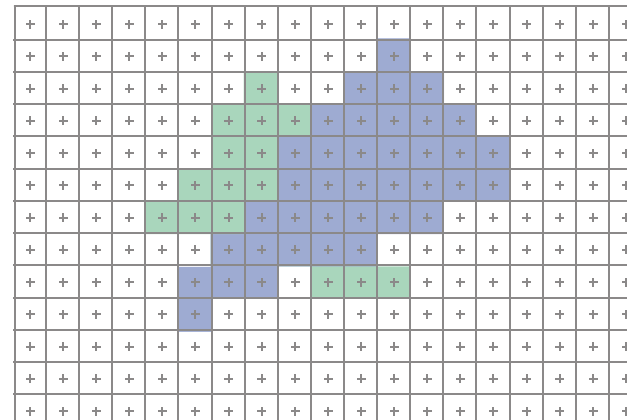


Computer–Graphik I

Polygon Scan Conversion



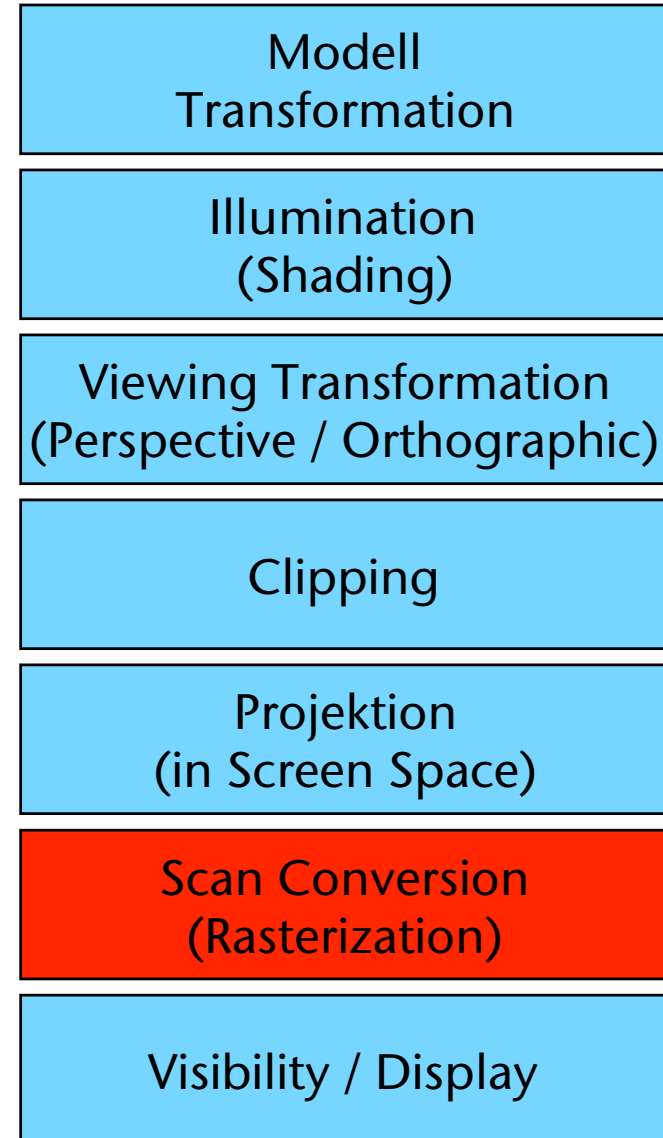
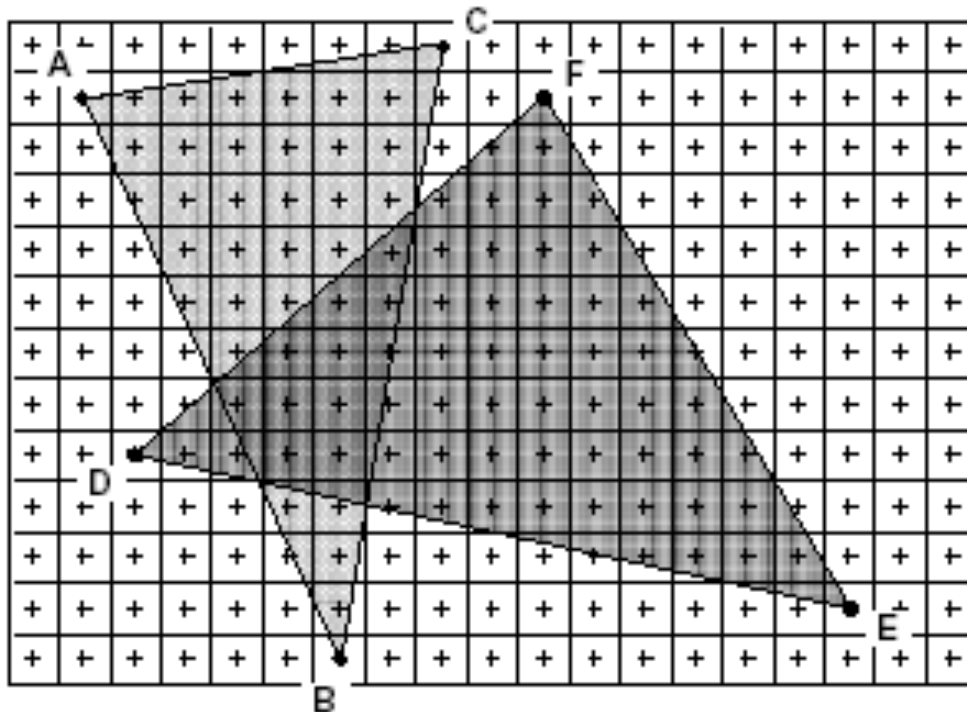
G. Zachmann

University of Bremen, Germany

cgvr.informatik.uni-bremen.de

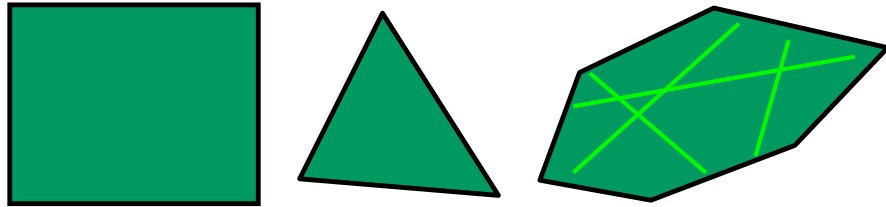
Einordnung in die Pipeline

- Rasterisierung der Objekte in Pixel
- Ecken-Werte interpolieren (Farbe, Tiefenwert, ...)



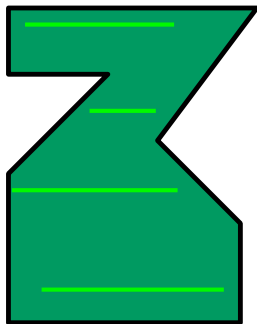
Klassifikation der Polygone

Konvex



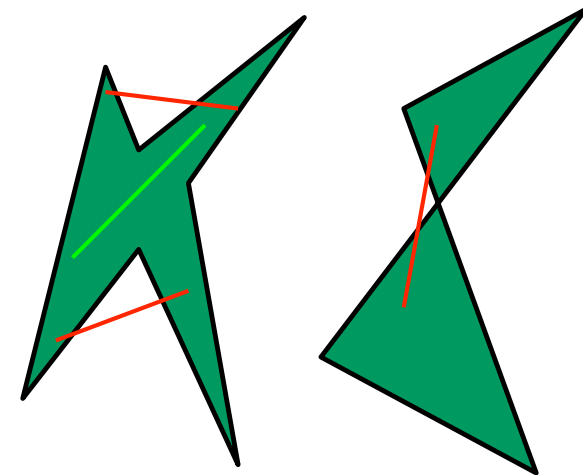
Für jedes Punktepaar in einem konvexen Polygon liegt die Verbindung auch innerhalb des Polygons

Horizontal Konvex



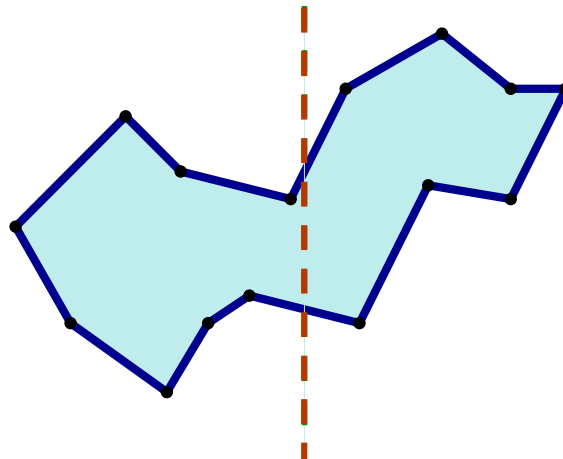
Selbe Definition, gilt hier aber nur für Punkte auf der selben horizontalen Linie

Konkav

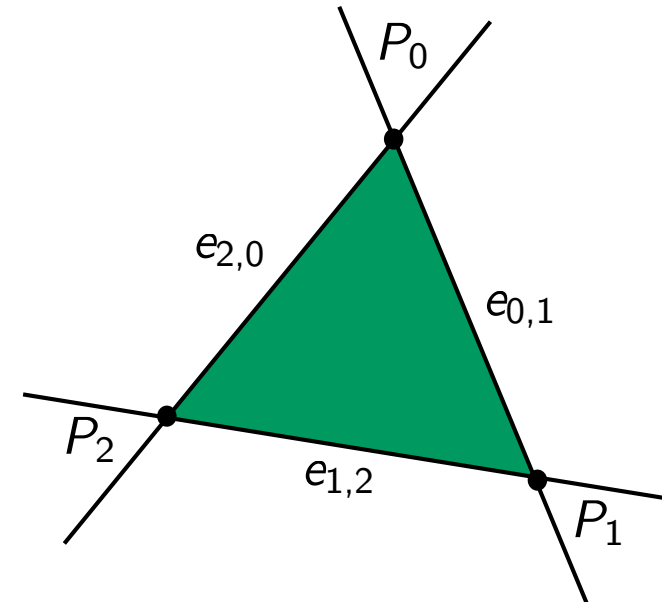


- Monotone Polygone:
Ein Polygon heißt **monoton bzgl. einer Geraden L** , falls jede Gerade senkrecht zu L das Polygon in höchstens zwei Punkten schneidet.

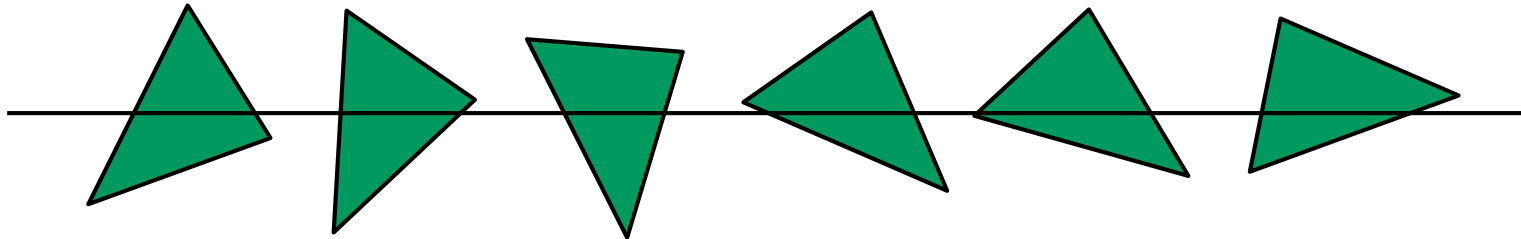
- Beispiel:



- 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)



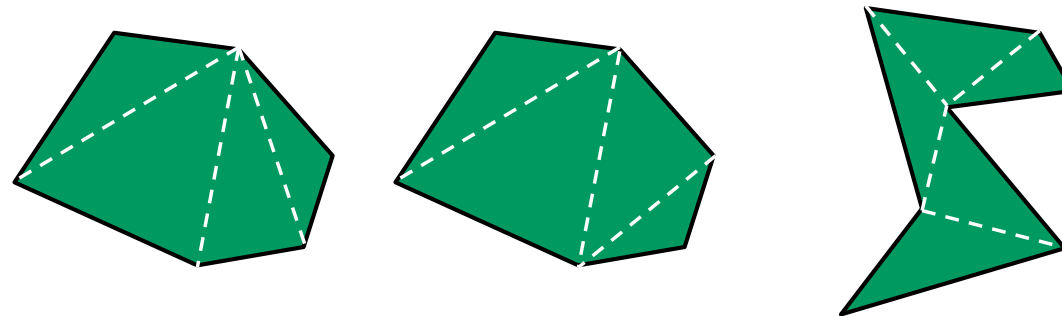
- Dreiecke sind *immer* konvex → egal wie man ein Dreieck dreht, es gibt nur ein Schnittintervall (= *Span*) für jede Gerade (*Scanline*)



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

- Definition:

Triangulierung eines Polygons = Partitionierung des Polygons ausschließlich durch **innere Diagonalen** und ohne zusätzliche Punkte



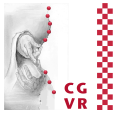
- Satz:

Jedes beliebige Polygon kann trianguliert werden.

- Fragen:

- Hängt die Anzahl Dreiecke von der Wahl der Diagonalen ab?

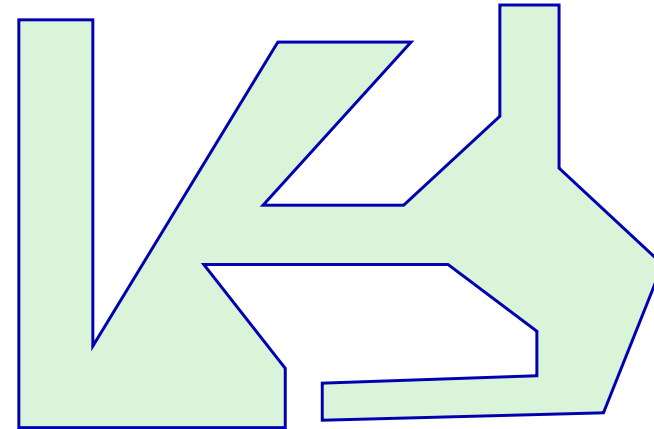
Exkurs: das *Art Gallery Theorem*



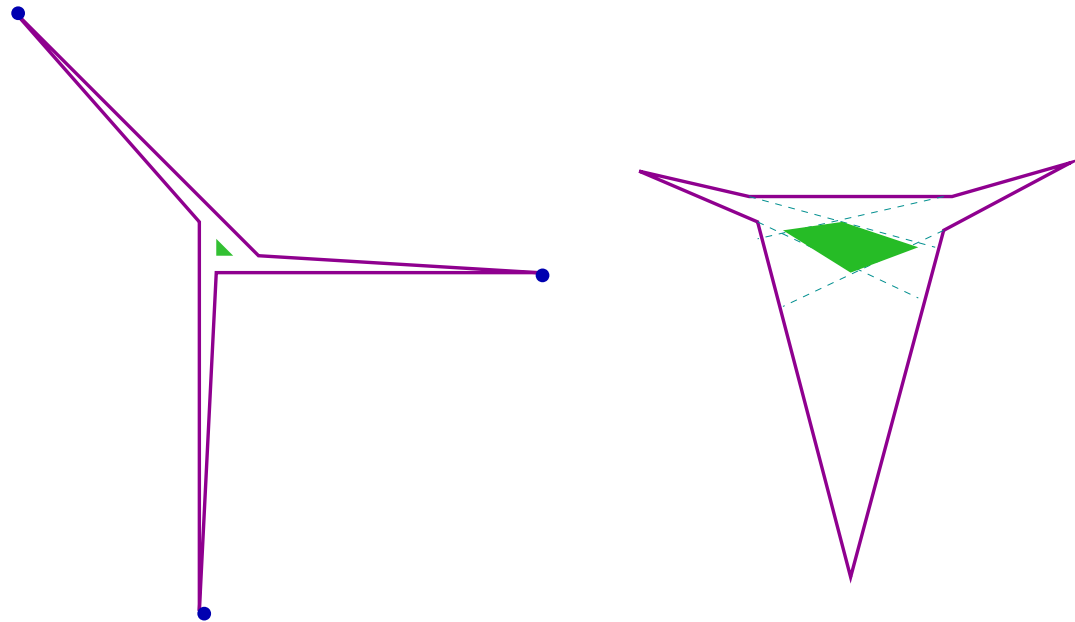
- Aufgabenstellung:
 - Gegeben ist der Grundriß (*floor plan*) einer Galerie als einfaches Polygon mit n Eckpunkten
 - Jeder Wächter (*guard*) ist fest an einem Punkt stationiert und kann 360° seiner Umgebung einsehen, aber nicht durch Wände schauen
 - Frage: wieviele Wächter benötigt man?
- Satz (Beweis kommt später):

Jede Gallery benötigt höchstens $n/3$ Guards.
- Geschichte:

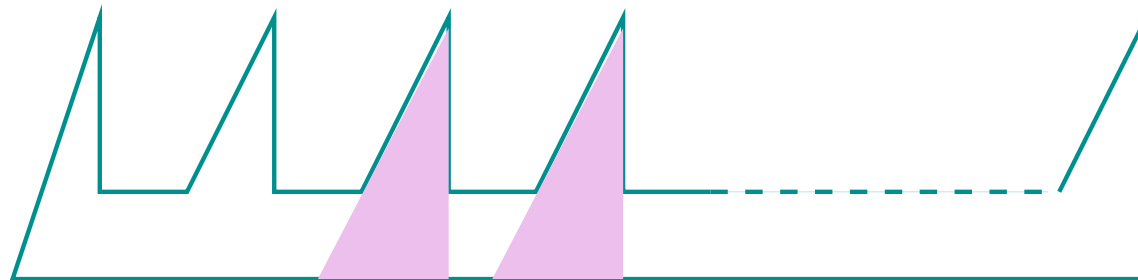
Victor Klee stellte dieses Problem Václav Chvátal auf eine Mathematiker-Konferenz 1973. Dieser fand sofort einen (sehr komplizierten) Beweis, der später mittels Triangulierung sehr vereinfacht wurde.



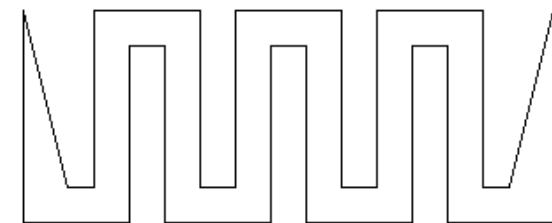
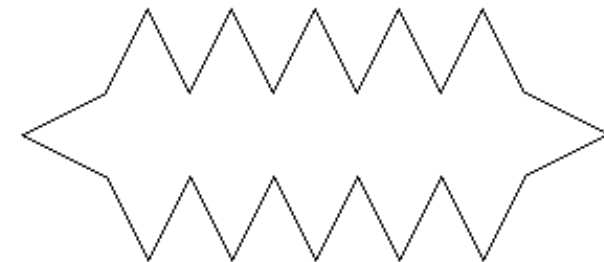
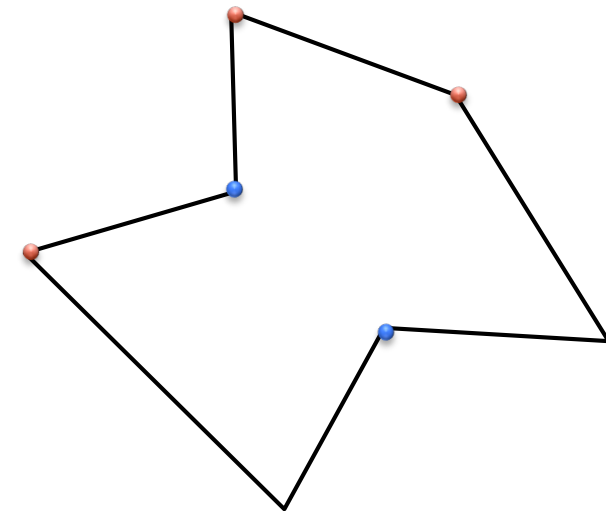
- Satz gibt nur obere Schranke, nicht die optimale Anzahl!
- Pathologische Fälle:



- Obere Schranke wird auch angenommen:

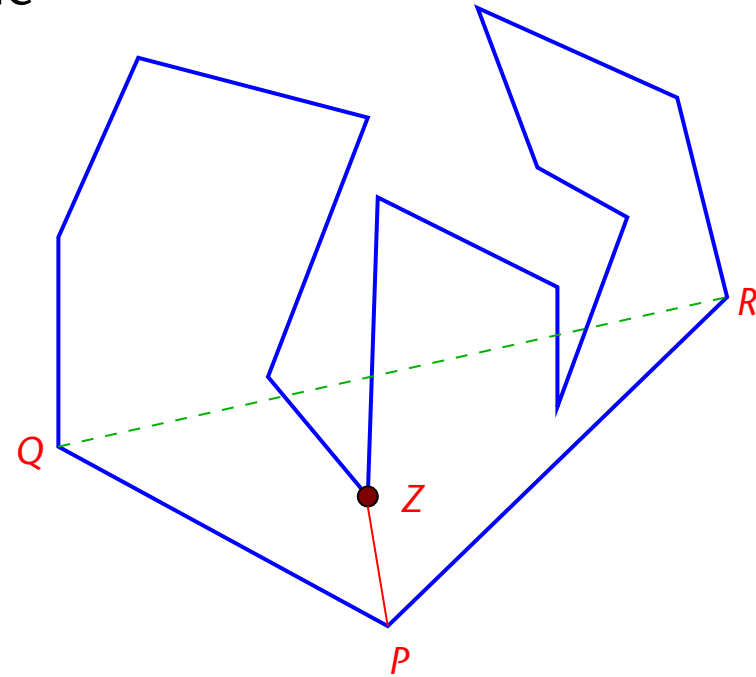


- **Konvexer Vertex** / Eckpunkt:
Innenwinkel $< 180^\circ$
- **Reflex-Vertex** = konkaver Vertex:
Innenwinkel $> 180^\circ$
(= **Mund**)
- **Ohr / Diagonale**:
 $V_{i-1}V_iV_{i+1}$ heißt **Ohr** \Leftrightarrow Strecke $\overline{V_{i-1}V_{i+1}}$
ist komplett innerhalb des Polygons;
in diesem Fall heißt $\overline{V_{i-1}V_{i+1}}$ **Diagonale**.
- **Satz (Zwei-Ohren-Satz)**:
Jedes Polygon (außer Dreiecken)
hat (mindestens) 2 Ohren.

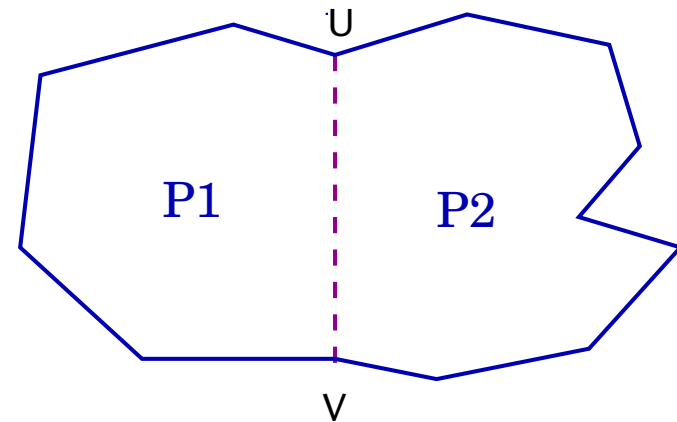


Beweis des Satzes (Jedes Pgon kann trianguliert werden)

- Vereinfachung: nur einfache Polygone
- Beweis durch Induktion
 - Basisfall = Dreieck
- Induktionsschritt:
 - Wähle konvexe Ecke P ;
seien Q & R Vorgänger bzw.
Nachfolger von P
 - Falls QR innere Diagonale des Polygon:
→ füge diese hinzu; fertig
 - Andernfalls:
 - Sei Z derjenige Reflex-Vertex (konkave Vertex), der am weitesten von QR
entfernt und gleichzeitig innerhalb ΔPQR
 - Füge Diagonale PZ zur Traingulierung hinzu
 - Trianguliere "linkes" und "rechtes" Teilpolygon

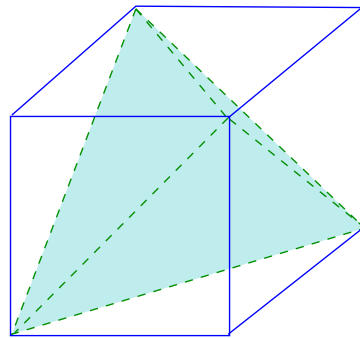


- Satz:
 - Jede Triangulierung eines n -gons hat $n-2$ Dreiecke.
- Bezeichne mit $t(P)$ = Anzahl Dreiecke in irgendeiner Triangulation des Polygons P
- Beweis: mittels Induktion
- Basisfall: einzelnes Dreieck, $t(P) = 1 = 3-2$
- Induktionsschritt:
 - Wähle eine Diagonale UV in der gegebenen Triangulation
 - Diese zerlegt P in P_1 und P_2
 - $t(P) = t(P_1) + t(P_2) = n_1 - 2 + n_2 - 2$
 - Da $n_1 + n_2 = n + 2 \Rightarrow t(P) = n - 2$

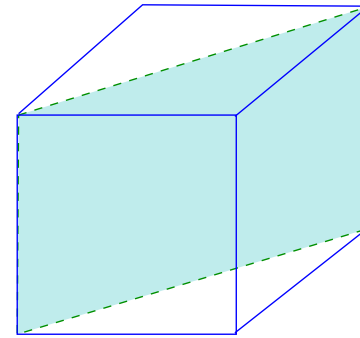


Zum Vergleich: Triangulation (= "Tetraedrisierung") in 3D

- Verschiedene Triangulierung → verschiedene Anzahl Tetraeder:

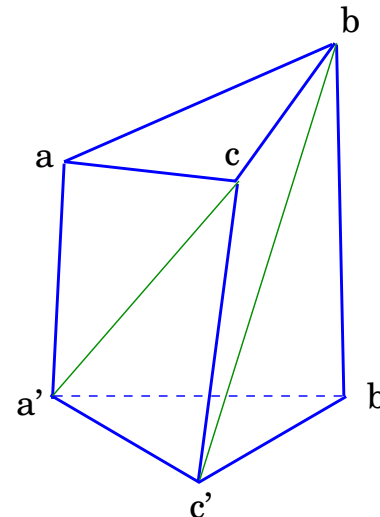
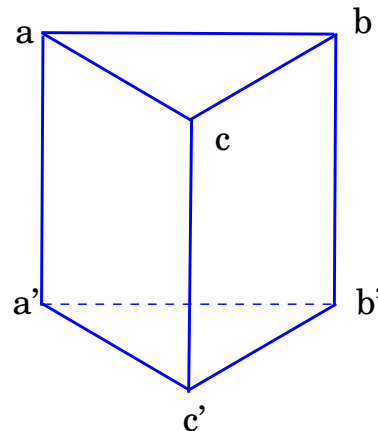


5 Tetraeder



6 Tetraeder

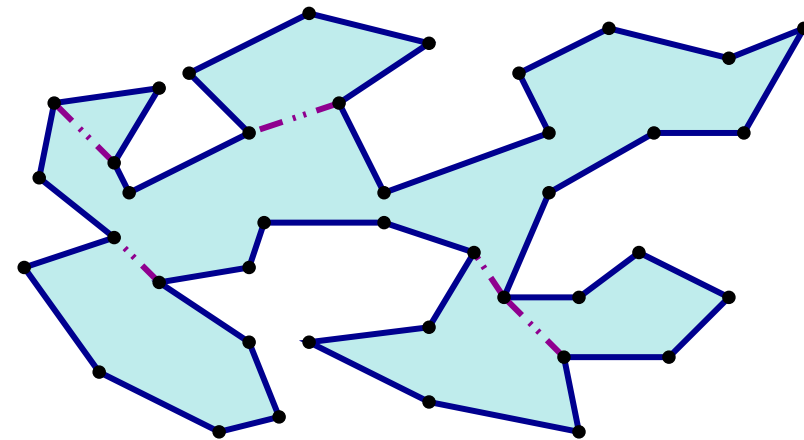
- Ein untriangulierbares ("un-tetraedrisierbares") Polyeder:



- Einfacher Algorithmus (*ear clipping*):
 - Finde ein Ohr in $O(n)$ Zeit
 - Schneide dieses ab und wiederhole
 - Laufzeit: $O(n^2)$
- Erst 1991 fand Chazelle einen $O(n)$ -Algo
 - Dieser ist aber so kompliziert, dass er völlig unpraktikabel ist ;-)
- Im Folgenden: ein einfacher $O(n \log n)$ - Algorithmus

Überblick des einfachen Algorithmus'

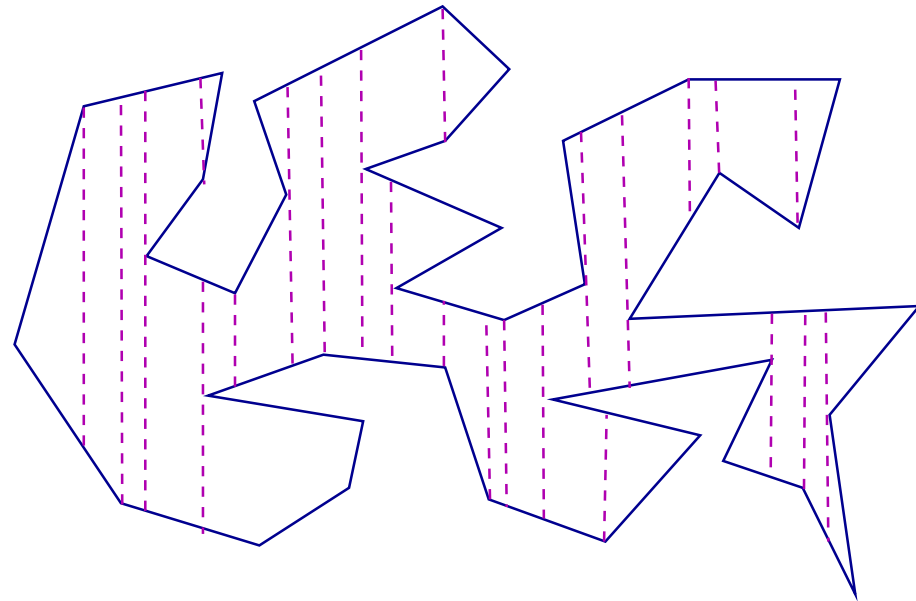
1. Polygon in Trapezoide zerlegen
2. Trapezoide zu x-monotonen Polygonen zusammenfassen
3. Die x-monotonen Polygone triangulieren



X-monotone Zerlegung

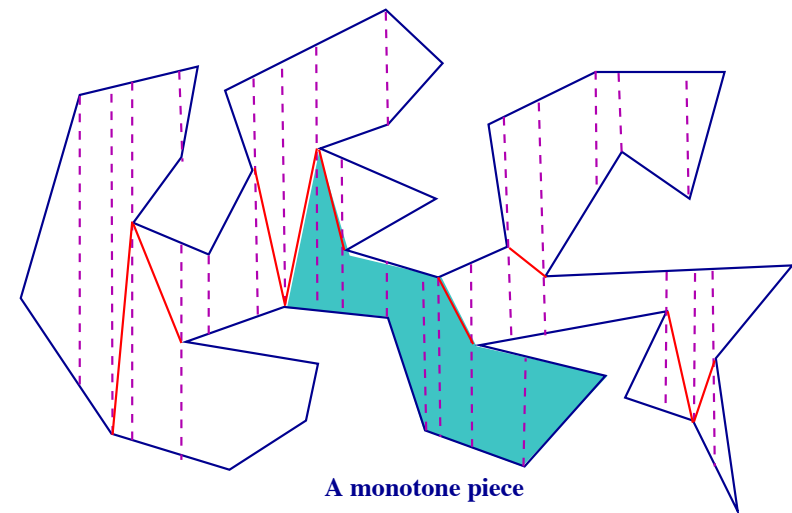
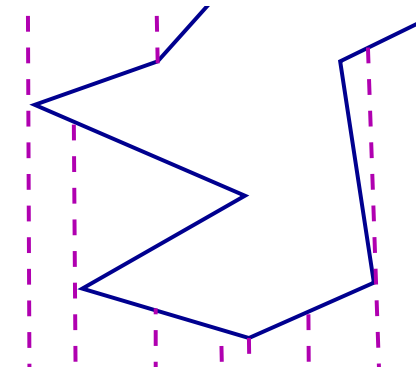
Zerlegung in Trapezoide

- Idee: verwende die Algorithmentechnik **Line-Sweep**
- Bei jedem Vertex: verlängere Linie nach oben / unten, bis eine Kante getroffen wird
- Jede Fläche dieser Zerlegung ist ein Trapezoid (das zu einem Dreieck degeneriert sein kann)
- Details: ...
 - Benötigt Segment-Tree / Intervall-Tree (s. VL "Geometrische Datenstrukturen")
- Laufzeit: $O(n \log n)$



Zerlegung in x-monotone Polygone

- Bezeichnung:
Ein Reflex-Vertex heie **linker/rechter Reflex-Vertex**, falls die beiden inzidenten Kanten nach **rechts/links** zeigen.
- Beobachtung:
Nicht-Monotonizitt kommt genau von linken oder rechten Reflex-Ecken (nicht von anderen Reflex-Ecken)
- Idee (o. Bew.):
Fge zu jedem **linken/rechten** Reflex-Vertex eine Diagonale zu der *Polygon-Ecke* seines **linken/rechten** Trapezoids ein



- Behauptung (o. Bew.):
Mit der "richtigen" Datenstruktur (sog. **DCEL**, in CG2) kann man den zugehörigen Vertex zu solch einer Diagonalen in $O(1)$ finden.
- Behauptung 2:
Eine explizite Konstruktion der Trapezoide ist gar nicht nötig; man kann die Zerlegung in monotone Polygone direkt beim Line-Sweep machen.

- Beobachtung: in einem x-monotonen Polygon gibt es eine "obere" und eine "untere" Vertex-Kette

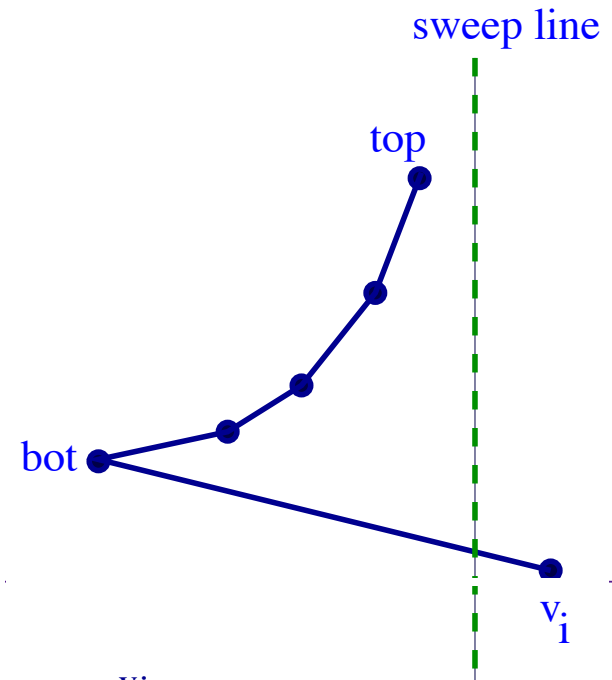
```

Sortiere alle Vertices  $v_1, v_2, \dots, v_n$  von links nach rechts
Pushe  $v_1, v_2$  auf den Stack
For  $i = 3 \dots n$ :
    # z.Z. sind Vertices  $v_q$  (bot), ...,  $v_j$  (top) auf dem Stack
    Fall 1:  $v_i$  und  $\text{top}(\text{stack})$  sind in derselben Kette
        Füge Diagonalen  $v_i v_j, \dots, v_i v_k$  ein, wobei  $v_k$  der letzte
            zulässige Vertex ist; evtl. auch keine Diagonale
        Falls Diagonalen möglich waren:
            Pop  $v_j, \dots, v_{k-1}$ 
            Push  $v_i$ 
    Fall 2:  $v_i$  ist auf der anderen Kette als  $\text{top}(\text{stack})$ 
        Füge alle Diagonalen  $v_i v_j, \dots, v_i v_q$  ein
        Merke  $v_j$ , lösche Stack
        Push  $v_j$  und  $v_i$  # Neustart für nächste Reflex-Kette
    
```

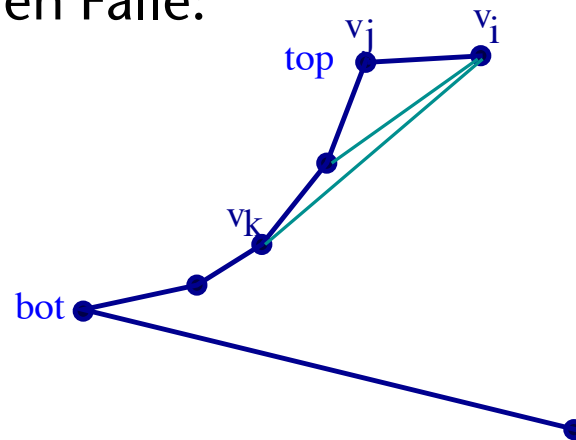
Korrektheitsbetrachtungen (und Verständnis)

■ Schleifen-Invarianten:

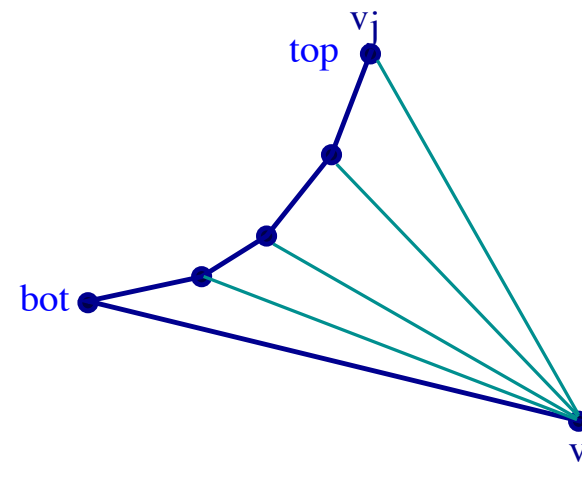
- Alle Vertices auf dem Stack bilden eine Kette von Reflex-Vertices (**Reflex-Kette**) (keine linken oder rechten Reflex-Vertices!)
- Der nächste Vertex in der "anderen" Kette liegt rechts von der aktuellen Sweep-Line
- Das "Rest-Polygon" ist wieder einfach (und x-monoton)



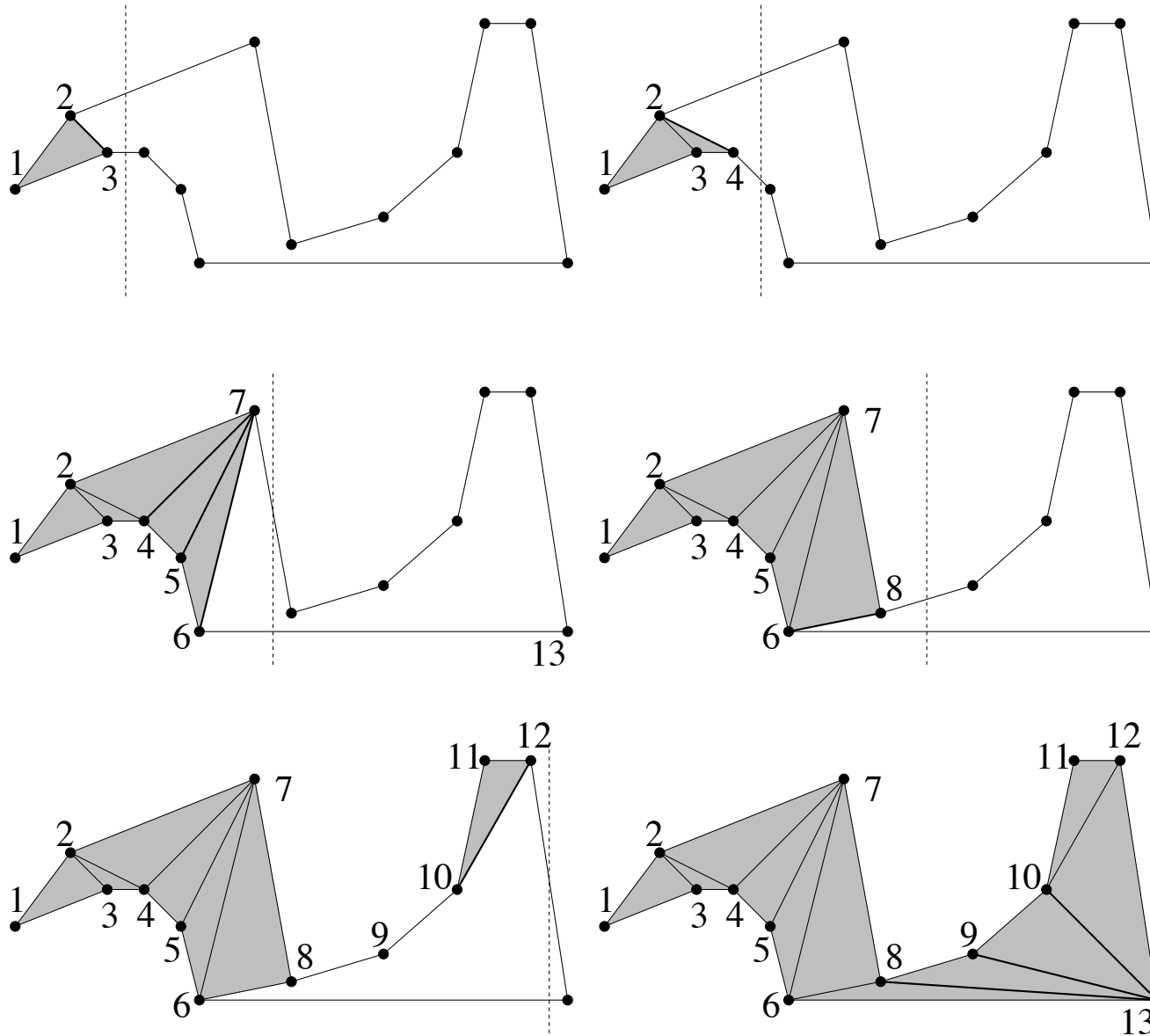
■ Die beiden Fälle:



New stack: (bot, ..., v_k , v_i)

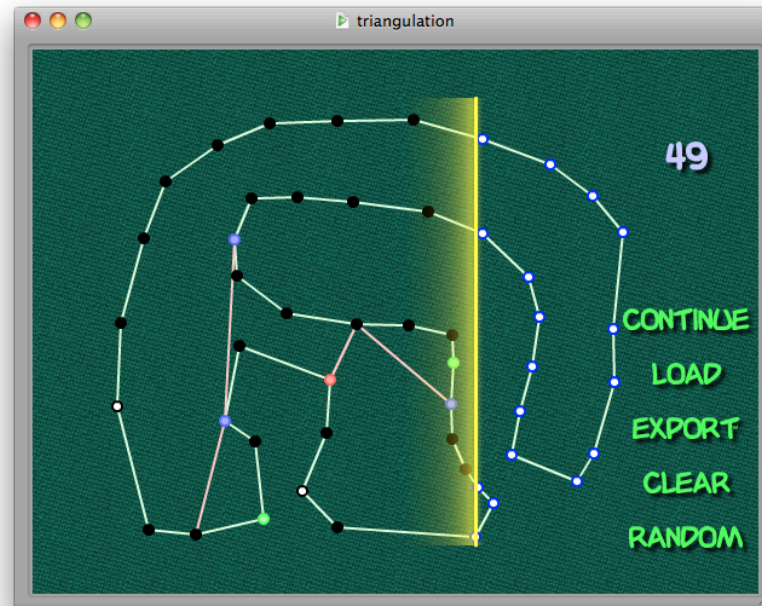


New stack: (v_j , v_i)



Laufzeit

- Laufzeit für Schritt 3 (Triangulierung eines monotonen Pgons)
 $= O(n)$
- Gesamtlaufzeit: $O(n \log n)$
- Demo:



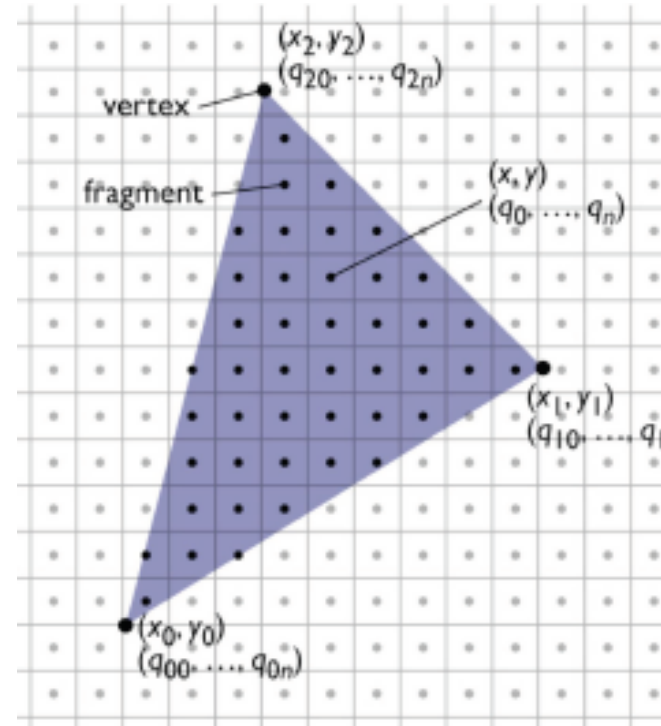
<http://computacion.cs.cinvestav.mx/~anzures/geom/triangulation.php>

Auflösung: Beweis für *Art Gallery Theorem*

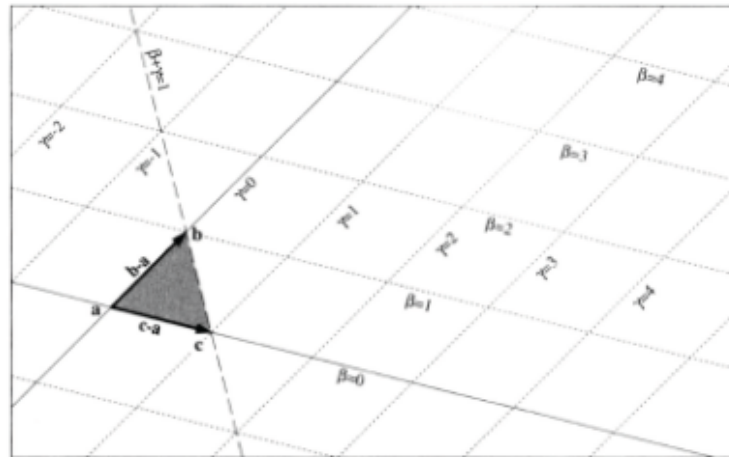
- Mit Triangulations-Satz ganz einfach ...

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}

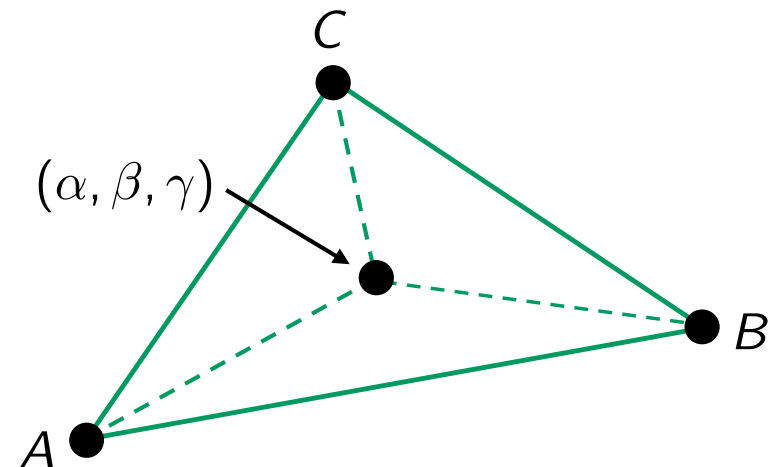


- Baryzentrische Koordinaten:



- Test ob Punkt im Dreieck liegt:

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

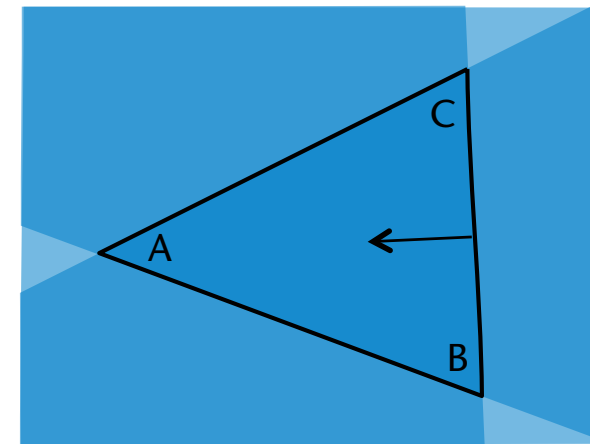
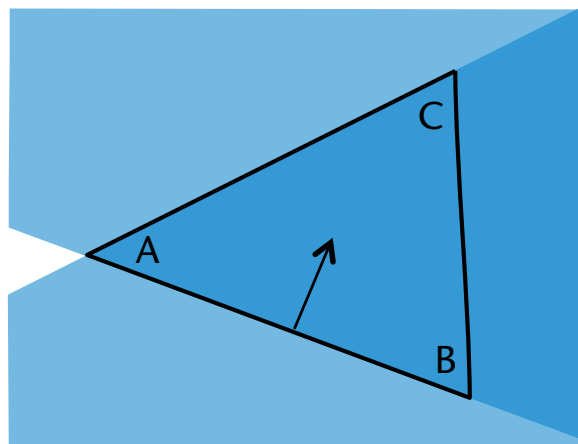
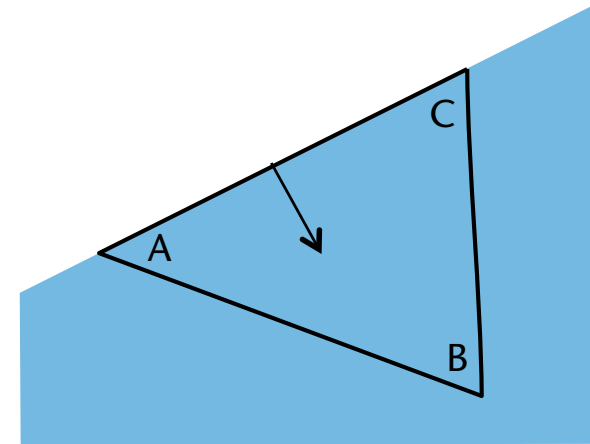


- In der Ebene ist ein Dreieck die Schnittmenge von 3 Halbebenen:

$$(X - C) \cdot (A - C)^\perp > 0$$

$$(X - A) \cdot (B - A)^\perp > 0$$

$$(X - B) \cdot (C - B)^\perp > 0$$



Lineare Interpolation von Farben im Dreieck

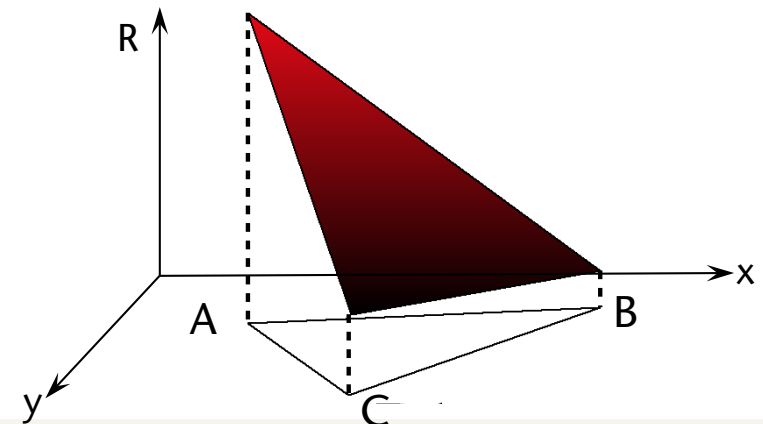
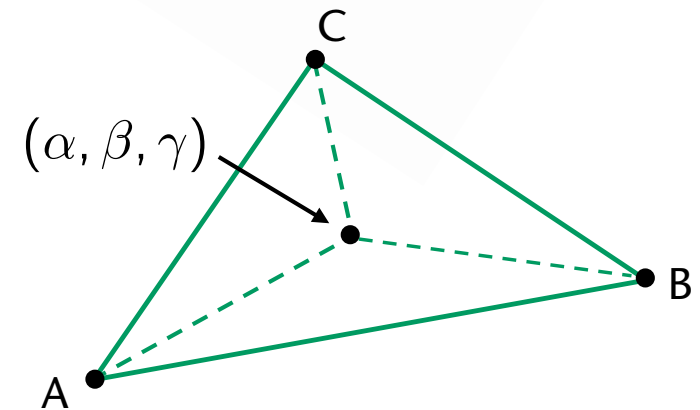
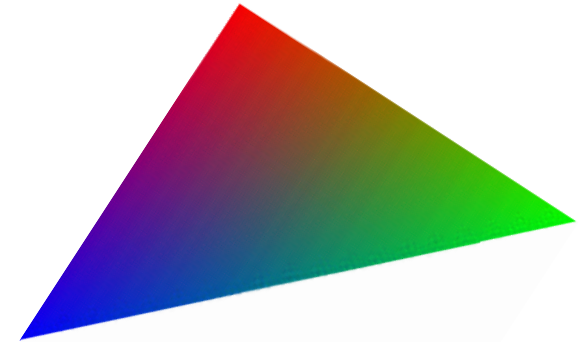
- Idee: benutze die baryzentrischen Koordinaten auch zur Interpolation
- Interpoliere die 3 Farbkanäle (z.B. RGB) unabhängig voneinander
- Am Beispiel des Rot-Kanals R:

- Wenn

$$X = \alpha A + \beta B + \gamma C$$

- Setze

$$R(X) = \alpha R(A) + \beta R(B) + \gamma R(C)$$



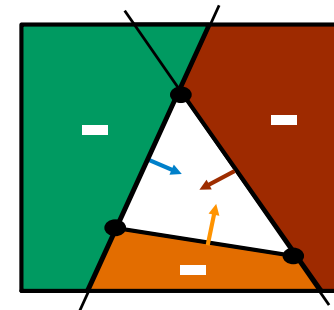
■ Idee:

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

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

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

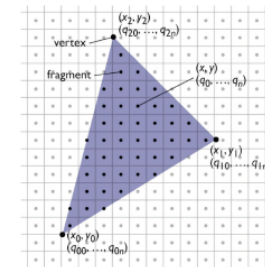
$$C = \alpha C_A + \beta C_B + \gamma C_C$$



■ Algo:

```

for y = y_min ... y_max:
  for x = x_min ... x_max:
    berechne α, β, γ
    if α > 0 and β > 0 and γ > 0:
      C = αC_A + βC_B + γC_C
      zeichne Pixel (x,y) mit Farbe C
  
```



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

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

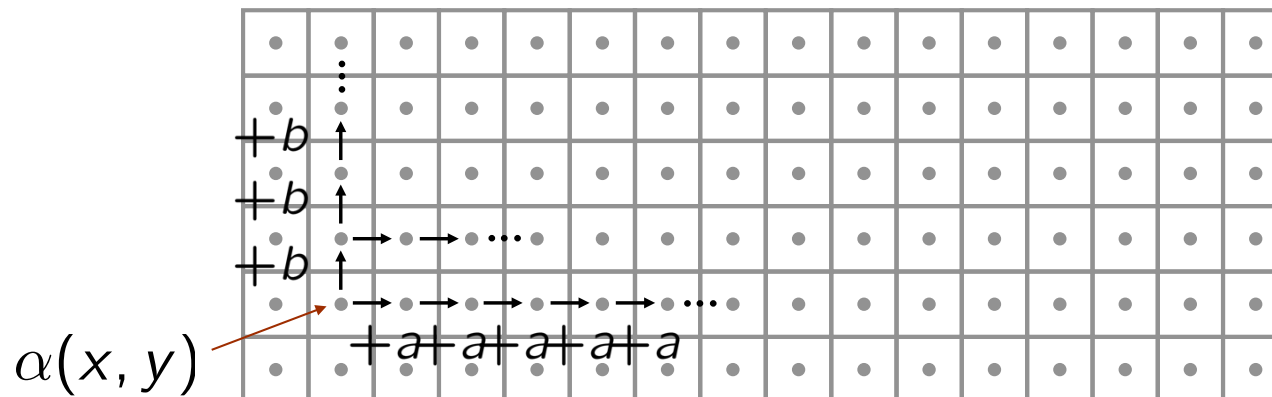
$$\alpha = ax + by + c$$

Dito für β, γ

- Lineare Funktionen können sehr effizient **inkrementell** auf einem Gitter ausgewertet werden (think DDA):

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

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

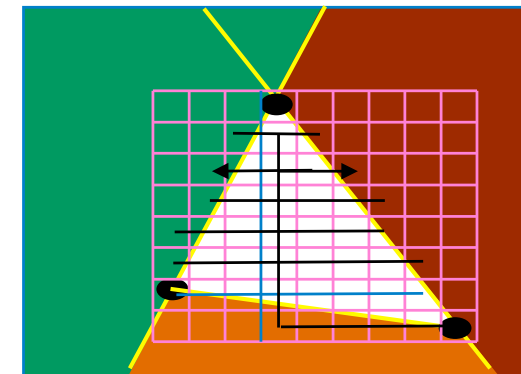
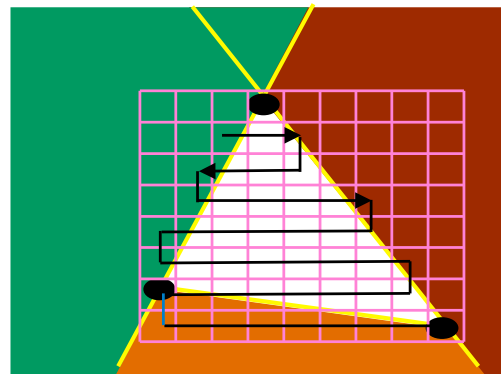
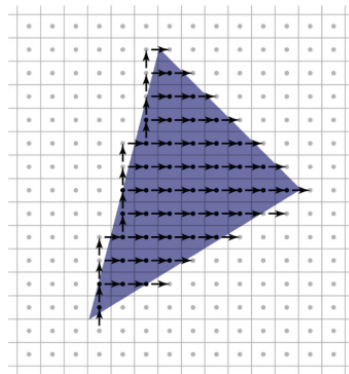
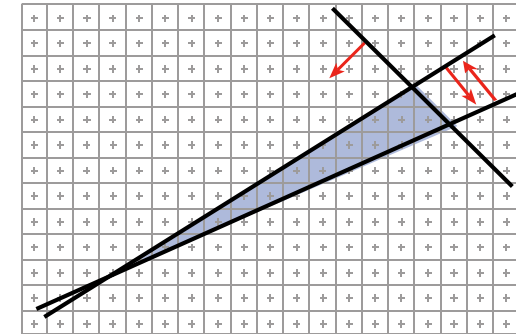


```
linEval(xl, xh, yl, yh, cx, cy, ck):  
  
# setup  
compute a, b, c ...  
qRow = a*xl + b*yl + c  
  
# traversal  
for y = Ymin ... Ymax:  
    qPix = qRow  
    for x = Xmin ... Xmax:  
        draw(x, y, qPix)  
        qPix += a  
    qRow += b
```



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

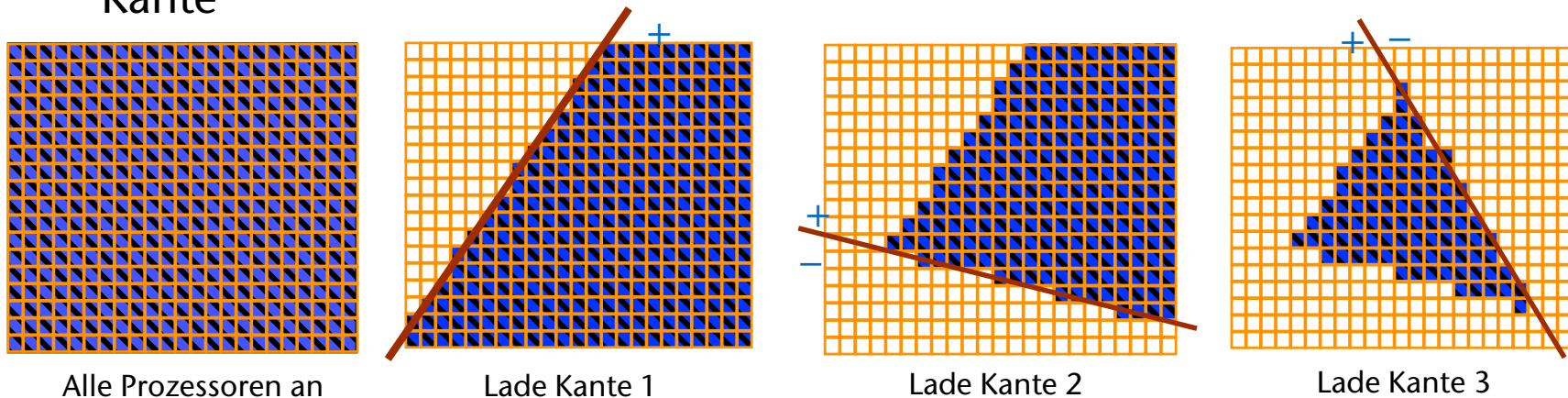
- Problem: wenn das Dreieck lang und schmal ist, dann werden viele unnötige Berechnungen durchgeführt
- Erinnerung: Dreieck ist konvex
 - Folge: wenn man in der x–Schleife einmal ein Pixel außerhalb erreicht, dann sind alle folgenden in dieser Scanline auch außerhalb



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

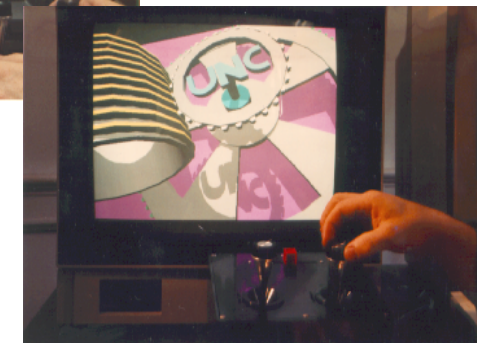
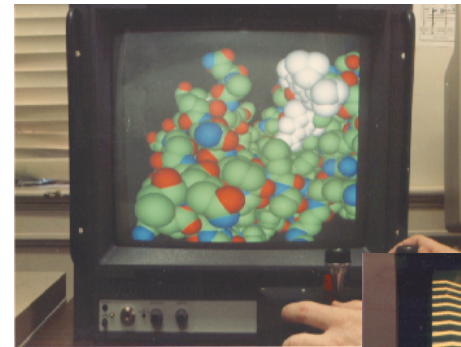
Kurzer Exkurs: Die Pixel-Planes-Architektur

- 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



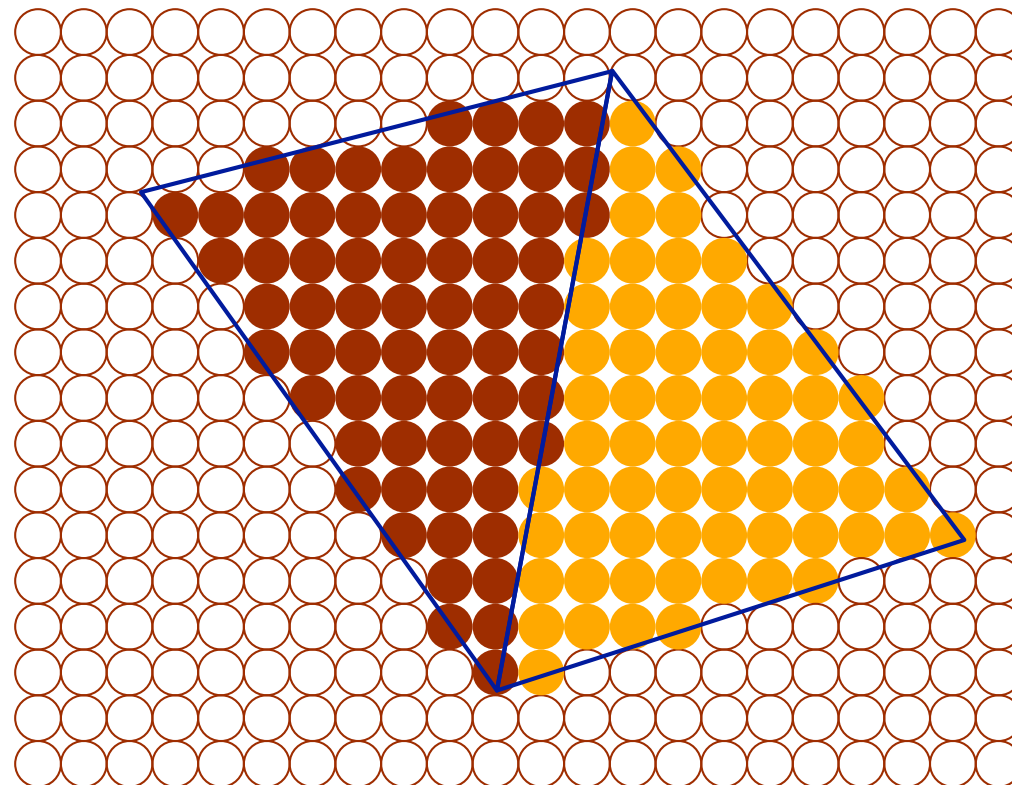
Pixel-Planes 4 (1986)

- Features:
 - Full-size (512 x 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

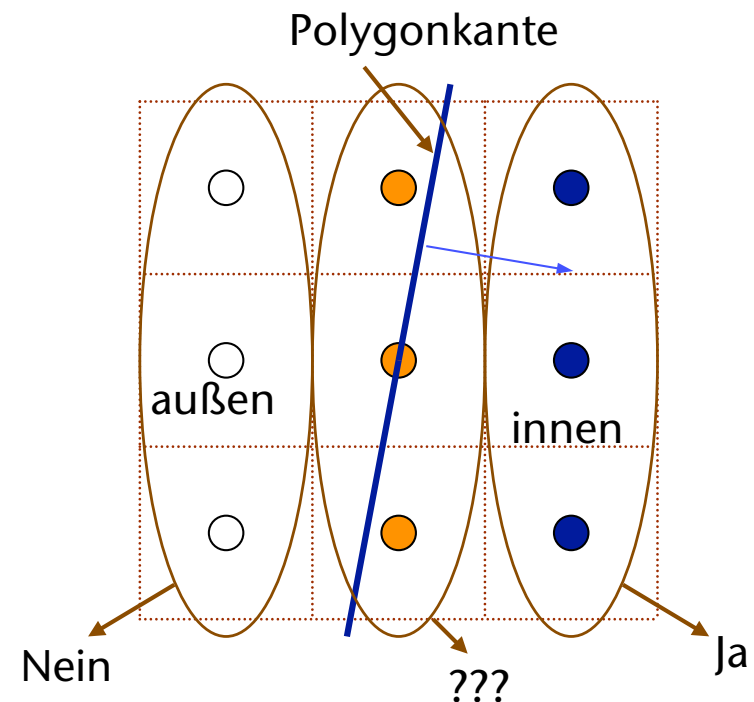


Vorsicht bei angrenzenden Polygonen

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

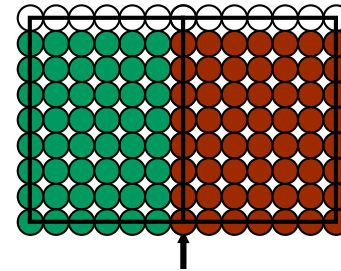


- Pixel vollständig im Polygon → wird gezeichnet
- Pixel zum Teil im Polygon → ... ?
- Vereinbarung (für den Moment): zeichne nur die Pixel, deren **Zentren** im **Inneren** des Polygons liegen
- Problem, falls Zentrum des Pixels genau auf der Ecke des Polygons liegt :
 - Nicht zeichnen → Loch
 - Zeichnen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

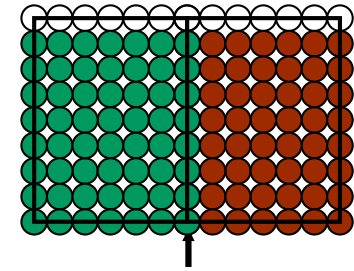


- Problem beim 2x Zeichnen:

- Das zuletzt gezeichnete Polygon "gewinnt"



Rot zuletzt



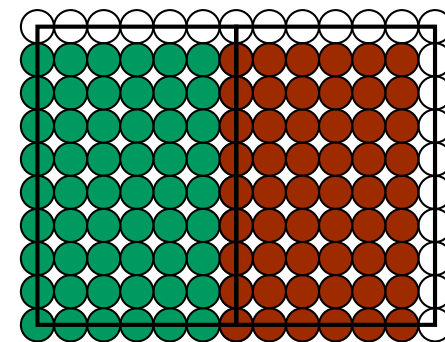
Grün zuletzt

- 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 Kante aufgespannte Halbraumes liegt (erkennt man an der Normale)
 - Verfahrre bei konvexen Polygonen genau wie bei Rechtecken

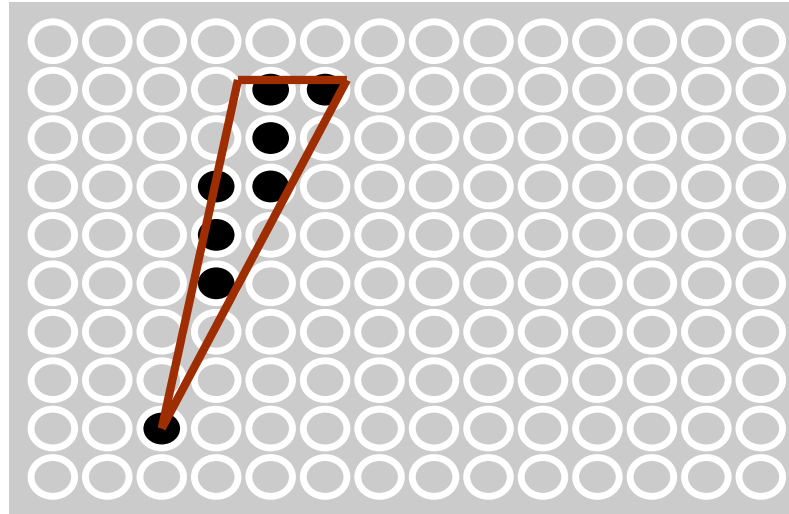
- Folgerungen für Rechtecke:

- Spans lassen das rechteste Pixel weg (falls direkt auf Kante)
 - Bei jedem Polygon fehlt oberster Span (falls direkt auf Kante)

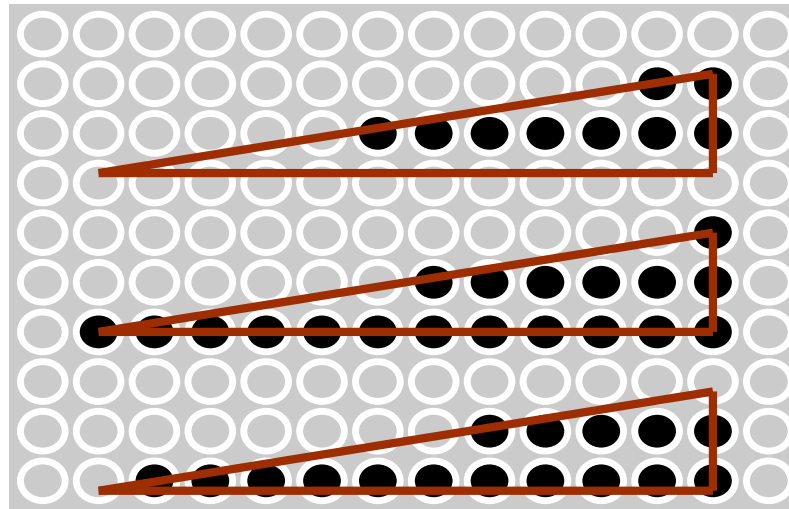


Weiteres Problem

- Sogenannte "*Slivers*":

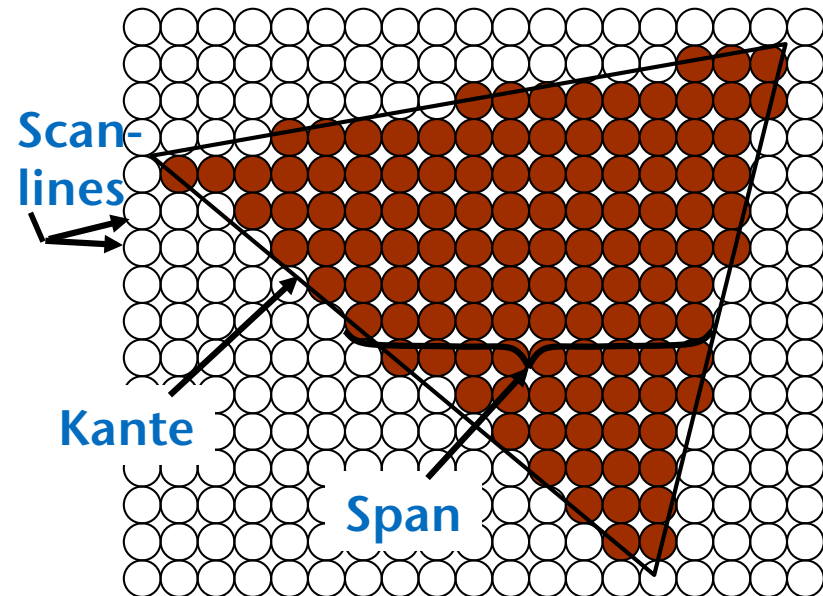


- *Moving Slivers*:



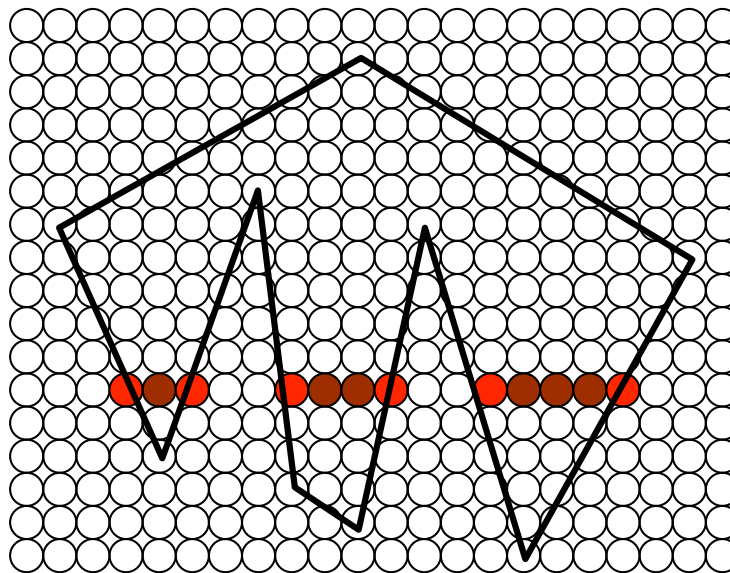
Das allgemeine Konzept der Scan Conversion

- 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

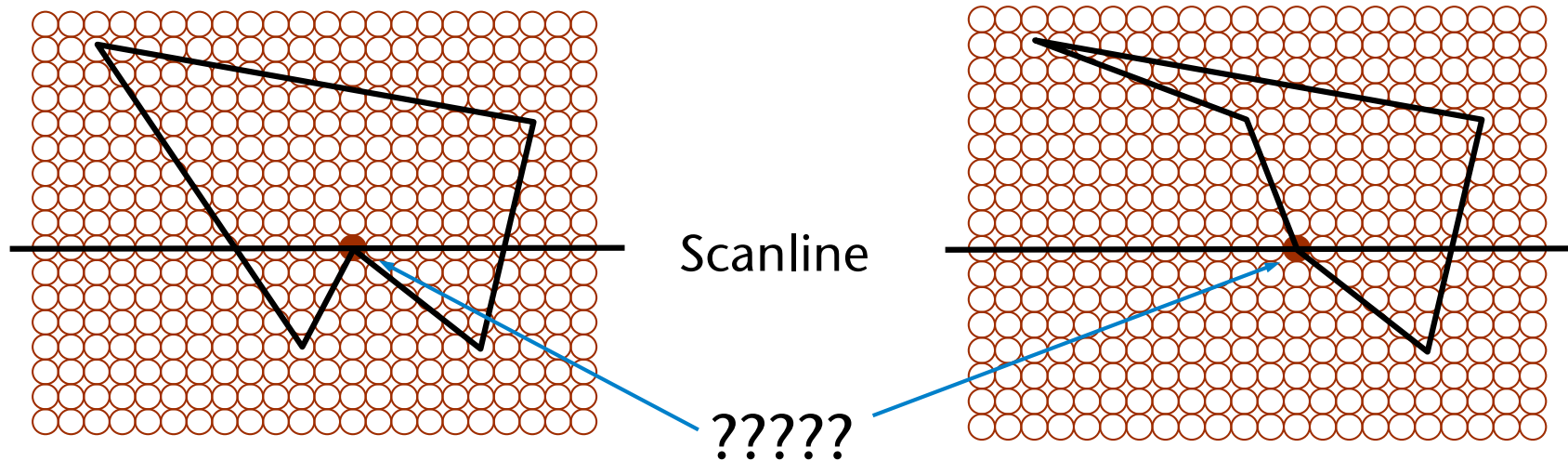
- Annahme: gesamtes Polygon ist auf dem Bildschirm / Framebuffer
- 1. Bestimme alle Punkte auf der Scanline, welche die Kante eines Polygons schneiden
- 2. Sortiere Schnittpunkte von links nach rechts
- 3. Gruppierere Schnittpunkte in *Spans* und färbe die Pixel dazwischen



- Schnittpunkte
- Restliche Punkte des Spans

Entscheidung "innerhalb" / "außerhalb"

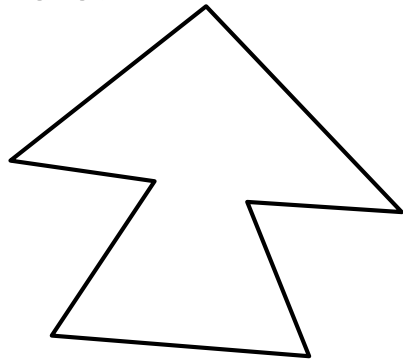
- Wie werten wir Eckpunkte genau auf der Scanline?
 - M.a.W.: begrenzt solch ein Eckpunkt einen Span?



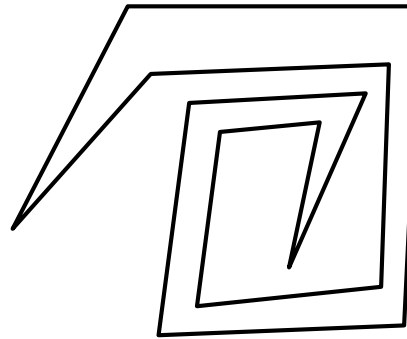
- Lösung: zähle einen Eckpunkt genau dann, wenn er das untere Ende der einen, und das obere Ende der anderen Kante ist
- Alternative: "wackle" am Eckpunkt ein wenig ("perturbation")
 - Dann bekommt man im linken Beispiel 2 oder 0 Schnittpunkte, im rechten genau 1 Schnittpunkt

Füllen nicht-einfacher Polygone

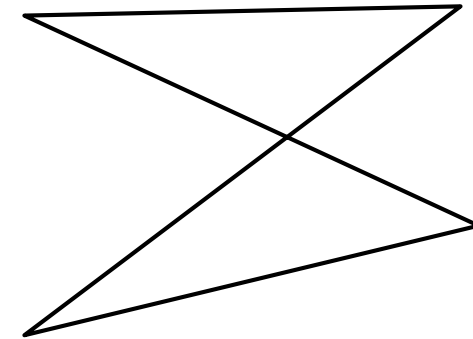
- Def.: Polygon ist **einfach** \leftrightarrow Randkurve hat keinen Schnittpunkt
- Beispiele:



einfach (& hor. konvex)



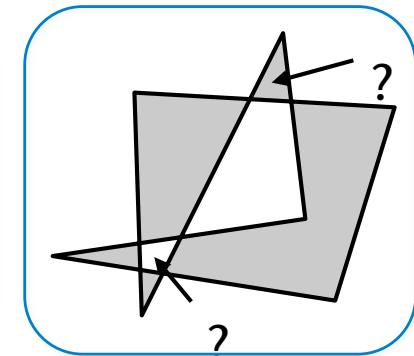
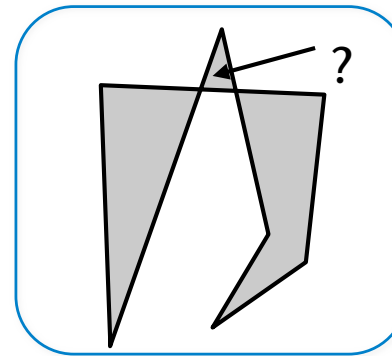
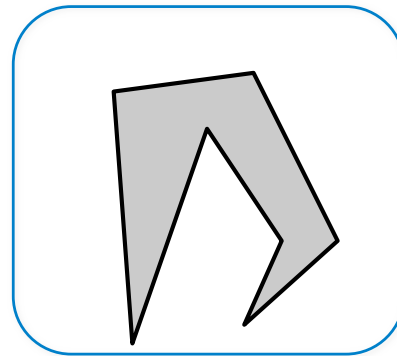
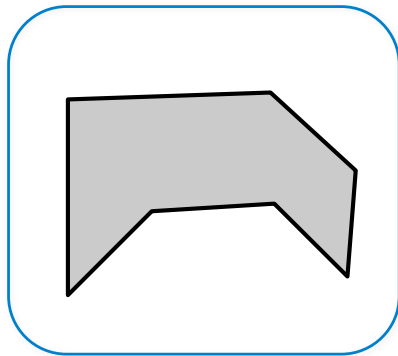
einfach



nicht einfach

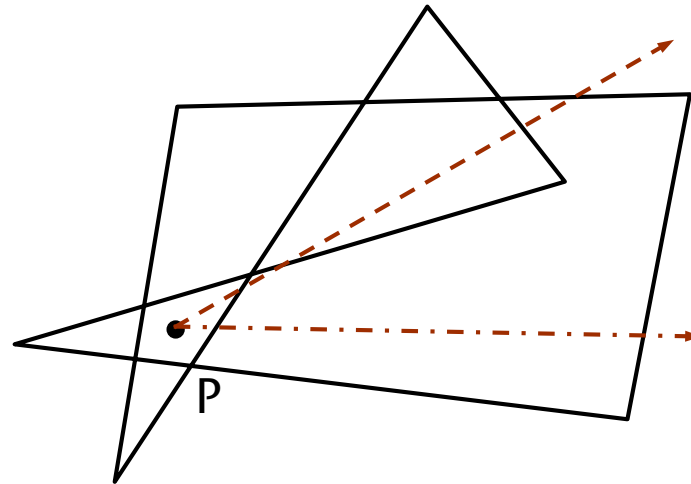
- Eigenschaften einfacher Polygone:
 - Topologisch äquivalent zu einer 2-dimensionalen Scheibe
 - Folge: es gibt ein wohldefiniertes Inneres/Äußeres
- Wie kann man auch *nicht-einfache* Polygone füllen?
 - Definiere für Punkte einen intuitiv „korrekten“ Inside-/Outside-Test

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



Test/Prädikat 1: Odd-Even Rule

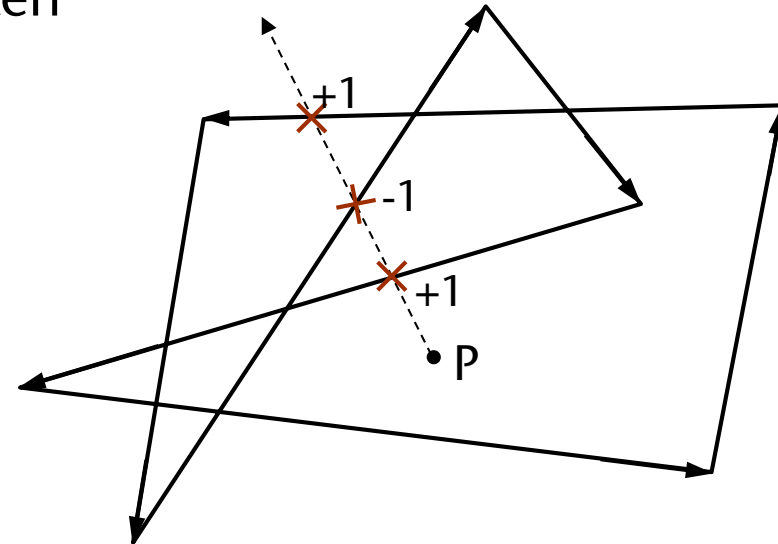
- Zeichne Strahl von Punkt P nach unendlich in irgend eine Richtung
- Zähle Anzahl Schnittpunkte mit dem Kantenzug
- Falls Anzahl ungerade \rightarrow P innerhalb
- Für effiziente Schnittberechnung: wähle horizontalen Strahl

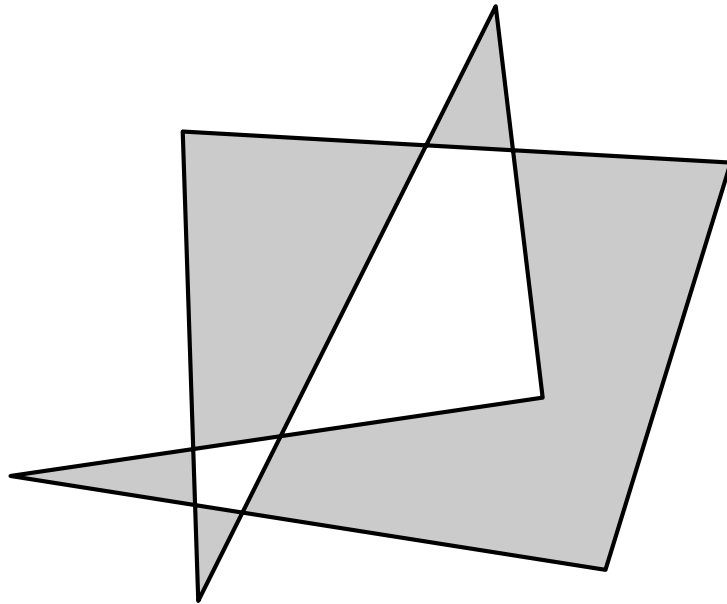


- Vorteil: funktioniert genauso mit Polyedern im 3D (und höherdim.)
- Achtung, falls Strahl Eckpunkt genau trifft!

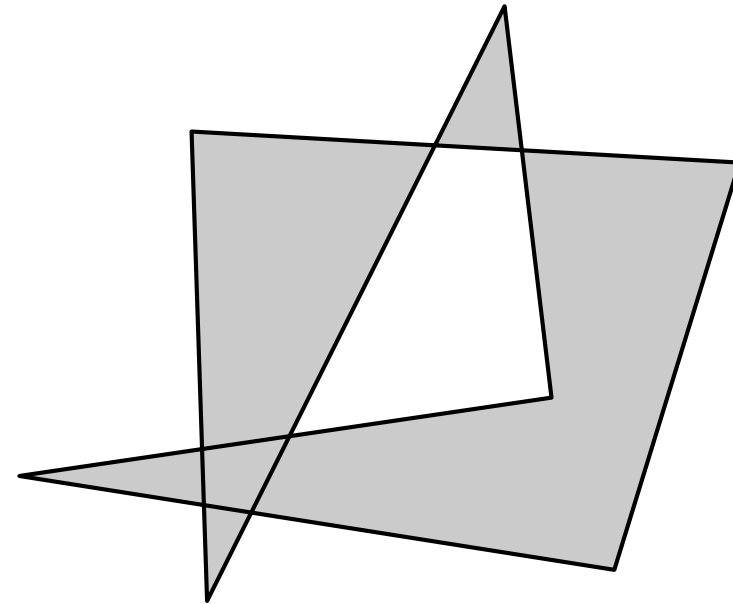
Test/Prädikat 2: Winding-Number Rule

- 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 $w \neq 0 \rightarrow P$ innerhalb
- Anmerkung: damit definiert man gleichzeitig „positiv“ bzw. „negativ“ orientierte Regionen





Odd-even Rule



Non-zero Winding Number



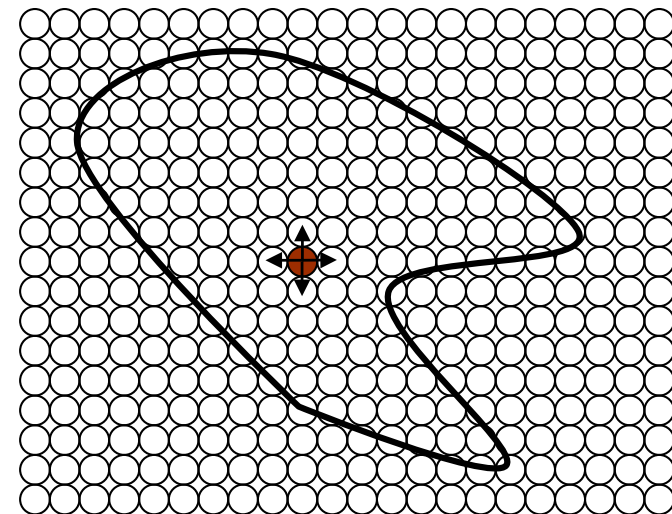
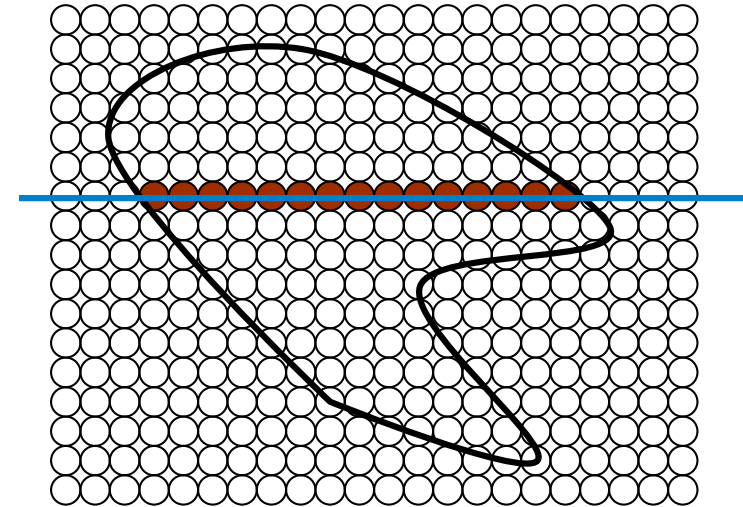
Ist die weiße Linie genau ein einfaches Polygon?



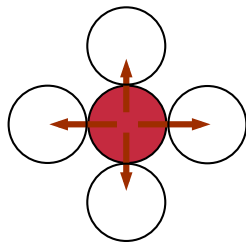
Robert Bosch

Füllen von Regionen

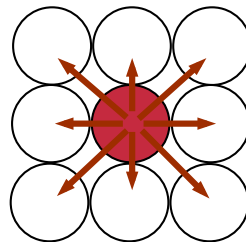
- Gegeben:
 - Rand einer Region, definiert durch Pixel mit einer bestimmten Farbe
 - Häufig in 2D-Zeichenprogrammen
 - Ähnliche Aufgabe zu Polygon-Scan-Conversion
- Erste Methode: wie Polygon-Scan-Conversion
- Zweite Methode: **Flood Fill**
 - Wähle ein Pixel innerhalb des Polygons
 - Färbe rekursiv angrenzende Pixel bis das gesamte Polygon gefüllt ist



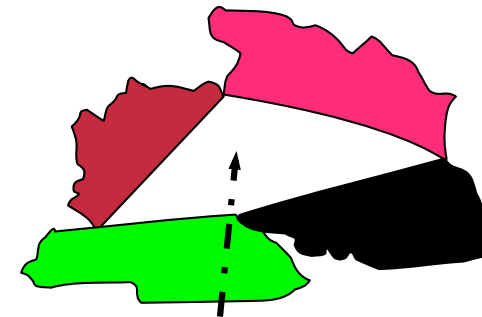
- Region ist identifiziert durch bestimmte Farbe (z.B. Weiß)
- Wähle ein Pixel innerhalb der Region
 - Region ist definiert als **zusammenhängendes** Gebiet mit **alter Farbe**
- Rekursion:
 1. Hat Pixel die alte Farbe, dann weise diesem Pixel die Füllfarbe zu
 2. Färbe rekursiv alle diejenigen Nachbarn,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



4 Verbindungen

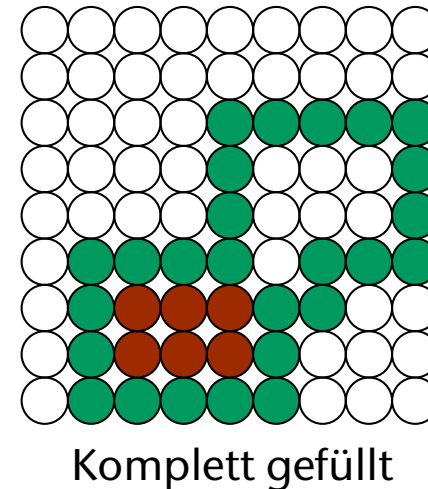
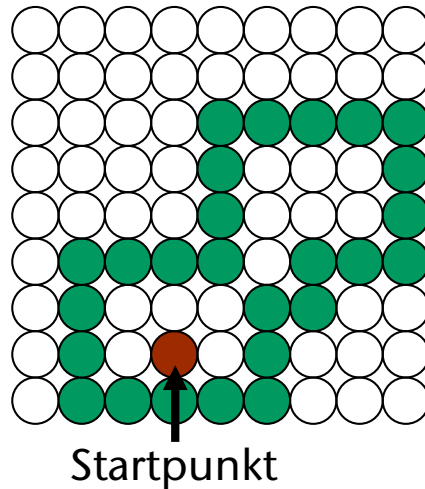


8 Verbindungen



Zu füllende Region

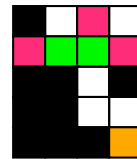
- Z.B. bei 4-Nachbarschaft pro Punkt:



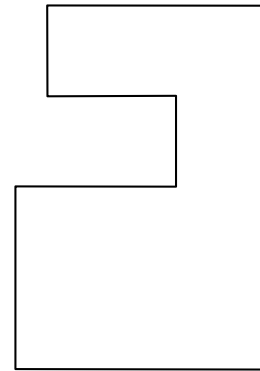
- Ein weiteres Problem: der Algorithmus hat große Rekursionstiefe
 - Kann Stack aus Spans verwenden um Rekursionstiefe zu verringern
 - Nur eine Rekursion pro Scanline
 - Verkompliziert aber der innere "Logik" des Algo

Bereiche mit Mustern

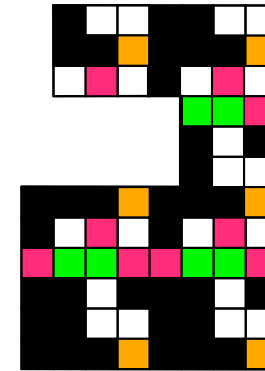
- Oft möchten wir einen Bereich mit einem Muster (z.B. gestrichelt) versehen, nicht nur eine Farbe (*stippling*)
- Definiere ein $n \times m$ **Pixmap** (oder **Bitmap**), welche wir auf den Bereich abbilden möchten:



5x4 pixmap



Mit Muster zu
versehendes Objekt

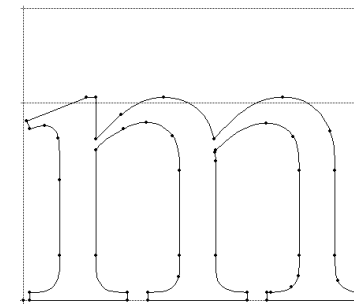


Gemustertes
Objekt

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

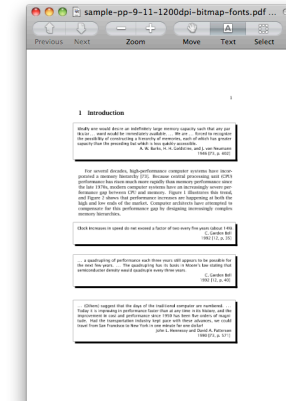
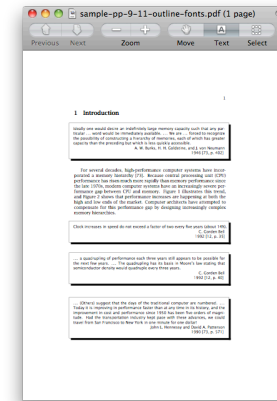
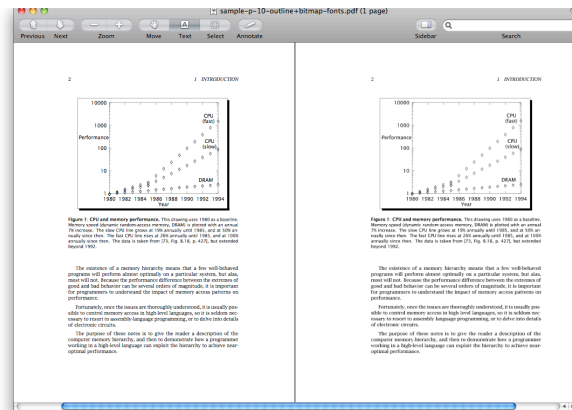
Font-Rendering

- 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



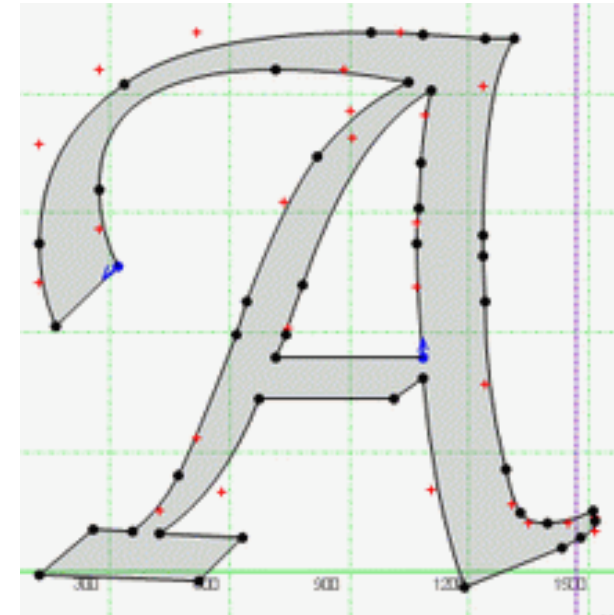
- **Glyph** = graphische Repräsentation eines oder mehrerer Zeichen

A A A A A A *A* fi

- **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, ...

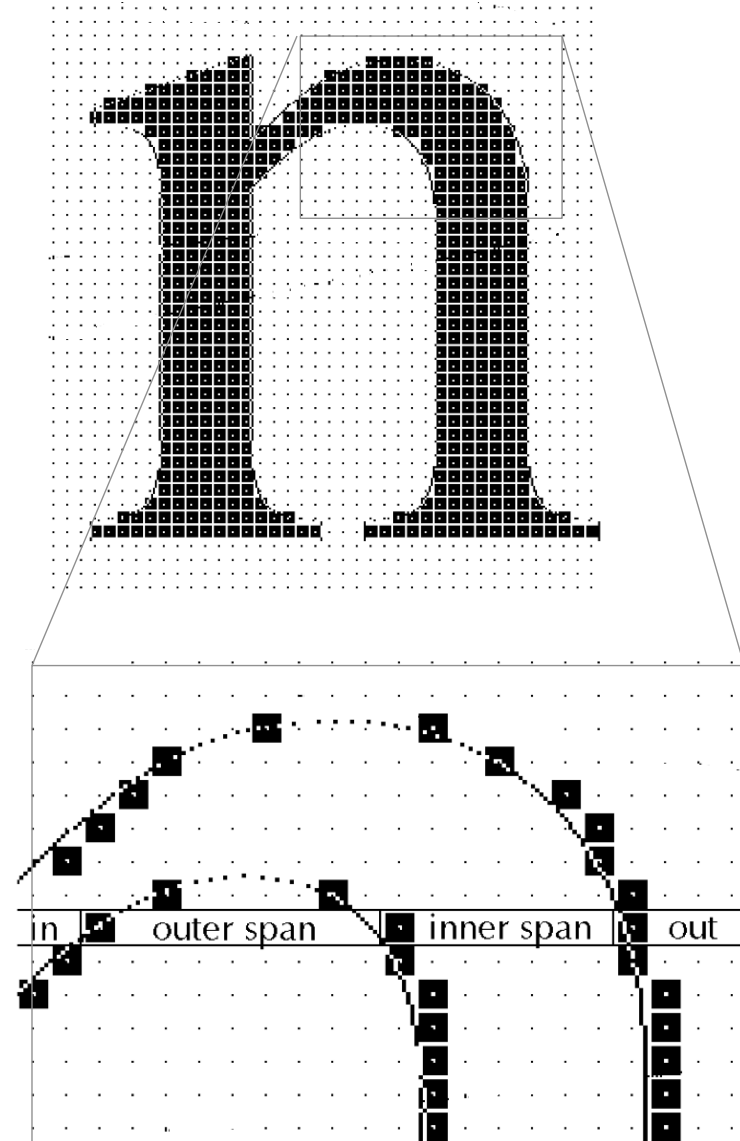
Beschreibung von Outline-Fonts

- 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



- Annahme zunächst:
 - Die einzelnen Pixel sind sehr klein im Vergleich zu dem Zeichen
 - Innerhalb eines Pixels kann man die Kontur durch eine Gerade approx.
 - Die Überdeckung des Pixels $> 50\%$ \Leftrightarrow 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 \rightarrow **Flags**
 - Fülle die inneren Spans

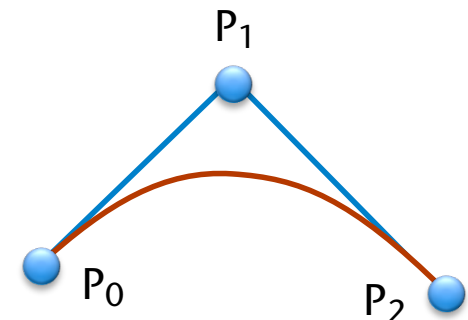


- Outlines setzen sich zusammen aus quadratischen und kubischen Bézier-Kurven

- Quadratische Bézier-Kurven:

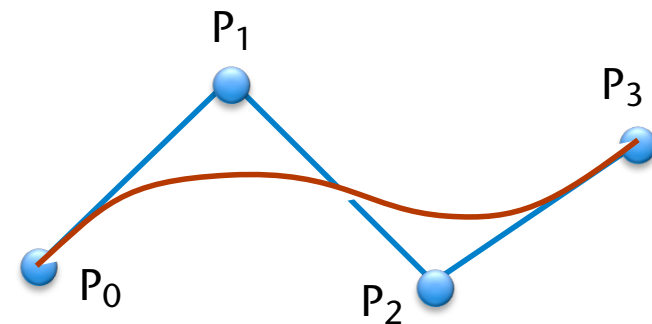
- Definiert durch ein Kontroll-Polygon aus 3 Punkten
- Polynom 2-ten Grades:

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



- Kubische Bézier-Kurven:

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



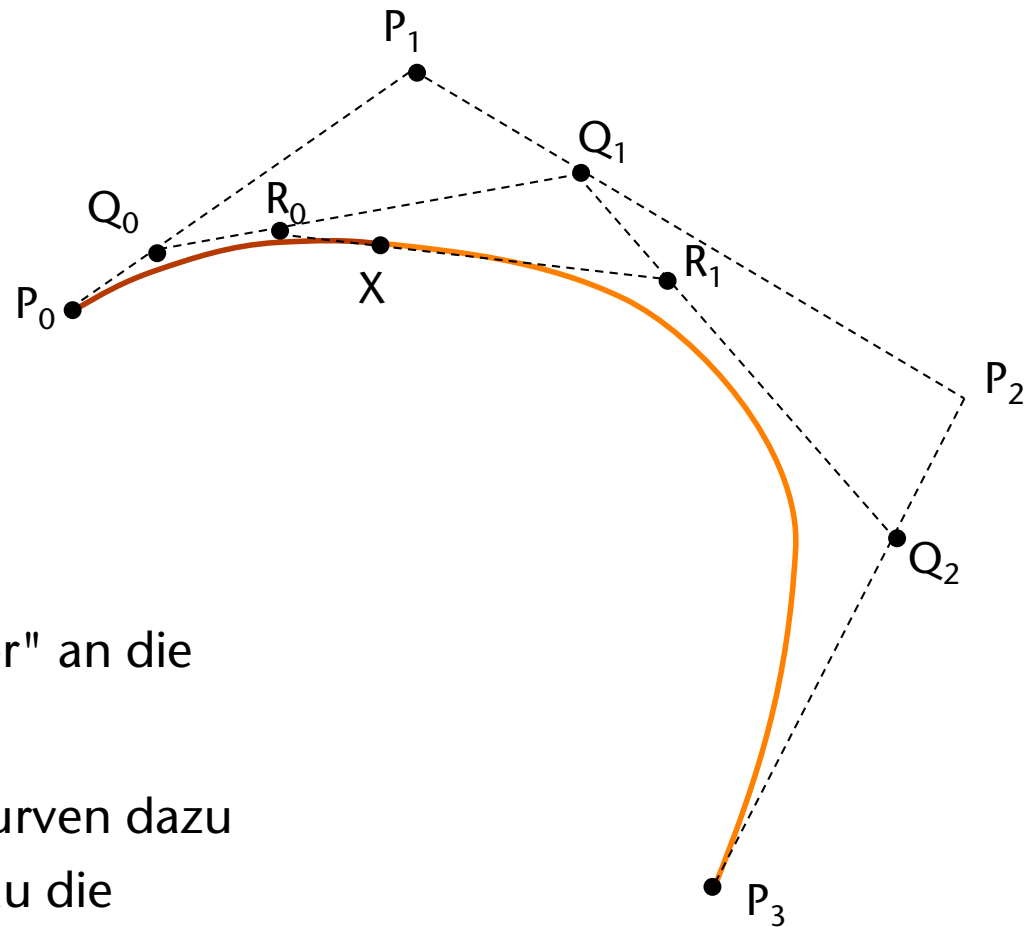
$$P(t) = (1 - t)^3 P_0 + 3(1 - t)^2 t P_1 + 3(1 - t)t^2 P_2 + t^3 P_3, \quad t \in [0, 1]$$

■ Approximation durch rekursive Subdivision:

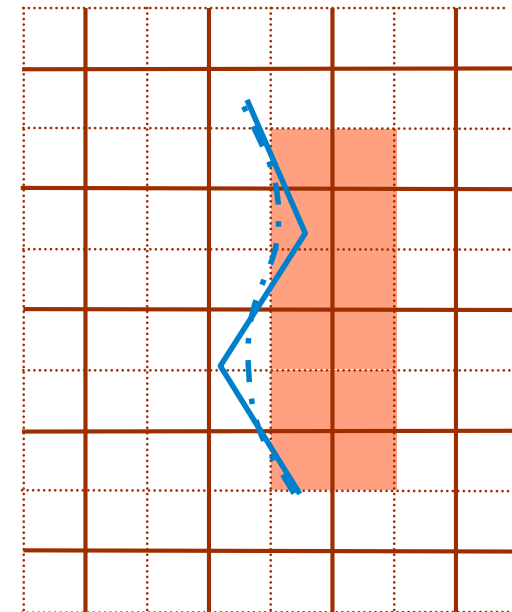
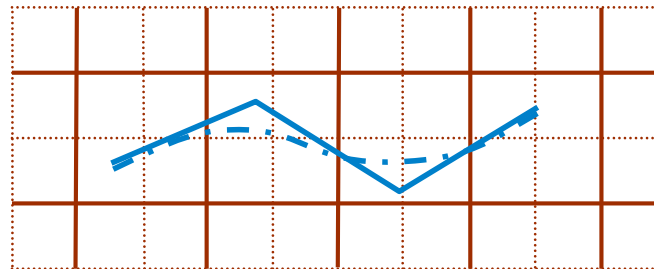
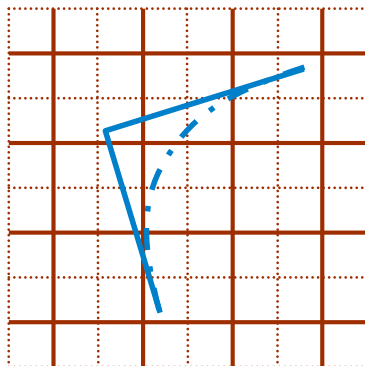
- Aus P_0, \dots, P_3 kann man **zwei neue Kontroll-Polygone**

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



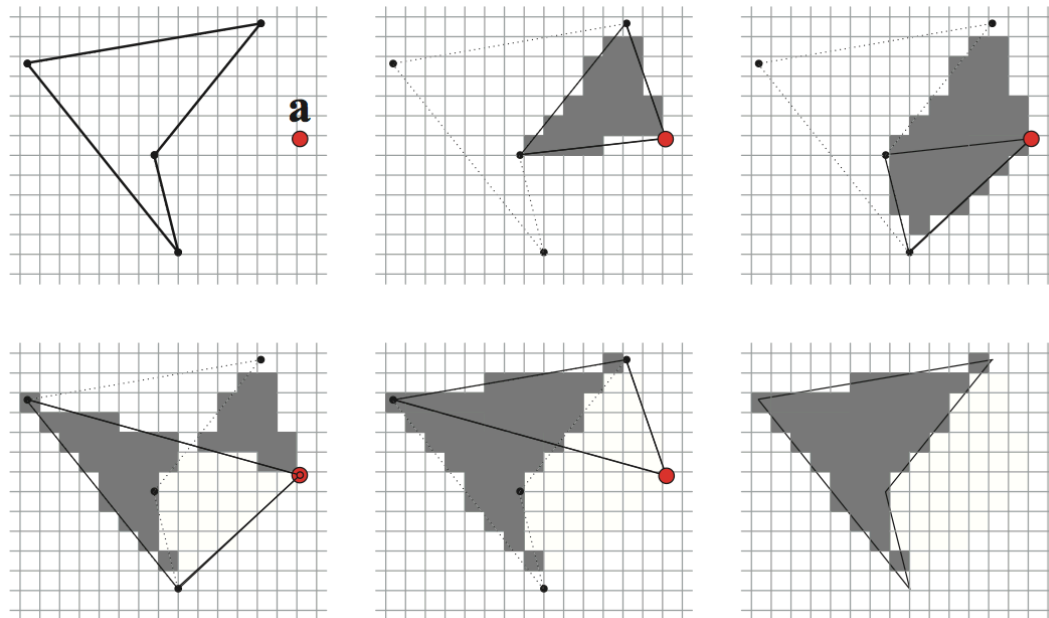
- Betrachte einzeln jede Bézier-Kurve nacheinander, d.h., betrachte deren Kontroll-Polygon
- Fallunterscheidung & Rekursion:
 1. Kontroll-Polygon schneidet keine (horizontale) Scanline → verwerfen
 2. Kontroll-Polygon schneidet eine oder mehrere Scanlines, und schneidet keine vertikale Gitterlinie →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



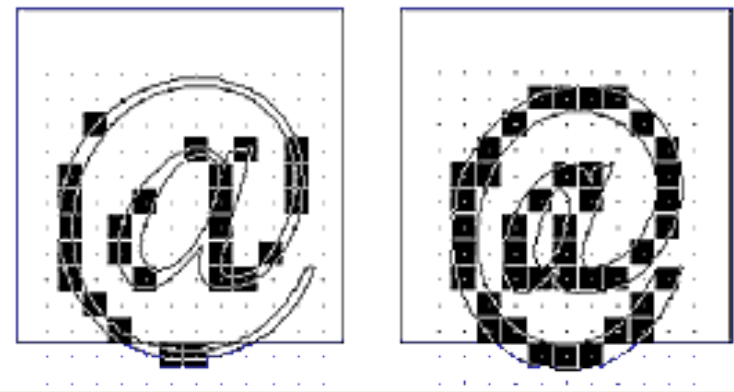
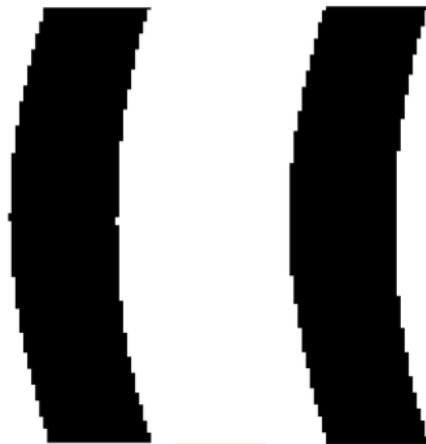
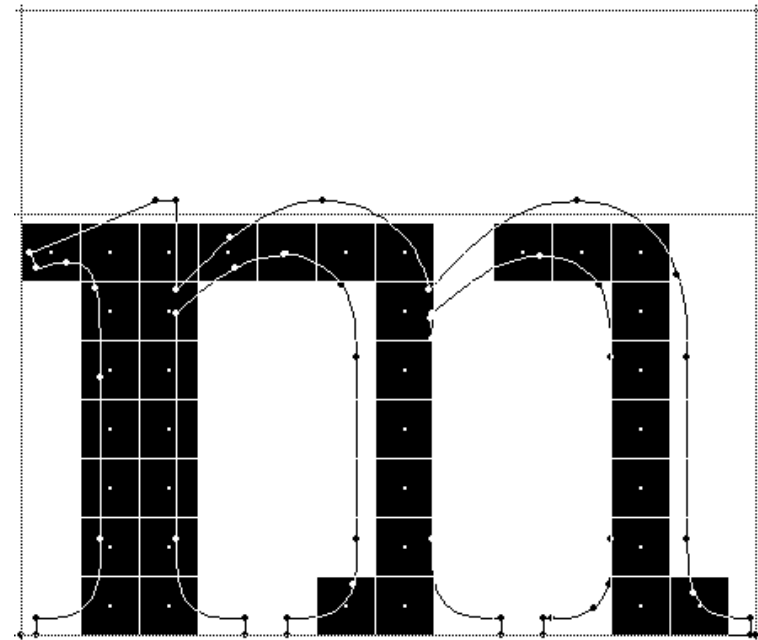
Der XOR-Algorithmus

- 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:

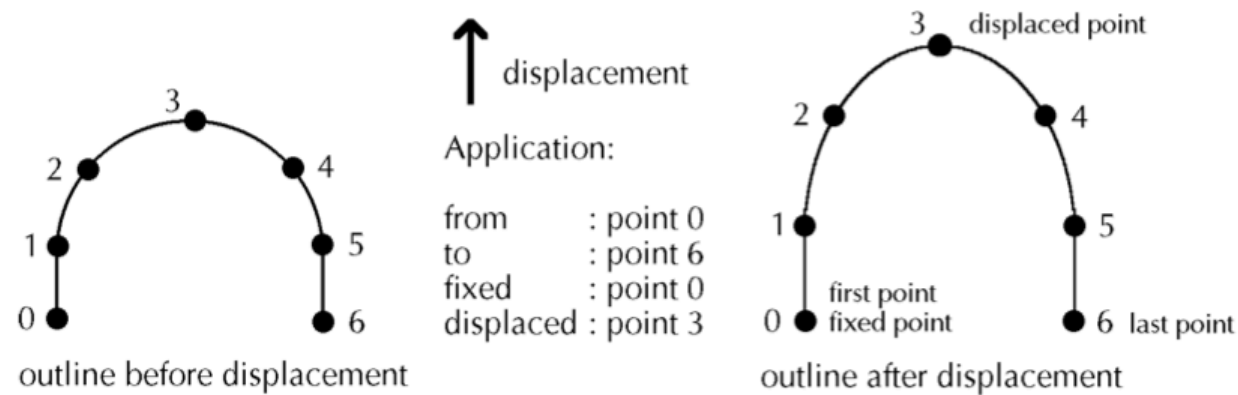


- Bei kleinen Font-Größen können, je nach "Phase", folgende Probleme auftreten:
 - Drop-Outs
 - Ungleiche Dicke der Stämme
 - Serifen in verschiedene Richtung
- **Phase** = Abstand zwischen linkem Rand der BBox und vertikale Gitterlinie links davon



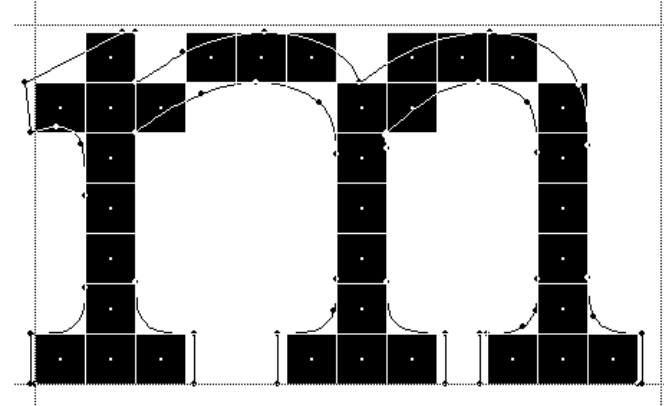
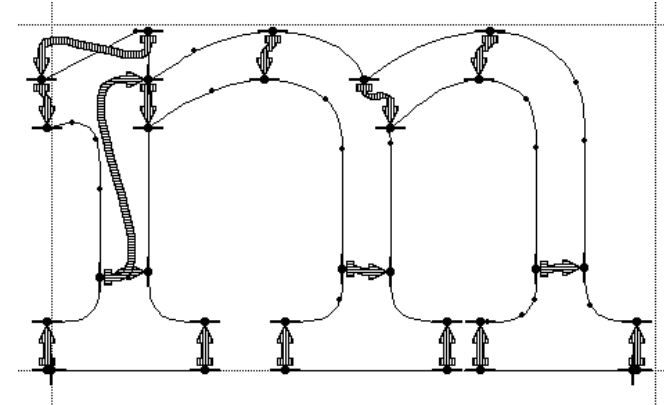
Hinting (grid fitting)

- Lösung: der Rasterizer verzerrt die Konturkurven vor dem Rendering 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;



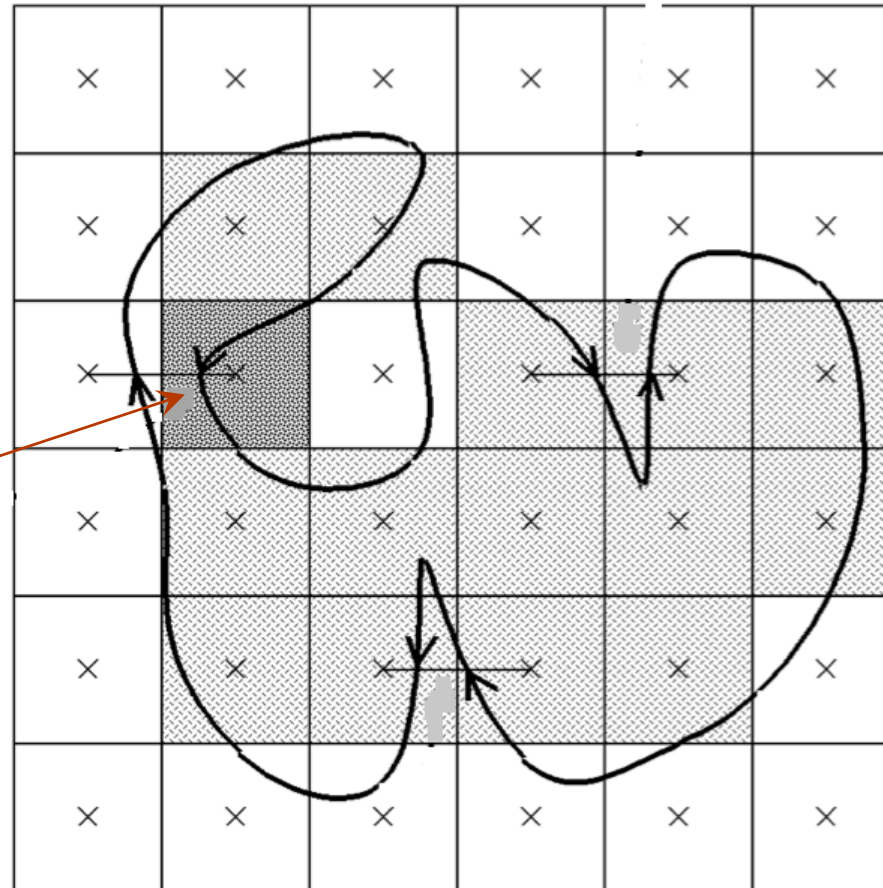
- Hinting (Fortsetzung):
 - welche anderen Punkte dann mitverschoben werden müssen (**Constraints**)
 - Muß vom Font-Designer gemacht werden

- NB: Diese Art Hinting wird nur bei TrueType-Fonts gemacht
 - Völlig anders bei Type1 ...



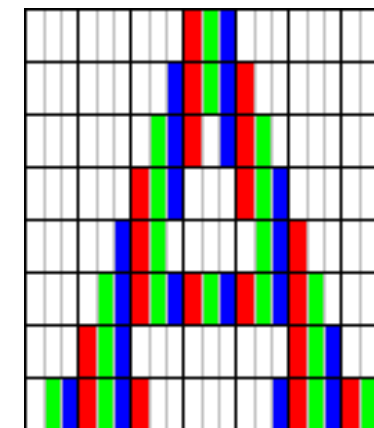
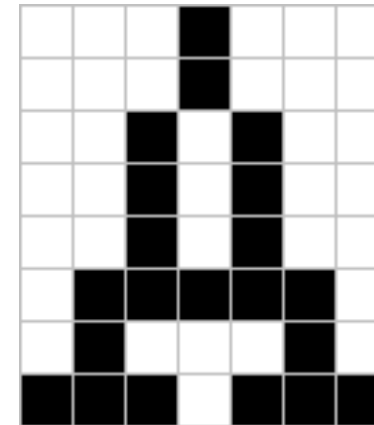
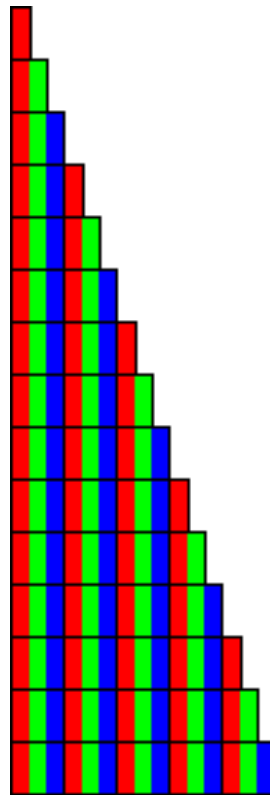
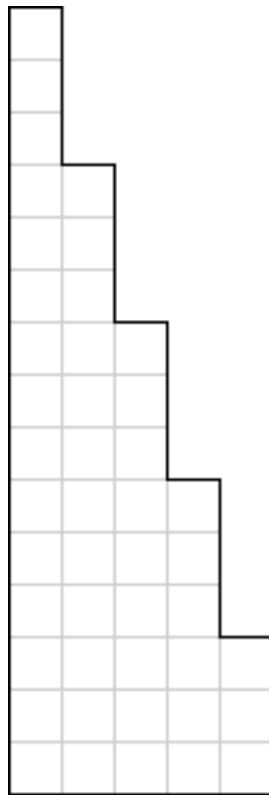
- Dropout-Kontrolle:

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



Sub-Pixel Font-Rendering

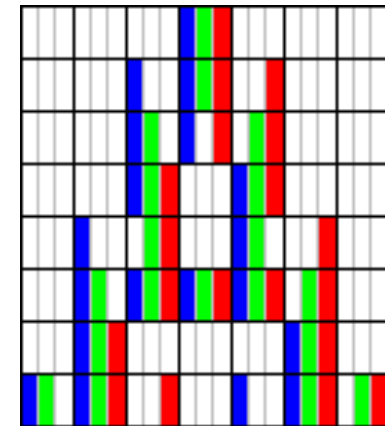
- Die Idee: betrachte jedes "Primär-Pixel" (R, G, und B) als eigenständiges Pixel:



- 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)
 - Skalieren Fonts vor dem Rasterisieren um Faktor 2
 - Modelliere die menschliche Farbwahrnehmung und korrigiere die Color Fringes entsprechend

