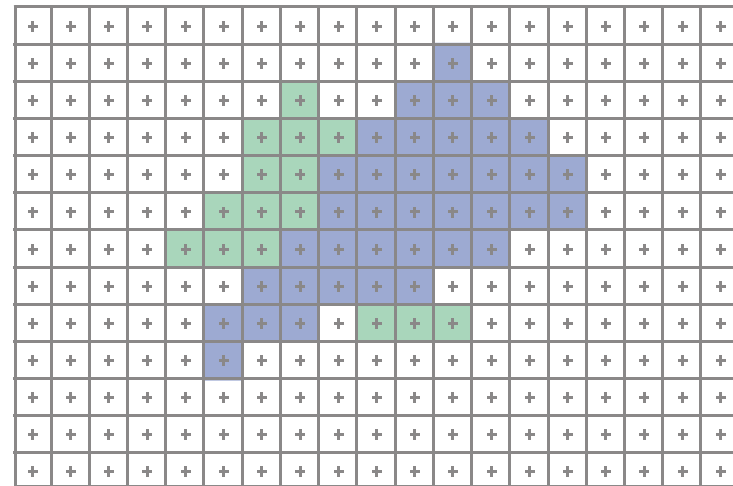


Computer–Graphik I

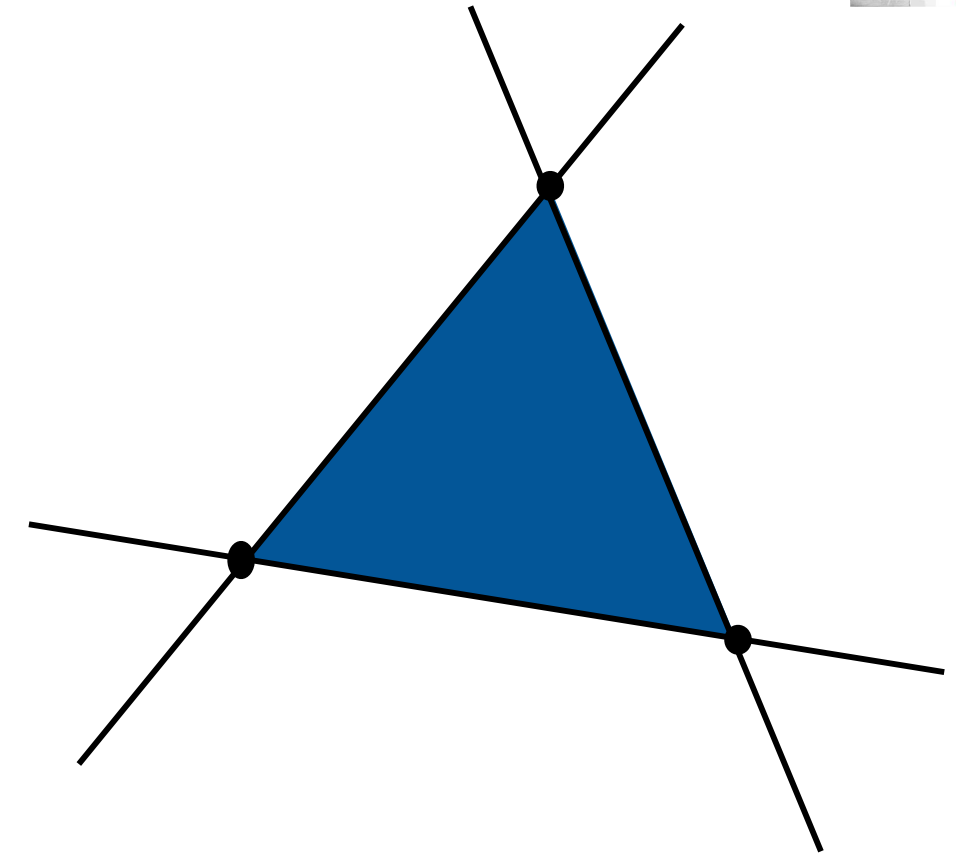
Polygon Scan Conversion, Triangulation, Filling, etc.



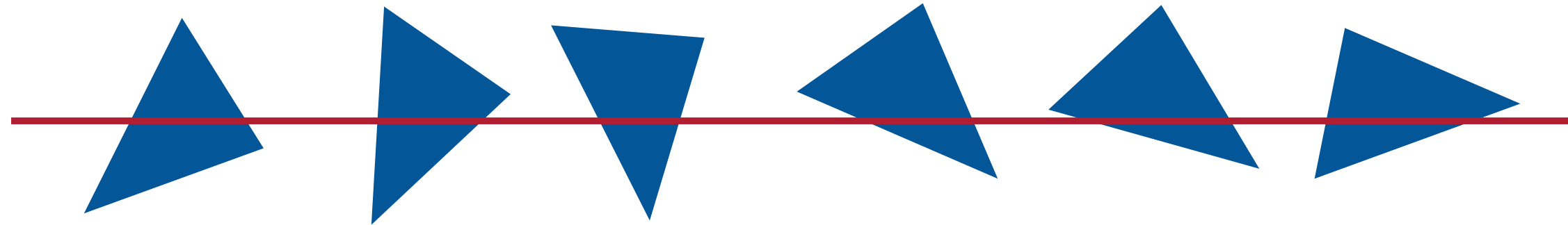
G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

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" geometrisches Objekt, das echt 2-dimensional ist
- Somit sind Dreiecke sehr einfach (sowohl mathematisch wie auch geometrisch)



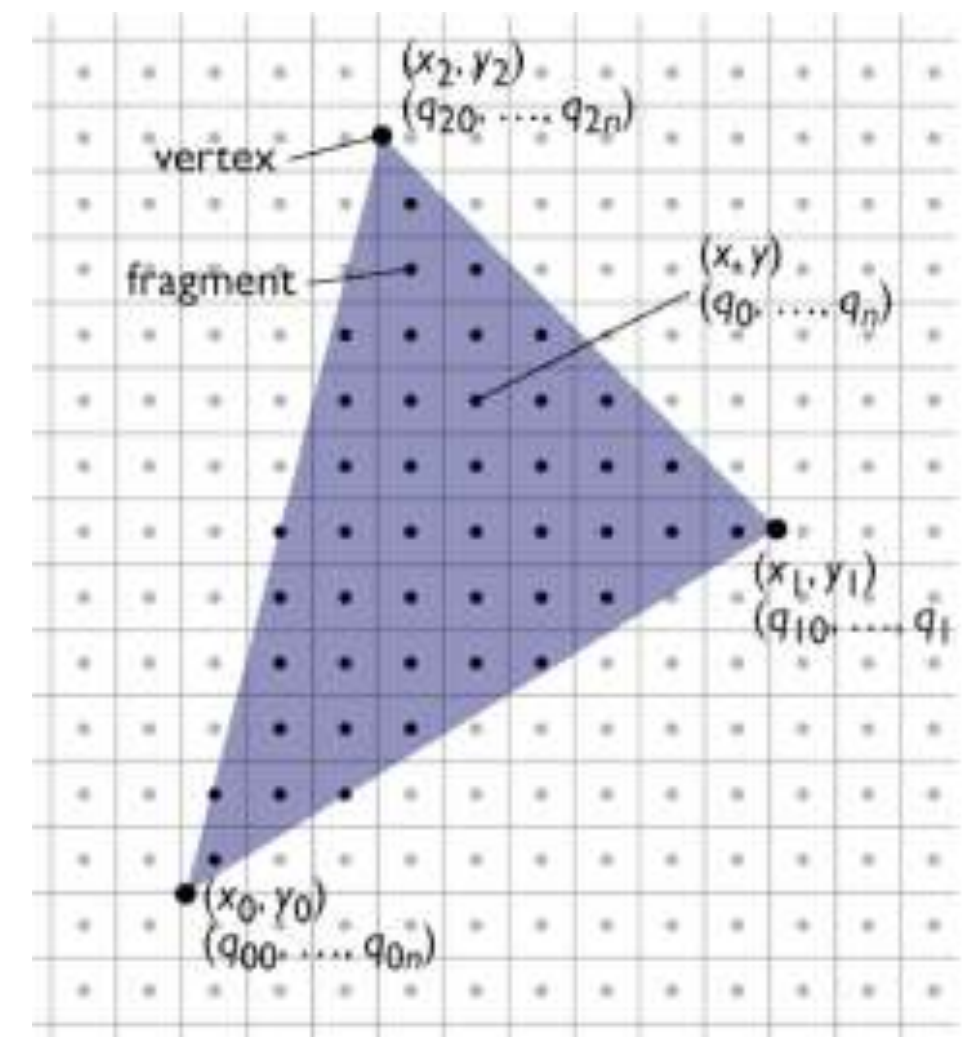
- Dreiecke sind *immer* konvex $\rightarrow$ egal wie man ein Dreieck dreht, es gibt nur ein Schnittintervall (= *Span*) für jede Gerade (*Scanline*)



- 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

Rasterisierung von Dreiecken

- Eingabe = Vertices:
 - Drei 2D **Vertices** (Eckpunkte) im **screen space** (integer coord):
 $(x_0, y_0); (x_1, y_1); (x_2, y_2)$
 - Mit Attributen q für jeden Vertex, z.B. Farbe
- Ausgabe = Fragmente:
 - Ganzzahlige Pixel-Koordinaten (x, y)
 - Interpolierte Attributwerte q_{xy}



Erinnerung

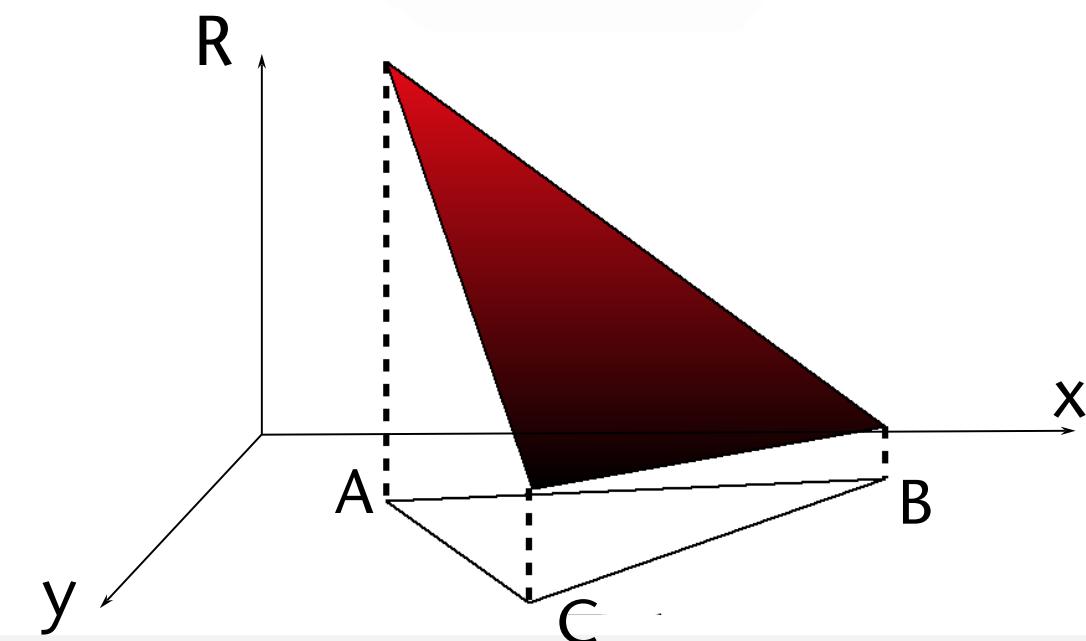
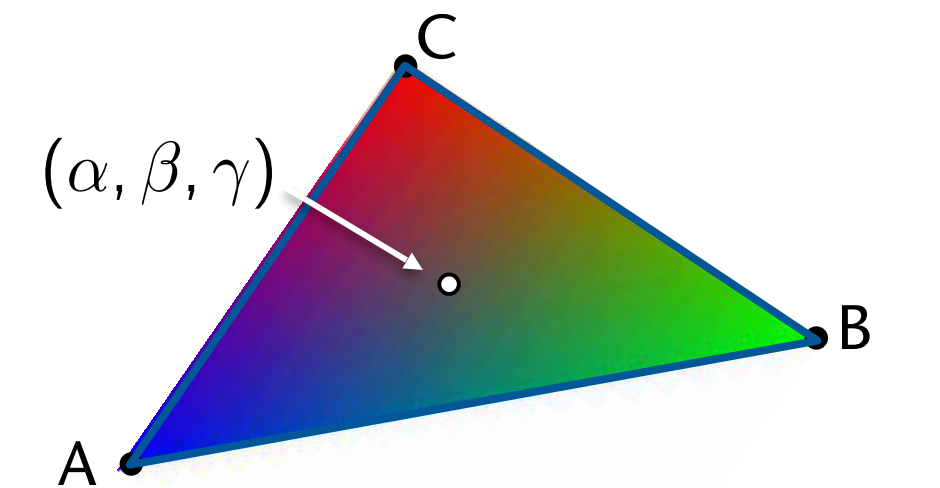
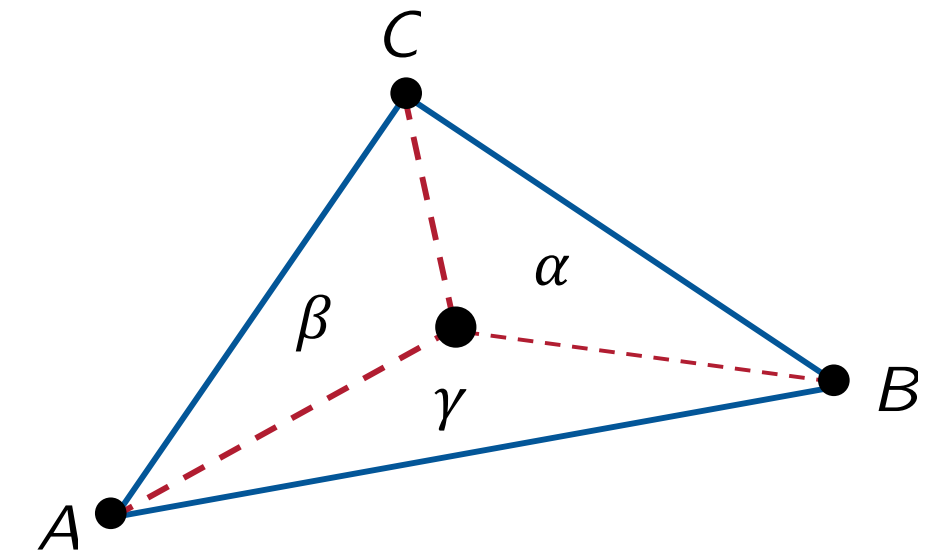
- Prädikat (Test) ob Punkt im Dreieck liegt: verwende baryzentrische Koordinaten

$$X = \alpha A + \beta B + \gamma C \quad \alpha > 0 \wedge \beta > 0 \wedge \gamma > 0$$

- Interpolation von Farben im Dreieck:

$$\begin{pmatrix} r_X \\ g_X \\ b_X \end{pmatrix} = \alpha \begin{pmatrix} r_A \\ g_A \\ b_A \end{pmatrix} + \beta \begin{pmatrix} r_B \\ g_B \\ b_B \end{pmatrix} + \gamma \begin{pmatrix} r_C \\ g_C \\ b_C \end{pmatrix}$$

- Bemerkung: das ist im Grunde eine Interpolation der 3 Farbkanäle unabhängig voneinander



- Das Verfahren:

- Berechne baryzentrische Koordinaten für alle Pixel (-mittelpunkte)
- Falls innerhalb des Dreiecks: berechne interpolierte Farbe und setze das Pixel
- Bemerkung: $(x_{\min}, x_{\max}, y_{\min}, y_{\max})$ ist die **Bounding-Box** des Dreiecks

- Optimierung (nur bei sequ. Berechnung):

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

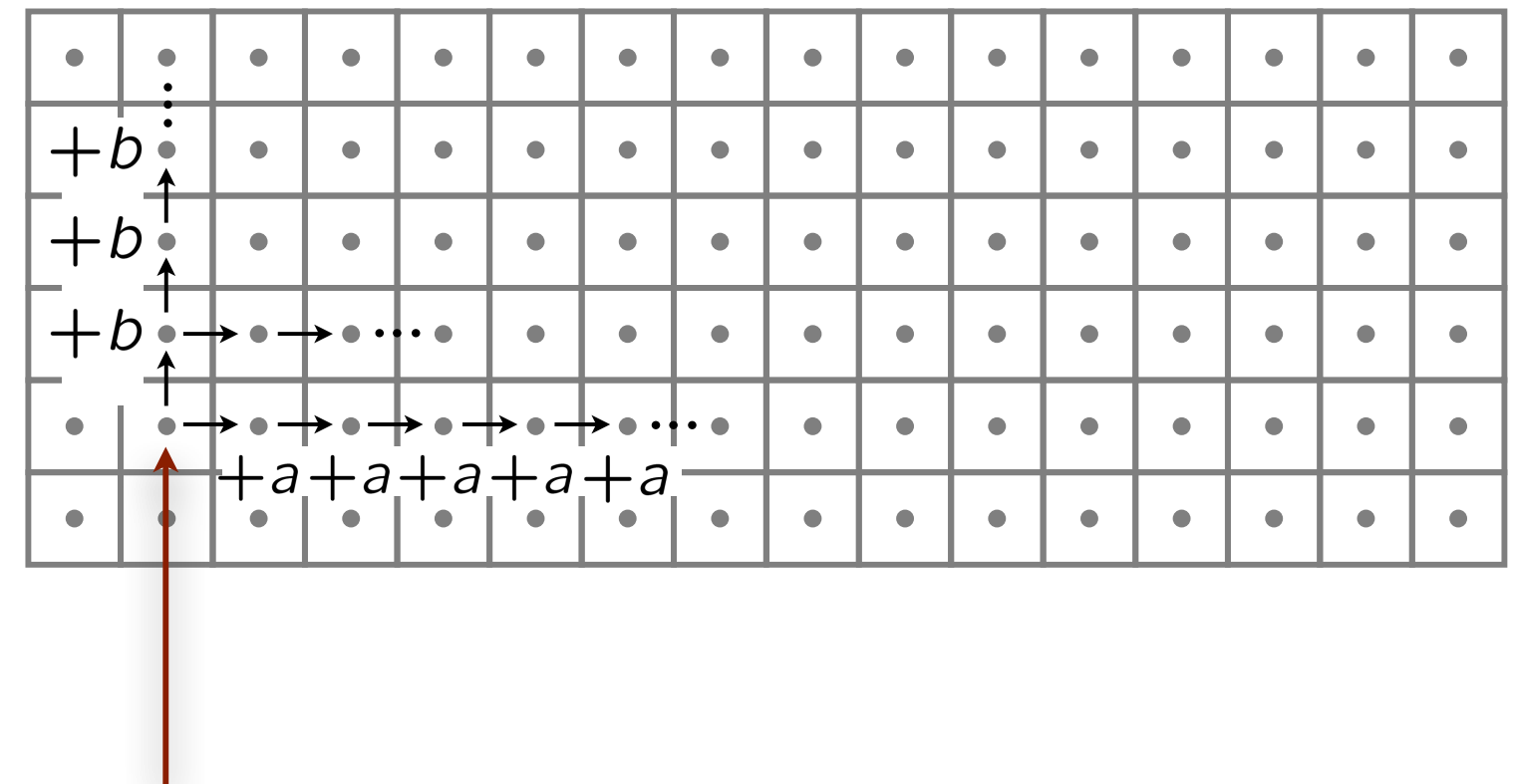
$$\alpha = ax + by + c$$

Dito für β, γ

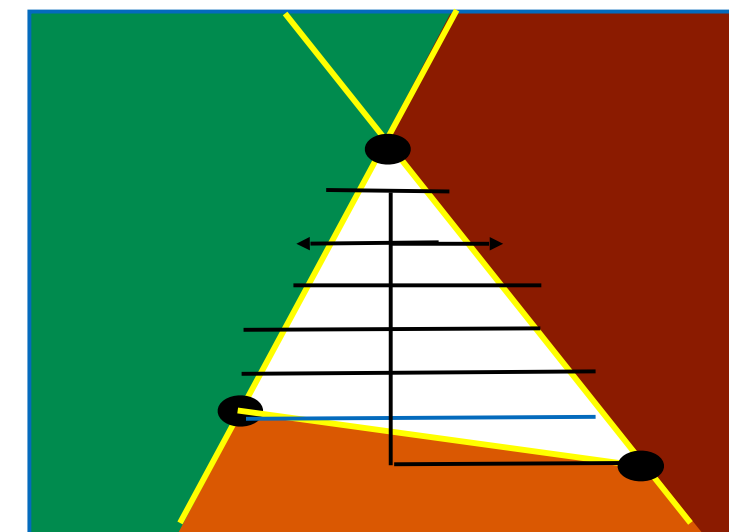
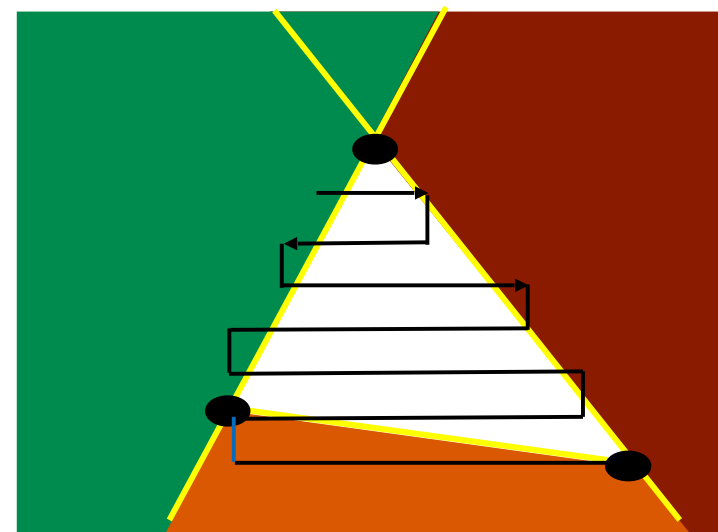
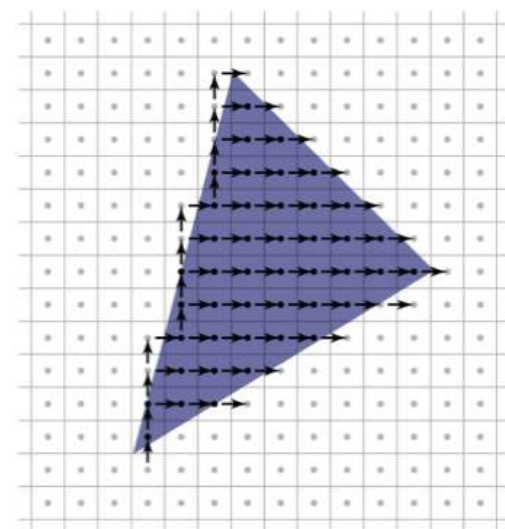
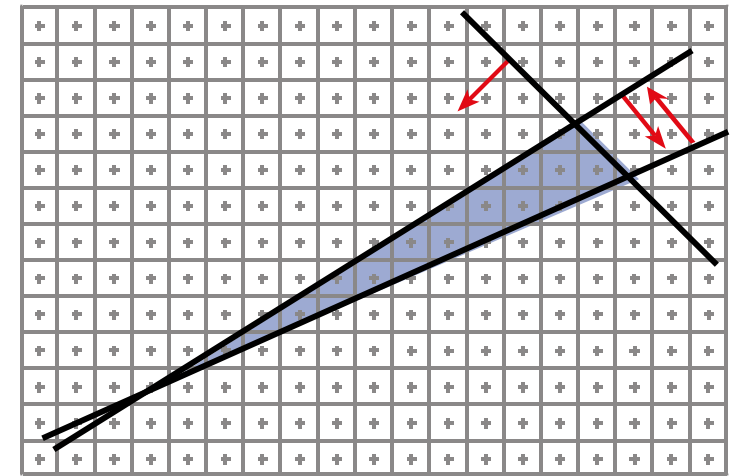
- Verwende DDA-Ansatz, um diese

inkrementell (auf Gitter) auszuwerten: $\alpha(x+1, y) = \alpha(x, y) + a$, $\alpha(x, y+1) = \alpha(x, y) + b$

```
for y = y_min ... y_max:
  for x = x_min ... x_max:
    berechne  $\alpha, \beta, \gamma$ 
    if  $\alpha > 0$  and  $\beta > 0$  and  $\gamma > 0$ :
       $C = \alpha C_A + \beta C_B + \gamma C_C$ 
      zeichne Pixel (x,y) mit Farbe C
```



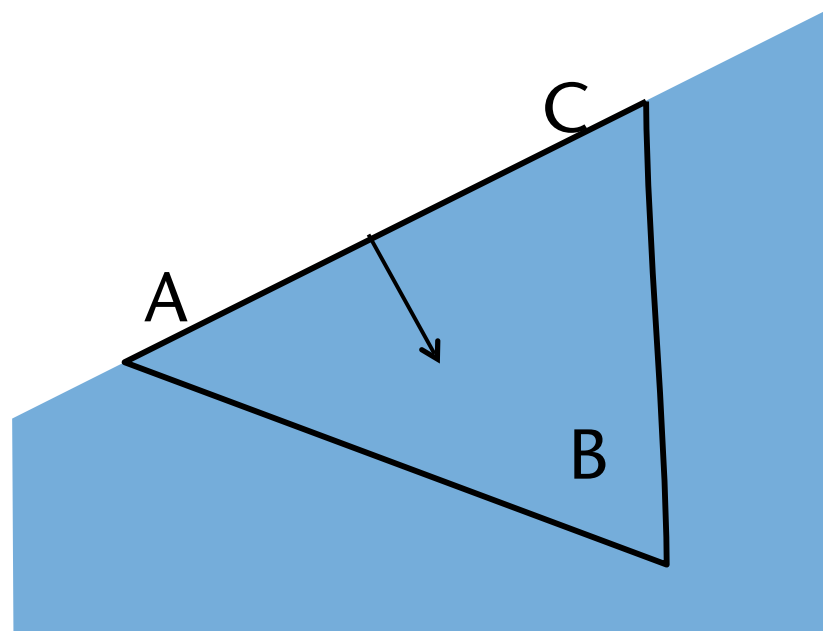
- Problem: wenn das Dreieck lang und schmal ist, dann werden viele unnötige Berechnungen durchgeführt
- Erinnerung: Dreieck ist konvex
 - Folge: wenn man in der x–Schleife einmal ein Pixel außerhalb erreicht, dann sind alle folgenden in der selben Scanline auch außerhalb
- Optimierung:



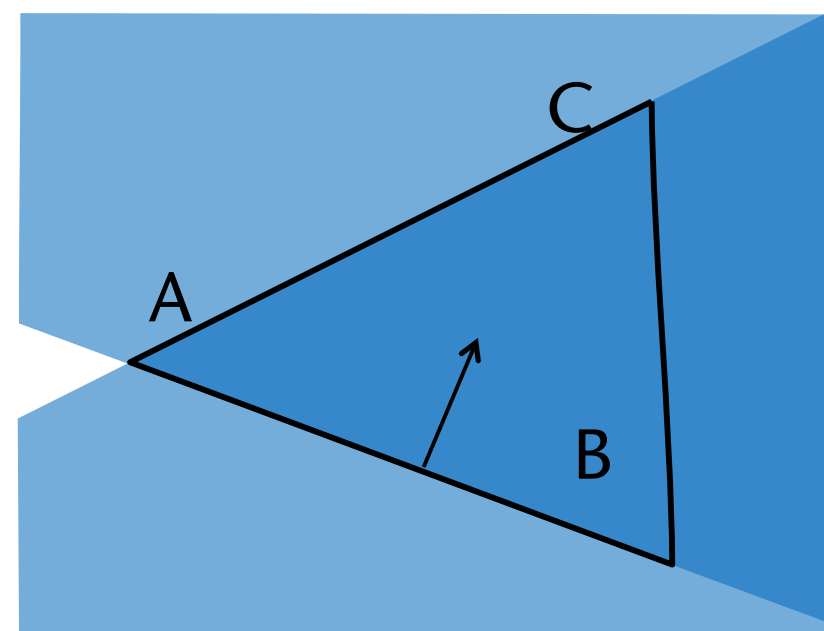
- Weiterer Vorteil des Algorithmus von Pineda: lässt sich trivial parallelisieren

Alternative Betrachtungsweise

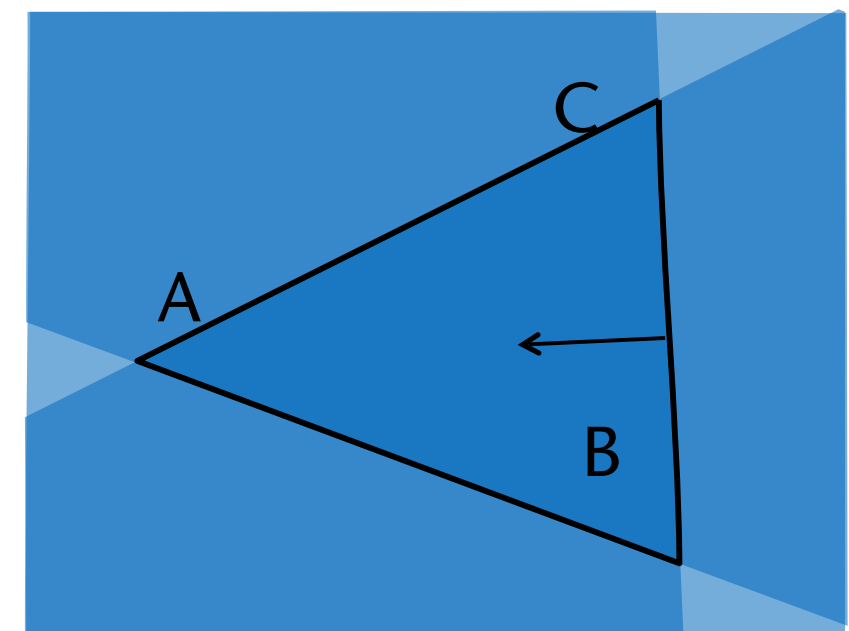
- In der Ebene ist ein Dreieck die Schnittmenge von 3 Halbebenen:



$$(X - C) \cdot \mathbf{n}_{AC} > 0$$



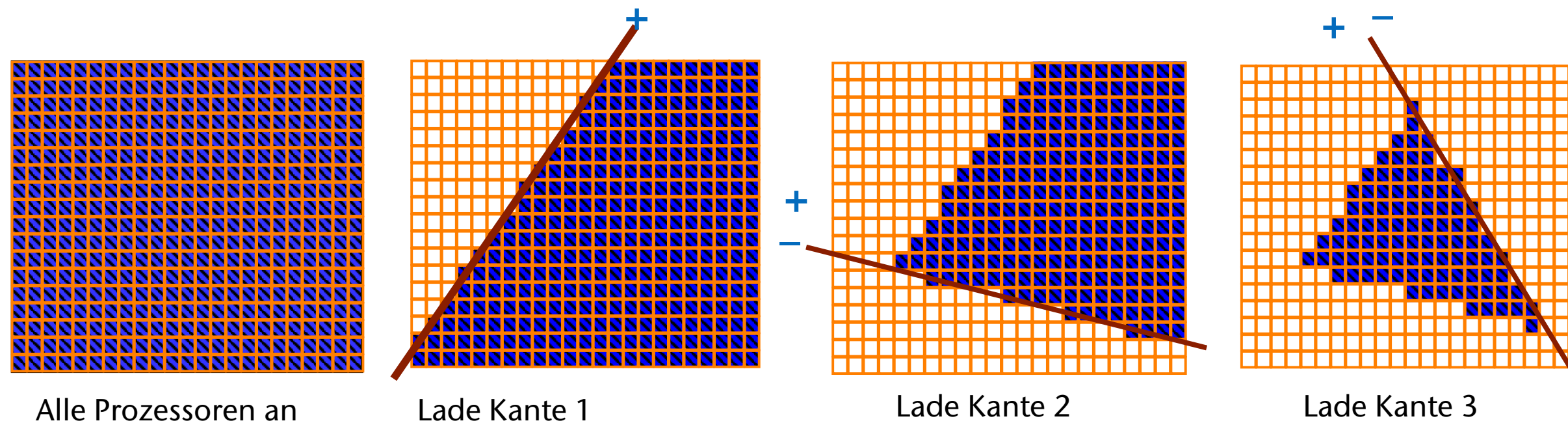
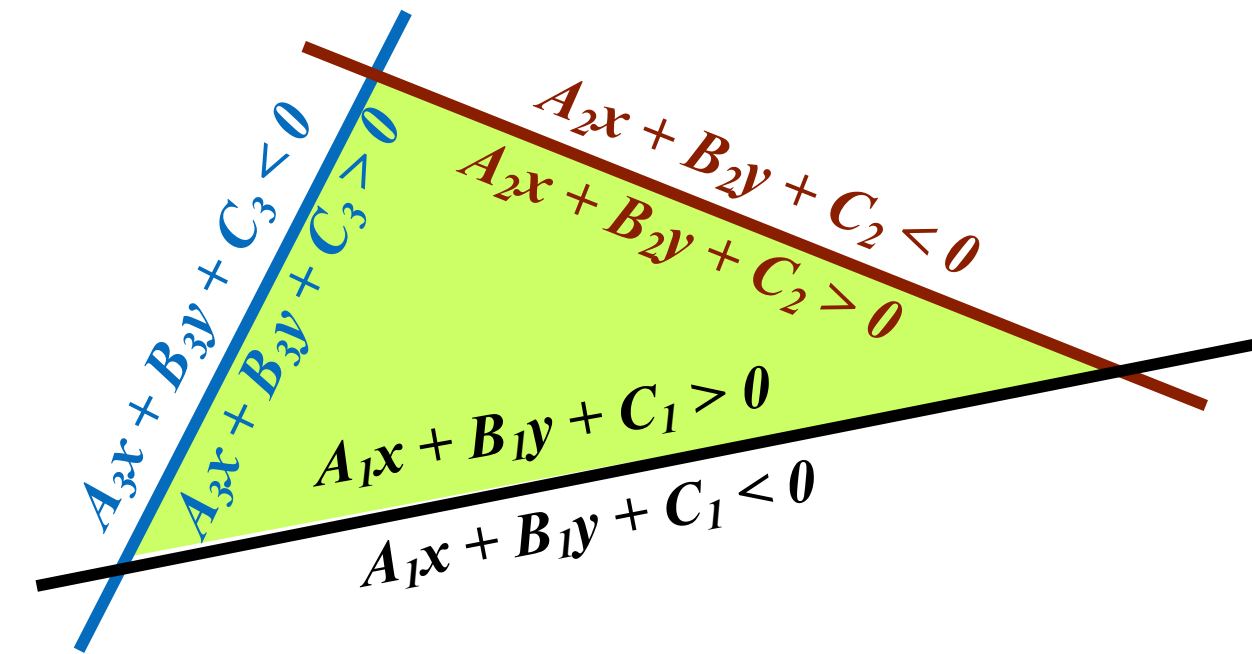
$$(X - B) \cdot \mathbf{n}_{AB} > 0$$



$$(X - A) \cdot \mathbf{n}_{BC} > 0$$

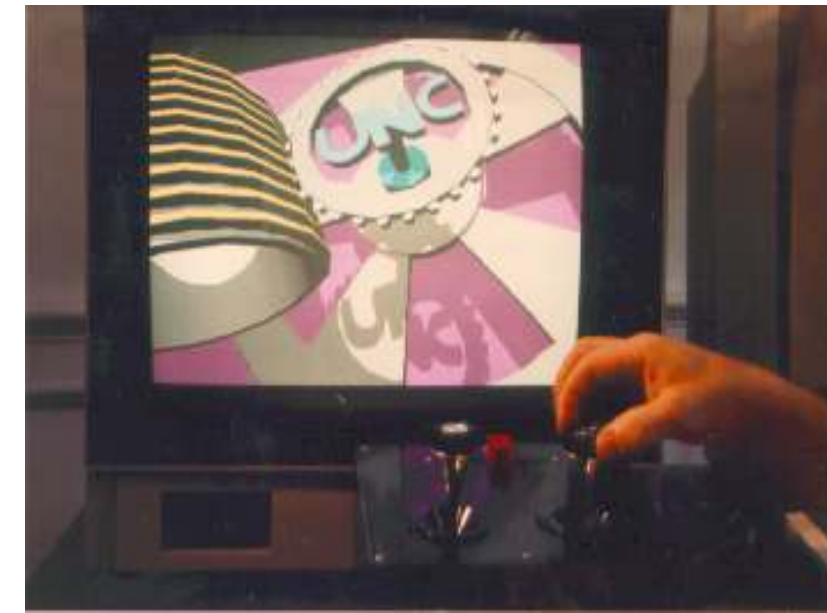
Kurzer Exkurs: Die Pixel-Planes-Architektur

- Eine Geschichte mit "*mixed end*" ...
- Die Idee:
 - Ein *processor per pixel* → jeder Pixel-Prozessor wertet für "seine" Koordinaten eine Gleichung der Form $Ax + By + C$ aus
 - Betrachte die 3 Kanten der Reihe nach
 - Lade alle Prozessoren *gleichzeitig* mit den Koeff. A,B,C der aktuellen Kante



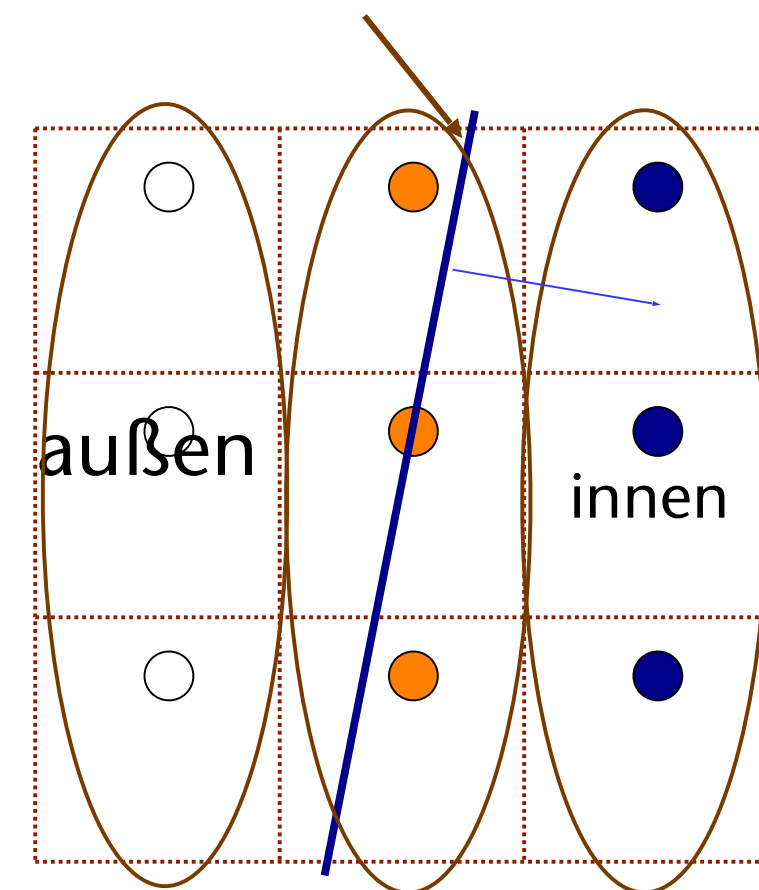
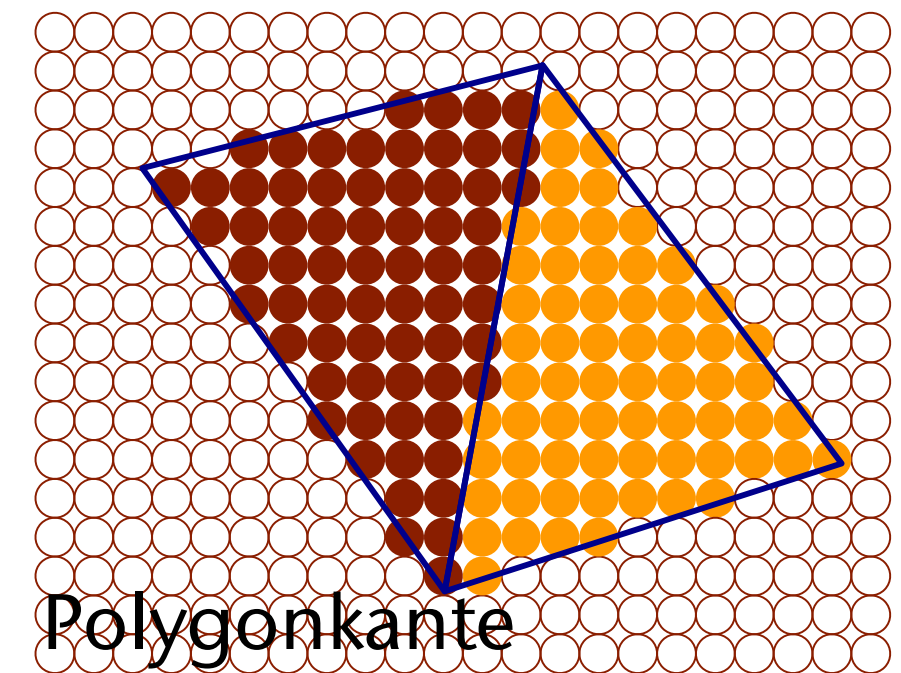
- Features:
 - Full-size (512 x 512 pixel) prototype
 - Used 2048 enhanced memory ICs
 - 1 Geometry Processor
 - 72 bits per pixel
- Performance:
 - 35K triangles/sec
 - Kugeln als Primitive
 - CSG
 - Schatten
- "Lessons Learned":
 - Dreiecke sind klein, daher viele Proc idle
 - Für noch mehr Performance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

Pixel-Planes 4 (1986)

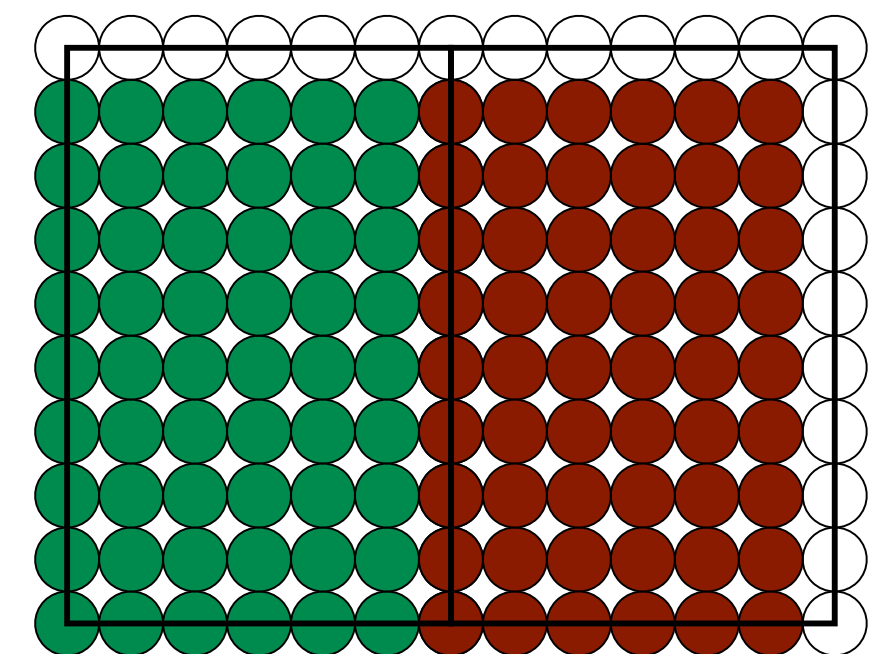
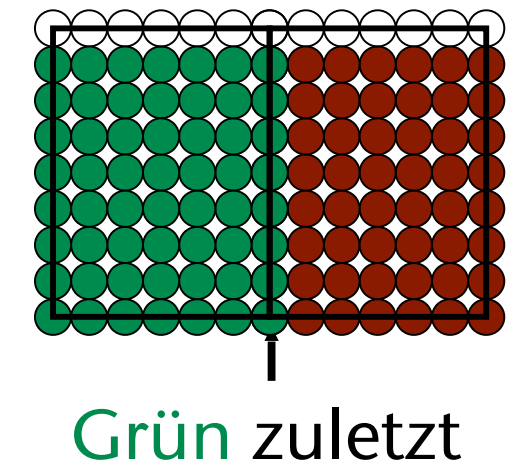
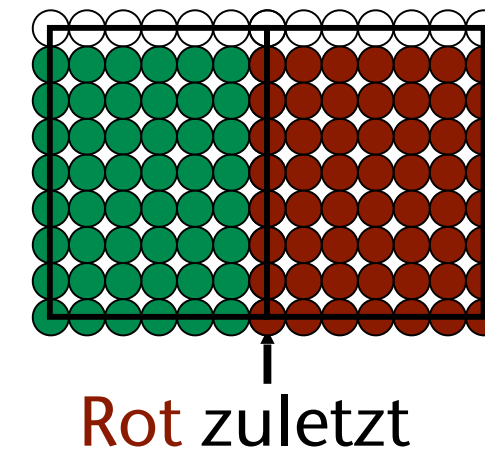


Konsistente Behandlung angrenzender Polygone

- Zu vermeiden: Risse, Überschneidungen, Abhängigkeit von der Zeichenreihenfolge
- Pixel zum Teil im Polygon → ... ?
- Lösung (ohne Anti-Aliasing): 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)

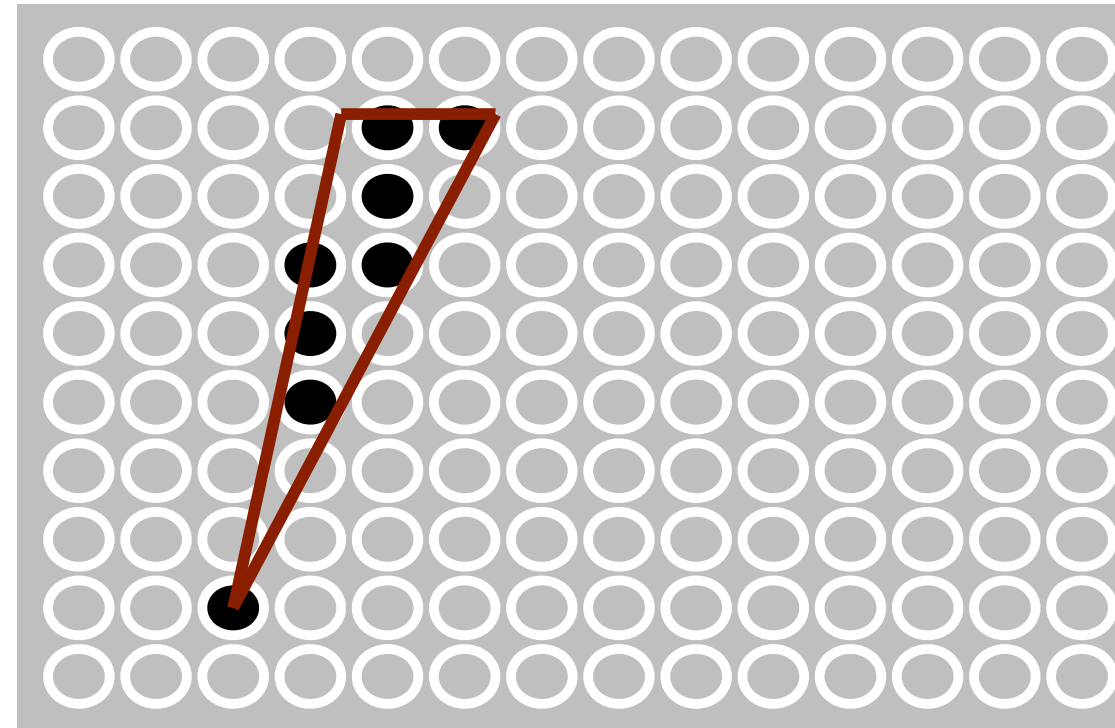


- Problem beim 2x Zeichnen:
 - Das zuletzt gezeichnete Polygon "gewinnt"
- Mögliche Lösung (gibt noch andere):
 - Ein Pixel (dessen Zentrum "genau" auf der Kante liegt) gehört **nicht** zu einem Primitiv, wenn das Primitiv links bzw. unterhalb des durch die Kante aufgespannten Halbraumes liegt (erkennt man an der Normale)
 - Verfahren 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)

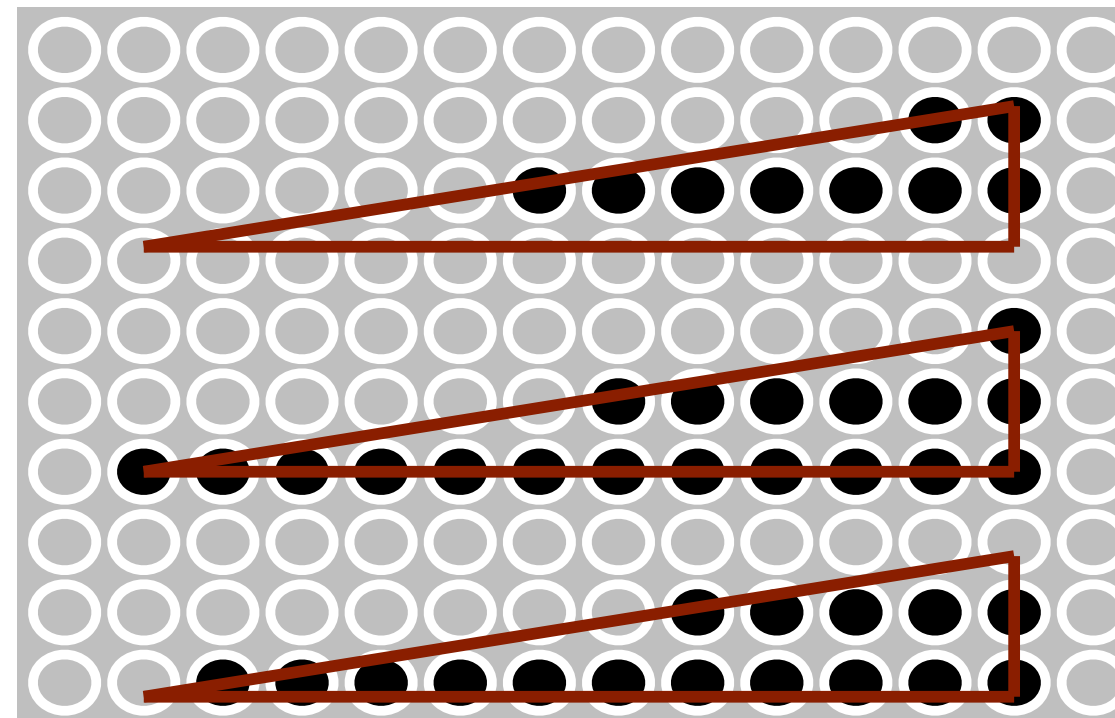


Weiteres Problem

- Sogenannte "*Slivers*":



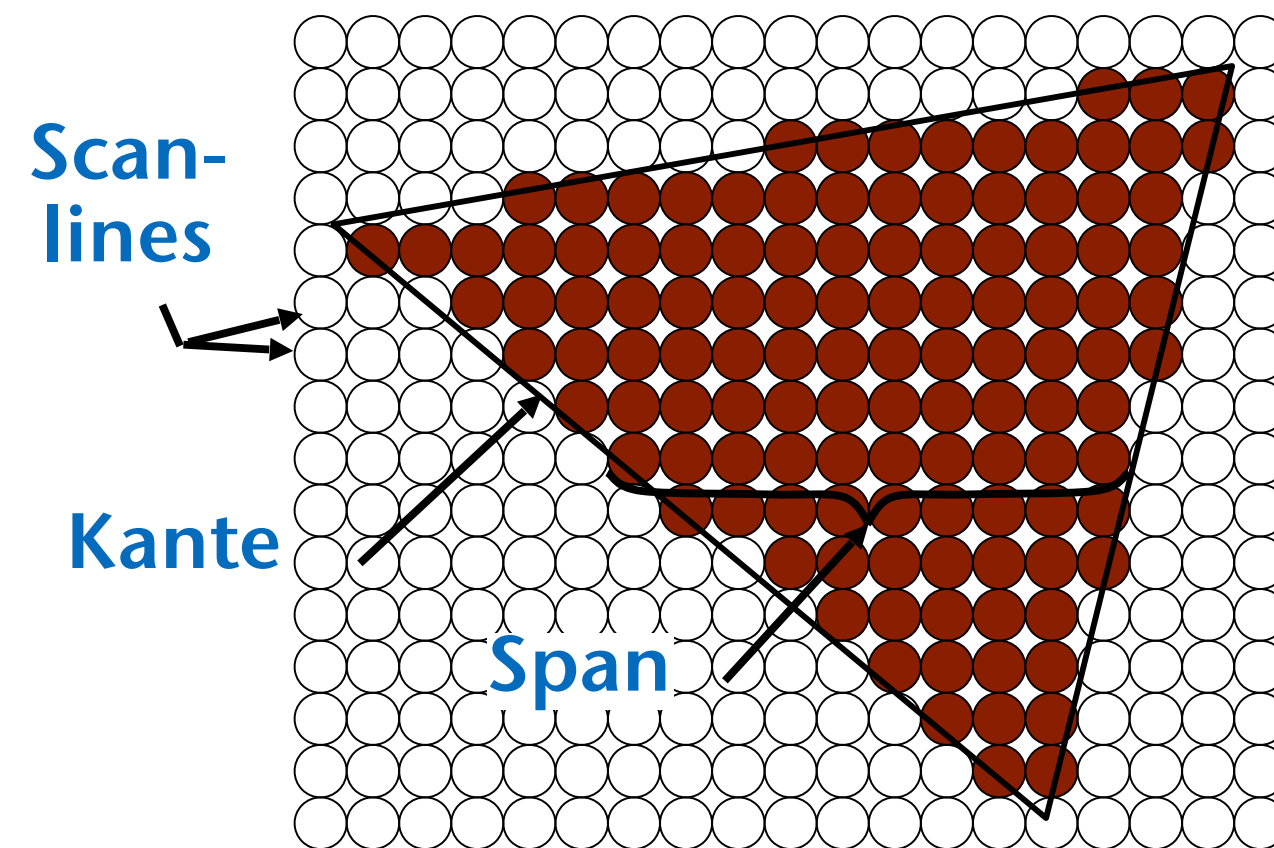
- *Moving Slivers*:



Scan Conversion für allgemeine Polygone

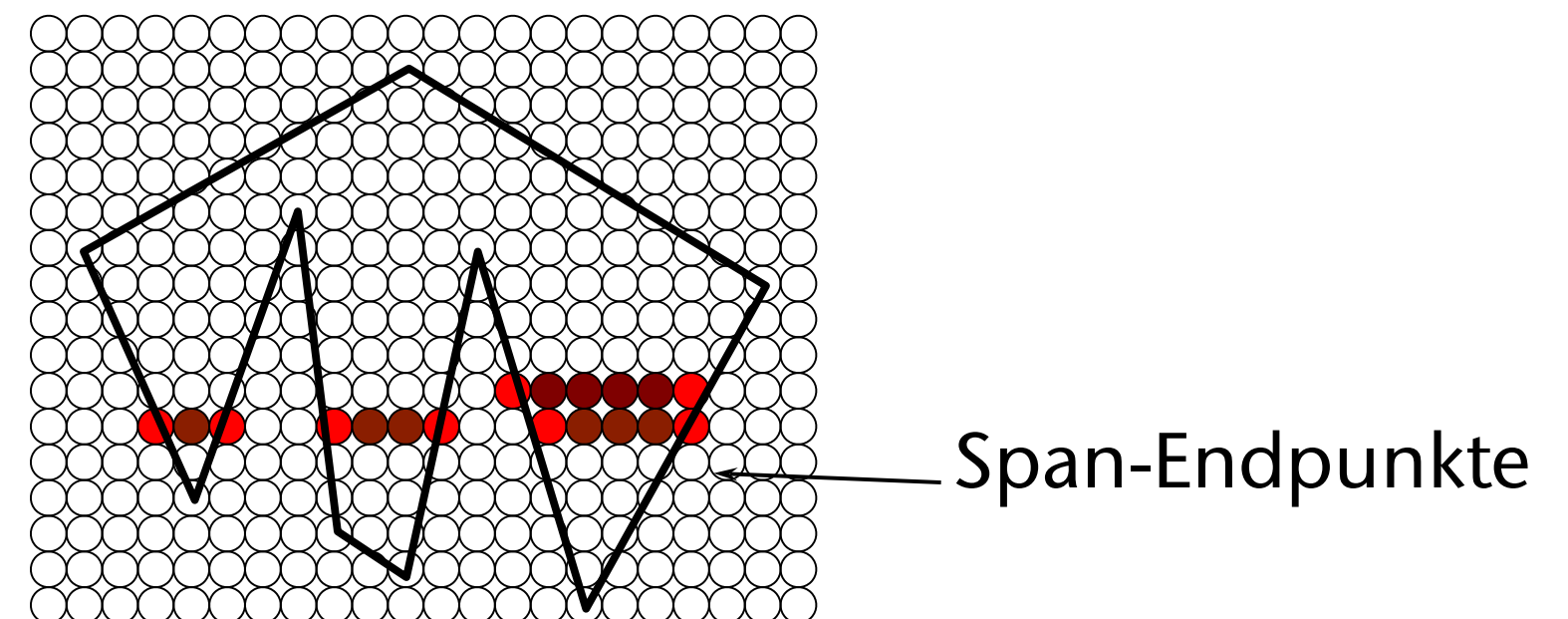
Optional - nicht prüfungsrelevant

- 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**

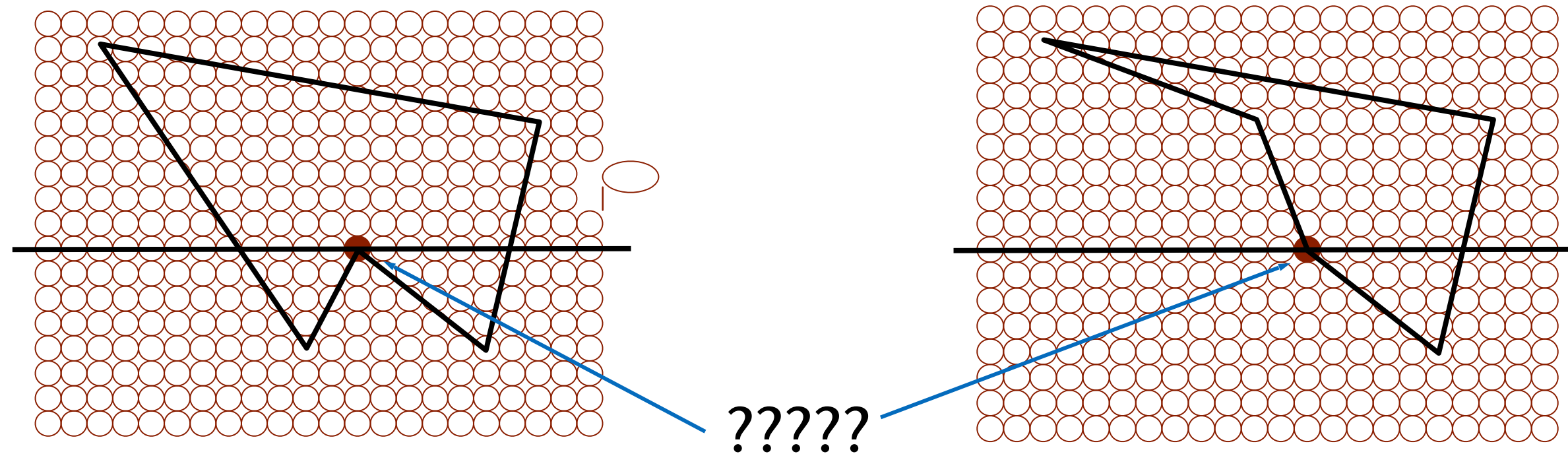


Überblick über den Algorithmus

- 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
- Der wesentliche Trick: berechne die Span-Endpunkte wieder mittels DDA bei Schritt um eine Scanline nach oben

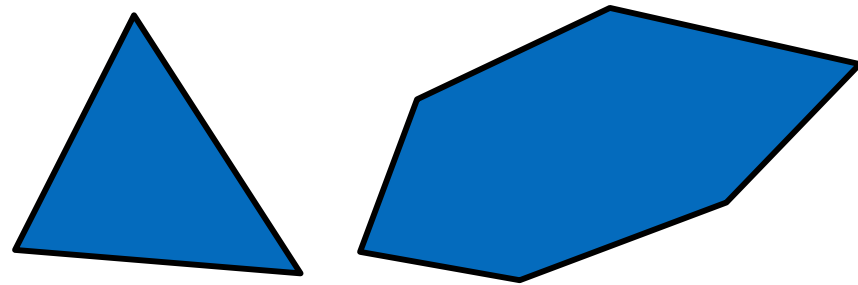


- Sonderfall: wie werten wir Eckpunkte genau auf der Scanline?
 - M.a.W.: begrenzt solch ein Eckpunkt einen Span?

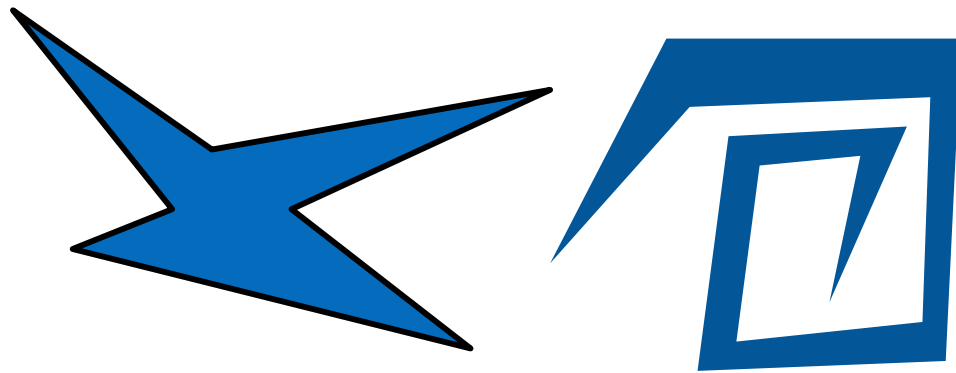


- Lösung: zähle einen Eckpunkt genau dann, wenn er das untere Ende der einen, und das obere Ende der anderen Kante ist
- Alternative: "wackle" am Eckpunkt ein wenig ("perturbation")
 - Dann bekommt man im linken Beispiel 2 oder 0 Schnittpunkte, im rechten genau 1 Schnittpunkt

Klassifikation der Polygone



Konvex: für jedes Punktepaar in einem konvexen Polygon liegt die Verbindung auch innerhalb des Polygons



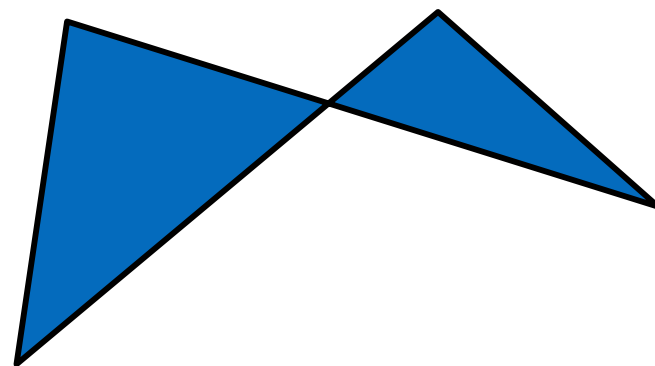
Nicht-Konvex ("konkav"), aber einfach

Definition:

ein Polygon ist **einfach** $\Leftrightarrow$ Randkurve hat keinen Schnittpunkt

Eigenschaften einfacher Polygone:

- "Topologisch äquivalent" zu einer 2-dimensionalen Scheibe
- Folge: es gibt ein wohldefiniertes Inneres/Äußeres



Nicht-einfach

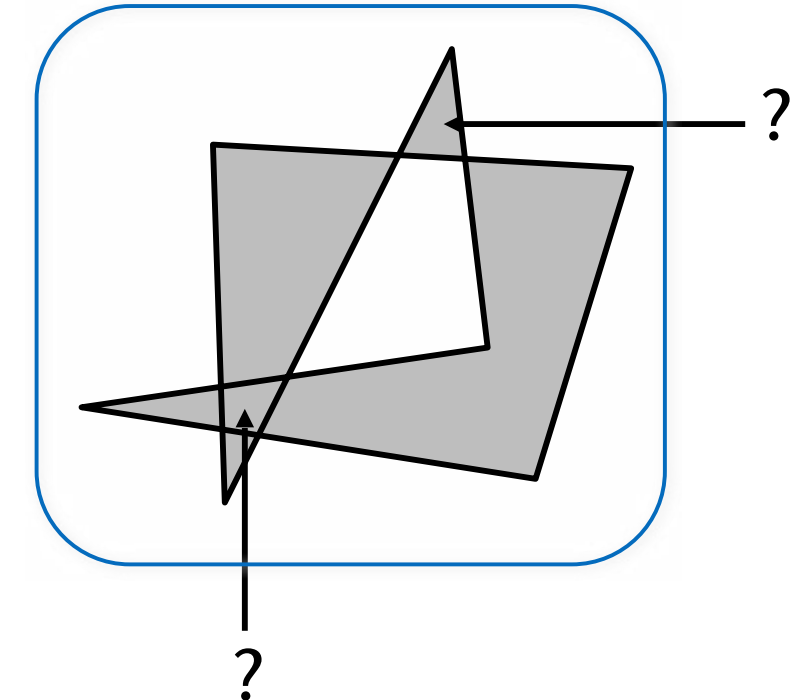
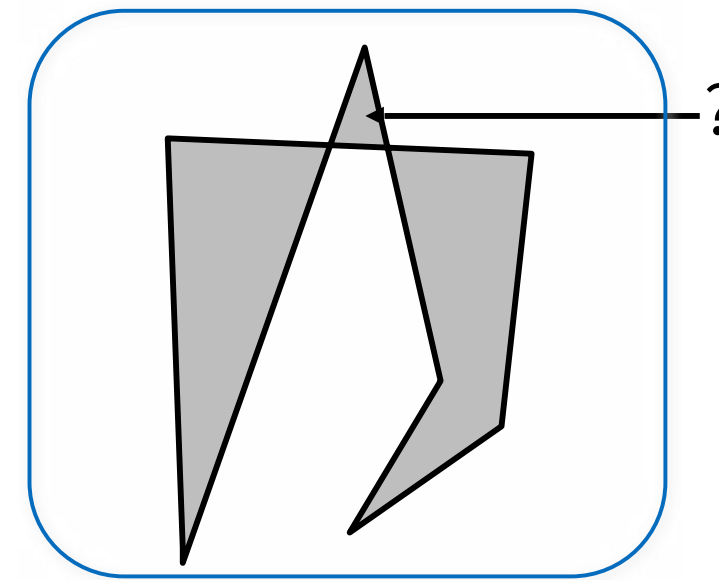
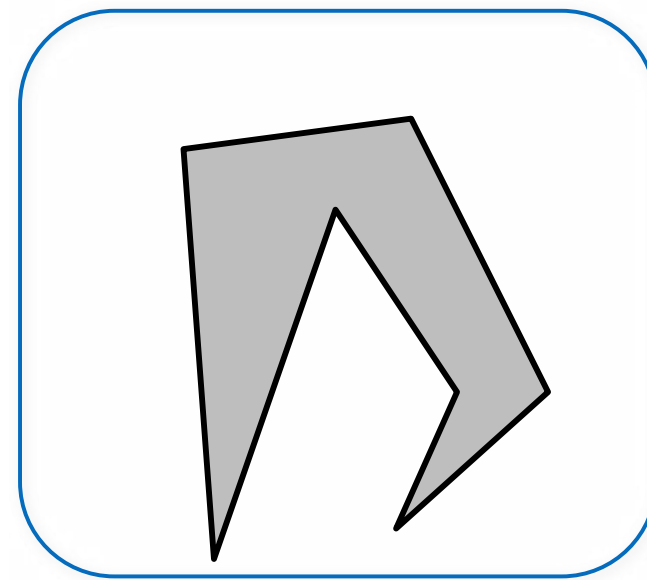
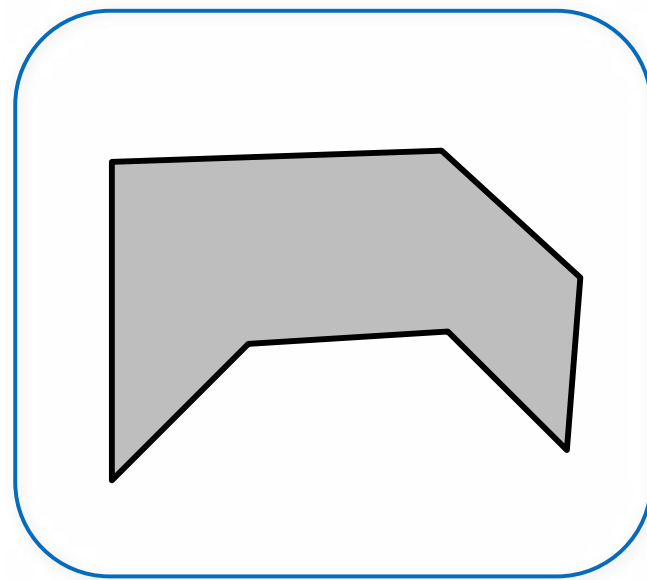
Umrandet die weiße Linie genau ein einfaches Polygon?



Robert Bosch

Direktes Füllen nicht-einfacher Polygone

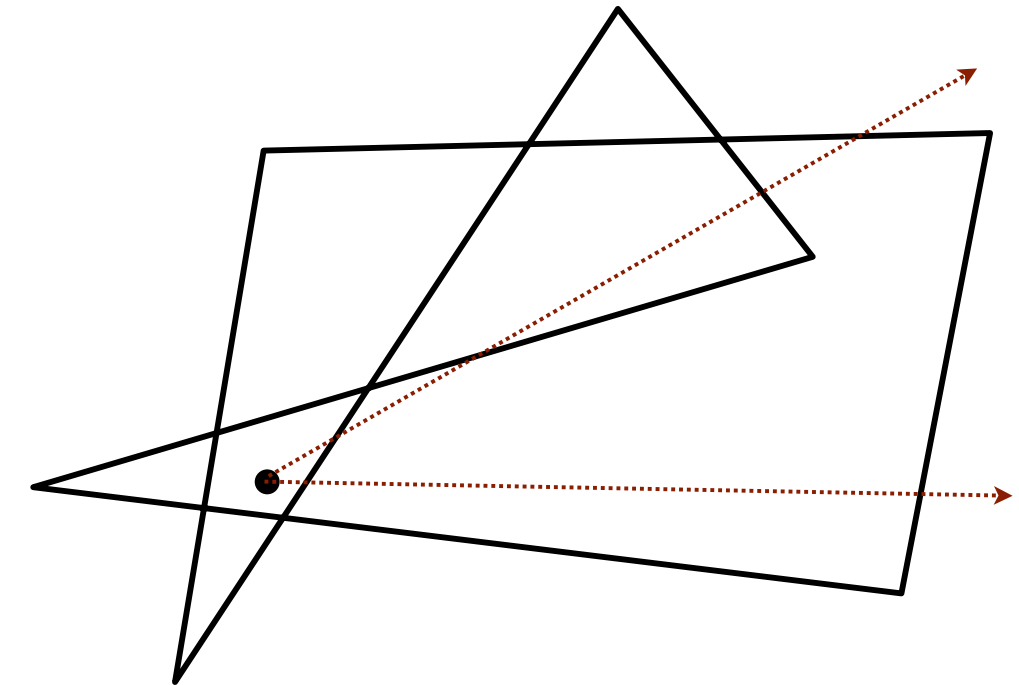
- Wesentliche Frage: wie definiert man "innen" und "außen"?



- Gesucht: ein Test für das Prädikat
"Punkt X ist im Inneren des Polygons P"

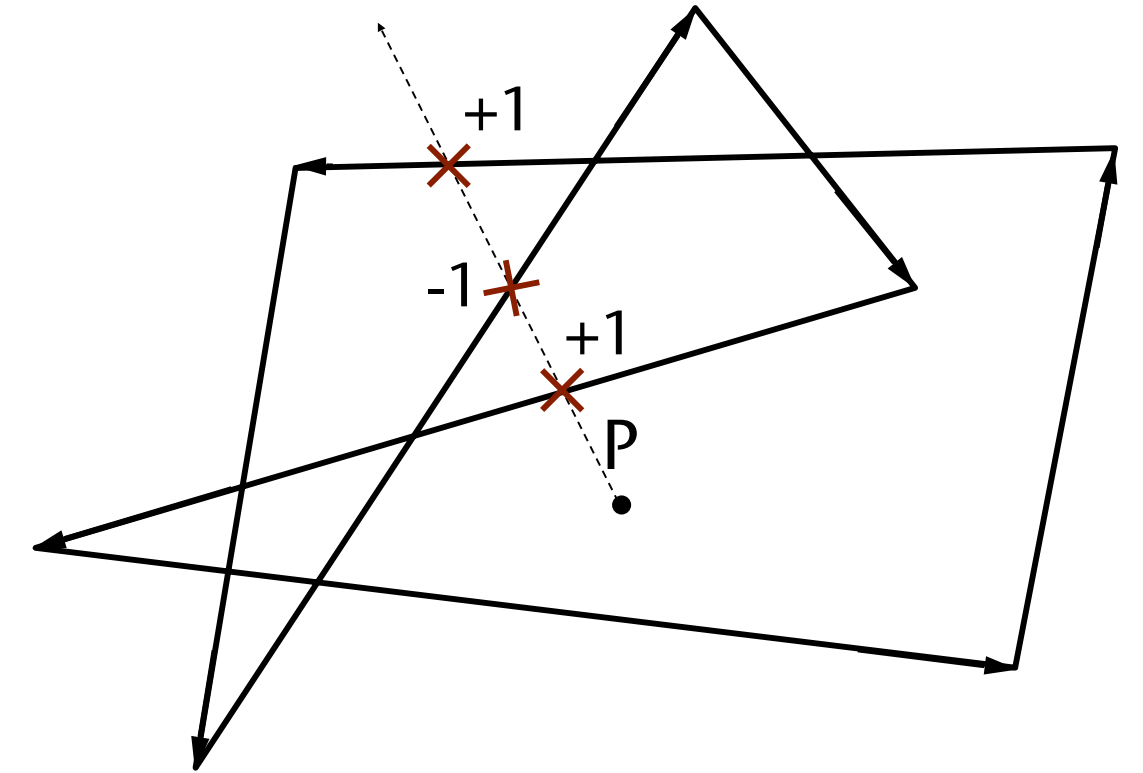
Test/Prädikat 1: Odd-Even Rule

- Zeichne Strahl von Punkt P nach unendlich in irgend eine Richtung
- Zähle Anzahl Schnittpunkte mit dem Polygon
- 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!

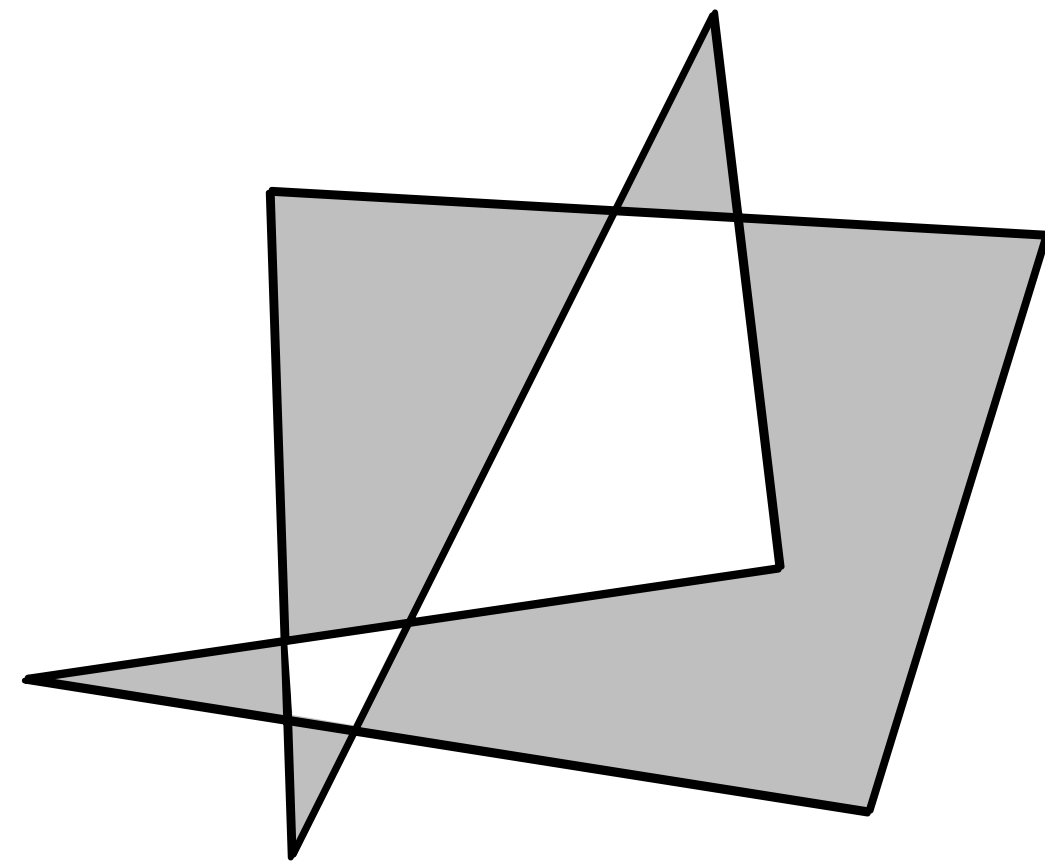


Test/Prädikat 2: Winding-Number Rule

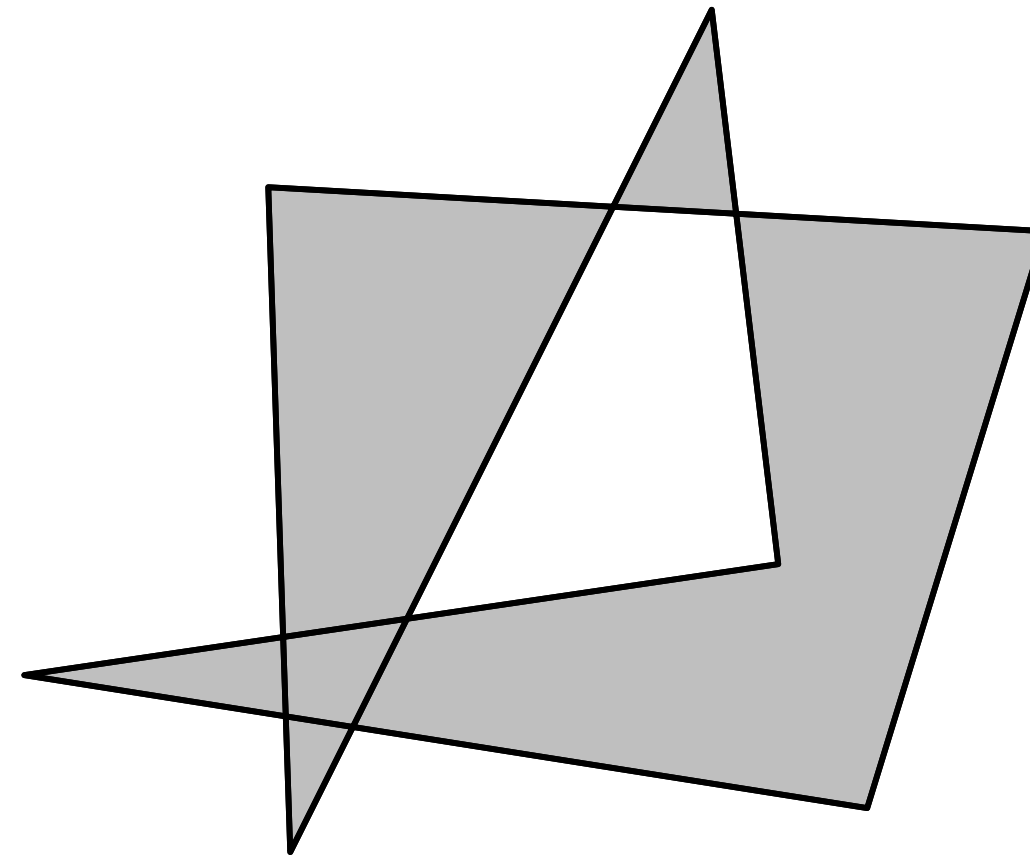
- Versehe Polygon mit konsistentem Umlaufsinn
- Schneide Strahl von P aus mit Kanten
- Setze Winding-Number $w := 0$
- Für Schnitt mit Kante „von rechts nach links“ erhöhe w ; sonst erniedrige w
- Falls $w \neq 0 \rightarrow P$ innerhalb
- Anmerkung: damit definiert man gleichzeitig "positiv" bzw. "negativ"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



Vergleich



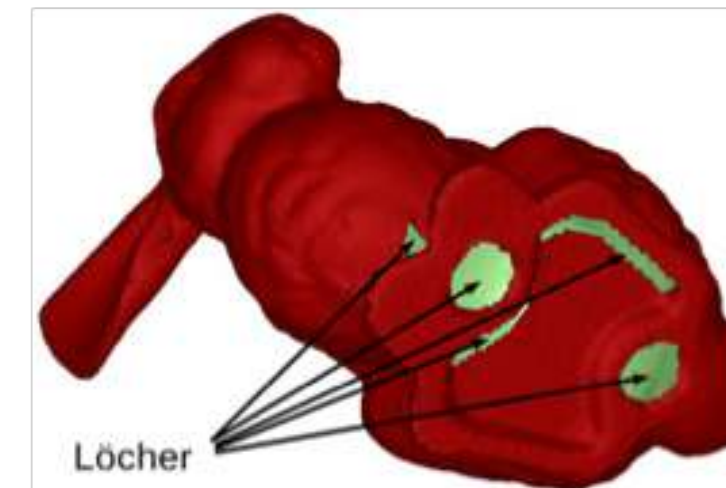
Odd-even Rule



Non-zero Winding Number

Verallgemeinerung in 3D

- Der Odd-Even-Test funktioniert genauso bei 3D Polyedern
 - In 2D: Schnitt Strahl-Kante $\Rightarrow$ in 3D: Schnitt Strahl-Polygon
- Achtung: der Test funktioniert nur dann robust, wenn der Polyeder "wasserdicht" (*watertight*) ist
- Nicht wasserdicht:



Verallgemeinerungen der Winding Number

- Alternative Definition der Winding Number (in 2D):
 - Summe der Winkel (mit Vorzeichen!)

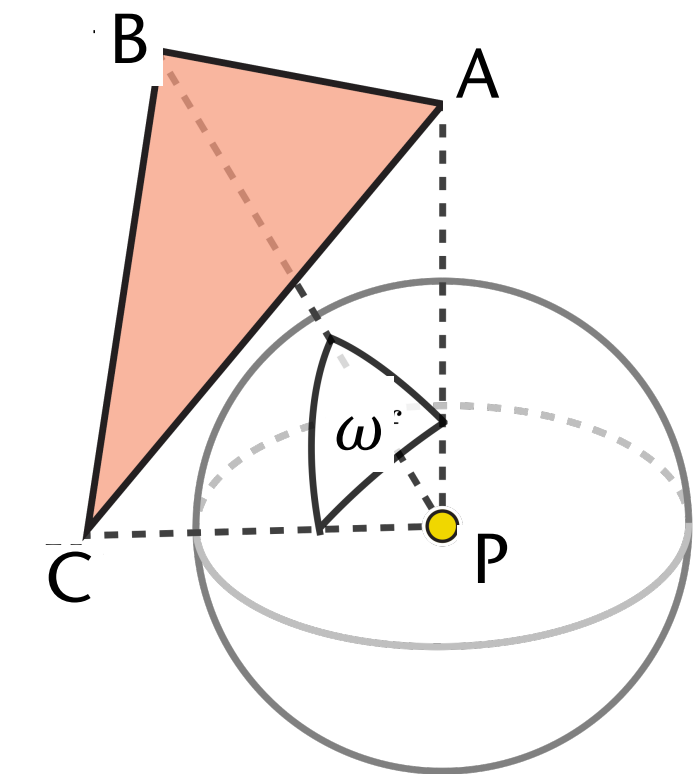
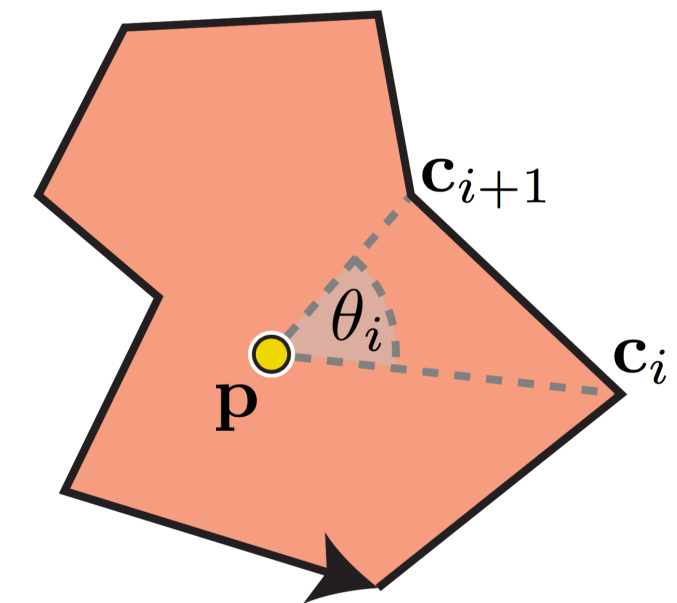
$$w(P) = \frac{1}{2\pi} \sum_i^n \theta_i$$

- Übertragung auf 3D:

$$w(P) = \frac{1}{4\pi} \sum_i^m \omega_i$$

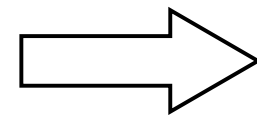
wobei

$$\tan\left(\frac{\omega_i}{2}\right) \sim \det(\mathbf{a}, \mathbf{b}, \mathbf{c}), \quad \mathbf{a} = A - P, \quad \mathbf{b} = B - P, \quad \mathbf{c} = C - P$$

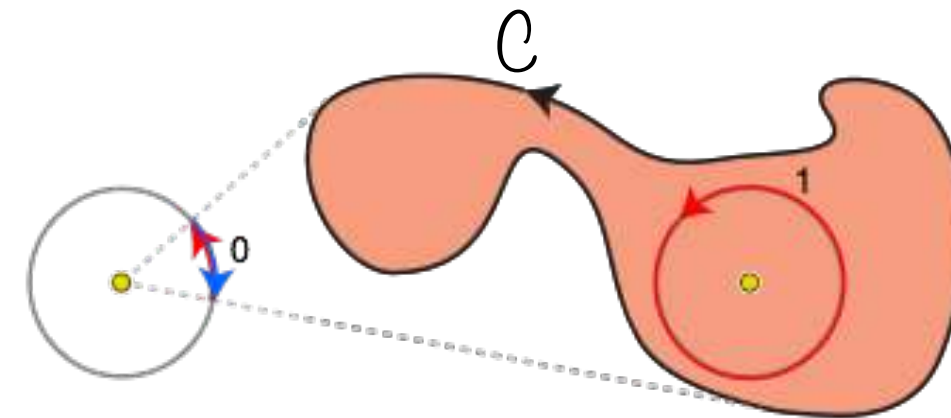
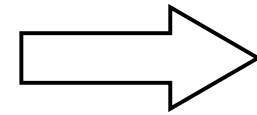
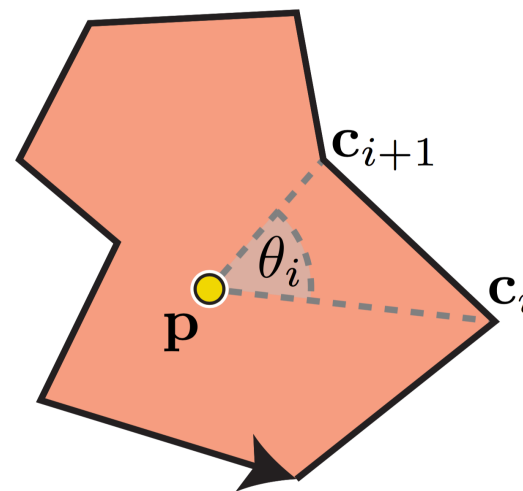


- Kontinuierliche Version in 2D:

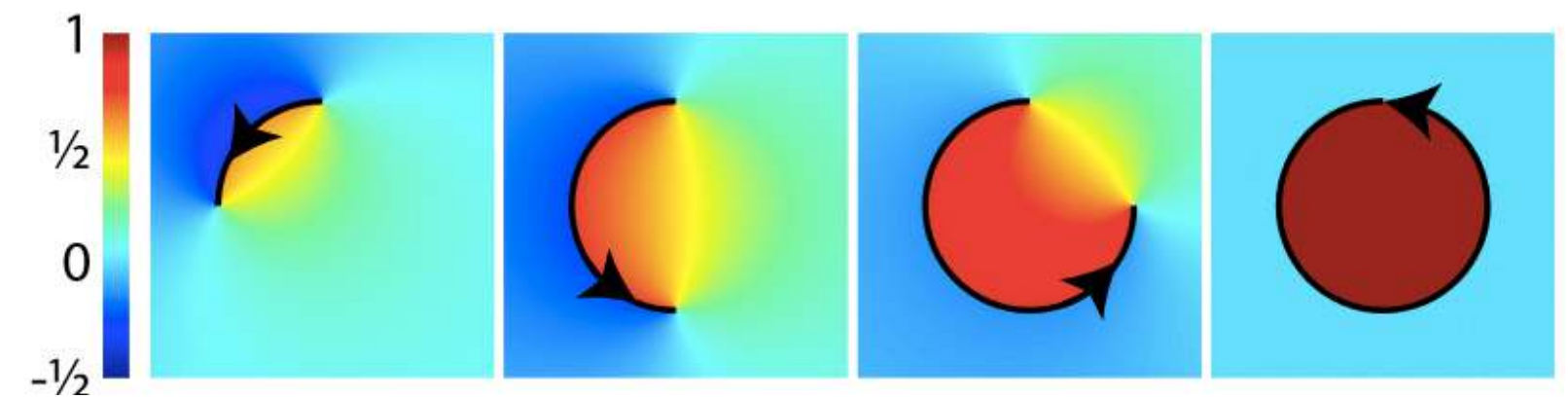
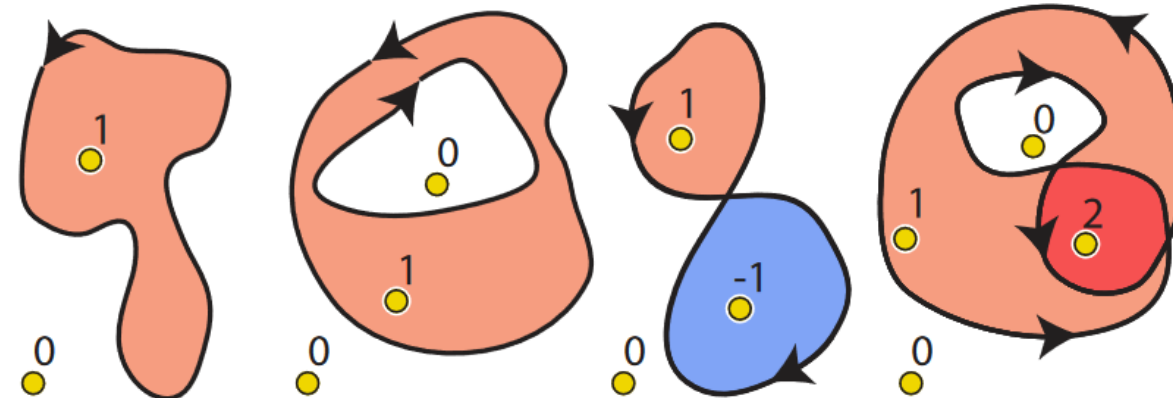
$$w(P) = \frac{1}{2\pi} \sum_i^n \theta_i$$



$$w(P) = \frac{1}{2\pi} \oint_C d\theta$$



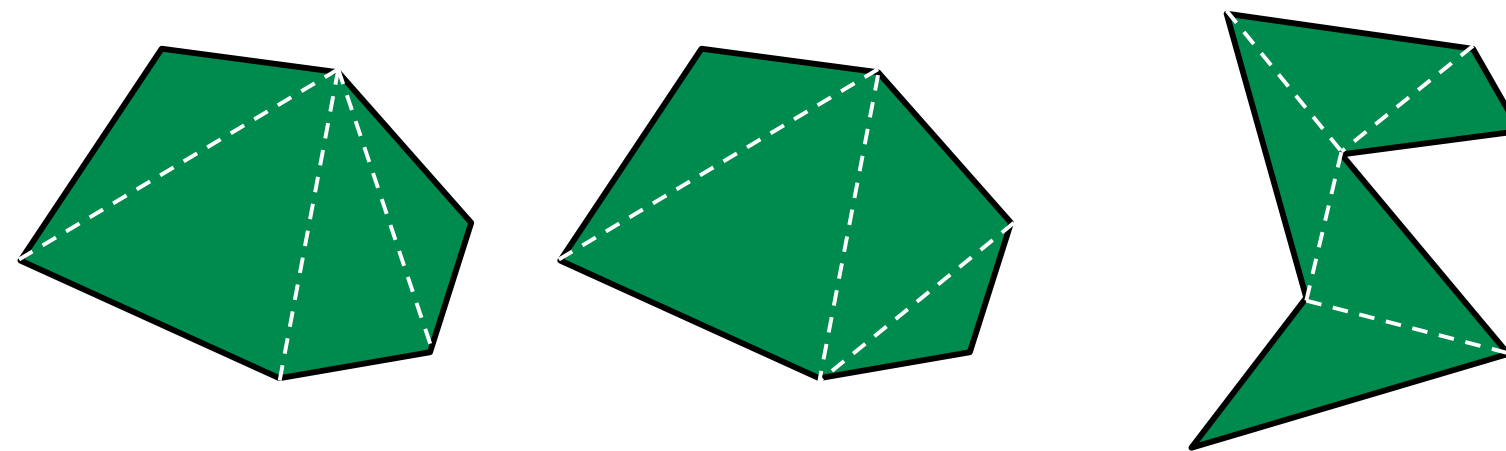
- Beispiele:



Triangulation

- Definition:

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



- Fragen:

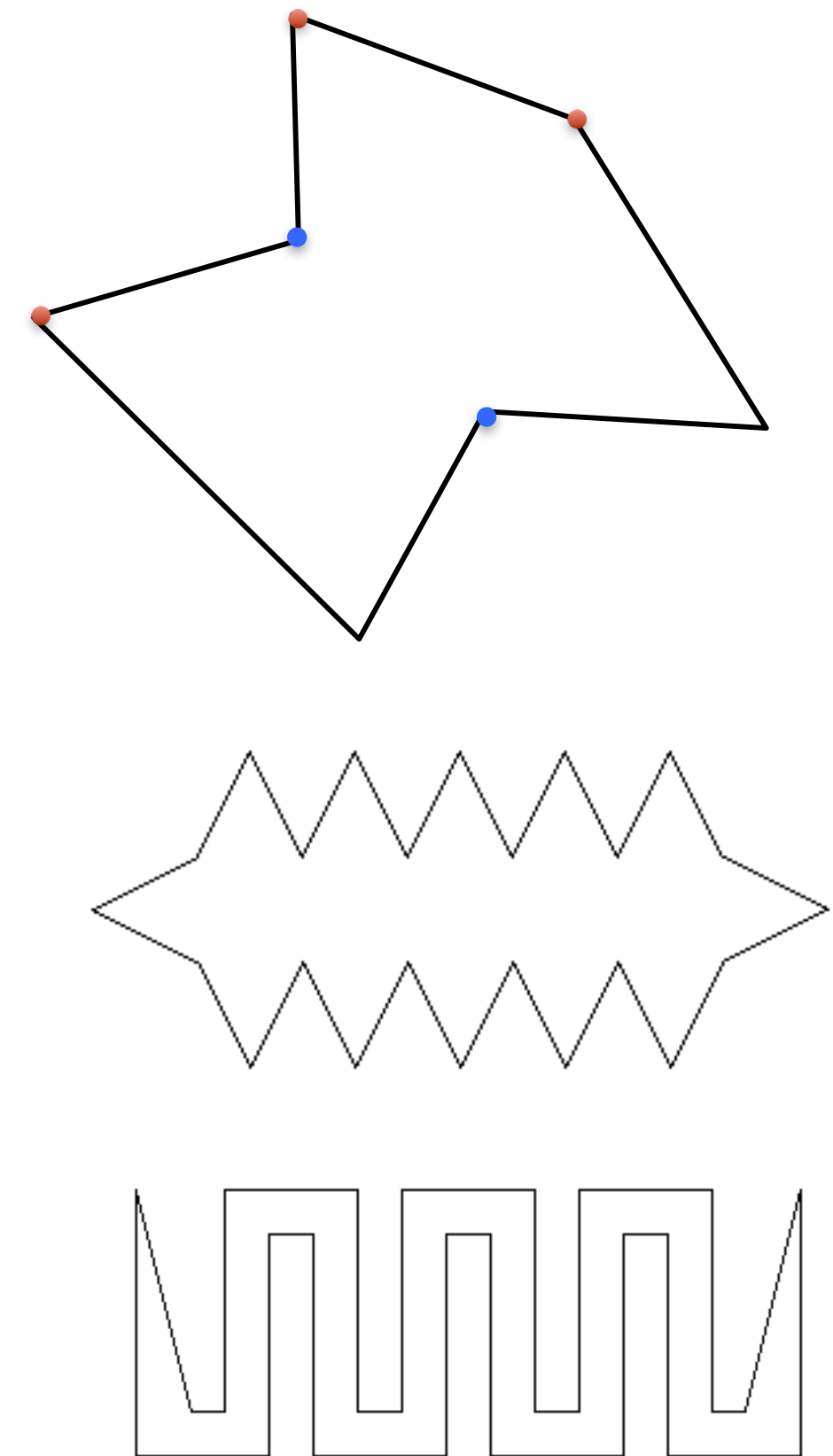
- Kann man jedes Polygon triangulieren?
- Hängt die Anzahl Dreiecke von der Wahl der Diagonalen ab?

- Satz:

Jedes beliebige Polygon kann trianguliert werden.

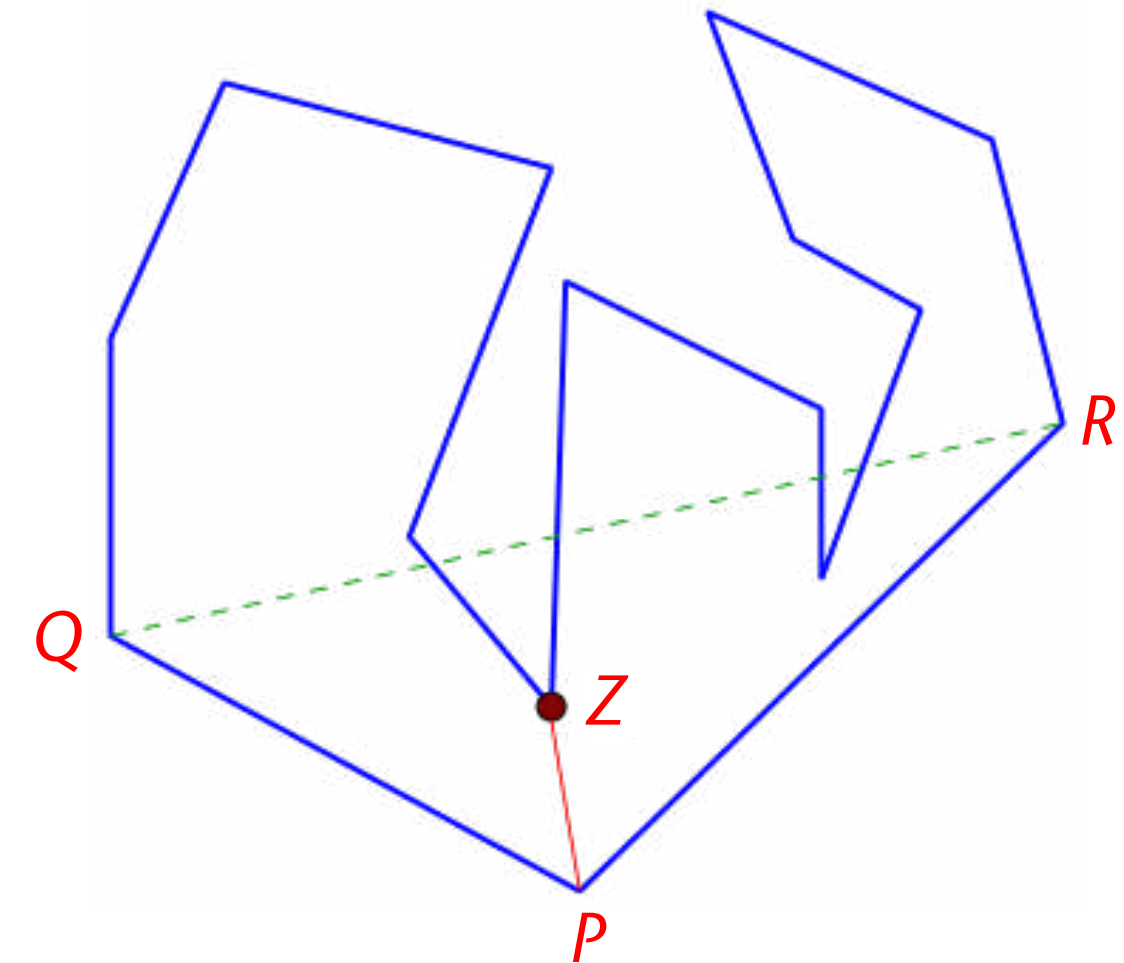
Begriffe

- **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 $V_{i-1}V_{i+1}$ ist komplett innerhalb des Polygons; in diesem Fall heißt $V_{i-1}V_{i+1}$ **Diagonale**.
- Satz (Zwei-Ohren-Satz):
Jedes Polygon hat (mindestens) 2 Ohren.

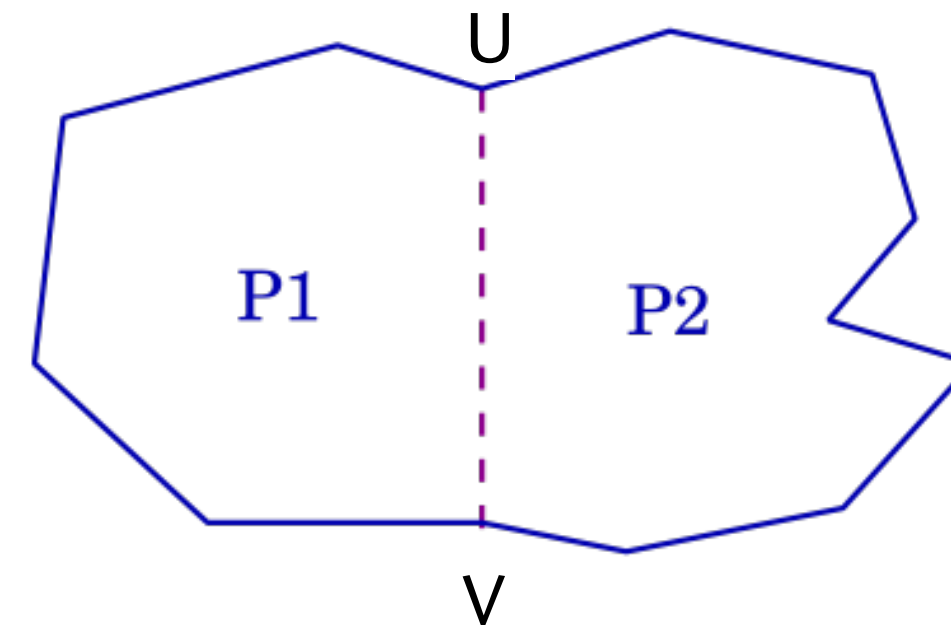


Beweis des Satzes "Jedes Polygon kann trianguliert werden"

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

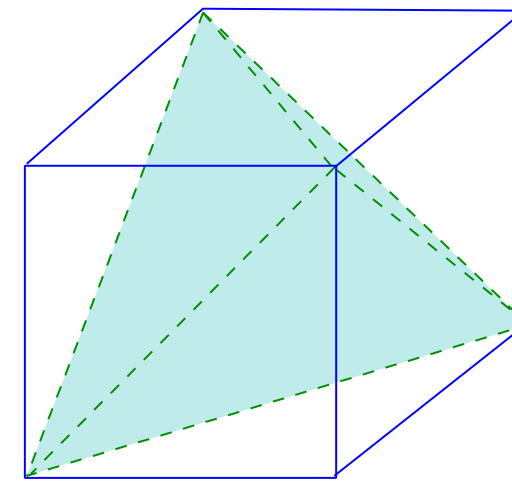


- Satz:
Jede Triangulierung eines n -gons hat $n-2$ Dreiecke.
- Bezeichne mit $t(P)$ = Anzahl Dreiecke in irgendeiner Triangulation des Polygons P
- Beweis: mittels Induktion
- Basisfall: einzelnes Dreieck, $t(P) = 1 = 3-2$
- Induktionsschritt:
 - Wähle eine Diagonale UV in der gegebenen Triangulation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

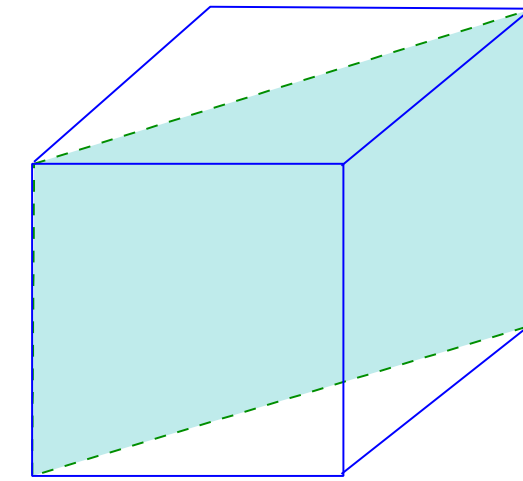


Zum Vergleich: Triangulation in 3D (= "Tetraedrisierung")

- Verschiedene Triangulierungen → verschiedene Anzahl Tetraeder

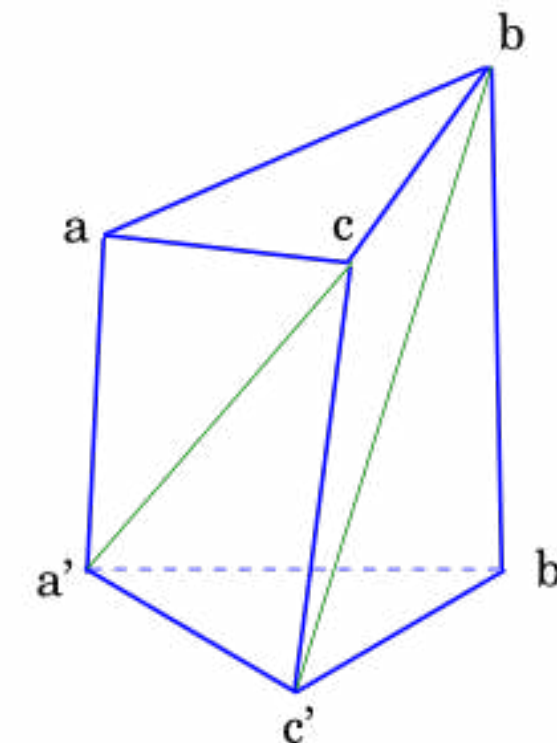
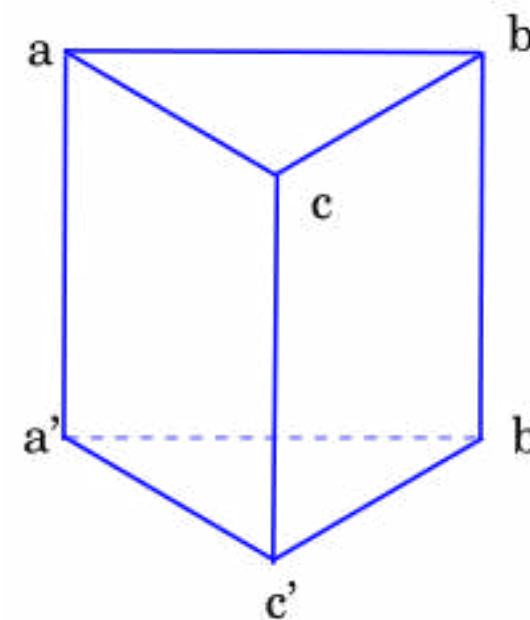


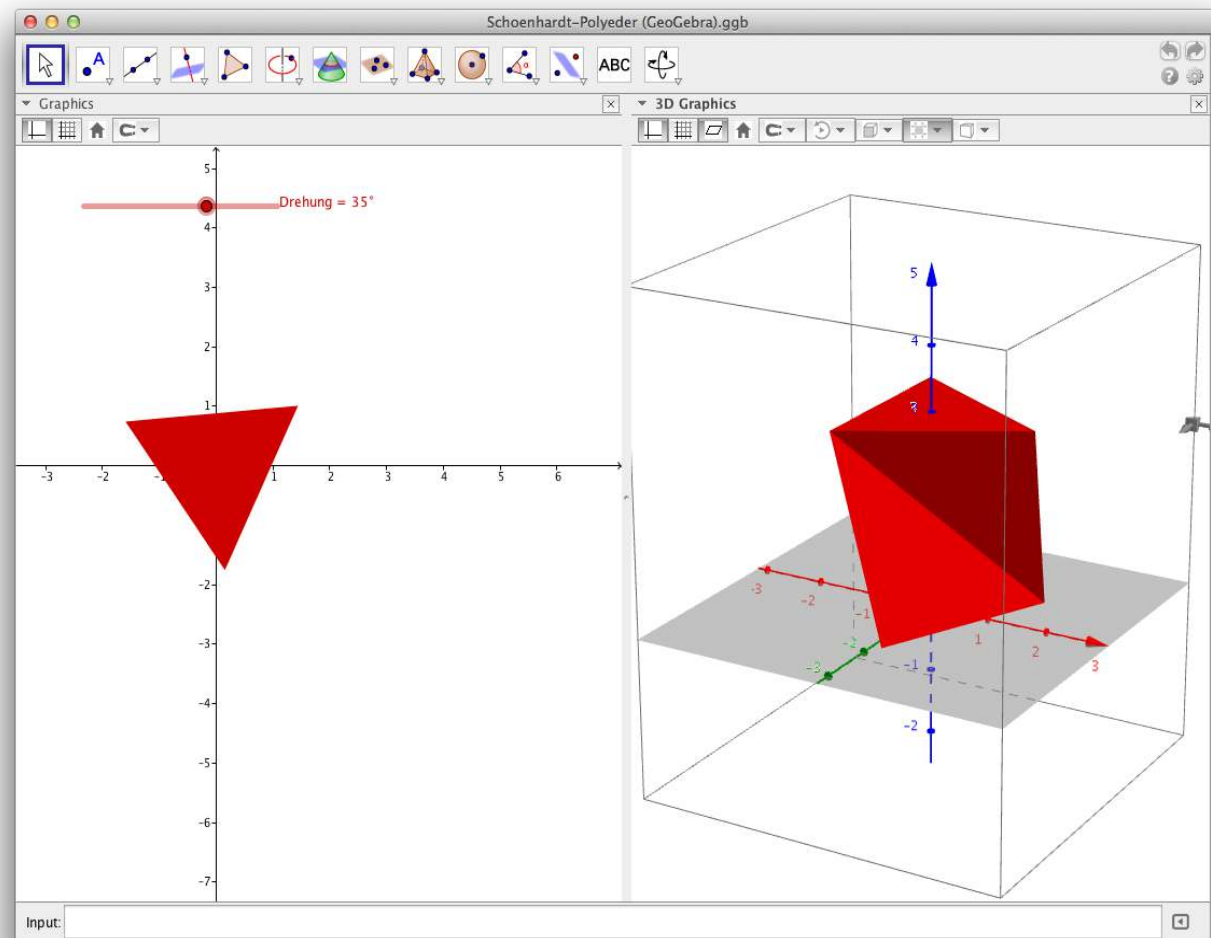
5 Tetraeder



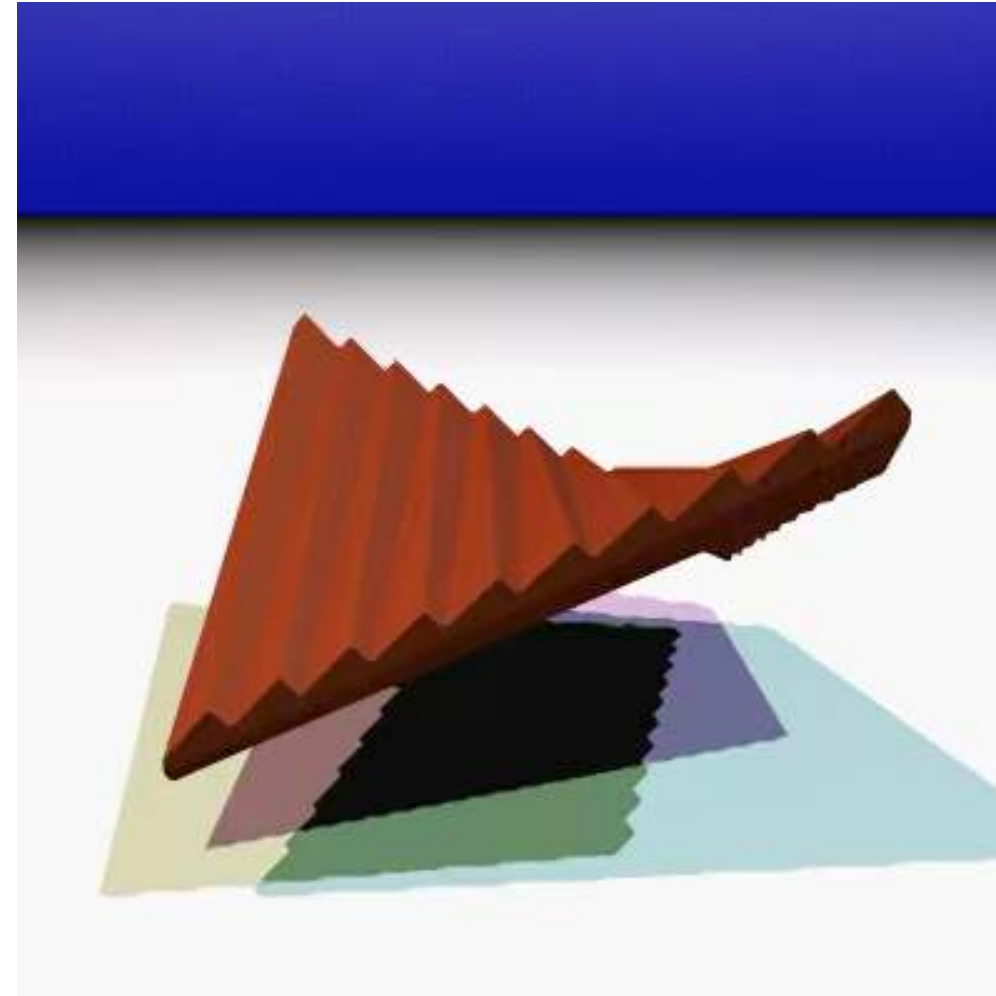
6 Tetraeder

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

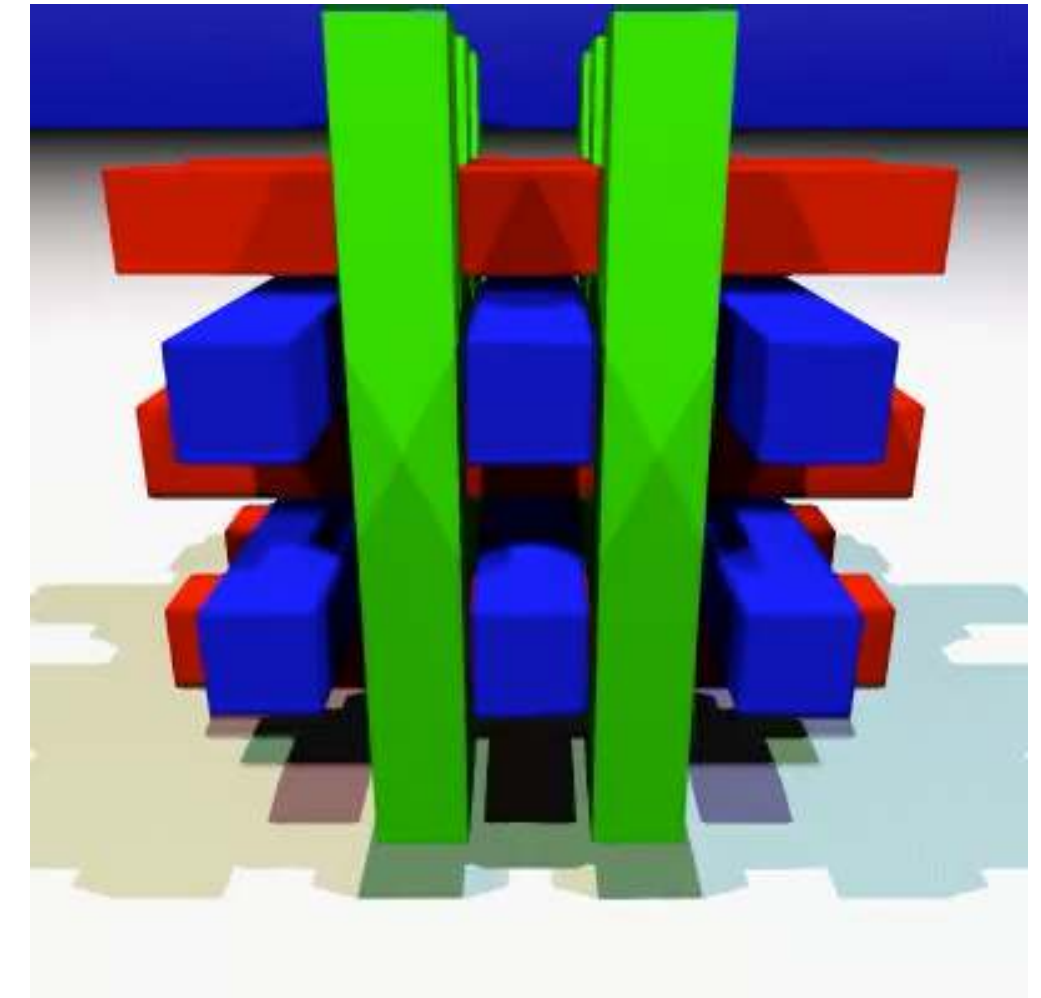




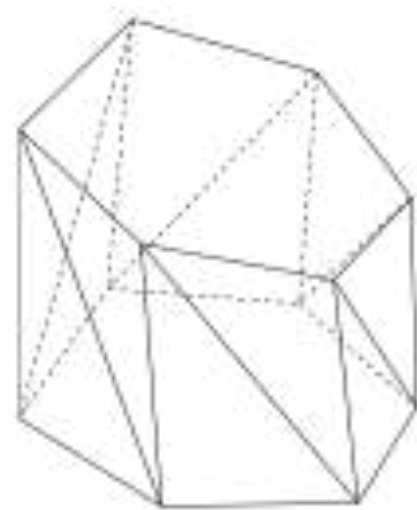
Schönhardt Polyhedron



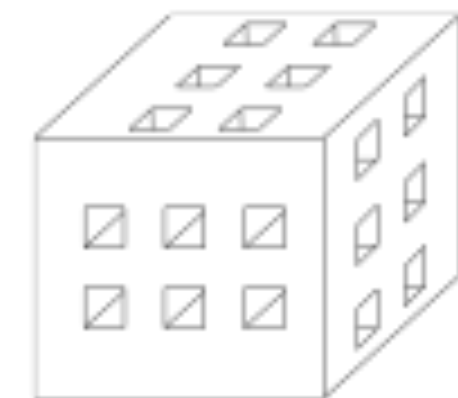
Chazelle Polyhedron



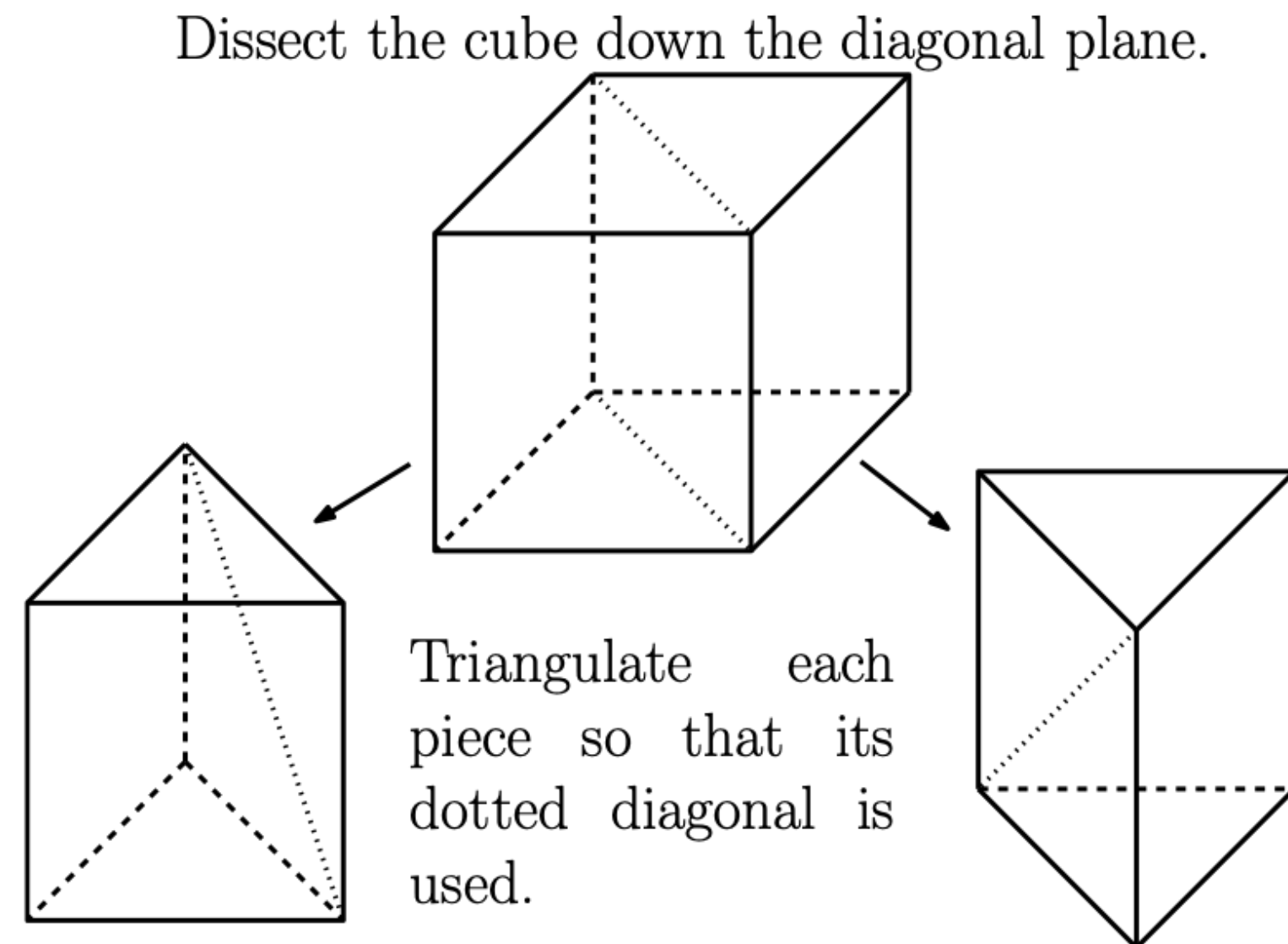
Thurston Polyhedron



Generalization of
Schönhardt by Rambau



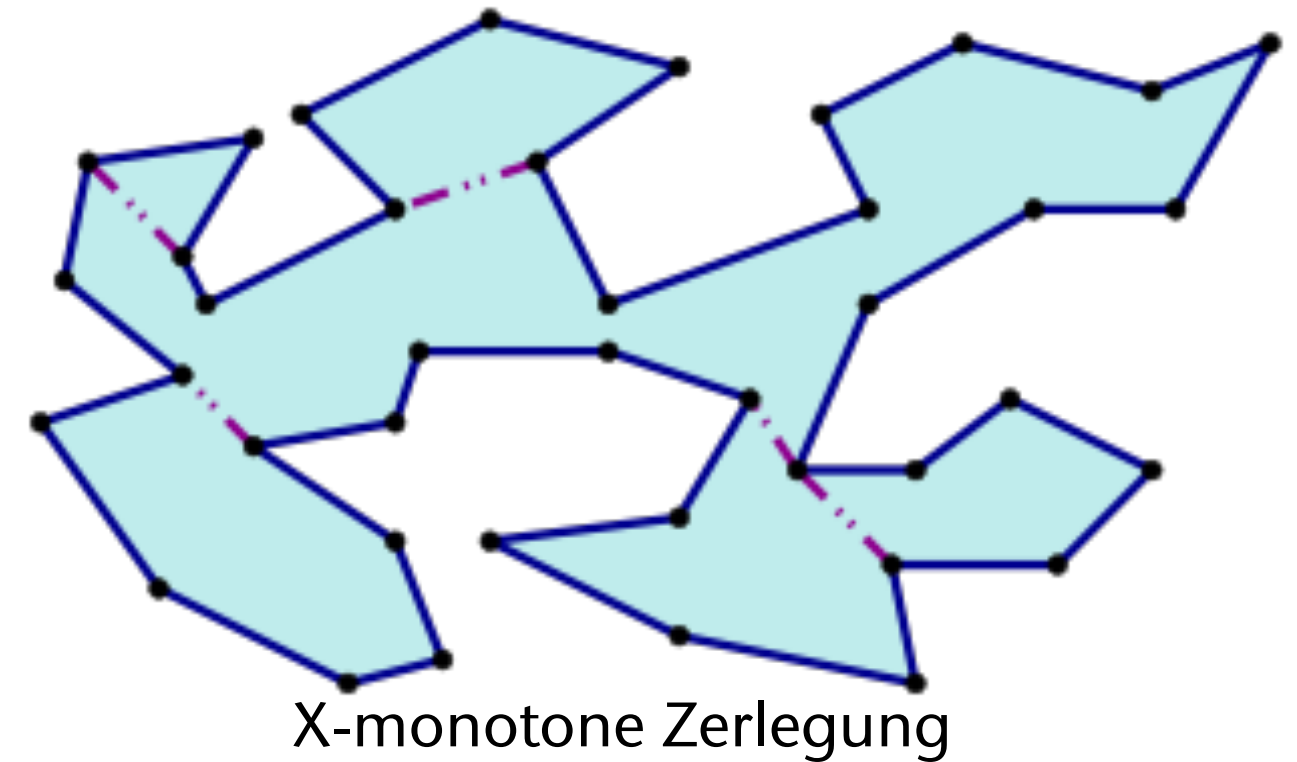
- In 3D, even more things can go wrong: there are tilings (partitionings) using only tetrahedra and no extra vertices, which are *still not a proper triangulation*
- Example:



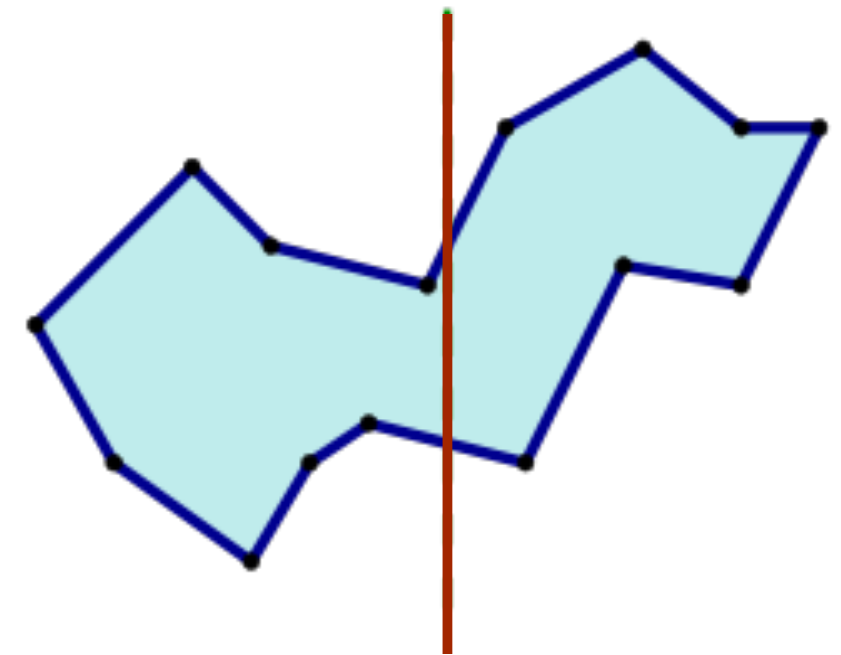
Triangulations-Algorithmen

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

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

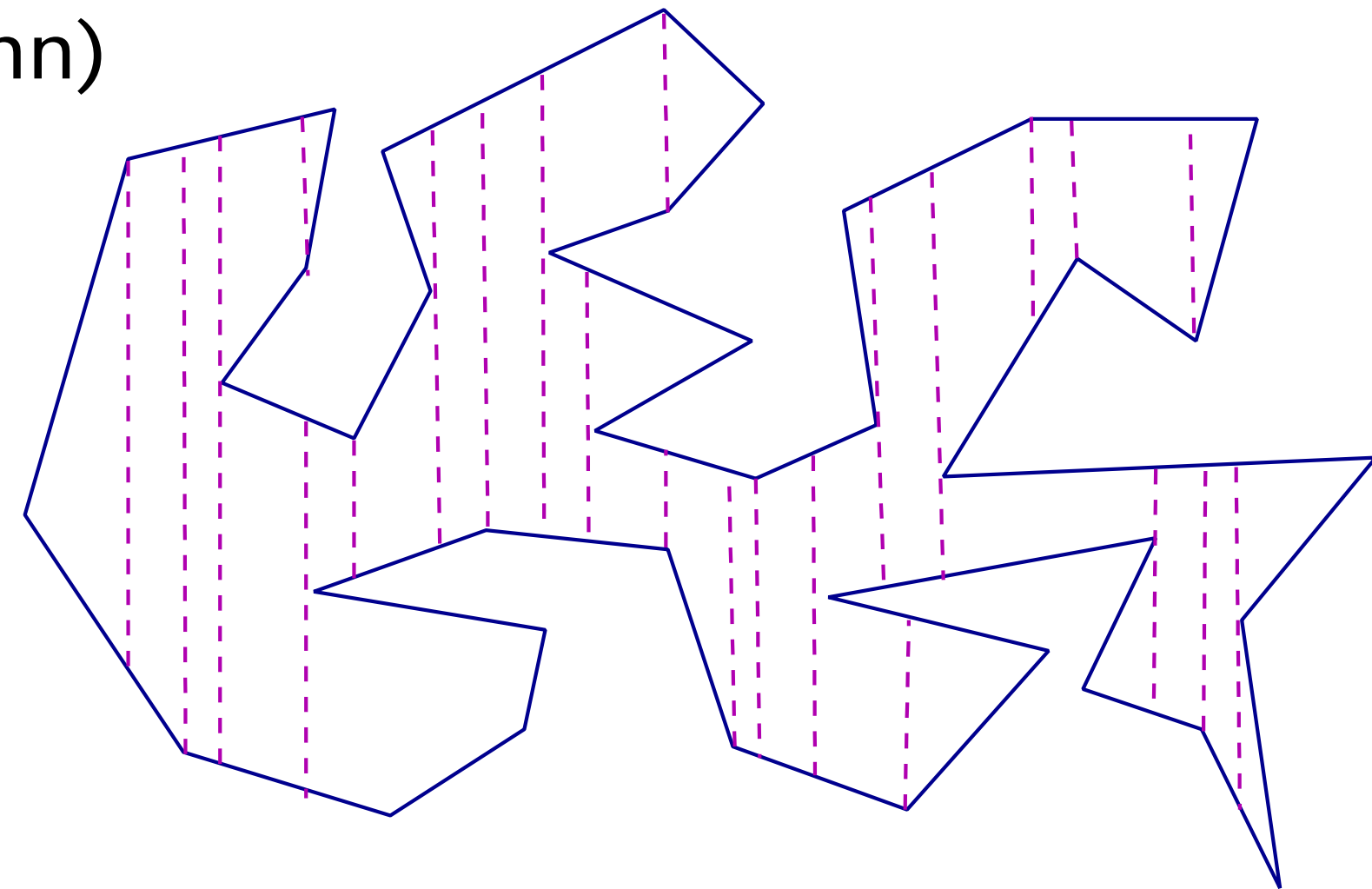


- Def. monotonen Polygon:
Ein Polygon heißt **monoton bzgl. einer Geraden L** , falls jede Gerade senkrecht zu L das Polygon in genau zwei Punkten schneidet.



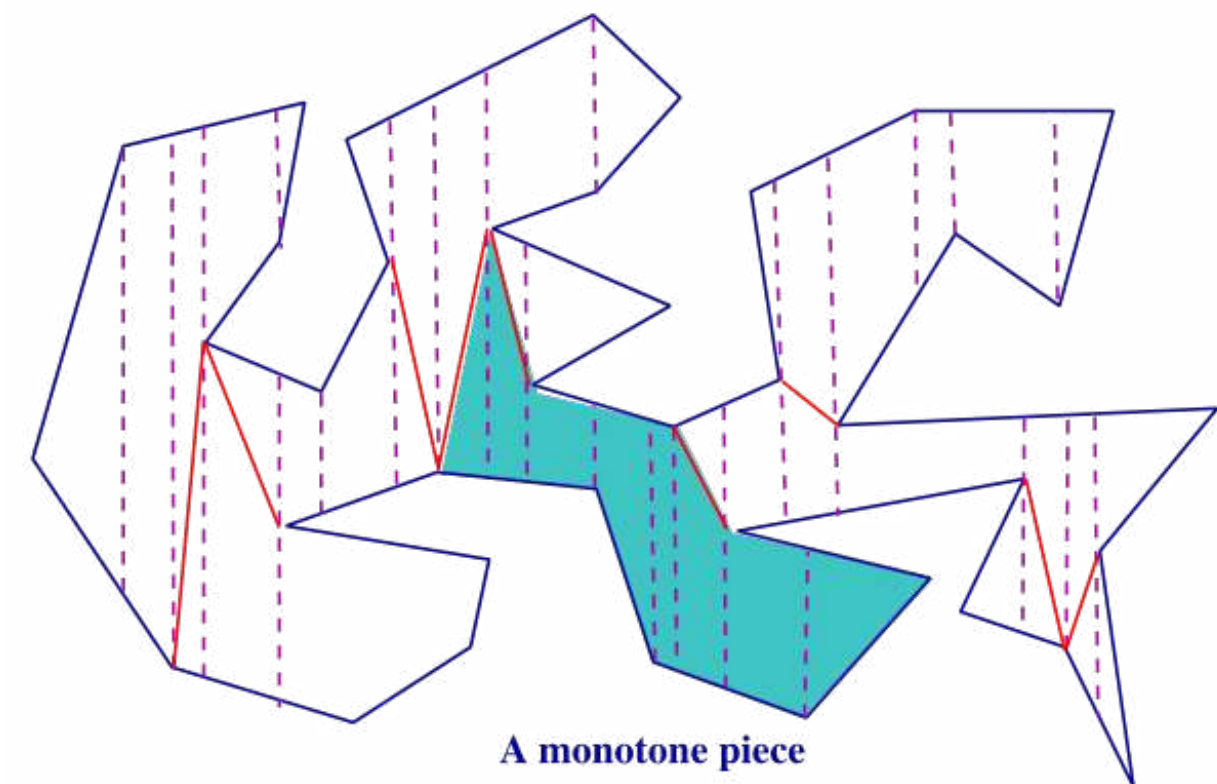
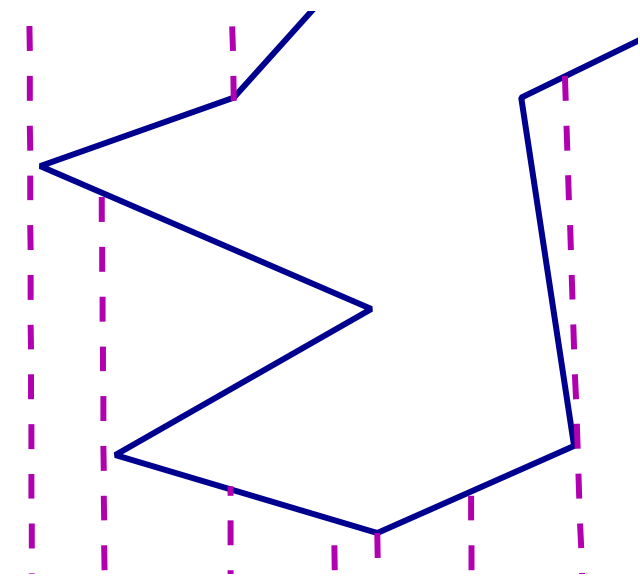
Zerlegung in Trapezoide

- Idee: verwende die Algorithmentechnik **Line-Sweep**
- An jedem Vertex: lege senkrechte Linie durch, verlängere diese 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 / Interval-Tree (→ "Computational Geometry")
- 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 1 (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.

Der Algorithmus zur Triangulation eines x-monotonen Polygons

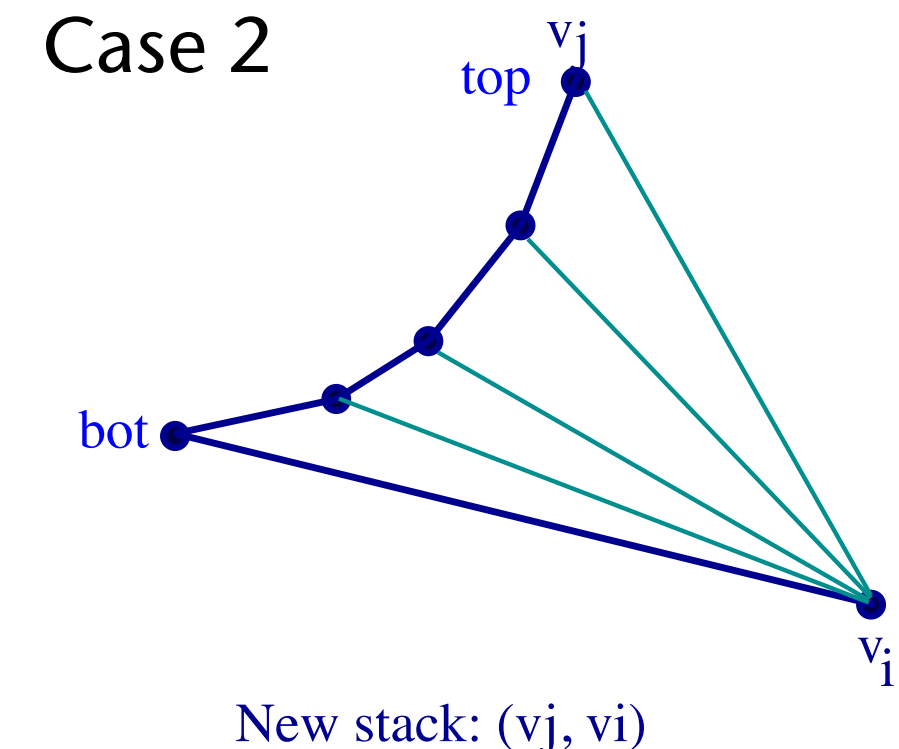
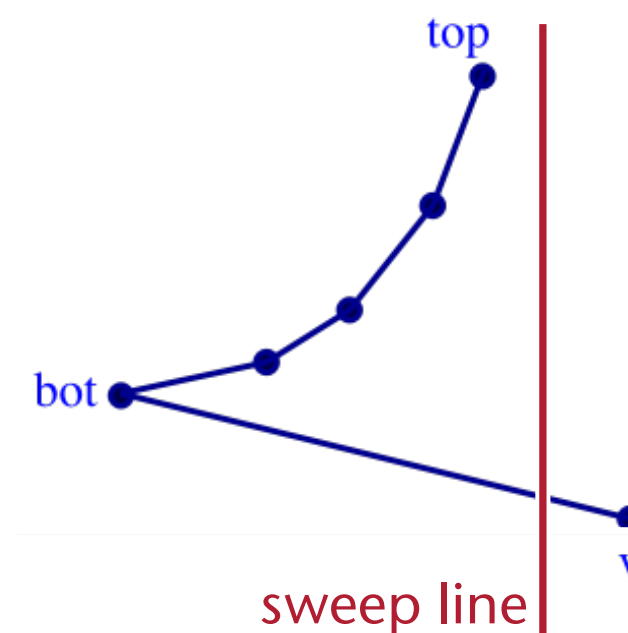
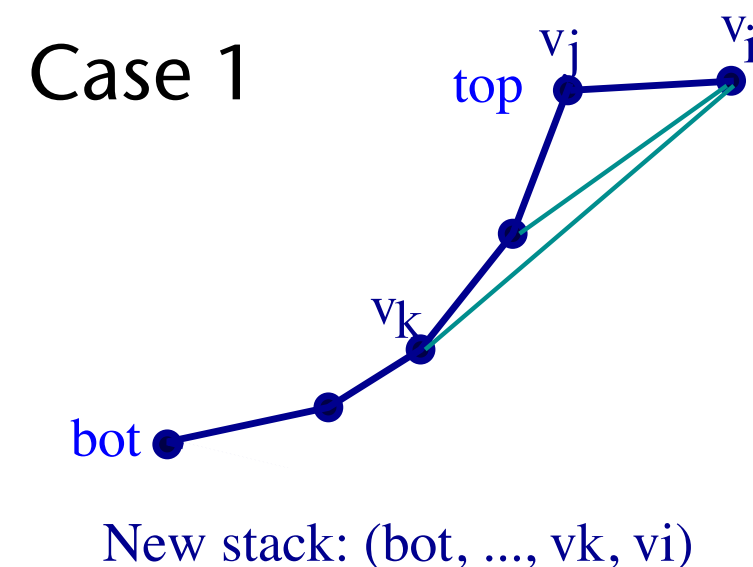
- Beobachtung: in einem x-monotonen Polygon gibt es eine "obere" und eine "untere" Vertex-Kette

```

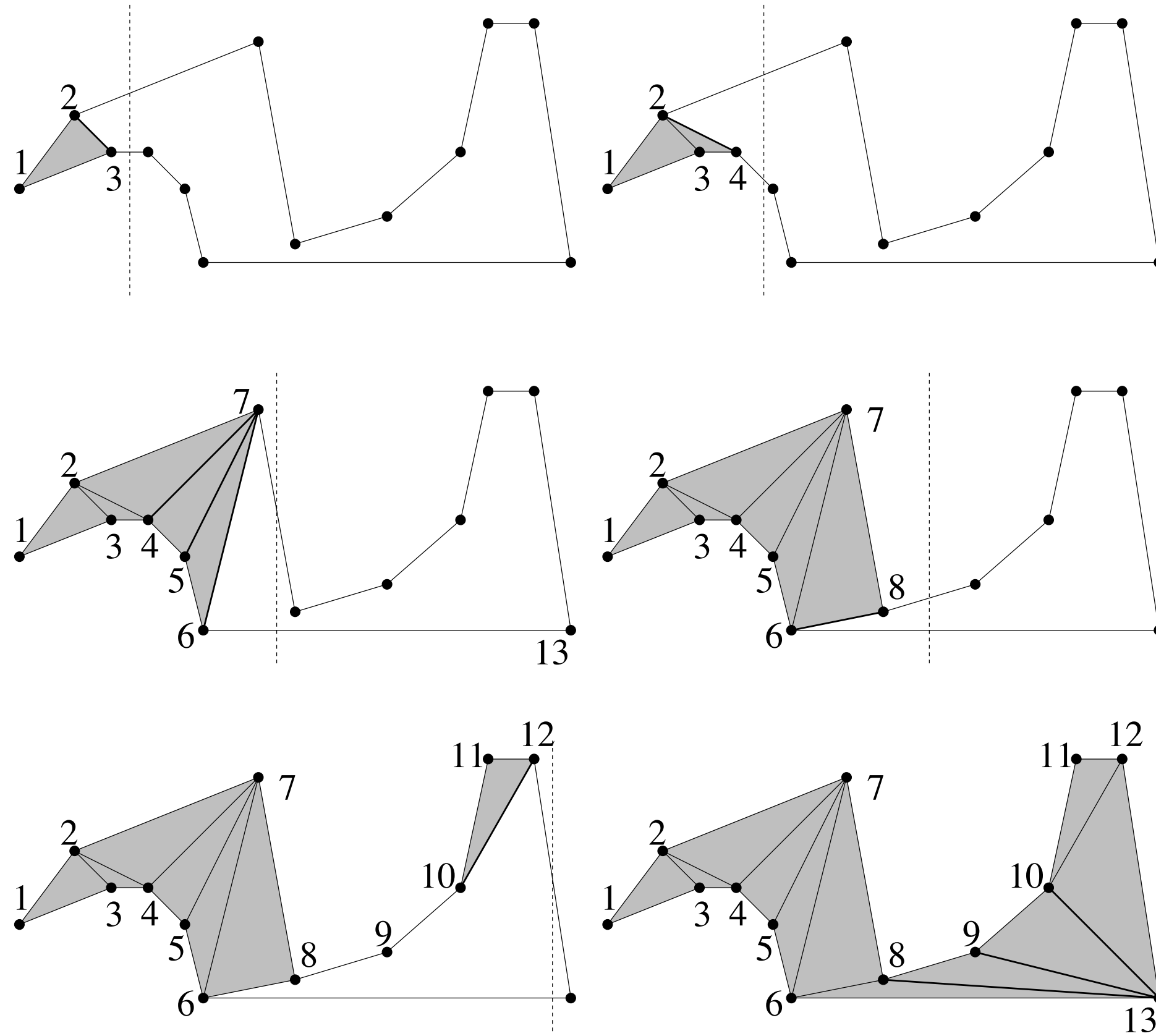
sort all vertices  $v_1, v_2, \dots, v_n$  along  $x$ , monotonically increasing
push  $v_1, v_2$  on stack
for  $i = 3 \dots n$ :
    # at this point, vertices  $v_q$  (=bottom),  $\dots, v_j$  (=top) are on stack
    case 1:  $v_i$  and  $v_j$  are on the same chain (upper or lower)
        insert diagonals  $v_i v_j, \dots, v_i v_k$  (if any),
            where  $v_k =$  last feasible vertex
        if some diagonals were feasible ( $k < j$ ):
            pop  $v_j, \dots, v_{k+1}$ 
            push  $v_i$                                      # now,  $v_q, \dots, v_k, v_i$  are on stack
    case 2:  $v_i$  is on "opposite" chain from  $v_j$  (=top of stack)
        insert all diagonals  $v_i v_j, \dots, v_i v_q$ 
        save  $v_j$ ; clear stack
        push  $v_j$  and  $v_i$  onto stack                     # start new sequence of reflex vertices
    
```


Korrektheitsbetrachtungen (und Verständnis)

- Schleifen-Invarianten:
 - Alle Vertices auf dem Stack bilden eine Folge von Reflex-Vertices (**Reflex-Kette**) (keine linken oder rechten Reflex-Vertices!)
 - Sie liegen alle auf der oberen Vertex-Kette, oder alle auf der unteren Vertex-Kette
 - 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:

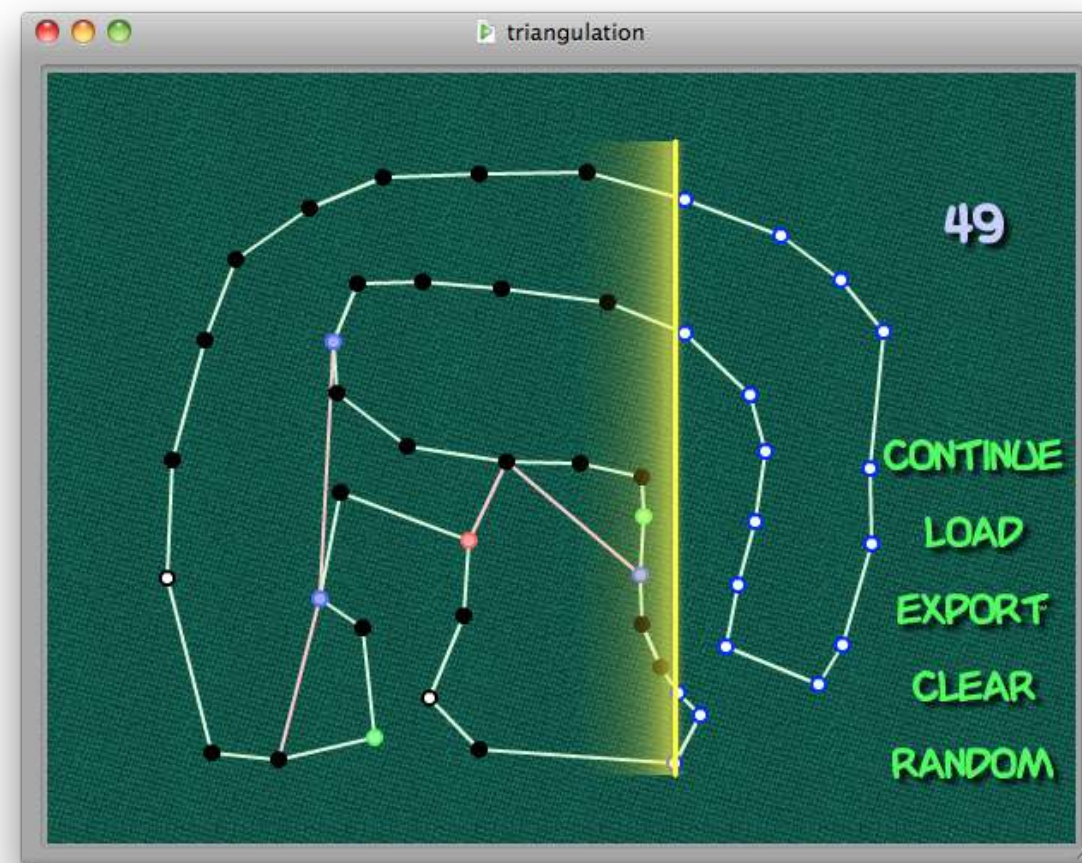


Beispiel



Laufzeit

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

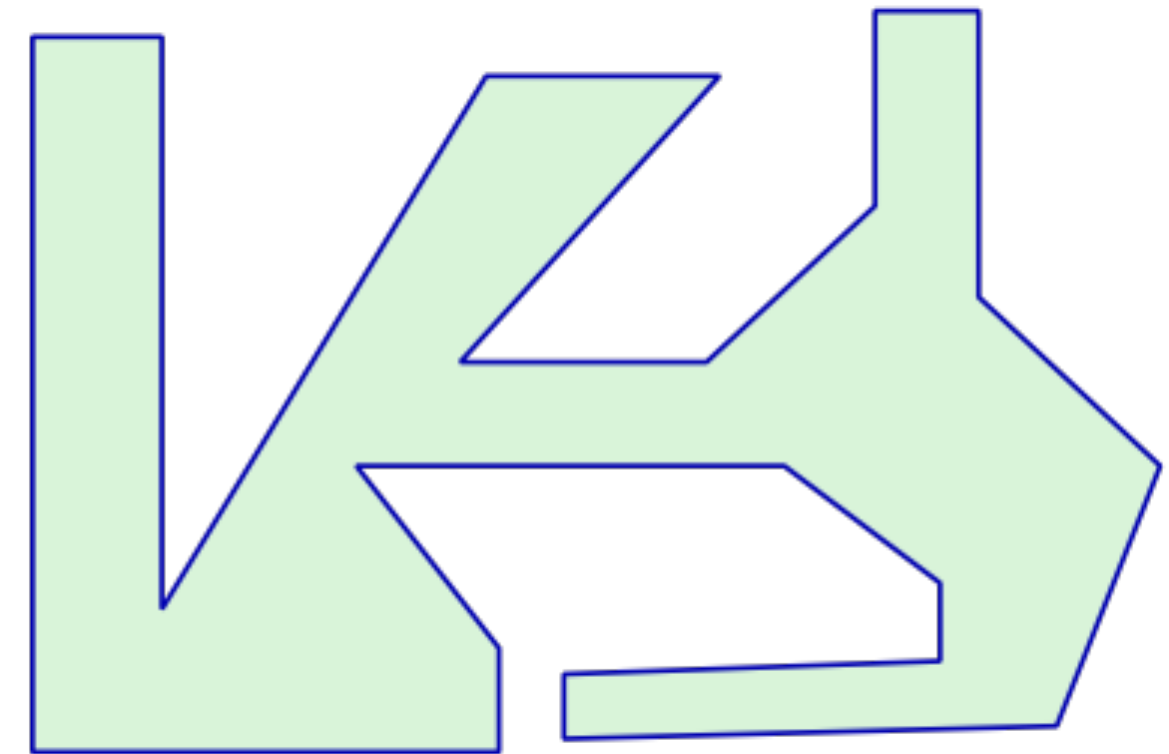


<http://computacion.cs.cinvestav.mx/~anzures/geom/triangulation.php>

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.

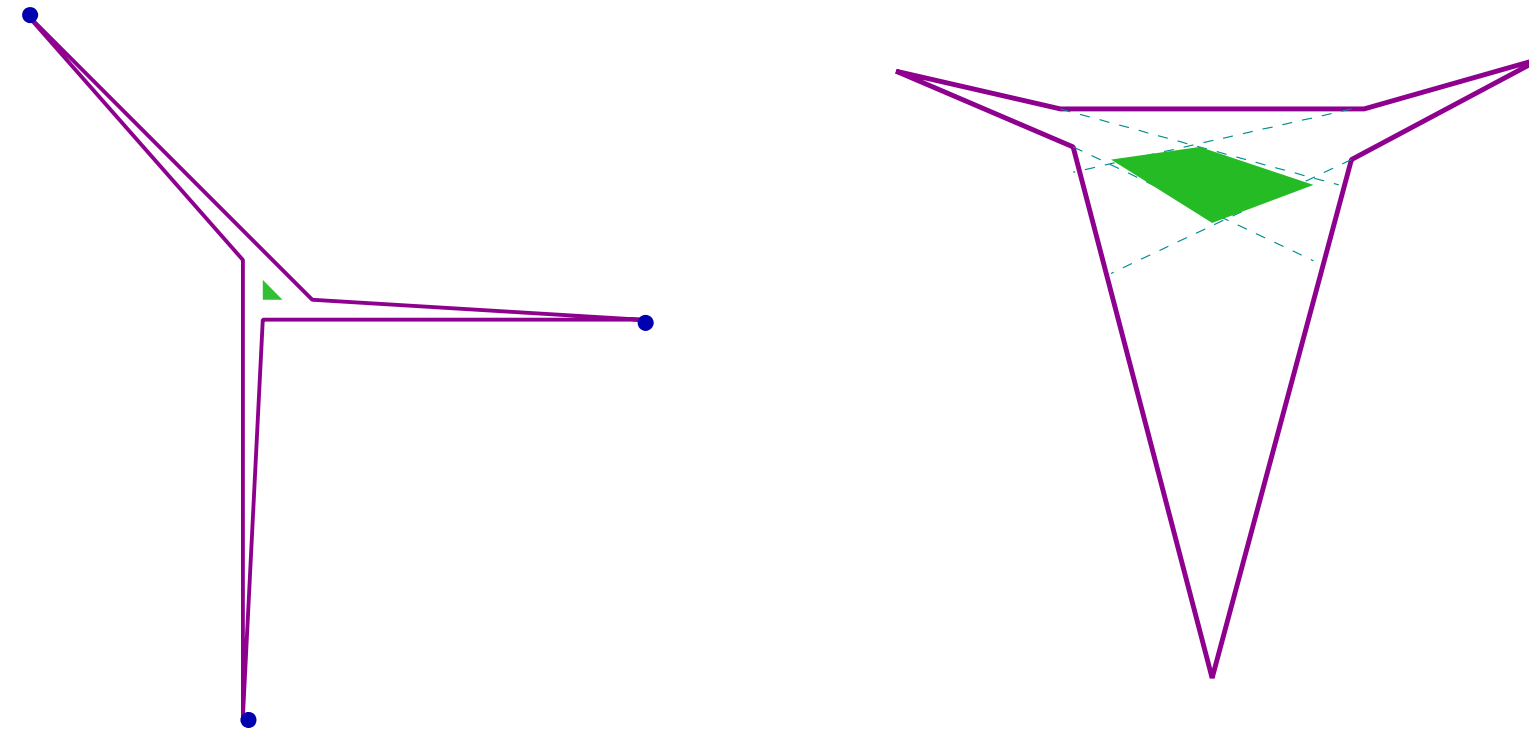


- Geschichte:

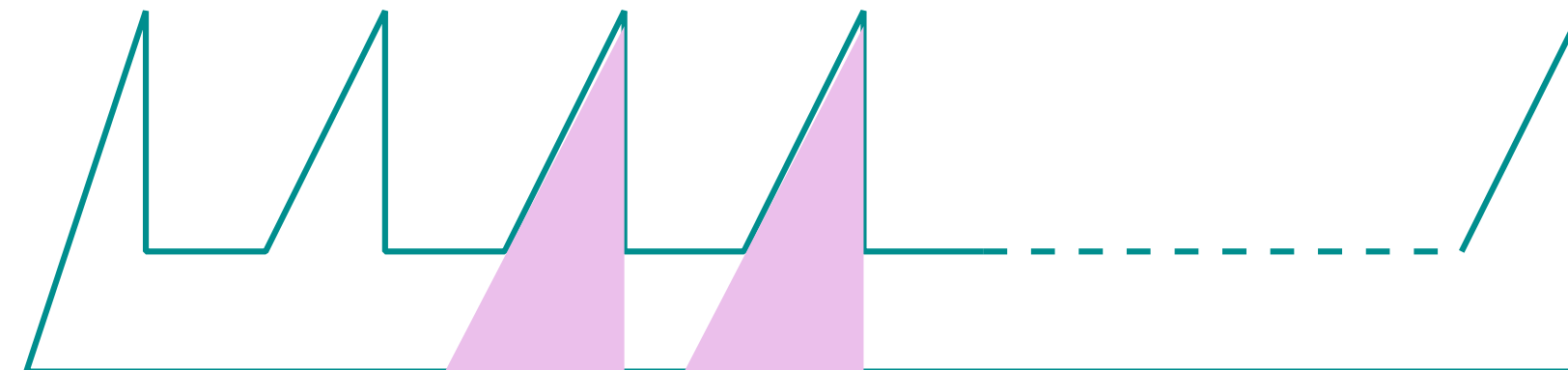
Victor Klee stellte dieses Problem Václav Chvátal auf einer Mathematiker-Konferenz 1973. Dieser fand sofort einen (sehr komplizierten) Beweis, der später mittels Triangulierung sehr vereinfacht wurde.

Bemerkungen

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



- Obere Schranke wird auch angenommen:

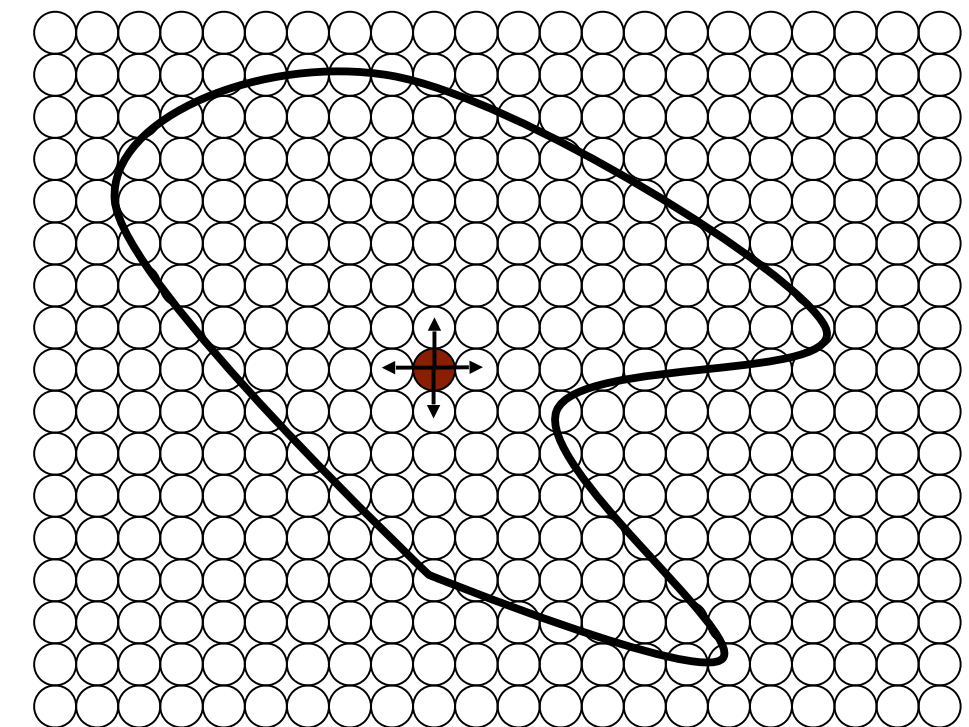
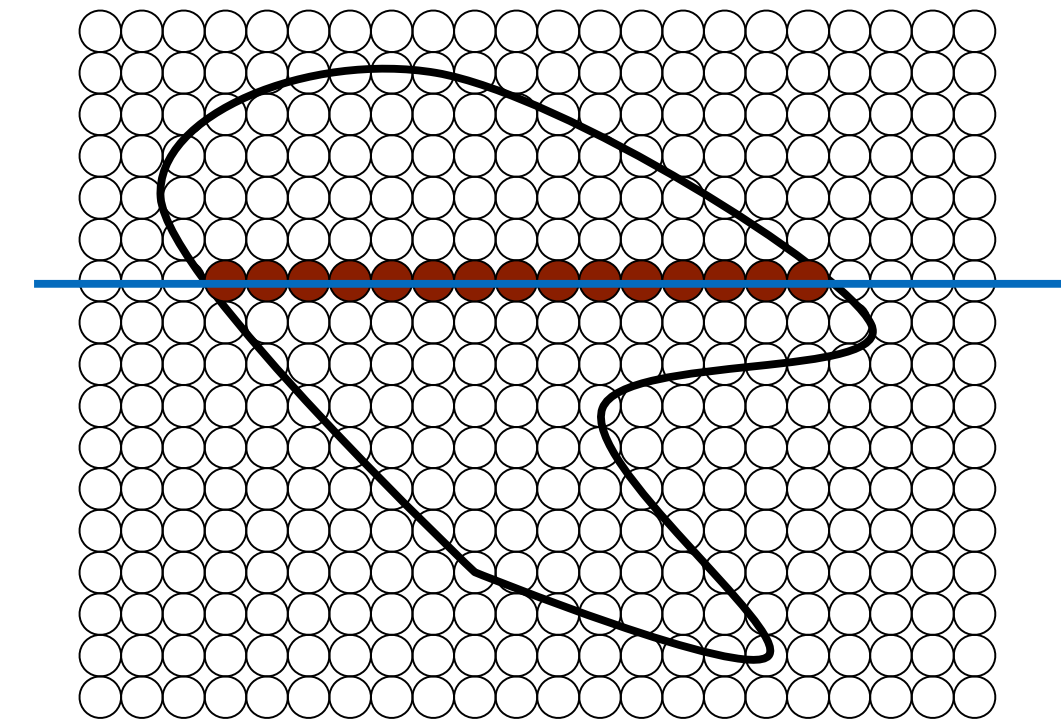


Auflösung: Beweis für *Art Gallery Theorem*

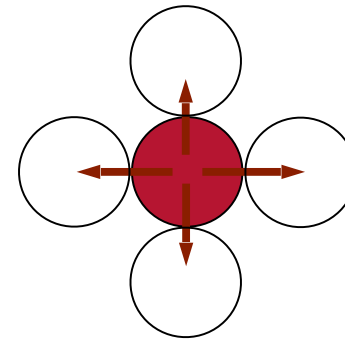
- Mit Triangulation ganz einfach ...
 - Achtung: es muss die Triangulation sein, die durch Ear-Clipping entsteht!

Das Füllen nicht-polygonaler Regionen

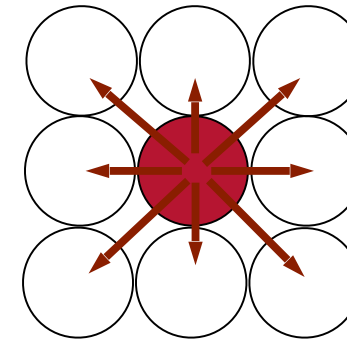
- Gegeben:
 - Rand einer Region, definiert durch Pixel mit einer bestimmten Farbe
 - Häufig in 2D-Zeichenprogrammen
 - Ähnliche Aufgabe zu Polygon-Scan-Conversion
- Erste Methode: wie Polygon-Scan-Conversion
- Zweite Methode: **Flood Fill**
 - Wähle ein Pixel innerhalb des Polygons
 - Färbe Nachbar-Pixel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



- Definition der Zusammenhangsnachbarn:



4-Nachbarschaft

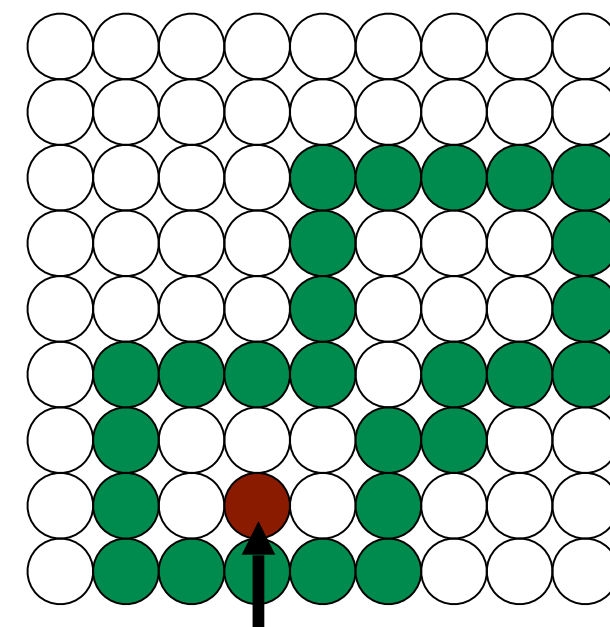


8-Nachbarschaft

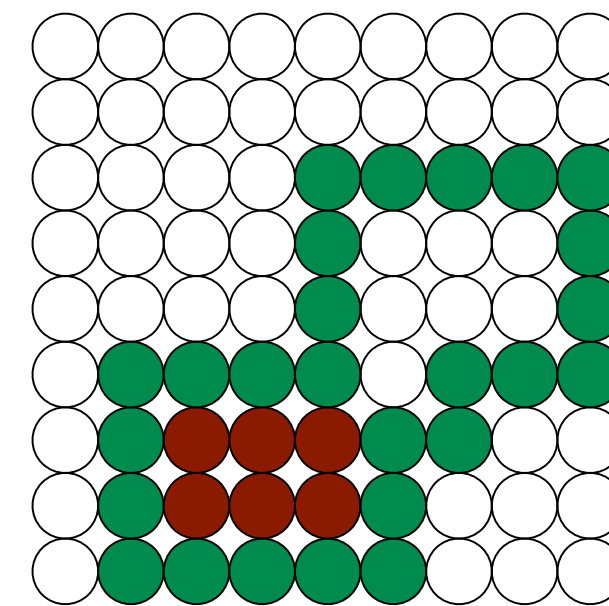
- Rekursion (depth-first search):
 1. Setze das Pixel auf die neue Füllfarbe
 2. Besuche rekursiv alle diejenigen Nachbarn, die noch nicht die neue Farbe haben, und nicht die Randfarbe
- Abbruchkriterium: Region ist definiert als **zusammenhängendes** Gebiet mit Pixeln der
 - Farbe $\neq$ Weiß (Rand = Weiß), oder
 - Farbe $\approx$ user-defined Color (Rand = verschiedene Farben)
 - Benötigt Abstandsmaß zwischen Farben (euklidische Distanz?)

Probleme

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



Startpunkt



Komplett gefüllt

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

Fonts

- **Glyph** = graphische Repräsentation eines oder mehrerer Zeichen (code points)



- Ligaturen werden meist als eigenständiges Glyph entworfen
- **Typeface** = "Design" einer Menge von Glyphs
 - Z.B. Helvetica, Times Roman, Frutiger, Lucida, etc.
 - Wird von Typographen entworfen
 - **Stil** = aufrecht (roman), kursiv (italic), fett (bold), halbfett (semibold), ..
- **Font** = Repräsentation aller Glyphs eines Typefaces, **in einem bestimmten** Format (z.B. TTF oder OTF)
 - Plus Zusatzinfos = Hinting, Kerning, Ligaturen, Font-Metrik, ...



- Kategorien der Glyph-Repräsentation:

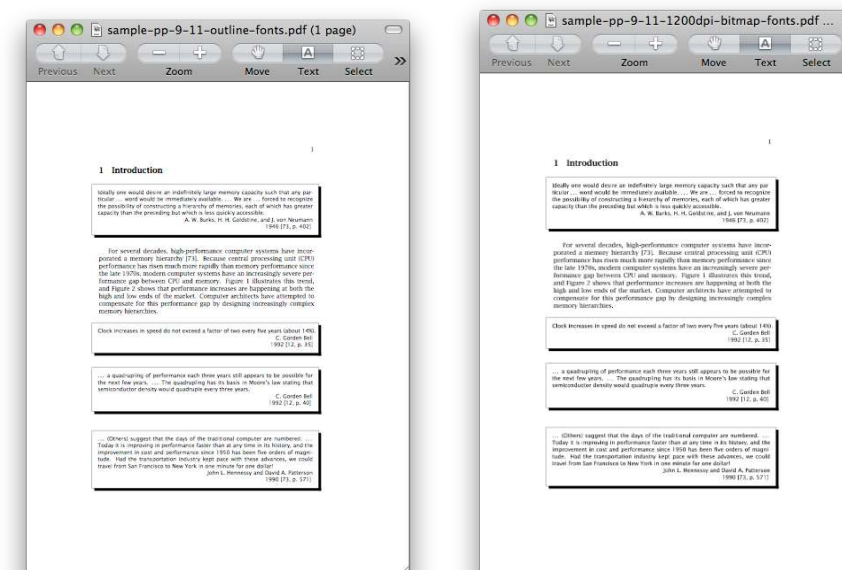
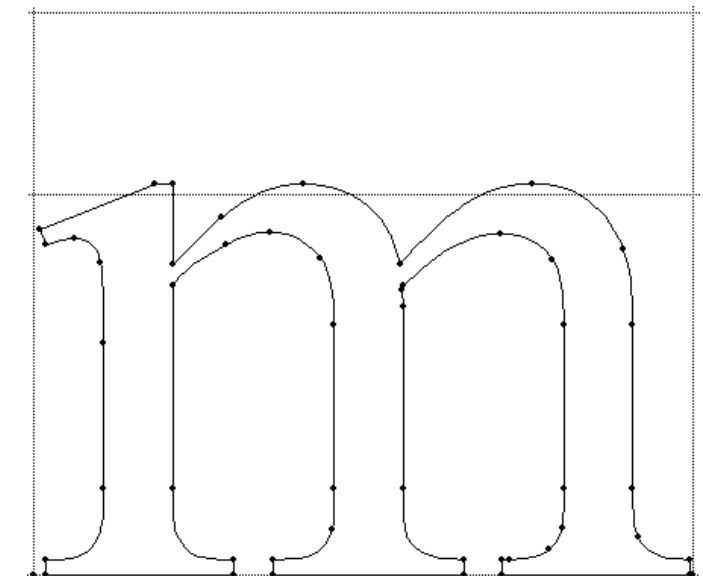
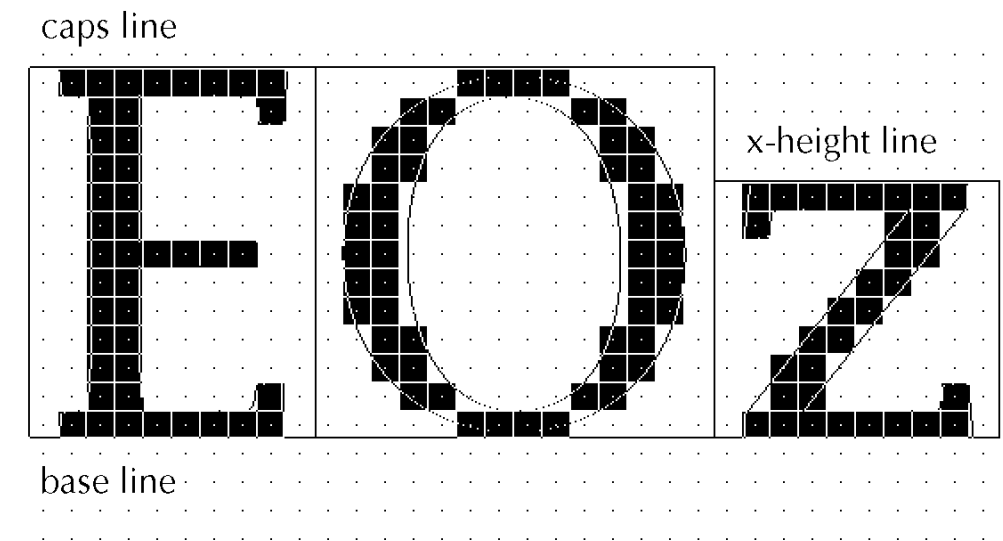
- **Bitmap-Font** = jedes Zeichen ist eine Bitmap (oder mehrere für verschiedene Font-Größen)

- Beispiel-Formate: Adobe Type 3, Portable Compiled Format (PCF)

- **Outline-Font** = jedes Zeichen wird aus sog. Bézier- oder B-Spline-Kurven zusammengesetzt

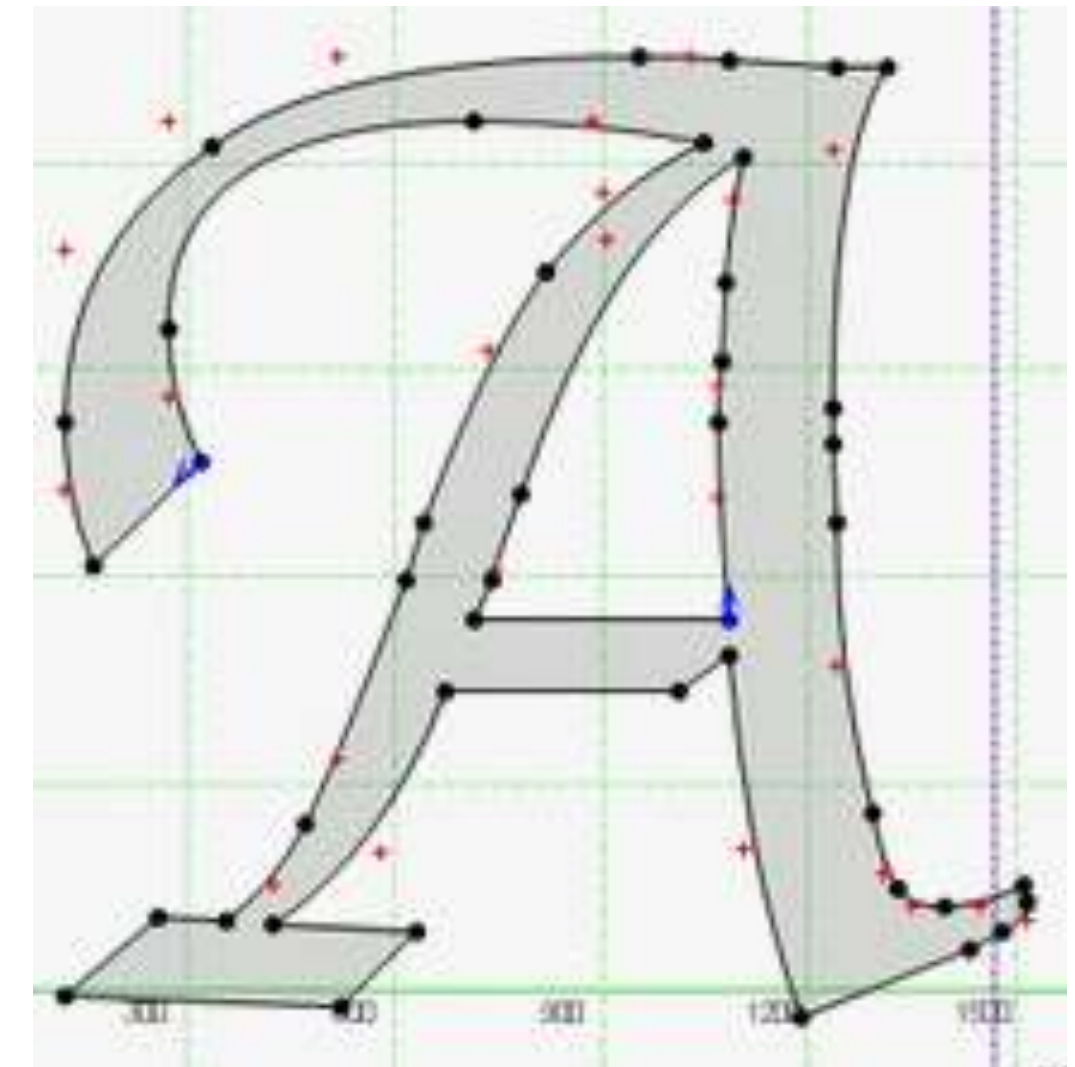
- Adobe Type 1, TrueType, OpenType

- Demo: side-by-side Vergleich



Beschreibung von Outline-Fonts

- Glyph = 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



- Outlines setzen sich zusammen aus quadratischen und kubischen Bézier-Kurven (und Liniensegmenten)

- Quadratische Bézier-Kurven:

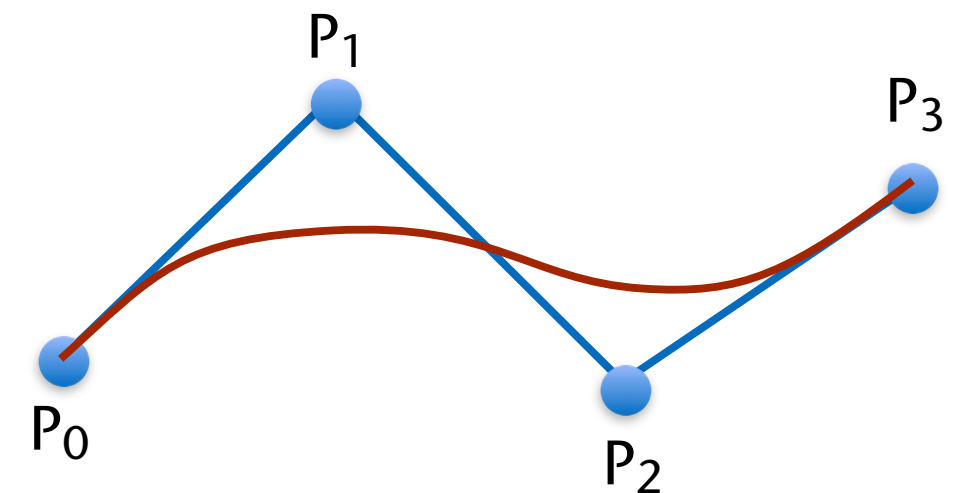
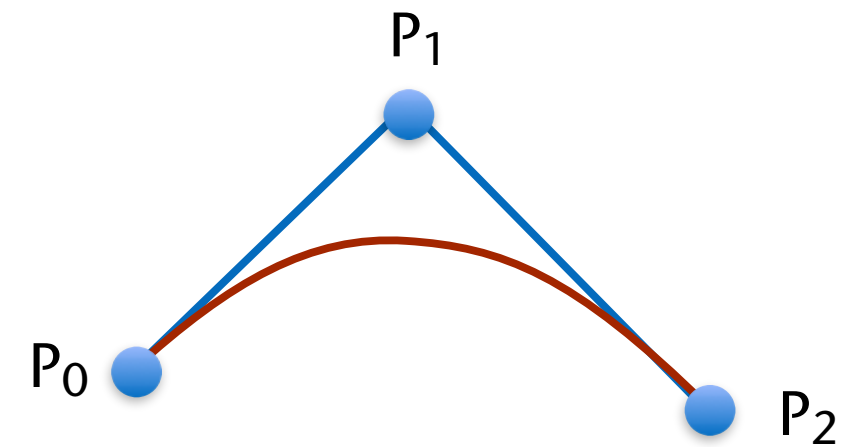
- Definiert durch ein Kontroll-Polygon aus 3 Punkten
- Polynom 2-ten Grades (quadratische Interpol.):

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

- Kubische Bézier-Kurven:

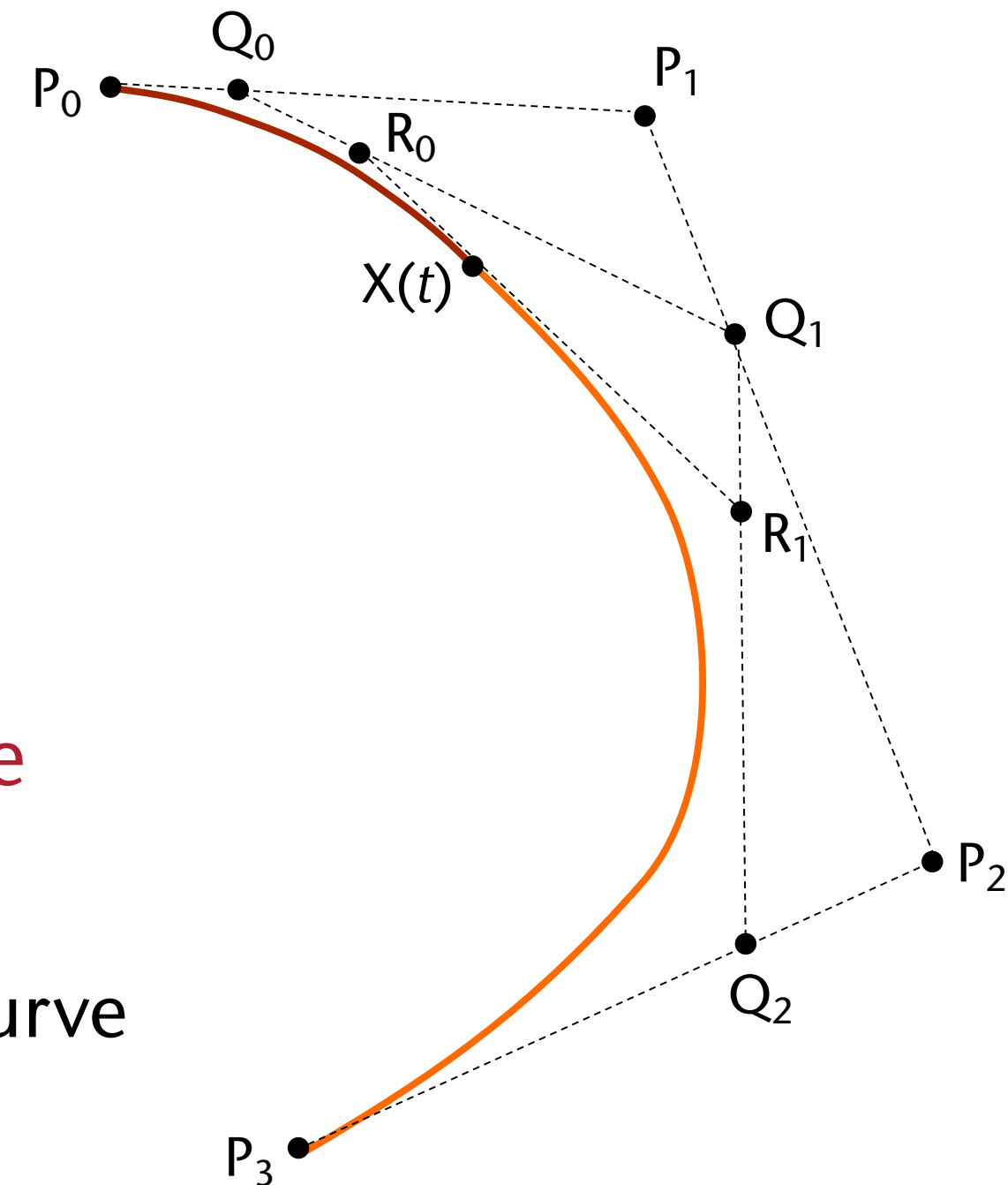
- Kontroll-Polygon hat 4 Punkte
- Polynom ist vom Grad 3:

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



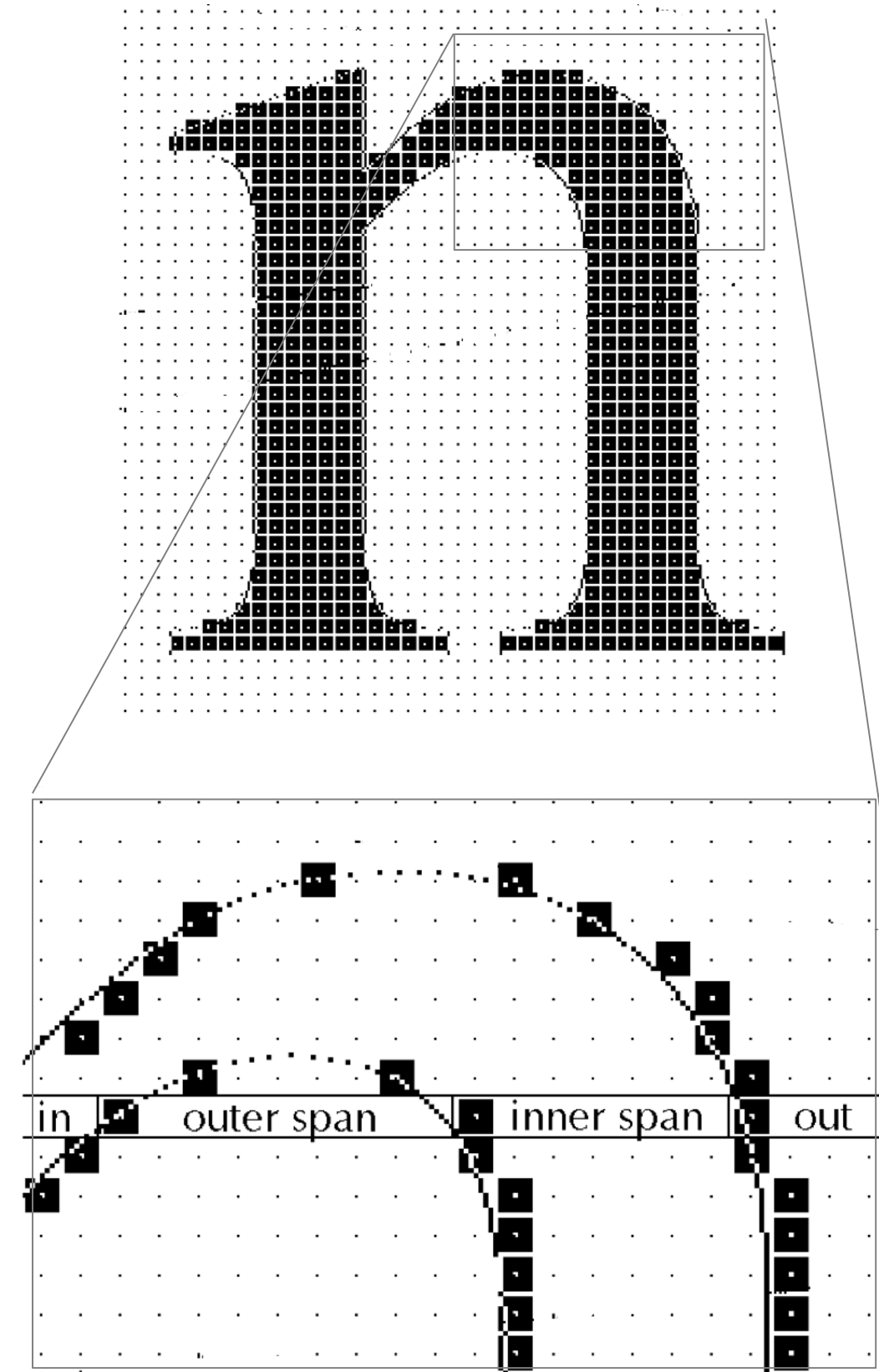
Approximation durch rekursive Subdivision

- Berechnung von $X(t)$ durch rekursive lineare Interpolation (**DeCasteljau-Algorithmus**):
 - Konstruiere Q_i durch lineare Interpolation aus $P_i P_{i+1}$ (z.B. mit $t=0.3$)
 - Analog: konstruiere R_i aus $Q_i Q_{i+1}$
 - Schließlich X durch lin. Interpol. zwischen $R_0 R_1$
- 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
 - Diese schmiegen sich "dichter" an die ursprüngliche Kurve
 - Die kubischen Bézier-Kurven dazu bilden zusammen genau die ursprüngliche Kurve

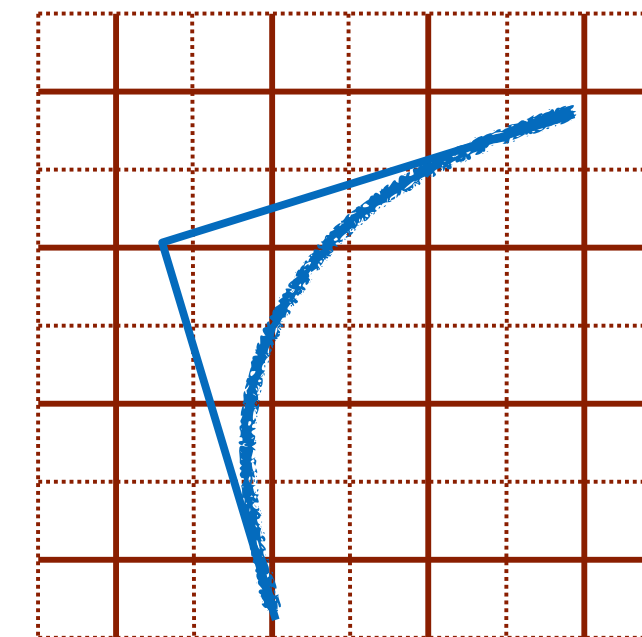
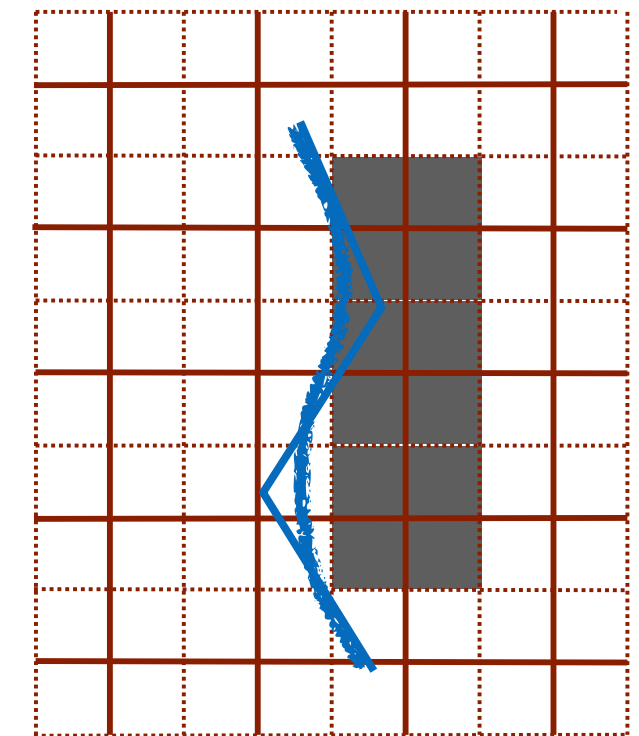
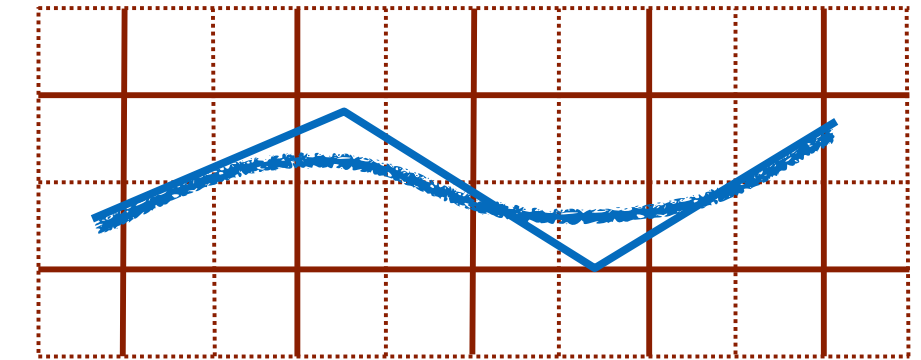


Der Flag-Fill-Algorithmus von Ackland

- Annahmen zunächst:
 - Die einzelnen Pixel sind sehr klein im Vergleich zu dem Zeichen
 - Innerhalb eines Pixels kann man die Kontur durch eine Gerade approximiert
 - Überdeckung des Pixels $> 50\%$ $\Leftrightarrow$ Pixelmittelpunkt liegt "innen"
- Idee des Algorithmus:
 - Berechne die Pixel *auf* den Bézier-Kurven $\rightarrow$ **Flags**
 - Zerlege die Scan-Lines in der BBox des Zeichens in innere und äußere Spans
 - Fülle die inneren Spans

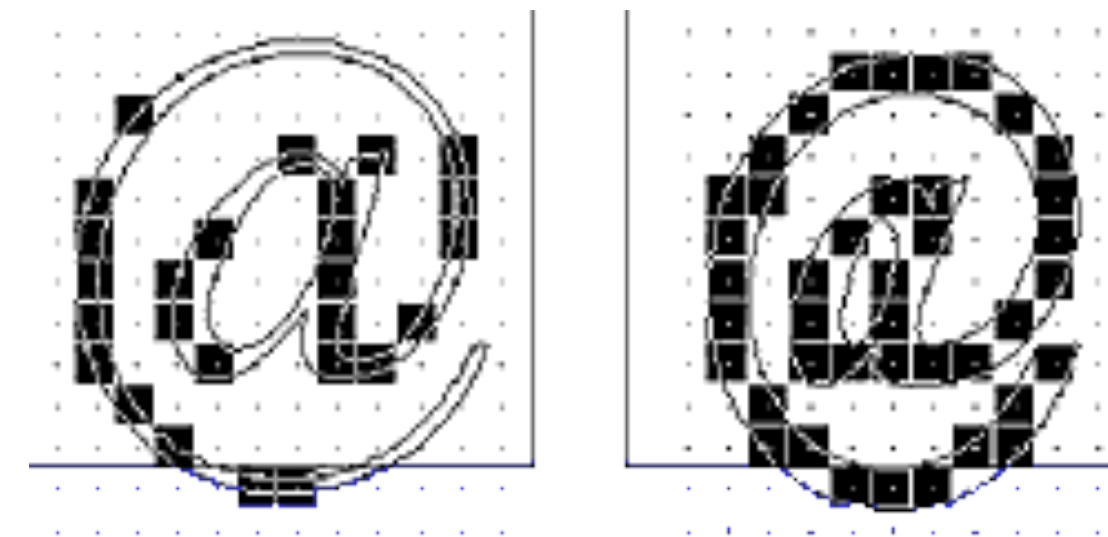
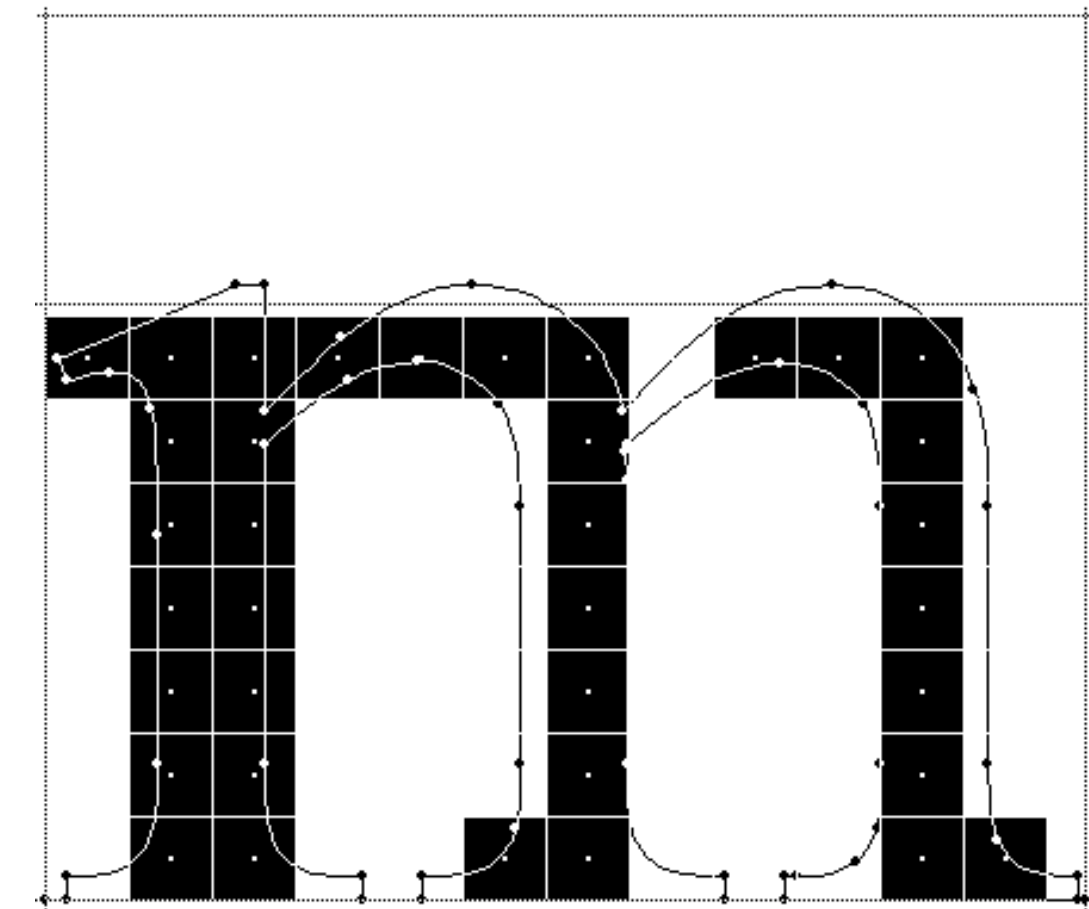


- Betrachte einzeln jede Bézier-Kurve nacheinander, d.h., betrachte deren Kontroll-Polygon
- Fallunterscheidung & Rekursion:
 1. Kontroll-Polygon schneidet keine (horizontale) Scanline → verwerfen
 2. Kontroll-Polygon schneidet eine oder mehrere Scanlines, und schneidet keine vertikale Gitterlinie →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



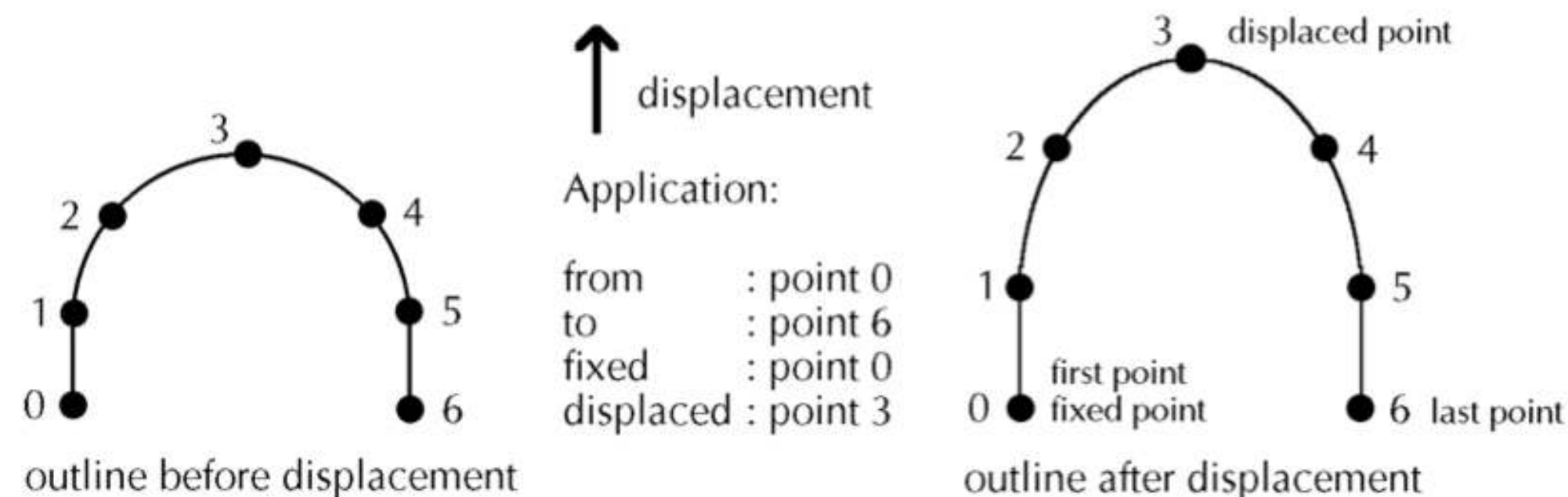
Probleme

- 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 Pixel-Gitterlinie links davon

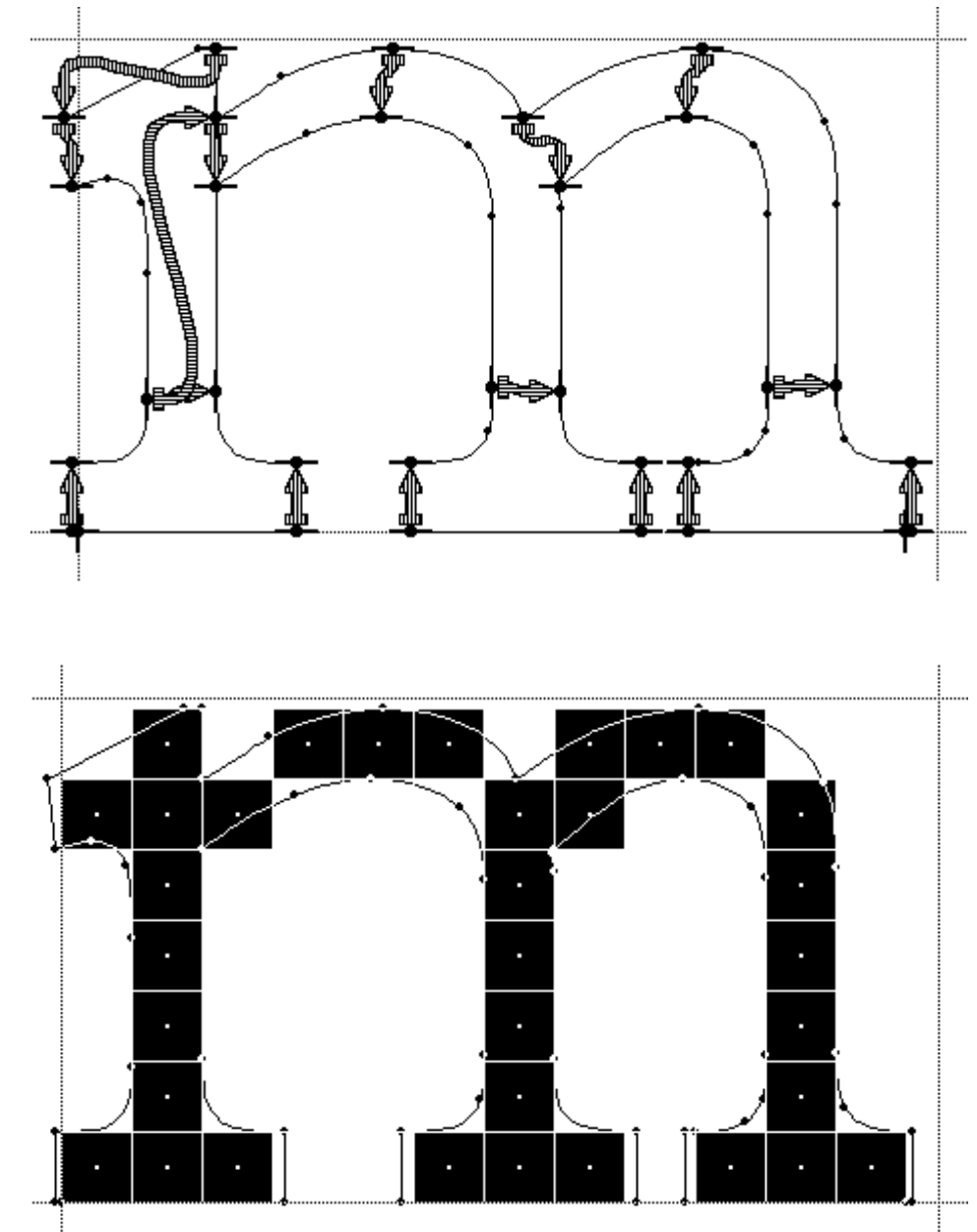


Hinting (a.k.a. Grid Fitting)

- 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;

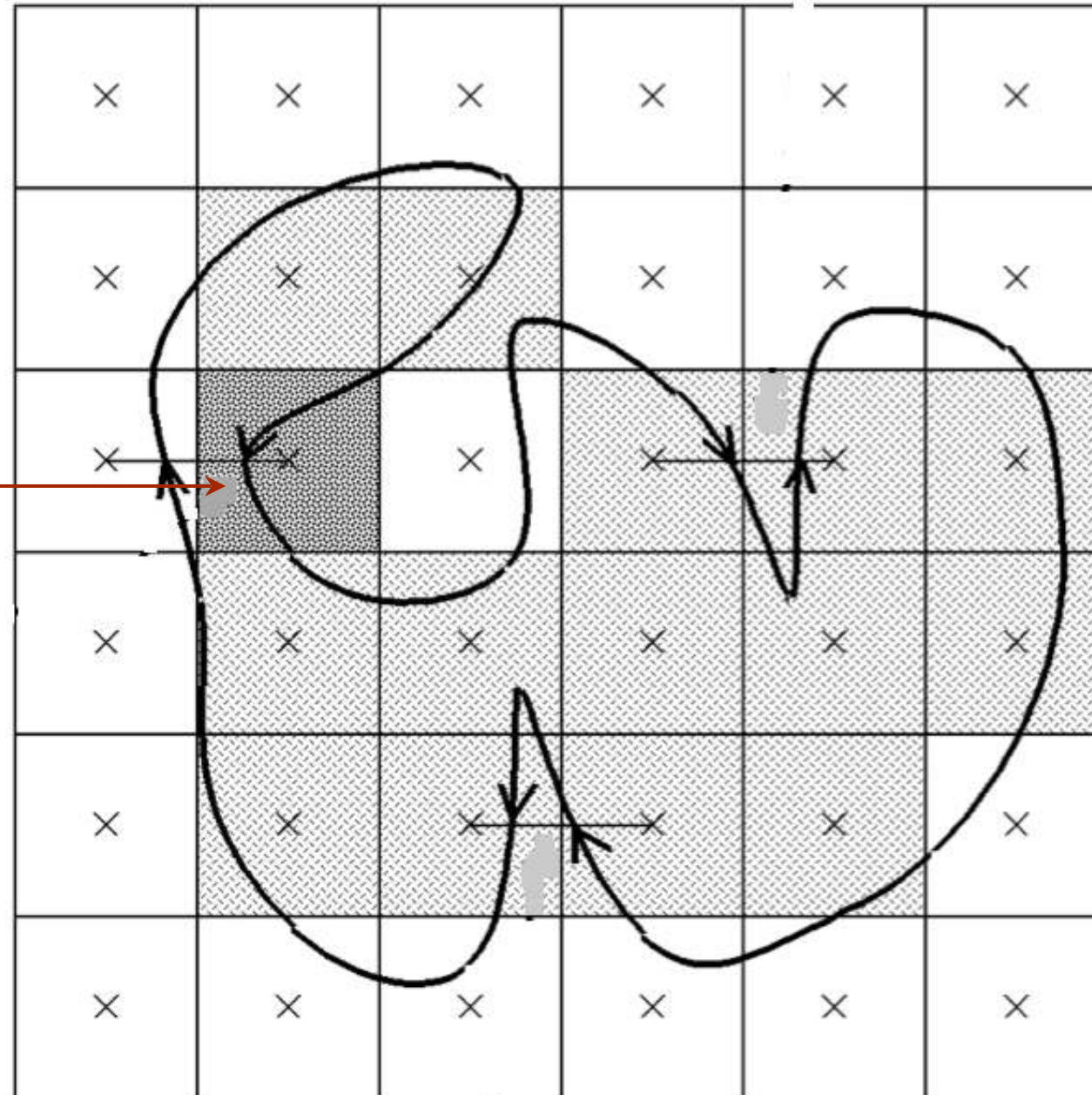


- **Constraints** =
 - welche anderen Punkte mitverschoben werden müssen; und
 - Spezielle Punkte (z.B. Mittelpunkt des O) müssen spezielle Bedingungen erfüllen (z.B. immer in der Mitte eines Pixels liegen)
 - Muß vom Font-Designer gemacht werden
- NB: Diese Art Hinting wird nur bei TrueType-Fonts gemacht; völlig anders bei Type1 (Adobe) ...



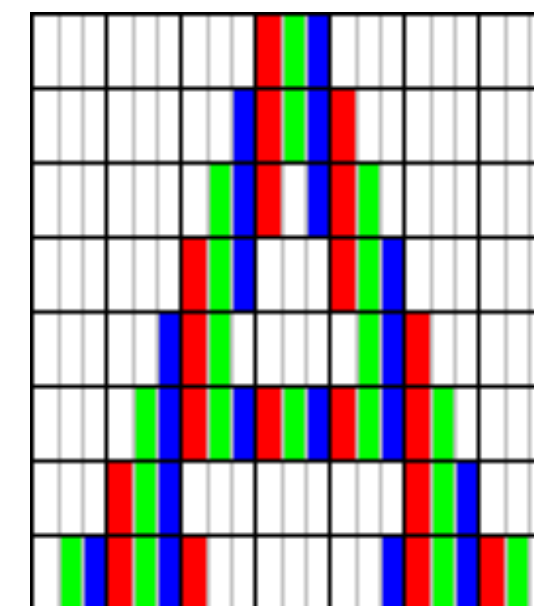
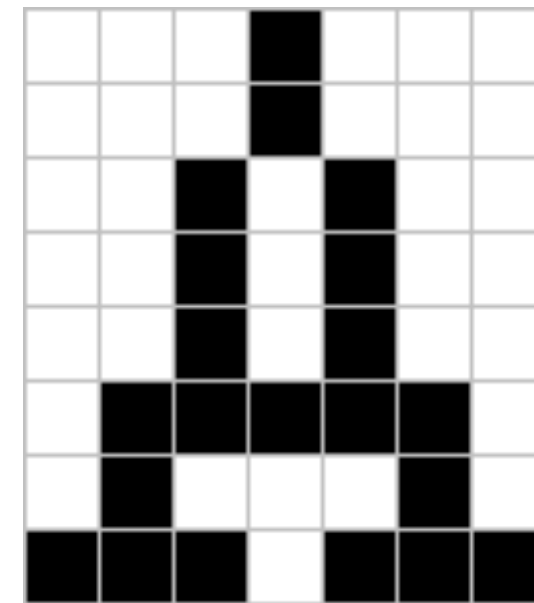
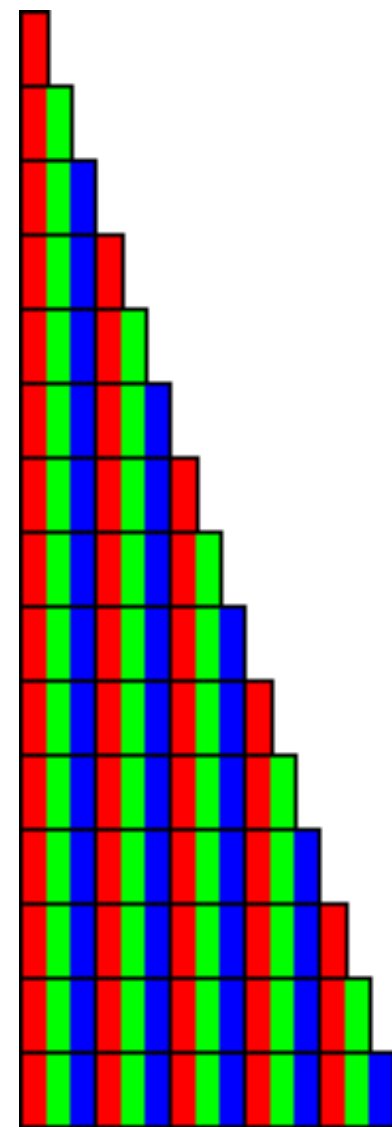
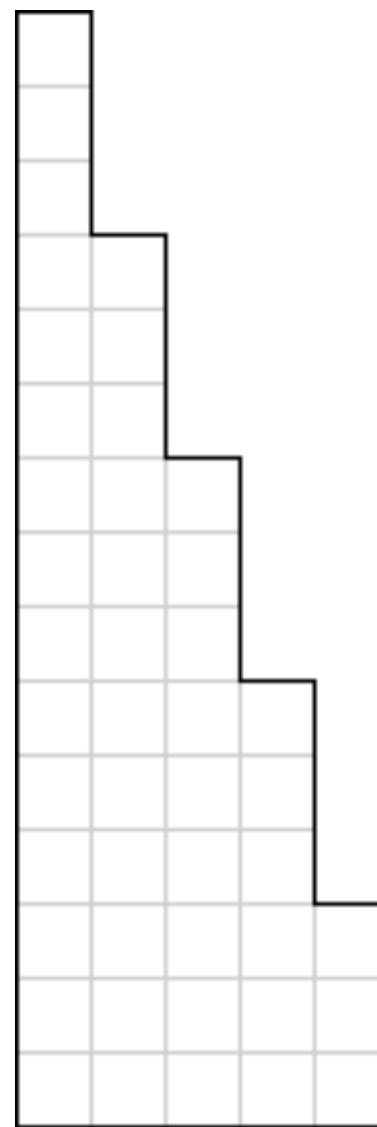
- Dropout-Kontrolle:

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

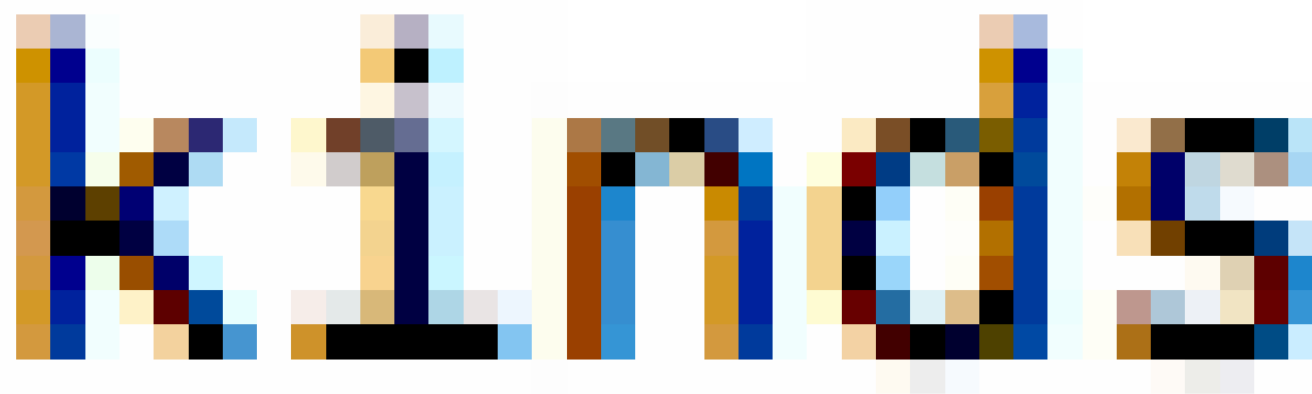
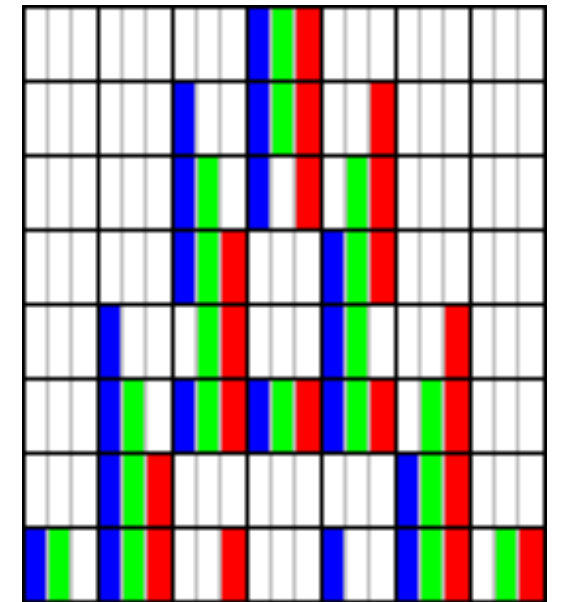


Sub-Pixel Font-Rendering (aka. Sub-Pixel Anti-Aliasing)

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



- Einfacher Trick 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)
 - Evtl. *Color fringing*



- 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