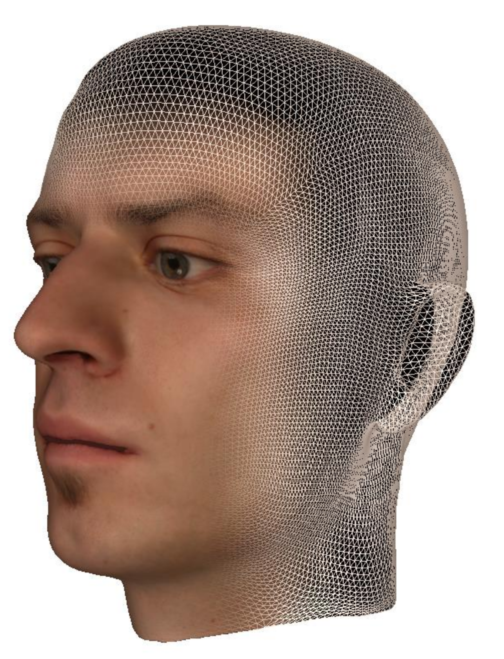
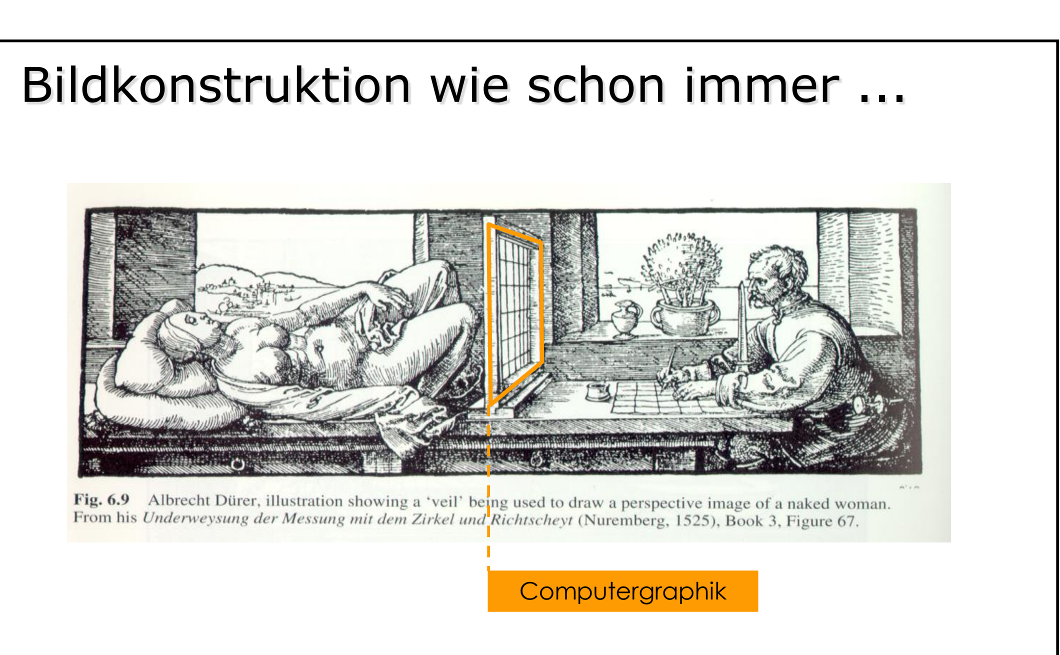
Rastergraphik
&
Rasteralgorithmen



3D Computer Graphik

(*Bemerkungen*)


Bildkonstruktion wie schon immer ...
Fig. 6.9 Albrecht Dürer, illustration showing a 'veil' being used to draw a perspective image of a naked woman.
From his Underweysung der Messung mit dem Zirkel und Richtscheyt (Nuremberg, 1525), Book 3, Figure 67.
Computergraphik

Bilderzeugung ?
Welt & Bildmodell
Computergraphik
Komplexität des Weltmodells ist von den Anforderungen abhängig!
Photorealismus
versus
Interaktivität (Komplexität)

Welt- & Bildmodelle : zwei extreme Standpunkte

A: Physikalisches Modellieren von Licht und Materie.

- der traditionelle Ansatz (Schwerpunkt der Vorlesung)
- nur Ausreichend wenn genügend Modellwissen existiert

B: Neu Bilder aus Bildvorlagen: „**Image Based Rendering IBR**“

- in Diskussion seit Anfang der 90er
- meist eingeschränkte Interaktion

Der richtige Weg??

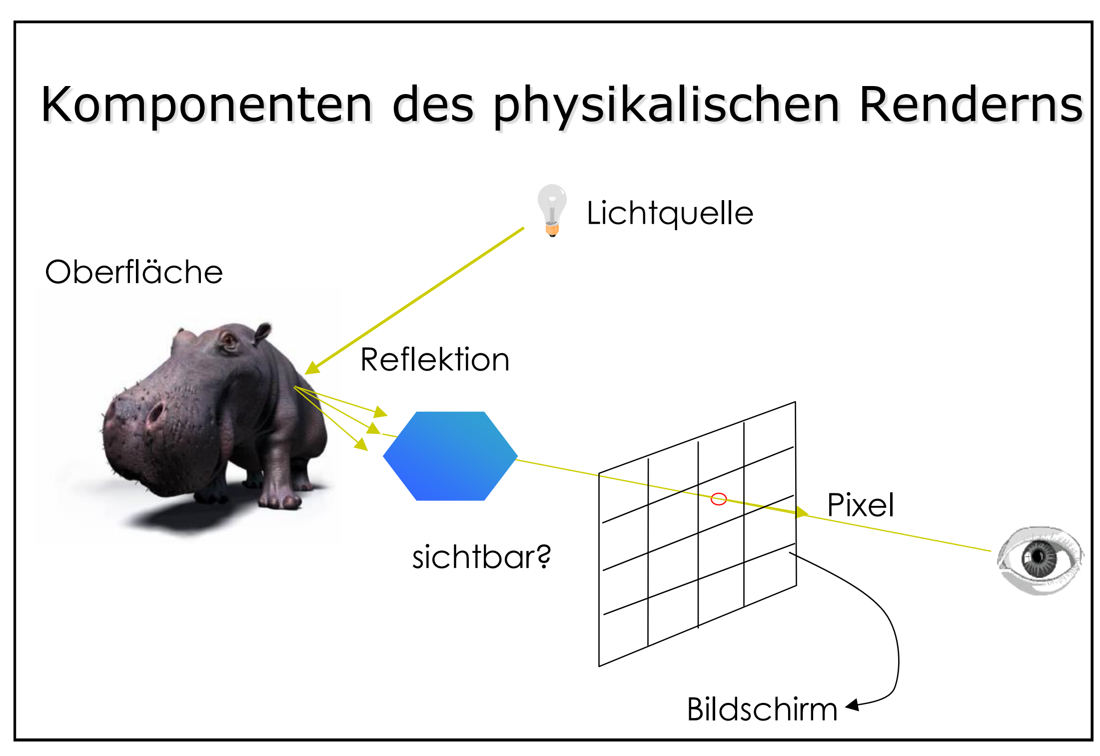
Vermutlich in der Mitte:

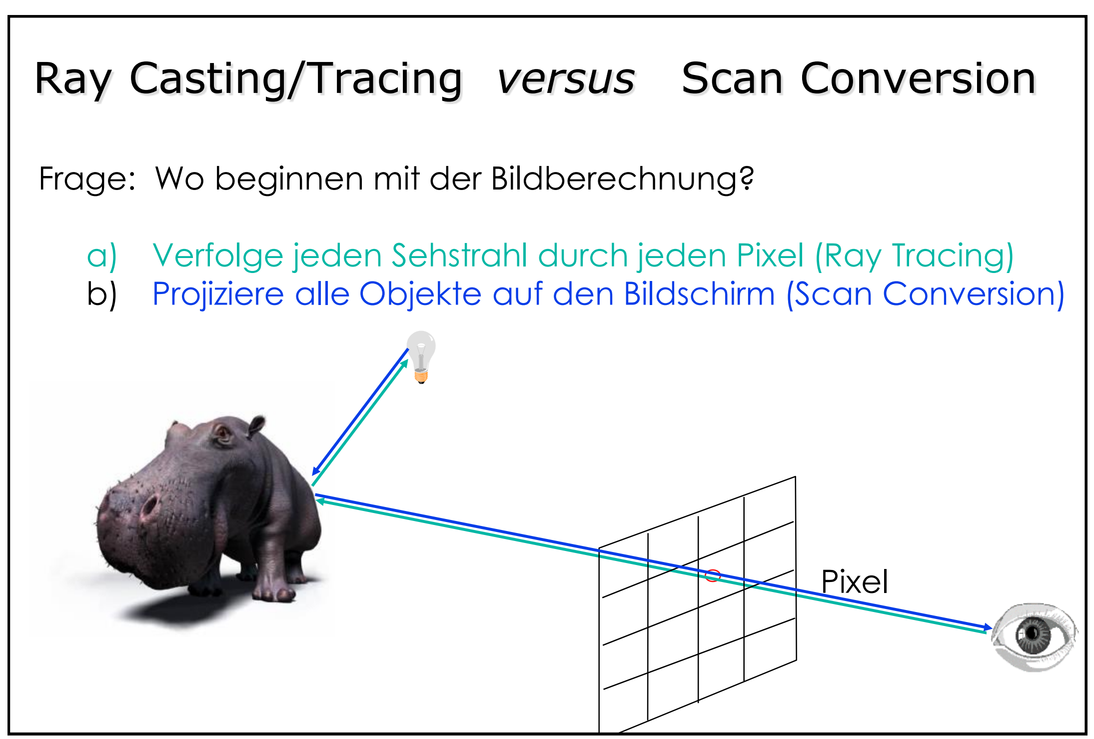
? Bilder mit 3D Geometrie

? Erlernen von Bildmodellen

Anforderungen & Methoden der Bilderzeugung

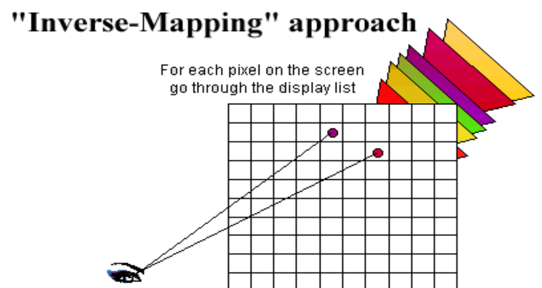
	Polygonale Objekte	+ Textur	Bildvorlagen „Image Based Rendering IBR“
Interaktion:			
- Wahl des Blickwinkels	++++	++++	++
- Wahl der Beleuchtung	++++	++	—
- Manipulation der Objekte	++++	—	—
Bildqualität:			
- Realismus	+	++	++++
- **Einfache Herstellung**	++	++	+++


Komponenten des physikalischen Renderns
Lichtquelle
Oberfläche
Reflektion
sichtbar?
Pixel
Bildschirm


Ray Casting/Tracing *versus* Scan Conversion
Frage: Wo beginnen mit der Bildberechnung?
a) Verfolge jeden Sehstrahl durch jeden Pixel (Ray Tracing)
b) Projiziere alle Objekte auf den Bildschirm (Scan Conversion)
Pixel

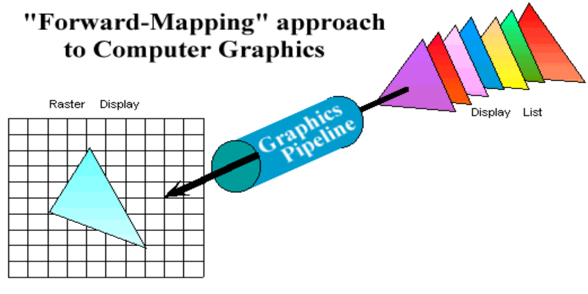
Ray Casting / Tracing

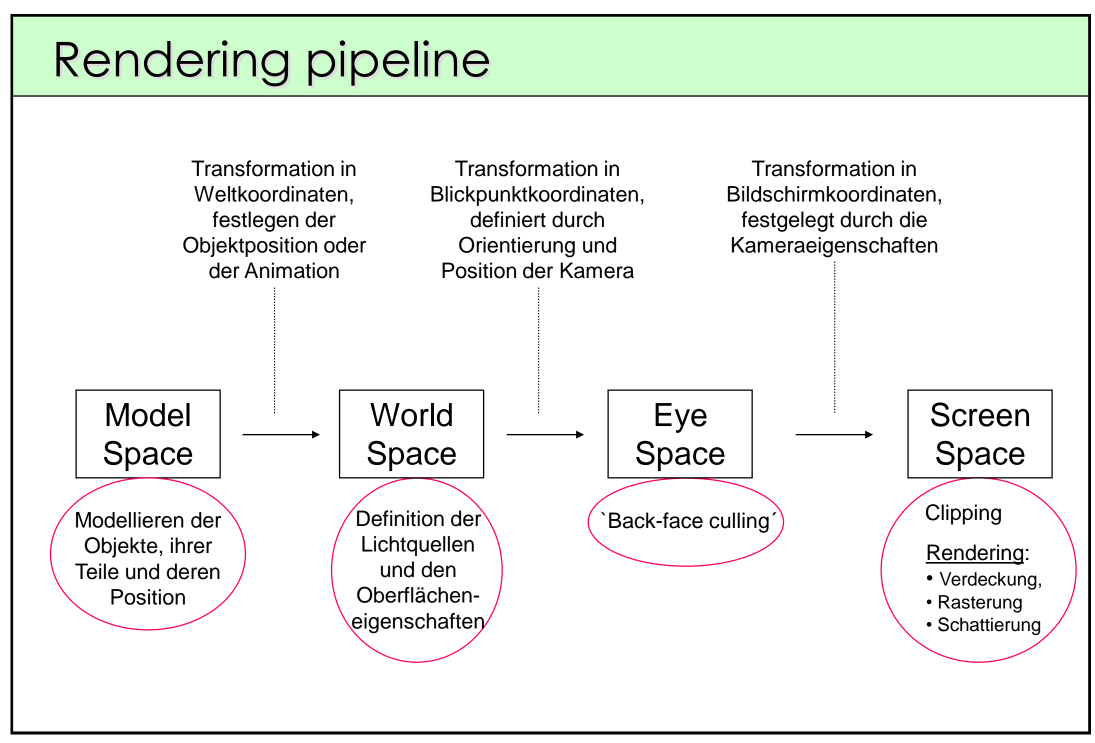
```
for every pixel, construct a ray from the eye
   for every object in the scene
         intersect ray with object
      find closest intersection with the ray
      compute normal at point of intersection
      compute color for pixel (shoot secondary rays)
```

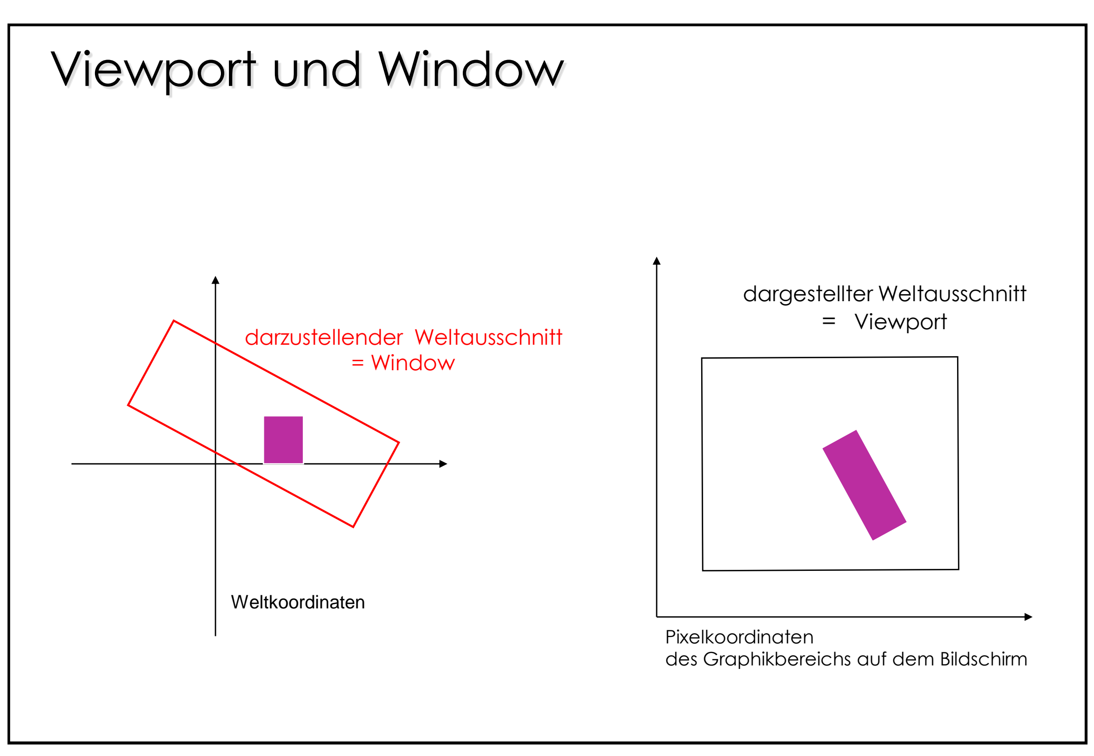


Scan Conversion – Graphics Pipeline

```
for every object in the scene
   shade the vertices
   scan convert the object to the framebuffer
   interpolate the color computed for each vertex
   remember the closest value per pixel
```

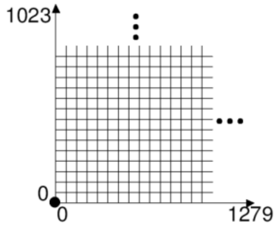



Rendering pipeline
Transformation in Weltkoordinaten, festlegen der Objektposition oder der Animation
Transformation in Blickpunktkoordinaten, definiert durch Orientierung und Position der Kamera
Transformation in Bildschirmkoordinaten, festgelegt durch die Kameraeigenschaften
Model Space
World Space
Eye Space
Screen Space
Modellieren der Objekte, ihrer Teile und deren Position
Definition der Lichtquellen und den Oberflächen-eigenschaften
`Back-face culling´
Clipping
Rendering:
• Verdeckung,
• Rasterung
• Schattierung


Viewport und Window
darzustellender Weltausschnitt = Window
Weltkoordinaten
dargestellter Weltausschnitt = Viewport
Pixelkoordinaten des Graphikbereichs auf dem Bildschirm

Rastertechnik

Viewport entspricht einem linearem Speicherarray

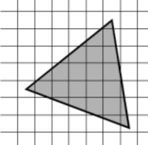
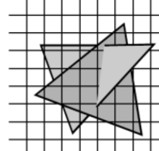
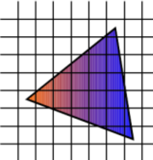
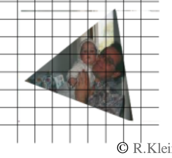


Adressierung als 2D-Array

- Startadresse
  - links oben (X11, Java AWT)
  - links unten (Open GL)

Rastertechnik

Im Rastergraphiksystem werden die einzelnen graphischen Primitiva (Dreiecke, Polygone), aus den die Szene zusammengesetzt ist, in Rasterpunkte (Pixel) zerlegt.

© R.Klein

Für jedes Pixel werden dabei zusätzlich Operationen

- zur Verdeckungsrechnung (inclusive Transparenz)
- zum Shading und
- zur Texturierung

durchgeführt.

Punkte und Linien

Vorerst besteht ein Punkt aus einem Eintrag im Bildarray (z.B. 1248x1024). Ein Punkt entspricht einem Pixel und seine Intensität ist binär (0/1).

Eine Linie ist eine Abfolge von Punkten.

PROBLEM:

1. Welche Punkte repräsentieren die Linie am besten?
2. Welches ist der schnellste Algorithmus?

Welche Punkte repräsentieren die Linie ?

Geradengleichung: $y = m * x + b$

Mit 2 ausgewählten Punkten $P(x_1, y_1), P(x_2, y_2)$

folgt

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

$$b = y_1 - m * x_1$$

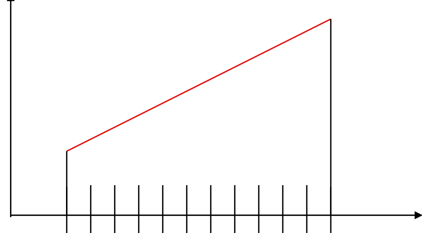
$$\Delta y = m * \Delta x$$

DDA Algorithmus (digital differential analyzer)

Angenommen die Gerade hat eine positive Steigung *m*

$$\text{mit } m < 1 \quad \Rightarrow \quad \Delta x = 1$$

$$y_{k+1} = y_k + m$$



Durch Runden der berechneten x,y Positionen, wird dann der zu setzende Pixel ausgewählt.

$$\text{falls } m \geq 1 \quad \Rightarrow \quad \Delta y = 1$$

$$x_{k+1} = x_k + \frac{1}{m}$$

DDA Algorithmus (Nachteile)

Der Algorithmus vermeidet zwar die Multiplikation wie sie in der ursprünglichen Geradengleichung vorkommt,

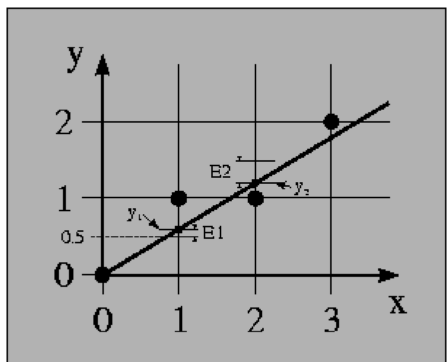
aber, Rundungsfehler bei der Berechnung von m werden akkumuliert.

Floating-point Arithmetik und Rundungsoperation!

Der Bresenham Algorithmus

Idee: Eigentlich gibt es doch gar keine große Auswahl an möglichen Pixeln.

Annahme $m<1$ und x,y sind die Anfangspixel der Geraden! Entweder ist $(x+1,y)$ oder $(x+1,y+1)$ der nächste Pixel.



Der Bresenham Algorithmus (2)

Liegt die Mitte zwischen den 2 nächsten möglichen Pixeln unterhalb oder oberhalb der Geraden?

Gerade durch Implizitefunktion darstellen!

$$F(x,y) = ax + by + c = 0$$

$F(x,y) = 0$ für alle Punkte auf der Geraden

$F(x,y) > 0$ für alle Punkte unter der Geraden

$F(x,y) \leq 0$ für alle Punkte über der Geraden

aus $y = \frac{\Delta y}{\Delta x} x + b \Rightarrow F(x,y) = \Delta y \cdot x - \Delta x \cdot y + b \cdot \Delta x = 0$

Der Bresenham Algorithmus (3)

Wir testen den Mittelpunkt $F(x_0+1, y_0+\frac{1}{2})$!

Testvariable $E = F(x_0+1, y_0+\frac{1}{2})$

(a) falls $E \le 0 \Rightarrow$ nächster Punkt (x_0+1, y_0)

(b) falls $E > 0 \Rightarrow$ nächster Punkt (x_0+1, y_0+1)

E berechnen :

$$E = \Delta y \cdot (x_0+1) - \Delta x \cdot (y_0+\frac{1}{2}) + b \cdot \Delta x$$

Der Bresenham Algorithmus (4)

Fall (a) nächster Punkt (x_0+1, y_0)

$$E_{new} = F(x_0+2, y_0+\frac{1}{2}) = \Delta y \cdot (x_0+2) - \Delta x \cdot (y_0+\frac{1}{2}) + \Delta x \cdot b$$

mit $$E_{old} = \Delta y \cdot (x_0+1) - \Delta x \cdot (y_0+\frac{1}{2}) + \Delta x \cdot b$$

$$\Rightarrow \quad E_{new} = E_{old} + \Delta y$$

Der Bresenham Algorithmus (5)

Fall (b) nächster Punkt $(x_0 + 1, y_0 + 1)$

$$E_{new} = F(x_0 + 2, y_0 + \frac{3}{2}) = \Delta y \cdot (x_0 + 2) - \Delta x \cdot (y_0 + \frac{3}{2}) + \Delta x \cdot b$$

mit $$E_{old} = \Delta y \cdot (x_0 + 1) - \Delta x \cdot (y_0 + \frac{1}{2}) + \Delta x \cdot b$$

$$\Rightarrow \quad E_{new} = E_{old} + \Delta y - \Delta x$$

Der Bresenham Algorithmus (6)

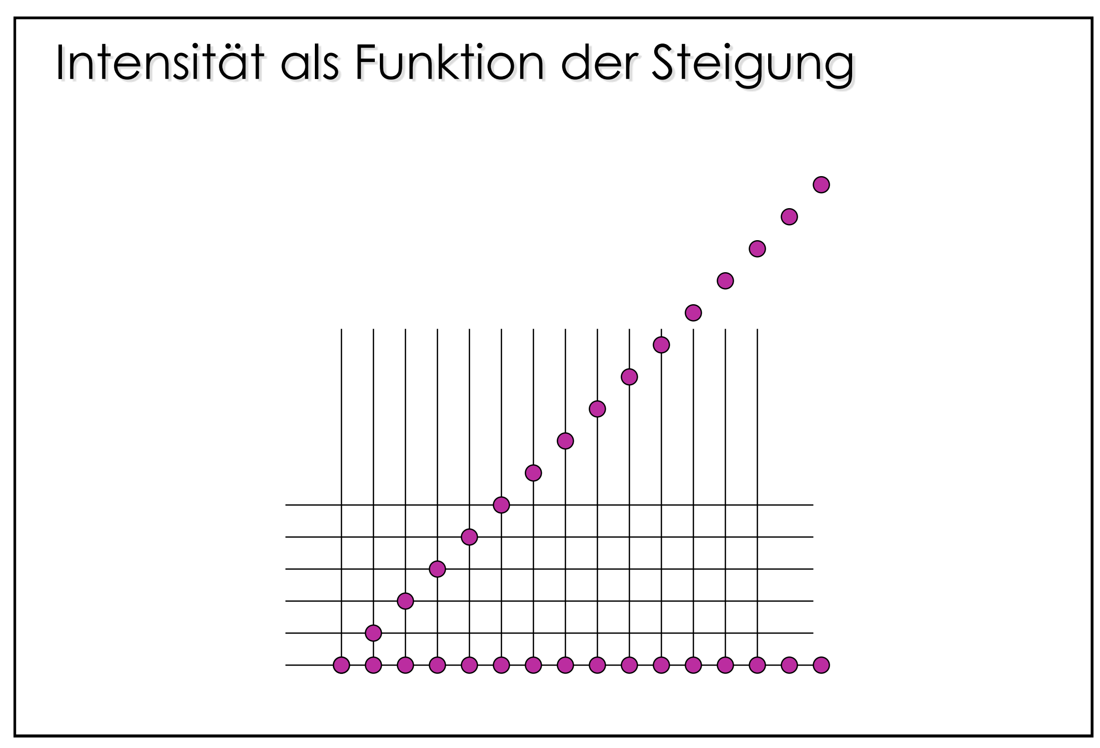
Testvariable $E_{start} = F(x_0 + 1, y_0 + \frac{1}{2})$ berechnen :

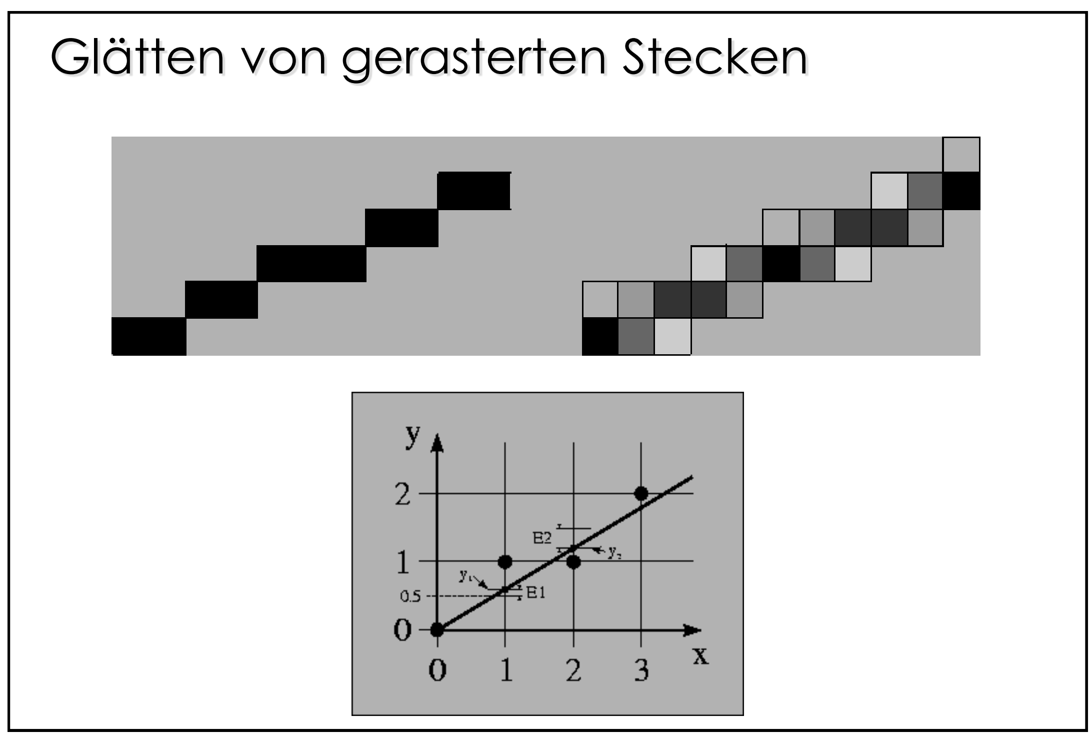
$$E_{start} = \Delta y \cdot (x_0 + 1) - \Delta x \cdot (y_0 + \frac{1}{2}) + b \cdot \Delta x$$

$$E_{start} = F(x_0, y_0) + \Delta y - \frac{\Delta x}{2}$$

$$\Rightarrow E_{start} = \Delta y - \frac{\Delta x}{2}$$

Die Division in E_{start} kann durch eine Multiplikation der Testvariable mit 2 vermieden werden, somit benötigt der Bresenham Algorithmus nur Additionen!


Intensität als Funktion der Steigung


Glätten von gerasterten Stecken
y
2
1
0.5
0
E2
E1
0
1
2
3
x

Filling
&
Clipping

Füllen von Flächen

0.) Füllen von Dreiecken (*Spezial Fall ist die Regel*)

1.) Rasterlinien Algorithmen für allg. Polygone
(*Scan-Line Algorithms*)

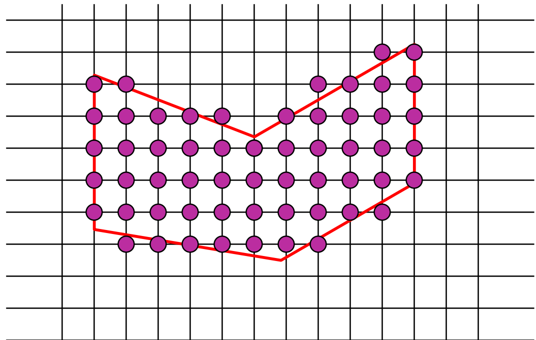
2.) Randfüll-Algorithmen

3.) Flutungs-Algorithmen

Füllen von allgemeinen Polygonen

IDEE:

1. Rand mit Bresenham-Algorithmus markieren
2. Inneres Auffüllen



Probleme: Beim Vertauschen von Innen und Außen ??

Füllen von Flächen

~~Der Bresenham-Algorithmus entscheidet, welcher Pixel zur Begrenzung am nächsten liegt!~~

Für Flächen benötigen wir ein anderes Kriterium:

Liegt ein Pixel ***innerhalb,***

außerhalb,

auf, der Begrenzung?

Pixel auf der Begrenzung?

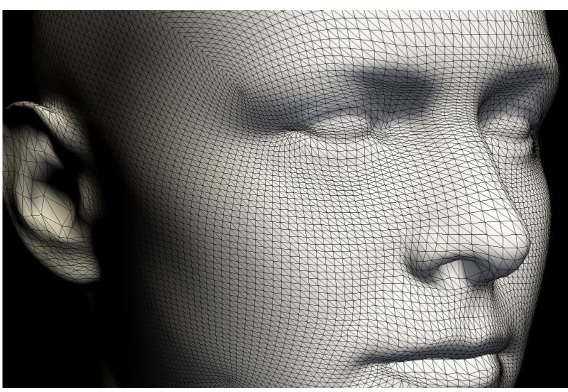
Vereinbarung: Damit bei gemeinsamen Kanten zwischen zwei benachbarten Flächen keine Problem entstehen, wird folgendes vereinbart:

Nur die Pixel ***auf*** der ***linken*** und ***unteren*** Begrenzung werden zur Fläche gezeichnet. Dies bedeutet, Punkte ***auf*** der obersten und der ganz rechten Rasterzeile werden nicht gesetzt.

Füllen von Flächen

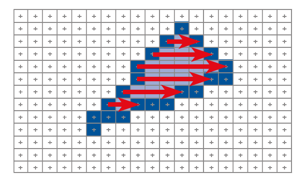
0.) Füllen von Dreiecken: Der Spezialfall ist die Regel.

Objektoberflächen werden fast immer mit Dreiecken approximiert, da hierzu effiziente Algorithmen existieren.

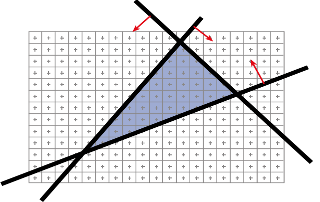


Rastern von Dreiecken (-> Übungen)

- schlau:
  Skanline Algorithmus von Kante zu Kante.



- mit roher Gewalt:
  Alle Pixel testen ob innerhalb oder außerhalb des Dreiecks.



siehe "E.S.K- Graph. Datenverarbeitung" (Kapitel 2.5.3)

Brute force solution

Für jeden Pixel

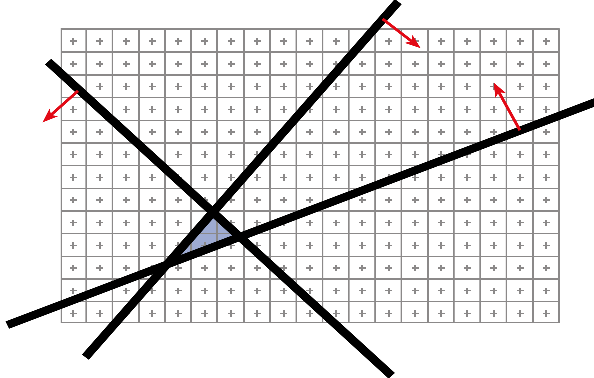
- Berechne Geradengleichung aller Kanten
- “clip” gegen das Dreieck

Problem?

Brute force solution

Für jeden Pixel

- Berechne Geradengleichung aller Kanten
- “clip” gegen das Dreieck



Problem?

Bei kleinen Dreiecken viele nutzlose Berechnungen!

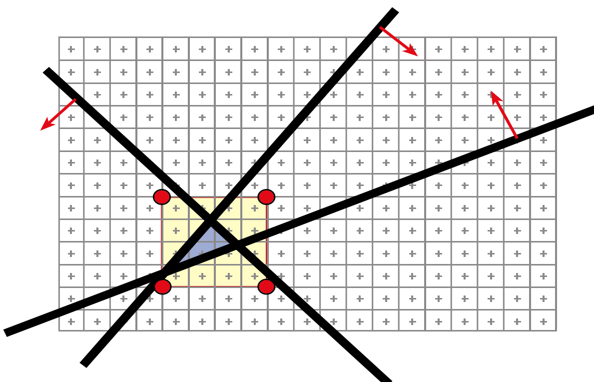
Brute force solution

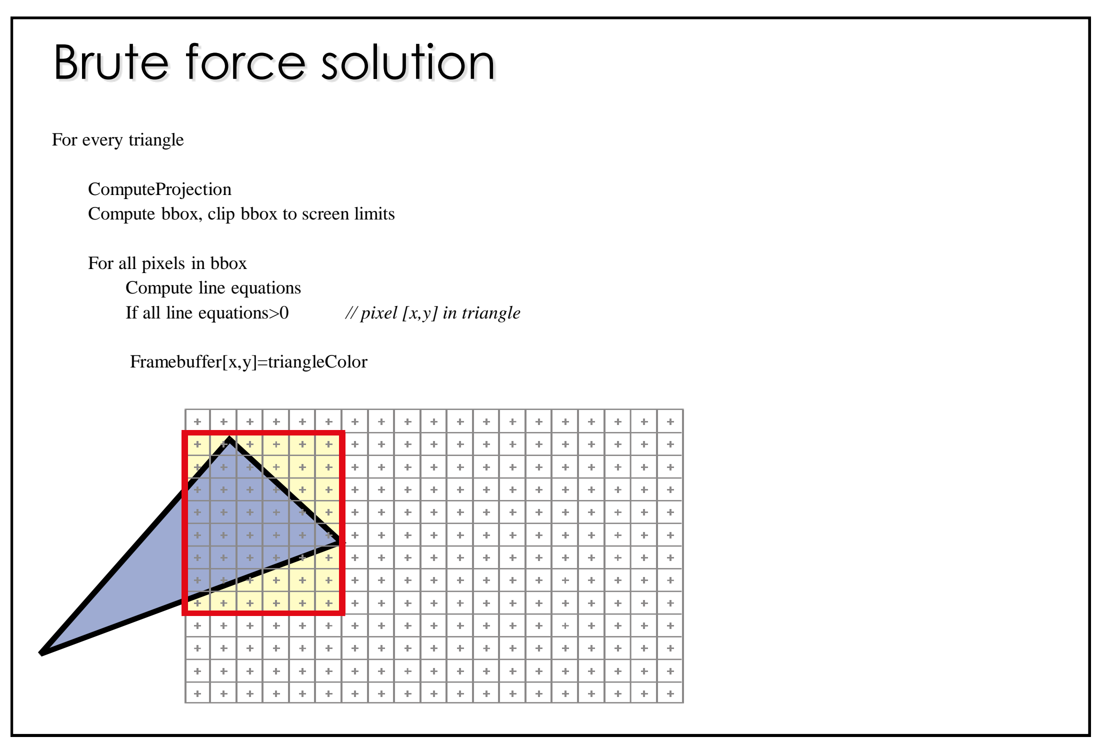
Verbesserung:

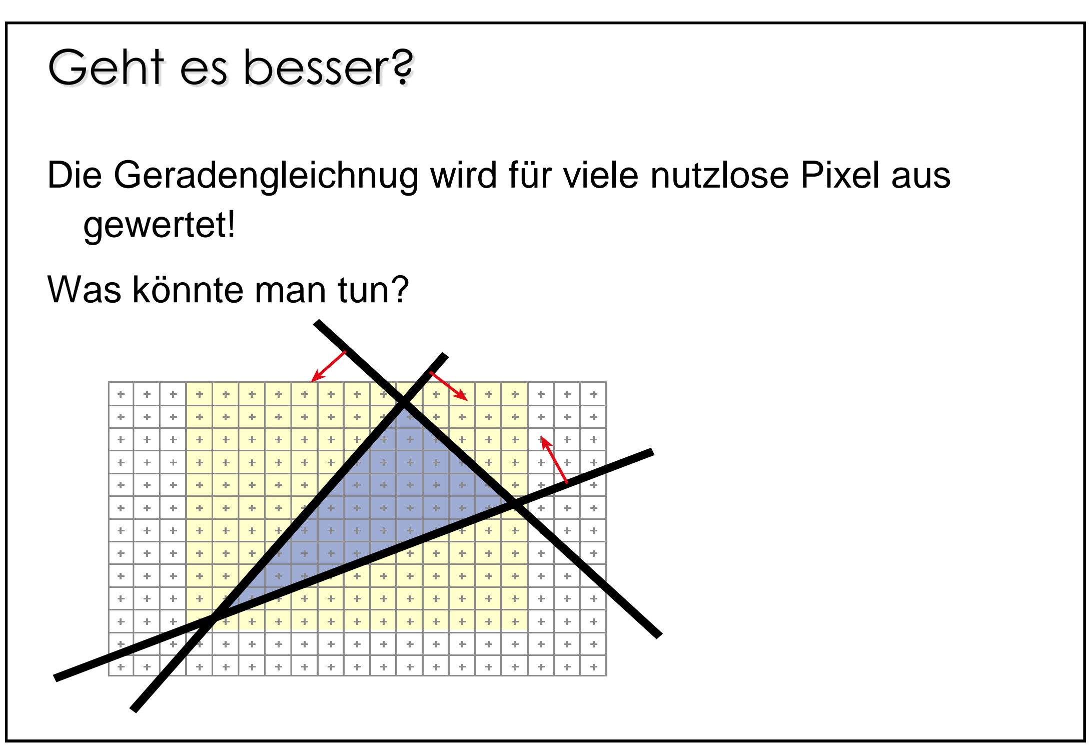
Berechnung nur für die Screen Bounding Box des Dreiecks.

Wie bestimmt man diese?

- Xmin, Xmax, Ymin, Ymax der Vertices des Dreiecks.

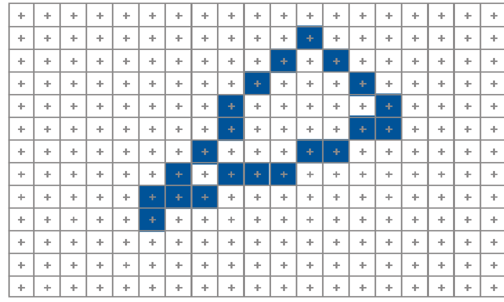



Brute force solution
For every triangle
ComputeProjection
Compute bbox, clip bbox to screen limits
For all pixels in bbox
Compute line equations
If all line equations>0
// pixel [x,y] in triangle
Framebuffer[x,y]=triangleColor


Geht es besser?
Die Geradengleichnug wird für viele nutzlose Pixel aus gewertet!
Was könnte man tun?

Scan-line Rasterung

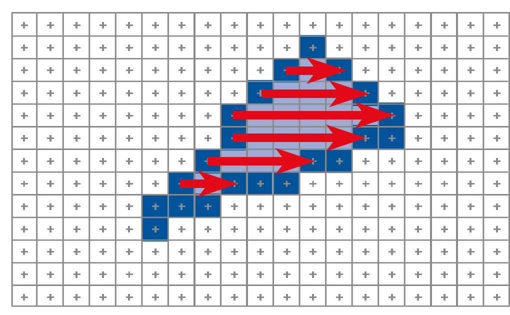
Bestimme alle aktiven Randpixel



Scan-line Rasterung

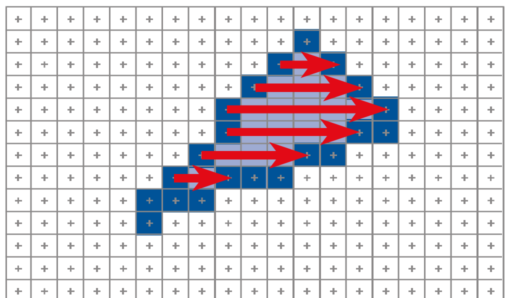
Bestimme alle aktiven Randpixel

Fülle alle Rasterlinien



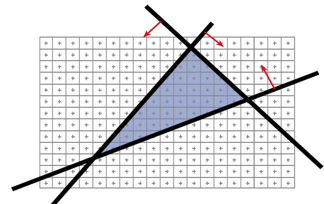
Scan line Rasterung

Benötigt einiges an Fallunterscheidungen!



Rastern von Dreiecken

- Scan-Line ist nur schlau bei großen Dreiecken
- mit roher Gewalt ist schlau bei kleinen Dreiecken.



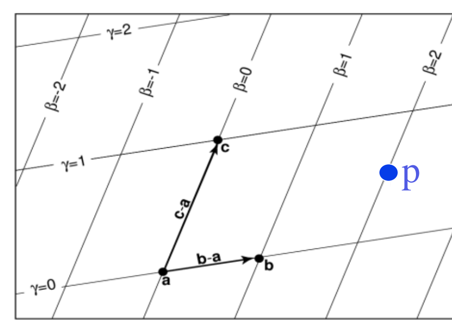
Interpolation über Dreiecke

Parametrisierung eines Dreiecks

$$p = a + \beta(b-a) + \gamma(c-a)$$

$$\Leftrightarrow$$

$$p = (1-\beta-\gamma)a + \beta b + \gamma c$$



$$\Rightarrow$$

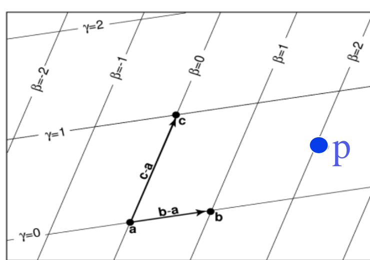
$$p(\alpha,\beta,\gamma) = \alpha a + \beta b + \gamma c \quad \text{mit} \quad \alpha+\beta+\gamma = 1$$

Baryzentrische Koordinaten

Baryzentrische Koordinaten

p liegt innerhalb des Dreiecks falls gilt:

$$0 < \alpha < 1$$
$$0 < \beta < 1$$
$$0 < \gamma < 1$$

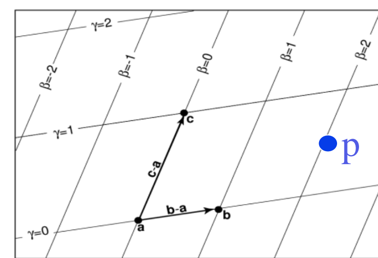


Baryzentrische Koordinaten

Berechnen:

$$p = a + \beta(b-a) + \gamma(c-a)$$

$$\begin{bmatrix} x_b - x_a & x_c - x_a \\ y_b - y_a & y_c - y_a \end{bmatrix} \begin{bmatrix} \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} x_P - x_a \\ y_P - y_a \end{bmatrix}$$

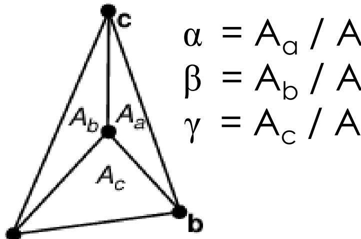


$$\gamma(x_P, y_P) = \frac{(y_a - y_b)x_P + (x_b - x_a)y_P + x_a y_b - x_b y_a}{(y_a - y_b)x_c + (x_b - x_a)y_c + x_a y_b - x_b y_a}$$

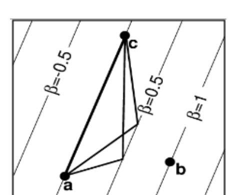
$$\beta(x_P, y_P = \frac{(y_a - y_c)x_P + (x_c - x_a)y_P + x_a y_c - x_c y_a}{(y_a - y_c)x_b + (x_c - x_a)y_b + x_a y_c - x_c y_a}$$

$$\alpha \quad = 1 - \beta - \gamma$$

Baryzentrische Koordinaten



$\alpha = A_a / A$
$\beta = A_b / A$
$\gamma = A_c / A$



Die Fläche eines Dreiecks

Eine “kurze” Formel,

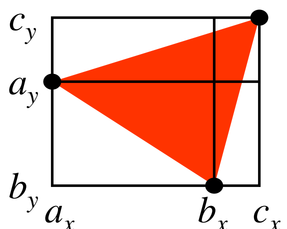
Fläche (ABC) = [$(b_x-a_x)(c_y-a_y) - (c_x-a_x)(b_y-a_y)$] / 2

Wie kommt sie zustande?

$$Fläche(ABC) = \frac{1}{2}\begin{vmatrix} a_x & b_x & c_x \\ a_y & b_y & c_y \\ 1 & 1 & 1 \end{vmatrix}$$

$$= \left(\begin{vmatrix} b_x & c_x \\ b_y & c_y \end{vmatrix} + \begin{vmatrix} c_x & a_x \\ c_y & a_y \end{vmatrix} + \begin{vmatrix} a_x & b_x \\ a_y & b_y \end{vmatrix} \right) / 2$$

$$= (b_x c_y - c_x b_y + c_x a_y - a_x c_y + a_x b_y - a_y b_x) / 2$$

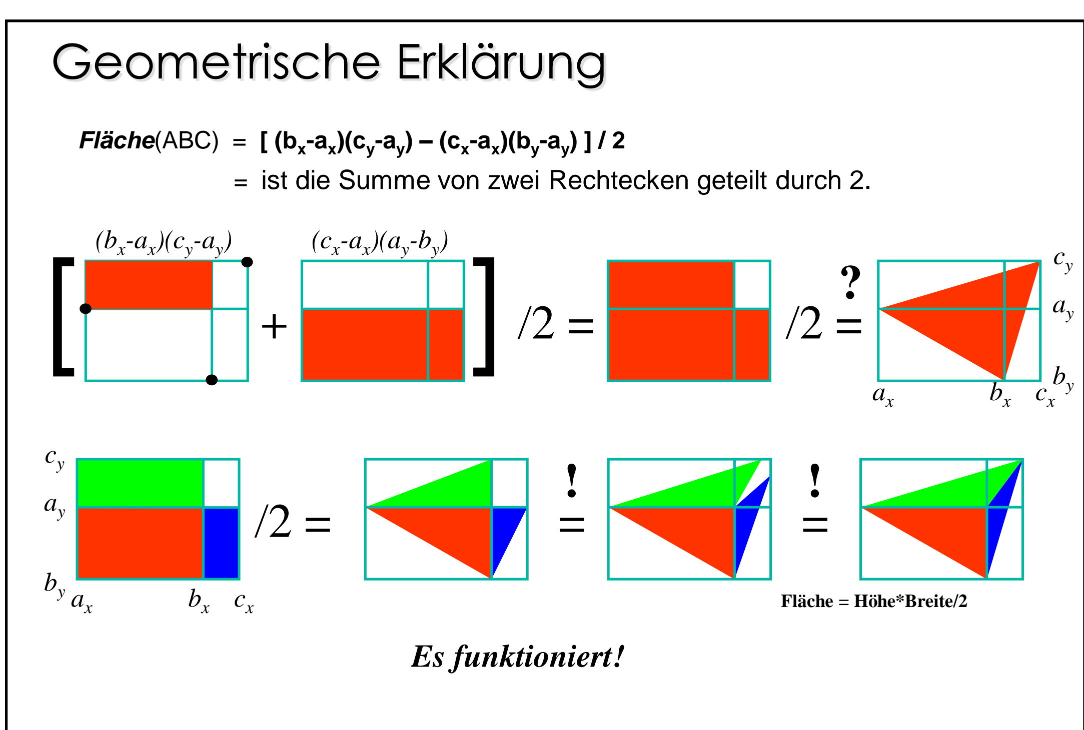


Die kürzere ist viel effizienter, weniger Multiplikationen!

Geometrische Erklärung

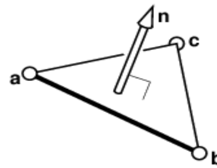
Fläche(ABC) = **[$(b_x-a_x)(c_y-a_y) - (c_x-a_x)(b_y-a_y)$] / 2**

= ist die Summe von zwei Rechtecken geteilt durch 2.



Es funktioniert!

Baryzentrische Koordinaten 3D



$$p = (1-\beta-\gamma)a + \beta b + \gamma c$$

$\alpha = A_a / A$
$\beta = A_b / A$
$\gamma = A_c / A$

$$\vec{n} = (\vec{b}-\vec{a}) \times (\vec{c}-\vec{a}) \qquad \text{Fläche} = \tfrac{1}{2} \left\| (\vec{b}-\vec{a}) \times (\vec{c}-\vec{a}) \right\|$$

mit $\vec{n}_a = (\vec{c}-\vec{b}) \times (\vec{p}-\vec{b})$

$\vec{n}_b = (\vec{a}-\vec{c}) \times (\vec{p}-\vec{c})$

$\vec{n}_c = (\vec{b}-\vec{a}) \times (\vec{p}-\vec{a})$

Testen ob n_a und n parallel sind: $n \cdot n_a = \|n\| \cdot \|n_a\| \cos(\alpha)$

$$\alpha = \frac{n \cdot n_a}{\|n\|^2} \qquad \beta = \frac{n \cdot n_b}{\|n\|^2} \qquad \gamma = \frac{n \cdot n_c}{\|n\|^2}$$

Interpolation über Dreiecke

Bilineare Interpolation über ein Dreieck (P_0, P_1, P_2)

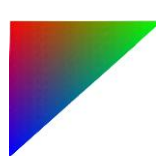
Interpolation der Position

$$P(\beta,\gamma) = (1-\beta-\gamma)P_0 + \beta P_1 + \gamma P_2$$

$$\text{mit} \quad P(0,0) = P_0, \quad P(1,0) = P_1 \text{ und } P(0,1) = P_2$$

Interpolation der Farben (f_0, f_1, f_2):

$$f(\beta,\gamma) = (1-\beta-\gamma)f_0 + \beta f_1 + \gamma f_2$$



oder der Textur, der Normalen (!!) und, und ....

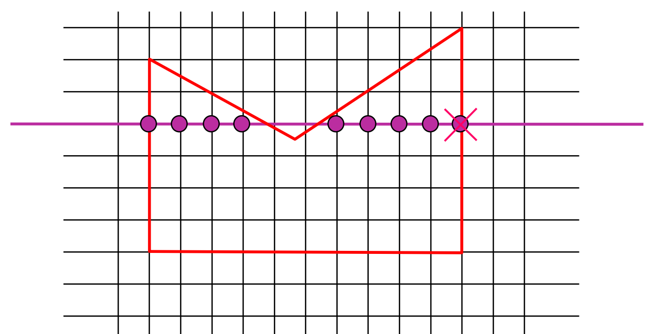
Scan-Line Polygon Fill Algorithm

Für jede Rasterzeile:

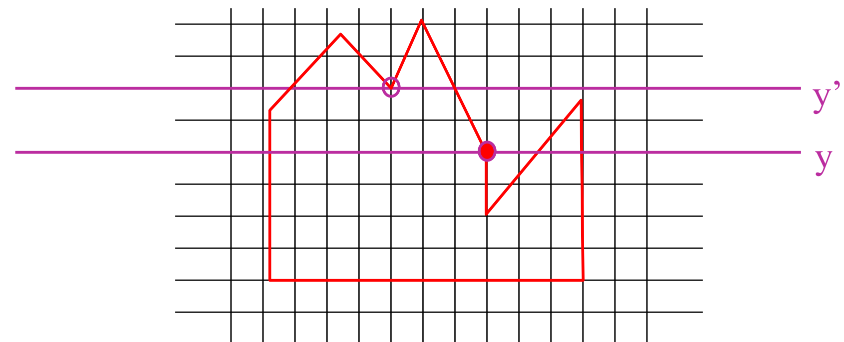
1.) Bestimme alle Schnittpunkte mit dem Polygonzug.

2.) Sortiere die Schnittpunkte nach aufsteigendem x-Wert.

3.) Fülle die Segmente zwischen den entsprechenden Paaren.



Scan-Line Polygon Fill Algorithm (2)



Rasterzeile ***y'*** hat 4 Schnittpunkte (x_0, x_1, x_1, x_2) ------->
2 innere Strecken (x_0, x_1), (x_1, x_2) .

Rasterzeile ***y*** hat 5 Schnittpunkte $(x_0, x_1, x_1, x_2, x_3)$ --->
auch *2* innere Strecken ? (x_0, x_1), (x_2, x_3) .

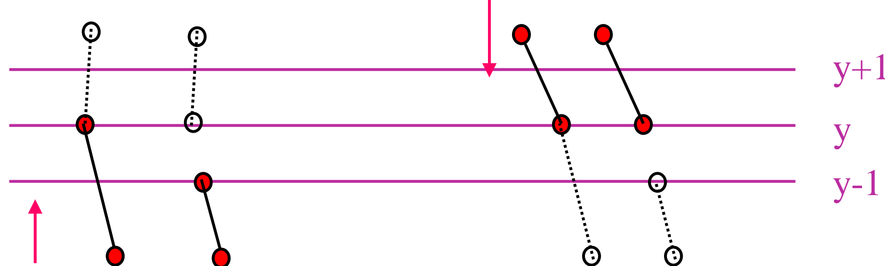
Scan-Line Polygon Fill Algorithm (3)

Erstellen einer Randliste *(**a**ctive-**e**dge **t**able AET):* (1)

a) eliminiere ungültige Schnittpunkte.

b) ordne einzelne Stecken zu Polygonzug

c) für alle monoton steigende und fallende Kantenpaare, verkürze die untere Kante.



Scan-Line Polygon Fill Algorithm (4)

Erstellen einer Randliste *(AET):* (2)

b) Berechnung der Schnittstellen entlang einer Kante.

$$m = \frac{y_{k+1} - y_k}{x_{k+1} - x_k} \quad \text{und} \quad y_{k+1} - y_k = 1$$

$$\Rightarrow x_{k+1} = x_k + \frac{1}{m}$$

Scan-Line Polygon Fill Algorithm (5)

Effiziente Berechnung der Schnittstellen entlang einer Kante.

Für die *k-te* Schnittstelle zwischen Rasterline und Kante berechnet sich

$$x_{aktuell}, y_{aktuell}$$

$$\Rightarrow \; y_{aktuell} = k + y_0$$

$$\Rightarrow x_{aktuell} = x_0 + \frac{k}{m}$$

Da die *x* -Koordinate der Schnittstelle immer ganzzahlig sein muss, kann die reelwertige Quotientenbildung *k/m* vermieden werden. -------->

Scan-Line Polygon Fill Algorithm (6)

Es ist nur interessant wenn der gebrochenrationale Anteil von *1/m* zum Überlauf führt.

$\frac{1}{m} = \frac{\Delta x}{\Delta y}$ wird nicht berechent!

Für jede Rasterzeile

wird Δx akkumuliert zu $x_{accu} \mathrel{+}= \Delta x$

und mit Δy verglichen.

Bei $x_{accu} > \Delta y$

setzte $x_{neu} = x_{alt} + 1$ und $x_{accu} \mathrel{-}= \Delta y$

Scan-Line Polygon Fill Algorithm (7)

Erstellen einer Randliste *(AET):*

2.) Erstelle Kantenliste, sortiert nach aufsteigendem y-Wert

Raster-
linie

y_c → y_B | x_C | $1/m_{CB}$

y_D → $y_{C'}$ | x_D | $1/m_{DC}$ → y_E | x_D | $1/m_{DE}$

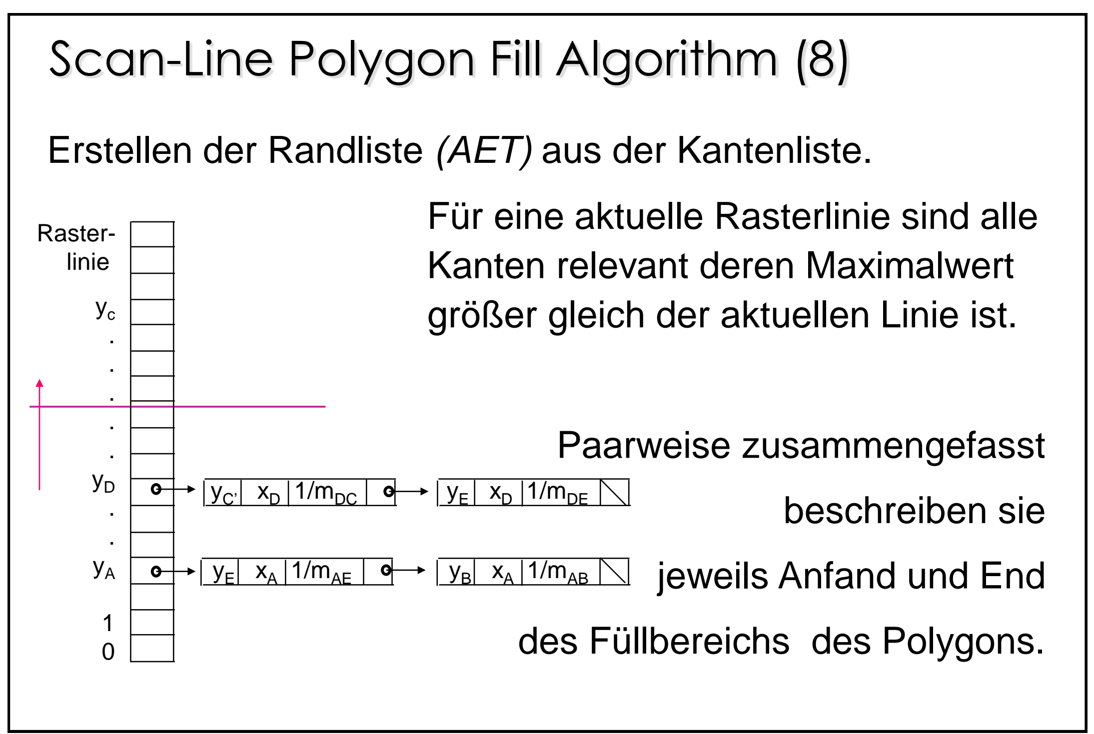
y_A → y_E | x_A | $1/m_{AE}$ → y_B | x_A | $1/m_{AB}$

1

0

y_{max} x_{min}

B, C, C', E, D, A



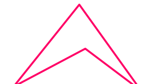
Scan-Line Polygon Fill Algorithm (9)

Alternative Vorgehensweise:

Zerlege das gesamte Polygon in Dreiecke und fülle die einzelnen Dreiecke.

Vorteil: Scan-Line Algorithmen für Dreiecke sind extrem einfach.

Nachteil: Die Zerlegung konkaver Polygone in Dreiecke ist sehr aufwändig.

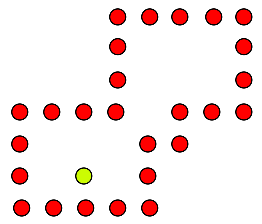


Zerlegung in Dreiecke ist jedoch bei vielen Anwendungen gegeben.

Randfüll Algorithmen

Bei nicht **parametrisierbaren Rändern** kann die Randbestimmung für eine Scan-Line Algorithmus sehr aufwändig sein!

--> Auffüllen der Fläche von innen: Starte bei beliebigem inneren Punkt und fülle die Fläche von innen nach außen!



Wie viele Nachbarpunkte sind zu beachten?

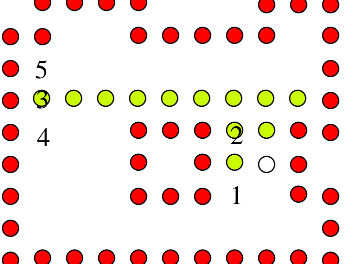
Rekursiver Randfüll Algorithmus (4, 8 Punkte)

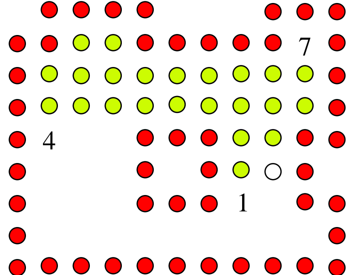
```
void boundaryfill4 ( int x, int y, int fill, int boundary)
{
   int current;

   current  = getpixel( x, y );
   if  ( ( current != boundary ) && (current != fill ) ) {
      setColor ( fill );
      setPixel  (  x, y );
      boundaryfill4 ( x+1,    y, fill, boundary);
      boundaryfill4 ( x -1,    y, fill, boundary);
      boundaryfill4 ( x ,   y+1, fill, boundary);
      boundaryfill4 ( x ,   y -1, fill, boundary);
  }
 }
```

Rekursive Randfüll Algorithmen

Nachteil: Enorme Buchhaltungskosten!

----> Durch Kombination mit Rasterlinien Algorithmen lassen sich die Buchhaltungskosten auf die Anfangspunkte neuer Rasterlinien beschränken.





Flutungs Algorithmen

Flächen die nicht durch den Rand definiert sind, können mit Flutungsalgorithmen verändert werden.

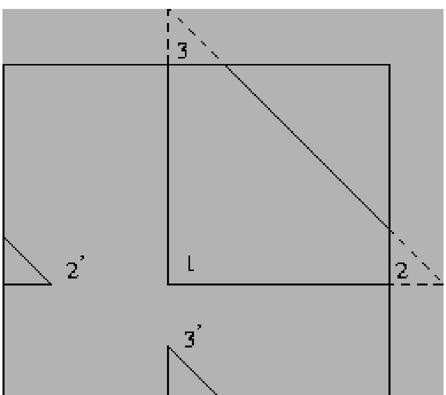
Flutungsalgorithmen sind den Randfüllalgorithmen ähnlich.

```
void floodFill4 ( int x, int y, int fillColor, int oldColor)
{
if  ( getPixel (x,y) == oldColor) {
      setColor ( fillColor);
      setPixel  (  x, y );
      floodFill4  ( x+1,     y, fill, boundary);
      floodFilll4 ( x -1,    y, fill, boundary);
      floodFill4  ( x ,   y+1, fill, boundary);
      floodFill4  ( x ,   y -1, fill, boundary);
  }
 }
```

Clipping

Clipping

Wraparound



Überlauf des Bildspeichers ergibt meist Störungen des eigentlichen Bildes.

Clipping

Eine einfache Lösung:

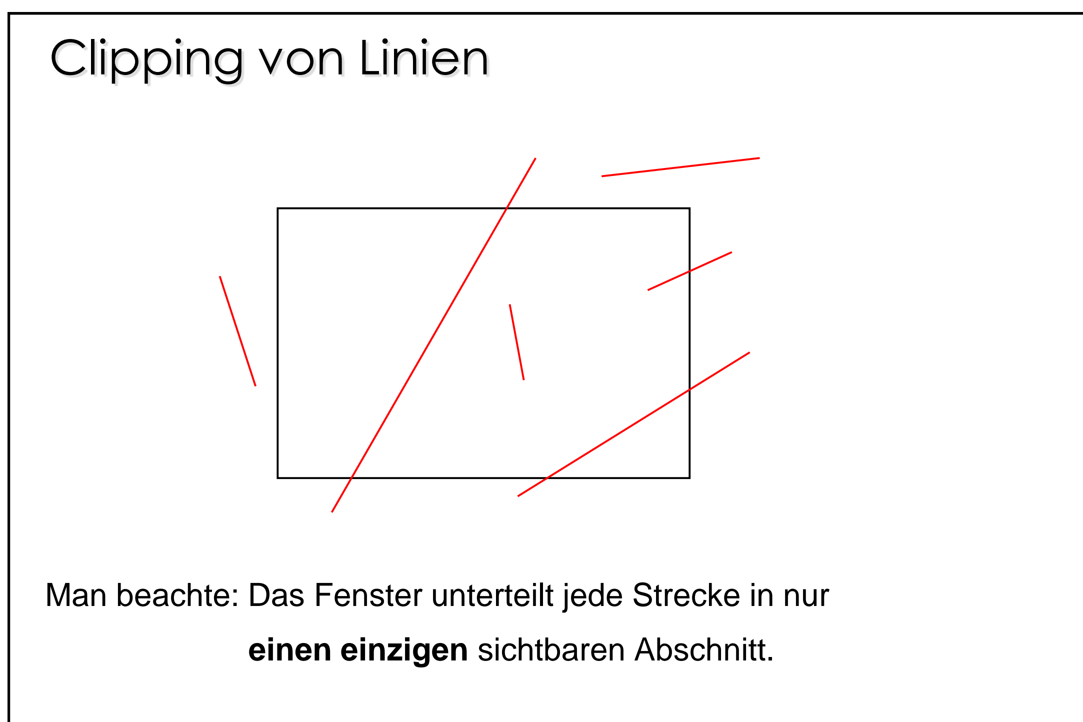
Zeichne das Bild durch isolierte Punkte an P(*X,Y)*

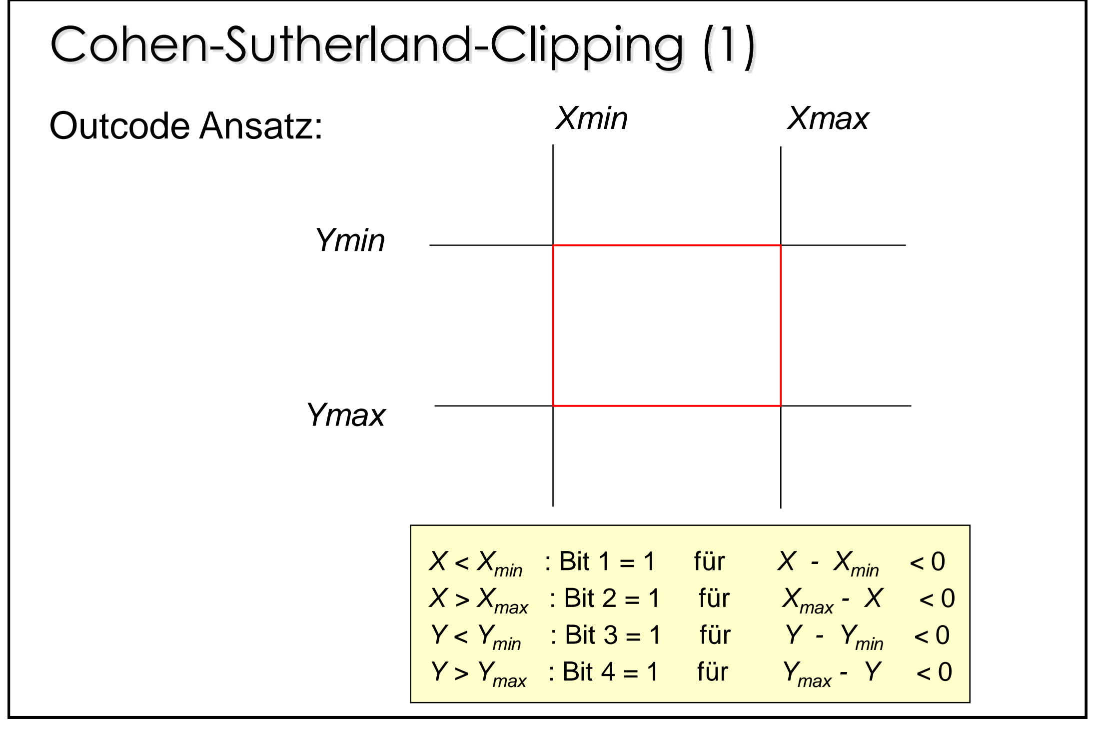
wenn

$$X_{Min} \leq X \leq X_{Max} \quad \text{und} \quad Y_{Min} \leq Y \leq Y_{Max}$$

Der Vergleich aller möglichen Bildpunktkoordinaten *(X,Y)* mit den Fensterkoordinaten *Xmin, Xmax, Ymin,Ymax* ist nicht effizient.

(Es kann sehr aufwendig sein Bildpunktkoordinaten zu berechnen!!!)


Clipping von Linien
Man beachte: Das Fenster unterteilt jede Strecke in nur
einen einzigen sichtbaren Abschnitt.


Cohen-Sutherland-Clipping (1)
Outcode Ansatz:
Xmin
Xmax
Ymin
Ymax
$X < X_{min}$: Bit 1 = 1 für $X - X_{min} < 0$
$X > X_{max}$: Bit 2 = 1 für $X_{max} - X < 0$
$Y < Y_{min}$: Bit 3 = 1 für $Y - Y_{min} < 0$
$Y > Y_{max}$: Bit 4 = 1 für $Y_{max} - Y < 0$

Cohen-Sutherland-Clipping (2)

1001	1000	1010
0001	0000	0010
0101	0100	0110

1. Stufe

Ein Vector liegt völlig:

- innerhalb des Fensters, wenn der Code für alle Endpunkte 0000 ist.
- außerhalb des Fensters, wenn der Durchschnitt (logisches UND) beider Endpunktkodierungen ungleich Null ist.

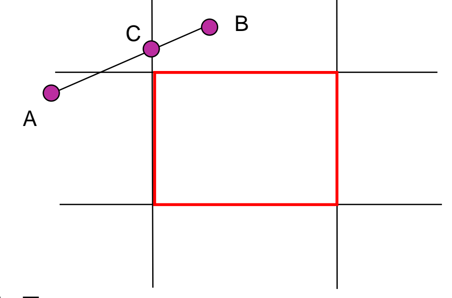
Falls beides nicht zutrifft, geht der Algorithmus in die **2.Stufe . ->**

Cohen-Sutherland-Clipping (3)

A: 0001

B: 1000

A & B : 0000



---> Berechne Schnittpunkte mit Fenstergrenzen

für X_{min} ergibt sich Schnittpunt C

Teste Strecke AC und CB

---> Outcode ist für beide Strecken != 0000

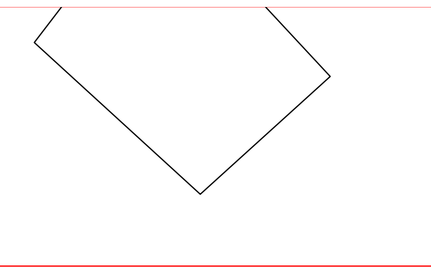
====> Beide Stecken liegen entweder ganz oberhalb oder ganz neben dem Fenster ----- clippen!

Clipping von Polygonen

Ansatz:

Clipping von Vektoren

ist nicht ausreichend !!!!



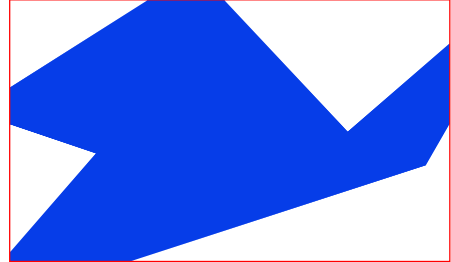
Um das Polygon zu füllen

ist ein geschlossener Polygonzug nötig!

Sutherland-Hodgeman Polygon Clipping

1.) Die gesamte Polygonbegrenzung wird jeweils an den Grenzen $X_{min}, X_{max}, Y_{min}, Y_{max}$, 'geclipped'.

2.) Es werden alle Fensterbegrenzungen nacheinander abgearbeitet.



Wie werden neu Punkte zur Polygonbegrenzung eingefügt?

Es gibt 4 Regeln! =====>

Sutherland-Hodgeman Polygon Clipping (2)

Für jeden Vektor von *A -> B* wird entschieden ob man die Grenze des erlaubten Bereichs überschreitet.

Für *A -> B*

außen --> innen : speichere S_{AB}, *B*
innen --> außen : S_{AB}
innen --> innen : B
außen --> außen : -----