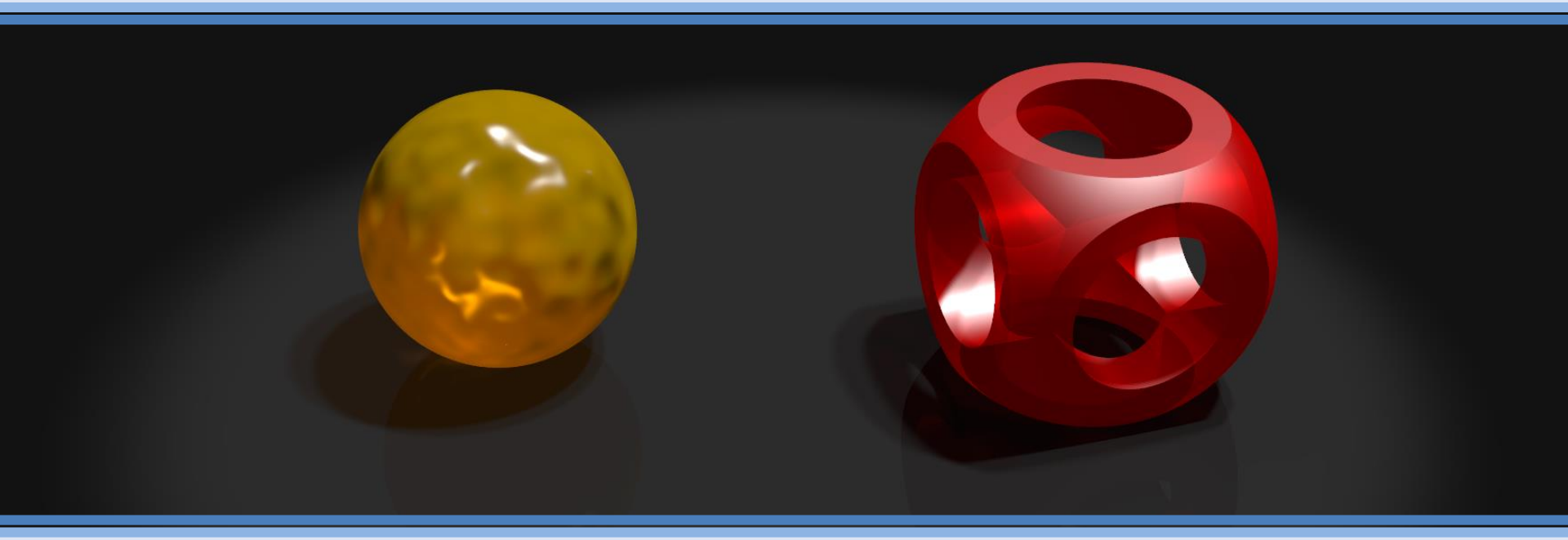
# Computergrafik

T. Hopp



# Themenübersicht

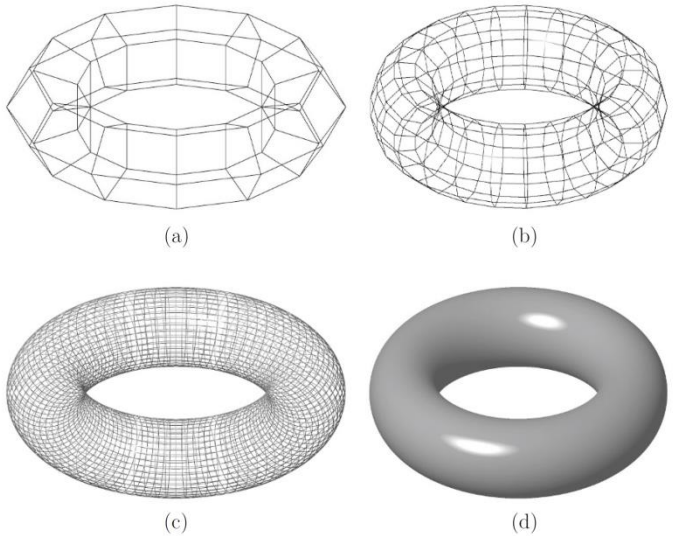
1. Einführung
2. Programmierbibliotheken / OpenGL
3. **Geometrische Repräsentation von Objekten**
4. Koordinatensysteme und Transformationen
5. Zeichenalgorithmen
6. Buffer-Konzepte
7. Farbe, Beleuchtung und Schattierung
8. Texturen
9. Animationen
10. Raytracing
11. Volumenvisualisierung

# Einordnung

- Nachbildung von (realen) Objekten durch ***abstrakte*** Objekte
- Meist nur Abbildung der opaken Oberflächen eines Objektes
- Typische Fragestellungen:
  - Direkte Repräsentation von einfachen geometrischen Objekten ⇔ Annäherung an komplexe geometrische Objekte aus mehreren einfachen geometrischen Objekten
  - Exakte Beschreibung ⇔ ausreichende Approximation zur Darstellung

# Planare Polygone

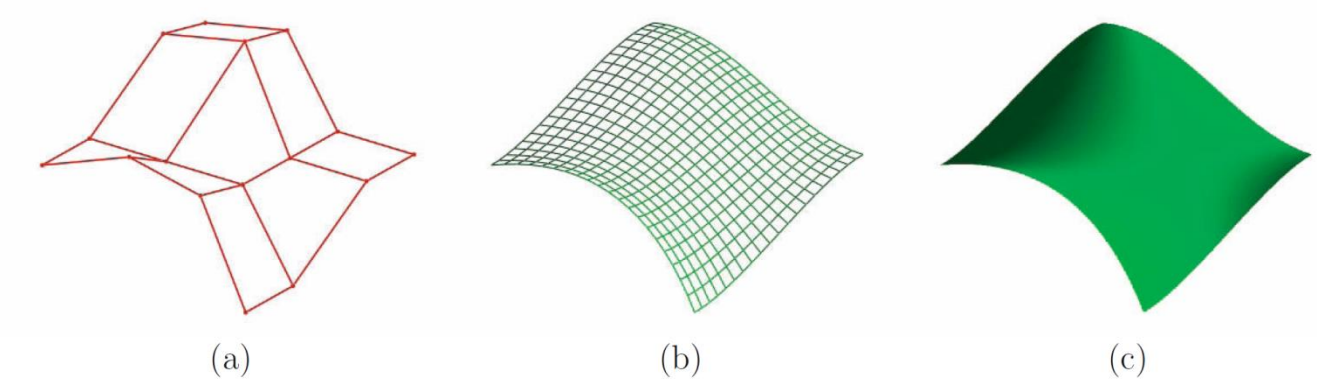
- Oberflächennetze aus planaren Polygonen am häufigsten zur Approximation von Objekten eingesetzt (→ Kapitel 3.2)
- Grundobjekte meist (→ Kapitel 3.1)
  - Dreiecke (*triangles*)
  - Viereck (*quads*)
- Vorteil: schnell berechenbar
- Nachteil: Genauigkeit der Approximation abhängig von Polygonauflösung




A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

# Gekrümmte Flächen

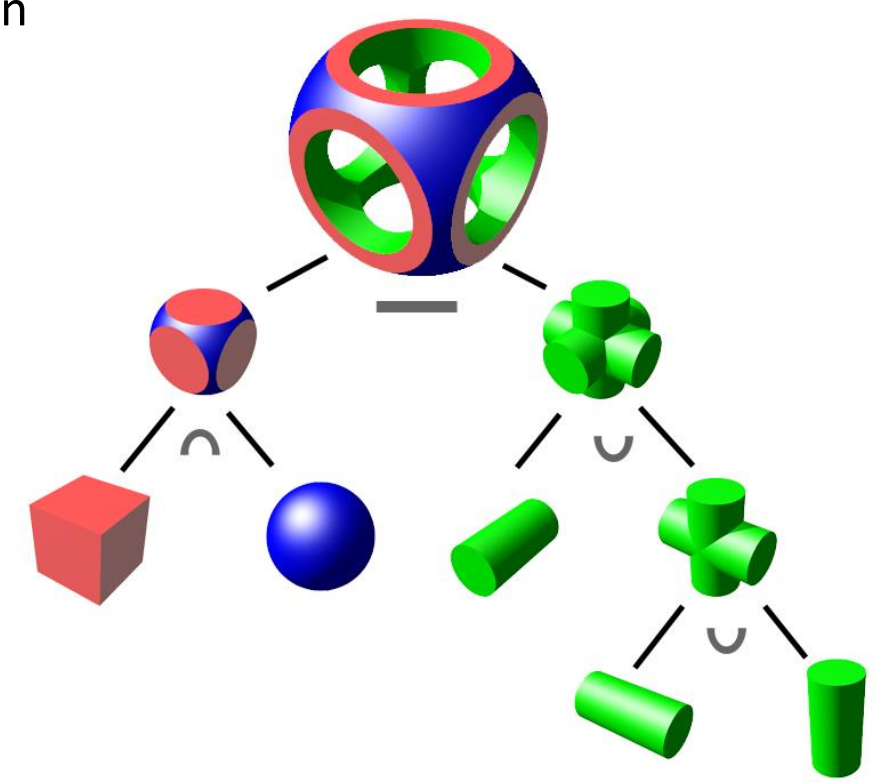
- Räumlich gekrümmte Flächen zur Approximation einer Oberfläche (→ Kapitel 3.3)
  - Bézier-Flächen, B-Splines, NURBS
- Kontrollpunkte = Parametrisierung der Fläche
- Vorteil: wenig Speicherplatz, exaktere Approximation möglich
- Nachteil: rechenaufwändigere Operationen




A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

# Konstruktive Körpergeometrie

- Zusammensetzen komplexer Objekte aus elementaren Körpern (CSG = Constructive Solid Geometry)
  - Bool‘sche Operationen oder lineare Transformationen zur Vereinigung von elementaren Objekten
- Baumstruktur
  - Blätter = Grundobjekte
  - Knoten = Kombinationen

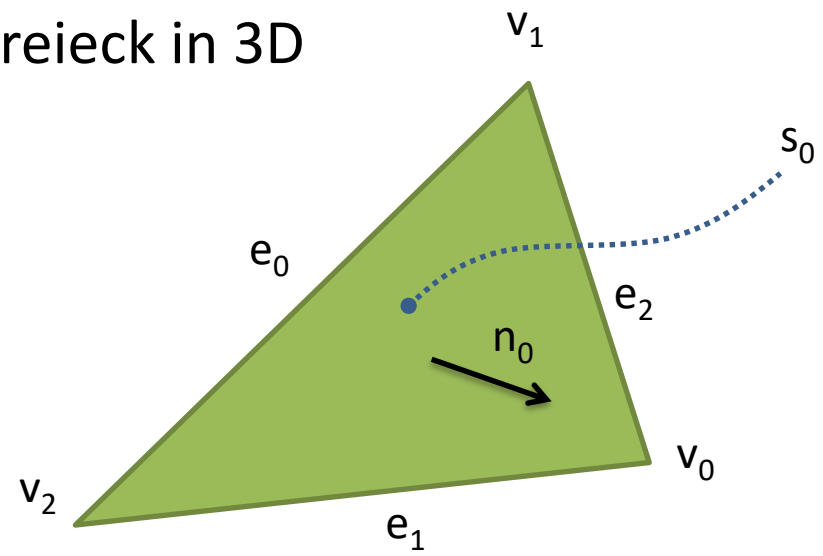


https://de.wikipedia.org/wiki/Geometrische_Modellierung

# 3.1. GRUNDKÖRPER UND PLANARE POLYGONE

# Beschreibung eines Körpers

- Die Geometrie eines Körpers wird in der CG beschrieben durch
  - Punkte (*Vertices*)
  - Kanten (*Edges*)
  - Flächen (*Surfaces*)
  - Normalen
- Beispiel: Dreieck in 3D




$$v_0 = (v_{0,x}, v_{0,y}, v_{0,z})$$

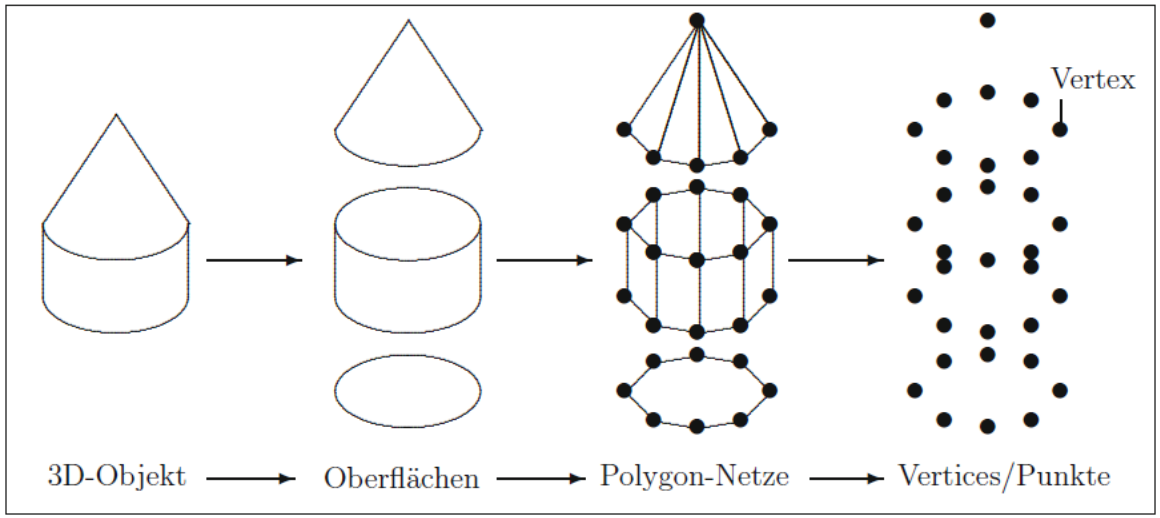
# Geometrische Grundobjekte in OpenGL

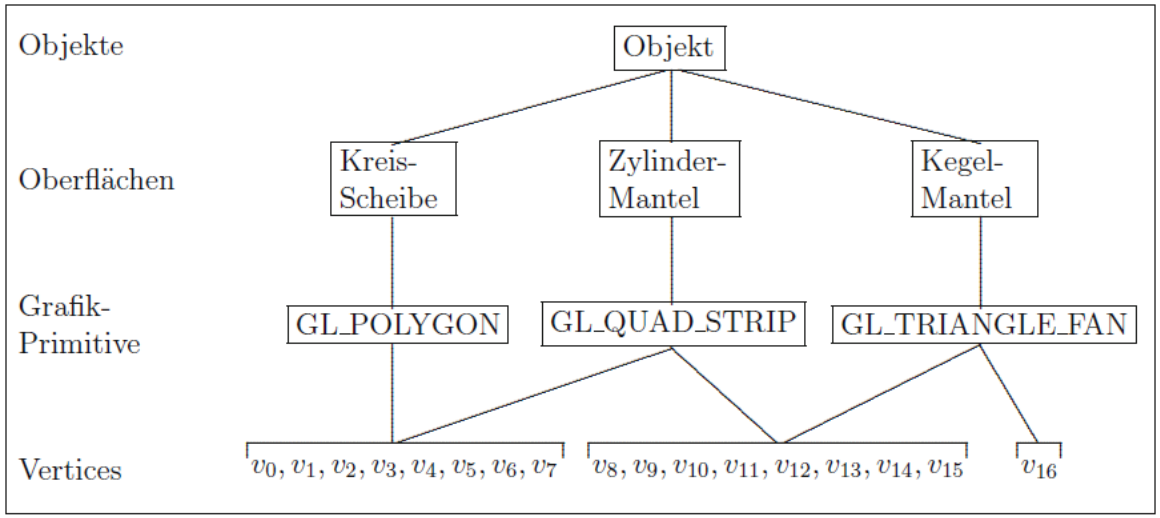
- In OpenGL: Beschreibung aller Objekte durch Vertices, Kanten und Flächen (planare Polygone)
  - Geordneter Satz von Vertices
  - Verbindung der Vertices durch Kanten
  - Eine bzw. Verbindung mehrerer Kanten ergeben Objekte bzw. Flächen

  ➔ **Polygonnetze**

- Definition komplexer Oberflächen durch Grundobjekten (OpenGL Grafik-Primitive)

# Geometrische Objekte in OpenGL




A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

# Vertex-Definition in OpenGL

- „Vertex Array“-Methoden

| Skalarform | Vektor-Form |
|---|---|
| `glVertex2f(x,y)` | `glVertex2fv(vec)` |
| `glVertex2d(x,y)` | `glVertex2dv(vec)` |
| `glVertex2s(x,y)` | `glVertex2sv(vec)` |
| `glVertex2i(x,y)` | `glVertex2iv(vec)` |
| `glVertex3f(x,y,z)` | `glVertex3fv(vec)` |
| `glVertex3d(x,y,z)` | `glVertex3dv(vec)` |
| `glVertex3s(x,y,z)` | `glVertex3sv(vec)` |
| `glVertex3i(x,y,z)` | `glVertex3iv(vec)` |
| `glVertex4f(x,y,z,w)` | `glVertex4fv(vec)` |
| `glVertex4d(x,y,z,w)` | `glVertex4dv(vec)` |
| `glVertex4s(x,y,z,w)` | `glVertex4sv(vec)` |
| `glVertex4i(x,y,z,w)` | `glVertex4iv(vec)` |

← 2D: Kartesische Koordinate eines Punktes $v_0 = (x, y)$

← 3D: Kartesische Koordinate eines Punktes $v_0 = (x, y, z)$

← w = inverser Streckungsfaktor
→ Homogene Koordinaten

Kart. Koordinate eines Punktes $v_0 = (x/w, y/w, z/w)$

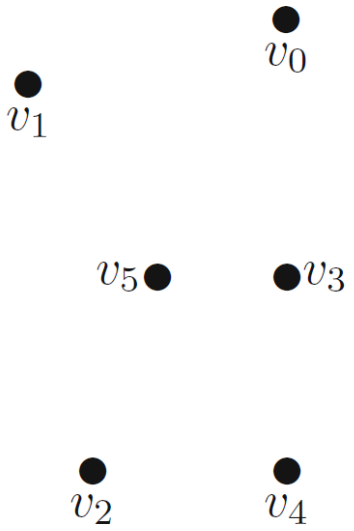
- Vektorform: Zeiger auf Array. Flexibler, schneller!

# Definition von Grafik-Primitiven in OpenGL

- Verbindung von Vertices zu Flächen, Linien, Punkten

  **glBegin / glEnd**

1) GL_POINTS
   - Für jeden Vertex wird ein Punkt gerendert




```
glBegin(GL_POINTS);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
glEnd();
```

# Punkteigenschaften in OpenGL

- Größe eines Punktes
  - *Default*: 1 Pixel
  - Änderung über Zustandsvariable: **`glPointSize(Glfloat size)`**
  - Zulässige Punktgröße ist hardwareabhängig. Abfrage mit:

```
Glfloat sizes[2];
Glfloat incr;

glGetFloatv(GL_POINT_SIZE_RANGE, sizes);
glGetFloatv(GL_POINT_SIZE_GRANULARITY, &incr);
```

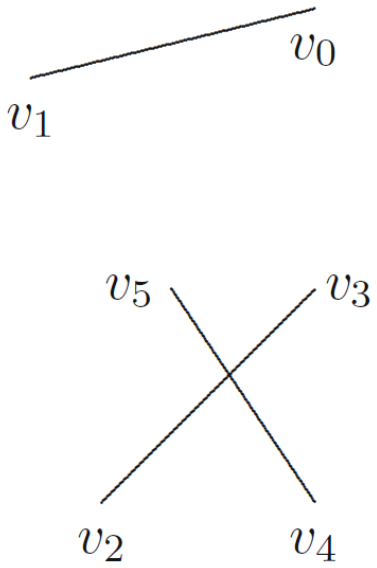
- Form eines Punktes: quadratisch
  - Runde Punkte nur durch Vortäuschung per Anti-Aliasing und Transparenzberechnung

```
glEnable(GL_POINT_SMOOTH);   glEnable(GL_BLEND);
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

# Definition von Grafik-Primitiven in OpenGL

## 2) GL_LINES

- Nicht-verbundene Liniensegmente zwischen jeweils zwei aufeinanderfolgenden Vertices



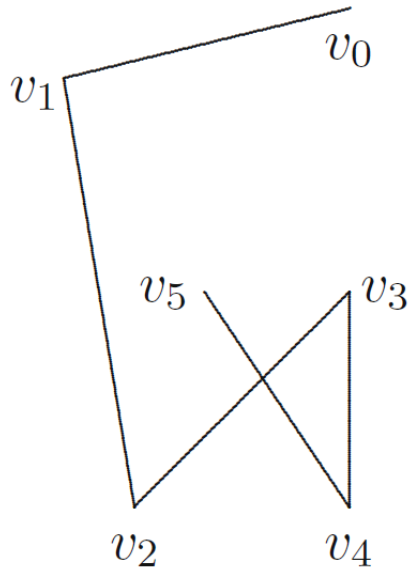

```
glBegin(GL_LINES);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
glEnd();
```

- Linienbreite analog zu Punktgröße: **`glLineWidth(Glfloat width)`**

# Definition von Grafik-Primitiven in OpenGL

## 3) GL_LINE_STRIP

- Verbundene Linien zwischen jeweils zwei aufeinanderfolgenden Vertices



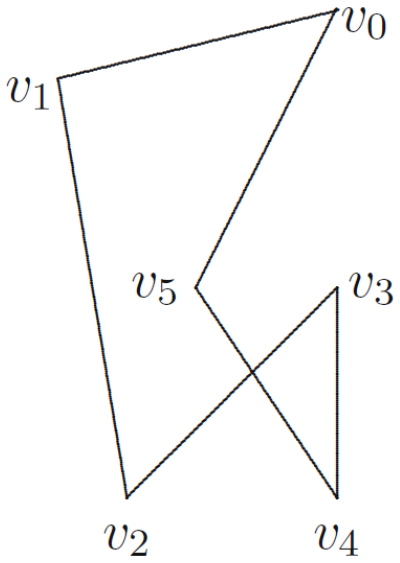

```
glBegin(GL_LINE_STRIP);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
glEnd();
```

- Keine Verbindung zwischen erstem und letztem Vertex

# Definition von Grafik-Primitiven in OpenGL

## 4) GL_LINE_LOOP

- Verbundene Linien zwischen jeweils zwei aufeinanderfolgenden Vertices
- Verbindung zwischen erstem und letztem Vertex (geschlossener Linienzug)

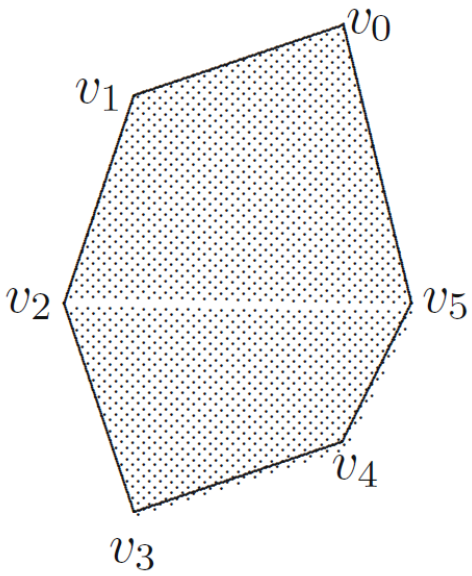



```
glBegin(GL_LINE_LOOP);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
glEnd();
```

# Definition von Grafik-Primitiven in OpenGL

## 5) GL_POLYGON

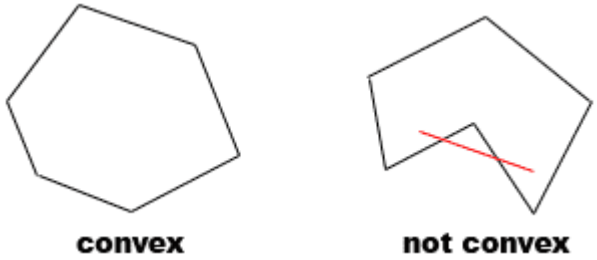
- Gefülltes konvexes Polygon
- Vieleck mit Anzahl Ecken = Anzahl Vertices
- *deprecated* – kann durch GL_TRIANGLE_FAN ersetzt werden




```
glBegin(GL_POLYGON);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
glEnd();
```

# Polygone

- Anforderung in OpenGL: konvex und planar
  - Ein Polygon ist konvex, wenn alle Linien, die zwei beliebige Punkte des Polygons verbinden, vollständig innerhalb des Polygons liegen




  - Test auf Konvexität durch Ablaufen des Umrisses. Wenn man an Vertices immer in die gleiche Richtung abbiegt, ist das Polygon konvex

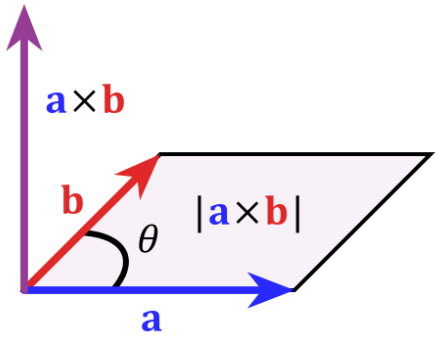
  - Planar: Alle Punkte liegen in einer Ebene

http://www.sunshine2k.de/coding/java/Polygon/Convex/polygon.htm

# Mathematische Grundlagen

## Kreuzprodukt zweier Vektoren

$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix}$$




Geometrisch: entspricht einem senkrechten Vektor zu beiden Vektoren (**=Normale**)

## Skalarprodukt zweier Vektoren

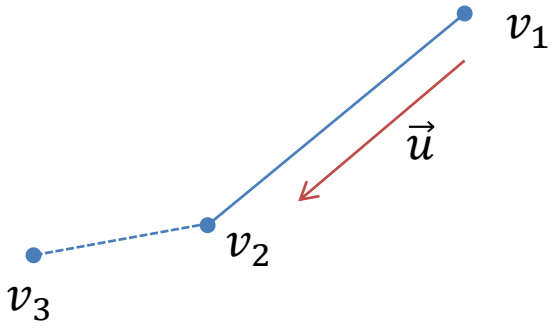
$$\begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} = a_1 b_1 + a_2 b_2 + a_3 b_3$$

Geometrisch: Kosinus des Winkels zwischen den beiden Vektoren.

- Skalarprodukt zweier Vektoren gegebener Länge = 0 wenn sie senkrecht zueinander stehen, und maximal wenn sie die gleiche Richtung haben.

# Polygone: Test auf Konvexität

- Bestimmung der „Abbiegerichtung“ (Algorithmus nach P. Bourke)




Gerade $g$ durch $v_1$ und $v_2$:

$g: \vec{w} = \vec{v}_1 + r\vec{u}$ mit $\vec{u} = \overrightarrow{v_2} - \overrightarrow{v_1}$ , $r \epsilon \mathbb{R}$

Gleichungssystem:

$$v_{3_x} = v_{1_x} + ru_x$$
$$v_{3_y} = v_{1_y} + ru_y$$

Jeweils auflösen nach $r$ und gleichsetzen ergibt:

$$(v_{3_x} - v_{1_x})u_y = (v_{3_y} - v_{1_y})u_x$$

Gleichung erfüllt wenn $v_3$ auf der Geraden liegt, d.h.

$$f(v_3) = (v_{3_x}u_y - v_{3_y}u_x) - (v_{1_x}u_y - v_{1_y}u_x)$$

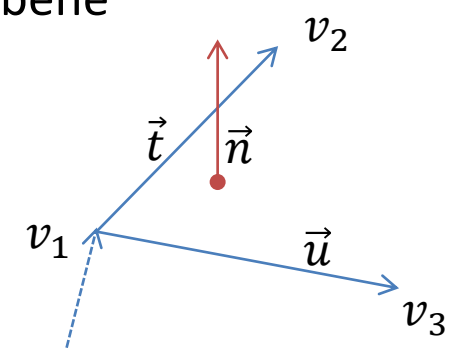
| | |
|---|---|
| $f(v_3) = 0$ | $v_3$ liegt auf der Geraden $g$ |
| $f(v_3) > 0$ | $v_3$ liegt rechts von der Geraden $g$ |
| $f(v_3) < 0$ | $v_3$ liegt links von der Geraden $g$ |

Implizite Form der Geradengleichung.
Auch 3D-Equivalent!

# Polygone: Test auf Planarität

- Drei Punkte eines Polygons definieren eine Ebene

$$E: x = \overrightarrow{v_1} + r\underbrace{(\overrightarrow{v_2} - \overrightarrow{v_1})}_{\vec{t}} + s\underbrace{(\overrightarrow{v_3} - \overrightarrow{v_1})}_{\vec{u}}$$




- Test ob alle weiteren Punkte in dieser Ebene liegen
  - Z.B.: Einsetzen des Punktes in Ebenengleichung. Bei Lösung des LGS liegt der Punkt in der Ebene
  - Z.B.: Bestimmung des Normalenvektors der Ebene durch Kreuzprodukt

$$\begin{pmatrix} n_1 \\ n_2 \\ n_3 \end{pmatrix} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \times \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} t_2 u_3 - t_3 u_2 \\ t_3 u_1 - t_1 u_3 \\ t_1 u_2 - t_2 u_1 \end{pmatrix}$$

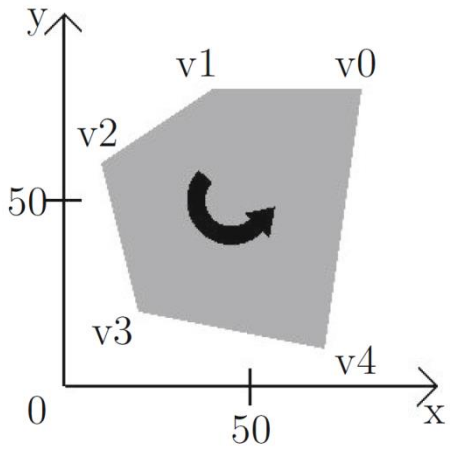
Damit $v_4$ in der Ebene liegt muss die Normale orthogonal zu $\vec{w} = (v_4 - v_1)$ sein:

$$f(v_4) = \vec{n} \cdot \overrightarrow{w} = n_1 w_1 + n_2 w_2 + n_3 w_3, \quad \text{orthogonal wenn } f(v_4) = 0$$

# Polygone: Orientierung

- Vorder- und Rückseite bestimmt durch Vertex-Reihenfolge:
  - Definition gegen den Uhrzeigersinn: Vorderseite

```
glBegin(GL_POLYGON);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
glEnd();
```



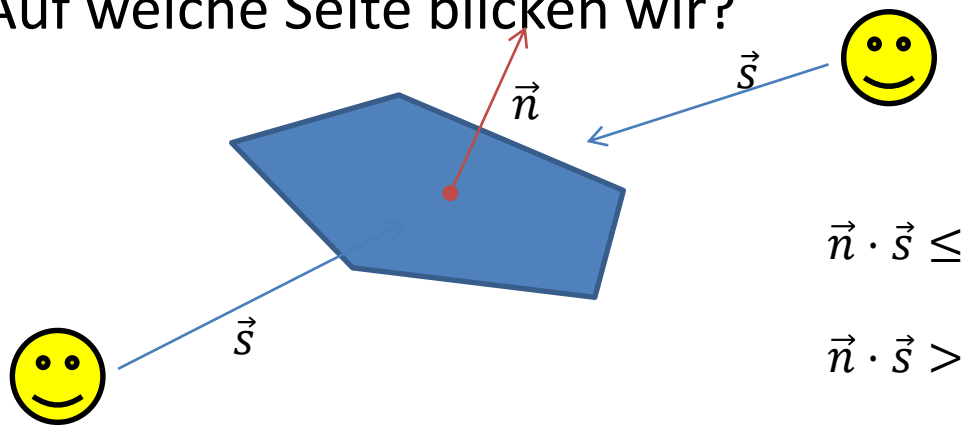

- Umdrehen der Konvention in OpenGL: **`glFrontFace(Glenum mode)`**

**`GL_CW`** **`GL_CCW`** (*default*)

A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

# Polygone: Front/Back Face Culling

- OpenGL rendert standardmäßig Vorder- und Rückseite
- Deaktivieren des Renderns von Vorder- oder Rückseite durch Culling: **`glCullFace(Glenum mode)`**
  - **`GL_FRONT`**
  - **`GL_BACK`**
  - **`GL_FRONT_AND_BACK`**
- Erhöhung der Darstellungsgeschwindigkeit
- Auf welche Seite blicken wir?




$\vec{n} \cdot \vec{s} \leq 0$ Blick auf Vorderseite

$\vec{n} \cdot \vec{s} > 0$ Blick auf Rückseite

# Polygone: Füllmodi

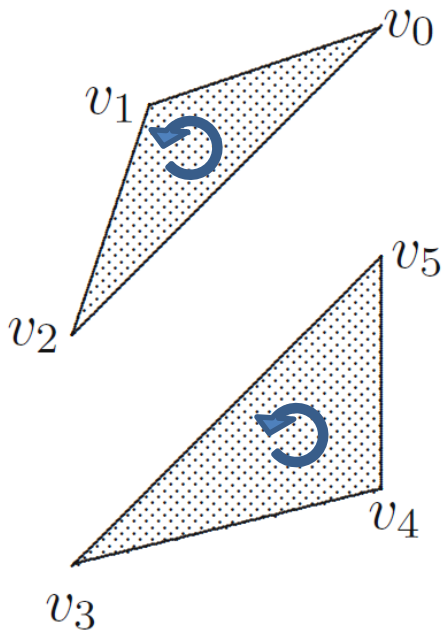
- Festlegung des Füllmodus durch

  **`glPolygonMode(GLenum face, Glenum mode)`**
- Modi:
  - Füllung der Polygonflächen: **`GL_FILL`**
  - Nur Zeichnen der Linien: **`GL_LINE`** (Drahtgittermodell)
  - Nur Zeichnen der Vertices: **`GL_POINT`**
- Für welche Polygonseite der Füllmodus gilt wird durch den **`face`**-Parameter festgelegt.

# Definition von Grafik-Primitiven in OpenGL

## 6) GL_TRIANGLES

- Nicht verbundene Dreiecke aus jeweils drei aufeinanderfolgenden Vertices
- Reihenfolge wichtig, damit Dreiecke die gleiche Orientierung bekommen




```
glBegin(GL_TRIANGLES);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
glEnd();
```

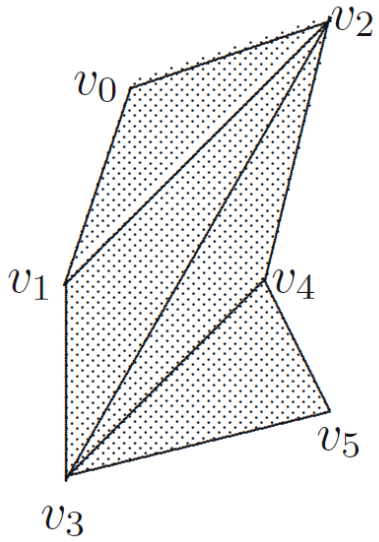
# Definition von Grafik-Primitiven in OpenGL

## 7) GL_TRIANGLE_STRIP

- Verbundene Dreiecke
- Implizite Änderung der Vertexreihenfolge beim Rendern:
  - Dreieck 1: $v_0, v_1, v_2$
  - Dreieck 2: $v_2, v_1, v_3$
  - Dreieck 3: $v_2, v_3, v_4$

Ungerade n: $[v_{n-1}, v_n, v_{n+1}]$

Gerade n: $[v_n, v_{n-1}, v_{n+1}]$




```
glBegin(GL_TRIANGLE_STRIP);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
glEnd();
```

A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

# Verbundene Dreiecke

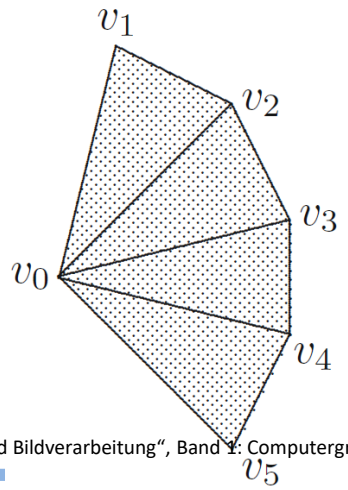
- Am häufigsten verwendete Grafik-Primitive
  - Approximation von komplexen Oberflächen:
    - Beliebig genau
    - Speichersparend
    - Am schnellsten zu zeichnen

- Konsistente Dreieckorientierung innerhalb eines Triangle-Strips durch Reihenfolge der Vertex-Verwendung.

# Definition von Grafik-Primitiven in OpenGL

## 8) GL_TRIANGLE_FAN

- Fächer aus Dreiecken. Wie GL_TRIANGLE_STRIP, nur andere Vertex-Reihenfolge:
  - Dreieck 1: $v_0, v_1, v_2$ $[v_0, v_n, v_{n+1}]$
  - Dreieck 2: $v_0, v_2, v_3$
- Fläche entspricht dem eines konvexen Polygons aller Vertices
- Oft Verwendung für runde oder kegelförmige Flächen
- Orientierung konsistent



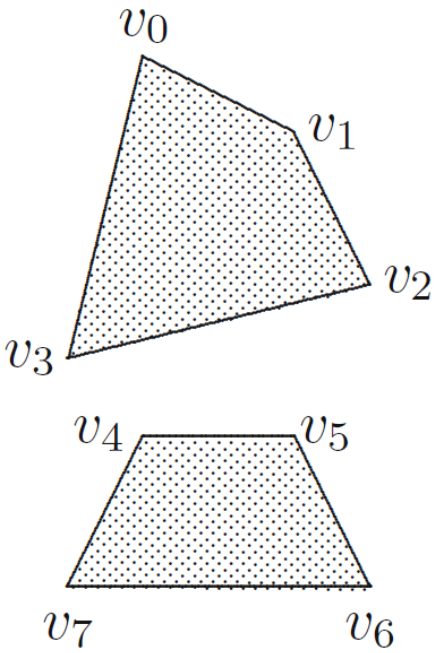

A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

```
glBegin(GL_TRIANGLE_FAN);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
glEnd();
```

# Definition von Grafik-Primitiven in OpenGL

## 9) GL_QUADS

- Einzelvierecke aus jeweils vier aufeinander folgenden Vertices
- (*deprecated* – kann durch GL_TRIANGLE_STRIP ersetzt werden*)




```
glBegin(GL_QUADS);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
   glVertex3fv(v6);
   glVertex3fv(v7);
glEnd();
```

**Reihenfolge der Vertices beachten!*

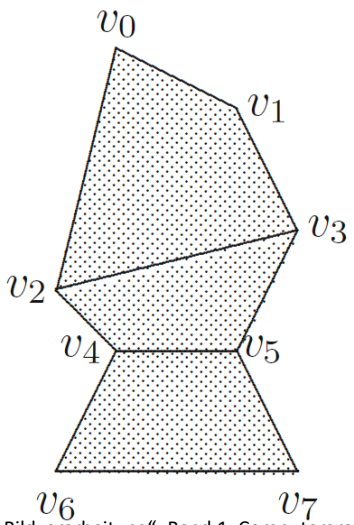
A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

# Definition von Grafik-Primitiven in OpenGL

## 10) GL_QUAD_STRIP

- Verbundene Vierecke aus jeweils vier aufeinander folgenden Vertices
- (*deprecated* – kann durch GL_TRIANGLE_STRIP ersetzt werden)
- Reihenfolge wichtig für die Orientierung der Quads
  - Viereck 1: $[v_0, v_1, v_3, v_2]$
  - Viereck 2: $[v_2, v_3, v_5, v_4]$
  - Viereck 3: $[v_4, v_5, v_7, v_6]$

Vorschrift allgemein:
$[v_{2(n-1)}, v_{2(n-1)+1}, v_{2n+1}, v_{2n}]$



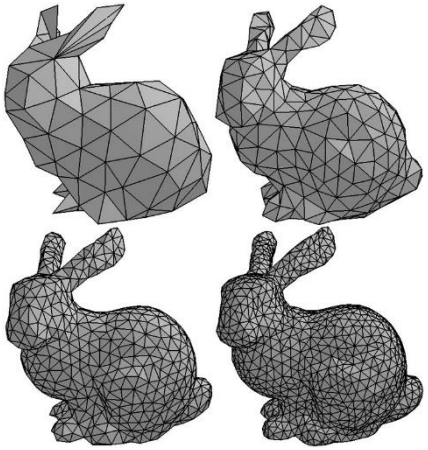

A. Nischwitz et al. „Computergrafik und Bildverarbeitung“, Band 1: Computergrafik“

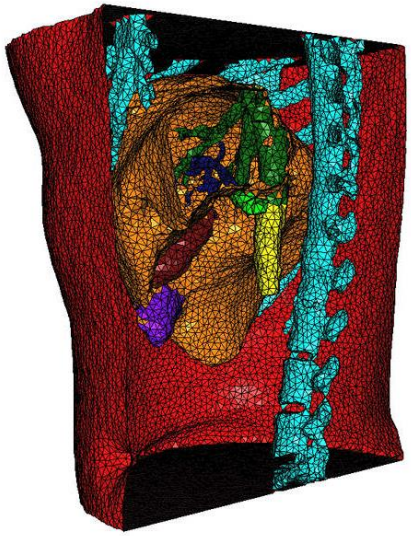
```
glBegin(GL_QUAD_STRIP);
   glVertex3fv(v0);
   glVertex3fv(v1);
   glVertex3fv(v2);
   glVertex3fv(v3);
   glVertex3fv(v4);
   glVertex3fv(v5);
   glVertex3fv(v6);
   glVertex3fv(v7);
glEnd();
```

# 3.2. POLYGONNETZE

# Polygonnetze

- Zusammenfügen von Grundobjekten (i.d.R. Dreiecke, Vierecke) zur Approximation komplexer Geometrien

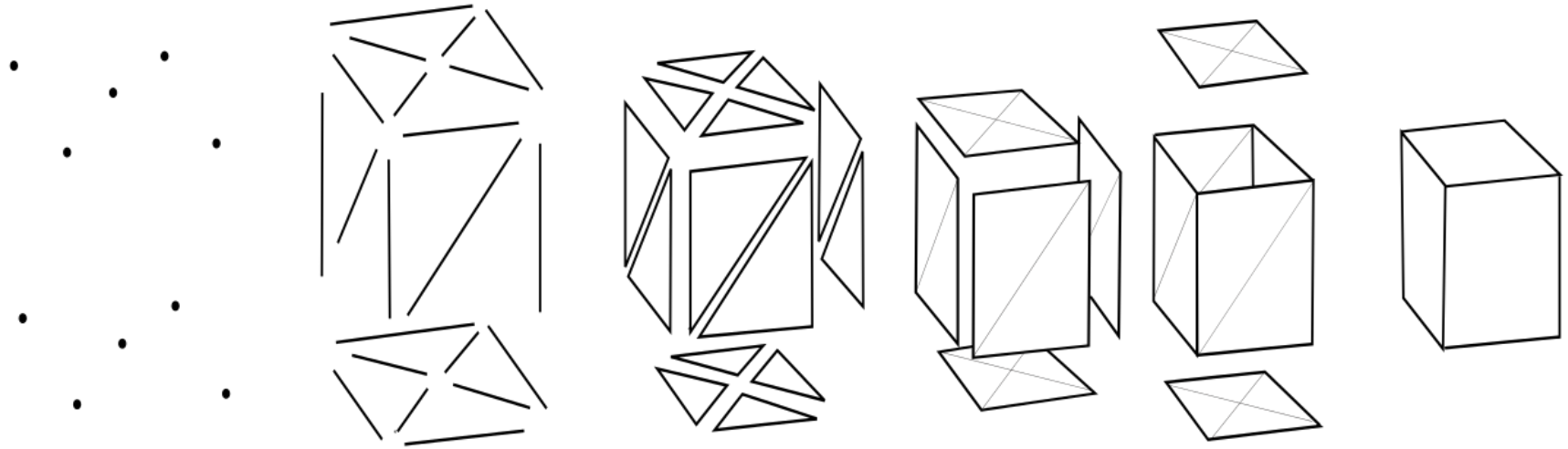




- Bestehend aus
  - Knoten (= Vertices)
  - Kanten (= Edges) → Flächen (= Faces)/Polygone
- Jeder Knoten muss mindestens eine Verbindung zum Restnetz haben
- In der Computergrafik üblicherweise Oberflächennetze

http://www.cmap.polytechnique.fr/~peyre/geodesic_computations/
https://de.wikipedia.org/wiki/Polygonnetz
http://doc.cgal.org/latest/Mesh_3/

# Terminologie bei Polygonnetzen (Meshes)



vertices — Knoten

edge — Kanten

faces — Facetten

polygons — Polygone

surfaces — Oberflächen

https://en.wikipedia.org/wiki/Polygon_mesh

# Datenstrukturen bei Polygonnetzen

## Knotenliste

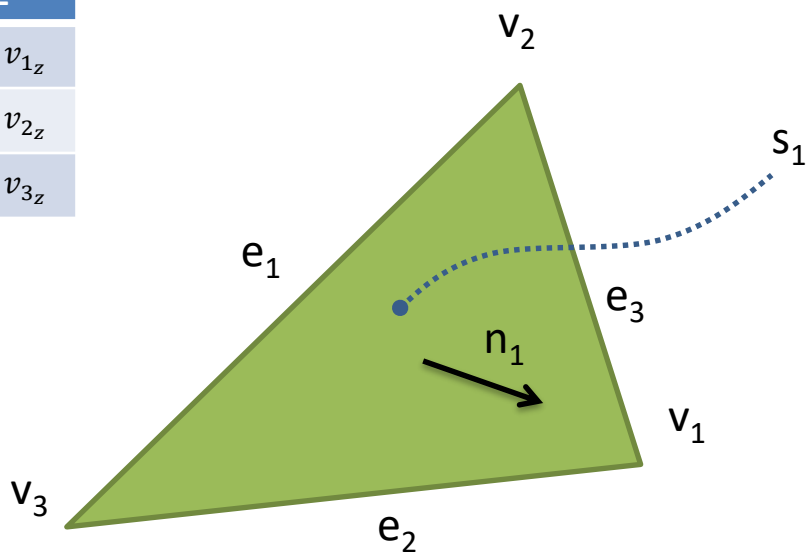
*Beispiel: Dreiecksnetz*

Polygone

| ID | Knoten 1 | Knoten 2 | Knoten 3 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |

Knoten

| ID | X | Y | Z |
|---|---|---|---|
| 1 | $v_{1_x}$ | $v_{1_y}$ | $v_{1_z}$ |
| 2 | $v_{2_x}$ | $v_{2_y}$ | $v_{2_z}$ |
| 3 | $v_{3_x}$ | $v_{3_y}$ | $v_{3_z}$ |




- Polygon definiert als Liste von Zeigern auf Vertices

# Datenstrukturen bei Polygonnetzen

## Kantenliste

*Beispiel: Dreiecksnetz*

Polygone

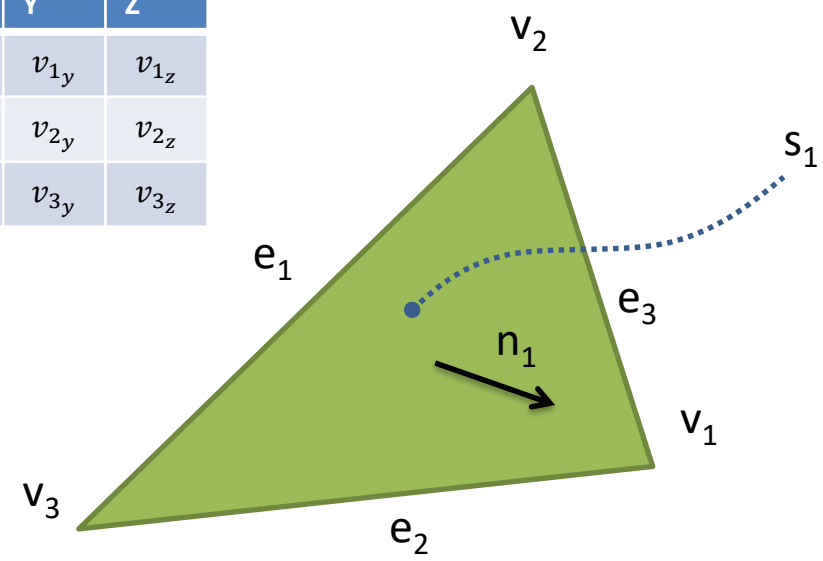
| ID | Kante 1 | Kante 2 | Kante 3 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |

Kanten

| ID | Knoten 1 | Knoten 2 |
|---|---|---|
| 1 | 2 | 3 |
| 2 | 3 | 1 |
| 3 | 1 | 2 |

Knoten

| ID | X | Y | Z |
|---|---|---|---|
| 1 | $v_{1_x}$ | $v_{1_y}$ | $v_{1_z}$ |
| 2 | $v_{2_x}$ | $v_{2_y}$ | $v_{2_z}$ |
| 3 | $v_{3_x}$ | $v_{3_y}$ | $v_{3_z}$ |



# Vergleich der Datenstrukturen

| | Vorteile | Nachteile |
|---|---|---|
| **Knotenliste** | • Trennung von Geometrie und Netztopologie<br>• Geringer Speicherbedarf | • Kanten werden mehrmals definiert<br>• Suche nach Polygonen, die eine Kante enthalten ineffizient |
| **Kantenliste** | • Trennung von Geometrie und Netztopologie<br>• Schnelle Bestimmung von Randkanten (Kanten mit nur einem Verweis auf Polygon | • Suche nach Polygonen, die einen Vertex enthalten ineffizient |

# Winged Edge Datenstruktur

- Zusätzlich zu Kantenliste: Zeiger auf ankommende/abgehende Kanten
- Ermöglicht effizientere Abfragen, z.B. welche Polygone zu einer Kante gehören

Polygone

| ID | Knoten 1 | Knoten 2 | Knoten 3 |
|---|---|---|---|
| 1 | 1 | 2 | 3 |

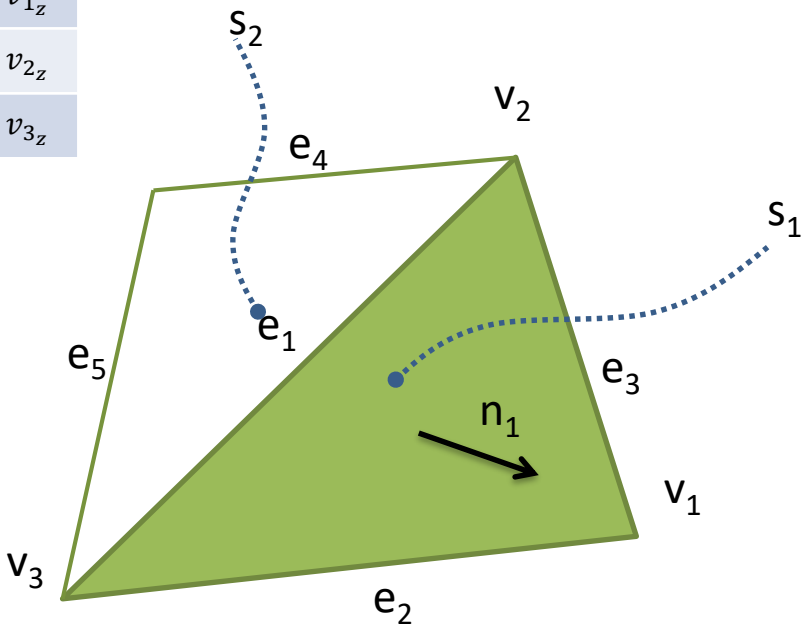
Knoten

| ID | X | Y | Z |
|---|---|---|---|
| 1 | $v_{1_x}$ | $v_{1_y}$ | $v_{1_z}$ |
| 2 | $v_{2_x}$ | $v_{2_y}$ | $v_{2_z}$ |
| 3 | $v_{3_x}$ | $v_{3_y}$ | $v_{3_z}$ |

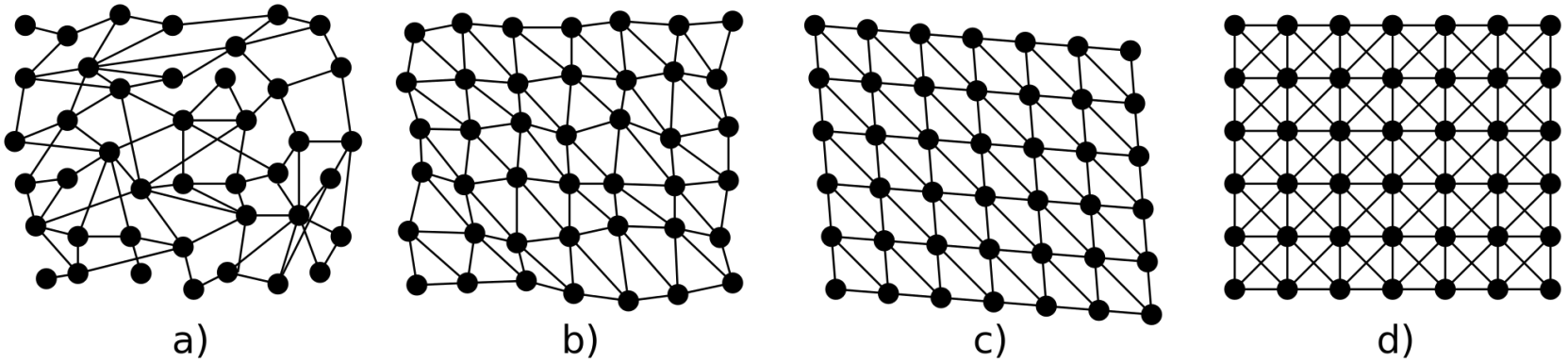
Kanten

| ID | Vertex Start | Vertex Ende | Polygon links | Polygon rechts | Linke Traverse, vorher | Linke Traverse, nach | Rechte Traverse, vorh. | Rechte Traverse, nach |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 1 | 2 | 3 | 2 | 4 | 5 |
| 2 | 3 | 1 | 1 | … | 1 | 3 | … | … |
| 3 | 1 | 2 | 1 | … | 2 | 1 | … | … |

*Beispiel: Dreiecksnetz*



# Eigenschaften von Polygonnetzen

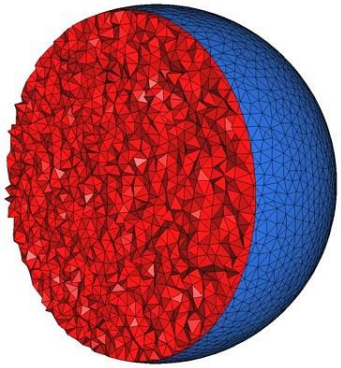



a) Keine besonderen Eigenschaften
b) Strukturiertes Polygonnetz
c) Strukturiertes, reguläres Polygonnetz
d) Strukturiertes, reguläres, orthogonales Polygonnetz

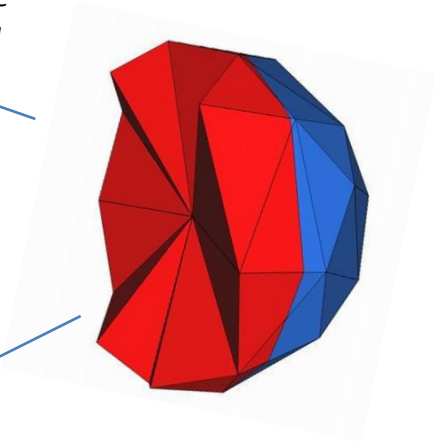
https://de.wikipedia.org/wiki/Polygonnetz

# Eigenschaften von Polygonnetzen

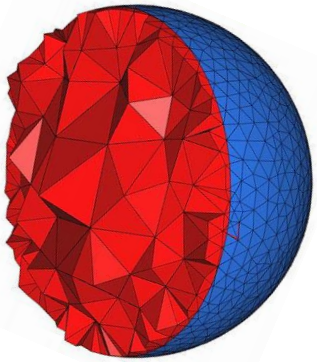
- Nicht-adaptive Polygonnetze
  - Global gleiche Auflösung



*Nicht-adaptive Verfeinerung*



*adaptive Verfeinerung*



- Adaptive Polygonnetze
  - Lokale Verfeinerung der Auflösung

http://doc.cgal.org/latest/Mesh_3/

# Polygonisierung

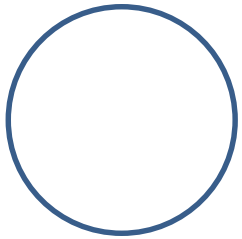
- Polygonisierung (Meshing) = Prozess der Berechnung einer Repräsentation einer gegebenen Oberfläche durch einfache Oberflächenpolygone (meist Dreiecke).

- Oberfläche in ***impliziter*** Darstellung: Lösung einer Gleichung, z.B. Kugelgleichung

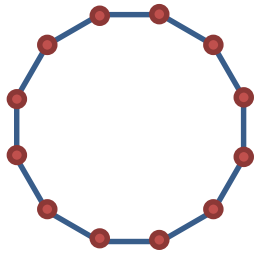
  $$f\colon r^2 = x^2 + y^2 + z^2$$

- Extraktion aus Daten, z.B. Oberflächenscan, Iso-Fläche aus Volumendaten

- Für die Computergrafik meist ***explizite*** Darstellung einer Oberfläche notwendig → Polygonisierung („Meshing“)
- Annahme für Algorithmen: glatte Oberfläche, keine Singularitäten



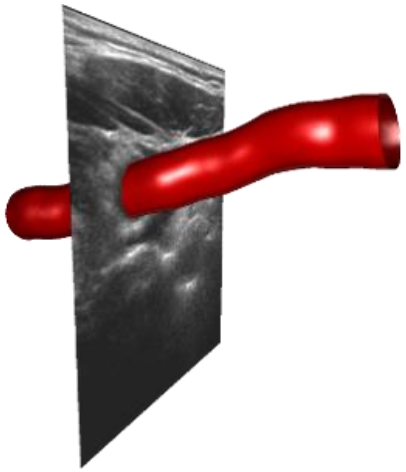
Implizite Darstellung
(analytische Gleichung)

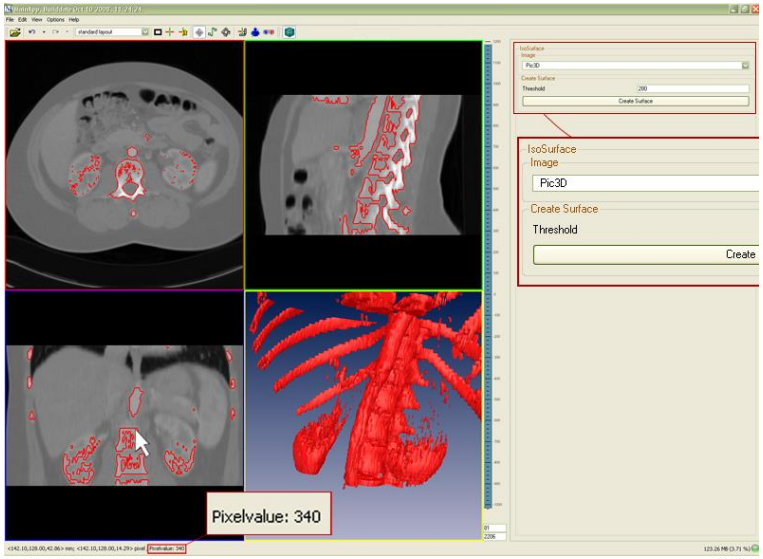


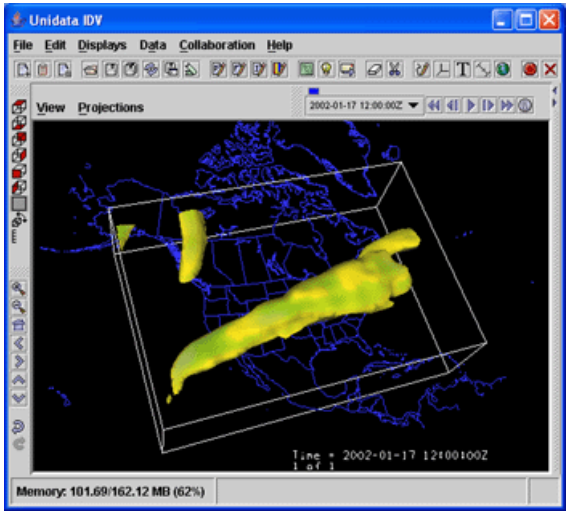
Explizite Darstellung
(Knoten, Kanten, …)

# Isoflächen

- Flächen, die im Raum benachbarte Punkte gleicher Merkmale oder Werte miteinander verbinden.
- Oft verwendet um aus Bildern/Volumendaten Oberflächen zu extrahieren,
  - z.B. Organe in der Medizin
  - z.B. Meteorologie um Gebiete gleicher Eigenschaften räumlich darzustellen



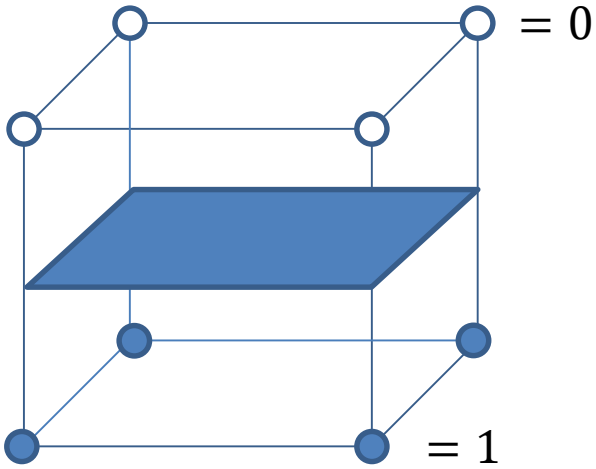



http://www.imfusion.de/products/imfusion-suite
http://docs.mitk.org/2014.10/IsoSurfaceGUI.png
http://www.unidata.ucar.edu/software/idv/docs/userguide/examples/3DSurface.html

# Marching Cubes Algorithmus

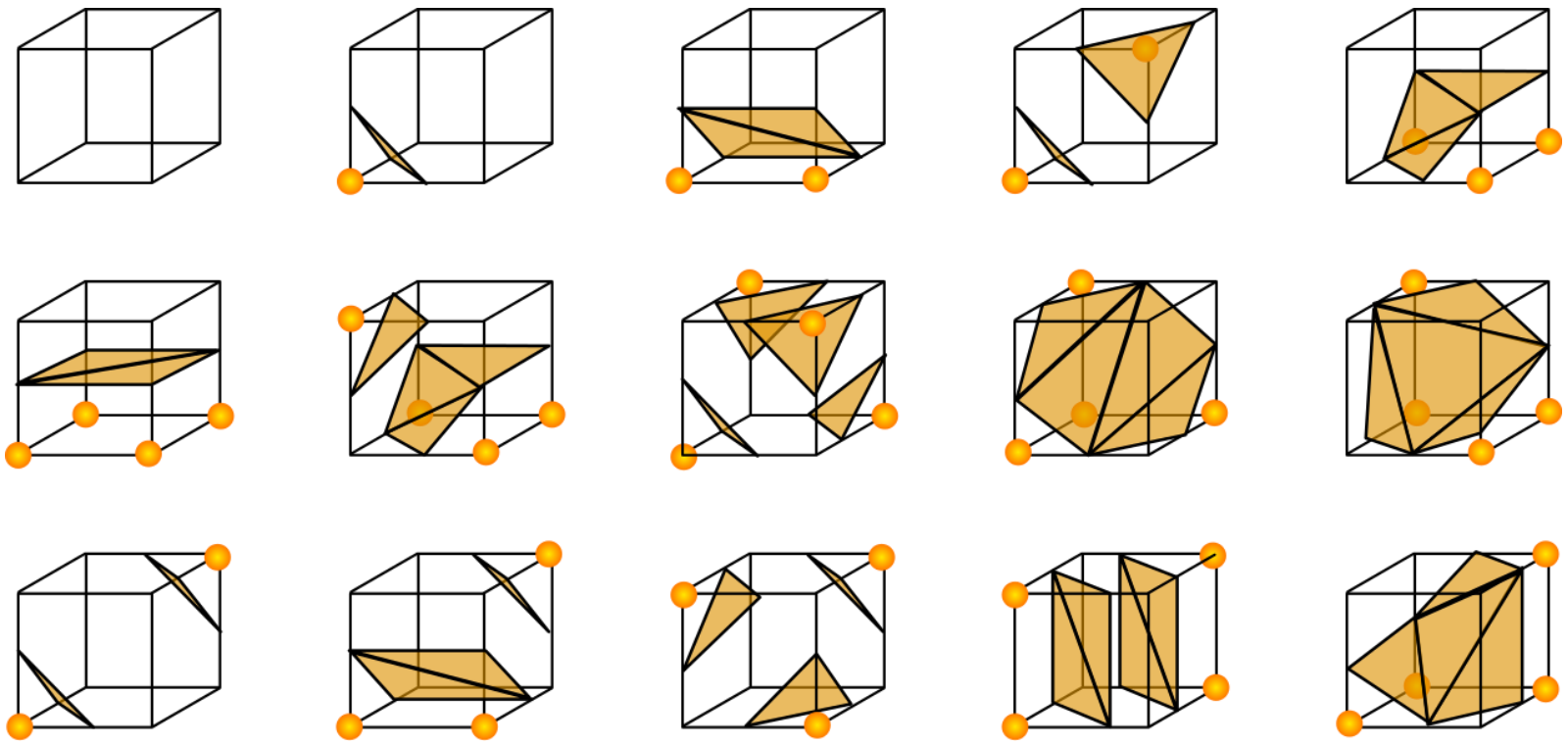
- Oft verwendet bei Erzeugung von Polygonnetzen aus Volumendaten: Annäherung einer Isofläche mit Polygonen (meist Dreiecke).
- Grundidee:
  - Unterteilung des Raums in kleine Würfel (*Cubes*)
  - Für jeden Würfel: Schnitt mit Objektoberfläche (Isofläche) bestimmen (*lokales Meshing*)
- Umsetzung für Meshgenerierung aus Voxeldaten
  - Jeder Knoten $v$ des Würfels liegt auf einem Voxel des Volumendatensatzes
  - Voxel-Grauwert $I$ + Schwellwert $T$ bestimmt ob Knoten innerhalb oder außerhalb des Objektes liegt. Zuordnung eines Wertes an jedem Knoten:




$$0 \quad \text{falls } I(v) < T$$
$$1 \quad \text{falls } I(v) > T$$

# Marching Cubes Algorithmus

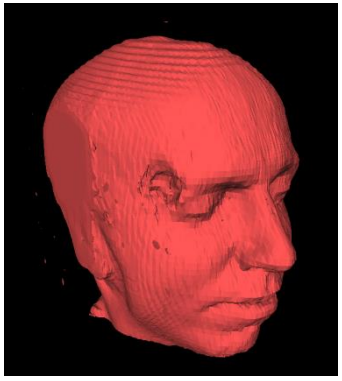
- 15 mögliche Würfelklassen entsprechend der Verteilung der Werte an den Knoten



https://de.wikipedia.org/wiki/Marching_Cubes

# Marching Cubes Algorithmus

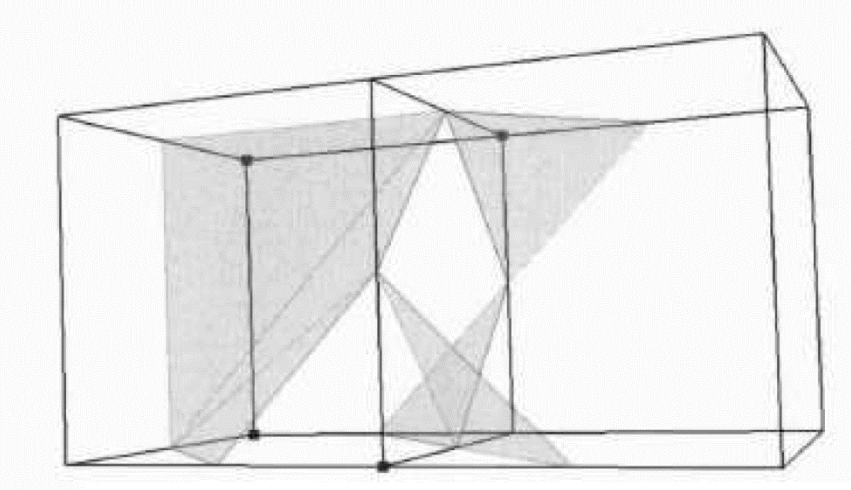
- Knoten des Polygonnetzes = Schnittpunkte der Oberfläche mit den Würfelkanten
  - Bestimmt durch lineare Interpolation der Grauwerte der zwei Vertices einer Würfelkante
- Bestimmung der Normalen der Oberfläche:
  - Schätzen des Grauwertgradienten an den Knoten des Würfels
  - Interpolation des Gradienten am berechneten Knoten des Polygonnetzes
- Zusammensetzen der Oberflächen aller Würfel ergibt gesamtes Polygonnetz des Objektes



https://de.wikipedia.org/wiki/Marching_Cubes

# Marching Cubes Algorithmus

- Vorteile:
  - Schnelle Implementierung über Lookup-Table
- Nachteile:
  - Löcher in der Oberfläche, Mehrdeutigkeitsprobleme
  - Hohe Anzahl Polygone
  - Probleme bei spitzen Details an Oberflächen



G.M. Treece, R.W. Prager, A.H. Gee, “Regularised marching tetrahedra: improved iso-surface extraction”; Computers & Graphics Volume 23, Issue 4, Pages 583–598, August 1999

# Triangulation

- = Teilung einer Oberfläche in Dreiecke

- **Polygon-Triangulation**
  - Unterteilung gegebener Polygone in Dreiecke (= *Tesselation*)
  - Algorithmen i.d.R. abhängig von Eigenschaften des Polygons

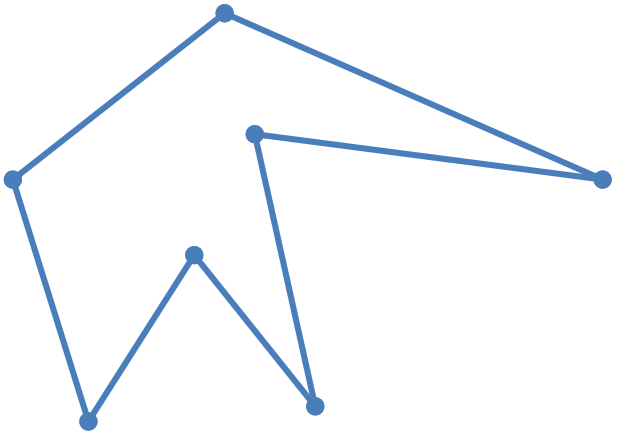
- **Punktwolken-Triangulation**
  - Triangulation komplexer Geometrien (z.B. aus Oberflächenscan)
  - Verbinden der Vertices zu Dreiecken

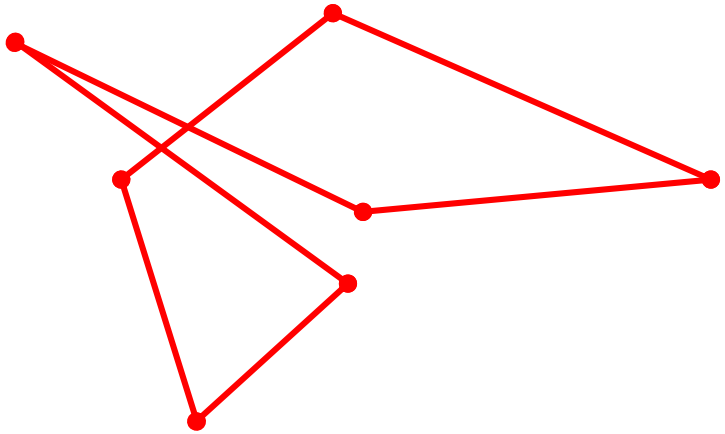
- Mehr oder weniger Einschränkung der Eigenschaften und Meshqualität
  - z.B. Maximierung des Innenwinkels des Dreiecks

# Polygontriangulation

- **Triangulations-Theorem:**
  - Jedes *einfache* Polygon hat eine Triangulation
  - Jede Triangulation eines $n$-gons besteht aus genau $n - 2$ Dreiecken

- Ein *einfaches* Polygon ist ein geschlossener Kantenzug der sich nicht selbst schneidet.

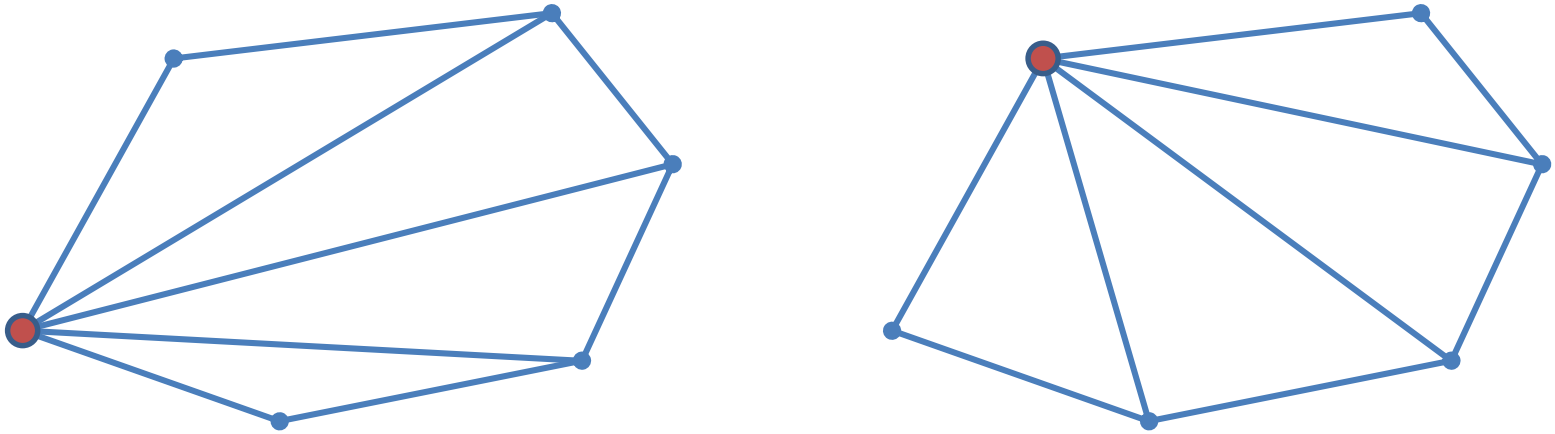




- Jedes konvexe Polygon ist ein einfaches Polygon.

# Polygontriangulation: konvexe Polygone

- trivial, siehe **`GL_TRIANGLE_FAN`**
- Verbinden eines Vertex mit allen anderen

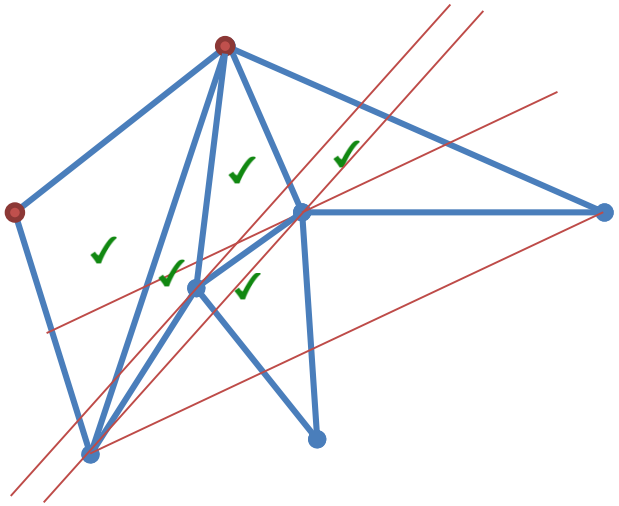


- Startvertex bestimmt u.U. Ergebnis

# Polygontriangulation: einfache Polygone

- Naiver Algorithmus für einfache Polygone: Bruteforce Diagonalen-Suche:
  - Durchlaufen aller Vertices des Polygons entlang der Polygonkante
  - Vorhergehenden und nachfolgenden Vertex verbinden
  - Rekursives Vorgehen für entstehende Teil-Polygone

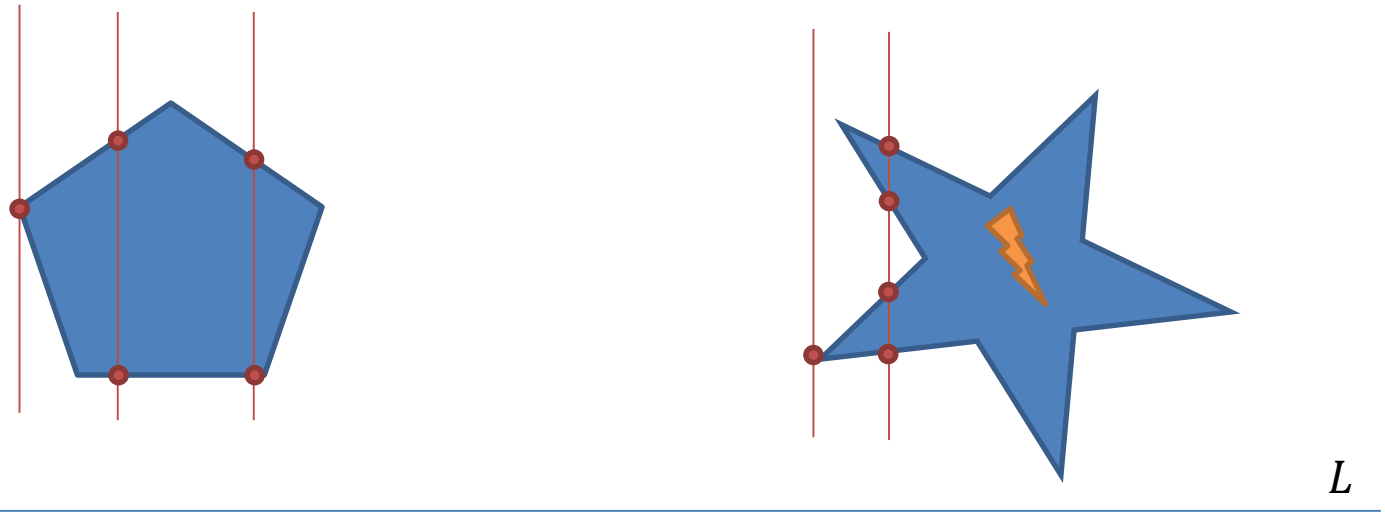
*Beginn an linker Ecke*



  - ***Achtung:*** Befindet sich ein anderer Vertex im entstandenen Dreieck: Finde nächste parallele Gerade zu dieser Kante durch einen der Vertices im Dreieck
  - Aktueller Startvertex: Position halten bis Vertex nicht mehr Teil eines weiter teilbaren Polygons ist.

# Monotonie von Polygonen

- Ein planares Polygon $P$ ist monoton bzgl. einer Geraden $L$ wenn jede Senkrechte zu $L$ das Polygon maximal zweimal schneidet.




- Jedes konvexe Polygon ist auch monoton
- Jedes Polygon, das monoton zu jeder beliebigen Geraden $L$ ist, ist auch konvex

# Polygontriangulation: monotone Polygone

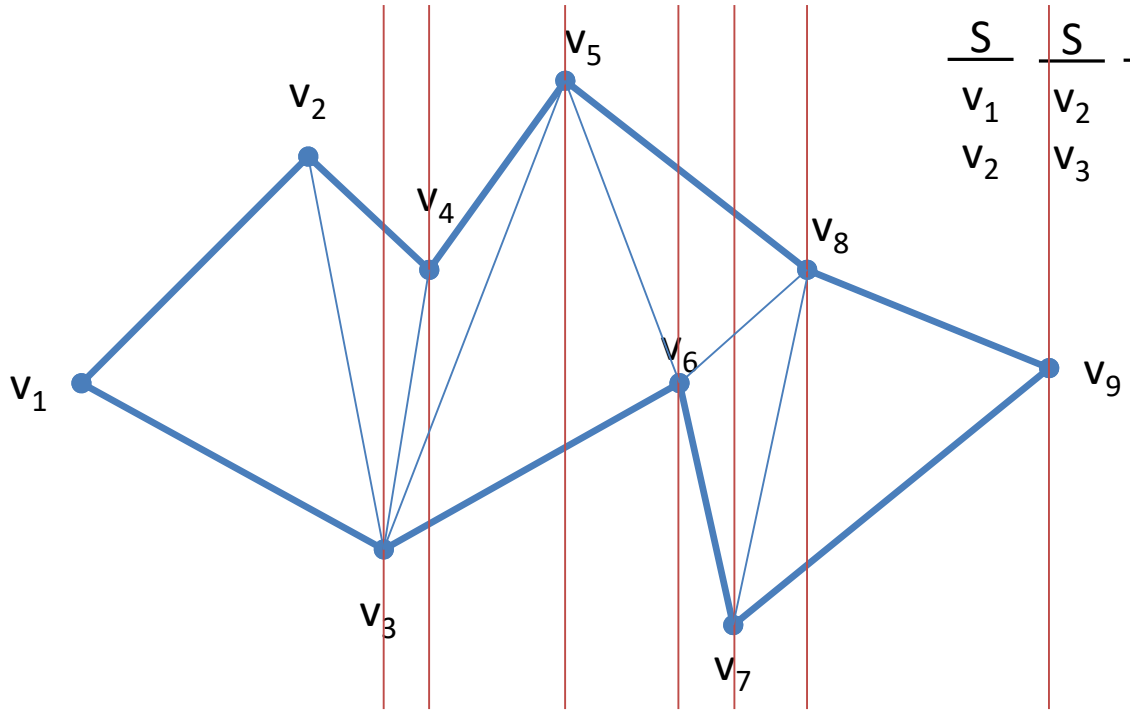
- Sweep-Line-Algorithmus: am Beispiel x-monotones Polygon
- Initialisierung:
  - Sortiere Vertices nach x-Koordinate (1. Kriterium) → $v_1$ … $v_n$
  - Erzeuge Stack S mit nicht-bearbeiteten Punkten: $v_1$ (S.top-1) und $v_2$ (S.top)
- Pseudo-Code:

```
FOR i=3 bis n
    IF v_i auf anderer Seite als S.top
        Kante von v_i zu Punkten in S bis auf Untersten
        Entferne alle Punkte aus S
        Lege v_i-1 und v_i auf S
    ELSE
        WHILE S.top-1 nicht für v_i von S.top verdeckt wird
            Erstelle Kante von v_i nach S.top-1 und entferne S.top
        Lege v_i auf S

Füge Kante von v_n zu Punkten in S bis auf Obersten und Untersten ein
```

# Polygontriangulation: monotone Polygone

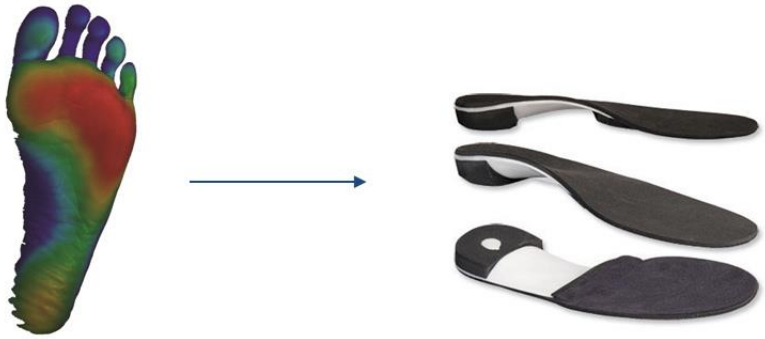
Beispiel:

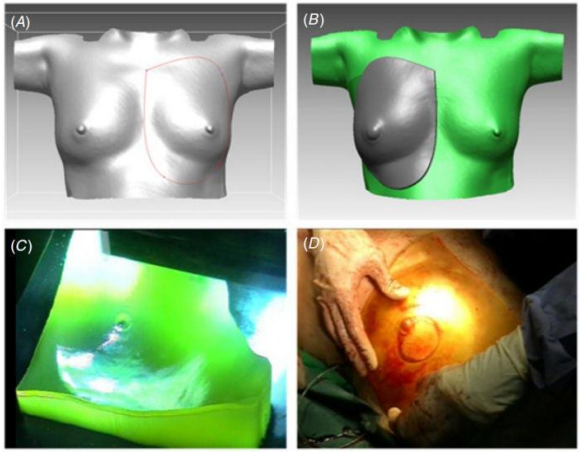



| S | S | S | S | S | S | S |
|---|---|---|---|---|---|---|
| $v_1$ | $v_2$ | $v_3$ | $v_3$ | $v_5$ | $v_5$ | $v_7$ |
| $v_2$ | $v_3$ | $v_4$ | $v_5$ | $v_6$ | $v_6$ | $v_8$ |
| | | | | | $v_7$ | |

# Punktwolkengenerierung

- Z.B. Automatisiert durch 3D Scanner: Abtastung der Oberfläche an diskreten Punkten
- Z.B. Manuelle oder automatische Extraktion aus Bilddaten



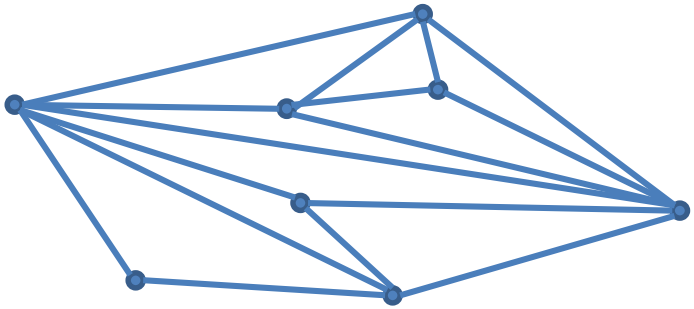



http://www.lmi3d.com/blog/medical-applications-3d-scanning
http://medicalphysicsweb.org/cws/article/research/47264
http://gispoint.de/news-einzelansicht/1395-neue-wege-der-datenerfassung-im-baubereich.html

# Punktwolkentriangulation

**Triangle Splitting**

- Bilden der konvexen Hülle aus der Punktmenge
- Triviale Triangulation des konvexen Polygons
- Für jeden inneren Punkt: Einfügen von Kanten zu den Vertices des umgebenden Dreiecks (in 3D Tetraeder!)

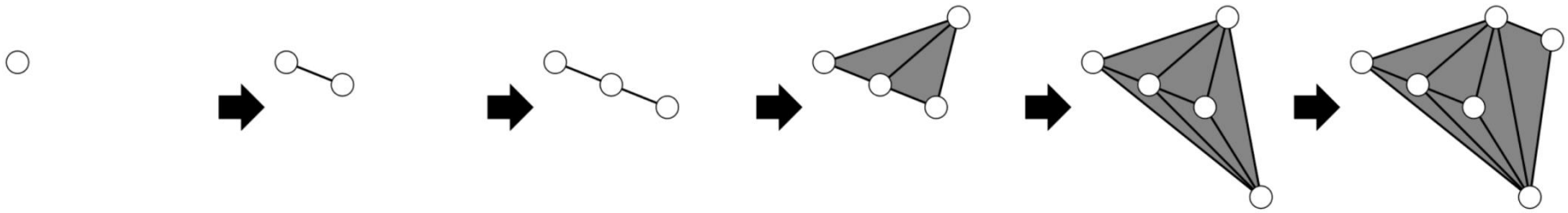


- 3D Punktwolken: Evtl. Identifizierung der Oberfläche aus dem resultierenden 3D Mesh!
- Vorsicht bei nicht-konvexen Punktwolken: schlecht-gestelltes Problem

# Punktwolkentriangulation

## Incremental construction

- Sortiere Vertices nach Ihrer x-Koordinate
- Die ersten drei Punkte bilden ein Dreieck
- Verbinde jeden weiteren Punkt mit den für ihn „sichtbaren“ bisherigen Punkten
- *Sichtbarkeit*: die neue Kante darf kein bisheriges Dreieck schneiden

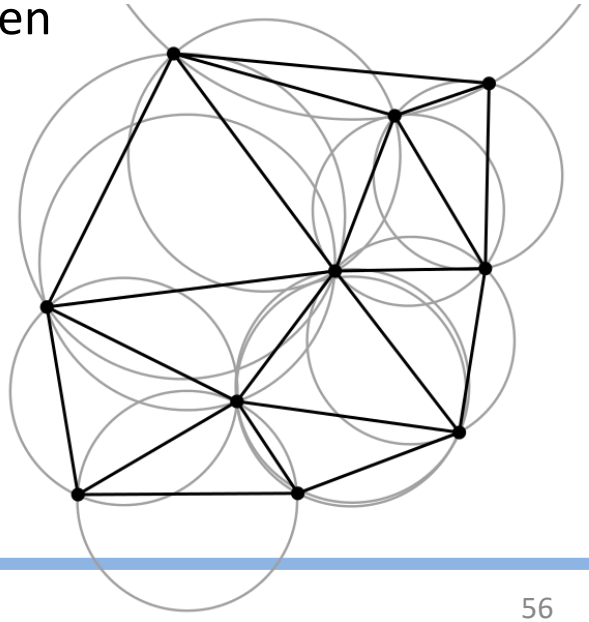


http://www.cse.ust.hk/~scheng/book/Delmesh/chapter2.pdf

# Delaunay Triangulation

- Für Interpolationen - z.B. bei der Beleuchtung - sind Dreiecksnetze mit möglichst großen Innenwinkeln besser geeignet.
- Delaunay Triangulation maximiert den minimalen Winkel in einem Dreieck
- Benannt nach russischem Mathematiker Boris Nikolajewitsch Delone (1890–1980, franz. Form des Nachnamens: Delaunay)

- **Grundprinzip**: Der Umkreis jedes Dreiecks des Netzes darf keine weiteren Vertices der vorgegebenen Verticemenge enthalten
  - In 3D: Umkugel-Bedingung für Tetraeder-Generierung



# Delaunay Triangulation

- Test ob ein Punkt $v$ im Umkreis eines Dreiecks $abc$ liegt (2D):

  Berechnung der Determinante $\det$ der Matrix $D$

$$\det(D) = \begin{vmatrix} a_x & a_y & {a_x}^2 + {a_y}^2 & 1 \\ b_x & b_y & {b_x}^2 + {b_y}^2 & 1 \\ c_x & c_y & {c_x}^2 + {c_y}^2 & 1 \\ v_x & v_y & {v_x}^2 + {v_y}^2 & 1 \end{vmatrix}$$

- $\det(D) < 0$: $v$ liegt außerhalb des Umkreises des Dreiecks $abc$
- $\det(D) = 0$ : $v$ liegt auf dem Umkreises des Dreiecks $abc$
- $\det(D) > 0$ : $v$ liegt innerhalb des Umkreises des Dreiecks $abc$

# Delaunay Triangulation

- Konstruktion – Dualität mit **Voronoi-Diagrammen**:
  - *Gegeben*: eine Menge $M$ von $n$ Vertices.
  - Das Voronoi-Diagramm von $M$ zerlegt die Ebene in $n$ disjunkte Gebiete (sogenannte Voronoi-Zellen)
  - Die Voronoi-Zelle $V$ eines Vertex $v$ enthält genau einen Vertex aus $M$ sowie alle geometrischen Punkte $w$, die näher an $v$ als an jedem anderen Vertex $v'$ liegen:

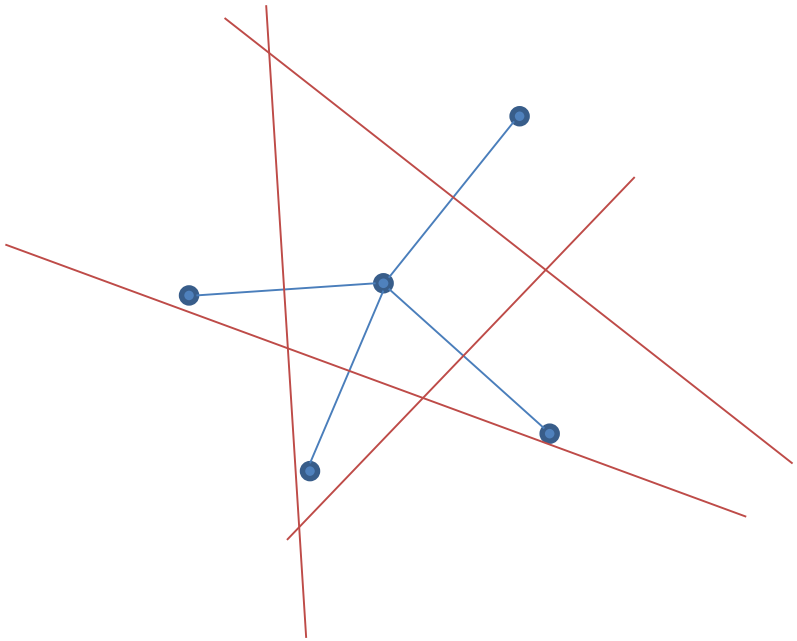
$$V(v) = \{w \in \mathbb{R}^2 : \forall v' \in M \backslash \{v\} : dist(w, v) < dist(w, v')\}$$

  - Naive Konstruktion: Bilden von Halbebenen $h$ zwischen den Vertices $v$ und $v'$:

$$h(v, v') = \{w \in \mathbb{R}^2 : dist(w, v) < dist(w, v')\}$$

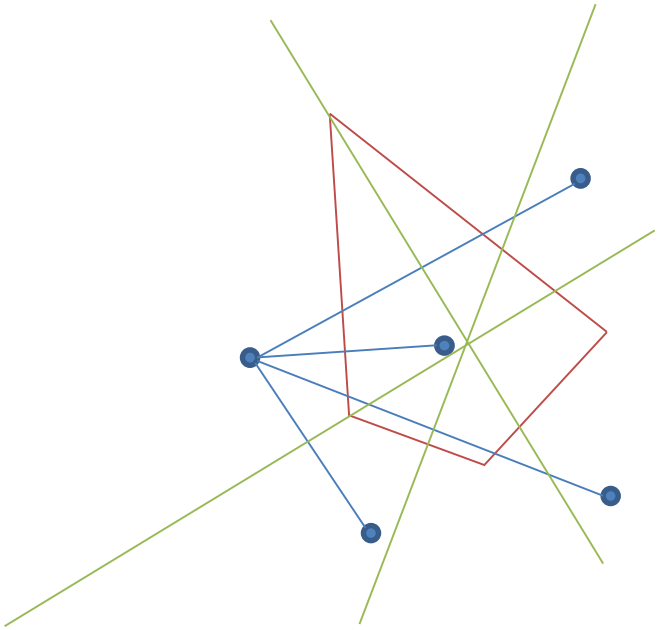
# Delaunay Triangulation

- Konstruktion eines Voronoi-Diagramms



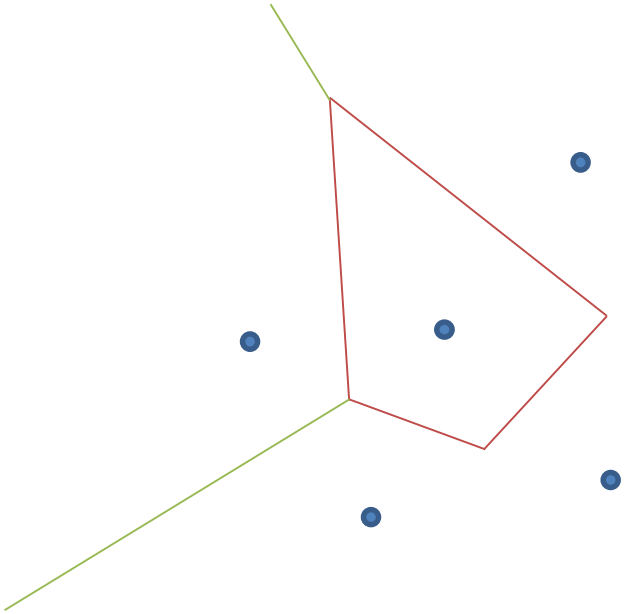
# Delaunay Triangulation

- Konstruktion eines Voronoi-Diagramms



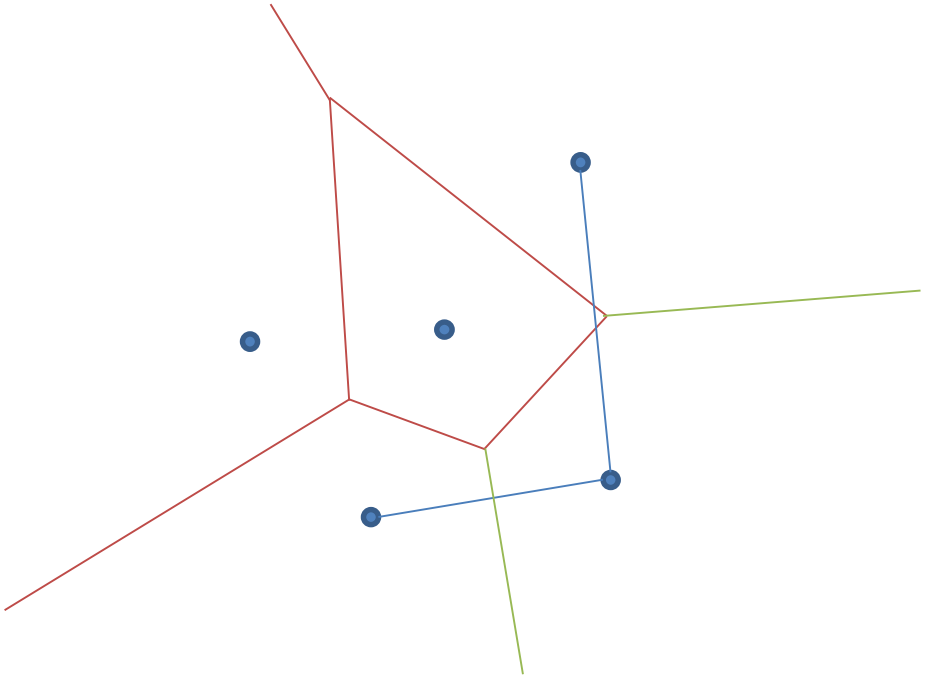
# Delaunay Triangulation

- Konstruktion eines Voronoi-Diagramms



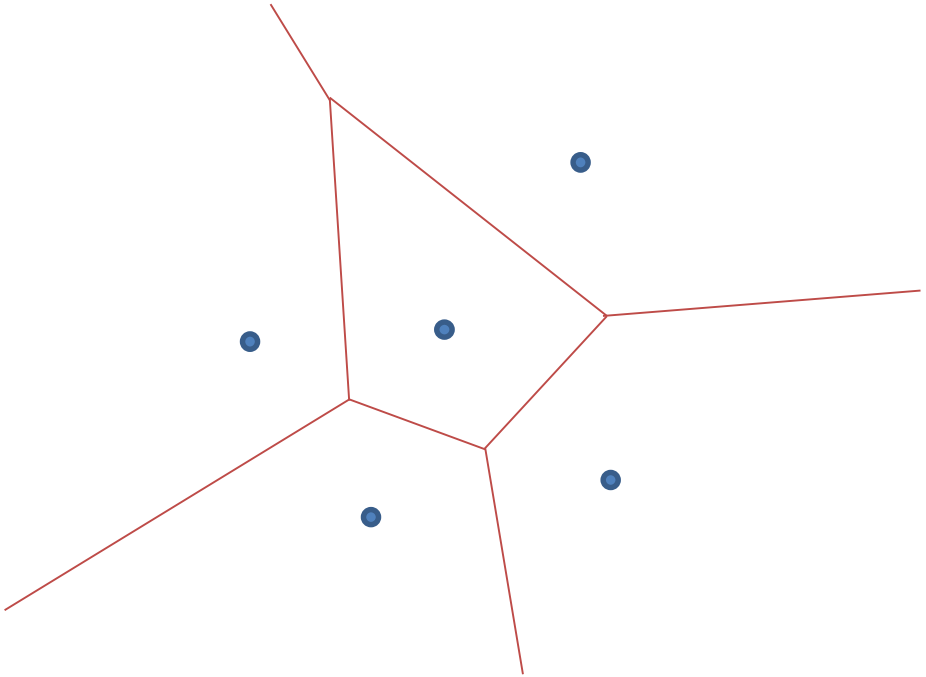
# Delaunay Triangulation

- Konstruktion eines Voronoi-Diagramms



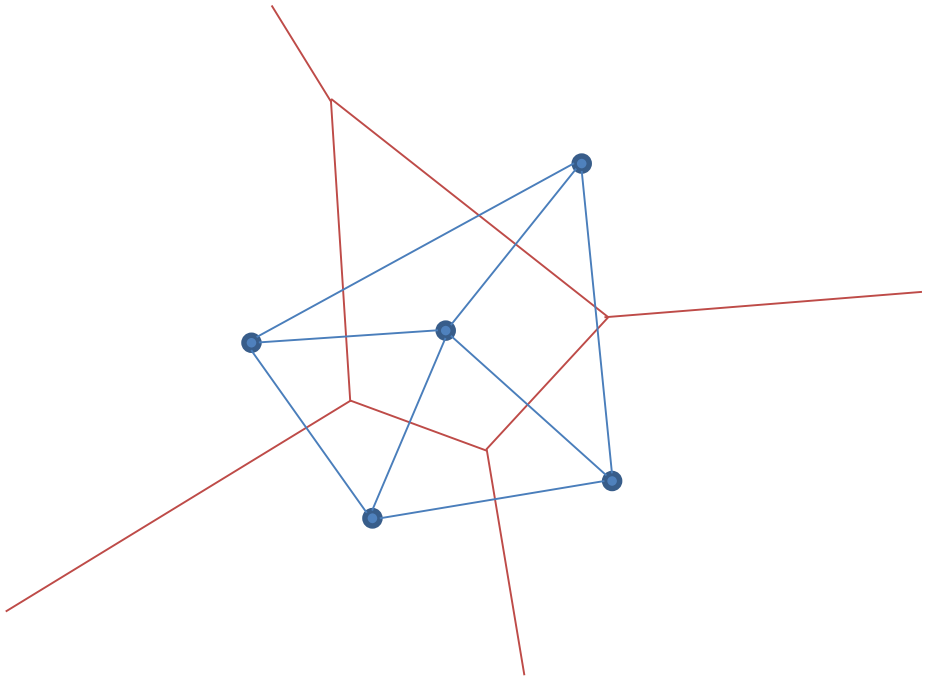
# Delaunay Triangulation

- Konstruktion eines Voronoi-Diagramms



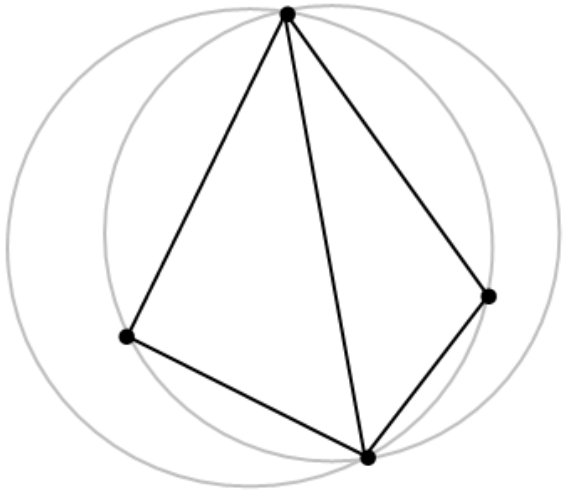
# Delaunay Triangulation

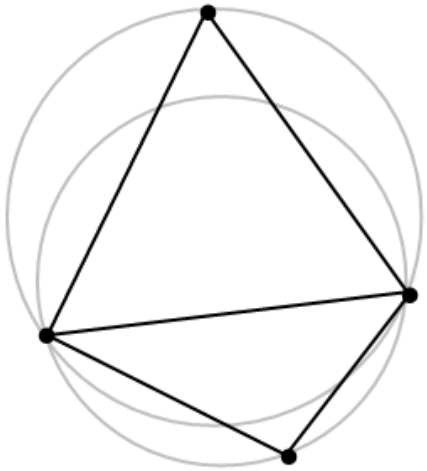
- Die Delaunay Triangulation einer Punktmenge ist der duale Graph eines Voronoi-Diagramms
- Konstruktion durch orthogonale Linie zu jeder Voronoi-Kante



# Delaunay Triangulation

- Zahlreiche Algorithmen zur Berechnung der Delaunay Triangulation aus einer Punktmenge vorhanden
- **Edge Flipping**
  - Erzeugen eines beliebigen Dreiecksnetzes
  - Für jedes Dreieck: prüfen ob der Umkreis einen weiteren Punkt einschließt, der Teil eines angrenzenden Dreiecks ist.
  - Ist dies der Fall, wird ein Flip der gemeinsamen Kante durchgeführt.





https://de.wikipedia.org/wiki/Delaunay-Triangulation

# Delaunay Triangulation

- **Inkrementelle Methode:**
  - Start: initiales Dreiecksnetz, das alle zu erwartenden Vertices einschließt
  - Einfügen eines beliebigen neuen Vertex: Suche des Dreiecks, das den Vertex enthält
  - Neuer Punkt wird mit den drei Vertices des gefundenen Dreiecks verbunden

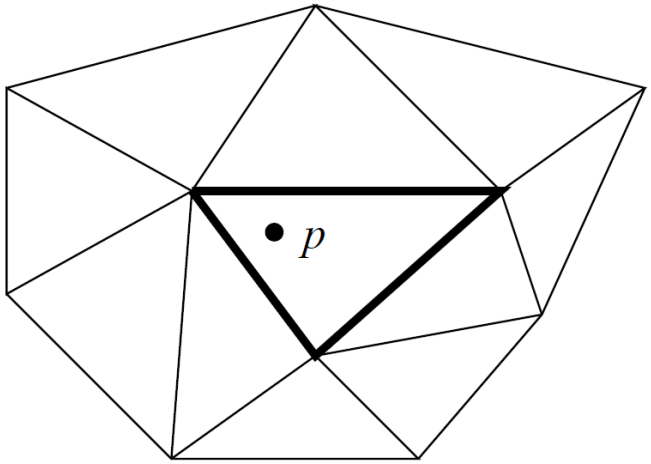
    → es entstehen drei neue Dreiecke, die nicht mehr unbedingt die Umkreisbedingung erfüllen
  - Test jedes neuen Dreiecks auf Umkreisbedingung
  - Korrekturen der Umkreisbedingung mit Flip-Algorithmus
  - Nach jeder Korrektur gibt es möglicherweise Dreiecke die die Umkreisbedingung nicht mehr erfüllen: iteratives Vorgehen.

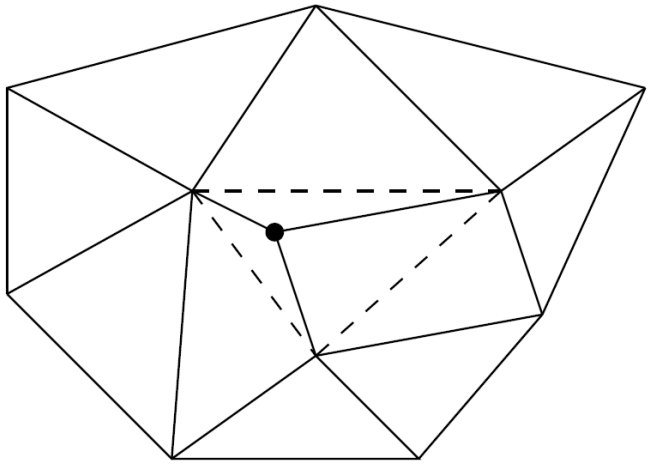
- **Modifikation der inkrementellen Methode**
  - Pro Schritt: Anfügen eines benachbarten Dreiecks (statt eines beliebigen Dreiecks bei der inkrementellen Methode)

- **Ableitung aus Voronoi-Graph**

# Delaunay Triangulation

## Beispiel: Inkrementelle Methode

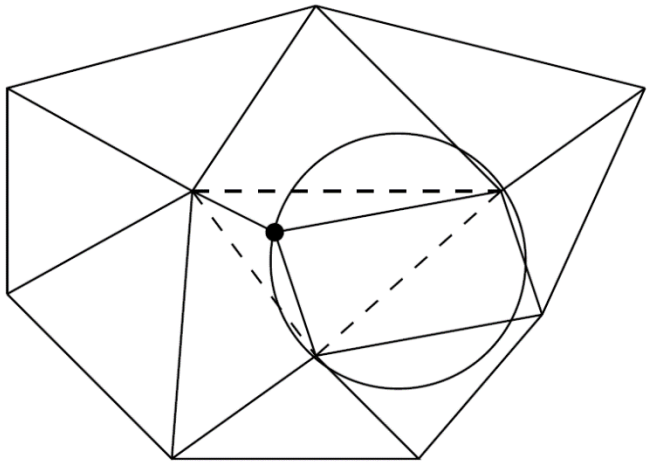


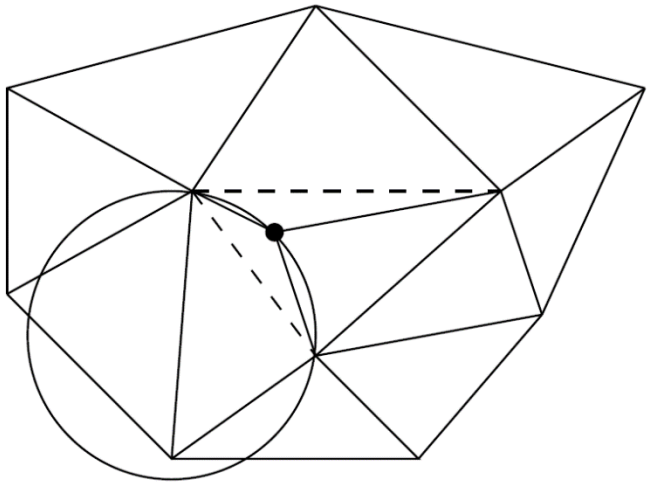



D. Lischinski: Incremental Delaunay Triangulation, in: P.S. Heckbert: „Graphics Gems“, Academic Press, 1994

# Delaunay Triangulation

## Beispiel: Inkrementelle Methode

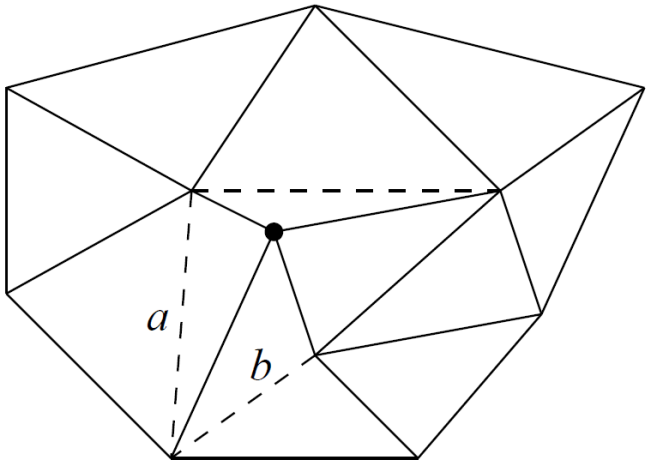


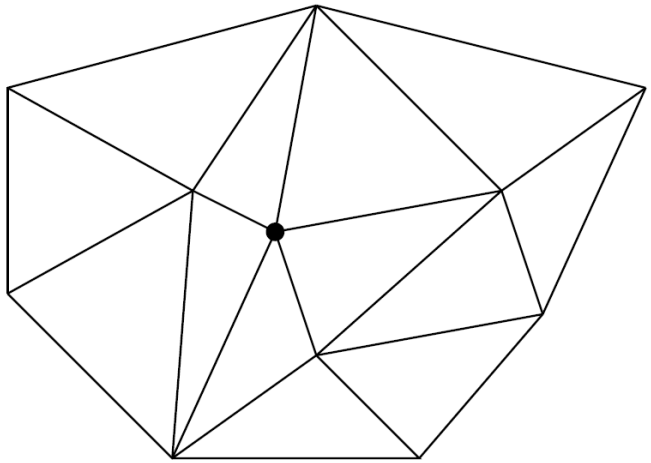


D. Lischinski: Incremental Delaunay Triangulation, in: P.S. Heckbert: „Graphics Gems“, Academic Press, 1994

# Delaunay Triangulation

## Beispiel: Inkrementelle Methode

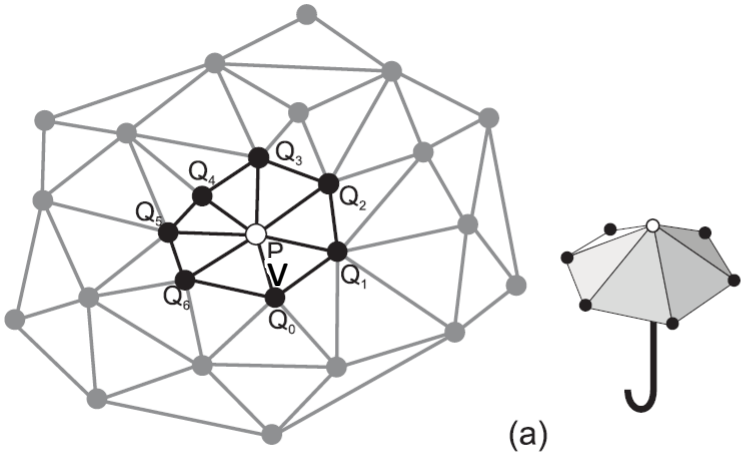


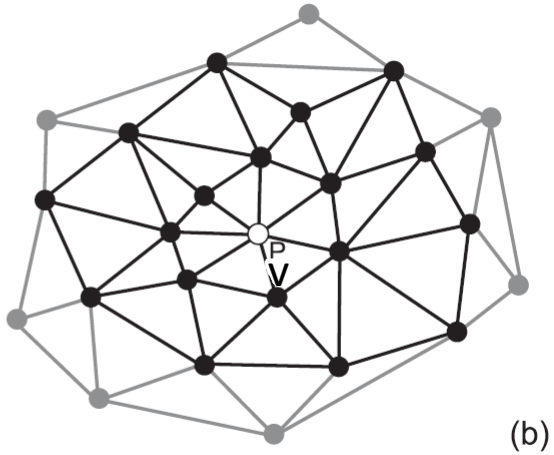
D. Lischinski: Incremental Delaunay Triangulation, in: P.S. Heckbert: „Graphics Gems“, Academic Press, 1994

# Mesh-Glättung

- Bei Generierung aus Volumendaten oft Ausreißer, Spikes, etc.
- Glättung im Allgemeinen durch Filter realisiert (siehe Bildverarbeitung)
- Zu filternde Region über *Umbrella*-Operator definiert (= Nachbarschafts-Operator)
  - Umbrella-Region 1. Grades: Alle Knoten $q_i$ die mit $v$ eine verbindende Kante haben
- Auswirkung der Glättung meist nur auf Knotenposition, Topologie bleibt erhalten




*Region 1. Grades*




*Region 2. Grades*

J. Haase: „Glättung von Polygonnetzen in medizinischen Visualisierungen“, Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, 2005

# Mesh-Glättung

- **Laplace-Filter:** Verschiebung des zentralen Knotens $v$ in das Zentrum seiner Nachbarn:

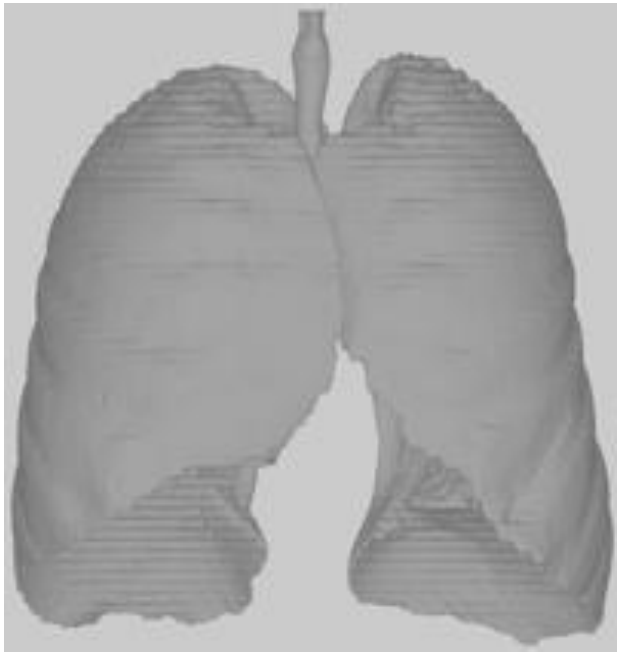
$$v' = \frac{1}{n}\sum_{i=0}^{n-1} q_i \qquad \forall v \in M$$

- Iteratives Vorgehen: Glättungsgrad kann durch Anzahl der Iterationen eingestellt werden
- Nachteil:
  - Polygonnetze schrumpfen
  - Invertierte Elemente möglich
- Abwandlung: Einführung eines Relaxationswertes $\lambda$

$$v' = v + \frac{\lambda}{n}\sum_{i=0}^{n-1} (q_i - v) \qquad \forall v \in M, 0 \leq \lambda \leq 1$$

# Mesh-Glättung

- Laplace-Glättung



- Weitere Varianten der Laplace-Glättung:
  - Z.B. HC-Algorithmus: Iterative Vor- und Zurückverschiebung

https://www.cg.tuwien.ac.at/courses/Visualisierung2/HallOfFame/2012/4-Cornel/Homepage/index.html

# Mesh-Glättung

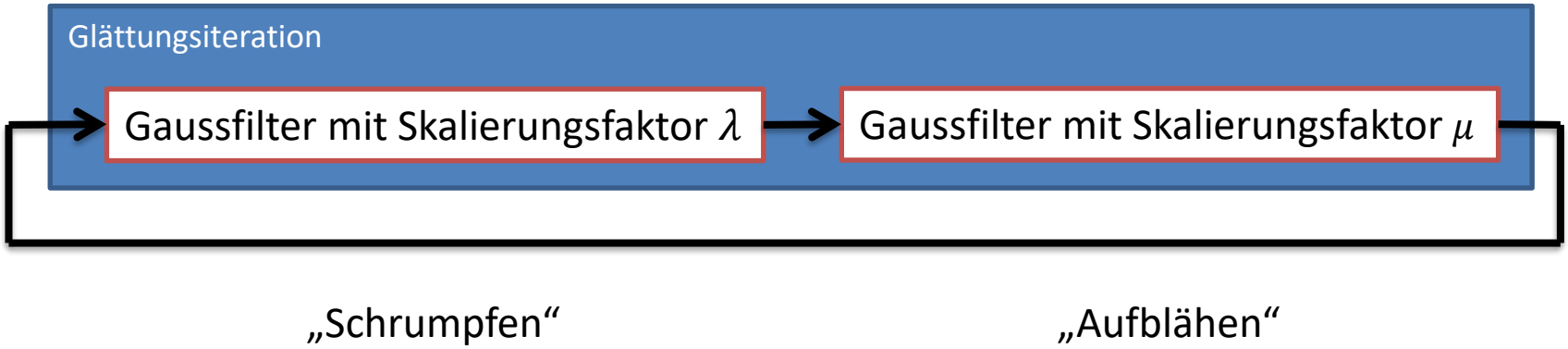
- **Gauß-Filter:** Knotenpositionen werden aus gewichtetem Mittel der Umbrella-Region 1. Grades berechnet:

$$v' = v + \lambda \sum_{i=0}^{n-1} w_i (q_i - v) \qquad , \forall v \in M, 0 \leq \lambda \leq 1$$

- Iteratives Vorgehen
- Wichtung $w$ kann pro Iteration verschieden eingestellt werden.
  - Summe der Wichtungen $w_i$ ergibt immer 1.
  - Gebräuchlich: $w_i = \frac{1}{n}$ oder Nachbarschaftsstrukturen mit einbeziehen (Distanzfunktion)
- Skalierungsfaktor $\lambda$ analog zu Relaxationswert bei Laplace-Glättung
- Im Allgemeinen sehr ähnliche Ergebnisse wie Laplace-Filterung
  - Gut zur Unterdrückung von Rauschen (kleinen Artefakten)
  - Neigt ebenfalls zum Schrumpfen

# Mesh-Glättung

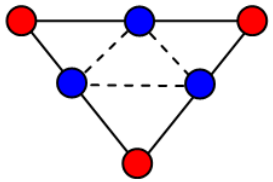
- **Lowpass-Filter:** zweifache Ausführung der Gauss-Filterung mit unterschiedlichen Skalierungsfaktoren
  - Einführung eines zweiten Skalierungsfaktors $\mu$ mit negativem Wert
  - $\mu$ erfüllt die Bedingung $0 < \lambda < -\mu$
  - $\mu$ sollte einen „geringfügig“ größeren Wert als $\lambda$ haben.



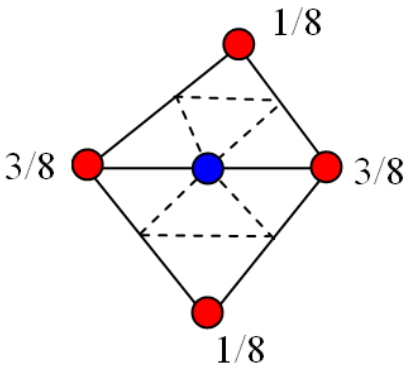

- Verringert Schrumpfen des Meshs
- Erhält Details besser als Laplace- und Gauss-Filterung
- Benötigt mehr Iterationen um optisch glatte Netze zu erzeugen

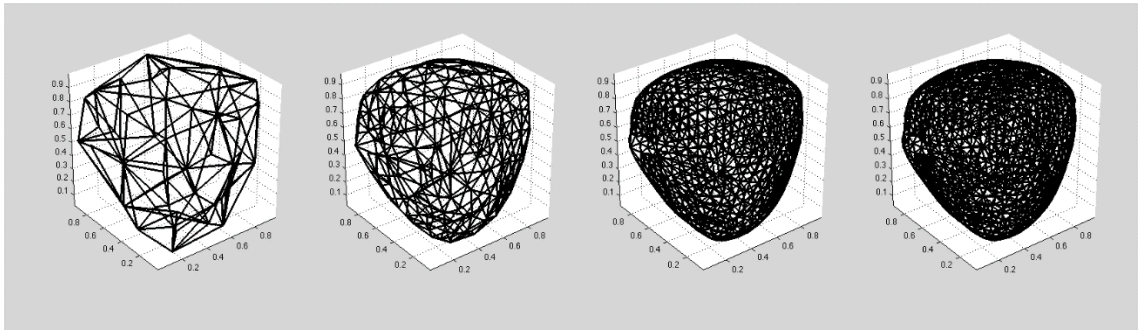
# Mesh-Glättung

- **Mesh Subdivision:** Verfeinerung des Mesh durch neue Knoten, gleichzeitige Verschiebung der Knoten (Topologie-Änderung!)
- Zwei Verarbeitungsschritte:
  1) Topologische Unterteilung
  2) Geometrische Positionierung



- Beispiel: Loop Subdivision
  - Topologische Unterteilung: Dreieck in 4 kleinere Dreiecke zerlegen durch initiales Einfügen von Knoten in der Mitte der Kanten (=Edge Vertices)
  - Geometrische Positionierung:
    - Edge Vertices: Lineare Kombination der benachbarten Knoten
    - Ursprüngliche Vertices: Verschiebung anhand Umbrella-Region.



https://graphics.stanford.edu/~mdfisher/subdivision.html
http://www.mathworks.com/matlabcentral/fileexchange/32727-fast-loop-mesh-subdivision