gewünschte Eigenschaften für einen Linienrasteralgorithmus:
  - keine doppelte Dicke
  - einfache Implementierung
  - ganzzahlige Arithmetik
  - inkrementell
- Ansätze
  - DDA
  - Bresenham (1965)
  - Mittelpunkt (1967)



Rückführung auf einen Standardfall

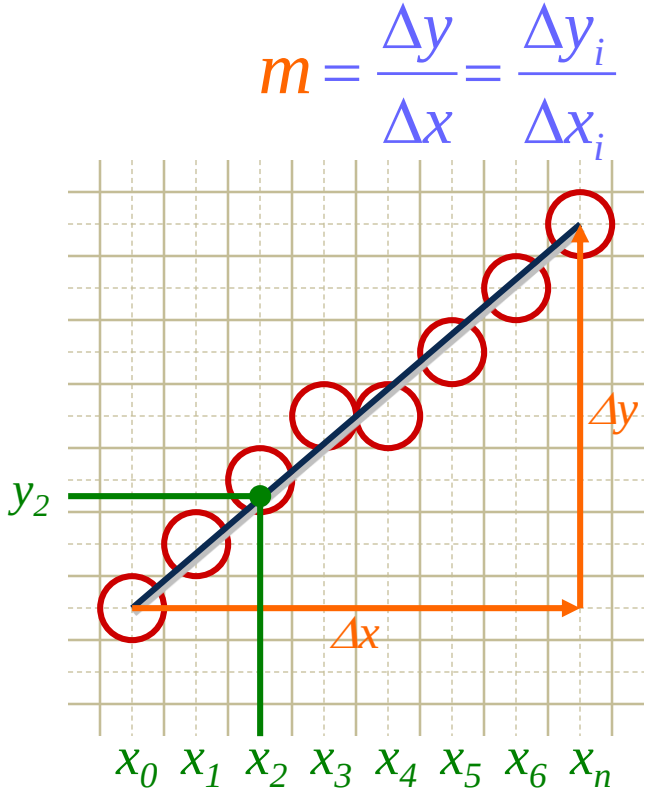
- vor dem Rastern transformieren wir die Eingabe auf den Standardfall:



Rasterisieren von Linien
Digitial Differential Analyzer (DDA)

- Idee: Steigung m ist konstant
- es folgt: $y_i = y_0 + m(x_i - x_0)$
- Pseudo-Code (Inkrementell)

```
y = y0
m = (yn-y0)/(xn-x0)
for x from x0 to xn do
    y = y0+m*(x-x0)
    setPixel(x,round(y))
    y += m
```



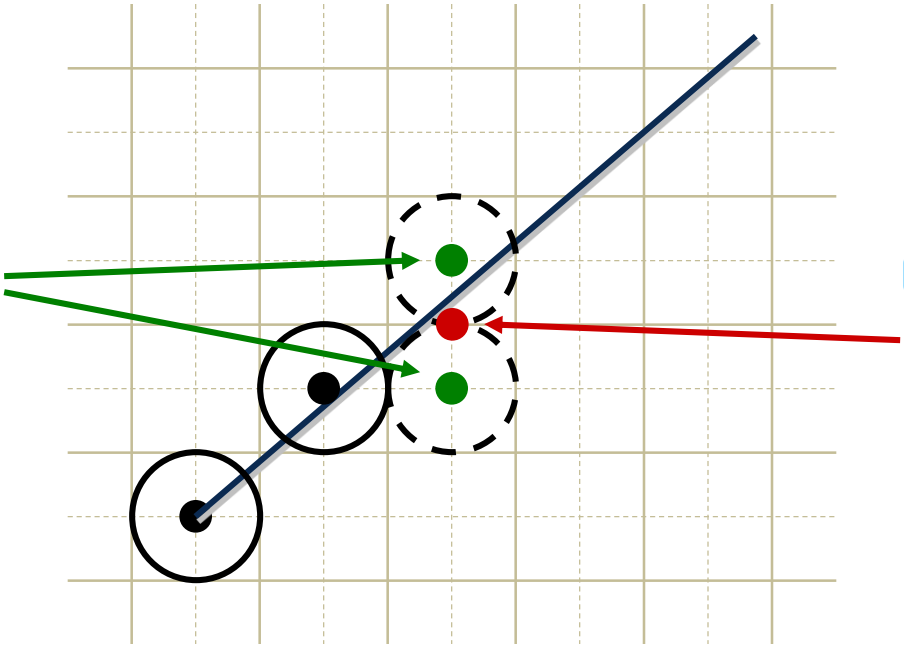
Computergraphik und Visualisierung

Bresenham / Mittelpunktsalgorithmus

- Idee: nutze Abstand zur Linie aus impliziter Darstellung als Entscheidungsvariable in inkrementellem Algorithmus

Bresenham

- Welcher Mittelpunkt ist näher ?



Mittelpunkts-algorithmus

- Auf welcher Seite liegt der Mittelpunkt ?

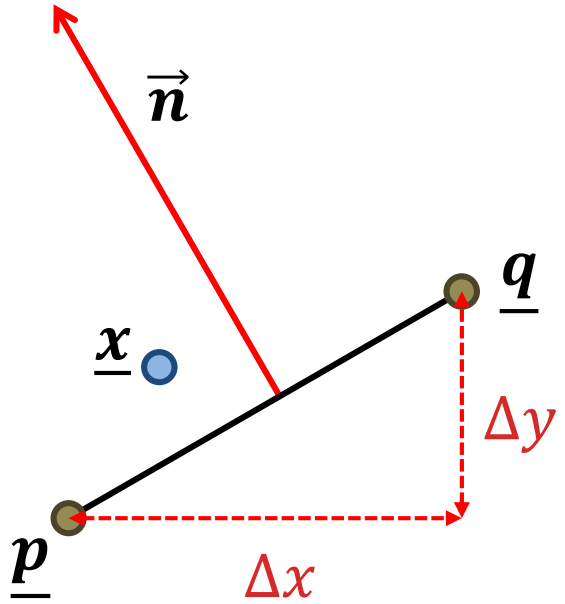
Rasterisieren von Linien
Abstandsberechnung

Abstand von Punkt zu Linie

- Für die Berechnung des Abstandes zu einer Linie wird der Normalenvektor benötigt.
- Dazu dreht man den Vektor von $\underline{\boldsymbol{p}}$ nach $\underline{\boldsymbol{q}}$ um 90 Grad nach.
- Der Abstand von $\underline{\boldsymbol{x}}$ zur Linie ist proportional zu

$$\mathrm{dist}(\underline{\boldsymbol{x}}) \propto \left\langle \underline{\boldsymbol{x}} - \underline{\boldsymbol{p}}, \vec{\boldsymbol{n}} \right\rangle$$

- Den euklidischen Abstand bekommt man, wenn man noch durch $\|\vec{\boldsymbol{n}}\|$ teilt.
- Für die Entscheidung welcher Punkt näher liegt, ist die Normierung nicht notwendig



$$\underline{\boldsymbol{q}} - \underline{\boldsymbol{p}} = \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix}$$

$$\vec{\boldsymbol{n}} = \begin{pmatrix} -\Delta y \\ \Delta x \end{pmatrix}$$

Rasterisieren von Linien
Inkrementeller Abstand

- Für einen inkrementellen Linienalgorithmus, kann der Abstand d_i zur Linie mitgeführt werden.
- Am Anfangspunkt (und Endpunkt) gilt:
  $d_0 = d_n = 0$
- Danach wandert man entweder nach rechts:

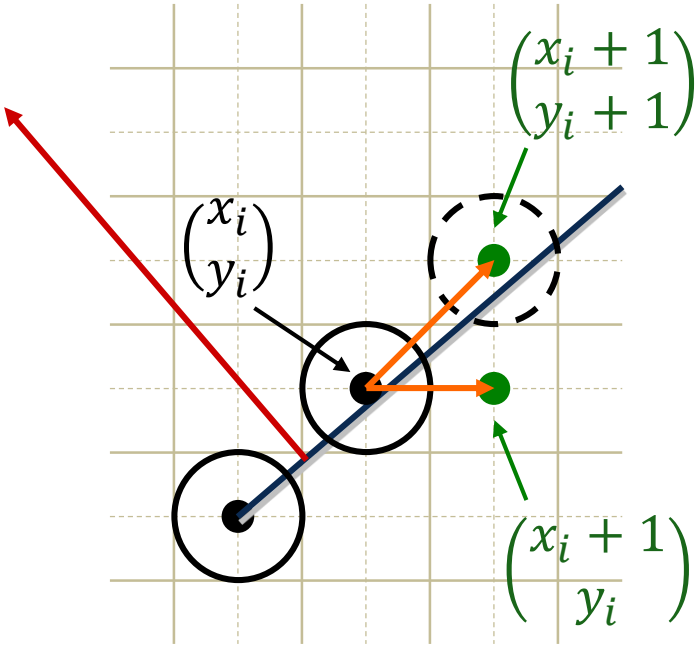
$$d_{i+1}^{\text{re}} = \left\langle \underline{\boldsymbol{x}}_i + \begin{pmatrix} 1 \\ 0 \end{pmatrix} - \underline{\boldsymbol{p}}, \vec{\boldsymbol{n}} \right\rangle = d_i - \Delta y$$

- oder nach rechts oben

$$d_{i+1}^{\text{re-ob}} = \left\langle \underline{\boldsymbol{x}}_i + \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \underline{\boldsymbol{p}}, \vec{\boldsymbol{n}} \right\rangle = d_i - \Delta y + \Delta x$$

$$\text{dist}(\underline{\boldsymbol{x}}) \propto \left\langle \underline{\boldsymbol{x}} - \underline{\boldsymbol{p}}, \vec{\boldsymbol{n}} \right\rangle$$

$$\vec{\boldsymbol{n}} = \begin{pmatrix} -\Delta y \\ \Delta x \end{pmatrix}$$



Rasterisieren von Linien
Entscheidungskriterium

- Wenn man immer richtig gegangen ist, gilt

$$d_{i+1}^{\text{re}} \leq 0 < d_{i+1}^{\text{re-ob}}$$

- Um zu entscheiden, ob man als nächstes nach rechts und nicht nach rechts-oben gehen soll, prüft man ob

$$-d_{i+1}^{\text{re}} \leq d_{i+1}^{\text{re-ob}}$$

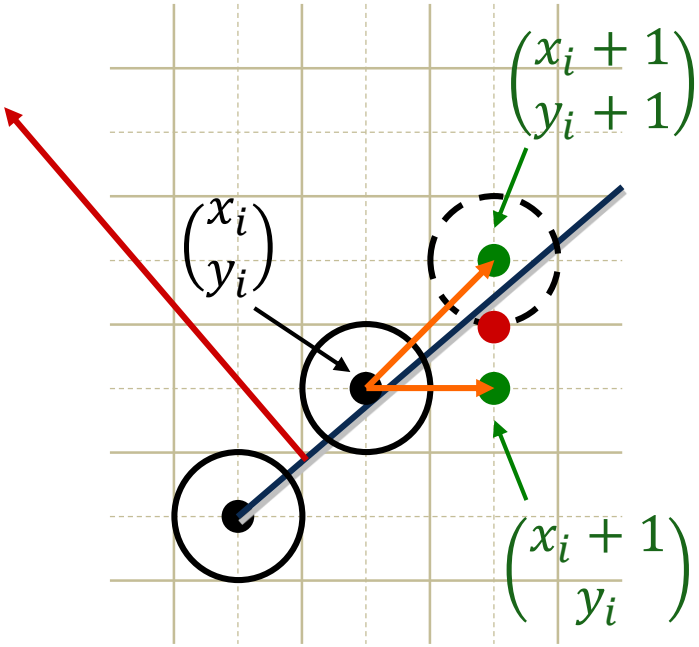
$$\Leftrightarrow -(d_i - \Delta y) \leq d_i - \Delta y + \Delta x$$

$$\Leftrightarrow 2\Delta y - \Delta x \leq 2d_i$$

- Denselben Test erhält man, wenn man prüft, ob der Abstand des Mittelpunkts kleiner gleich 0 ist

$$d_{i+1}^{\text{re}} = d_i - \Delta y$$

$$d_{i+1}^{\text{re-ob}} = d_i - \Delta y + \Delta x$$



Rechenbeispiel

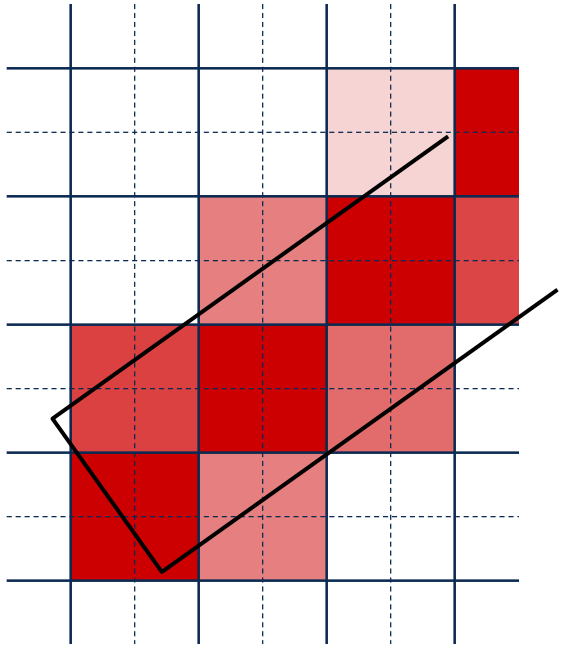
Inkrementelle Rasterisierung von Linien

```
[y, d]          = [ y0, 0 ]
[Δx,Δy]         = [(xn-x0), (yn-y0)]
[Δd^re,Δd^re-ob] = [-Δy, Δx-Δy]
c               = 2*Δy-Δx
for x from x0 to xn do
   setPixel(x, y)
   if c <= 2*d then
      d += Δd^re
   else
      y += 1; d += Δd^re-ob
```

Rasterisieren von Linien
Antialiasing

Treppenstufenartefakte

- Treppenstufenartefakte entsteht durch Unterabtastung auf dem Pixelgitter
- Die Vermeidung von Treppenstufen nennt man Antialiasing von Linien
- Gemeinsam sind diesen Techniken die Idee, Linie als Balken zu interpretieren
  - Darstellung einer dickeren Linie in höherer Auflösung mit anschließender Filterung
  - Filterung während des Zeichnens
    - erfordert Transparenz
    - kann zu Fehlern führen wenn sich mehrere Linien kreuzen



Rasterisieren von Linien
Diskussion

- gekrümmte Linien
  - inkrementelle Algorithmen bei impliziter Definition (Kreis siehe Übung)
  - für Schriften werden NURBS eingesetzt. Diese können adaptiv in Polygone unterteilt werden.
- dicke Linien
  - wiederholen einer Maske / Brush
    - ineffizient aber einfach
  - Randberechnung und Füllalgorithmus
    - wird oft mit Hilfe einer impliziten Darstellung gelöst

FÜLLALGORITHMEN

Füllalgorithmen
Übersicht

- Was muss gefüllt werden?
  - Formen (typischerweise definiert als Vektorgraphik)
    - Dreiecke
    - Polygone
    - Kreise
  - Raster-Bildbereiche gleicher oder ähnlicher Farbe
- Womit soll gefüllt werden?
  - Farbe
  - Muster
  - Farbverlauf
  - Struktur

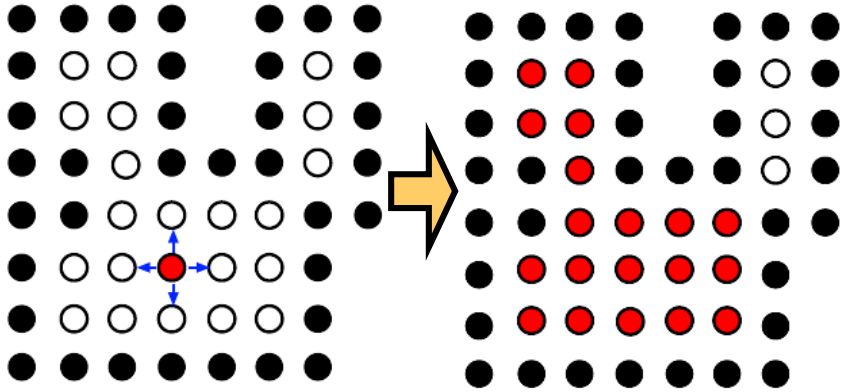
Füllalgorithmen
Auf beliebigen Bildern

- Idee Füll-Tool: überschreibe die Farbe aller Pixel um einen *Saatpixel*, die eine *ähnliche* Farbe haben
- Wenn einfache Ähnlichkeitsmaß nicht ausreicht muss ein Segmentierungsalgorithmus verwendet werden (siehe Bildverarbeitung / Computer Vision)

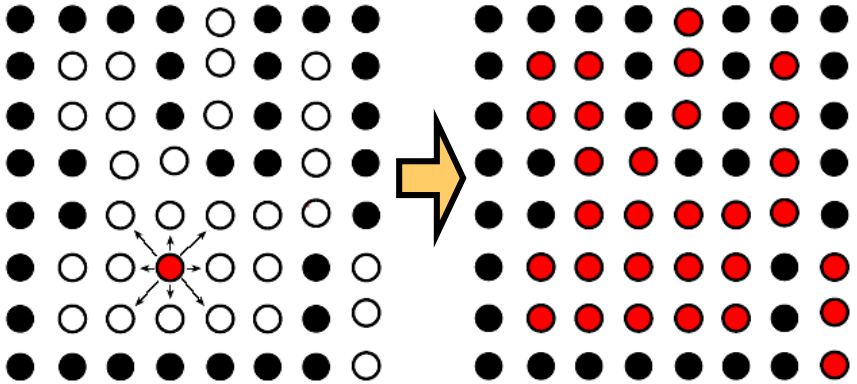
Bei natürlichen Bildern reicht ein einfaches Ähnlichkeitsmaß nicht aus auch wenn man den Füllalgorithmus wie rechts wiederholt startet

Füllalgorithmen
Nachbarschaften

- Welches sind die benachbarten Pixel?



4-er Nachbarschaft kann zu unvollständigem Füllen führen



8-er Nachbarschaft kann zu Auslaufen führen

Flood-Fill

- Pseudo-Code bei 4-er Nachbarschaft

```
void floodFill(x, y)
    if not toBeFilled(x, y) return
    setPixel(x,y,drawColor)
    floodFill(x-1,y)
    floodFill(x+1,y)
    floodFill(x,y-1)
    floodFill(x,y+1)
```

Füllalgorithmen
Rekursionsabbruch & Diskussion

- **`toBeFilled(x,y)`** prüft, ob
  - Pixelposition gültig ist,
  - FloodFill den Pixel nicht zuvor überschrieben hat,
  - Pixel-Farbe der Farbe des Saat-Pixel ähnelt
- ⊞ Vorteile:
  - ⊞ einfache Implementierung
  - ⊞ sehr allgemein
- ⊟ Nachteile:
  - ⊟ hohe Rekursionstiefe führt zu Stapelüberlauf
  - ⊟ Füllergebnis ist abhängig von Definition der Pixelnachbarschaft

Einschub
Graphisches Debuggen

- Typisch für die Entwicklung Graphischer Anwendung:
  - eine visuelle Darstellung kann extrem viel schneller verstanden werden als die intern gehaltenen Zahlenkolonnen
  - Fehler treten oft erst nach sehr vielen Iterationen auf
  - der Standarddebugger unterstützt keine Visualisierung während des Debuggens
- Deshalb müssen meist eigene Debug-Mechanismen geschaffen werden

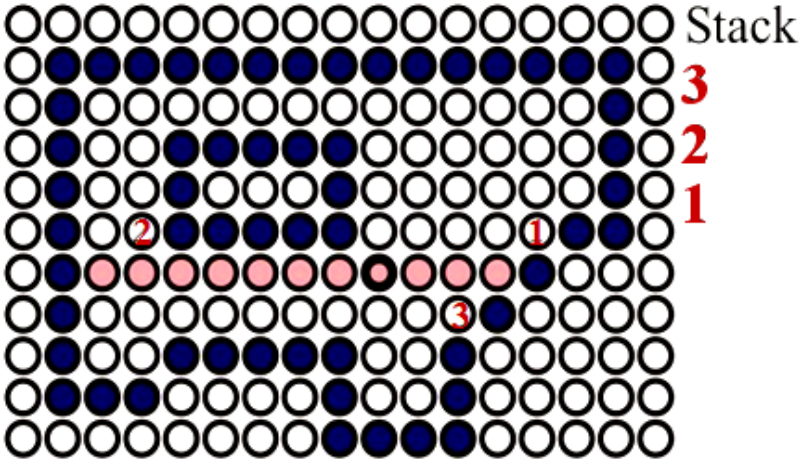
Einschub Implementierung

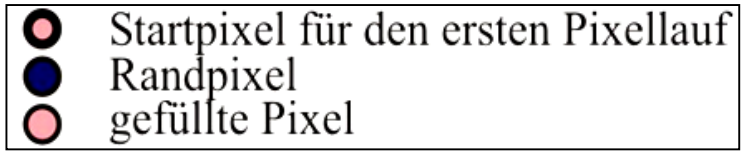
- Konzept des graphischen Debuggens:
  - definiere Iterationszähler, die dem Algorithmus bekannt sind
  - erweitere Algorithmus so, dass abgebrochen wird, sobald der Iterationszähler abgelaufen ist, und dennoch die visuelle Ausgabe erzeugt wird
  - speichere Aufrufparameter für den Algorithmus und Zustand der veränderten Datenstrukturen vor dem Aufruf
  - starte den Algorithmus mit verschieden initialisiertem Iterationszähler
- Erweiterung
  - Stapel von Iterationszählern für geschachtelte Schleifen
  - Integration der Interaktion ins GUI

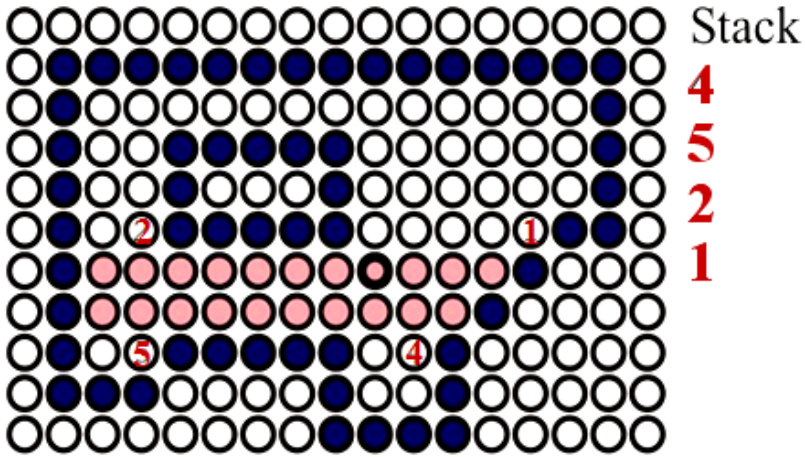
Füllalgorithmen
Vermeidung hoher Rekursionstiefe

Pixellaufalgorithmus

- Füllfunktion bearbeitet um den aktuellen Punkt gleichzeitig alle benachbarten Pixel einer Zeile. Das ist ein Pixellauf.
- Suche und fülle dazu nach links und rechts alle zu füllenden Pixel
- Starte Rekursion für alle Pixel der darüber und darunterliegenden Zeile, an denen ein Pixellauf von rechts beginnt







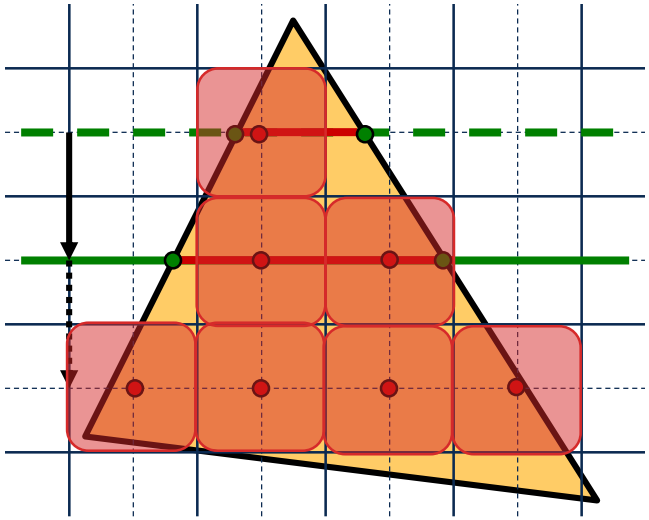
RASTERISIEREN VON DREIECKEN

Rasterisieren von Dreiecken
Mittels Punktinklusionstest

- Einfacher Brute-Force Algorithmus zum Füllen von Dreiecken
  - `B = BoundingBox(p0,p1,p2)`
  - `for each p in B`
  - `  sigma = barycentric(p,p0,p1,p2)`
  - `  if not outside(sigma) then`
  - `  setPixel(p,color(sigma))`
- kann inkrementell gemacht werden
- Iteration kann auch auf die überdeckten Pixel eingeschränkt werden

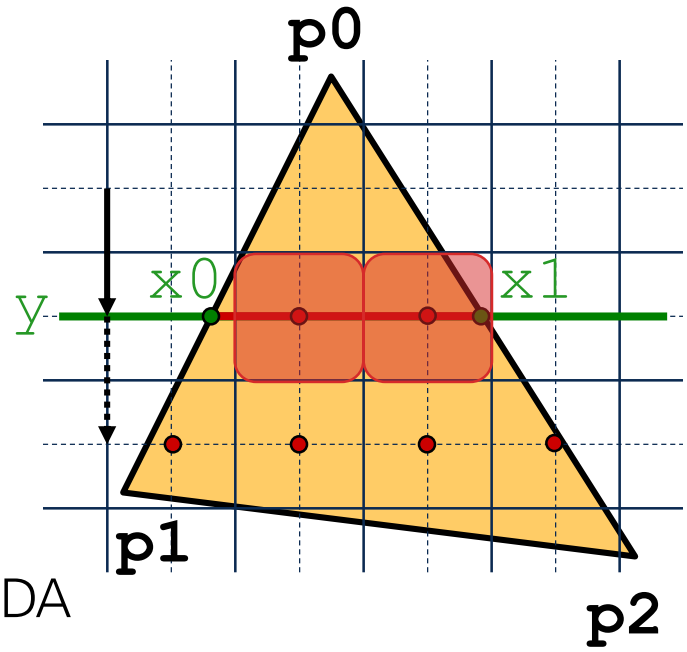
Rasterisieren von Dreiecken
Sweep Line Algorithmus

- Idee: Sweep-Line durchwandert alle vom Dreieck geschnittenen Pixelzeilen
- Beobachtung: da Dreieck konvex, ist der Schnitt ein Intervall
- Schnittintervall und baryz. Koord. werden von Zeile zu Zeile mitgeführt
- alle inneren Pixel werden auf die interpolierte Farbe gesetzt



Rasterisieren von Dreiecken
Sweep Line Algorithmus

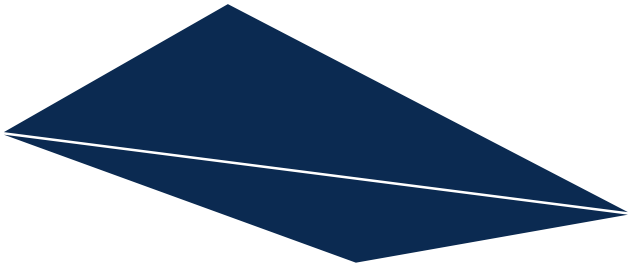
- Input: Punkte **p0,p1,p2** in Pixelkoordinaten
- sortiere Eckpunkte nach aufsteigender y-Koordinate, oBdA seien **p0,p1,p2** schon so sortiert
- `for all y = p0.y … p1.y`
  - compute `x0` on segment **p0 p1** with DDA
  - compute x1 on segment **p0 p2** with DDA
  - fill [ceil(min(`x0,x1`)),floor(max(`x0,x1`))]
- `for all y=p1.y … p2.y`
  - compute `x0` on segment **p1 p2** with DDA
  - compute `x1` on segment **p0 p2** with DDA
  - fill [ceil(min(`x0,x1`)),floor(max(`x0,x1`))]



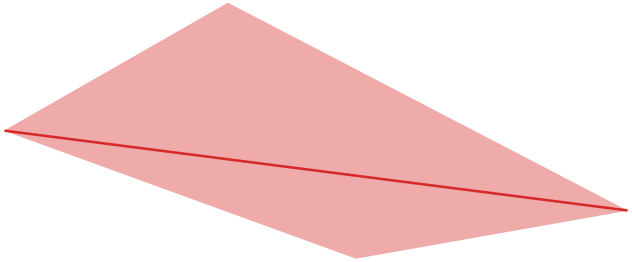
Rasterisieren von Dreiecken
Benachbarte Dreiecke

Wichtige Bedingung:

- Bei aneinander anliegenden Dreiecken soll jeder Pixel genau einmal gemalt werden, damit
  - keine Löcher entstehen

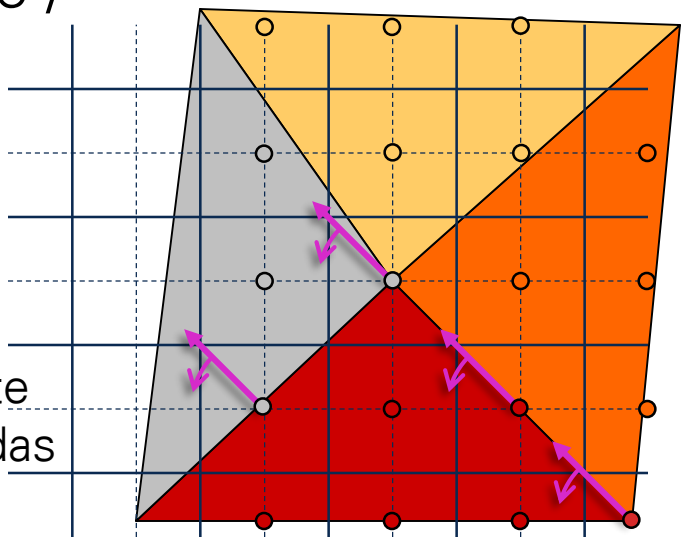


  - und bei Transparenz keine Pixel doppelt gezeichnet



Rasterisieren von Dreiecken jeden Pixel genau einmal malen

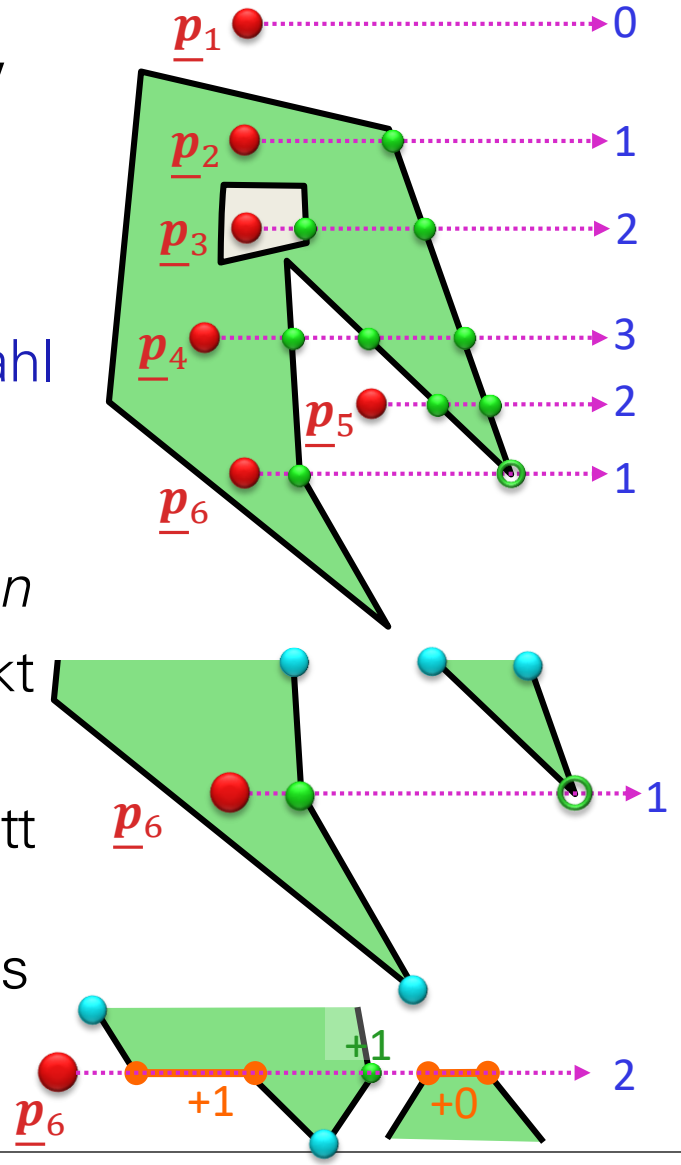
- Idee: zeichne Pixel, wenn Zentrum innerhalb des Dreiecks liegt
- Welchem Dreieck werden Pixel zugeordnet, die genau auf der Kante / Ecke liegen?
- Lösung:
  - definiere eine feste Richtungen
  - Pixel auf Kante oder Ecke
    - Wenn Richtung nicht entlang einer Kante zeigt, weise Pixel dem Dreieck zu, auf das die Richtung zeigt
    - Sonst weise Pixel dem Dreiecke zu, das gegen den Uhrzeigersinn an die Richtung angrenzt



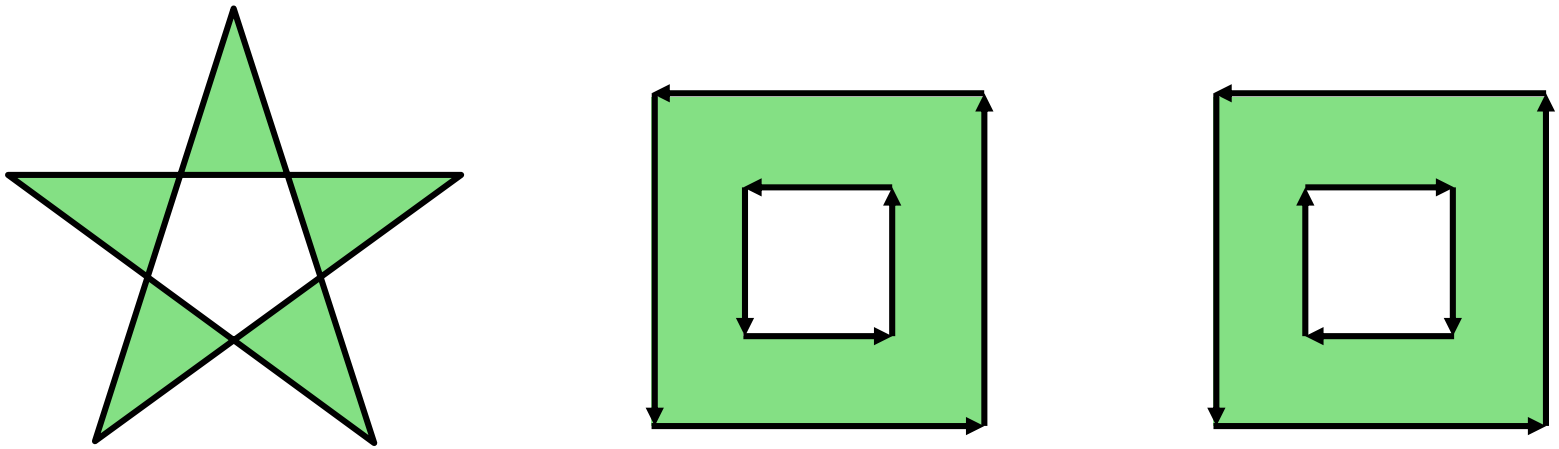
RASTERISIEREN VON POLYGONEN

Rasterisieren von Polygonen
Füllbereich – Paritätsregel

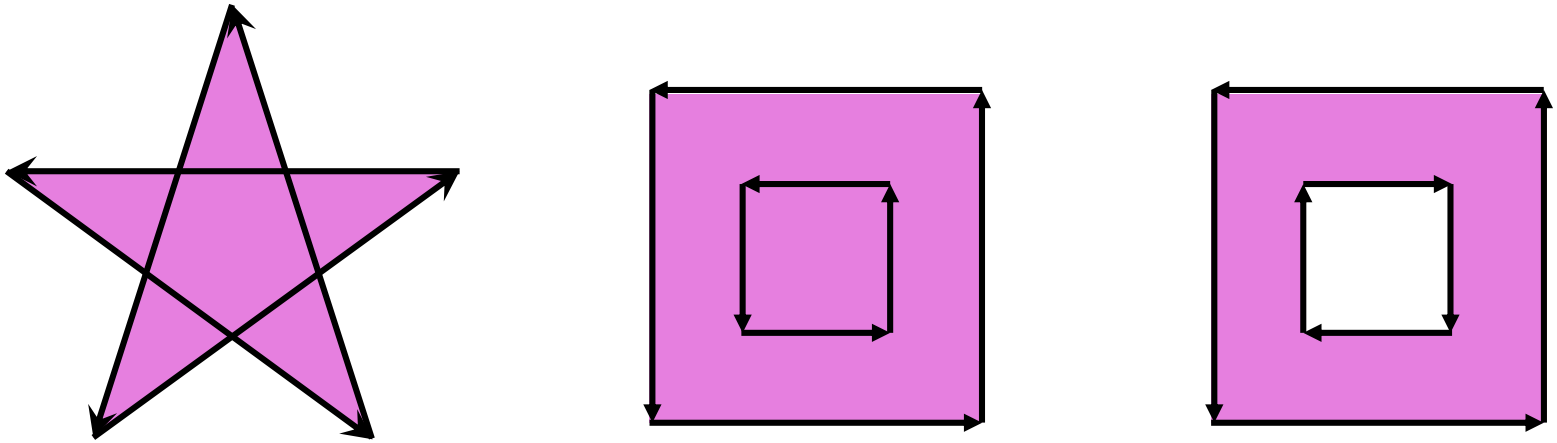
- Bei einfachen Polygonen ist intuitiv klar, welches der zu füllende Bereich ist.
- Formal kann man für jeden Punkt $\underline{p}_i$ einen Strahl in eine beliebige Richtung (hier x-Richtung) schießen und die Anzahl der Schnittpunkte mit Kanten zählen
- Paritätsregel: Bei einer *geraden* Anzahl ist der Punkt *außen* bei *ungerader innen*
- **Vorsicht** wenn der Strahl einen Eckpunkt schneidet (siehe $\underline{p}_6$) oder parallel zu Kante[n] ist: es liegt nur dann ein Schnitt vor, wenn die benachbarten Eckpunkte auf unterschiedlichen Seiten des Strahls liegen



Rasterisieren von Polygonen
Füllbereich – Non-Zero Rule

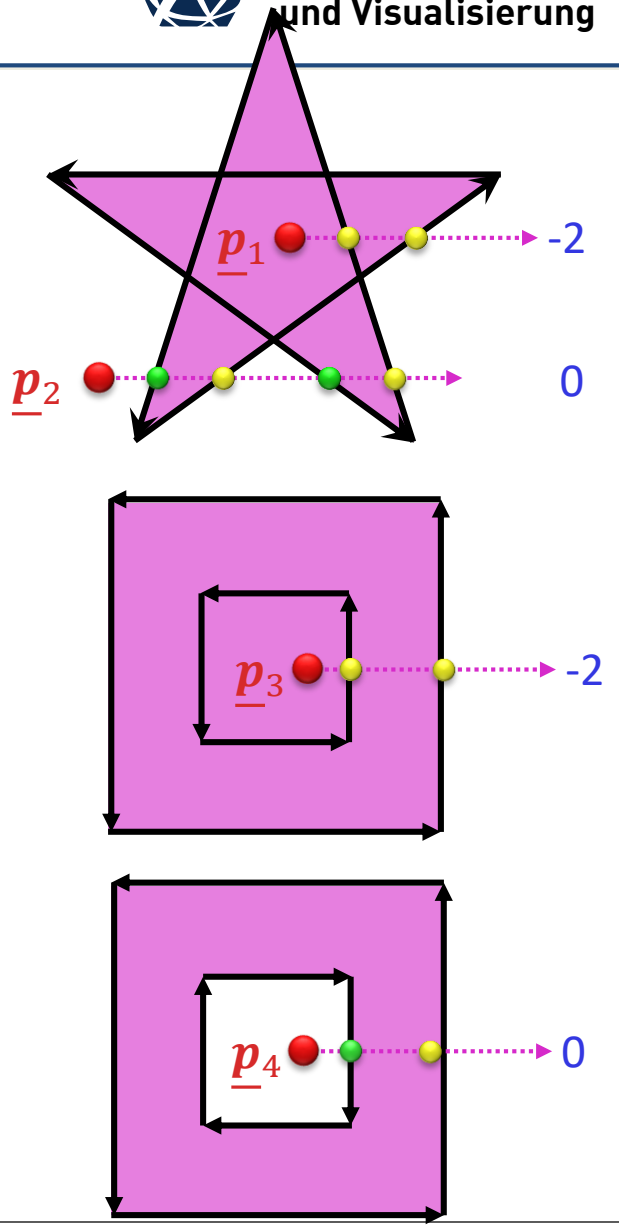


- auch bei allgemeinen Polygonen wird meist die Paritätsregel verwendet
- alternativ gibt es die Nicht-Null-Regel (non-zero):



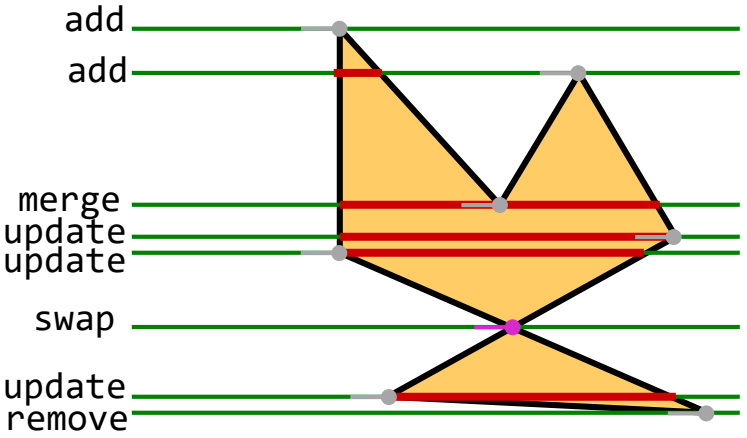
Rasterisieren von Polygonen
Füllbereich – Non-Zero Rule

- Bei der Nicht-Null-Regel wird der Umlaufsinn der Konturen berücksichtigt.
- pro Strahlschnitt wird geprüft, ob die Kante in Strahlrichtung
  (a) von rechts nach links oder
  (b) von links nach rechts
  orientiert ist
- davon abhängig wird bei
  (a) runtergezählt und bei
  (b) hochgezählt
- Nicht-Null-Regel: abschließend werden Punkte als innen klassifiziert, wenn das Zählen ungleich 0 ergibt.



Rasterisieren von Polygonen
Polygonfüllen mittel Sweep-Line

- sortiere Knoten nach y-Koordinate
- führe Liste aktiver x-Intervalle mit und speichere zugehörige Randkante
- Liste ändert sich bei folgenden Ereignissen, die in sweep-line Reihenfolge verarbeitet werden:
  - `add/split` – Intervall einfügen/spalten (Knoten außerhalb|innerhalb aktueller Intervalle, anliegende Kanten nach unten)
  - `update` – Intervallgrenzenkante neu (Knoten mit anliegenden Kanten – eine nach oben und eine nach unten)
  - `merge/remove` – Intervalle mergen/löschen (Knoten, anliegende Kanten nach oben und sie begrenzen unterschiedliche/dasselbe Intervall[e])
  - `swap` – Intervallgrenzen tauschen (Kantenschnittpunkt)
- Füllregel wird pro Zeile ausgewertet mit horizontalen unendlichen Strahlen.



RECHENBEISPIEL

Rechenbeispiel 1

- Berechne Normale aus gegebenen 2D Punkten

$$\underline{\boldsymbol{p}} = \begin{pmatrix} -3 \\ 2 \end{pmatrix}; \underline{\boldsymbol{q}} = \begin{pmatrix} 7 \\ 5 \end{pmatrix}$$

- 1. Schritt: Differenzvektor $\underline{\boldsymbol{q}} - \underline{\boldsymbol{p}} = \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} = \begin{pmatrix} 10 \\ 3 \end{pmatrix}$

- 2. Schritt: 90° rotieren: $\vec{\boldsymbol{n}} = \begin{pmatrix} -\Delta y \\ \Delta x \end{pmatrix} = \begin{pmatrix} -3 \\ 10 \end{pmatrix}$

- optional normieren:

$$\left\|\vec{\boldsymbol{n}}\right\| = \sqrt{3^2 + 10^2} = \sqrt{109} \approx 10{,}44 \Rightarrow \hat{\boldsymbol{n}} = \frac{1}{\sqrt{109}} \begin{pmatrix} -3 \\ 10 \end{pmatrix} \approx \begin{pmatrix} -0{,}29 \\ 0{,}96 \end{pmatrix}$$

Rechenbeispiel 2 (Fortsetzung von 1)

- berechne vorzeichenbehafteten Abstand von Punkt $\underline{\boldsymbol{x}}$ von der Geraden durch $\underline{\boldsymbol{p}}$ und $\underline{\boldsymbol{q}}$:

$$dist(\underline{x}) \propto \left\langle \underline{x} - \underline{p}, \vec{n} \right\rangle$$

$$\underline{\boldsymbol{p}} = \begin{pmatrix} -3 \\ 2 \end{pmatrix}; \underline{\boldsymbol{q}} = \begin{pmatrix} 7 \\ 5 \end{pmatrix}; \underline{\boldsymbol{x}} = \begin{pmatrix} 5 \\ 3 \end{pmatrix} \qquad \vec{\boldsymbol{n}} = \begin{pmatrix} -3 \\ 10 \end{pmatrix} \qquad \hat{\boldsymbol{n}} = \frac{1}{\sqrt{109}} \begin{pmatrix} -3 \\ 10 \end{pmatrix}$$

- 1. Schritt: Differenzvektor $\underline{\boldsymbol{x}} - \underline{\boldsymbol{p}} = \begin{pmatrix} 8 \\ 1 \end{pmatrix}$

- nicht normierte Variation:

$$dist(\underline{\boldsymbol{x}}) \propto \left\langle \begin{pmatrix} 8 \\ 1 \end{pmatrix}, \begin{pmatrix} -3 \\ 10 \end{pmatrix} \right\rangle = -24 + 10 = -14$$

- normierter Abstand: $dist(\underline{\boldsymbol{x}}) = \frac{1}{\sqrt{109}} \cdot (-14) \approx -1{,}34$

zurück