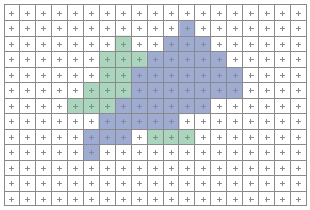
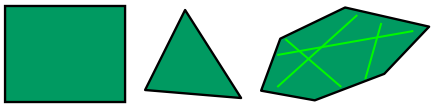
Computer–Graphik I
Polygon Scan Conversion



G. Zachmann
Clausthal University, Germany
zach@in.tu-clausthal.de

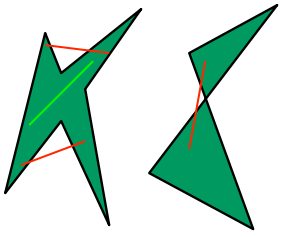
Klassifikation der Polygone

Konvex

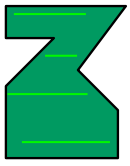


Für jedes Punktepaar in einem konvexen Polygon liegt die Verbindung auch innerhalb des Polygons

Konkav



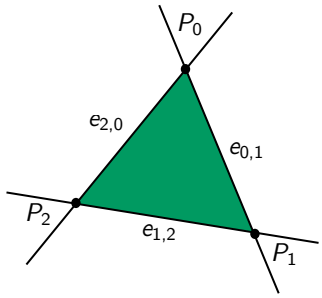
Horizontal Konvex



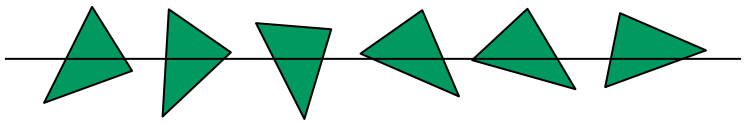
Selbe Definition, gilt hier aber nur für Punkte auf der selben horizontalen Linie

Dreiecke

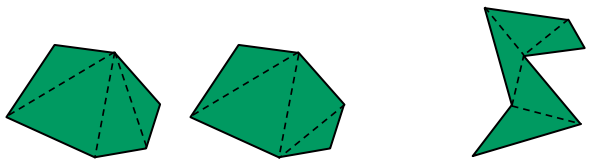
- Dreiecke sind besondere Polygone
- 3D Dreiecke sind immer eben
  - 3 Punkte beschreiben eine Ebene
  - Mit weniger als 3 Punkten kann man keine Fläche beschreiben
- Dreieck = 2D–Simplex ("einfachstes" geom. Objekt, das echt 2-dim. ist)
- Somit sind Dreiecke sehr einfach (sowohl mathematisch wie auch geometrisch)



- Dreiecke sind immer konvex → egal wie man ein Dreieck dreht, es gibt nur ein Schnittintervall (= *Span)* für jede Gerade (*Scanline)*



- Satz (o. Bew.):
  Jedes beliebige Polygon kann in eine Menge von Dreiecken zerlegt werden ("Triangulierung")

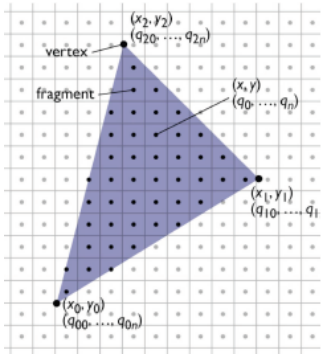


Konvex Konkav

- Der Algorithmus zum Rasterisieren erzielt aus den Eigenschaften von Dreiecken Vorteile
- Deswegen
  - ist die Graphik-Hardware optimiert für Dreiecke
  - unterteilen viele Graphikkarten Polygone in Dreiecke
  - Einige Systeme rendern eine Linie, indem sie 2 schmale Dreiecke rendern

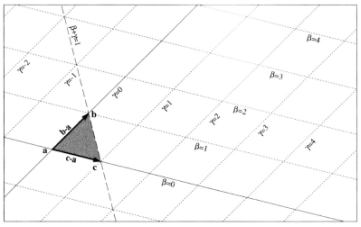
Rasterisierung von Dreiecken

- Eingabe:
  - Drei 2D Punkte (Eckpunkte im Framebuffer / "Pixel-Raum"):
    (x_0, y_0); (x_1, y_1); (x_2, y_2)
  - Mit Attributen q für jeden Eckpunkt, z.B. Farbe
- Ausgabe:
  - Ganzzahlige Pixel-Koordinaten (x, y)
  - Interpolierte Parameterwerte q_{xy}



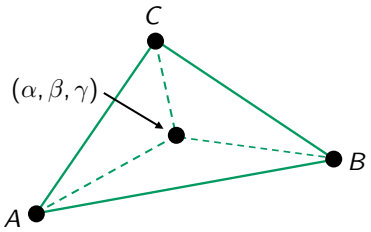
Erinnerung

- Baryzentrische Koordinaten



- Test ob Punkt im Dreieck liegt:

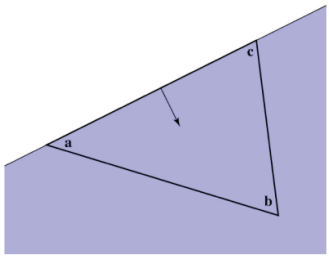
$$\alpha > 0 \ \wedge \ \beta > 0 \ \wedge \ \gamma > 0$$

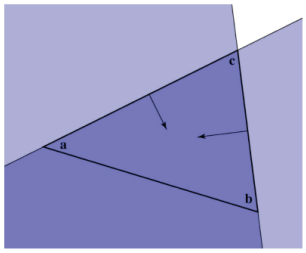


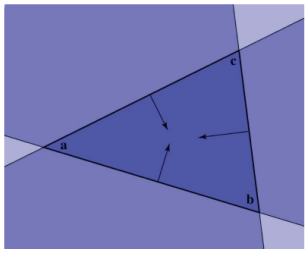
Alternative Betrachtungsweise

- In der Ebene ist ein Dreieck die Schnittmenge von 3 Halbebenen

$$(x - c)\cdot(a - c)^{\perp} > 0$$
$$(x - b)\cdot(c - b)^{\perp} > 0$$
$$(x - a)\cdot(b - a)^{\perp} > 0$$



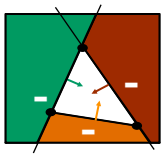




Algorithmus von Pineda [1988]

- Idee:
  - Berechne baryzentrische Koordinate für alle Pixel(-mittelpunkte)

$$\alpha = F_{BC}(x) = \frac{\mathbf{n}_c \cdot (X - B)}{\mathbf{n}_c \cdot (C - B)} \quad , \quad \beta = \ldots$$

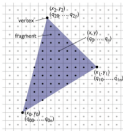


  - Zeichne Pixel, falls innerhalb des Dreiecks
  - Setze interpolierte Farbe für Pixel:

$$C = \alpha C_A + \beta C_B + \gamma C_C$$

- Algo:

```
for y = ymin ... ymax:
   for x = xmin ... xmax:
      berechne α, β, γ
      if α > 0 and β > 0 and γ > 0:
         c = αcA + βcB + γcC
         zeichne Pixel (x,y) mit Farbe c
```



wobei $x_{min}, x_{max}, y_{min}, y_{max}$ = Bounding-Box von *A, B, C*

Optimierung

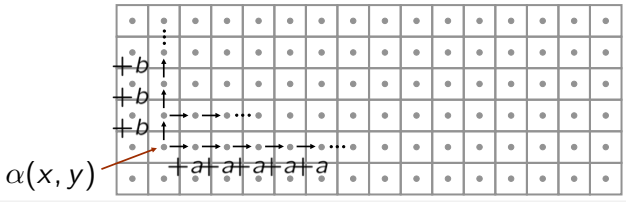
- Beobachtung: α ist eine lineare (eigtl. affine) Funktion in der Ebene, m.a.W., α hat die Form

$$\alpha = ax + by + c$$

  Dito für β, γ

- Lineare Funktionen können sehr effizient inkrementell auf einem Gitter ausgewertet werden:

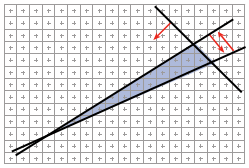
$$\alpha(x+1, y) = \alpha(x, y) + a$$
$$\alpha(x, y+1) = \alpha(x, y) + b$$

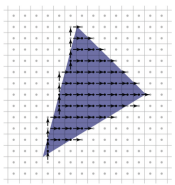


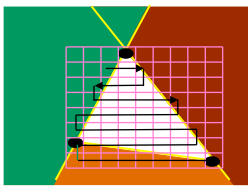
```
linEval(xl, xh, yl, yh, cx, cy, ck):

# setup
compute a, b, c ...
qRow = a*xl + b*yl + c

# traversal
for y = ymin ... ymax:
    qPix = qRow
    for x = xmin ... xmax:
        draw(x, y, qPix)
        qPix += a
    qRow += b
```

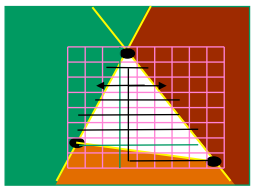
a = .005; b = .005 c = 0
(Bildgröße 100x100)

- Problem: wenn das Dreieck lang und schmal ist, dann werden viele unnötige Berechnungen durchgeführt
- Erinnerung: Dreieck ist konvex
  - Folge: wenn man in der x–Schleife einmal ein Pixel außerhalb erreicht, dann sind alle folgenden in dieser Scanline auch außerhalb





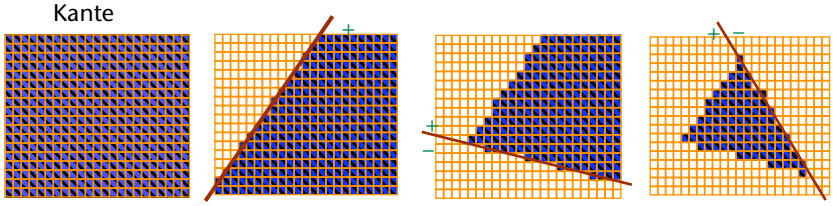




- Weiterer Vorteil des Algorithmus von Pineda: lässt sich relativ leicht parallelisieren

Kurzer Exkurs: Die Pixel-Planes-Architektur

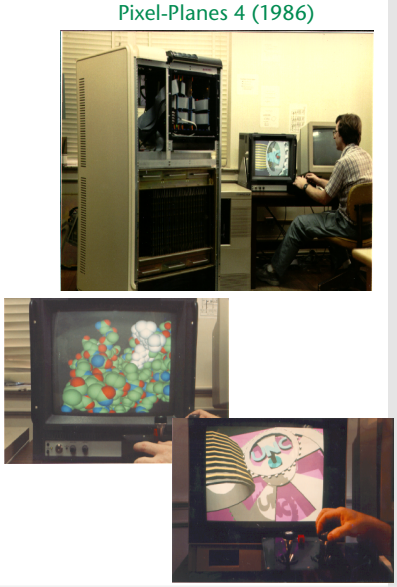
- Eine Geschichte mit *"sad end"* ...
- Die Idee:
  - Die Berechnung pro Pixel ist extrem einfach, nämlich Auswertung einer linearen Funktion Ax + By + C
  - Also: baue Framebuffer, in dem jedes Pixel ein einfacher Prozessor ist, der solch eine Gleichung für "seine" Koordinaten auswerten kann! (*"processor per pixel"*)
  - Betrachte die 3 Kanten der Reihe nach
  - Lade alle Prozessoren gleichzeitig mit den Koeff. A,B,C der aktuellen Kante



Alle Prozessoren an | Lade Kante 1 | Lade Kante 2 | Lade Kante 3

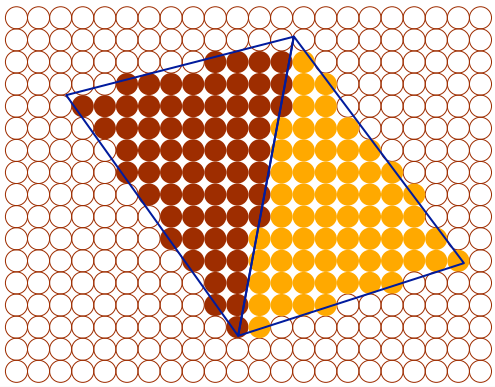
- Features:
  - Full-size (512 by 512 pixel) prototype
  - Used 2048 enhanced memory ICs
  - 1 Geometry Processor
  - 72 bits per pixel
- Performance:
  - 35K triangles/sec
  - Kugeln als Primitive
  - CSG
  - Schatten
- "Lessons Learned":
  - Dreiecke sind klein, daher viele Proc idle
  - Für noch mehr Performance braucht man auch auf dem Geometrie-Level Parallelisierung

Pixel-Planes 4 (1986)

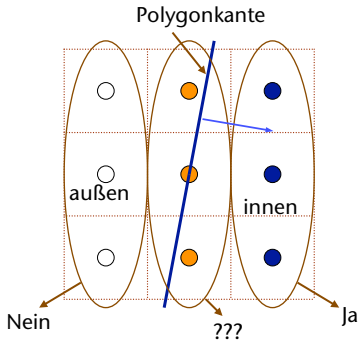


Vorsicht bei angrenzenden Polygonen

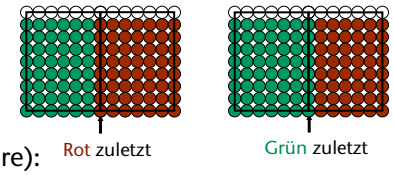
- Behandle angrenzende Polygone korrekt!
  - Vermeide Risse
  - Vermeide Überschneidungen
  - Unabhängig von der Zeichenreihenfolge



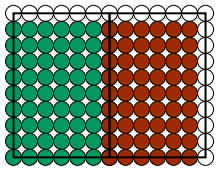
- Pixel vollständig im Polygon → wird gezeichnet
- Pixel zum Teil im Polygon → ... ?
- Vereinbarung (für den Moment): zeichne nur die Pixel, deren Zentren im Inneren des Polygons liegen
- Problem, falls Zentrum des Pixels genau auf der Ecke des Polygons liegt :
  - Nicht zeichnen → Loch
  - Zeichnen → wird möglicherweise 2x gezeichnet (ergibt sog. "*z flickering*" u.a. Artefakte)



- Problem beim 2x Zeichnen:
  - Das zuletzt gezeichnete Polygon "gewinnt"

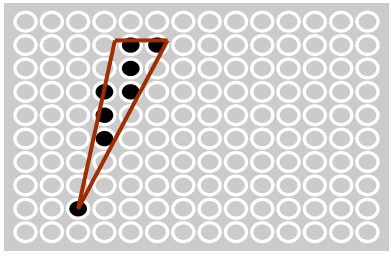


- Mögliche Lösung (gibt noch andere):
  - Ein Begrenzungspixel (dessen Zentrum genau auf der Kante liegt) gehört nicht zu einem Primitiv, wenn das Primitiv links bzw. unterhalb des durch die Kanten aufgespannte Halbraumes liegt (erkennt man an der Normale)
  - Verfahre bei konvexen Polygonen genau wie bei Rechtecken
- Folgerungen für Rechtecke:
  - Spans lassen das rechteste Pixel weg (falls direkt auf Kante)
  - Bei jedem Polygon fehlt oberster Span (falls direkt auf Kante)

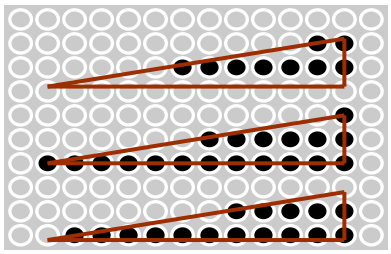


Weiteres Problem

- Sogenannte *"Slivers"*:

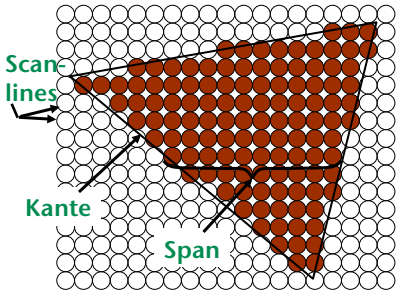


- *Moving Slivers*:



Das allgemeine Konzept der Scan Conversion

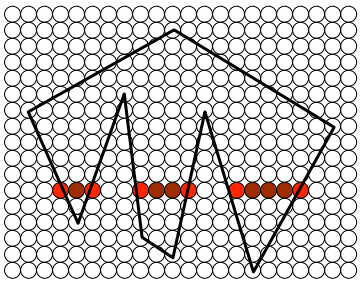
- Gegeben: beliebiges (einfaches) Polygon
- Span: Folge benachbarter Pixel auf einer Scanline innerhalb des Polygons
- Hauptgedanke beim Rasterisieren:
  - Durchlaufe aufeinander folgende Scanlines
  - Berechne pro Scanline alle Spans innerhalb des Polygons
- Allg. Algorithmentechnik:
  - *Sweep-Line-Algorithmus*
  - Im Prinzip nutzt man dabei:
    - räumliche Kohärenz
    - Dimensionsreduktion



Polygon Scan Conversion

- Annahme: gesamtes Polygon ist auf dem Bildschirm / Framebuffer

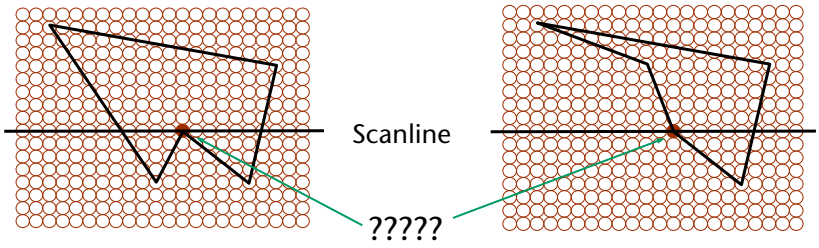
1. Bestimme alle Punkte auf der Scanline, welche die Kante eines Polygons schneiden
2. Sortiere Schnittpunkte von links nach rechts
3. Gruppiere Schnittpunkte in *Spans* und färbe die Pixel dazwischen



- Schnittpunkte
- Restliche Punkte des Spans

Entscheidung "innerhalb" / "außerhalb"

- Wie werten wir Eckpunkte genau auf der Scanline?
  - M.a.W.: begrenzt solch ein Eckpunkt einen Span?



- Lösung: zähle einen Eckpunkt genau dann, wenn er das untere Ende der einen, und das obere Ende der anderen Kante ist
- Alternative: "wackle" am Eckpunkt ein wenig ("perturbation")
  - Dann bekommt man im linken Beispiel 2 oder 0 Schnittpunkte, im rechten genau 1 Schnittpunkt

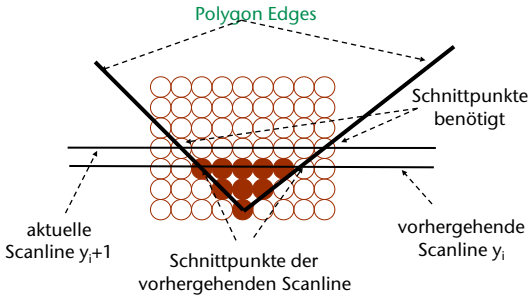
Übersicht des Algorithmus'

- Für jede Scanline:
  - berechne Schnittpunkte zwischen Scanline und Kanten des Polygons
  - Führe Scan-Verarbeitung einzeln für jede Scanline durch
- Kann durch Tabelle aller Kanten optimiert werden
- Für jede neue Scanline: schaue in der Tabelle nach, ob eine Kante geschnitten wird
- Haben wir einen Schnitt zwischen Scanline und Kante berechnet, so verwenden wir diese Information in der nächsten Iteration (inkrementell)

Inkrementelle Schnittberechnung mit Scanline

- Verwende Kantenkohärenz bei Aktualisierung der X-Schnittpunkte
- Wir kennen $y = mx + b, \quad m = \frac{y_1 - y_0}{x_1 - x_0}$
- Sei (x_i, y_i) der Schnittpunkt mit der i-ten Scanline
- Somit

$$x_{i+1} = \frac{y_i + 1 - b}{m} = \frac{y_i - b}{m} + \frac{1}{m} = x_i + \frac{1}{m}$$



Die Active Edge Table

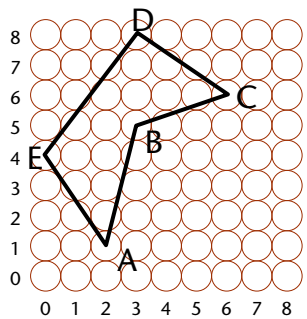
- Verwende *active edge table* (AET) zur Verwaltung der Kohärenz
- AET enthält 1 Eintrag pro Schnitt mit der aktuellen Scanline:
  - y_{max}-Wert der Kante
  - x-Wert des Schnittpunktes zwischen Kante und Scanline
  - Schrittweite in x-Richtung (= 1/m)
- Für jede neue Scanline:
  - Berechne neue Schnittpunkte für alle Kanten
  - Füge jeden neuen Schnittpunkt zur AET hinzu
  - Entferne Kanten, die nicht mehr geschnitten werden
- Um die AET effizient zu aktualisieren, muß man eine *global edge table* (GET) führen

Die Global Edge Table

- GET hat 1 "Slot" für jede Scanline (z.B. 1024)
- Jeder dieser Slots enthält eine Liste von Kanten, deren y_{min}-Wert gleich dem y-Wert der Scanline dieses Slots sind
- Jede Kante des Polygons hat also in genau 1 Slot der GET einen Eintrag
- Jeder Eintrag in der GET für eine Kante enthält:
  - y_{max}-Wert (= oberes Ende) der Kante
  - $x_{@y_{min}}$-Wert (der X-Wert beim y_{min}-Punkt)
  - Schrittweite in X-Richtung (= 1/m)

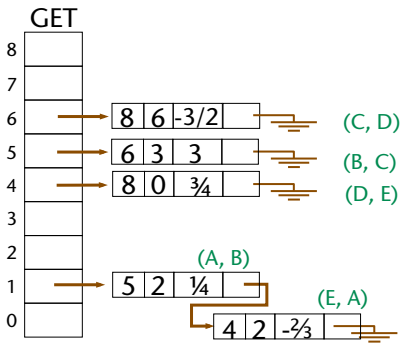
Beispiel einer Global Edge Table

- Gegeben folgendes Polygon:

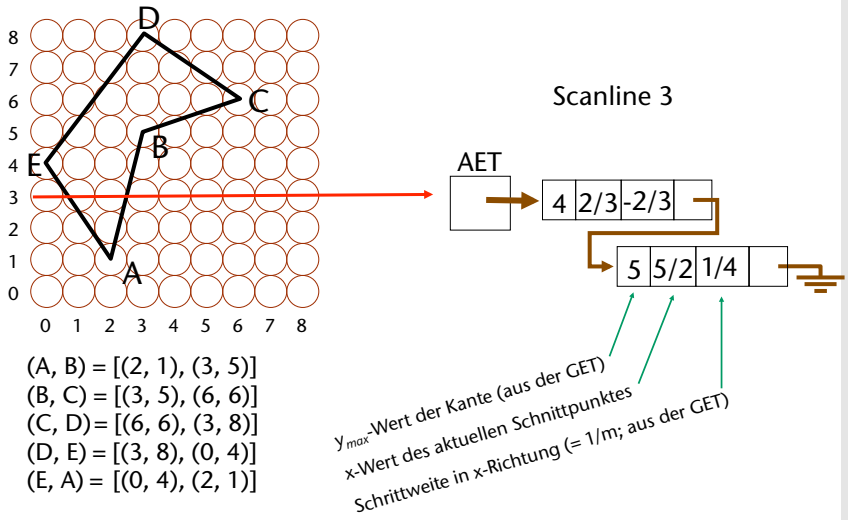


(A, B) = [(2, 1), (3, 5)]
(B, C) = [(3, 5), (6, 6)]
(C, D) = [(6, 6), (3, 8)]
(D, E) = [(3, 8), (0, 4)]
(E, A) = [(0, 4), (2, 1)]

GET Entries (y_{max}, $x_{@ymin}$, 1/m)

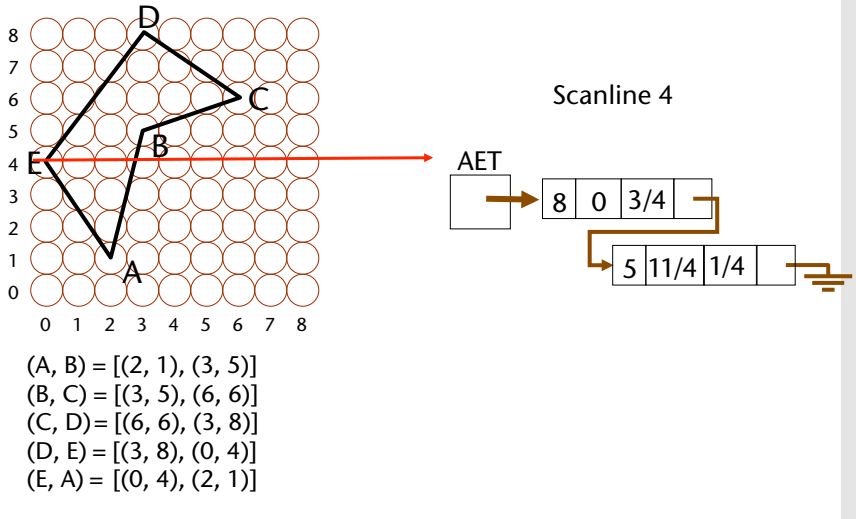


Beispiel für die Aktualisierung der AET

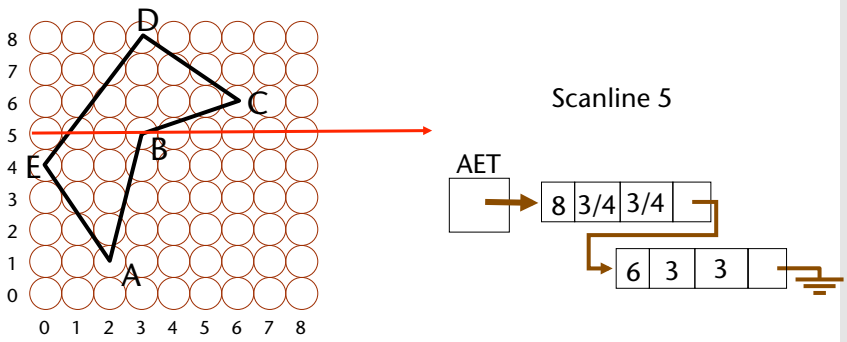


(A, B) = [(2, 1), (3, 5)]
(B, C) = [(3, 5), (6, 6)]
(C, D) = [(6, 6), (3, 8)]
(D, E) = [(3, 8), (0, 4)]
(E, A) = [(0, 4), (2, 1)]

Beispiel für die Aktualisierung der AET



(A, B) = [(2, 1), (3, 5)]
(B, C) = [(3, 5), (6, 6)]
(C, D) = [(6, 6), (3, 8)]
(D, E) = [(3, 8), (0, 4)]
(E, A) = [(0, 4), (2, 1)]



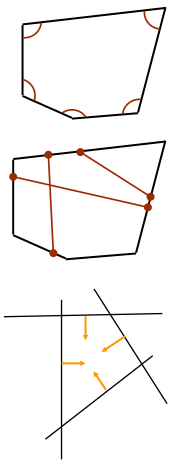
(a, b) = [(2, 1), (3, 5)]
(b, c) = [(3, 5), (6, 6)]
(c, d) = [(6, 6), (3, 8)]
(d, e) = [(3, 8), (0, 4)]
(e, a) = [(0, 4), (2, 1)]

Der Scan-Line Algorithmus für einfache Polygone

1. Erstelle Global Edge Table (GET) mit allen Kanten des Polygons
2. Y erhält den Wert der kleinsten y-Koordinate aller Einträge in GET
3. Initialisiere eine leere Active Edge Table (AET)
4. Wiederhole bis GET und AET leer sind:
   1. Aufnehmen aller Kanten aus GET in AET, für die $y_{min} = Y$
   2. Entferne alle Kanten aus AET, für die $y_{max} = Y$
   3. Sortiere AET nach x (kann man einsparen)
   4. Färbe Pixel zwischen den Schnittpunktpaaren (= spans) aus AET
   5. Für jede Kante aus AET, setze x = x + 1/m
   6. Setze Y = Y + 1 zur Aktivierung der nächsten Scanline

Zerlegung konkaver Polygone

- OpenGL kann nur konvexe Polygone zeichnen!
- Definition "konvex":
  - Innenwinkel ≤ 180°
  - Alle Verbindungsstrecken zwischen Randpunkten bleiben innerhalb
  - Konvexe Hülle der Eckpunkte
  - Schnitt von Halbebenen
- Ann. im Folgenden: keine kollinearen Punkte!
  - Sonst: Spezialfall mit spezieller Behandlung (z.B. Punkt überspringen)
- Aufgabe im Folgenden:
  1. Teste, ob Polygon konvex / konkav
  2. Zerlege gegebenenfalls das Polygon



Lösung 1: Kreuzprodukt

- Berechne

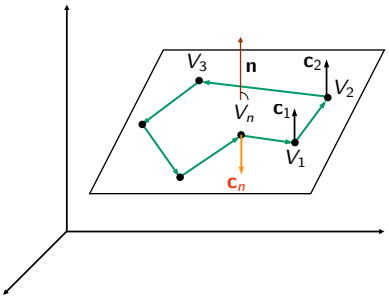
  $\mathbf{c}_i = (V_i - V_{i-1}) \times (V_{i+1} - V_i)$

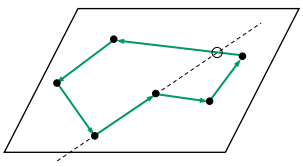
- Teste

  $\mathbf{n} \cdot \mathbf{c}_i > 0$

  (theoretisch müßte $\mathbf{n} \cdot \mathbf{c}_i = \pm|\mathbf{c}_i|$ sein, wenn $|\mathbf{n}| = 1$.)

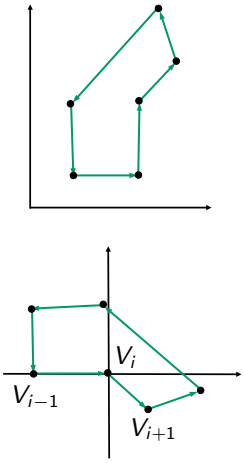
- Bei korrekten Kanten: zerlege Polygon entlang Kante $\overline{V_{i-1} V_i}$
  - Achtung kann mehrere Schnitte geben





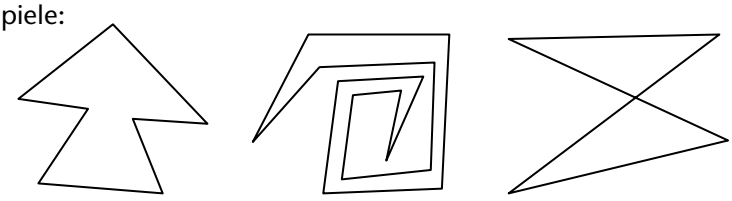
Lösung 2: „Rotationsmethode“

- Transformiere Polygon in die XY-Ebene
- Für jeden Eckpunkt V_i:
  - Transformiere/rotiere Polygon so, daß
    - V_i im Ursprung
    - V_{i-1} auf der negativen X-Achse
  - Teste: Y-Koordinate von $V_{i+1} > 0$
- Für konkave Ecken: schneide Polygon durch X-Achse in 2 oder mehr(!) Teile
  - Zerlege Teile rekursiv weiter



Füllen nicht-einfacher Polygone

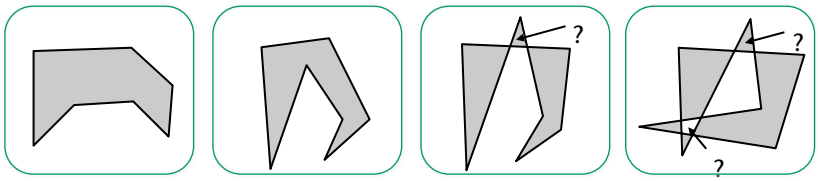
- Def.: Polygon ist einfach ↔ Randkurve hat keinen Schnittpunkt
- Beispiele:



einfach (& hor. konvex) einfach nicht einfach

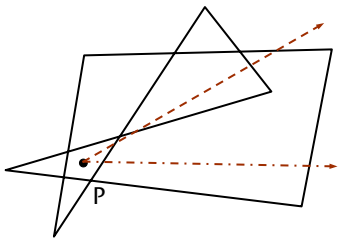
- Eigenschaften:
  - Topologisch äquivalent zu einer 2-dimensionalen Scheibe
  - Folge: es gibt ein wohldefiniertes Inneres/Äußeres
- Wie kann man solche Polygone füllen?
  - Verwende für jedes Pixel einen intuitiv „korrekten“ Inside-/Outside-Test

- Wesentliche Frage: wie definiert man "innen" und "außen"?



Test 1: Odd-Even Rule

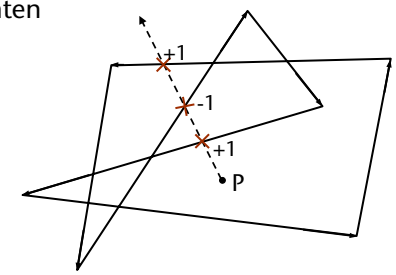
- Zeichne Strahl von Punkt P nach unendlich in irgend eine Richtung
- Zähle Anzahl Schnittpunkte mit dem Kantenzug
- Falls Anzahl ungerade → P innerhalb
- Achtung, falls Strahl Eckpunkt genau trifft!
- Effiziente Schnittberechnung: wähle horizontalen Strahl



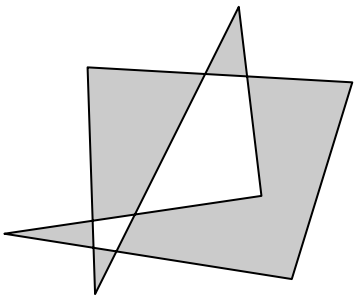
- Vorteil: funktioniert genauso mit Polyedern im 3D (und höherdim.)

Test 2: Winding-Number Rule

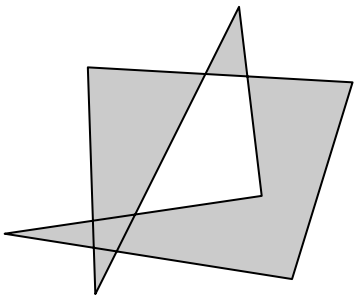
- Versehe Polygon mit konsistentem Umlaufsinn
- Schneide Strahl von P aus mit Kanten
- Setze Winding-Number w := 0
- Für Schnitt mit Kante „von rechts nach links“ erhöhe w;
  sonst erniedrige w
- Falls w ≠ 0 → P innerhalb
- Anmerkung: damit definiert man gleichzeitig „positiv“ bzw. „negativ“ orientierte Regionen



Vergleich



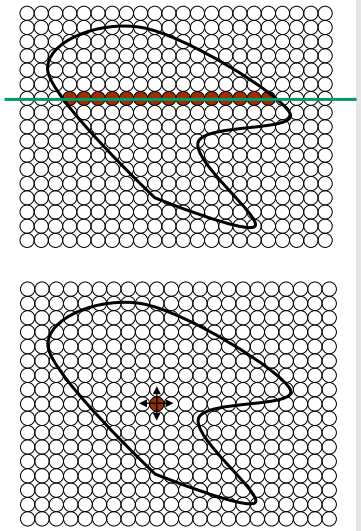
Odd-even Rule



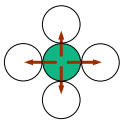
Non-zero Winding Number

Füllen von Regionen

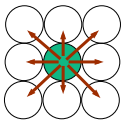
- Ähnliche Aufgabe zu Polygon-Scan-Conversion
  - Häufig in 2D-Zeichenprogrammen
- Erste Methode: wie Polygon-Scan-Conversion
- Zweite Methode:
  - Flood Fill: Wähle ein Pixel innerhalb des Polygons. Färbe rekursiv angrenzende Pixel bis das gesamte Polygon gefüllt ist



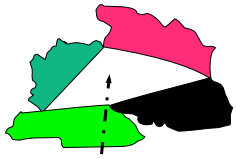
- Region ist identifiziert durch bestimmte Farbe (z.B. Weiß)
- Wähle ein Pixel innerhalb der Region
  - Region ist definiert als zusammenhängendes Gebiet mit alter Farbe
- Rekursion:
  1. Hat Pixel die alte Farbe, dann weise diesem Pixel die Füllfarbe zu
  2. Färbe rekursiv alle diejenigen Nachbarn, die noch die alte Farbe haben
- Alternative Begrenzung : Randkurve mit bestimmter Farbe
- Wahl der Nachbarn:



4-Verbindungen



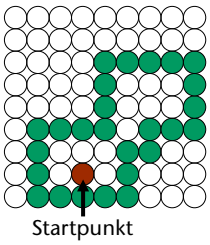
8-Verbindungen

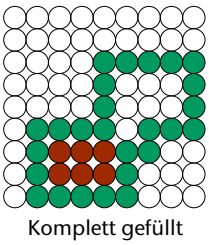


Zu füllende Region

Probleme

- Z.B. bei 4-Verbindungen pro Punkt:


Startpunkt

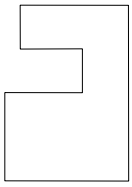

Komplett gefüllt

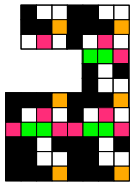
- Ein weiteres Problem: der Algorithmus hat große Rekursionstiefe
  - Kann Stack aus Spans verwenden um Rekursiontiefe zu verringern
  - Verkompliziert aber der innere "Logik" des Algo

Bereiche mit Mustern

- Oft möchten wir einen Bereich mit einem Muster (z.B. gestrichelt) versehen, nicht nur eine Farbe (*stippling*)
- Definiere ein *n* x *m* Pixmap (oder Bitmap), welche wir auf den Bereich abbilden möchten:


5x4 pixmap

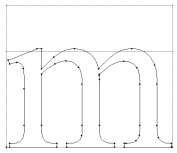

Mit Muster zu versehendes Objekt

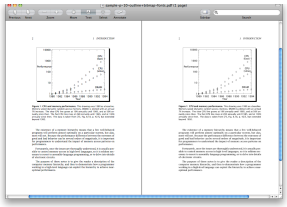

Gemustertes Objekt

- Für jeden Punkt (*x,y*) : verwende die Farbe vom Muster an der Stelle (*x* mod *m*, *y* mod *n*)
- Fragen: soll das Muster relativ zum Polygon oder relativ zum Bildschirm stationär bleiben?

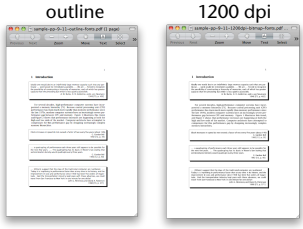
Font-Rendering

- Rendering-Arten bzw. Font-Arten:
  - Bitmap-Font = jedes Zeichen ist eine Bitmap (oder mehrere für verschiedene Font-Größen)
  - Outline-Font = jedes Zeichen wird aus sog. Bézier- oder B-Spline-Kurven zusammengesetzt
    - Adobe Type 1 & Type 3, TrueType, OpenType





outline 1200 dpi



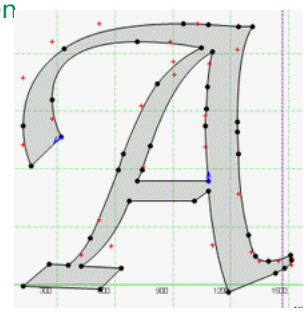
Begriffe

- Glyph = graphische Repräsentation eines oder mehrerer Zeichen

- Typeface = "Design" einer Menge von Zeichen
  - Z.B. Helvetica, Times Roman, Frutiger, Lucida, etc.
  - Wird von Typographen entworfen
- Font = Menge aller Glyphs, die in einer Sprache benötigt werden, in einem bestimmten Typeface und Stil, plus Zusatzinfos
  - Stil = aufrecht (roman), kursiv (italic), fett (bold), halbfett (semibold), ..
  - Zusatzinfos = Hinting, Kerning, Ligaturen, Font-Metrik, ...

Beschreibung von Outline-Fonts

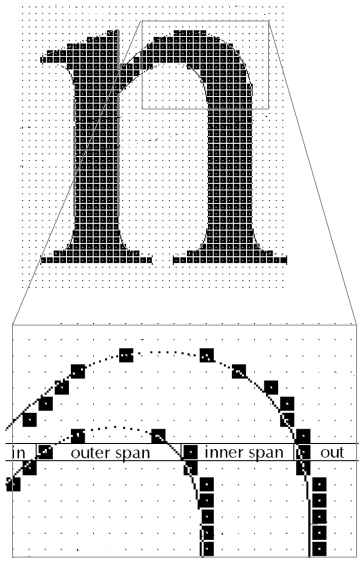
- Zeichen = Menge von geschlossenen Kurvenzügen (*Outlines*)
- Kurvenzug = Menge von Kontrollpunkten
- Umlaufsinn definiert innen / außen:
  - "Links von der Kurve" = innen (oder umgekehrt ...)
- Achtung: Kurvenzüge müssen überschneidungsfrei sein!



- Vorteil: beliebige Skalierung ist ohne Verlust (im Prinzip) möglich → skaliere einfach die Koordinaten der Kontrollpunkte

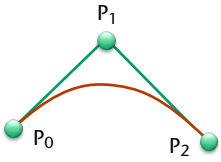
Der Flag-Fill-Algorithmus von Ackland [1981]

- Annahme zunächst:
  - Die einzelnen Pixel sind sehr klein im Vergleich zu dem Zeichen
  - Innerhalb eines Pixels kann man die Kontour durch eine Gerade approx.
  - Die Überdeckung des Pixels > 50 % ⇔ Pixelmittelpunkt liegt "innen"
- Idee des Algorithmus:
  - Zerlege die Scan-Lines in der BBox des Zeichens in innere und äußere Spans
  - Berechne die Start-Pixel jedes Spans aus den Konturen → Flags
  - Fülle die inneren Spans

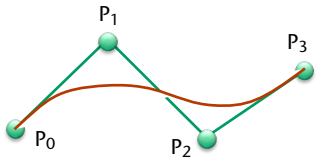


- Outlines setzen sich zusammen aus quadratischen und kubischen Bézier-Kurven
- Quadratische Bézier-Kurven:
  - Definiert durch ein Kontroll-Polygon aus 3 Punkten
  - Polynom 2-ten Grades:

$$P(t) = (1-t)^2 P_0 + 2(1-t)tP_1 + t^2 P_2$$

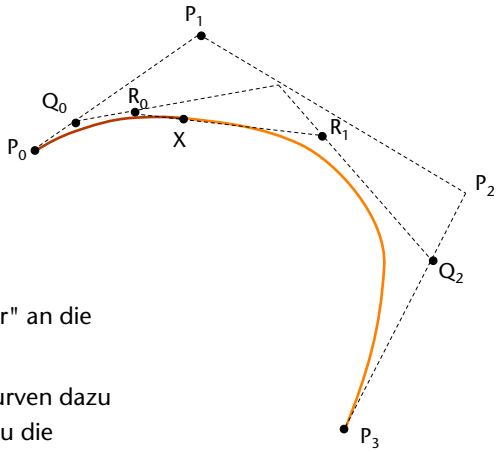


- Kubische Bézier-Kurven:
  - Kontroll-Polygon hat 4 Punkte
  - Polynom ist vom Grad 3:

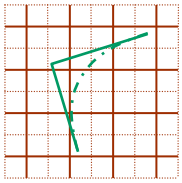
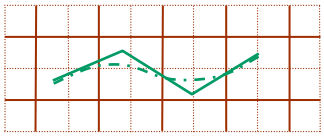
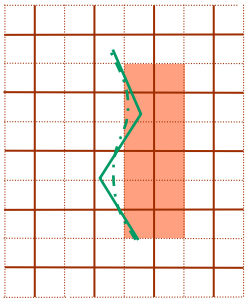


$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)t^2 P_2 + t^3 P_3\,, \quad t \in [0,1]$$

- Approximation durch rekursive Subdivision:
  - Aus P_0, ..., P_3 kann man zwei neue Kontroll-Polygone P_0, Q_0, R_0, X und X, R_1, Q_2, P_3 konstruieren
  - Schmiegen sich "dichter" an die ursprüngliche Kurve
  - Die kubischen Bézier-Kurven dazu bilden zusammen genau die ursprüngliche Kurve

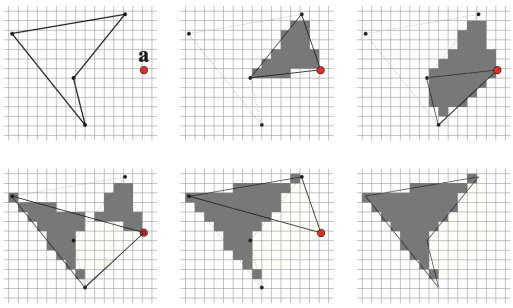


- Betrachte einzeln jede Bézier-Kurve nacheinander, d.h., betrachte deren Kontroll-Polygon
- Fallunterscheidung:
  1. Kontroll-Polygon schneidet keine (horizontale) Scanline → verwerfen
  2. Kontroll-Polygon schneidet eine oder mehrere Scanlines, und schneidet keine vertikale Gitterlinie → Flags (Pixel) rechts der Schnittpunkte setzen
  3. Sonst (Kontroll-Polygon schneidet horizontale und vertikale Gitterlinien) → 1x Subdivision machen und diese rekursiv behandeln

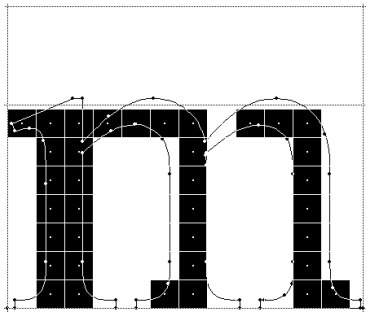
Der XOR-Algorithmus

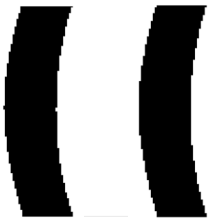
- Gegeben ein geschlossener, überschneidungsfreier Polygonzug, oder mehrere, die eine Region in der Ebene definieren
- Wähle einen beliebigen Anker-Punkt A
- Betrachte alle Kanten PQ der Reihe nach:
  - Invertiere alle Pixel im Dreieck ΔAPQ
- Beispiel:

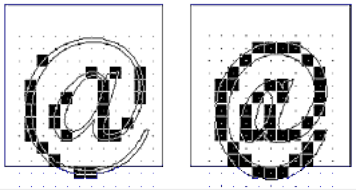


Probleme

- Bei kleinen Font-Größen können, je nach "Phase", folgende Probleme auftreten:
  - Drop-outs
  - Ungleiche Dicke der Stämme
  - Serifen in verschiedene Richtung
- Phase = Abstand zwischen linkem Rand der BBox und vertikale Gitterlinie links davon

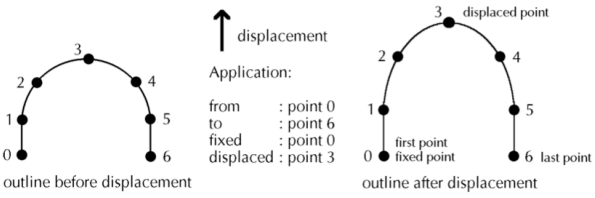




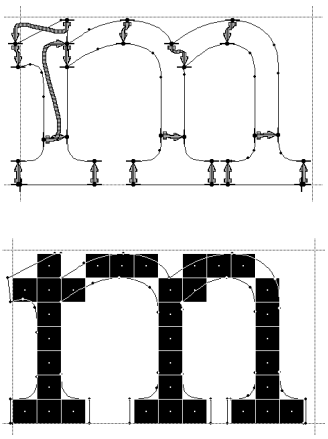


Hinting (grid fitting)

- Lösung: der Rasterizer verzerrt die Konturkurven ein klein wenig und passt sie dem Gitter an, durch Verschieben einzelner Kontrollpunkte
- Hinting = Regeln, die besagen ...
  - welche Punkte verschoben werden dürfen;
  - welche Punkte proportional mit verschoben werden müssen;

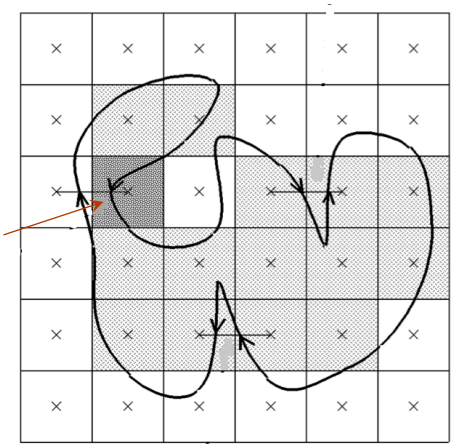


- Hinting (Fortsetzung):
  - welche anderen Punkte dann mitverschoben werden müssen (Constraints)
  - Muß vom Font-Designer gemacht werden
- NB: Diese Art Hinting wird nur bei TrueType-Fonts gemacht
  - Völlig anders bei Type1 ...



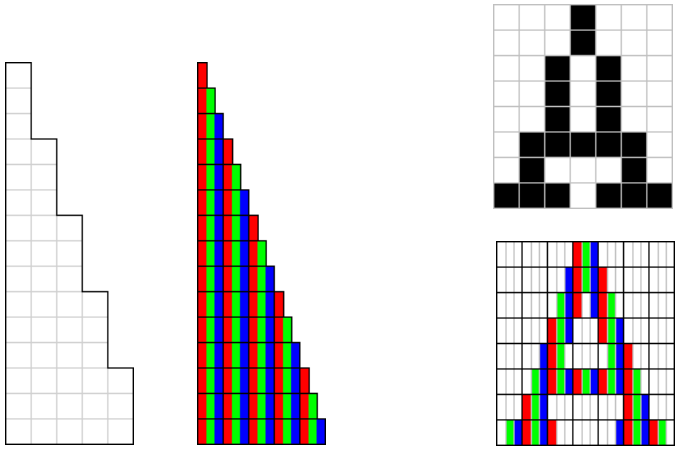
- Dropout-Kontrolle:

Dropout-Pixel wird nachträglich eingefügt, nachdem ein leerer Span detektiert wird



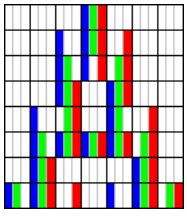
Sub-Pixel Font-Rendering

- Die Idee: betrachte jedes "Primär-Pixel" (R, G, und B) als eigenständiges Pixel:



- Anwendung beim Font-Rendering: skaliere einfach ein Zeichen horizontal um den Faktor 3 bevor rasterisiert wird

- Einschränkungen:
  - Die Software muß den Monitor-Typ kennen (RGB-, oder BGR-, oder Delta-Anordnung)
  - Bringt nichts für hochkant gestellte Monitore (gerade bei Text ist die horizontale Auflösung viel wichtiger)
  - *Color fringing*
  - Patentiert von Micro$oft



- Bessere Alternative (?):
  - Fasse Rot- und Blau-Pixel von benachbarten Tripeln zu einem Primär-Pixel zusammen
  - Verwende nur Grün- und Rot/Blau-Pixel (= nur doppelte Auflösung)
  - Skaliere Fonts vor dem rasterisieren um Faktor 2
  - Modelliere die menschliche Farbwahrnehmung und korrigiere die Color Fringes entsprechend