Vektorisierung von Konturen

Prof. Dr. Klaus Jung

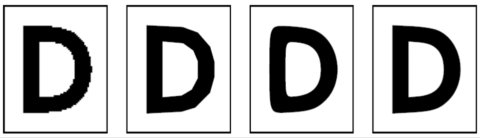
Hochschule für Technik und Wirtschaft Berlin
University of Applied Sciences

Vektorisierung nach *potrace*

- Algorithmus von Peter Selinger, 2003
- Quellen
  - http://potrace.sourceforge.net/
  - Artikel mit Erläuterung des Algorithmus: http://potrace.sourceforge.net/potrace.pdf
- Implementierung in C
  - Optimiert auf Geschwindigkeit

Überblick / Verarbeitungsschritte

- Kontur des Rasters finden
- Kontur durch Polygon approximieren
  - Optimales Polygon finden
  - Eckpunkte anpassen
- Glätten des Polygons durch Bezier-Kurven
- Ecken finden

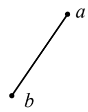


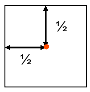
Abstand

- Abstand (Maximums-Norm)

$$a = \begin{pmatrix} a_x \\ a_y \end{pmatrix}, \quad b = \begin{pmatrix} b_x \\ b_y \end{pmatrix}$$

$$d(a,b) = \max(|a_x - b_x|, |a_y - b_y|)$$



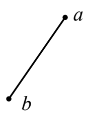


Straight Path

- Gerade *approximiert* Pfad ⇔
  - Gerade durch $a,b \rightarrow \overline{ab}$
  - Pfad $p = \{v_0,\ldots,v_n\}$
  - Gerade approximiert Pfad, wenn

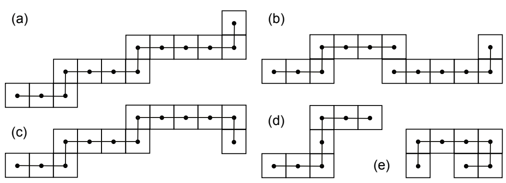
$$d(v_0,a) \le \tfrac{1}{2}, \quad d(v_n,b) \le \tfrac{1}{2}$$

$$\forall i \in \{0,\ldots,n\} \ \ \exists w \in \overline{ab} : d(v_i,w) \le \tfrac{1}{2}$$

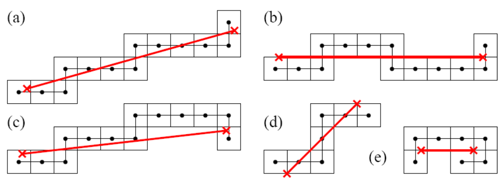


- Ein Pfad *p* heißt *straight* ⇔
  - Es gibt eine Gerade, die *p* approximiert
  - *p* enthält weniger als 4 Richtungen

Welche Pfade sind *straight pathes*?



Welche Pfade sind *straight pathes*?



Beispiele von *straight pathes*

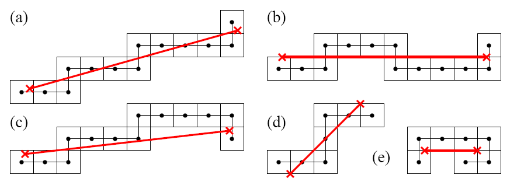


Figure 4: Examples of straight and non-straight paths. The vertices of the path are shown as dots, and their 1/2-neighborhoods are shown as squares. (a), (b), and (d) are straight, whereas (c) and (e) are not.

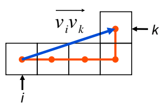
Suche aller *straight pathes*

- Satz:

  Abstand der Geraden durch v_i und v_k zum Punkt v_j kleiner gleich 1

  $$p = \{v_0,\dots,v_n\} \ \text{straight} \quad \Leftrightarrow$$
  $$\forall(i,j,k),\ 0 \le i < j < k \le n \ \ \exists w \in \overline{v_i v_k} : d(v_j, w) \le 1$$
- Komplexität
  - Naiv: *O(n³)*
  - Implementation: *O(n²)*
    - Für jedes i: Berechne die maximale Position k, bis zu der ein *straight path* von i nach k gezogen werden kann.
    - Verwende dazu zwei *constaints*, die die möglichen Pfadpunkte einschränken

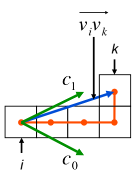
Suche nach maximalem *straight path*



- Sei i ein fester Startpunkt
- Seien c_0 und c_1 zwei mit $(0,0)$ initialisierte Vektoren (*constraints*)
- Für $k = i+1, i+2, \dots$
  - Falls mehr als 3 Richtungen → Abbruch
  - Berechne den Vektor $\overrightarrow{v_i v_k} = \begin{pmatrix} v_{kx} - v_{ix} \\ v_{ky} - v_{iy} \end{pmatrix}$
  - Falls $\overrightarrow{v_i v_k}$ den *constraint* verletzt → Abbruch
  - Aktualisiere den *constraint*
- Notiere $\mathrm{pivot}[i] = k$ als ersten Index, der den *straight path* beendet.

Definition: *constraint* verletzen

- Kreuzprodukt im R^2 $\quad a = \begin{pmatrix} a_x \\ a_y \end{pmatrix},\ b = \begin{pmatrix} b_x \\ b_y \end{pmatrix},\ \ a \times b = a_x \cdot b_y - a_y \cdot b_x$
- Der *constraint* ist verletzt, falls
  - $c_0 \times \overrightarrow{v_i v_k} < 0$ oder $c_1 \times \overrightarrow{v_i v_k} > 0$
- Bedeutung
  - Seite, auf der $\overrightarrow{v_i v_k}$ relativ zu c_1 bzw. c_0 liegt.



Constaint aktualisieren

- Sei $a = \overrightarrow{v_i v_k}$
- Falls $|a_x| \le 1$ und $|a_y| \le 1$
  - Lasse den *constraint* unverändert
- Ansonsten berechne c_0 und c_1 neu
  - Siehe nächste Folien

Constaint aktualisieren: c_0

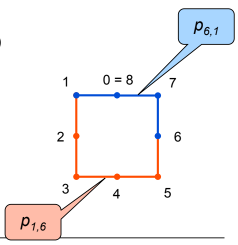
- Falls $a_y \geq 0$ und $\left(a_y > 0 \text{ oder } a_x < 0\right)$
  $d_x = a_x + 1$ ansonsten $d_x = a_x - 1$
- Falls $a_x \leq 0$ und $\left(a_x < 0 \text{ oder } a_y < 0\right)$
  $d_y = a_y + 1$ ansonsten $d_y = a_y - 1$
- Falls $c_0 \times d \geq 0$
  setze $c_0 = d$ ansonsten lasse c_0 unverändert

Constaint aktualisieren: c_1

- Falls $a_y \leq 0$ und $\left(a_y < 0 \text{ oder } a_x < 0\right)$
  $d_x = a_x + 1$ ansonsten $d_x = a_x - 1$
- Falls $a_x \geq 0$ und $\left(a_x > 0 \text{ oder } a_y < 0\right)$
  $d_y = a_y + 1$ ansonsten $d_y = a_y - 1$
- Falls $c_1 \times d \leq 0$
  setze $c_1 = d$ ansonsten lasse c_1 unverändert

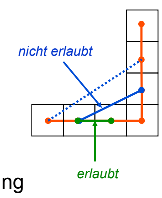
Teilpfad (*subpath*)

- Geschlossener Pfad *p* und Teilpfad $p_{i,j}$:
  $p = \{v_0, \ldots, v_n\} \quad v_0 = v_n$
  $\forall i, j \in \{0, \ldots, n-1\}$:
  $$p_{i,j} = \begin{cases} \{v_i, \ldots, v_j\} & (i \leq j) \\ \{v_i, \ldots, v_{n-1}, v_0, \ldots, v_j\} & (j < i) \end{cases}$$
- Zyklische Differenz
  $$j \dot{-} i = \begin{cases} j - i & (i \leq j) \\ j - i + n & (j < i) \end{cases}$$
  - Im folgenden einfach „-"



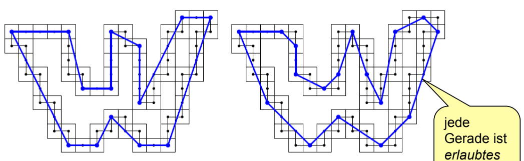
Erlaubtes Segment (*possible segment*)

- Geschlossener Pfad $p = \{v_0, \ldots, v_n\} \quad v_0 = v_n$
- Es existiert ein *erlaubtes* Segment von *i* nach *j*
  $:\Leftrightarrow \; j \dot{-} i \leq n - 3$ und $p_{i-1, j+1}$ straight
- Bedeutung:
  - Ein Segment ist *erlaubt*, wenn der an den Enden um 1 Punkt erweiterte Pfad noch *straight* ist.
  - Erhöht die Qualität der Vektorisierung an den Ecken



Polygon

- Geschlossener Pfad $p = \{v_0, \ldots, v_n\} \quad v_0 = v_n$
- Polygon ist Sequenz von Indizes $i_0 < i_1 < \ldots < i_{m-1}$ so dass $\forall k = 0, \ldots, m-2$ ein *erlaubtes* Segment von i_k nach i_{k+1} und von i_{m-1} nach i_0 existiert.



Frage: Welches Polygon ist „besser"? Wie kann ich entscheiden?

Optimales Polygon (1/2)

- Definiere *penalty* (Handicap) für ein *erlaubtes* Segment von v_i nach v_j
  - Länge des Segments mal der Standardabweichung der euklidischen Abstände aller Punkte des Pfades zur Geraden $v_i v_j$
- $P_{i,j} = \sqrt{cx^2 + 2bxy + ay^2}$ mit
  $$\begin{pmatrix} x \\ y \end{pmatrix} = v_j - v_i \text{ und } \begin{pmatrix} \bar{x} \\ \bar{y} \end{pmatrix} = (v_i + v_j)/2 \qquad E(x_k^2) = \frac{1}{j \dot{-} i + 1} \sum_{k=i}^{j} (v_k)_x^2$$
  $$a = E(x_k^2) - 2\bar{x}E(x_k) + \bar{x}^2$$
  $$b = E(x_k y_k) - \bar{x}E(x_k) - \bar{y}E(y_k) + \bar{x}\bar{y}$$
  $$c = E(y_k^2) - 2\bar{y}E(y_k) + \bar{y}^2$$

Summen von k=0 bis j vorneweg berechnen und in Tabelle speichern.
Summe von k=i bis j ist dann Differenz zweier Summen.

Optimales Polygon (2/2)

- Für zwei Polygone *p*, *p'* mit Anzahl der Segmente *k* und *k'* und der Summe der *penalties* aller Segmente *P* und *P'* gilt:
  - *p* ist besser als *p'* genau dann, wenn $k < k'$
    oder falls $k = k'$, dann wenn $P < P'$
- Mit anderen Worten:
  Lexikographische Ordnung in (*k*,*P*)

Implementation (1/2)

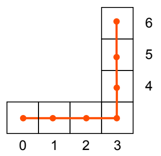
1. Pfad im Sinne von *potrace* berechnen
2. Für jeden Index i den maximalen Index k berechnen, bis zu dem ein *erlaubtes* Segment möglich ist.
3. Alle möglichen Polygone berechnen das optimale merken:
   - Erstes erlaubtes Segment von i = 0 nach j = 1,2,...
     - Nächstes erlaubtes Segment von j nach k = j+1, j+2, ...
       - Usw. solange bis Polygon geschlossen
       - Entscheiden, ob dieses Polygon besser als gemerktes Optimum
   - Wiederholen für jeden weiteren Startwert 1, 2, 3, ...

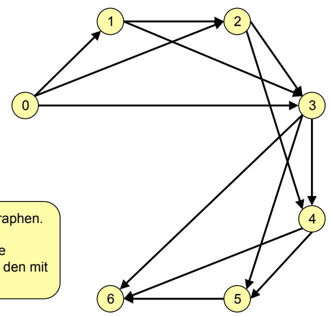
Problem: Finden eines optimalen Zyklus in einem gerichteten Graphen

Implementation (2/2)

- Entscheidung, ob aktuelles Polygon besser als gemerktes Optimum:
  - Aktuelles hat mehr Segmente als Optimum:
    - → verwerfen und weiter
  - Aktuelles hat weniger Segmente als Optimum:
    - → Aktuelles wird neues Optimum
    - → Penalty berechnen und merken
  - Gleiche Anzahl Segmente:
    - → Penalty berechnen und mit Penalty des Optimums vergleichen.
      - Penalty kleiner: → wird neues Optimum
      - Sonst: → verwerfen und weiter

Gerichteter Graph





Suche kürzesten Weg im Graphen.

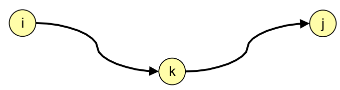
Gibt es mehrere gleich lange kürzeste Wege, dann wähle den mit der geringsten *penalty*.

Algorithmen

- Ziel:
  - Kürzesten Weg im gewichteten Graphen finden
- Algorithmen (Auswahl):
  - Dijkstra-Algorithmus
    - Komplexität: Ein Startpunkt / alle Endpunkte bei optimalen Datenstrukturen: $O(n * \log(n))$
    - Komplexität für alle Paare Start-/Endpunkt: $O(n^2 * \log(n))$
  - Floyd/Warshall
    - Komplexität für alle Paare Start-/Endpunkt: $O(n^3)$

Algorithmus von Floyd/Warshall

- Idee
  - Falls kürzeste Verbindung von Knoten i nach j bekannt und diese über Knoten k führt , dann
    - ist Teilstück von i nach k kürzeste Verbindung
    - ist Teilstück von k nach j kürzeste Verbindung



  - Nachteil:
    - Algorithmus findet nicht alle kürzesten Verbindungen

Initialisierung 1/2

- Initialisiere Distanzmatrix d^0
  - n Knoten, Eintrag (i,j) == Abstand von i nach j

$$d^0 = \begin{pmatrix} d^0(0,0) & \cdots & d^0(0,n-1) \\ \vdots & & \vdots \\ d^0(n-1,0) & \cdots & d^0(n-1,n-1) \end{pmatrix}$$

- In unserem Fall:
  - d^0(i,j) = 1 falls Verbindung von i nach j möglich
  - ansonsten d^0(i,j) = ∞ (unendlich)

Initialisierung 2/2

- Initialisiere Vorgängermatrix p^0
  - Eintrag (i,j) == Vorgänger von j auf dem Weg von i nach j

$$p^0 = \begin{pmatrix} p^0(0,0) & \cdots & p^0(0,n-1) \\ \vdots & & \vdots \\ p^0(n-1,0) & \cdots & p^0(n-1,n-1) \end{pmatrix}$$

$$p^0(i,j) = \begin{cases} \text{null} & \text{falls } i = j \text{ oder } d^0(i,j) = \infty \\ i & \text{falls } i \neq j \text{ und } d^0(i,j) < \infty \end{cases}$$

Iteration

- Für k = 0 bis n-1 iteriere
  - Für alle Paare (i,j):

$$d^{k+1}(i,j) = \min\left(d^k(i,j), d^k(i,k) + d^k(k,j)\right)$$

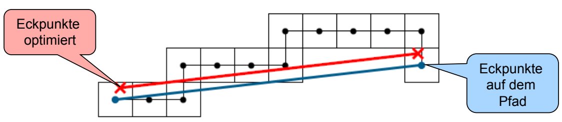
$$p^{k+1}(i,j) = \begin{cases} p^k(i,j) & \text{falls } d^k(i,j) \leq d^k(i,k) + d^k(k,j) \\ p^k(k,j) & \text{falls } d^k(i,j) > d^k(i,k) + d^k(k,j) \end{cases}$$

Ergebnis ablesen

- Suche Einträge in d^n(i,i) mit kleinstem Wert:
  - Beachte: Unser Weg ist zyklisch, d.h. Startpunkt gleich Endpunkt
  - Es könnte mehrere Einträge d^n(i,i) mit gleichem kleinsten Wert geben
    → Weg mit kleinster Penalty wählen
- Weg von i nach i konstruieren:
  - Endpunkt ist i
  - Vorgänger ist k = p^n(i,i)
  - Vorgänger vom Vorgänger ist l = p^n (i,k)
  - usw. bis man beim Startpunkt i ankommt.

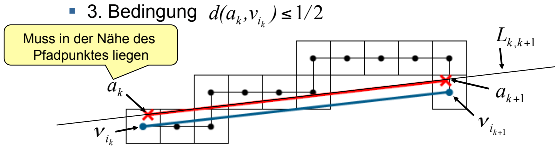
Anpassung der Eckpunkte

- Bisher:
  - Eckpunkte des Polygons liegen auf ursprünglichen Pfadpunkten
- Ziel:
  - Eckpunkte so anpassen, dass die Quadrate der euklidischen Abstände zwischen Segement und Pfadpunkten minimiert werden



Anpassung der Eckpunkte

- Geschlossener Pfad $p = \{v_0, \dots, v_n\}$ $v_0 = v_n$
- Indizes der Eckpunkte des Polygons $i_0, \dots, i_{m-1}$
- Eckpunkte $\{v_{i_0}, \dots, v_{i_{m-1}}\}$
- Suche neue Punkte a_k $(k = 0, \dots, m-1)$
  - 1. Optimierte Gerade $L_{k,k+1}$ $(k = 0, \dots, m-1)$
  - 2. a_k = Schnittpunkt $L_{k-1,k}, L_{k,k+1}$
  - 3. Bedingung $d(a_k, v_{i_k}) \leq 1/2$



1. Optimierte Gerade $L_{k,k+1}$

- Betrachte alle Pfadpunkte $v_{i_k},\dots,v_{i_{k+1}}$
- Ziel:
  - Wähle Gerade $L_{k,k+1}$ so, dass Summe der Abstandsquadrate zu den betrachteten Punkten minimal ist
- Berechne Schwerpunkt $(E(x_j),E(y_j))$ mit $j=i_k,\dots,i_{k+1}$
  - Definition des Erwartungswerts E() siehe Folie Optimales Polygon (1/2)
- Betrachte Matrix $\begin{pmatrix} a & b \\ b & c \end{pmatrix}$ mit $a=E(x_j^2)-E(x_j)^2$

  $b=E(x_j y_j)-E(x_j)E(y_j)$

  $c=E(y_j^2)-E(y_j)^2$
- Berechne Eigenwerte und Eigenvektoren

Eigenwerte

- Berechnung über das charakteristische Polynom

$$A=\begin{pmatrix} a & b \\ b & c \end{pmatrix}$$

$$0=\det(A-\lambda 1\!\!1)$$

$$0=\det\begin{pmatrix} a-\lambda & b \\ b & c-\lambda \end{pmatrix}=(a-\lambda)(c-\lambda)-b^2=\lambda^2-(a+c)\lambda+ac-b^2$$

$$\lambda_{1,2}=(a+c\pm\sqrt{(a+c)^2-4ac+4b^2})/2$$

$$\lambda_1=(a+c+\sqrt{(a-c)^2+4b^2})/2$$

Uns interessiert nur der größere Eigenwert

Eigenvektor zum größeren Eigenwert

Hier berechneter Eigenvektor ist auf Länge 1 normiert

$$Az=\lambda_1 z \quad \text{mit} \quad z=\begin{pmatrix} x \\ y \end{pmatrix}$$

$$\begin{pmatrix} 0 \\ 0 \end{pmatrix}=\begin{pmatrix} a-\lambda_1 & b \\ b & c-\lambda_1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix}$$

$$0=(a-\lambda_1)x+by$$

$$0=bx+(c-\lambda_1)y$$

Falls $|a-\lambda_1|\ge|c-\lambda_1|$:

$$l=\sqrt{(a-\lambda_1)^2+b^2}$$

$$\text{Falls } l\ne 0:\begin{pmatrix} x \\ y \end{pmatrix}=\begin{pmatrix} -b/l \\ (a-\lambda_1)/l \end{pmatrix}$$

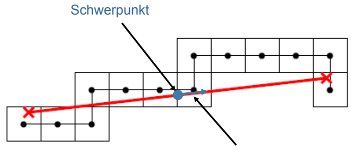
ansonsten:

$$l=\sqrt{(c-\lambda_1)^2+b^2}$$

$$\text{Falls } l\ne 0:\begin{pmatrix} x \\ y \end{pmatrix}=\begin{pmatrix} -(c-\lambda_1)/l \\ b/l \end{pmatrix}$$

1. Optimierte Gerade $L_{k,k+1}$

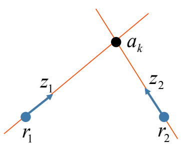
- $L_{k,k+1}$ ist gegeben durch
  - Den Schwerpunkt $(E(x_j),E(y_j))$
  - Und die Richtung des Eigenvektors $z=\begin{pmatrix} x \\ y \end{pmatrix}$



2. Schnittpunkt von $L_{k-1,k}$ und $L_{k,k+1}$

- Berechne a_k als Schnittpunkt von $L_{k-1,k},L_{k,k+1}$

$$a_k=r_1+tz_1=r_2+sz_2$$



$$\begin{pmatrix} p_1 \\ q_1 \end{pmatrix}+t\begin{pmatrix} x_1 \\ y_1 \end{pmatrix}=\begin{pmatrix} p_2 \\ q_2 \end{pmatrix}+s\begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$$

Nach *t* und *s* auflösen:
Nächste Folien

$$r_1=\begin{pmatrix} p_1 \\ q_1 \end{pmatrix},\ z_1=\begin{pmatrix} x_1 \\ y_1 \end{pmatrix},\ r_2=\begin{pmatrix} p_2 \\ q_2 \end{pmatrix},\ z_2=\begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$$

Schnittpunkt ausrechnen (1/2)

$$p_1+x_1t=p_2+x_2s \qquad (1)$$

$$q_1+y_1t=q_2+y_2s \qquad (2)$$

Falls $|y_1x_2-x_1y_2|\ge|y_2x_1-x_2y_1|>0$:

$$y_1(1):\quad y_1p_1+y_1x_1t=y_1p_2+y_1x_2s \qquad (1')$$

$$x_1(2):\quad x_1q_1+x_1y_1t=x_1q_2+x_1y_2s \qquad (2')$$

$$(1')-(2'):\quad y_1p_1-x_1q_1=y_1p_2-x_1q_2+(y_1x_2-x_1y_2)s$$

$$\Leftrightarrow \quad s=\frac{y_1(p_1-p_2)+x_1(q_2-q_1)}{y_1x_2-x_1y_2}$$

$a_k=r_2+sz_2$ berechnen

Schnittpunkt ausrechnen (2/2)

$$p_1 + x_1 t = p_2 + x_2 s \qquad (1)$$
$$q_1 + y_1 t = q_2 + y_2 s \qquad (2)$$

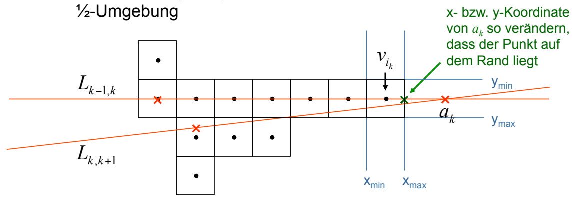
Ansonsten falls $|y_2 x_1 - x_2 y_1| > 0$:

$$y_2(1): \quad y_2 p_1 + y_2 x_1 t = y_2 p_2 + y_2 x_2 s \qquad (1'')$$
$$x_2(2): \quad x_2 q_1 + x_2 y_1 t = x_2 q_2 + x_2 y_2 s \qquad (2'')$$
$$(1'')-(2''): \quad y_2 p_1 - x_2 q_1 + (y_2 x_1 - x_2 y_1) t = y_2 p_2 - x_2 q_2$$
$$\Leftrightarrow \quad t = \frac{y_2(p_2 - p_1) + x_2(q_1 - q_2)}{y_2 x_1 - x_2 y_1}$$

$a_k = r_1 + t z_1$ brechnen

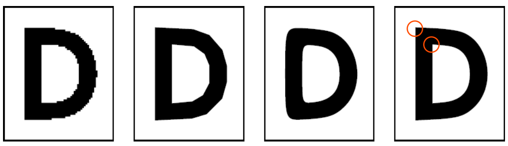
3. Bedingung $d(a_k, v_{i_k}) \leq 1/2$

- Falls Schnittpunkt a_k nicht in ½-Umgebung von v_{i_k}
  - Potrace sucht Punkt in ½-Umgebung so, dass die Summe der Quadrate der Abstände zu den Linien $L_{k-1,k}$ und $L_{k,k+1}$ minimal ist → Minimierung einer Quadratischen Form
  - Vereinfachung: Projiziere den Punkt auf den Rand der ½-Umgebung



Glättung durch Bezier-Kurven

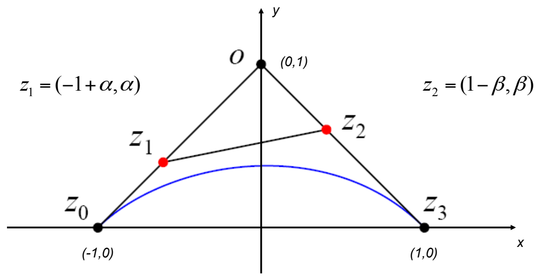
- Ziel:
  - Polygon durch eine glatte Kurve ersetzen
- Problematik:
  - Nicht an allen Stellen glätten
  - Manche Punkte besser als Ecken erhalten



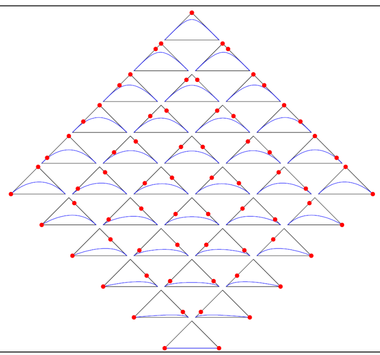
Raster | Polygon | Bezier | Ecken erhalten

Kubische Bezierkurven

- Gegeben durch 4 Kontrollpunkte $z_0,\ldots,z_3$



Eine Familie von Bezierkurven



Umsetzung in SVG

- Vollständige Beispiele siehe:
  - Dateien *bezier0.svg* bis *bezier3.svg*
- Ecke mit 2 geraden Linien
  - z_0=(50, 250), O=(250, 50), z_3=(450, 250)
  - `<path stroke="black" fill="none" stroke-width="1" d="M 50 250 L 250 50 L 450 250" />`
- Ecke mit Bezierkurze
  - z_0=(50, 250), z_1=(150, 150), z_2=(300, 100), z_3=(450, 250)
  - `<path stroke="blue" fill="none" stroke-width="1" d="M 50 250 C 150 150 300 100 450 250" />`

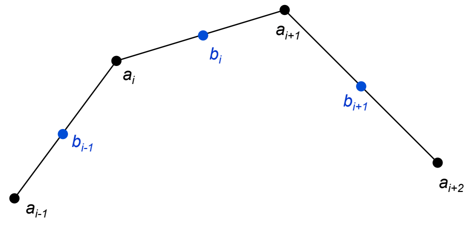
M = move to / L = line to / C = curve to
Großbuchstaben: absolute Koordinaten / Kleinbuchstaben = relative Koordinaten

Glätten des Polygons

- Vereinfachung gegenüber *potrace*
  - Überspringen des Schritts Anpassung der Eckpunkte (Folie Nr. 29 und folgende)
    - Polygonpunkte sind weiterhin ausgewählte Pfadpunkte
  - Vereinfachte der Berechnung von α
    - Betrachte runde statt quadratischer Umgebung

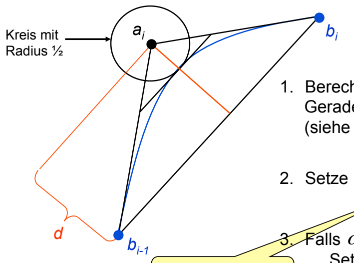
1. Mittelpunkte finden

- Seien a_i die Eckpunkte des Polygons
- Definiere b_i als Mittelpunkt zwischen a_i und a_{i+1}



2. Alpha für Ecke bestimmen

- Betrachte eine Ecke

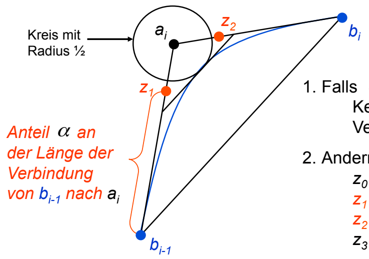


1. Berechne Abstand d von a_i zur Geraden durch b_{i-1} und b_i (siehe Folie 48)
2. Setze $\alpha = \frac{4}{3} \cdot \frac{d - 0.5}{d}$
3. Falls $\alpha < 0.55$ *(Minimum)* Setze $\alpha = 0.55$

Faktor 4/3 einstellbar machen!

Minimum 0.55 einstellbar machen!

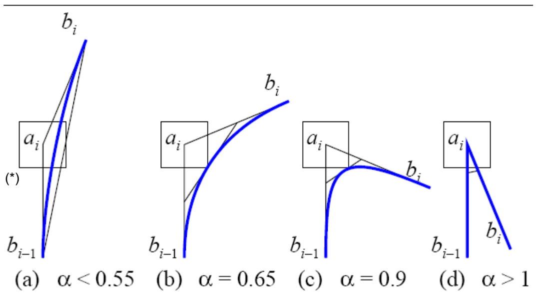
3. Ecke hinzufügen



Maximum 1.0 einstellbar machen!

1. Falls $\alpha > 1.0$ *(Maximum)* Keine Glättung: gerade Verbindung b_{i-1} → a_i → b_i
2. Andernfalls Bezierkurve mit
   $z_0 = b_{i-1}$
   z_1 auf Linie $b_{i-1} a_i$ mit Anteil α
   z_2 auf Linie $a_i b_i$ mit Anteil α
   $z_3 = b_i$

Beispielsituationen



(*) Umgebung quadratisch statt rund

Abstand Punkt zu Gerade

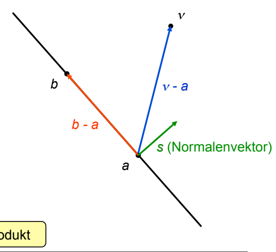
- Gegeben:
  - Gerade durch Punkte *a* und *b*
  - Punkt v

$$a = \begin{pmatrix} a_x \\ a_y \end{pmatrix}, \quad b = \begin{pmatrix} b_x \\ b_y \end{pmatrix}$$

$$\hat{s} = \begin{pmatrix} b_y - a_y \\ -(b_x - a_x) \end{pmatrix}, \quad s = \frac{\hat{s}}{|\hat{s}|}$$

$$d(v, \overline{ab}) = |< s, v - a >|$$

<·,·> Skalarprodukt



Umsetzung in SVG

- Einen langen Pfad berechnen
  - Entweder mit `C` eine Bezierkurve oder
  - mit `L` zwei gerade Linienstücke (Ecke) hinzufügen
- Füllfarbe angeben, z.B.
  - `fill="black"` für äußere Konturen und `fill="white"` für innere Konturen
- Beispiel
  - ```
    <path fill="black"
            d="M 100 150
            C 250 0 350 0 500 150
            L 550 200
            L 500 250
            C 350 400 250 400 100 250
            L  50 200
            L 100 150" />
    ```
- Siehe Datei *bezier_closed.svg*
    ```