Computergrafik

Übung: Rasterisierung

Prof. Dr.-Ing. Carsten Dachsbacher
Lehrstuhl für Computergrafik
Karlsruher Institut für Technologie

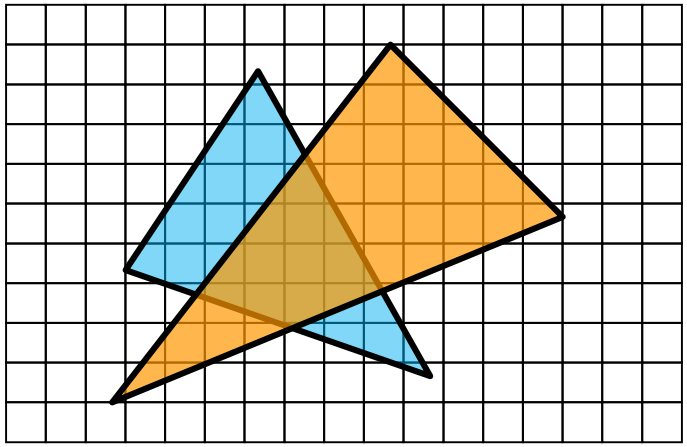
Organisatorisches

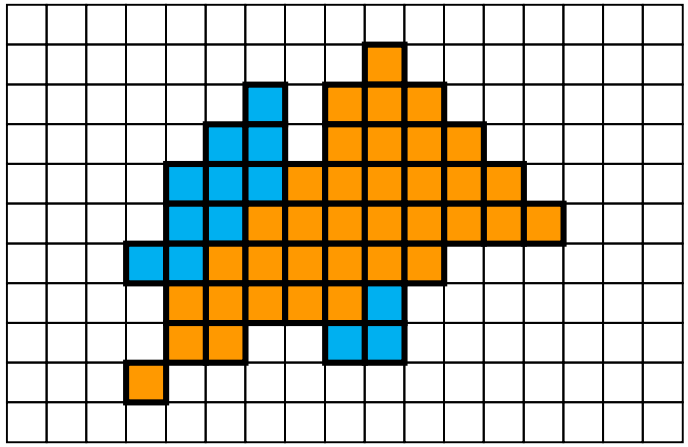
- Anmelden für den Übungsschein (und damit auch zum Modul)!
  - An- und Abmeldefrist am 28.11.2016.
  - Wir korrigieren Abgaben nur, wenn ihr angemeldet seid.
  - Wer sich nicht anmelden kann, schreibt uns bitte eine E-Mail.

- Übungsblatt wurde aktualisiert.

Inhalt

- Rasterisieren von Linien
- Polygon-Scanline-Rasterisierung

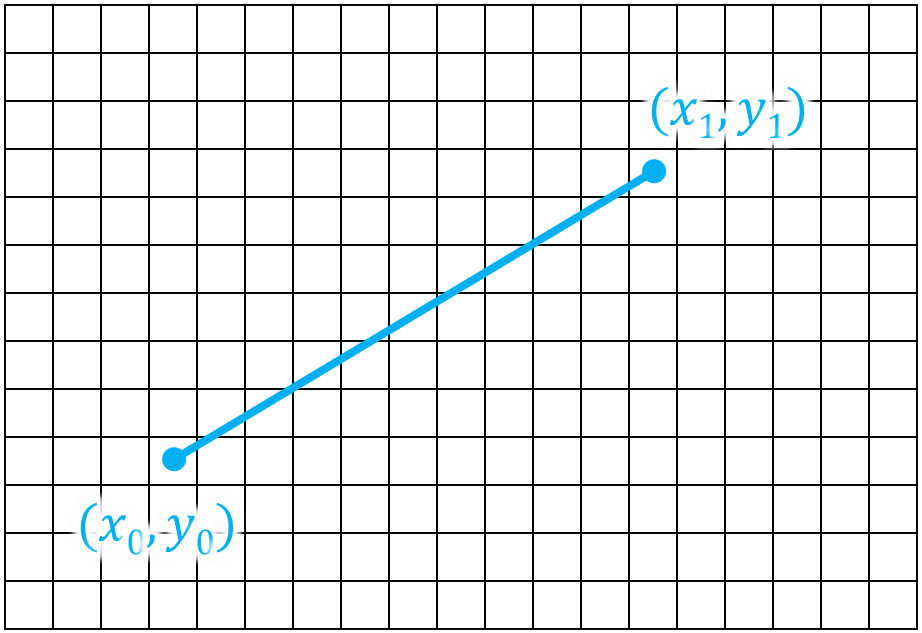




Rasterisierung

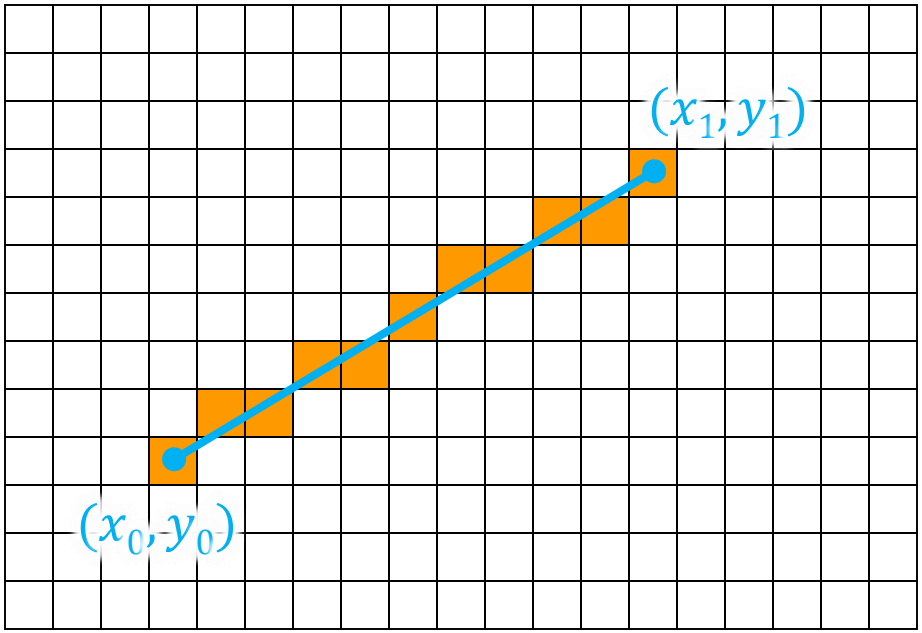
Rasterisierung von Linien

- geg.: Endpunkte des Liniensegments $(x_0, y_0), (x_1, y_1) \in \mathbb{Z}^2$
- ges.: Menge der Pixel, die gesetzt werden sollen
  - optional: wenn die Linie eine bestimmte Dicke hat – welche Pixel muss man mit welcher Intensität setzen



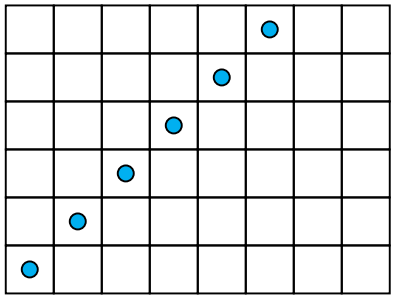
Rasterisierung von Linien

- Rasterisierung von 2D Liniensegmenten
  - transformiere kontinuierliches Primitiv in diskrete Samples/Pixel
  - uniforme Strichstärke und Helligkeit
  - zusammenhängende Pixel, keine Lücken
  - wie macht man das akkurat und schnell?



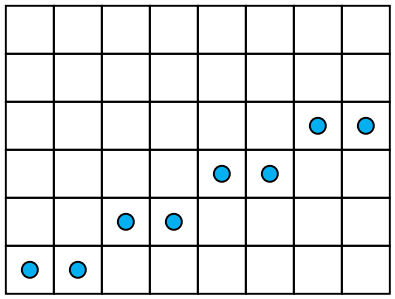
Rasterisierung von Linien

- zeichne Linie von $P_0 = (x_0, y_0)$ nach $P_1 = (x_1, y_1)$
- Steigung $m = \frac{\Delta y}{\Delta x} = \frac{y_1 - y_0}{x_1 - x_0}$
- Beispiele
  - von $(0,0)$ nach $(6,6)$



Steigung $m = 6 / 6$

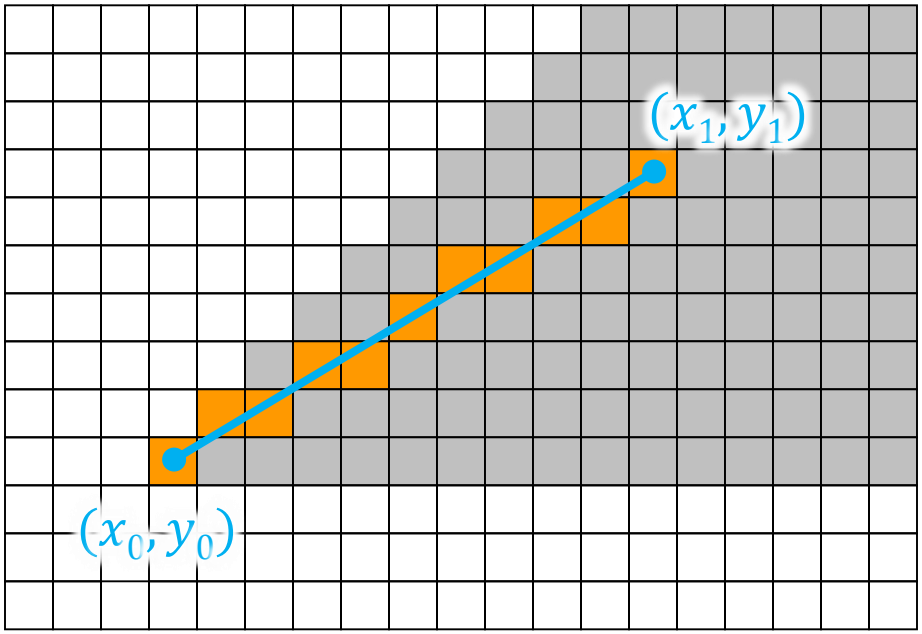
  - von $(0,0)$ nach $(8,4)$



Steigung $m = 4 / 8$

Rasterisierung von Linien

- Linien-/Geradengleichung: $y = mx + \beta$
  - mit $m = (y_1 - y_0)/(x_1 - x_0)$ und $\beta = y_0 - mx_0 = \frac{y_0 x_1 - y_1 x_0}{x_1 - x_0}$
  - $\Rightarrow y = mx + \beta = \frac{y_1 - y_0}{x_1 - x_0} x + \frac{y_0 x_1 - y_1 x_0}{x_1 - x_0}$
- Vereinfachung: betrachte nur den Fall $0 \leq m \leq 1$
  - alle anderen Fälle sind durch Ausnutzung von Symmetrie abgedeckt



Rasterisierung von Linien

- Brute Force Algorithmus
  - 3 Gleitkommaoperation pro Pixel: multiply, add, round

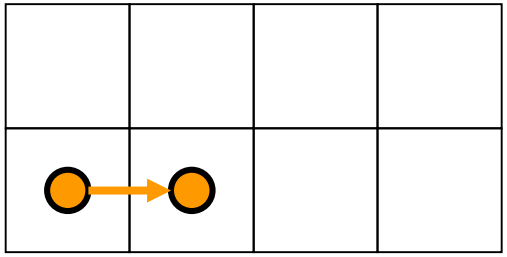
```
float m = (y1 – y0) / (x1 – x0);
float beta = ( y0 * x1 - y1 * x0 ) / ( x1 – x0 );

for ( int x = x0; x <= x1; x++ ) {
  float y = m * x + beta;
  set_pixel( x, round( y ) );
}
```

- erste Verbesserung: inkrementelle Berechnung
  - $y_n = mx_n + \beta$ und $y_{n+1} = m(x_n + 1) + \beta = y_n + m$
  - 2 Gleitkommaoperationen pro Pixel: add, round

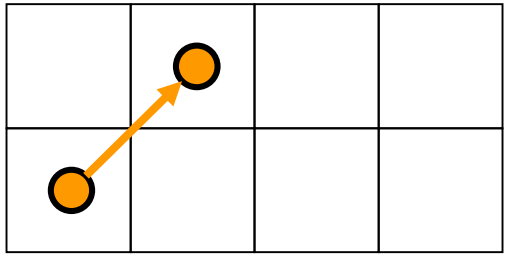
```
float m = (y1 - y0) / (x1 - x0);
float beta = ( y0 * x1 - y1 * x0 ) / ( x1 – x0 );
float y = m * x0 + beta;

for ( int x = x0; x <= x1; x++ ) {
  set_pixel( x, round( y ) )
  y += m;
}
```

Rasterisierung von Linien

- nur 2 mögliche Fälle, wenn man von einem Pixel x_n zu x_{n+1} geht:



East



North East

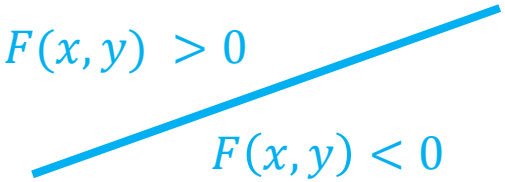
- Ziel des **Bresenham Algorithmus**:
  finde Bedingung **cond**, um zu entscheiden welcher Fall eintreten soll

```
int y = y0;

for ( int x = x0; x <= x1; x++ ) {
  set_pixel( x, y );
  if ( cond ) y++;
}
```

Rasterisierung von Linien

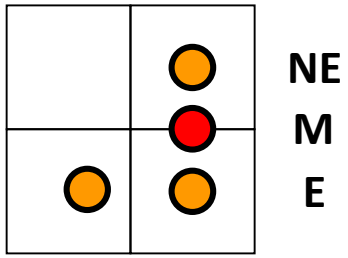
- implizite Darstellung einer Linie

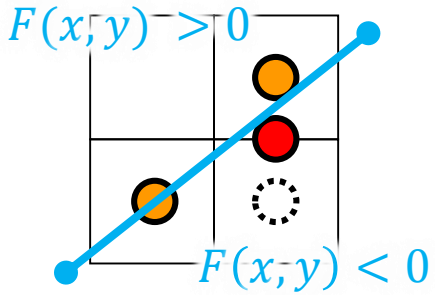
$$F(x,y) = (y - y_0) - m(x - x_0)$$

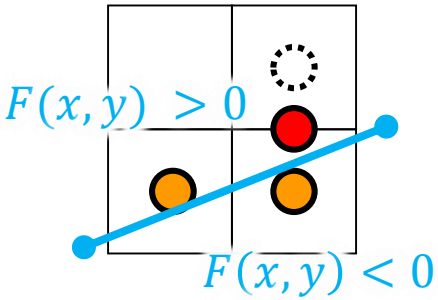


Rasterisierung von Linien

- werte $F(x, y)$ für Mitte $\mathrm{M} = (x + 1, y + \frac{1}{2})$ zwischen Pixel E und NE aus:
  - $F(M) < 0 \Rightarrow$ M unter Linie $\Rightarrow$ Linie verläuft hauptsächlich durch NE





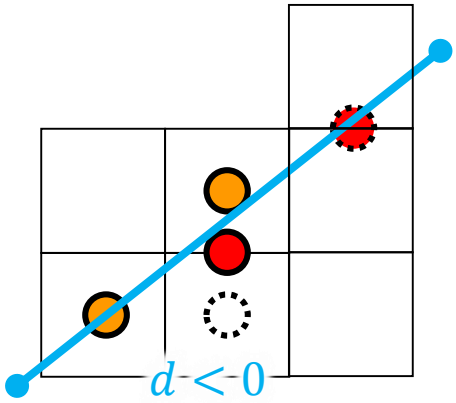



```
int y = y0;

for ( int x = x0; x <= x1; x++ ) {
  set_pixel( x, y );
  if ( F(x+1,y+0.5) < 0 ) y++;
}
```

Bresenham-Algorithmus

- basiert auf der implizierten Darstellung und arbeitet
  - inkrementell
  - nur mit Integer-Operationen

Schritt 1: inkrementelle Auswertung



- Initialisierung: $d := F\left(x_0 + 1, y_0 + \frac{1}{2}\right)$
- betrachte zunächst nur den ersten Schleifendurchlauf
- wenn $d < 0$ dann NE: $(x_0, y_0) \rightarrow (x_0 + 1, y_0 + 1)$
  - Veränderung von d:

$$F\left(x_0 + 2, y_0 + \frac{3}{2}\right) - F\left(x_0 + 1, y_0 + \frac{1}{2}\right) = (y_0 - y_1) + (x_1 - x_0)$$

  - $d \leftarrow d + (y_0 - y_1) + (x_1 - x_0)$

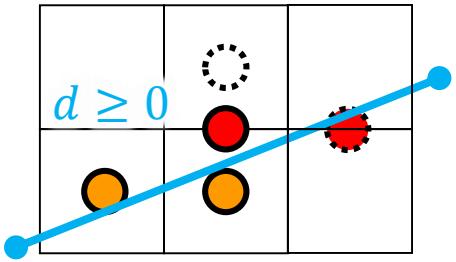
Bresenham Algorithmus

Schritt 1: inkrementelle Auswertung

- wenn $d < 0$ dann NE: $(x_0, y_0) \rightarrow (x_0 + 1, y_0 + 1)$
  - …
- wenn $d \geq 0$ dann E: $(x_0, y_0) \rightarrow (x_0 + 1, y_0)$
  - Veränderung von d:

$$F\left(x_0 + 2, y_0 + \frac{1}{2}\right) - F\left(x_0 + 1, y_0 + \frac{1}{2}\right) = (y_0 - y_1)$$

  - $\Rightarrow d \leftarrow d + (y_0 - y_1)$



Bresenham Algorithmus

Schritt 1: inkrementelle Auswertung

```
int y = y0;
float d = F( x0 + 1, y0 + 0.5);

for ( int x = x0; x <= x1; x++ ) {
  set_pixel( x, y );
  if ( d < 0 ) {
    // gehe nach NE
    y = y + 1;
    d = d + (x1 - x0) + (y0 - y1);
  } else {
    // gehe nach E
    d = d + (y0 - y1);
  }
}
```

Bresenham Algorithmus

Integer-Variante des Algorithmus

- verwende $d = 2F(x, y)$
  - möglich, weil nur das Vorzeichen für die Entscheidung relevant ist und

$$2F\left(x_0 + 1, y_0 + {}^{1}/_{2}\right) = F(2x_0 + 2, 2y_0 + 1)$$

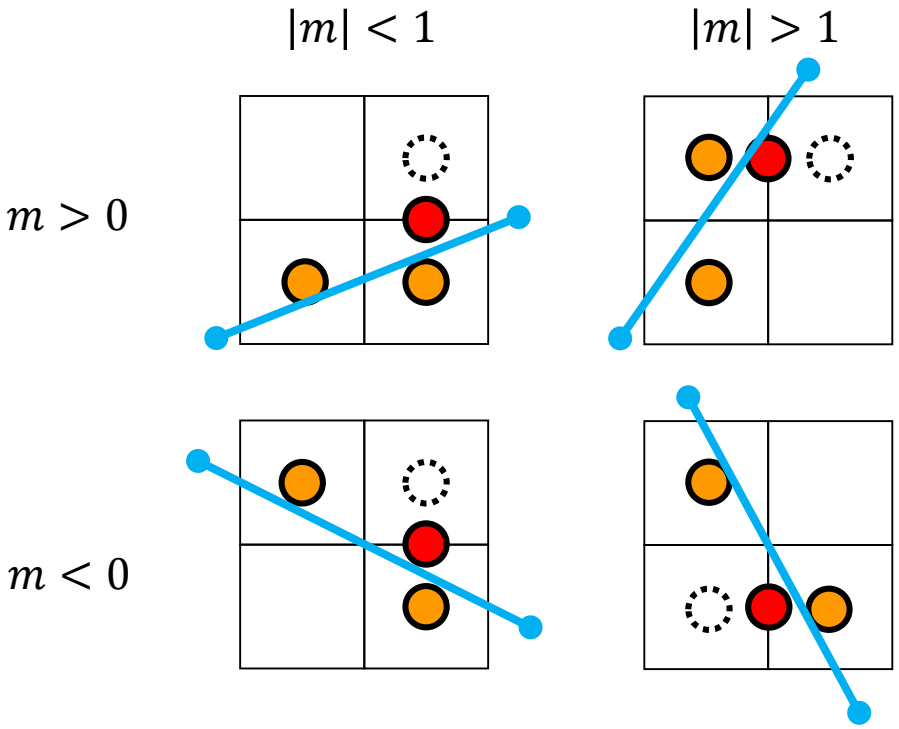
- die inkrementellen Updates sind dann

$$d \leftarrow d + 2(y_0 - y_1) \quad \text{(E)}$$

  - bzw.

$$d \leftarrow d + 2(y_0 - y_1) + 2(x_1 - x_0) \quad \text{(NE)}$$

Bresenham-Algorithmus



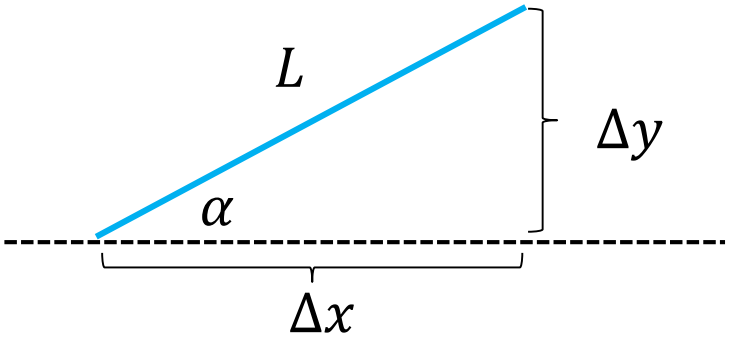
- in der Praxis:
  - nur Fallunterscheidungen bei der Initialisierung
  - Schleife über $\max(|x_1 - x_0|, |y_1 - y_0|)$ Pixel
  - alle weiteren Berechnungen inkrementell

Bresenham-Algorithmus

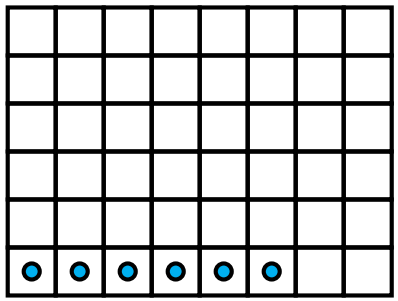
```
void line(int x0, int y0, int x1, int y1)
{
  int dx =  abs(x1-x0), sx = x0<x1 ? 1 : -1;
  int dy = -abs(y1-y0), sy = y0<y1 ? 1 : -1;
  int err = dx+dy, e2;

  while(1){
    setPixel(x0,y0);
    if (x0==x1 && y0==y1) break;
    e2 = 2*err;
    if (e2 > dy) { err += dy; x0 += sx; }
    if (e2 < dx) { err += dx; y0 += sy; }
  }
}
```

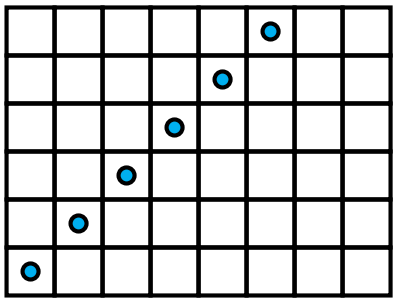
Rasterisierung von Linien

- Probleme mit den bisherigen Ansätzen
  - idealerweise würde für eine Linie der Länge L (gemessen in Pixel) auch L Pixel gesetzt werden
  - Bresenham Algorithmus setzt $\Delta x = L \cos \alpha$ Pixel



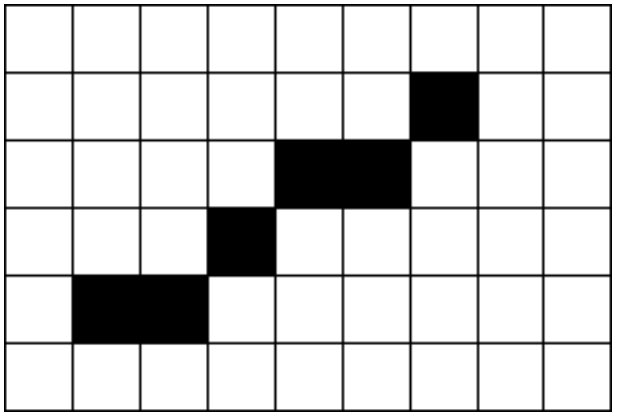
- Erhöhe die Helligkeit um den Faktor $\frac{1}{\cos \alpha}$ $(0 \leq m \leq 1)$



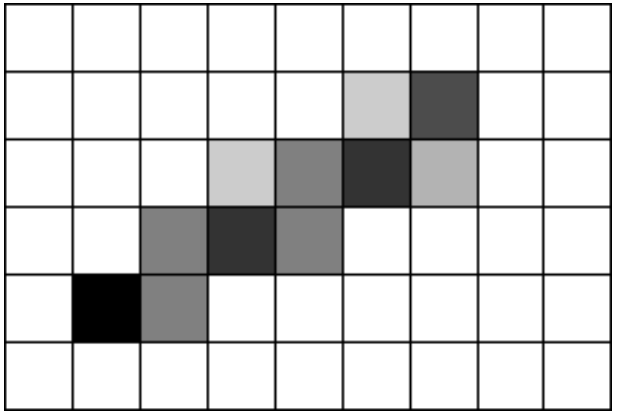


Rasterisierung von Linien mit Antialiasing

- der original Bresenham Algorithmus setzt einen Pixel pro Inkrement
  - die Intensität wird nur einem Pixel zugewiesen

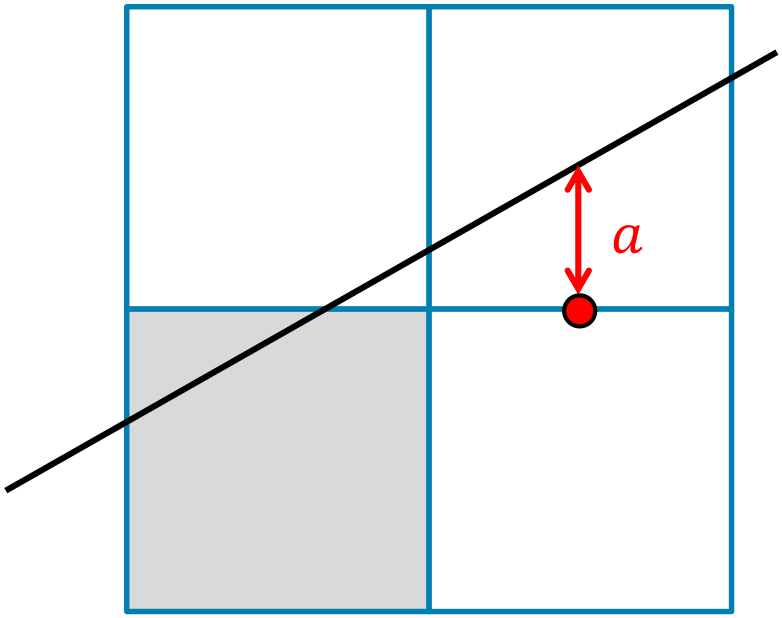


- modifizierter Bresenham
  - zwei Pixel werden gesetzt
  - Intensitäten summieren sich zur gewünschten Gesamtintensität auf



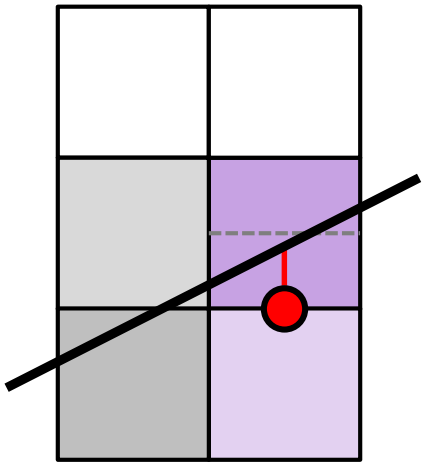
Rasterisierung von Linien mit Antialiasing

- für die Entscheidung welche Pixel mit welcher Intensität gesetzt werden benötigen wir den (vorzeichenbehafteten) Abstand a zw. der Linie und dem Mittelpunkt zw. dem E und NE Pixel
- Abstand kann aus Entscheidungsvariable berechnet werden: $a = {}^{d}/_{2}$

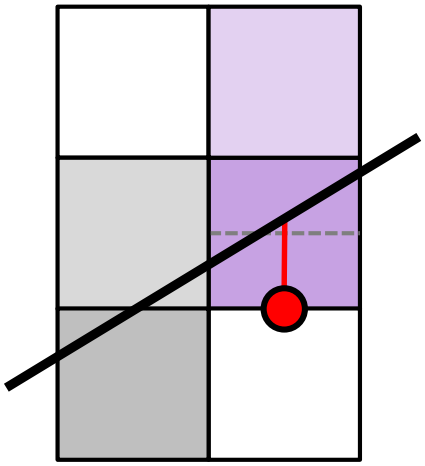


Rasterisierung von Linien mit Antialiasing

- welche Pixel sollen gesetzt werden?



$0 \geq a \geq -{}^{1}/_{2}$

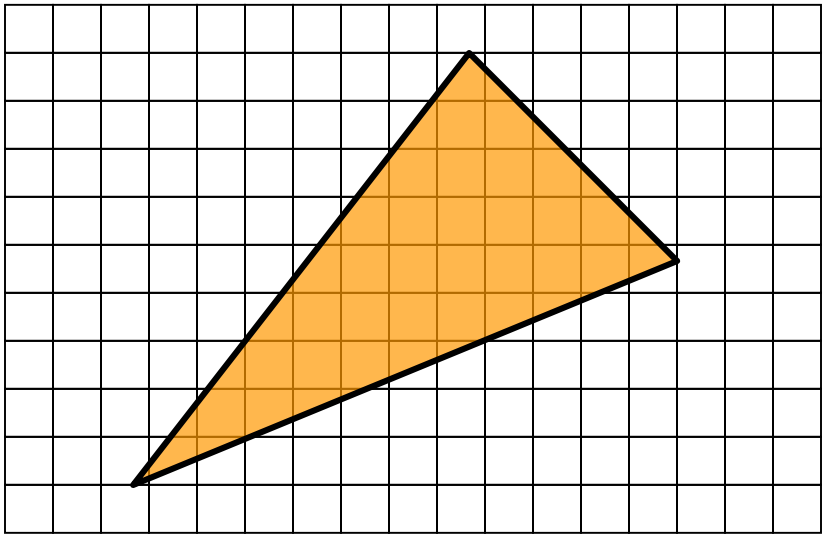


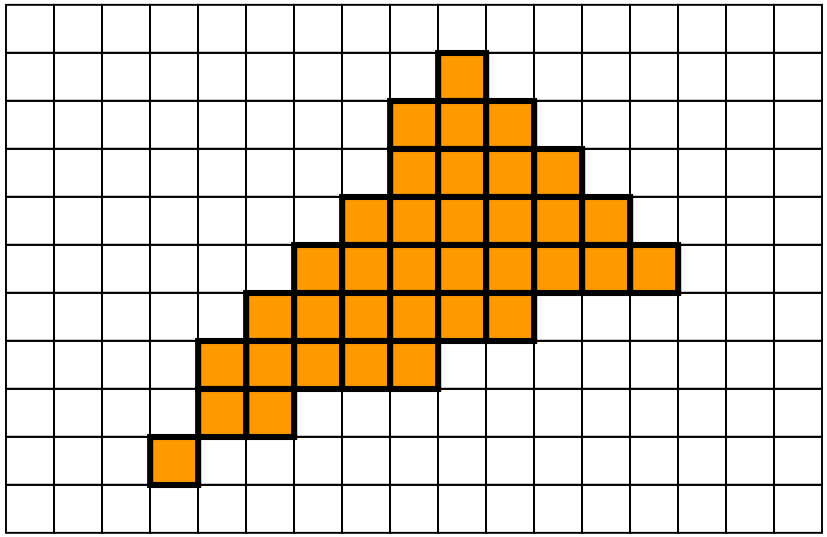
$a < -{}^{1}/_{2}$

- Rest analog.

Rasterisierung von Polygonen

- Polygon-Rasterisierung:
  - geg. ein 2D-Polygon mit n Eckpunkten P_1, …, P_n
  - färbe alle Pixel im Inneren des Polygons
- verschiedene Möglichkeiten
  - Seed-Fill (allgemeiner Flächenfüllalgorithmus, nicht praktikabel)
  - Brute Force Test: Pixel vs. Geradengleichungen der Kanten
  - Scanline Verfahren

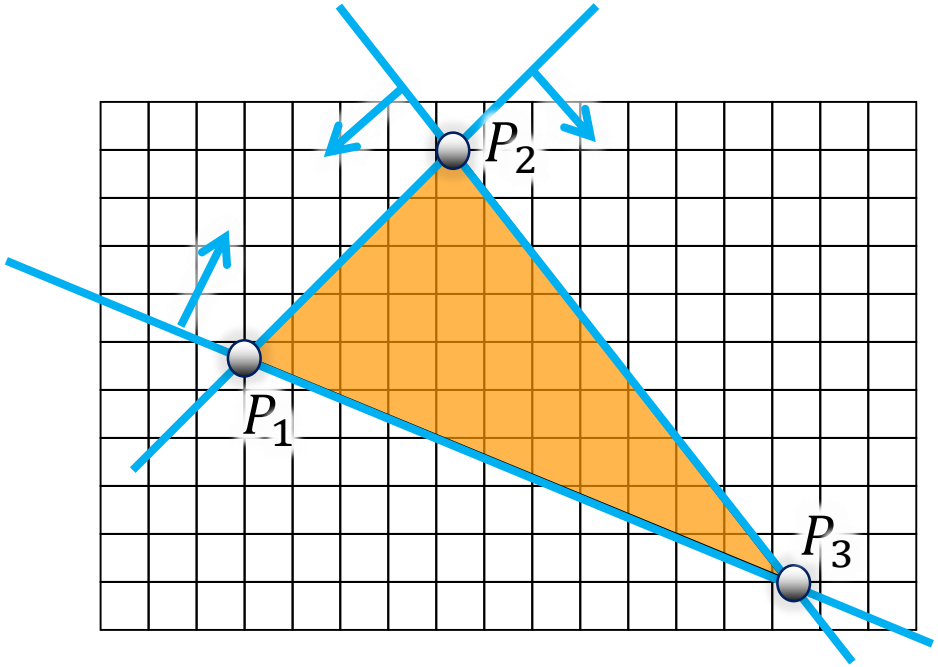




Rasterisierung von Polygonen

Brute Force Ansatz (nur für konvexe Polygone)

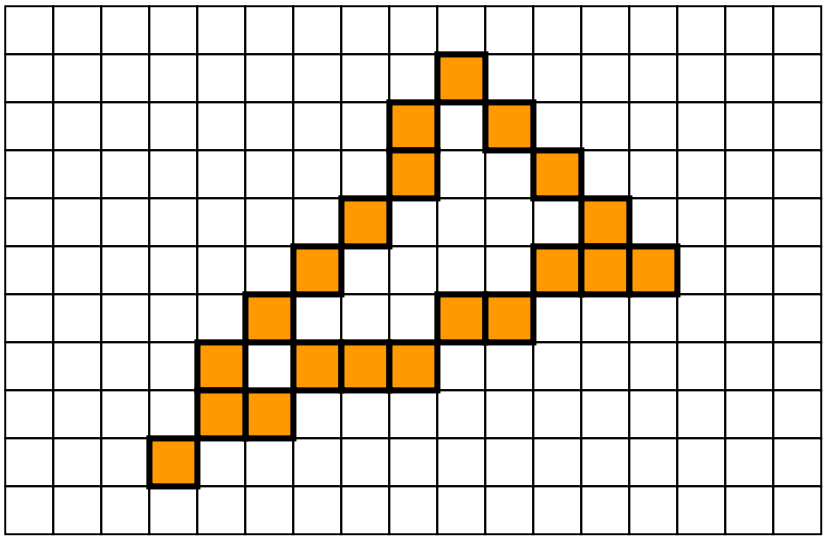
- stelle die Geradengleichungen der Kanten auf
- Orientierung so, dass der gerichtete Abstand aller Punkte im Inneren des Polygon zu den Kanten positiv ist → Test für jeden Pixel (Mittelpunkt)
- Konvention: lege fest, dass man ein Polygon „sieht“, wenn die Punkte im oder gegen den Uhrzeigersinn angeordnet sind

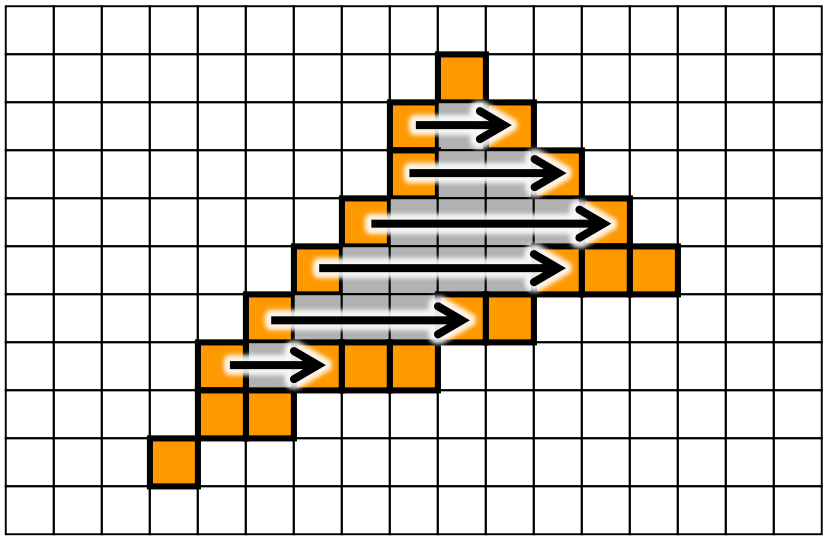


Polygon Rasterisierung

Scanline Polygon Rendering

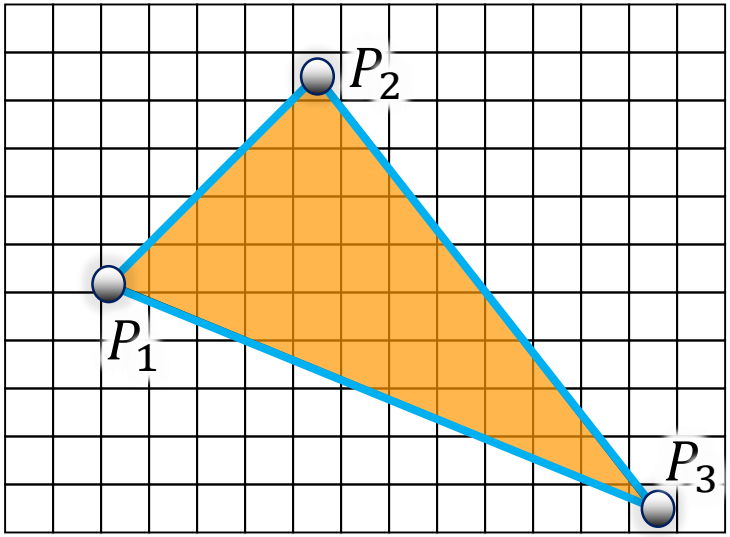
- behandle eine Scanline (Pixel-Zeile) nach der anderen
  - von unten nach oben (oder umgekehrt, je nach Implementation)
- finde Schnitte der Scanline mit dem Polygon
- setze die Pixel in diesem Teil
- wir besprechen im Folgenden das Rasterisieren von konvexen Polygonen





Polygon Rasterisierung

- Gegeben **umlaufend sortierte** Liste von Punkten $P_1 = (x_1, y_1)$, …, P_n.
- Rasterisiere iterativ Zeilen zwischen dem obersten und untersten Punkt.
- Nutze begrenzende Geraden um inkrementell Start- und Endpunkt (in der Zeile) zu bestimmen.
- Ersetze Geraden, wenn ein Zwischenpunkt (hier P_1) erreicht wird.



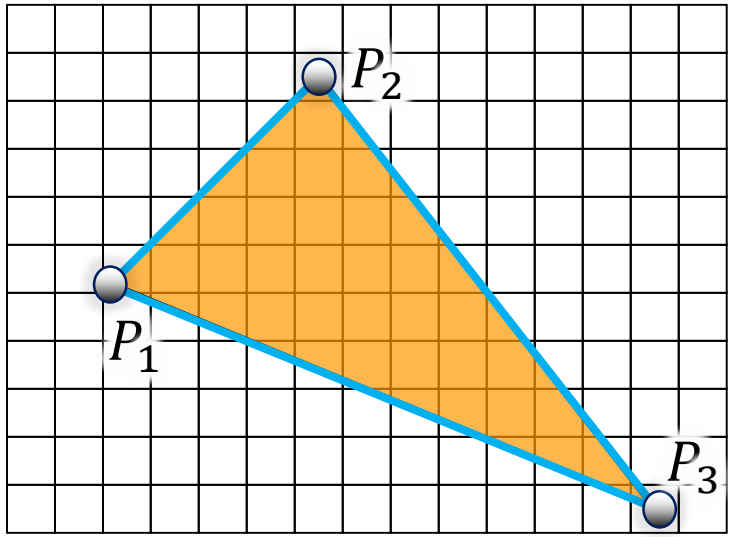
Polygon Rasterisierung

Rasterisierung von konvexen Polygonen

- Start bei P_t: $y = y_t$
  - linke Kante: $x_l = x_t$
    - Index von Start- und Endpunkt

      $cur_l = t$ und $next_l = t - 1$
  - rechte Kante $x_r = x_t$
    - Index von Start- und Endpunkt

      $cur_r = t$ und $next_r = t + 1$



- do {

  setze Pixel von x_l bis x_r in Zeile y

  y--;

  x_l -= $\Delta x_{next_l, cur_l}$;

  x_r -= $\Delta x_{next_r, cur_r}$;

  if ($y < y_{next_l}$) { $x_l = x_{next_l}$; $cur_l = next_l$; $next_l$--; }

  if ($y < y_{next_r}$) {$x_r = x_{next_r}$; $cur_r = next_r$; $next_r$++; }

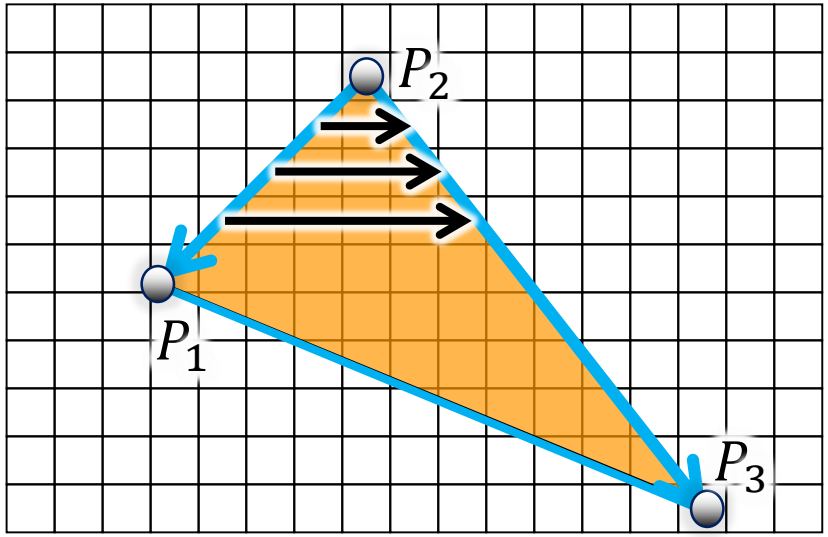
  } while ($y \geq y_b$);

- mögl. Optimierung: Bresenham für Kanten

Polygon Rasterisierung

Rasterisierung von konvexen Polygonen

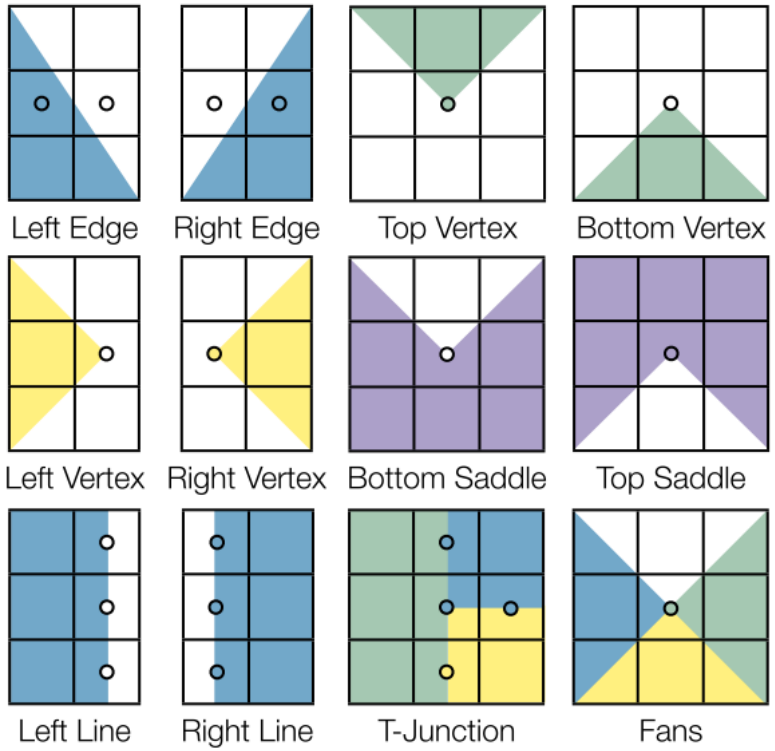
- Interpolation von Farben, Texturkoordinaten, etc.
  - interpoliere die Werte an den Vertizes linear entlang der Kanten
    - analog zur x-Koordinate ($s_l = s_t$, $\Delta s_{t-1,t} = \cdots$)
  - interpoliere dann entlang jeder Scanline zwischen den Werten an den Kanten
    - $s = s_l + (x - x_l)\frac{s_r - s_l}{x_r - x_l}$, …



Polygon Rasterisierung

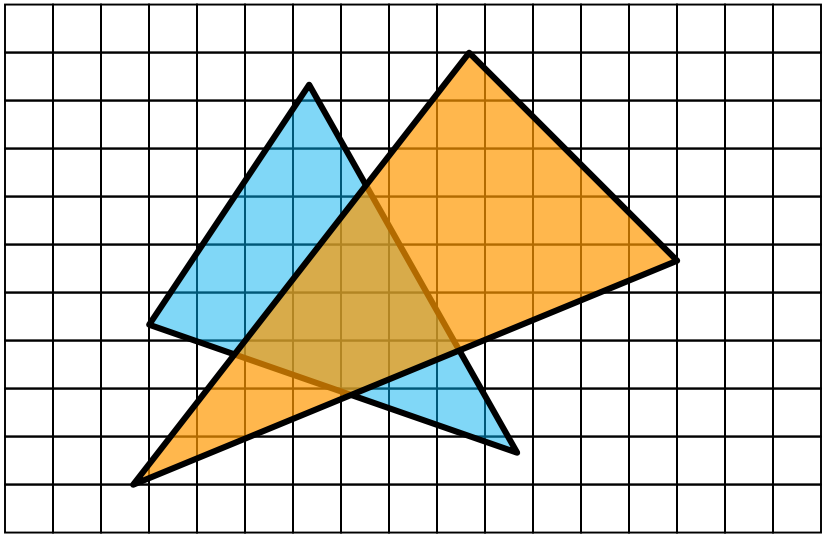
- bisher haben wir anhand des Pixelmittelpunktes unterschieden, ob der Pixel bei der Rasterisierung eines Polygons gesetzt wird
- was passiert wenn der Pixelmittelpunkt auf Kanten oder Vertizes liegt?
- hierfür gibt es Konsistenzregeln bei der Rasterisierung
  - legen fest, wann ein Pixel in einem der Spezialfälle gesetzt wird
  - bei benachbarten Dreiecken: verhindert, dass Pixel gar nicht oder mehrfach gesetzt werden

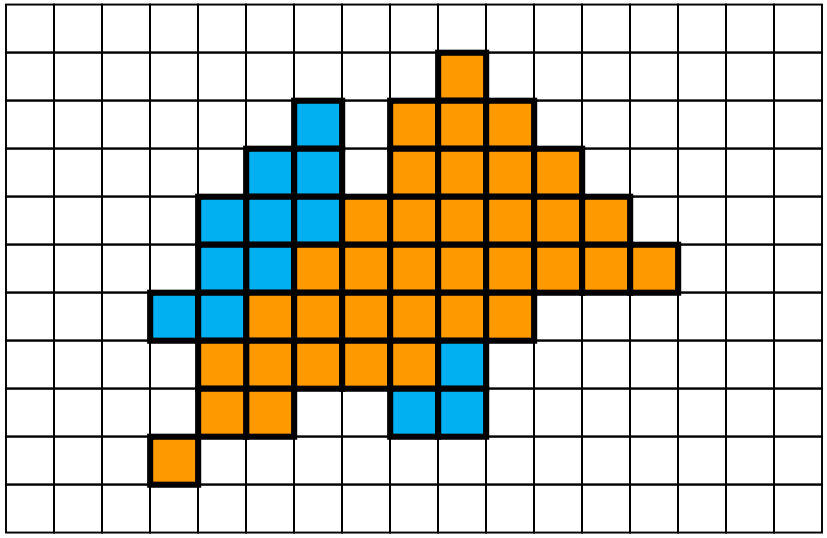
- nur bei Interesse: http://msdn.microsoft.com/en-us/library/cc627092(v=vs.85).aspx



Rasterisierung von mehreren Polygonen

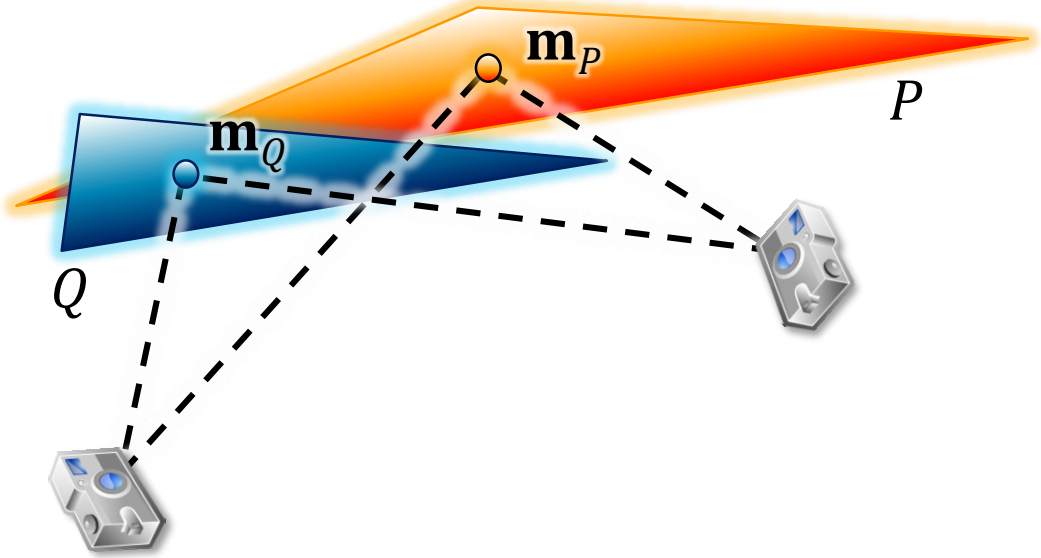
- wir möchten komplexe 3D Szenen rasterisieren
  - wir müssen das Verdeckungs- oder Sichtbarkeitsproblem lösen

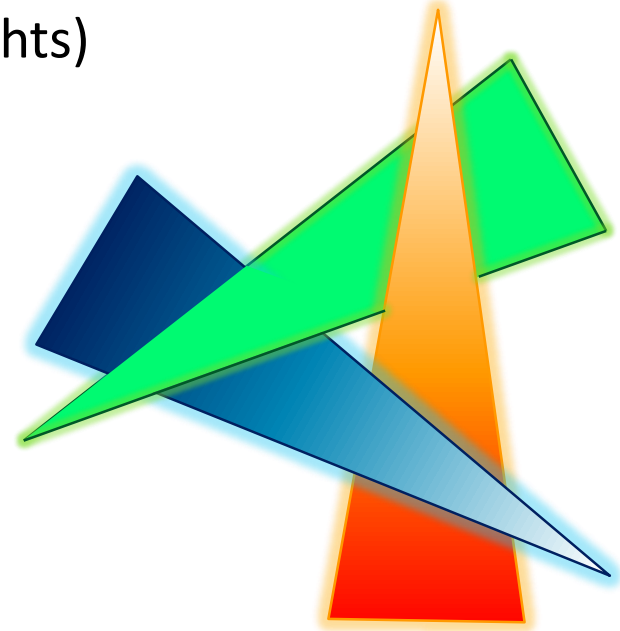




Maler-Algorithmus (Painter‘s Algorithm)

- nehmen wir an, wir könnten Polygone rasterisieren…
- einfachster Ansatz für das Sichtbarkeitsproblem: Maler-Algorithmus sortiere die Polygone (Dreiecke) nach dem Abstand zur Kamera („von hinten nach vorne“) und zeichne sie in dieser Reihenfolge
- Probleme
  - nach welchem Abstand soll sortiert werden?
    Abstand zu einem Punkt auf dem Polygon (hier: der Mittelpunkt) liefert nicht immer das richtige Ergebnis (links)
  - Zyklen können nicht sortiert werden (rechts)





Z-Puffer

- Unabhängige Erfinder: Edwin Cutmull und Wolfgang Strasser (1974)