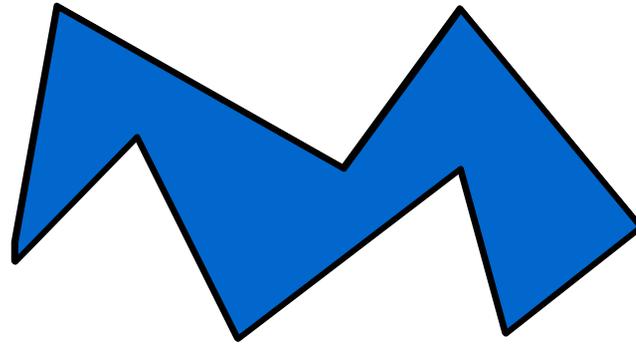


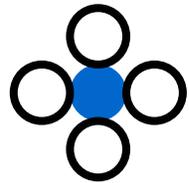
# Kapitel 4: 2D-Füllen

# Füllverfahren

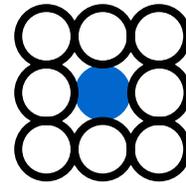


- Universelle Füllverfahren  
(Zusammenhangseigenschaften)
- Scan-Line-Verfahren  
(Geometrie)

# Universelle Füllverfahren



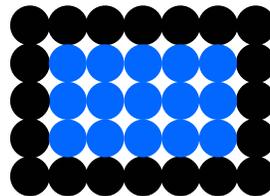
4-way-stepping



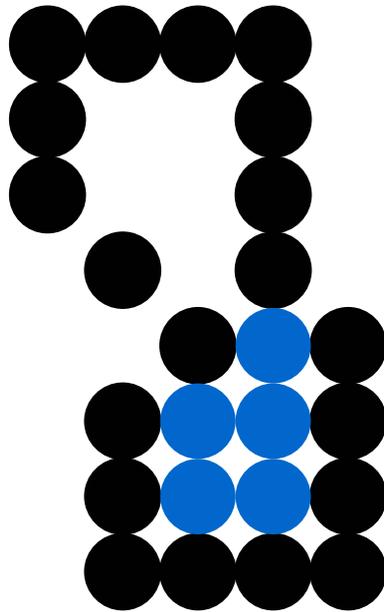
8-way-stepping

Beginnend beim Saatpixel:

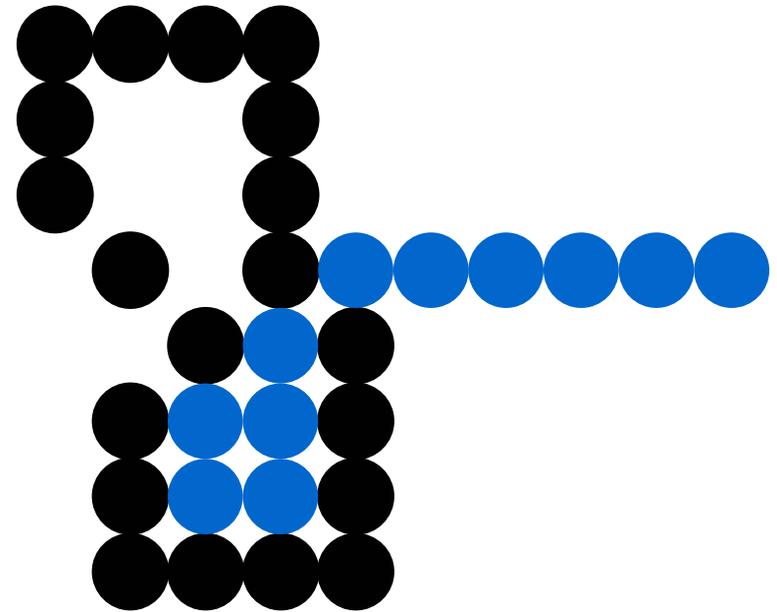
färbe alle Nachbarn, bis Umgrenzung erreicht ist.



# Probleme beim universellen Füllen



4-way-stepping



8-way-stepping

# Rekursives Füllen

rangeOK(x,y) true, falls Punkt x,y innerhalb des Bildbereichs

getPixel(x,y) true, falls Vordergrundfarbe an Punkt x,y

setPixel(x,y) setze Vordergrundfarbe an Punkt x,y

```
public void boundaryFill(int x, int y){  
    if (rangeOk(x,y) && !getPixel(x,y)){  
        setPixel(x,y);  
        boundaryFill(x+1,y);  
        boundaryFill(x, y+1);  
        boundaryFill(x-1,y);  
        boundaryFill(x, y-1);  
    }  
}
```

# Rekursives Leeren

delPixel(x,y) setze Hintergrundfarbe an Punkt x,y

```
public void boundaryEmpty(int x, int y){
    if (rangeOk(x,y) && getPixel(x,y)){
        delPixel(x,y);
        boundaryEmpty(x+1,y);
        boundaryEmpty(x+1,y+1);
        boundaryEmpty(x ,y+1);
        boundaryEmpty(x-1,y+1);
        boundaryEmpty(x-1,y);
        boundaryEmpty(x-1,y-1);
        boundaryEmpty(x ,y-1);
        boundaryEmpty(x+1,y-1);
    }
}
```

# boundaryFill-Implementation

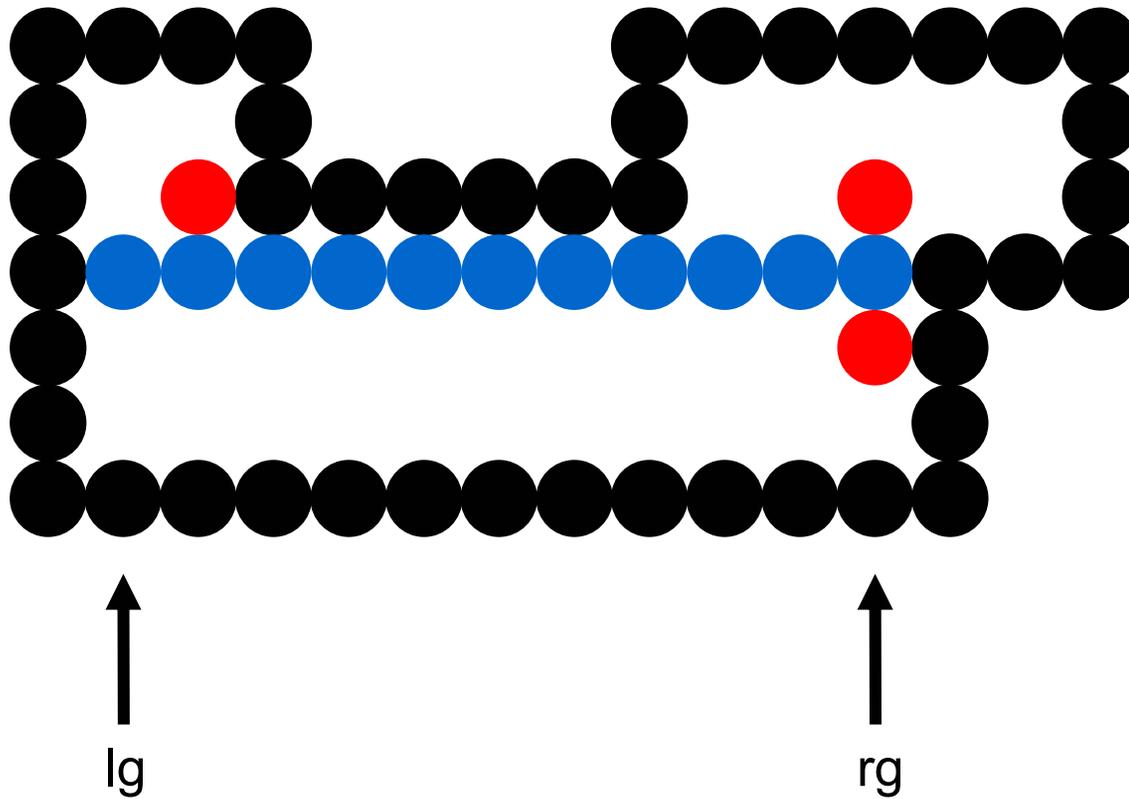
Java-Applet:

[~cg/2004/skript/node46.htm](#)

# Analyse von boundaryFill

- jedes Pixel (bis auf Randpixel) wird 4-mal auf den Keller gelegt
- besser: lege Repräsentant auf Keller
- Repräsentant färbt horizontale Linie und legt Repräsentanten für Nachbarlinien auf Keller

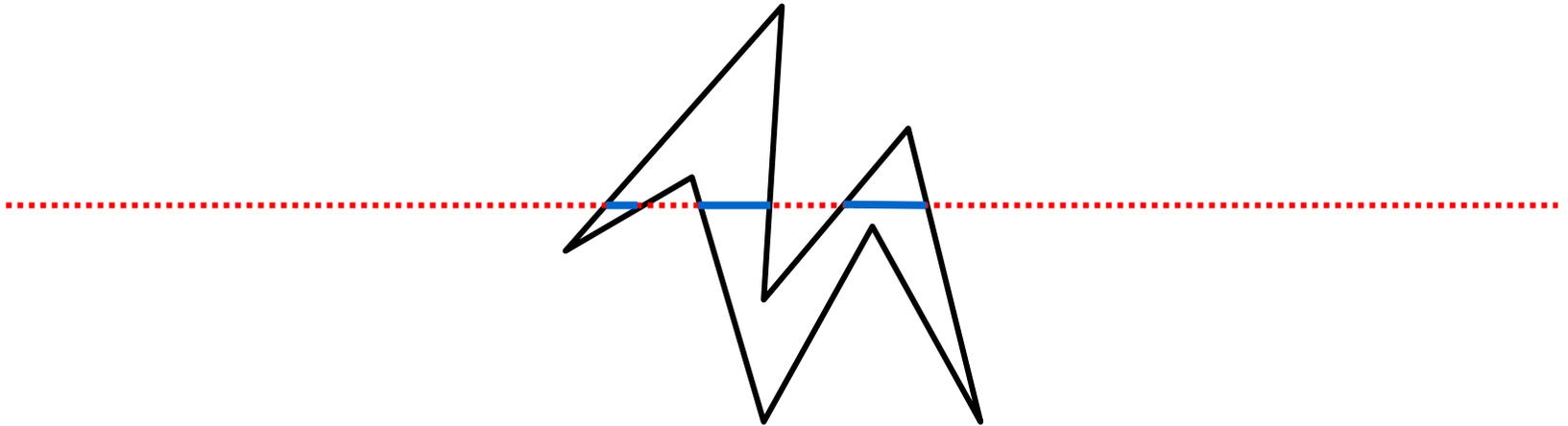
# fillRowByRow



# fillRowByRow: Algorithmus

```
public void fillRowByRow(int x, int y) {  
  
    int lg; int rg; int px = x;  
    while(!getPixel(x,y)) {setPixel(x,y); x--;}  
    lg = x+1;  
    x = px + 1;  
    while(!getPixel(x,y)) {setPixel(x,y); x++;}  
    rg = x-1;  
    for(x = rg; x >= lg; x--) {  
        if(!getPixel(x, y-1))  
            fillRowByRow(x,y-1);  
        if(!getPixel(x, y+1))  
            fillRowByRow(x,y+1);  
    }  
}
```

# Scan-Line-Verfahren

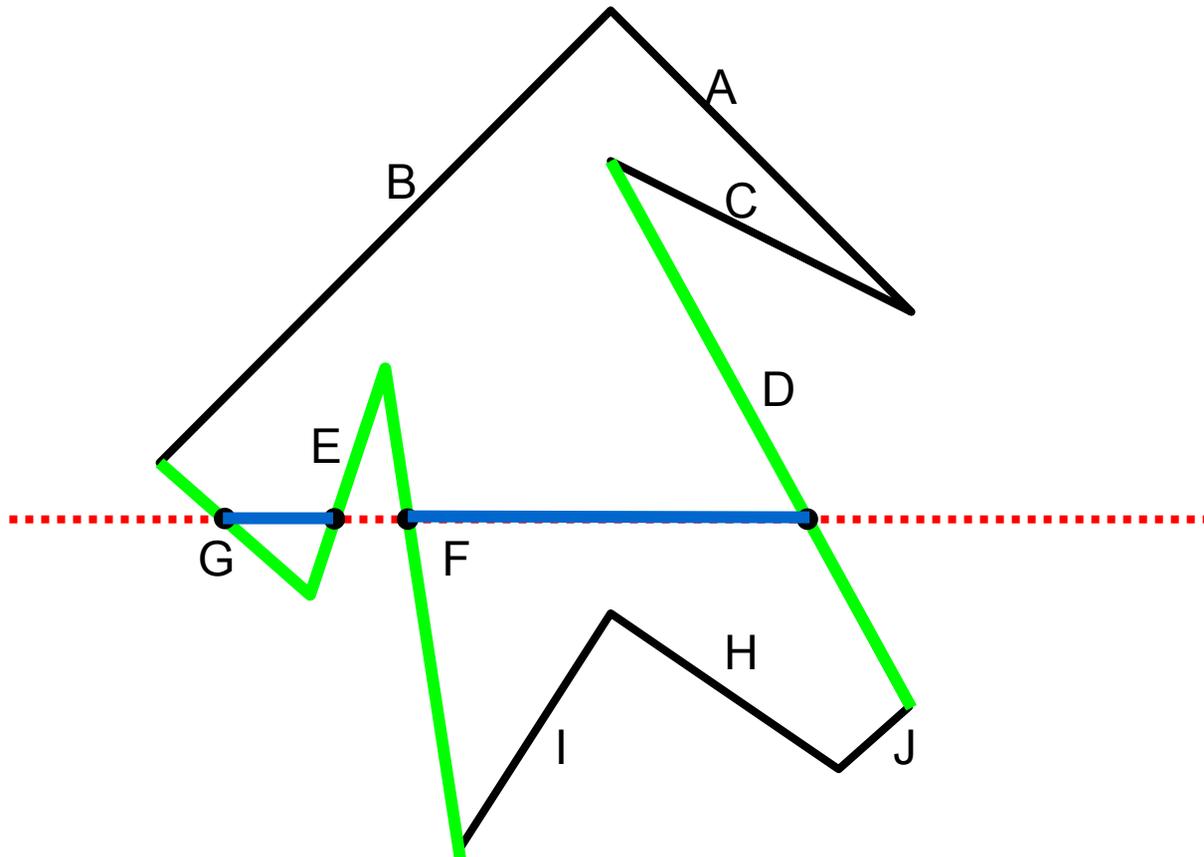


Bewege waagerechte Scan-Line von oben nach unten über das Polygon und färbe entsprechende Abschnittsgeraden

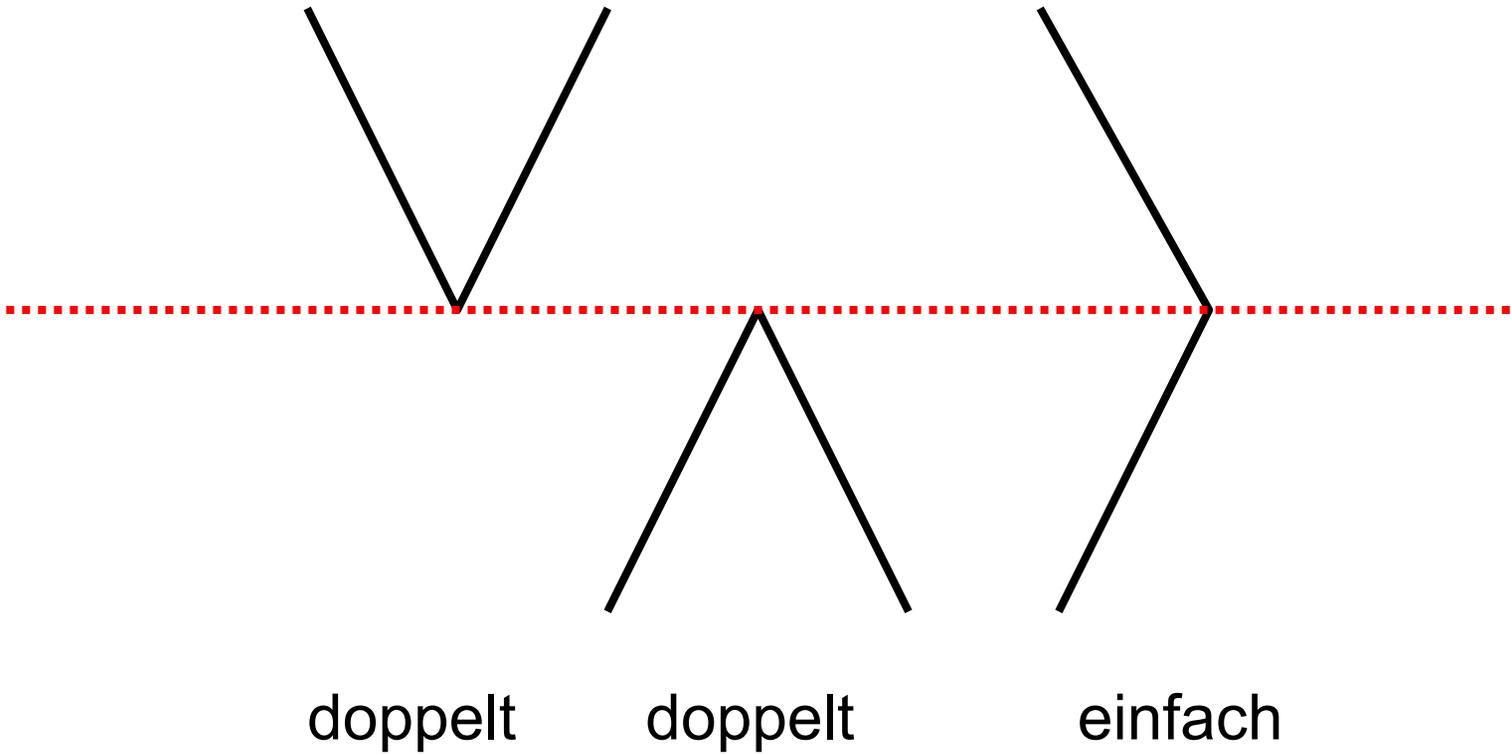
# Scan-Line-Verfahren: Detail

1. Sortiere Kanten nach größtem y-Wert
2. Bewege Scan-Line von oben nach unten
3. für jede Position der Scan-Line:
  - ermittle aktive Kanten
  - berechne Schnittpunkte mit Scan-Line
  - sortiere die Schnittpunkte nach x Werten
  - färbe abwechselnd zwischen Schnittpunkten

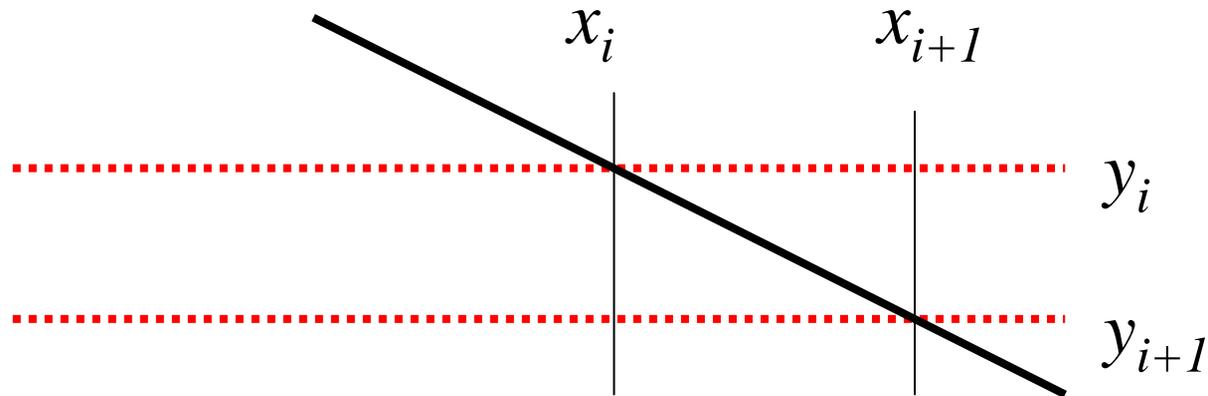
# Scan-Line-Verfahren: Beispiel



# Problemfälle



# Schnittpunkte fortschreiben



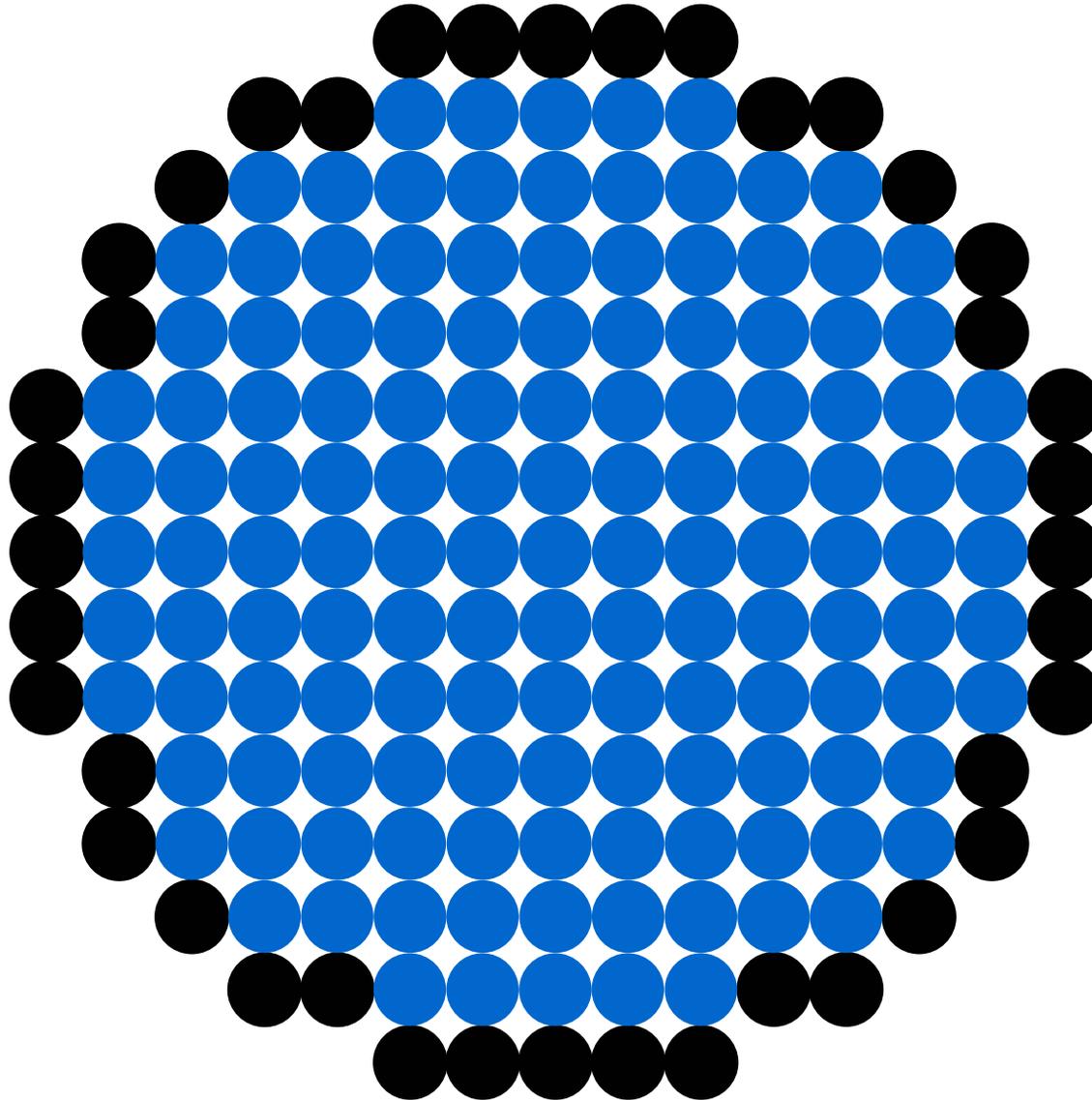
$$s = (y_i - y_{i+1}) / (x_i - x_{i+1})$$

$$(x_i - x_{i+1}) = (y_i - y_{i+1}) / s$$

$$y_i - y_{i+1} = 1$$

$$x_{i+1} := x_i - 1/s$$

# Scan-Line-Verfahren für Kreis



# Scan-Line-Implementation

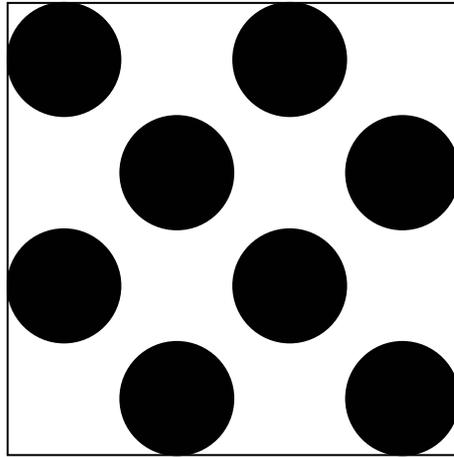
Quelltext:

[~cg/2004/Sources/scan.txt](#)

Java-Applet:

[~cg/2004/skript/node46.htm](#)

# Dither-Matrix: Definition

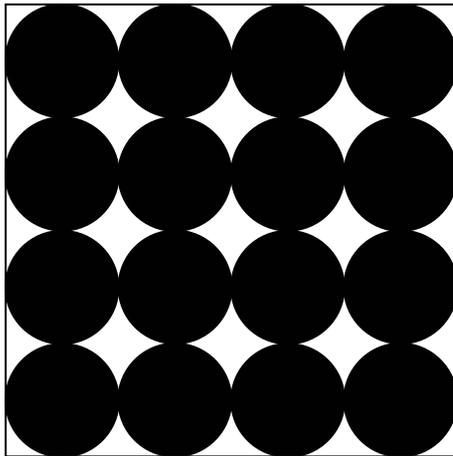


0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

Eine  $n \times n$  Dithermatrix enthält gleichmäßig verteilt alle Zahlen aus dem Intervall  $[0..n^2 - 1]$

Für Grauwert  $0 \leq k \leq n^2$   
setze alle Pixel mit Eintrag  $< k$

# Dither-Matrix: Beispiel



0	8	2	10
12	4	14	6
3	11	1	9
15	7	13	5

# Dither-Matrix: Konstruktion

$$D_0 = (0)$$

$U_n = n \times n$  - Matrix, besetzt mit 1

$$D_n = \begin{pmatrix} 4 \cdot D_{n-1} + 0 \cdot U_{n-1} & 4 \cdot D_{n-1} + 2 \cdot U_{n-1} \\ 4 \cdot D_{n-1} + 3 \cdot U_{n-1} & 4 \cdot D_{n-1} + 1 \cdot U_{n-1} \end{pmatrix}$$

$$D_1 = \begin{pmatrix} 0 & 2 \\ 3 & 1 \end{pmatrix}$$

$$D_2 = \begin{pmatrix} 0 & 8 & 2 & 10 \\ 12 & 4 & 14 & 6 \\ 3 & 11 & 1 & 9 \\ 15 & 7 & 13 & 5 \end{pmatrix}$$

# Dither-Matrix: Aufruf

```
public boolean kleinerSchwelle(  
    int x, int y, int schwelle){  
    return (mat[x%n][y%n] < schwelle)  
}
```

**Einfärben mit Grauwert k:**

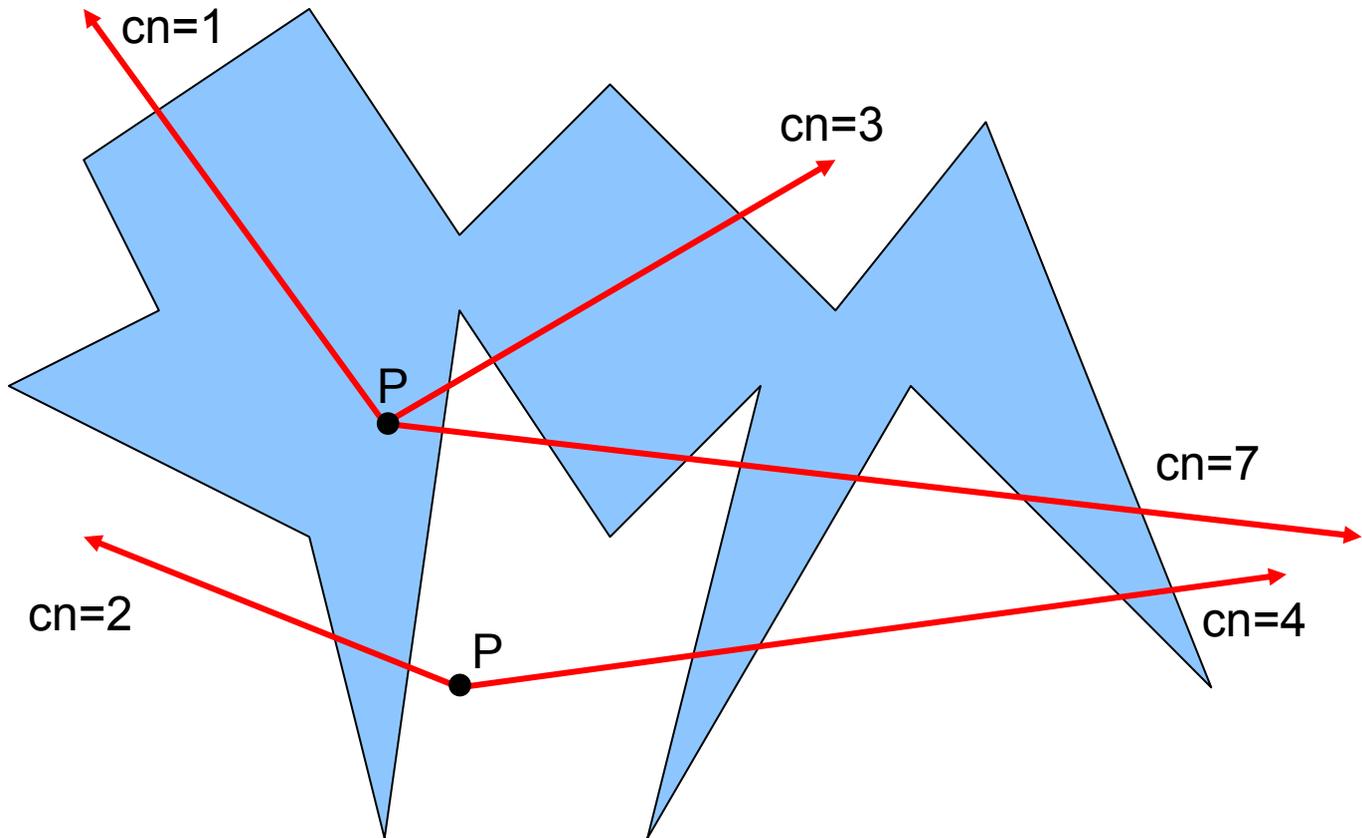
```
if (kleinerSchwelle(x,y,k))  
    setPixel(x,y);  
else  
    delPixel(x,y);
```

# Dither-Matrix-Implementation

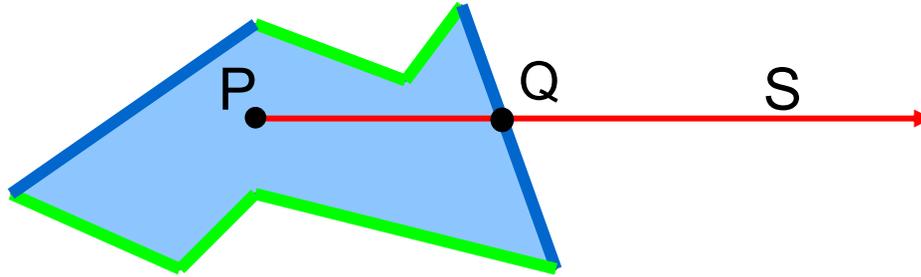
Java-Applet:

[~cg/2004/skript/node46.htm](#)

# Punkt in Polygon



# Kreuzungszahl berechnen



Sei  $S$  der von  $P$  nach rechts gehende Strahl

Für jede Polygonkante von  $P_1$  nach  $P_2$ :

falls  $P_1$  und  $P_2$  oberhalb: kein Schnittpunkt

falls  $P_1$  und  $P_2$  unterhalb: kein Schnittpunkt

falls  $P_1$  und  $P_2$  auf verschiedenen Seiten:

berechne Schnittpunkt  $Q$  mit  $S$

falls rechts von  $P$ : erhöhe Kreuzungszahl

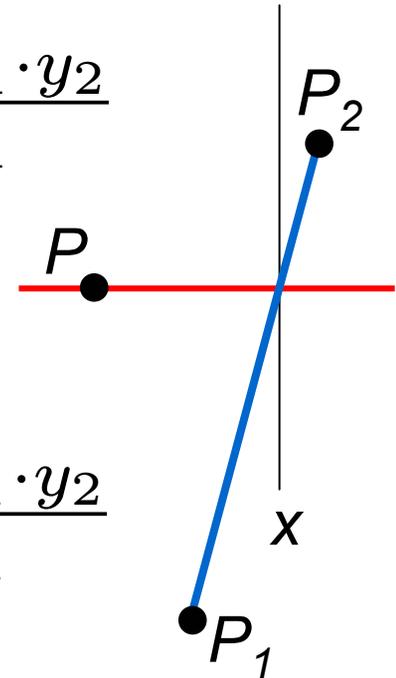
# Schnittpunkt berechnen

$$f(x) = \frac{y_2 - y_1}{x_2 - x_1} \cdot x + \frac{x_2 \cdot y_1 - x_1 \cdot y_2}{x_2 - x_1}$$

$$f(x) = y$$

$$y = \frac{y_2 - y_1}{x_2 - x_1} \cdot x + \frac{x_2 \cdot y_1 - x_1 \cdot y_2}{x_2 - x_1}$$

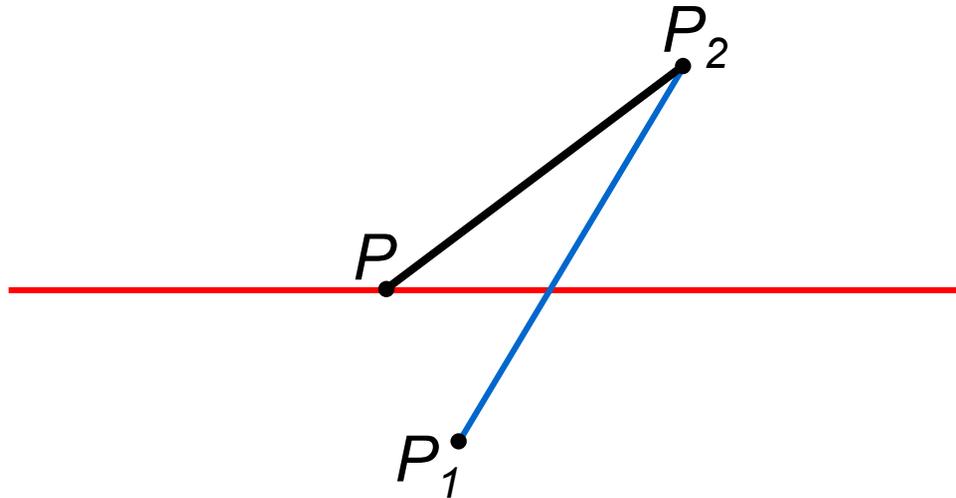
$$x = \frac{y \cdot (x_2 - x_1) - x_2 \cdot y_1 + x_1 \cdot y_2}{y_2 - y_1}$$



public boolean contains (int x, int y)

```
x1 = xpoints[n-1]; y1 = ypoints[n-1];
x2 = xpoints[0];   y2 = ypoints[0];
boolean inside = false;
boolean startUeber = y1 >= y ? true : false;
for (i=1; i<n; i++) {
    boolean endUeber = y2 >= y ? true : false;
    if ((startUeber != endUeber &&
        (double)(y*(x2-x1) - x2*y1 + x1*y2)/(y2-y1) >= x))
        inside = !inside;
    startUeber = endUeber;
    y1=y2; x1=x2; x2=xpoints[i]; y2=ypoints[i];
}
return inside;
```

# Steigung statt Schnittpunkt



Kante von  $P_1$  nach  $P_2$  schneidet  
Strahl rechts von  $P$ , falls  $\frac{y_2 - y_1}{x_2 - x_1} \geq \frac{y_2 - y}{x_2 - x}$

$$(y_2 - y_1) \cdot (x_2 - x) \geq (y_2 - y) \cdot (x_2 - x_1)$$

# public boolean contains (int x, int y)

```
...  
for (i=1; i<n; i++) {  
    boolean endUeber = y2 >= y ? true : false;  
    if (startUeber != endUeber && endUeber &&  
        (y2-y1)*(x2-x) >= (y2-y)*(x2-x1))  
        inside = !inside;  
    if (startUeber != endUeber && startUeber &&  
        (y2-y1)*(x1-x) >= (y1-y)*(x2-x1))  
        inside = !inside;  
    ...  
}
```

sich überschneidende Polygone

