Einführung in die Computergrafik

Matthias B. Hullin

Institut für Informatik II, Universität Bonn

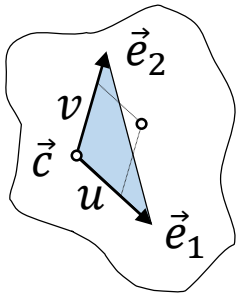
Objekte und Szenen repräsentieren

Darstellung von Oberflächengeometrie

Objektgeometrie kann auf verschiedene Weise definiert sein:

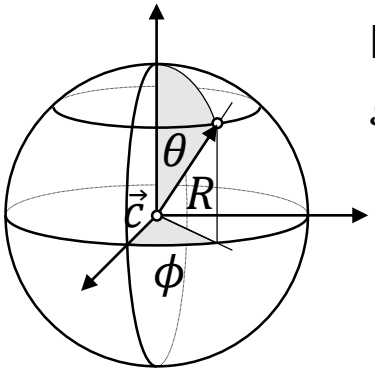
explizit: Funktion $f: \mathbb{R}^2 \rightarrow \mathbb{R}^3$
erzeugt Punkte auf der Oberfläche
$\vec{x} = f(u, v)$

z.B.:
Ebene $f_1(u, v)$
$= \vec{c} + u\vec{e}_1 + v\vec{e}_2$
Dreieck
$u > 0; v > 0; u + v \leq 1$

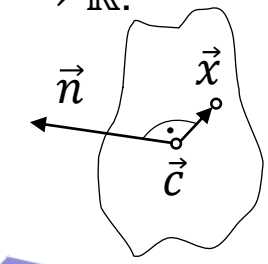


Kugel $f_2(\phi, \theta)$

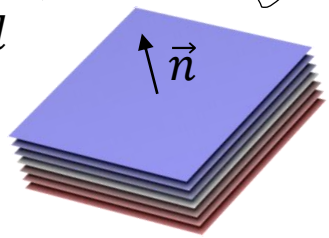
$$= \vec{c} + R \begin{pmatrix} \cos(\phi)\sin(\theta) \\ \sin(\phi)\sin(\theta) \\ \cos(\theta) \end{pmatrix}$$



implizit: Objektoberfläche als
Isofläche einer Funktion $g: \mathbb{R}^3 \rightarrow \mathbb{R}$:
$\{\vec{x} | g(\vec{x}) = 0\}$

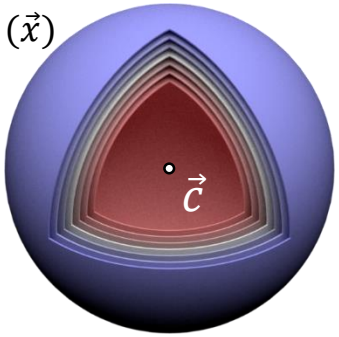


z.B.:
Ebene $g_1(\vec{x}) = \vec{n} \cdot (\vec{x} - \vec{c})$
$= \vec{n} \cdot \vec{x} - d$



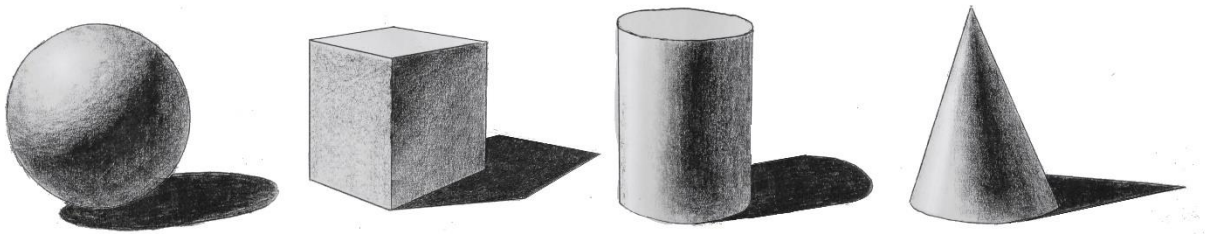
Isoflächen für verschiedene Werte von $g_1/g_2\,(\vec{x})$

Kugel
$g_2(\vec{x}) = |\vec{x} - \vec{c}| - R$



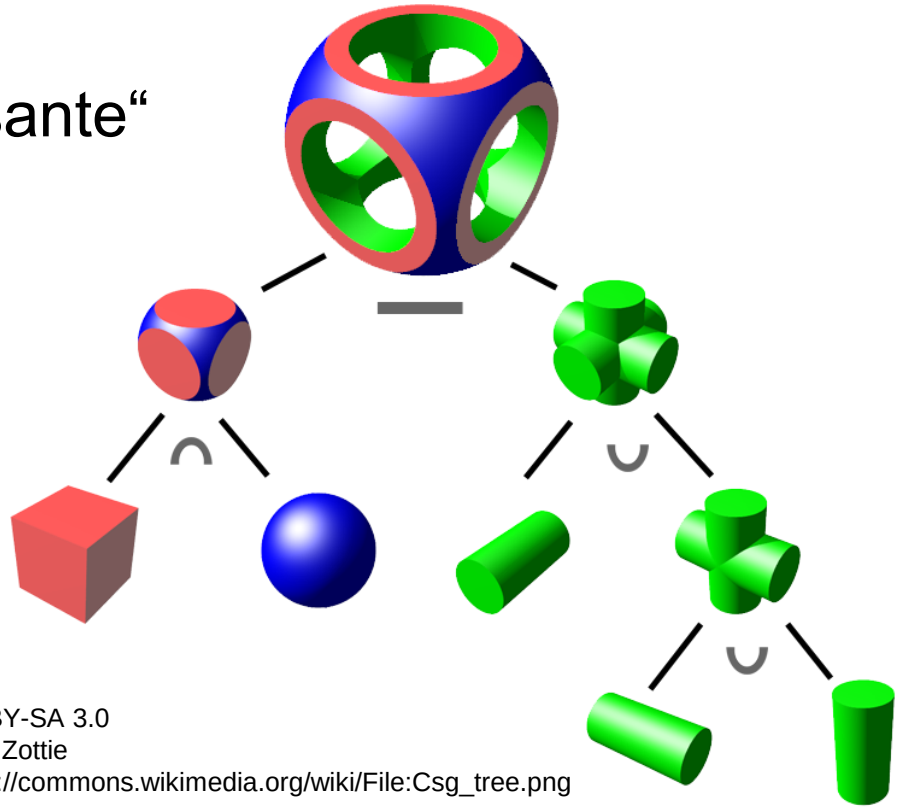
„Primitives“ - Grundkörper

Kugel, Würfel, Zylinder, Kegel, Ebene, Torus, Pyramide, ...

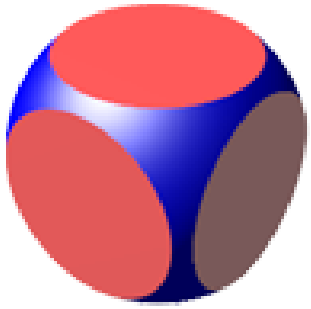


(C) Frank Robert Dixon

- Sehr umständlich „interessante“ Objekte aus diesen Grundformen zu bilden.
- Boolesche Operatoren CSG („constructive solid geometry“)
- Vor allem in CAD/CAM verwendet!



CSG bei implizit definierten Oberflächen

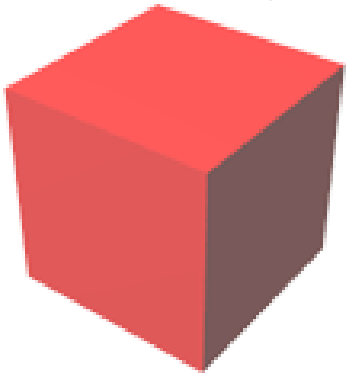


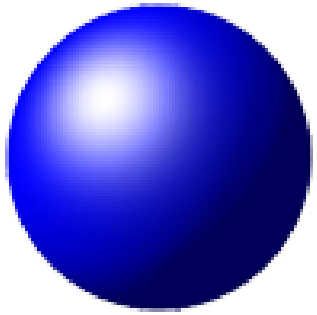
$\{\vec{x} | g_2(\vec{x}) = 0 \ \wedge g_1(\vec{x}) < 0\}$

$\{\vec{x} | g_1(\vec{x}) = 0 \ \wedge g_2(\vec{x}) < 0\}$

„Menge aller Punkte, die auf der Oberfläche von Objekt 1 und im Inneren von Objekt 2 liegen“

Schnittkörper





$g_1(\vec{x}) = \|\vec{x}\|_\infty - R_1 = 0$

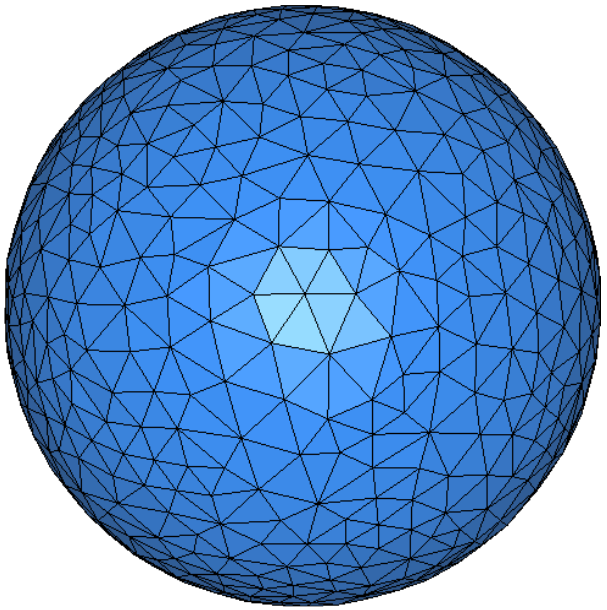
$g_2(\vec{x}) = \|\vec{x}\|_2 - R_2 = 0$

$\|\vec{x}\|_\infty = \max_i |x_i|$

Dreiecksnetze (Folien nach Foley,Marschner,Bala)

- Mindestens 90% aller Computergrafikanwendungen verwenden Dreiecksnetze

Andrzej Barabasz

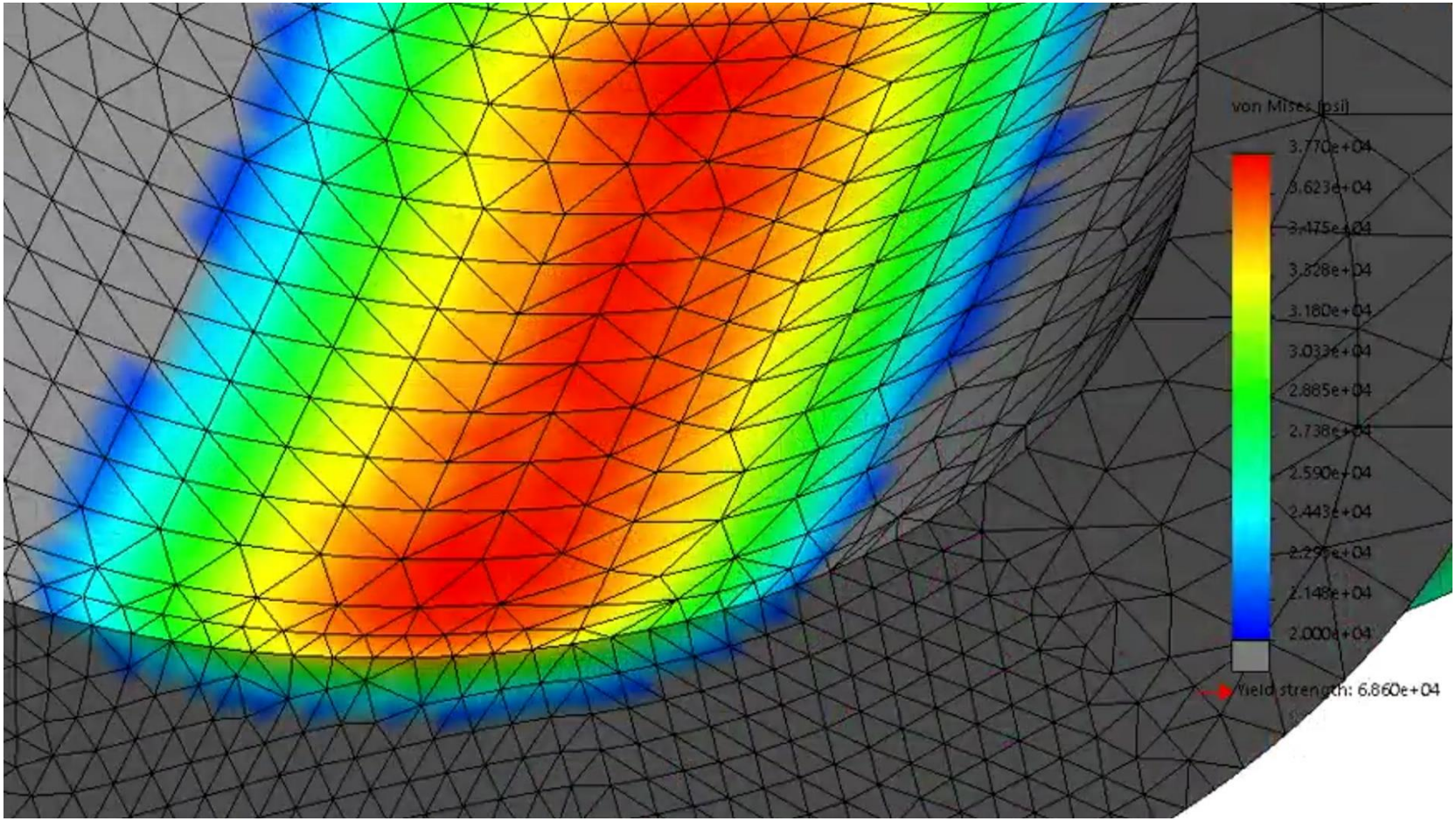


Rineau und Yvinec, CGAL Dokumentation

Exakte Kugel

Angenäherte Kugel

Finite-Elemente-Analyse



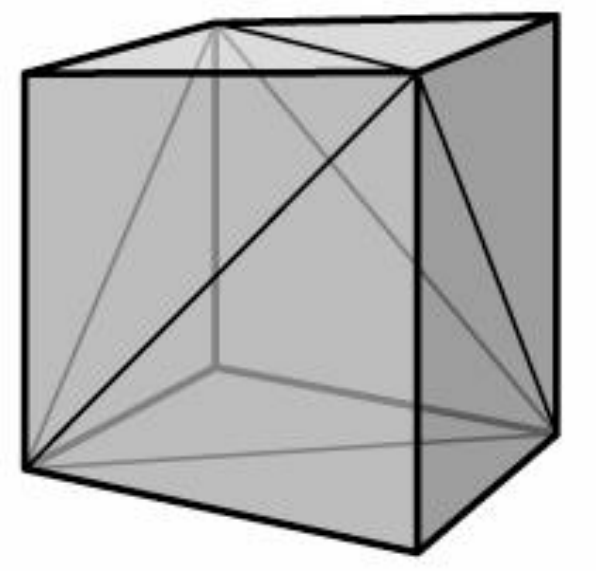
Tony Abbey, SOLIDWORKS: Simulation for Finite Element Analysis

Architektur

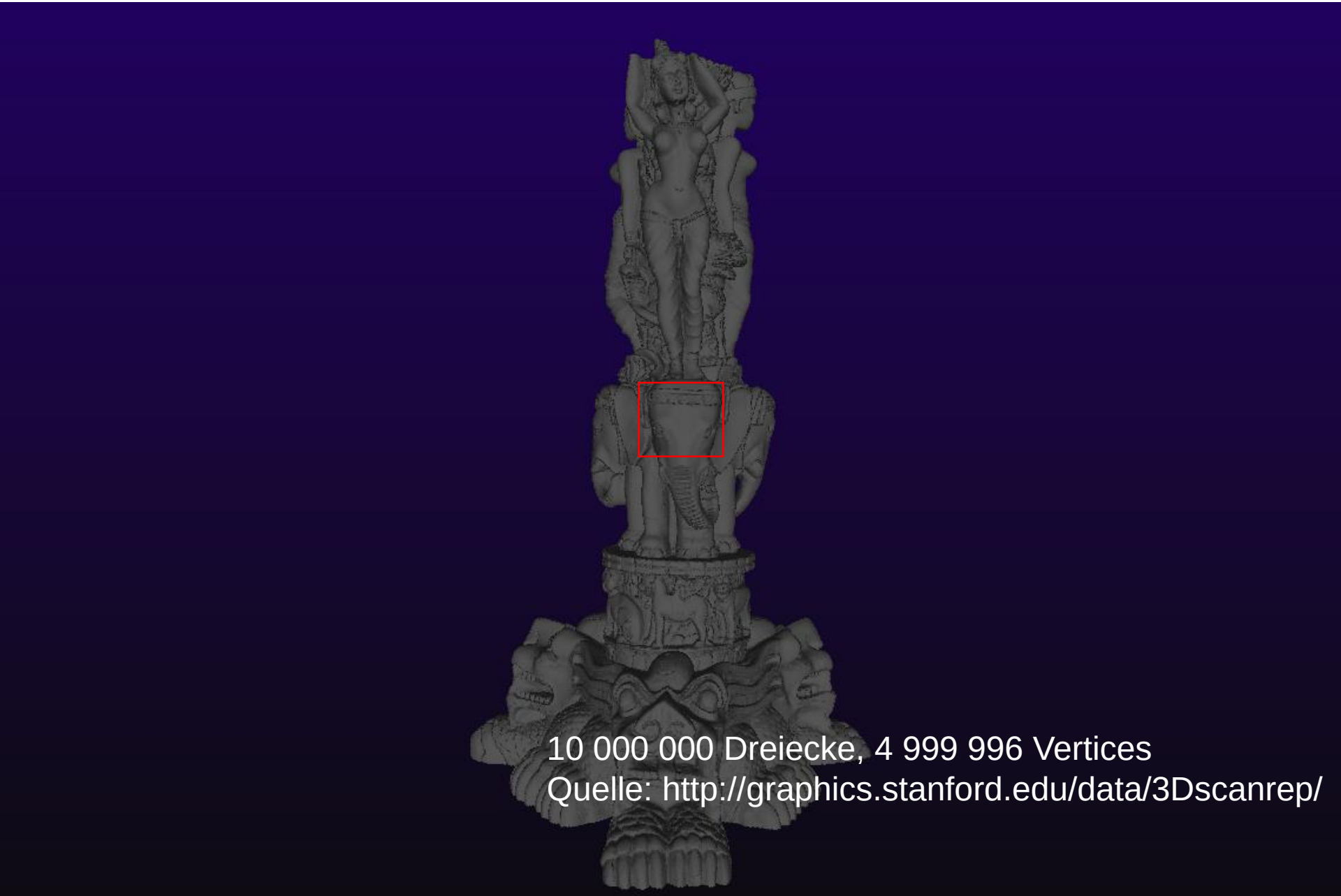
- Ottawa Convention Centre (Shaw Centre)

Ein kleines Dreiecksnetz

- 12 Dreiecke, 8 Vertices



Ein großes Dreiecksnetz



10 000 000 Dreiecke, 4 999 996 Vertices
Quelle: http://graphics.stanford.edu/data/3Dscanrep/

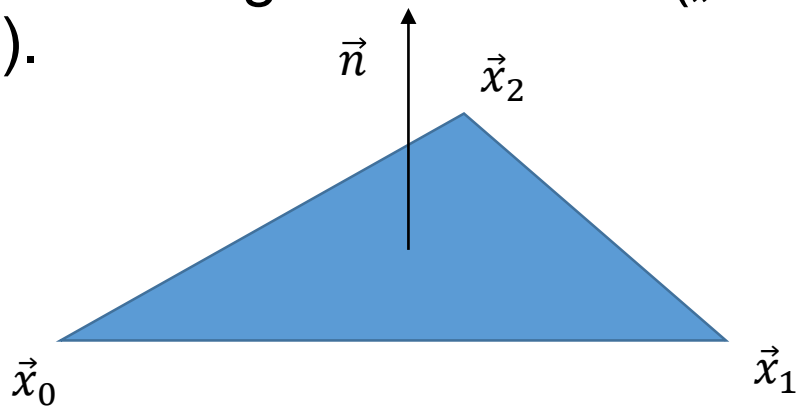
Ein großes Dreiecksnetz

10 000 000 Dreiecke, 4 999 996 Vertices
Quelle: http://graphics.stanford.edu/data/3Dscanrep/

Ein riesiges Dreiecksnetz

Dreiecke

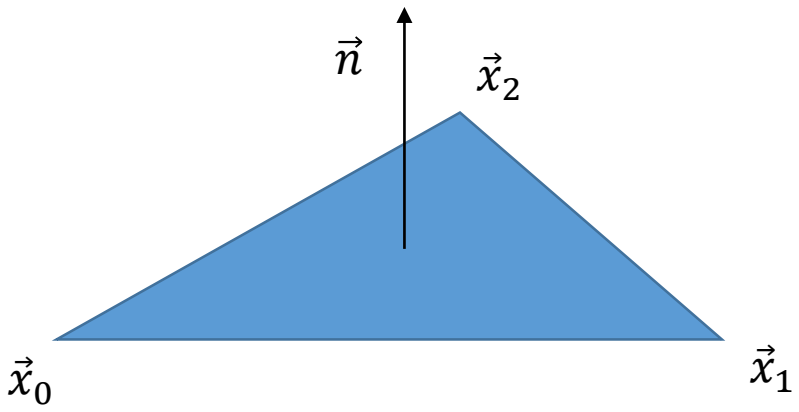
- Definiert durch drei Vertices $\vec{x}_0, \vec{x}_1, \vec{x}_2$
- Lebt in der Ebene, in der die drei Vertices liegen
- Umrandet von drei Kanten x_0x_1, x_1x_2, x_2x_0
- Normalenvektor der Ebene ist Normale $\vec{n}$ des Dreiecks
- Konventionen (in dieser Vorlesung; nicht universell gültig):
  - Von „außen“ oder „vorn“ betrachtet sind Vertices entgegen dem Uhrzeigersinn angeordnet
  - Oberflächennormale zeigt nach außen („outward-facing normals“).



Dreiecke

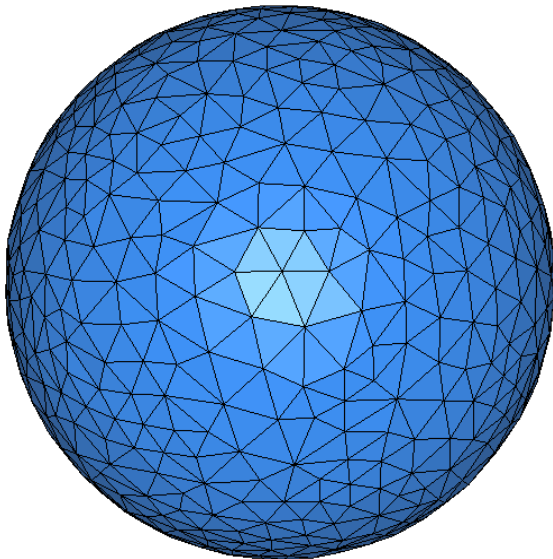
- Wie berechne ich die Normale $\vec{n}$, wenn $\vec{x}_0, \vec{x}_1, \vec{x}_2$ gegeben sind?

$$\vec{n} = \frac{(\vec{x}_1 - \vec{x}_0) \times (\vec{x}_2 - \vec{x}_0)}{|\vec{x}_1 - \vec{x}_0| \;\; |\vec{x}_2 - \vec{x}_0|}$$



Dreiecksnetz

- Eine Ansammlung von Dreiecken im dreidimensionalen Raum, die miteinander zu einer Oberfläche verbunden sind
- Geometrisch ist ein Dreiecksnetz eine stückweise ebene Oberfläche
  - Oberfläche *fast überall* eben
  - Ausnahme: Kanten zwischen Dreiecken
- Oft dienen Dreiecksnetze der stückweise ebenen Annäherung glatter Oberflächen
  - „Knicke“ zwischen Dreiecken sind dann unerwünschte Artefakte



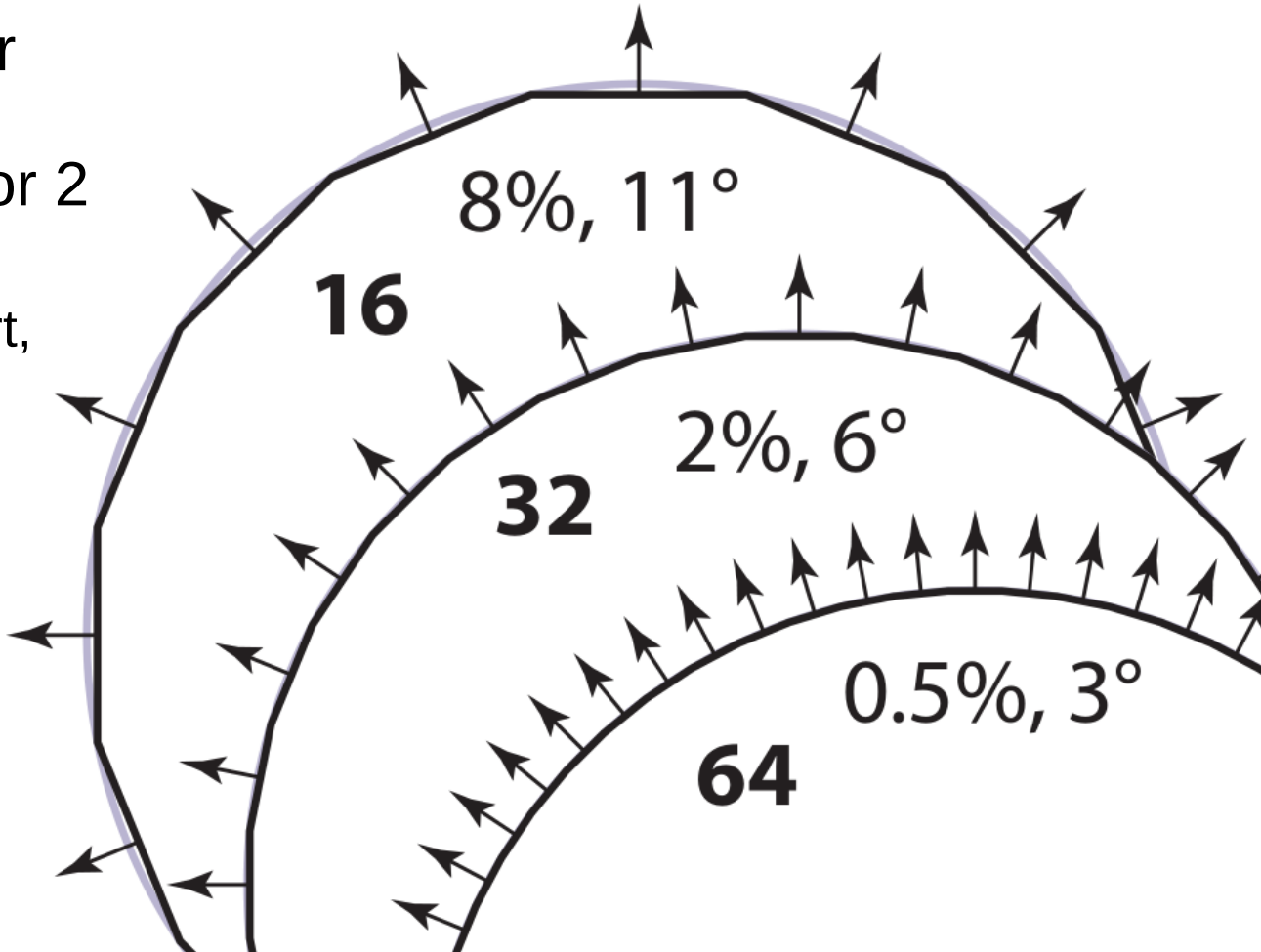
Dreiecksnetze als Annäherung glatter Flächen

Näherung eines Kreises mit zunehmender Segmentzahl

Maximaler Fehler der Position fällt um Faktor 4 mit jeder Verdopplung der Segmentzahl

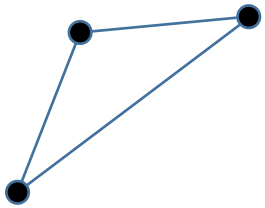
Maximaler Fehler der Normale fällt nur um Faktor 2

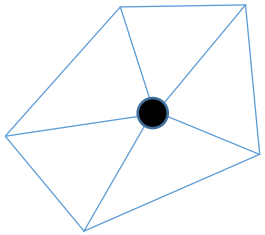
⇒ darum werden Normalen interpoliert, mehr hierzu später



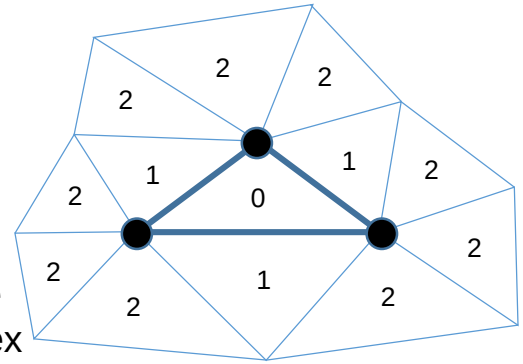
Repräsentation von Dreiecksnetzen

- Kompaktheit
- Effizienz für Rendering
  - Alle Dreiecke als Tripel von 3D-Punkten listen
- Effizienz für geometrische Abfragen, z.B. Nachbarschaftsverhältnisse:
  - Alle Vertices eines Dreiecks





  - Alle Dreiecke um einen gegebenen Vertex
  - Nachbardreiecke eines Dreiecks



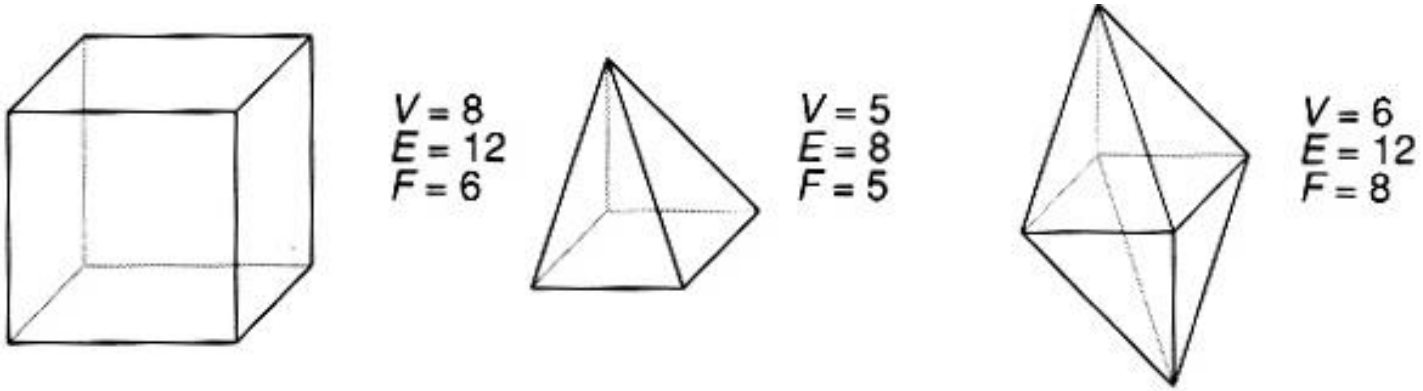
0 = Suchanfrage
1 = gemeinsame Kante
2 = gemeinsamer Vertex

  - (Weitere nach Anwendung)

Eulerscher Polyedersatz

Seien n_T, n_V, n_E Anzahl der Dreiecke, Vertices, Kanten

- Eulerscher Polyedersatz: $n_V - n_E + n_T = 2$ für eine einfache geschlossene Oberfläche

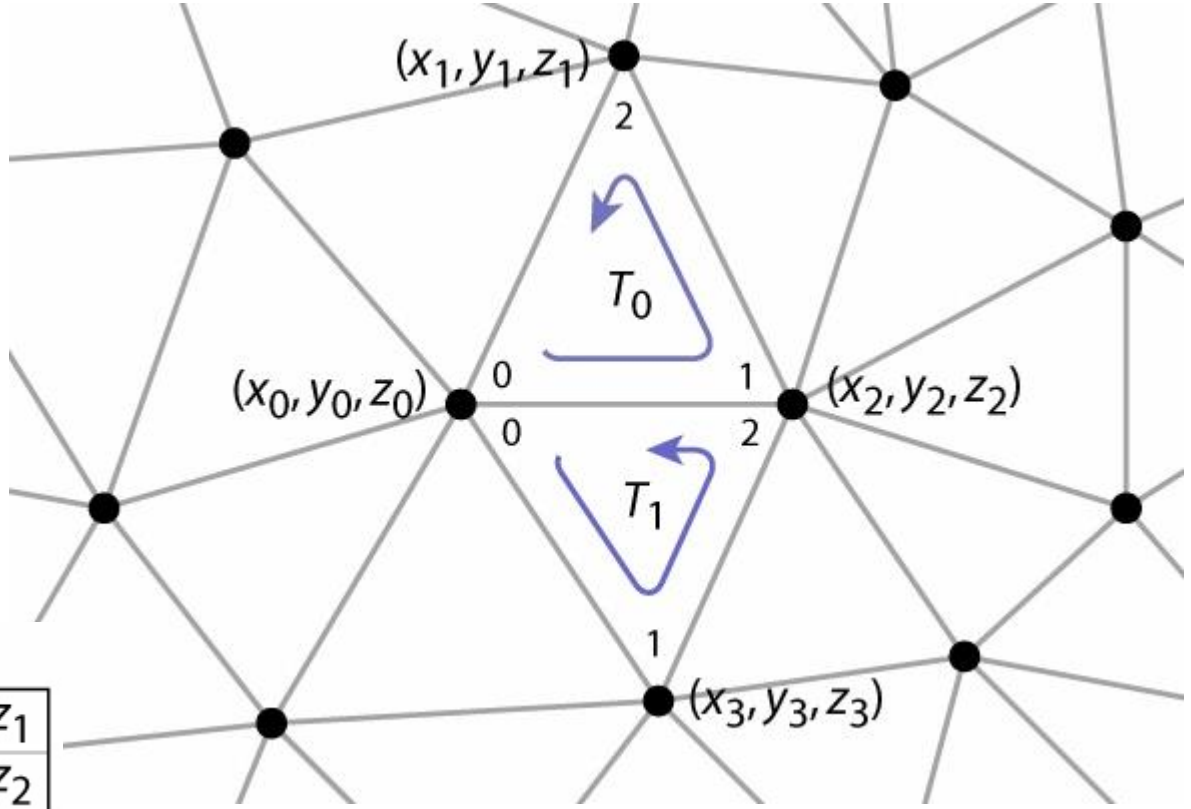


- Im Dreiecksnetz gehört jede Kante zu zwei Dreiecken, jedes Dreieck hat drei Kanten $\Rightarrow n_E = \frac{3}{2} n_T$
- $n_V - \frac{3}{2} n_T + n_T = n_V - \frac{n_T}{2} = 2$
  $\Rightarrow$ für große Netze $n_V \approx \frac{n_T}{2}$
- also $n_V : n_T : n_E$ etwa 1:2:3

Repräsentation von Dreiecksnetzen

- Einzelne Dreiecke
- Indizierte Dreiecksmenge
  - Vertices von mehreren Dreiecken verwendet
- Halbkanten-Datenstruktur
  - Nachbarschaftsabfragen in konstanter Zeit
- Triangle strips und fans
  - Kompressionsschemata für effiziente Übertragung

Einzelne Dreiecke



	[0]	[1]	[2]
tris[0]	x_0, y_0, z_0	x_2, y_2, z_2	x_1, y_1, z_1
tris[1]	x_0, y_0, z_0	x_3, y_3, z_3	x_2, y_2, z_2
	⋮	⋮	⋮

Einzelne Dreiecke

Array von Punkttripeln

- `float[nt][3][3]`
  - 3 Vertices pro Dreieck
  - 3 Koordinaten pro Vertex
  - 4 Bytes pro Koordinate

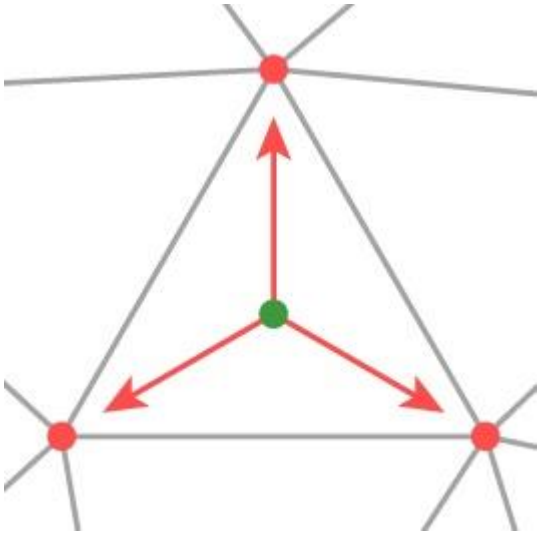
  ⇒ 36 Bytes pro Dreieck

- Verschwenderische, unpraktische Datenstruktur
  - In einem typischen Dreiecksnetz wird jeder Vertex ca. 6mal gespeichert
  - Nachbarsuche nur durch Koordinatenvergleich möglich
  - Bei Rundungsfehlern können Risse entstehen

Indizierte Dreiecke

- Speichere jeden Vertex einmal
- Jedes Dreieck zeigt auf seine drei Vertices

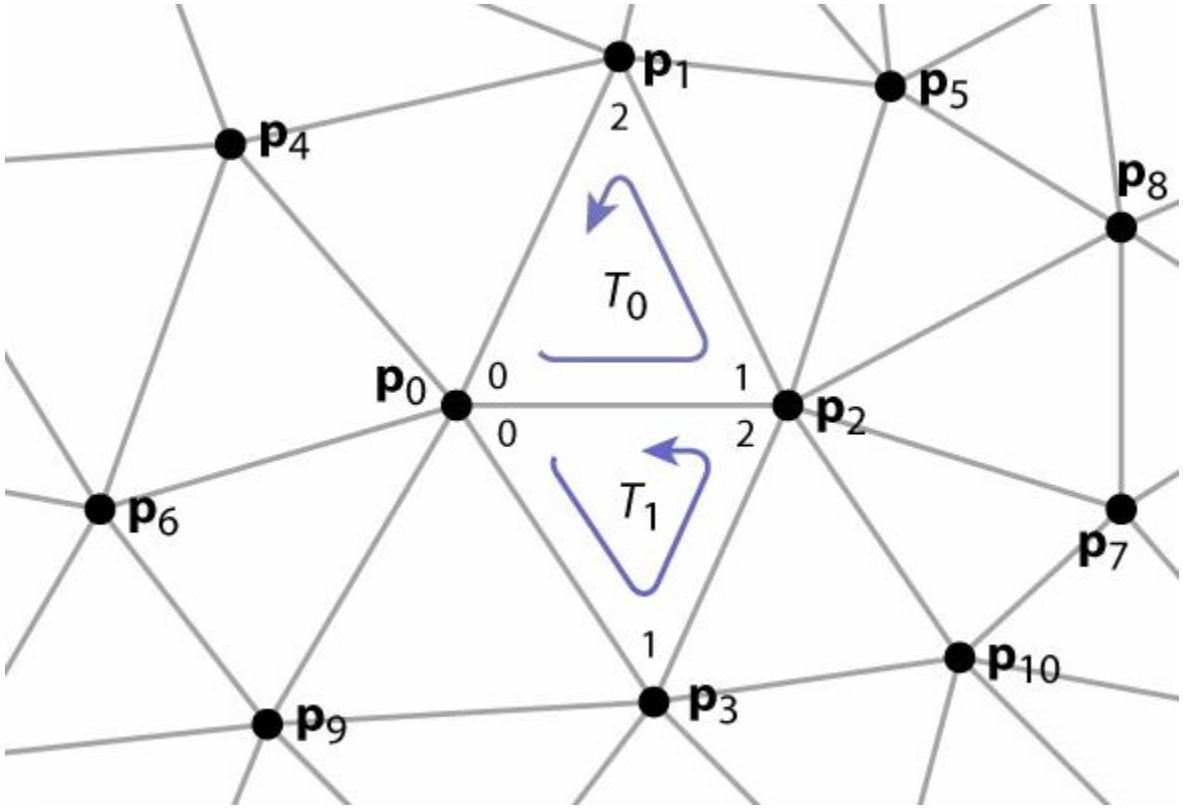
```
Mesh {
float verts[nv][3]; // vertex positions (or other data)
int tInd[nt][3]; // vertex indices
}
```



Indizierte Dreiecke

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

tInd[0]	0, 2, 1
tInd[1]	0, 3, 2
	⋮



Indizierte Dreiecke

Array der Vertices

- `float[nv][3]`: 12 Bytes pro Vertex
- Jeder Vertex gehört im Schnitt ca. 6 Dreiecken an, bzw. jedes Dreieck hat im Schnitt ca. einen halben „eigenen“ Vertex

Array von Vertextripeln

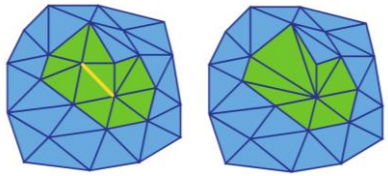
- `int[nt][3]`: 12 Bytes pro Dreieck

Speicherbedarf:

- je Dreieck 12 Bytes für Indices + ca. $^{12}/_{2}$ Bytes für Vertexkoordinaten –
  insgesamt ca. 18 Bytes / Dreieck

Indizierte Dreiecke

- **Nachbarschaftsabfrage** ist umständlich, aber wenigstens wohldefiniert
- Suche benachbarter Vertices, Kanten, Flächen ist $O(n_V)$



- Lokale Änderungen wie „edge collapse“ sind $O(n_V)$

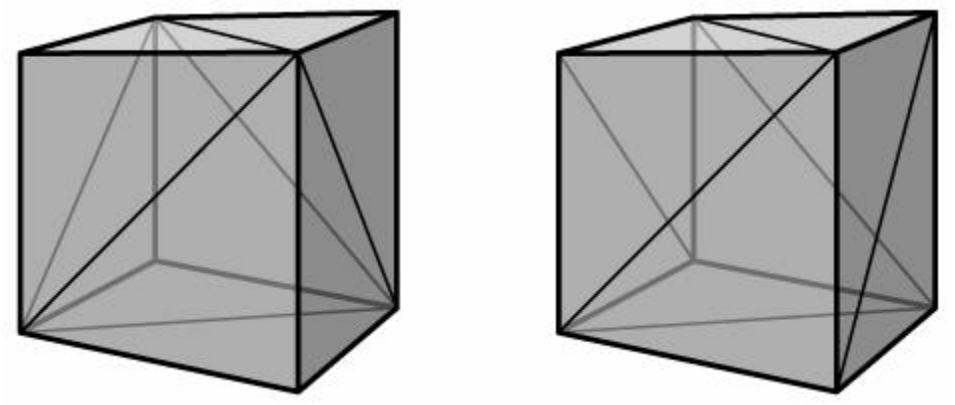
- **Topologie** und **Geometrie** sind getrennt repräsentiert

Zwei komplett verschiedene Konzepte:

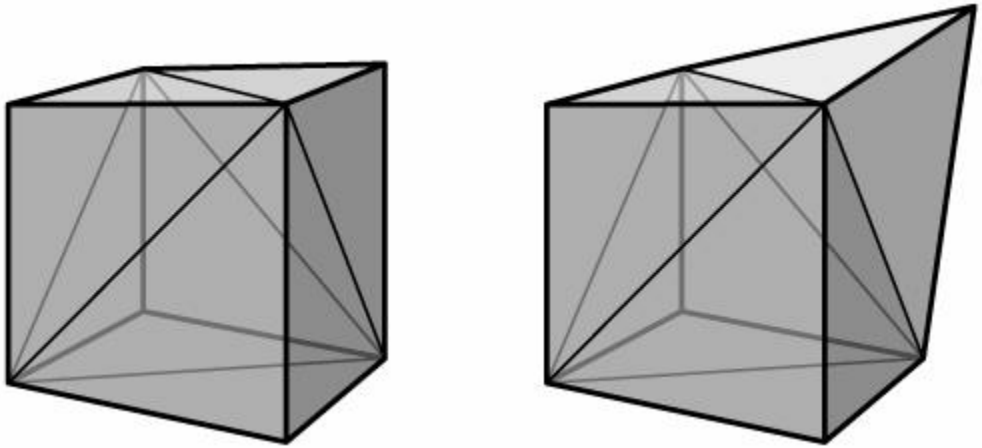
- Topologie: wie sind die Dreiecke verbunden? (unabhängig von Vertexpositionen)
- Geometrie: wo liegen die Dreiecke im 3D-Raum?

Topologie und Geometrie

- Gleiche Geometrie, verschiedene Topologie:

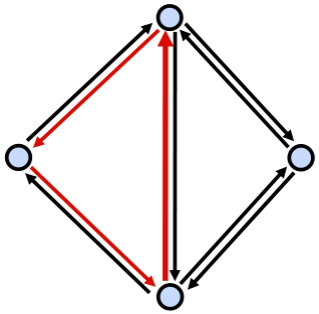


- Gleiche Topologie, verschiedene Geometrie:



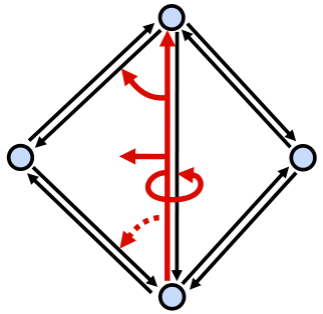
Halbkanten-Datenstruktur [Muller und Preparata 1978]

- Führe Orientierung in die Datenstruktur ein
  - Gerichtete Kanten



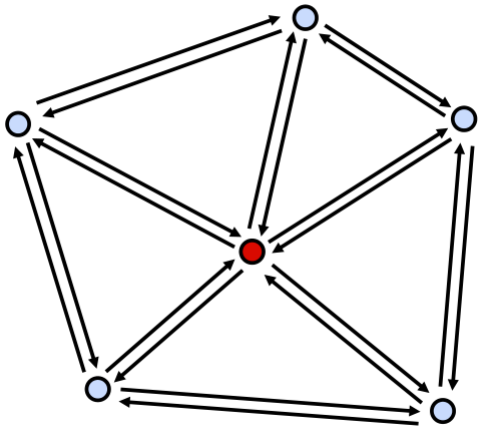
Halbkanten-Datenstruktur [Muller und Preparata 1978]

- Führe Orientierung in die Datenstruktur ein
  - Gerichtete Kanten
- Vertex:
  - Position
  - 1 Index einer abgehenden Halbkante
- Halbkante:
  - 1 Ursprungsvertex
  - 1 Index der zugehörigen Fläche
  - 3 Indizes zu nächster, voriger, Zwillingshalbkante
- Fläche:
  - 1 angrenzende Halbkante
- Leichte Suche, volle Konnektivität



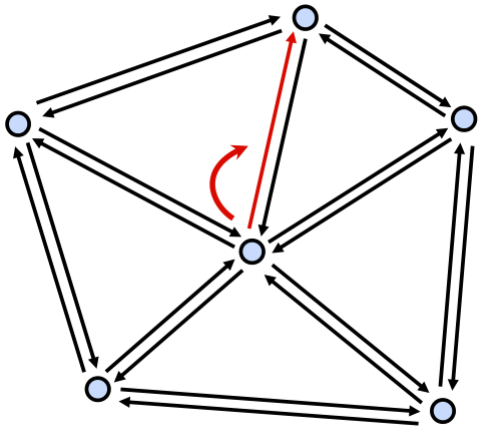
Suche eines Rings von Dreiecken

- Starte an einem Vertex



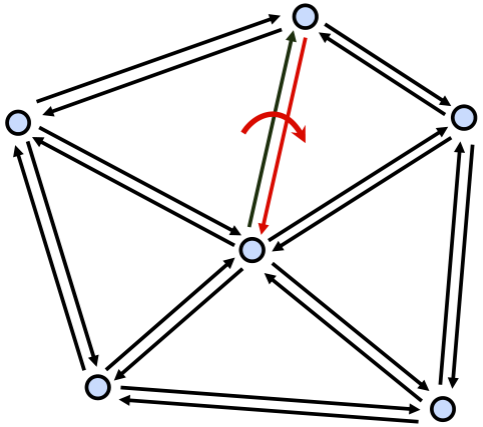
Suche eines Rings von Dreiecken

- Starte an einem Vertex
- Abgehende Halbkante



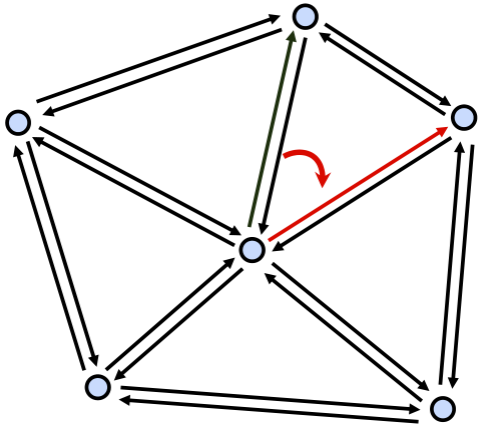
Suche eines Rings von Dreiecken

- Starte an einem Vertex
- Abgehende Halbkante
- Zwillingshalbkante



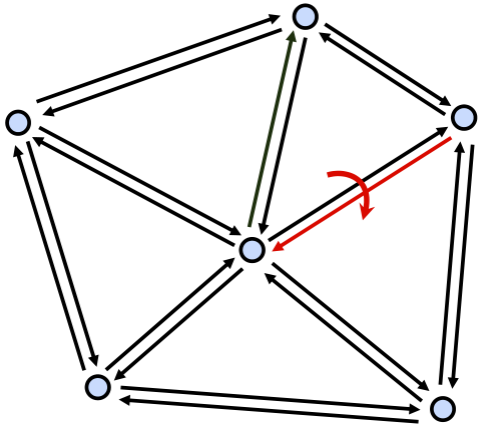
Suche eines Rings von Dreiecken

- Starte an einem Vertex
- Abgehende Halbkante
- Zwillingshalbkante
- Nächste Halbkante



Suche eines Rings von Dreiecken

- Starte an einem Vertex
- Abgehende Halbkante
- Zwillingshalbkante
- Nächste Halbkante
- Zwilling ...



Halbkanten-Datenstruktur

- Vorteile (angenommen, Vertices haben endliche Valenz)
  - $O(1)$ Zeit für Nachbarschaftsanfragen
  - $O(1)$ Zeit für lokale Änderungen (edge collapse, Vertex einfügen, usw.)

- Nachteile
  - Speicherintensiv – viele zusätzliche Zeiger nötig
  - Nicht ganz trivial mit OpenGL/VBOs zu rendern

Daten auf Dreiecksnetzen

- Oft benötigt man weitere Daten, die über die reine Geometrie hinausgehen

- Lege zusätzliche Daten an Flächen, Vertices oder Kanten ab

- Beispiele:
  - Farbe pro Fläche (für facettierte Objekte)
  - Informationen über scharfe Knicke an Kanten
  - Alles, was stetig veränderlich ist (ohne plötzliche Sprünge oder Unstetigkeiten) wird an Vertices gespeichert.

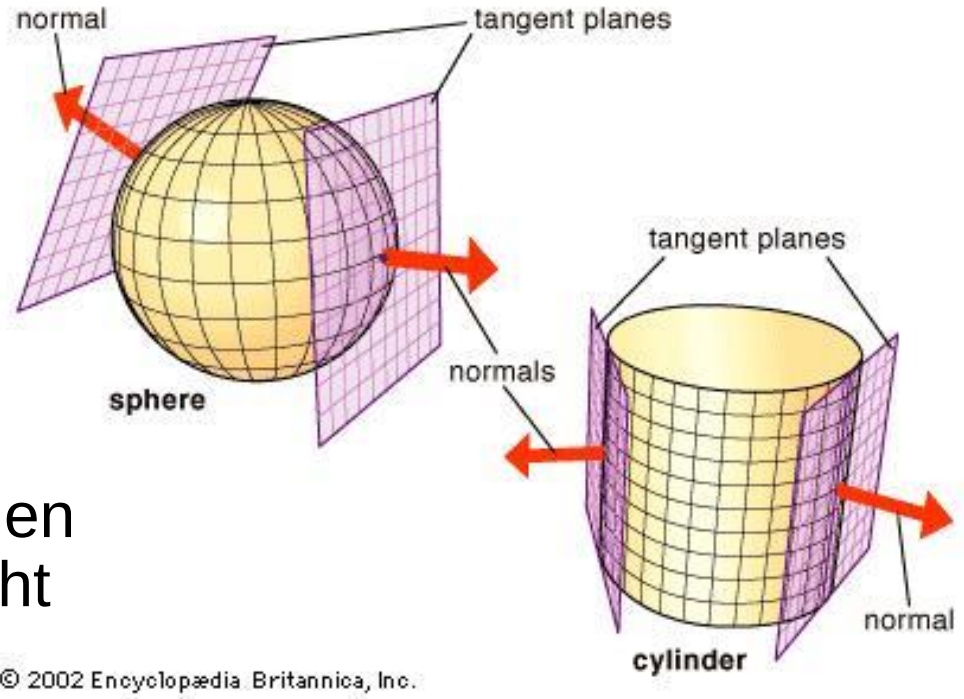
Typische Vertexdaten

- Oberflächennormalen
  - Wenn ein Netz eine glatte Oberfläche darstellen soll, speichere Normalenvektoren an den Vertices
- Texturkoordinaten
  - 2D-Koordinaten, die beschreiben, wie ein Bild auf dem Dreieck anzubringen ist
- Positionen
  - auch die Position ändert sich stetig zwischen Vertices, d.h. wenn man auf dem Dreieck entlangläuft

Normalen

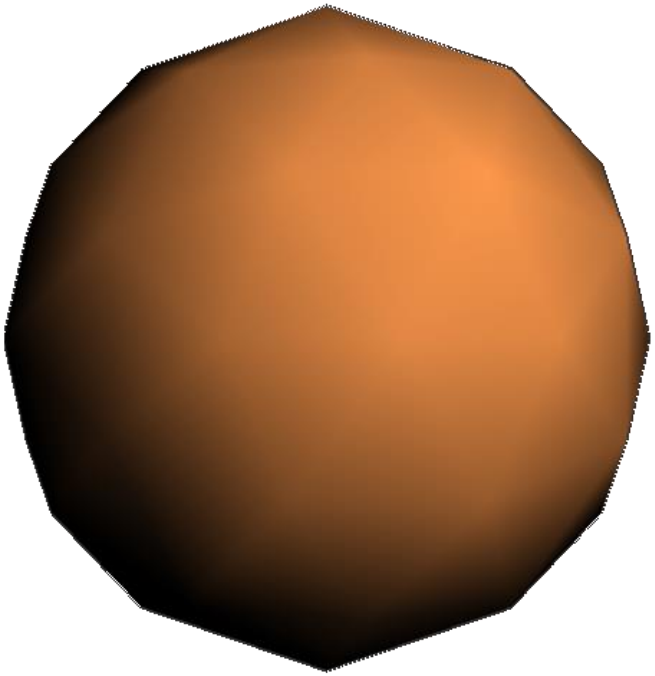
- Tangentialebene:
  - An einem Punkt auf einer glatten 3D-Oberfläche gibt es eine eindeutige Ebene, die die Oberfläche in der Umgebung des Punktes bestmöglich annähert (genannt Tangentialebene).

- Normalenvektor:
  - Vektor, der senkrecht auf der Oberfläche (und damit auf der Tangentialebene) steht
  - Nur für glatte Oberflächen eindeutig bestimmt (nicht an Ecken, Kanten, Spitzen, Rändern)

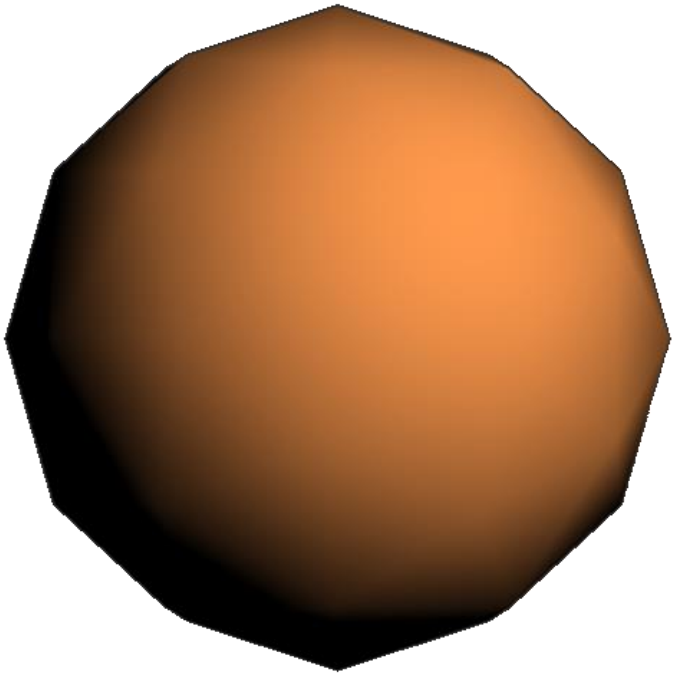


Vertexnormalen

- Dreiecksgitter als Annäherung an die „wirkliche“ Oberfläche
- -> speichere Normalenwerte an jedem Vertex, und *interpoliere*, um glatte, stetig veränderliche Normalen an jedem Punkt auf dem Dreieck zu haben



Gouraud-Interpolation
(Farbwerte)



Phong-Interpolation
(Normalenvektoren)

Gemischte (glatte und kantige) Flächen darstellen

Array der Vertices

- `float[nv][3]`

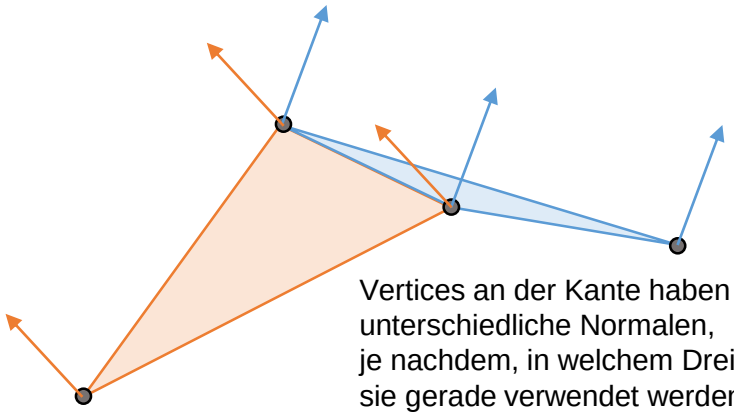
Array der Normalenvektoren

- `float[nn][3]`

Array von Vertex- und Normalenindices

- `int[nt][6]`: zeigt auf eine Position und eine Normale pro Vertex pro Dreieck

Ermöglicht Flächen
mit Knicken und Spitzen



Vertices an der Kante haben
unterschiedliche Normalen,
je nachdem, in welchem Dreieck
sie gerade verwendet werden

Datenformate

Wavefront OBJ:

```
mtllib material.mtl
v 0.062791 0.000000 -1.498027 #Vertices
v 0.062667 0.003943 -1.498027
v 0.062295 0.007870 -1.498027
v 0.061678 0.011766 -1.498027
⋮
vn 0.041879 0.000000 -0.999123 #Normals
vn 0.041796 0.002630 -0.999123
vn 0.041548 0.005249 -0.999123
vn 0.041137 0.007847 -0.999123
⋮
vt 0.000000 0.000000 #Texture coordinates
vt 0.010000 0.000000
vt 0.020000 0.000000
vt 0.030000 0.000000
⋮
usemtl material0
# Per vertex in a face, specify a pos, a tex coord, a normal.
f   1/  1/  1     2/  2/  2   103/103/103
f   1/  1/  1   103/103/103   102/102/102
f   2/  2/  2     3/  3/  3   104/104/104
```

Datenformate

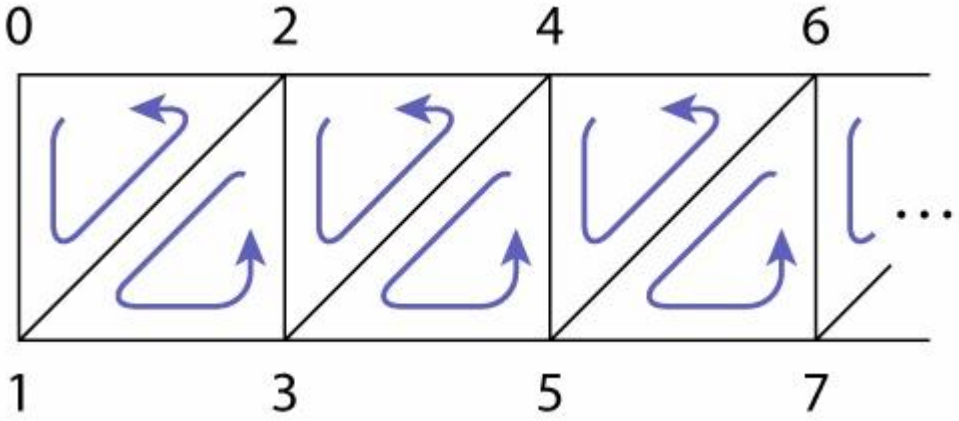
Stanford PLY:

- Viel flexibler als OBJ (definierbare *properties* und *elements* einschl. Kanten usw.)
- Daten können auch binär abgelegt werden
- Aber: weniger standardisiert; mehr Interpretationsspielraum beim Lesen der Daten

```
ply
format ascii 1.0          { ascii/binary, format version number }
element vertex 8          { define "vertex" element, 8 of them in file }
property float x          { vertex contains float "x" coordinate }
property float y          { y coordinate is also a vertex property }
property float z          { z coordinate, too }
element face 6            { there are 6 "face" elements in the file }
property list uchar int vertex_index { "vertex_index" = list of ints }
end_header                { delimits the end of the header }
0 0 0                     { start of vertex list }
0 0 1
0 1 1
⋮
3 0 1 2                   { start of face list }
3 0 2 3
3 0 4 1
⋮
```

Dreiecksstreifen (triangle strips)

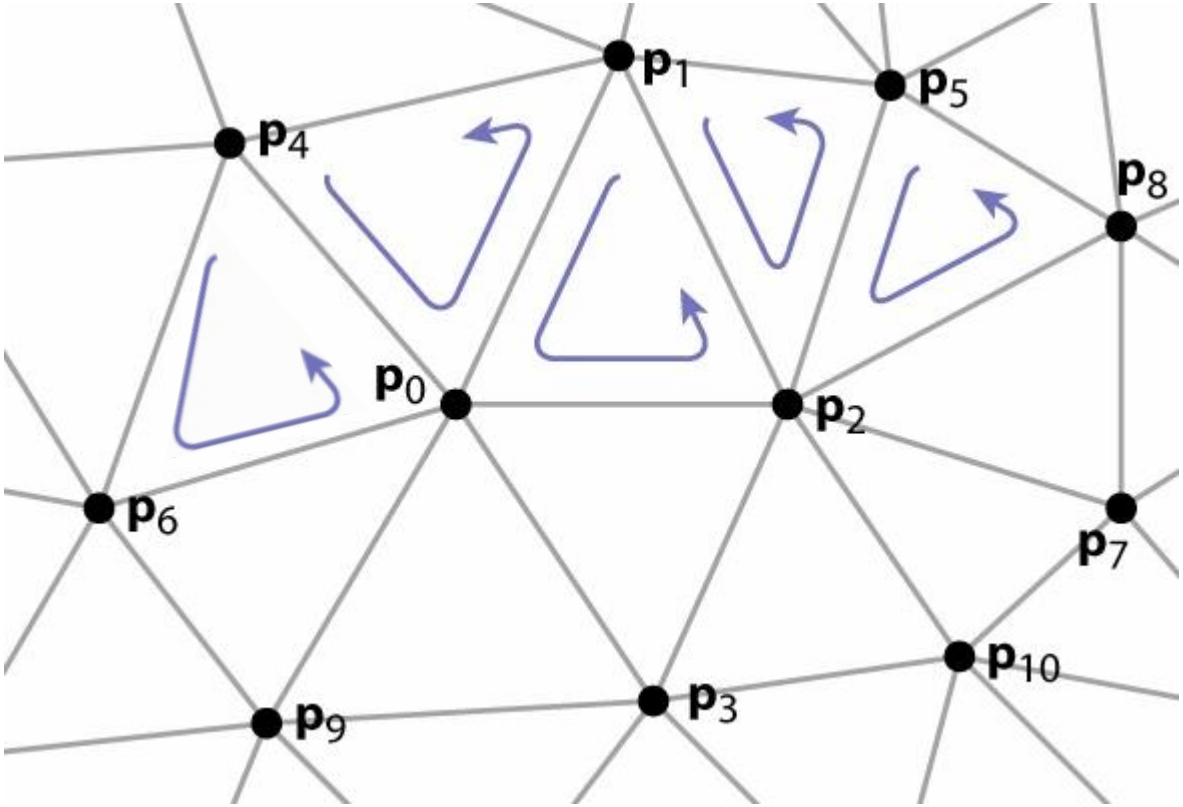
- Beobachtung: jedes Dreieck grenzt üblicherweise an das vorherige
- Jeder Vertex erzeugt ein Dreieck, indem die beiden vorherigen Vertices wiederverwendet werden
- Jede Folge von drei Vertices erzeugt ein Dreieck (aber nicht in derselben Reihenfolge)
- Z.B., 0,1,2,3,4,5,6,7 erzeugt Dreiecke (0 1 2),(2 1 3),(2 3 4),(4 3 5),(4,5,6),(6,5,7)
- Für lange Streifen konvergiert der Speicherbedarf gegen 1 Vertexreferenz pro Dreieck



Dreiecksstreifen

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

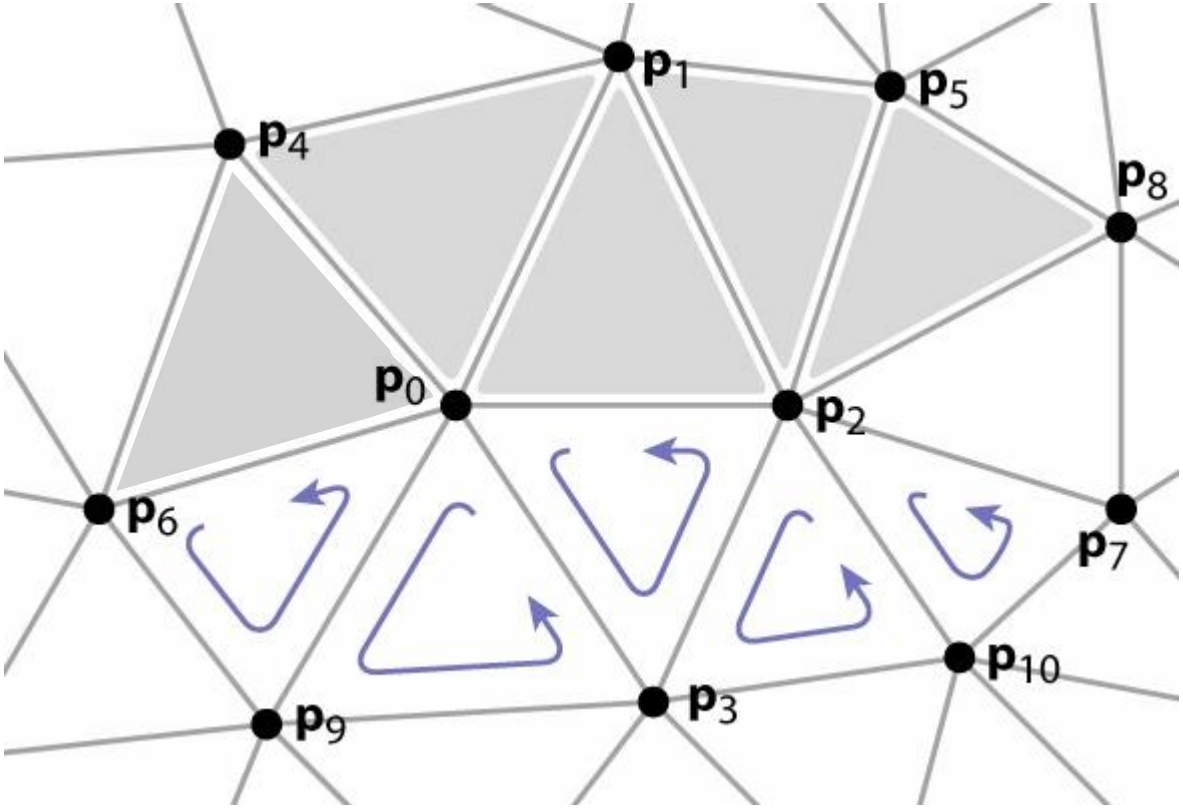
tStrip[0]	6, 4, 0, 1, 2, 5, 8
tStrip[1]	6, 9, 0, 3, 2, 10, 7
	⋮



Dreiecksstreifen

verts[0]	x_0, y_0, z_0
verts[1]	x_1, y_1, z_1
	x_2, y_2, z_2
	x_3, y_3, z_3
	⋮

tStrip[0]	6, 4, 0, 1, 2, 5, 8
tStrip[1]	6, 9, 0, 3, 2, 10, 7
	⋮



Trick: tStrips mit entarteten Dreiecken ermöglichen Sprünge/Unterbrechungen

```
tStrip = {6,4,0,1,2,5,8,8,6,6,9,0,3,2,10,7}
```

Dreiecksstreifen

Array der Vertices

- `float[nv][3]`: 12 Bytes pro Vertex

Array von Indexlisten

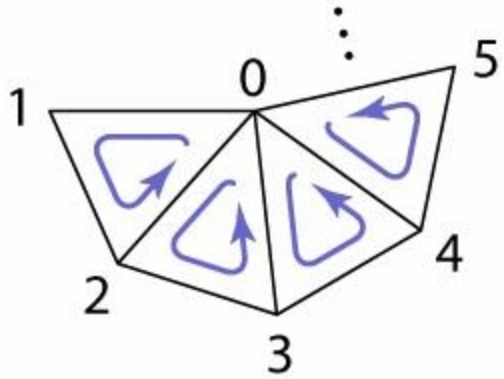
- `int[ns][`*`variable`*`]`: 2+n Indices pro Streifen
- im Schnitt, (1+ϵ) Indices pro Dreieck (Annahme: lange Streifen)
- ca. 4 Bytes pro Dreieck

Speicherbedarf:

- je Dreieck 4 Bytes für Indices + ca. $^{12}/_{2}$ Bytes für Vertexkoordinaten –
  insgesamt ca. 10 Bytes / Dreieck

Dreiecksfächer (triangle fans)

- Gleiche Idee, aber verwende älteste statt neuester Vertices wieder
- 0,1,2,3,4,5 erzeugt (0 1 2),(0 2 3),(0 3 4),(0 4 5)



- Speicherbetrachtungen siehe Dreiecksstreifen (jedoch werden Fächer in der Regel nicht so lang wie Streifen)

Gültigkeit von Dreiecksnetzen

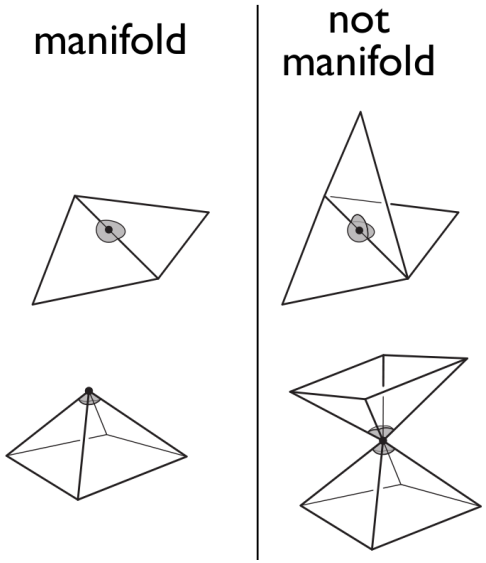
- Oft benötigen wir Netze, die einen Raum mathematisch wohldefiniert umschließen (z.B. fürs 3D-Drucken).
- Bei Geometrieverarbeitung benötigen wir Netze, die gewisse Annahmen der auf ihnen agierenden Algorithmen erfüllen

Zwei komplett verschiedene Konzepte:

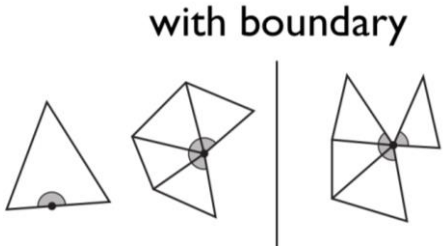
- **Topologie**: wie sind die Dreiecke verbunden? (unabhängig von Vertexpositionen)
- **Geometrie**: wo liegen die Dreiecke im 3D-Raum?

Topologische Gültigkeit

- Stärkste Eigenschaft: Mannigfaltigkeit (manifold)
  - Keine Punkte sollten „besonders“ sein
  - Jede Kante muss genau 2 Dreiecken angehören
  - Jeder Vertex muss eine umlaufende Schleife von Dreiecken haben



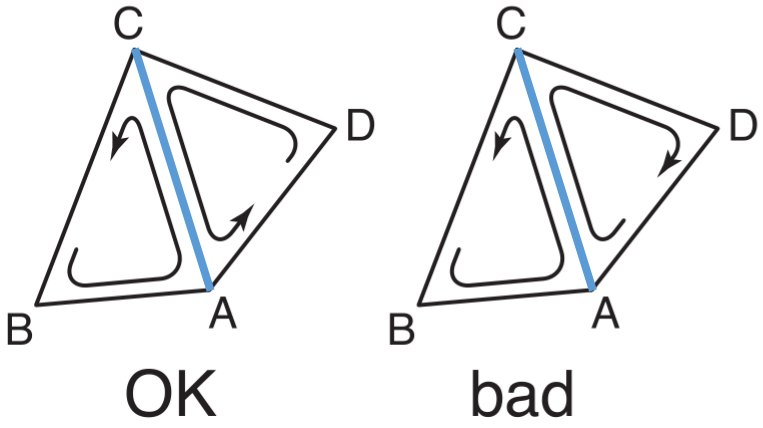
- Mannigfaltigkeit mit Rand
  - Gelockerte Regeln, um Flächen mit Rand zuzulassen

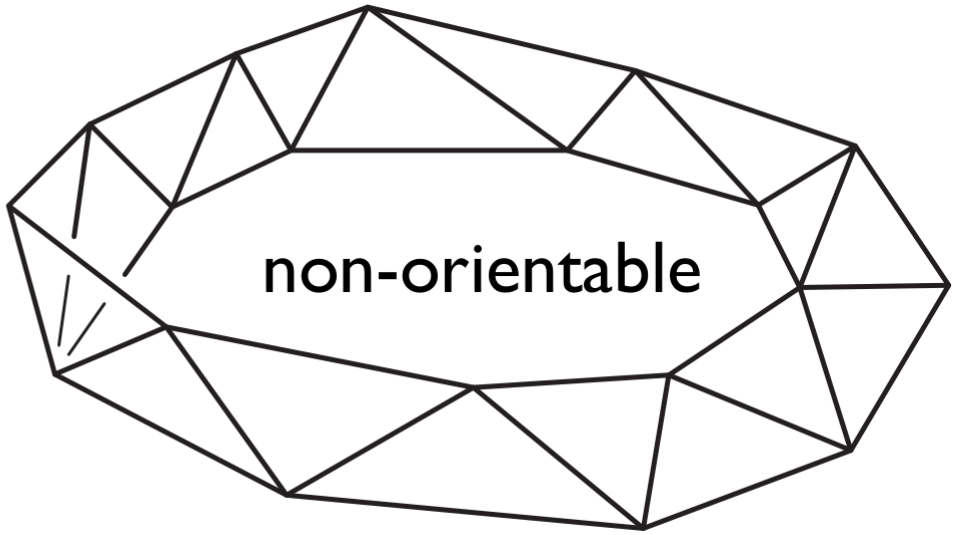


Topologische Gültigkeit

- Konsistente Orientierung
  - Welche Seite ist die Vorder- und welche die Rückseite der Oberfläche?
  - Regel: wenn wir die Vertices entgegen dem Uhrzeigersinn sehen, sind wir auf der Vorderseite.
  - In einem Dreiecksnetz sollten benachbarte Dreiecke gleichermaßen orientiert sein!
  - Dies ist aber nicht immer möglich.

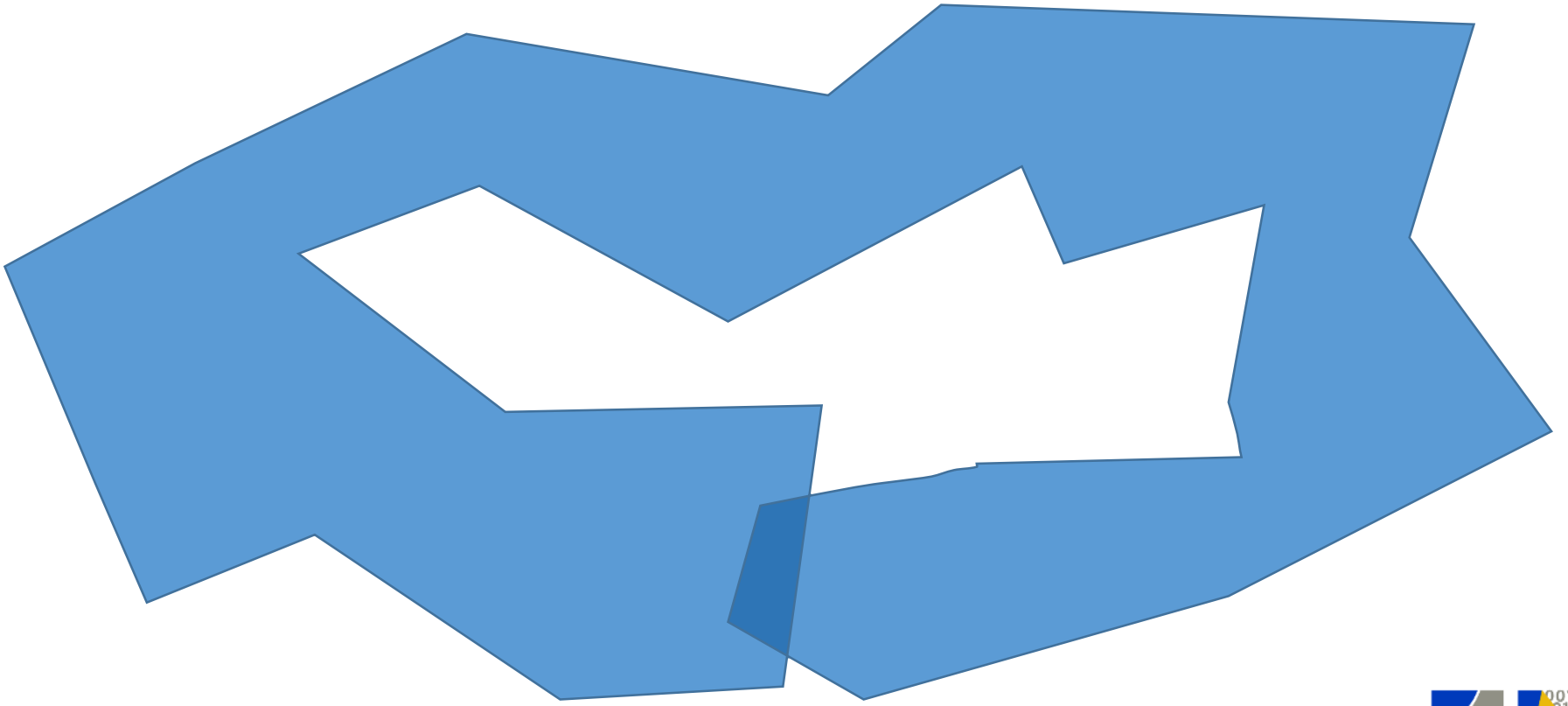
Prüfe Umlaufsinn der beiden Dreiecke an einer Kante





Geometrische Gültigkeit

- Oberfläche sollte sich nicht selbst schneiden
- im Allgemeinen schwer zu garantieren
- selbst weit entfernte Teile könnten sich schneiden



Bibliotheken

- **CGAL**
  - www.cgal.org
  - Sehr umfassende Library für Algorithmische Geometrie

- **OpenMesh**
  - www.openmesh.org
  - Verwendet Halbkanten-Datenstruktur