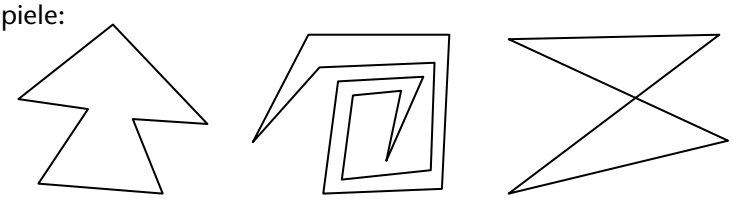
Füllen nicht-einfacher Polygone

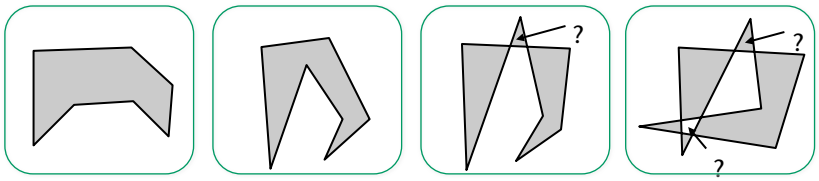
- Def.: Polygon ist einfach ↔ Randkurve hat keinen Schnittpunkt
- Beispiele:



einfach (& hor. konvex) | einfach | nicht einfach

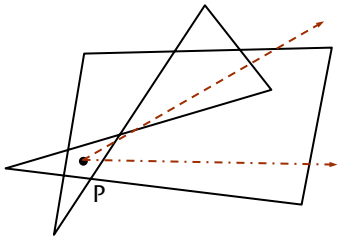
- Eigenschaften:
  - Topologisch äquivalent zu einer 2-dimensionalen Scheibe
  - Folge: es gibt ein wohldefiniertes Inneres/Äußeres
- Wie kann man solche Polygone füllen?
  - Verwende für jedes Pixel einen intuitiv „korrekten“ Inside-/Outside-Test

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



Test 1: Odd-Even Rule

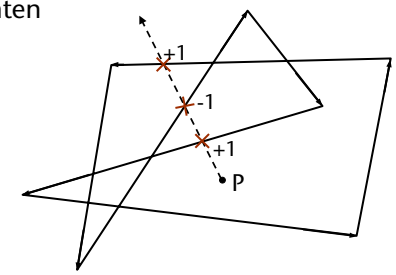
- Zeichne Strahl von Punkt P nach unendlich in irgend eine Richtung
- Zähle Anzahl Schnittpunkte mit dem Kantenzug
- Falls Anzahl ungerade → P innerhalb
- Achtung, falls Strahl Eckpunkt genau trifft!
- Effiziente Schnittberechnung: wähle horizontalen Strahl



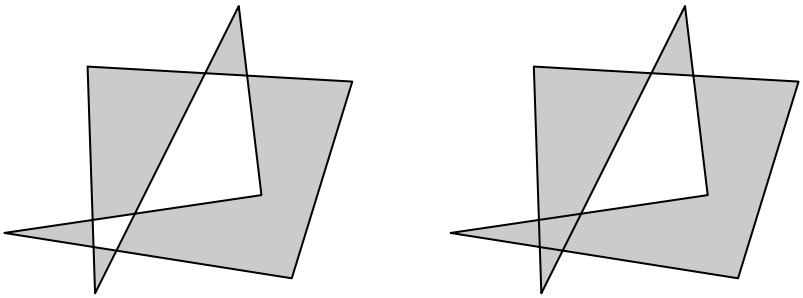
- Vorteil: funktioniert genauso mit Polyedern im 3D (und höherdim.)

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

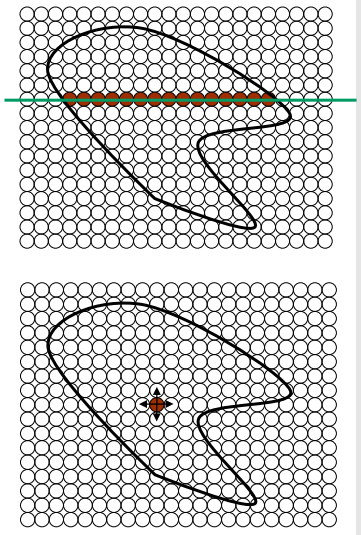


Odd-even Rule

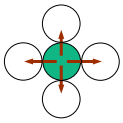
Non-zero Winding Number

Füllen von Regionen

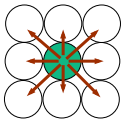
- Ähnliche Aufgabe zu Polygon-Scan-Conversion
  - Häufig in 2D-Zeichenprogrammen
- Erste Methode: wie Polygon-Scan-Conversion
- Zweite Methode:
  - Flood Fill: Wähle ein Pixel innerhalb des Polygons. Färbe rekursiv angrenzende Pixel bis das gesamte Polygon gefüllt ist



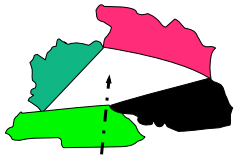
- Region ist identifiziert durch bestimmte Farbe (z.B. Weiß)
- Wähle ein Pixel innerhalb der Region
  - Region ist definiert als zusammenhängendes Gebiet mit alter Farbe
- Rekursion:
  1. Hat Pixel die alte Farbe, dann weise diesem Pixel die Füllfarbe zu
  2. Färbe rekursiv alle diejenigen Nachbarn,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



4-Verbindungen



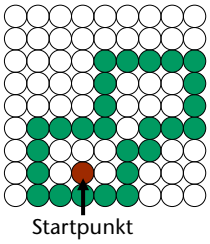
8-Verbindungen



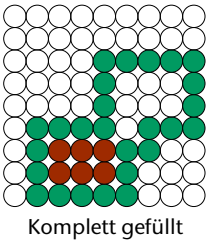
Zu füllende Region

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

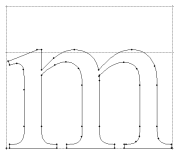
- Oft möchten wir einen Bereich mit einem Muster (z.B. gestrichelt) versehen, nicht nur eine Farbe (*stippling*)
- Definiere ein *n* x *m* Pixmap (oder Bitmap), welche wir auf den Bereich abbilden möchten:

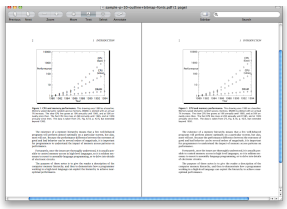
5x4 pixmap | Mit Muster zu versehendes Objekt | Gemustertes Objekt

- 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?

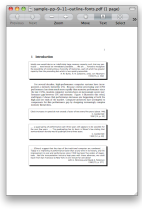
Font-Rendering

- 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

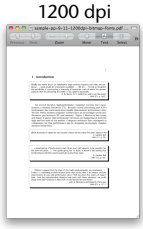




outline



1200 dpi



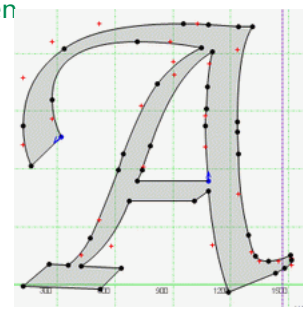
Begriffe

- 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

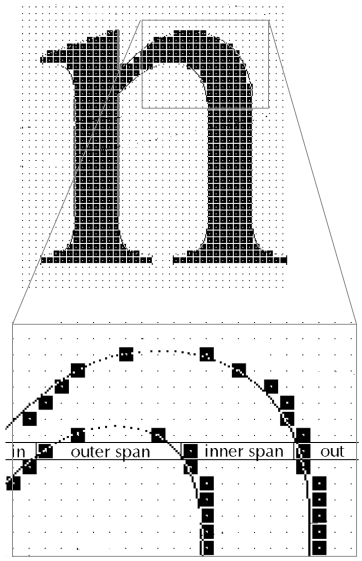
- Zeichen = Menge von geschlossenen Kurvenzügen (*Outlines*)
- Kurvenzug = Menge von Kontrollpunkten
- Umlaufsinn definiert innen / außen:
  - "Links von der Kurve" = innen (oder umgekehrt ...)
- Achtung: Kurvenzüge müssen überschneidungsfrei sein!



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

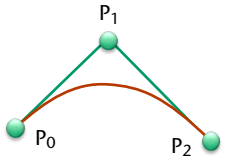
Der Flag-Fill-Algorithmus von Ackland [1981]

- 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

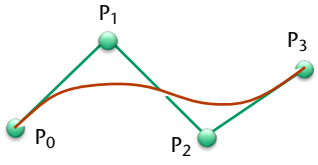


- Outlines setzen sich zusammen aus quadratischen und kubischen Bézier-Kurven
- Quadratische Bézier-Kurven:
  - Definiert durch ein Kontroll-Polygon aus 3 Punkten
  - Polynom 2-ten Grades:

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

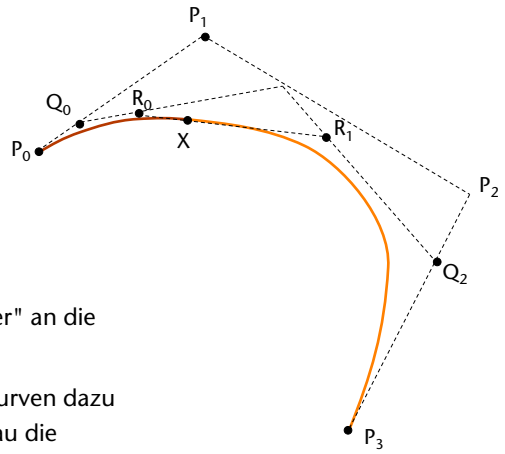


- Kubische Bézier-Kurven:
  - Kontroll-Polygon hat 4 Punkte
  - Polynom ist vom Grad 3:

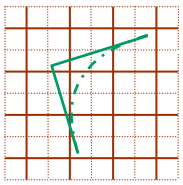


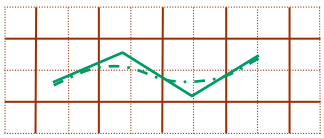
$$P(t) = (1-t)^3 P_0 + 3(1-t)^2 tP_1 + 3(1-t)t^2 P_2 + t^3 P_3\ , \quad t \in [0,1]$$

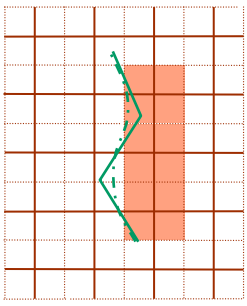
- Approximation durch rekursive Subdivision:
  - Aus P_0, ..., P_3 kann man zwei neue Kontroll-Polygone P_0, Q_0, R_0, X und X, R_1, Q_2, P_3 konstruieren
  - Schmiegen sich "dichter" an die ursprüngliche Kurve
  - Die kubischen Bézier-Kurven dazu bilden zusammen genau die ursprüngliche Kurve



- 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

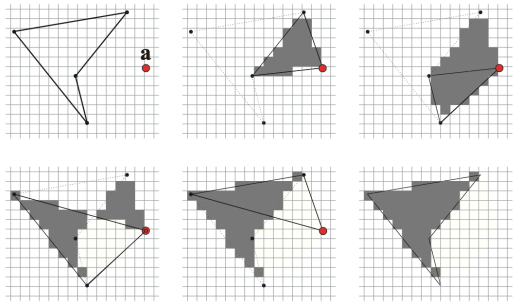






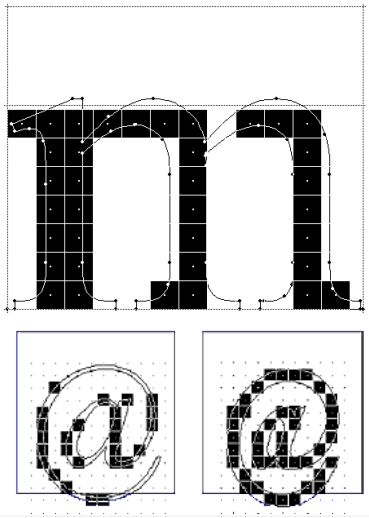
Der XOR-Algorithmus

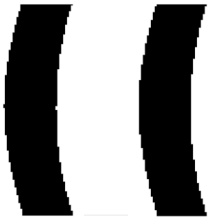
- Gegeben ein geschlossener, überschneidungsfreier Polygonzug, oder mehrere, die eine Region in der Ebene definieren
- Wähle einen beliebigen Anker-Punkt A
- Betrachte alle Kanten PQ der Reihe nach:
  - Invertiere alle Pixel im Dreieck ΔAPQ
- Beispiel:



Probleme

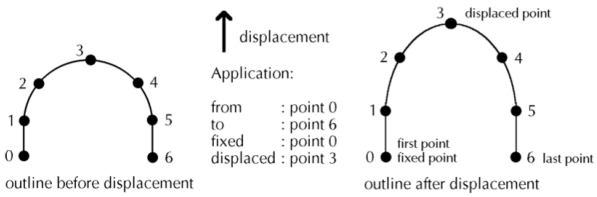
- Bei kleinen Font-Größen können, je nach "Phase", folgende Probleme auftreten:
  - Drop-outs
  - Ungleiche Dicke der Stämme
  - Serifen in verschiedene Richtung
- Phase = Abstand zwischen linkem Rand der BBox und vertikale Gitterlinie links davon



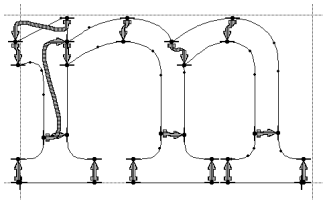


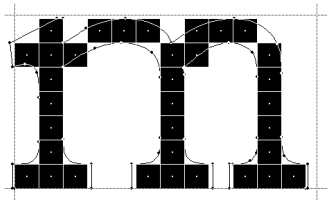
Hinting (grid fitting)

- 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;



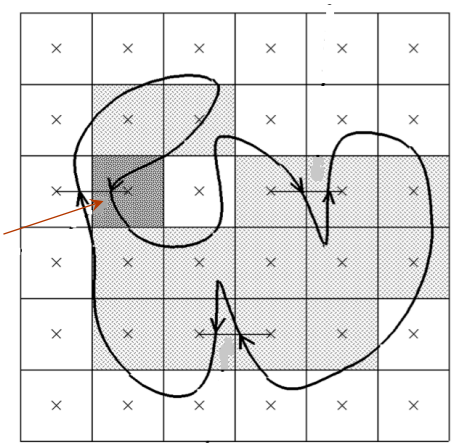
- Hinting (Fortsetzung):
  - welche anderen Punkte dann mitverschoben werden müssen (Constraints)
  - Muß vom Font-Designer gemacht werden
- NB: Diese Art Hinting wird nur bei TrueType-Fonts gemacht
  - Völlig anders bei Type1 ...





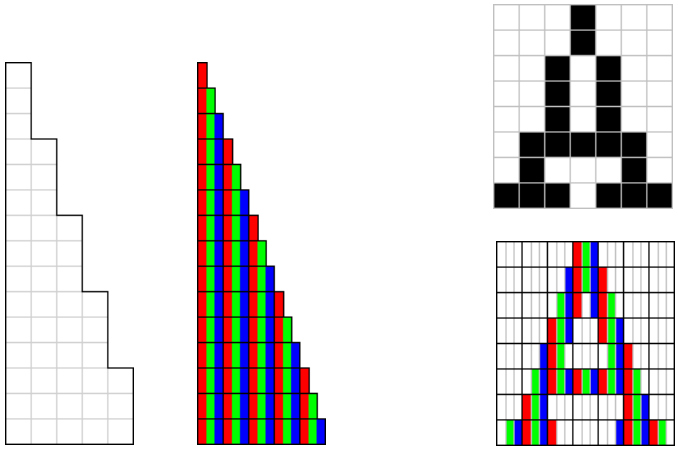
- Dropout-Kontrolle:

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

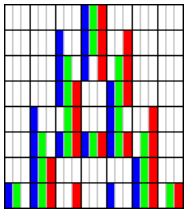


Sub-Pixel Font-Rendering

- 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