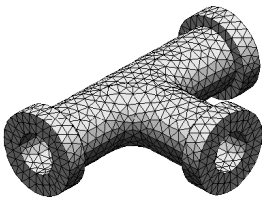
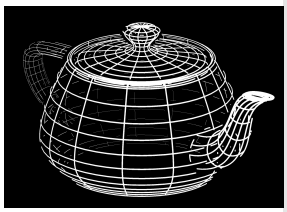
# Computer-Graphik II
## Randrepräsentationen für graphische Modelle

G. Zachmann
Clausthal University, Germany
cg.in.tu-clausthal.de

## Das Problem

- Wie werden diese Objekte gespeichert?

- Definition *Boundary-Representation* (*B-Rep*):
  Objekte "bestehen" aus
  1. Dreiecken, Quadraten und Polygonen (Geometrie)
  2. Nachbarschaftsbeziehungen ("Topologie", "connectivity")
- Im Gegensatz dazu gibt es auch Volumen- und Punkt-Repräsentationen

## Now all we need are some 3D models ...

**Nicht verwechseln!**

3D-Model

3D-Model

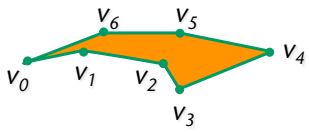


## Definitionen: Graphen

- Ein Graph ist ein Paar $G=(V, E)$, wobei $V=\{v_0,v_1,...,v_{n-1}\}$ eine nichtleere Menge von $n$ verschiedene Knoten (Punkten, Vertices) und $E$ eine Menge von Kanten $(v_i, v_j)$ ist.
- Ist V (diskrete) Teilmenge von $\mathbb{R}^d$ mit $d \geq 2$, so heißt $G=(V, E)$ ein geometrischer Graph.
- Zwei Kanten/Knoten heißen benachbart oder adjazent, falls sie einen Knoten / eine Kante gemeinsam haben.
- Ist $e=(v_i, v_j)$ eine Kante in $G$, so heißen $e$ und $v_i$ inzident zueinander (dito $e$ und $v_j$);
  $v_i$ und $v_j$ heißen benachbart oder adjazent.
- Kanten sind für unsere Zwecke nicht gerichtet (zunächst und im Prinzip) und werden oft einfacher mit $v_iv_j$ bezeichnet.

## Polygon

- Ein geometrischer Graph P=(*V, E*), wobei $V=\{v_0,v_1,...,v_{n-1}\} \subset \mathbb{R}^d$, $d \geq 2$, und $E=\{ (v_0,v_1), ..., (v_{n-1}, v_0) \}$ heißt Polygon.
- Die Knoten heißen auch Punkte oder Ecken oder Vertices.

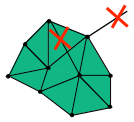


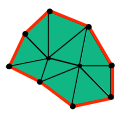

- Ein Polygon heißt
  - eben, falls alle Vertices in einer Ebene liegen;
  - einfach, falls der Schnitt jeweils zweier Kanten aus *E* leer oder ein Punkt aus *V* ist und jeder Eckpunkt nur zu höchstens zwei Kanten gehört (d.h. das Polygon sich selbst nicht schneidet).
- Per Definition betrachten wir nur geschlossene Polygone

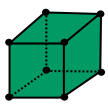


## Mesh (Polygonnetz)

- Sei *M* eine endliche Menge von geschlossenen, einfachen Polygonen $P_i$; sei $V = \bigcup_i V_i \qquad E = \bigcup_i E_i$
- M heißt Mesh gdw.
  - der Schnitt zweier Polygone aus *M* ist entweder leer, ein Punkt $v \in V$ oder eine Kante $e \in E$; und
  - jede Kante $e \in E$ gehört zu mindestens einem Polygon
- Die Menge aller Kanten, die nur zu einem einzigen Polygon gehören, heißt Rand des Meshes
- Ist der Rand des Meshes leer, dann heißt das Mesh geschlossen
- Die Menge aller Punkte *V* und Kanten *E* eines Meshes bilden wieder einen Graphen

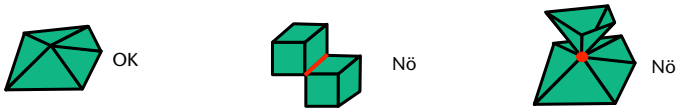






## Definition: Polyeder

- Ein Mesh heißt Polyeder, falls
  1. jede Kante $e \in E$ inzident ist zu genau zwei Polygonen (das Mesh ist geschlossen); und
  2. keine Teilmenge des Meshes erfüllt Bedingung 1.

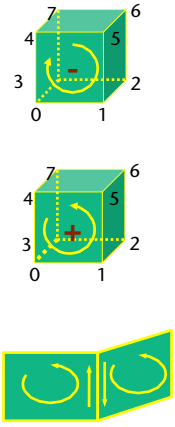



- Die Polygone werden auch als Facetten (*facets* / *faces*) bezeichnet
- Satz:
  Jeder Polyeder P teilt den Raum in drei Gebiete: seine Oberfläche, sein Inneres und sein Äußeres.



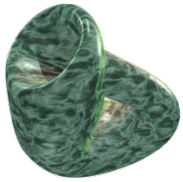
## Orientierbarkeit

- Jede Facette eines Meshes kann durch die Definition eines Umlaufsinns orientiert werden.
  - Jede Facette kann auf genau zwei Arten orientiert werden
- Zwei Facetten eines Meshes, die längs einer Kante benachbart sind, heißen gleichorientiert, falls die Kante entgegengesetzt durchlaufen wird, wenn man die beiden Facetten entlang ihrer Orientierung "abwandert".
- Die Orientierung bestimmt die Richtung der Oberflächennormale einer Facette. Dazu wird in der Regel die Rechte-Hand-Regel verwendet.



- Ein Mesh heißt orientierbar, wenn alle Facetten des Meshes so orientiert werden können, daß je zwei benachbarte Facetten gleichorientiert sind.
  - Besitzen alle Facetten eine Orientierung mit dieser Eigenschaft, so heißt das Mesh orientiert.
- Ein Mesh heißt nicht orientierbar, falls bei jeder Wahl von Orientierungen der Facetten mindestens zwei benachbarte Facetten nicht gleichorientiert sind.

- Bemerkungen:
  - Jede im dreidimensionalen Raum eingebettete, geschlossene, nicht orientierbare Fläche besitzt eine Selbstdurchdringung.
  - Die Oberfläche eines Polyeders ist stets orientierbar

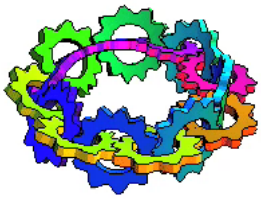




# Exkurs: das Möbiusband in der Kunst

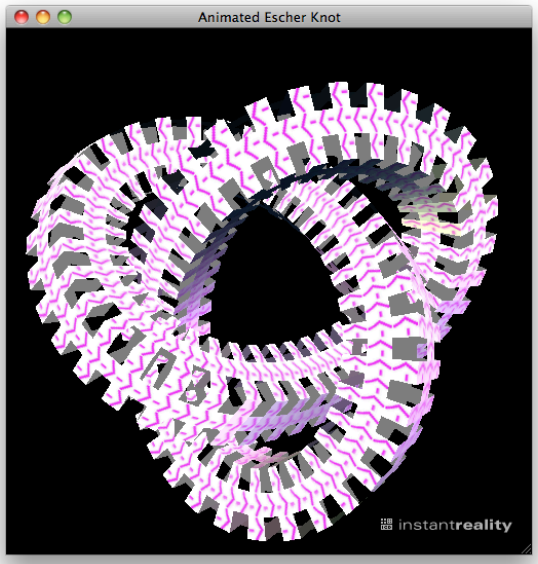
*Möbius Strip II*, woodcut, 1963

Max Bill



*Interlocked Gears*,
Michael Trott, 2001

## Ist der Escher-Knoten ein orientierbares Mesh oder nicht?



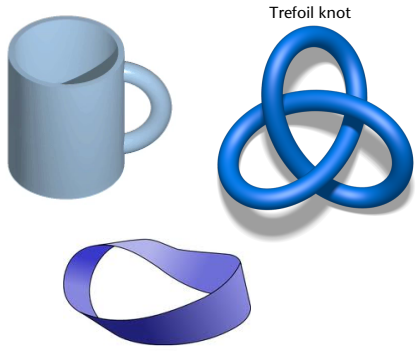

http://homepages.sover.net/~tlongtin



## Definition: Homöomorphismus

- Homöomorphismus = bijektive, stetige Abbildung zwischen zwei "Objekten" (eigtl.: topologischen Räumen, z.B. Flächen), deren Umkehrabbildung wieder stetig ist
- Achtung: nicht verwechseln mit Homomorphismus oder Homotopie!
- Veranschaulichung:
  - Stauchen, Dehnen, Verdrehen ist erlaubt, aber nicht z.B. Loch machen
  - Schneiden ist nur erlaubt, wenn das Objekt genau an der selben Stelle wieder "zusammengeklebt" wird
- Zwei Objekte heißen homöomorph gdw. es einen Homöomorphismus zwischen beiden gibt

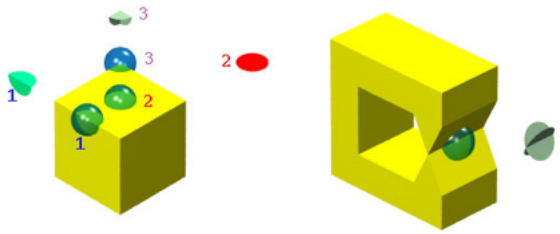
- Homöomorphe Objekte heißen auch topologisch äquivalent
- Beispiele:
  - Kreisscheibe und Quadrat
  - Tasse und Torus
  - Objekt und sein Spiegelbild
  - *Trefoil knot* und .... ?
  - Der Rand des Möbiusbandes und ... ?






## Definition: Zwei-Mannigfaltigkeiten (*2-manifolds*)

- Definition: Eine Fläche heißt zwei-mannigfaltig gdw. zu jedem Punkt der Fläche eine offene Kugel existiert, so daß der Schnitt zwischen Kugel und Fläche topologisch äquivalent zu einer Kreisscheibe ist.
- Beispiele:



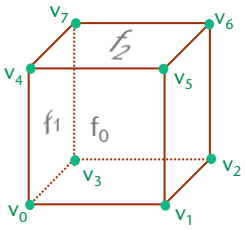

- Achtung: in der CG wird fast immer der Begriff "manifold" verwendet, wenn 2-manifold gemeint ist!
- Der Begriff "*piecewise linear manifold*" wird manchmal verwendet, um ein Mesh zu bezeichnen ...

# Datenstrukturen für Meshes

- Die naivste Datenstruktur:
  - Array von Polygonen; jedes Polygon = Array von Vertices
  - Beispiel:

| *face[0]* = | *face[1]* = | *face[2]* = |
|---|---|---|
| $x_0\ y_0\ z_0$ | $x_0\ y_0\ z_0$ | $x_4\ y_4\ z_4$ |
| $x_1\ y_1\ z_1$ | $x_4\ y_4\ z_4$ | $x_5\ y_5\ z_5$ |
| $x_5\ y_5\ z_5$ | $x_7\ y_7\ z_7$ | $x_6\ y_6\ z_6$ |
| $x_4\ y_4\ z_4$ | $x_3\ y_3\ z_3$ | $x_7\ y_7\ z_7$ |

...




- Probleme:
  - Vertices kommen mehrfach vor!
    - Speicherverschwendung, Problem bei Animation, ...
  - Wie findet man zu einem geg. Vertex alle Faces, in denen er vorkommt?
  - Verschieden große Arrays für verschieden lange Polygone

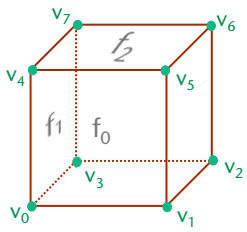


# Das *Indexed Face Set*

- Idee: gemeinsamer "Vertex-Pool" (*shared vertices*)
- Beispiel:

| *vertices* = |
|---|
| $x_0\ y_0\ z_0$ |
| $x_1\ y_1\ z_1$ |
| $x_2\ y_2\ z_2$ |
| $x_3\ y_3\ z_3$ |
| ... |

| *face* | *vertex index* |
|---|---|
| *0* | 0, 1, 5, 4 |
| *1* | 0, 3, 7, 4 |
| *2* | 4, 5, 6, 7 |
| ... | |



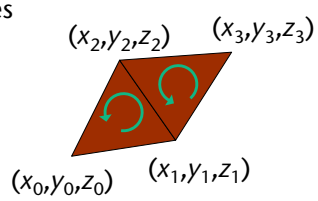

- Vorteil: großer Speichergewinn
  - 1 Vertex = 1 Punkt + 1 Vektor (V.-Normale) + uv-Texturkoord. = 32 Bytes
  - 1 Index = 1 Integer = 4 Bytes
- Vorteil bei Deformationen des Objektes
- Wahrscheinlich die häufigste Datenstruktur

- Probleme:
  - Kanten werden (implizit) 2x gespeichert
  - Immer noch keine Nachbarschaftsinformationen
- Folge:
  - Alle zu einem Vertex inzidenten Facetten zu finden kostet $O(n)$ Zeit, wobei $n$ = # Vertices im Mesh
  - Dito für alle zu einem Vertex adjazenten Vertices
  - Komplette Breitensuche kostet $O(n^2)$ Zeit
    (Mit einer Breitensuche kann z.B. festgestellt werden, ob es sich um ein geschlossenes Objekt handelt)



## Das OBJ-Fileformat

- OBJ = Indexed Face Set + weitere Features
- Zeilenbasiertes ASCII-Format
- Geordnete Liste von Vertices:
  - Eingeleitet durch "v"
  - Räumliche Koordinaten x, y, z
  - Index durch die Reihenfolge gegeben
- Polygonliste:
  - Polygon wird eingeleitet durch "f"
  - Geordnete Liste von Vertex-Indizes
  - Länge einer Liste = # der Seiten
  - Orientierung durch die Reihenfolge gegeben
- Im Prinzip können "v" und "f" beliebig gemischt werden

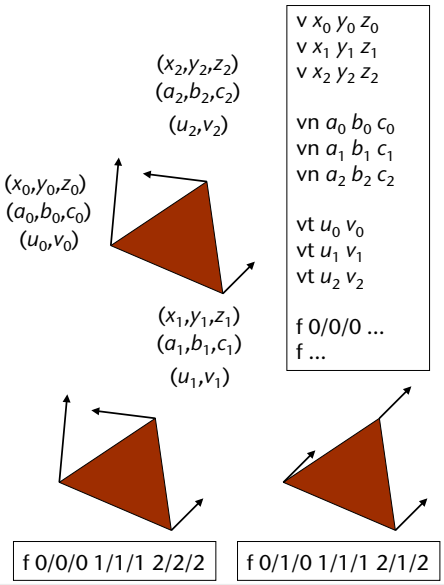



```
v x0 y0 z0
v x1 y1 z1
v x2 y2 z2
v x3 y3 z3

f 0 1 2
f 1 3 2
```

## Weitere Attribute

- Vertex-Normalen:
  - Präfix mit "vn"
  - Enthält x, y, z der Normalen
  - Nicht notw.weise in Einheitslänge
  - Nicht zwingend in Vertex-Reihenfolge
  - Indizes wie bei den Vertices
- Texturkoordinaten:
  - Präfix mit "vt"
  - Nicht zwingend in Vertices-Reihenfolge
  - Enthält u,v-Texturkoordinaten
- Polygone:
  - Benutzt "/" um die Indizes voneinander abzugrenzen
  - Vertex / Normale / Textur
  - Normale und Textur sind optional
  - Die Normale kann durch "//" eliminiert werden






f 0/0/0 1/1/1 2/2/2

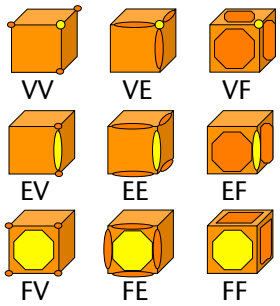


f 0/1/0 1/1/1 2/1/2

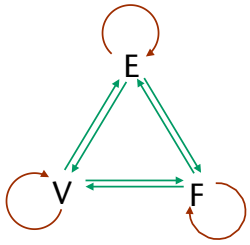


## Nachbarschaftsbeziehungen (Weiler 1985)

| | gegeben | gesucht ("alle benachbarte ..") | Abb |
|---|---|---|---|
| 1 | Vertex | Vertices | V → V |
| 2 | Vertex | Kanten | V → E |
| 3 | Vertex | Facetten | V → F |
| 4 | Kante | Vertices | E → V |
| 5 | Kante | Kanten | E → E |
| 6 | Kante | Facetten | E → F |
| 7 | Facette | Vertices | F → V |
| 8 | Facette | Kanten | F → E |
| 9 | Facette | Facetten | F → F |




Abstrakte Notation einer DS mit Nachbarschaftsbeziehungen:
Pfeile geben Inzidenz-/Adjazenz-Daten an

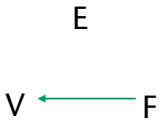


- Beispiel: Indexed Face Set

| *vertices* |
|---|
| $x_0\ y_0\ z_0$ |
| $x_1\ y_1\ z_1$ |
| $x_2\ y_2\ z_2$ |
| $x_3\ y_3\ z_3$ |
| . . . |

| *face* | *vertex index* |
|---|---|
| *0* | 0, 1, 5, 4 |
| *1* | 0, 3, 7, 4 |
| *2* | 4, 5, 6, 7 |
| . . . | |

=

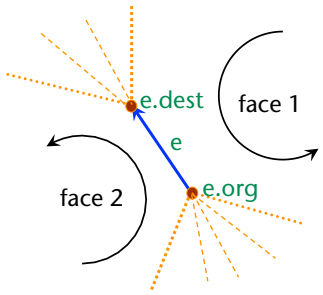



- Frage: welches ist die minimale Datenstruktur, die alle Nachbarschaftsbeziehungen in Zeit O(1) liefern kann?



## Die Winged-Edge-Datenstruktur [Baumgart '74]

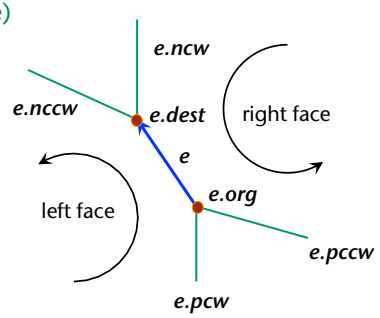
- Idee: kantenbasierte Datenstruktur (im Gegensatz zu Face-basiert)
- Beobachtungen:
  - Eine Kante speichert zwei Indizes auf 2 Vertices: ***e.org , e.dest***
    → ergibt eine Orientierung der Kante
  - In einem (orientierten) Polyeder grenzt eine Kante an genau 2 Facetten
  - Eine dieser Facetten ist gleichorientiert wie die Kante, die andere entgegen der Kante



## Die Datenstruktur

- Jede Kante hat 4 Zeiger auf 4 adjazente Kanten:
  1. ***e.ncw*** = Kante inzident zu ***e.dest*** und inzident zu *right face* (next clockwise)
  2. ***e.nccw*** = Kante inzident zu ***e.dest*** und inzident zu *left face* (next counter-clockwise)
  3./4. ***e.pcw*** / ***e.pccw*** = Kante inzident zu ***e.org*** und inzident zu *left* / *right face* (previous counter-/clockwise)

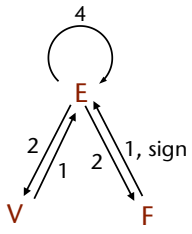
- Beobachtung: sind die Facetten konsistent orientiert, kommt jede Kante einmal "+" und einmal "-" vor.



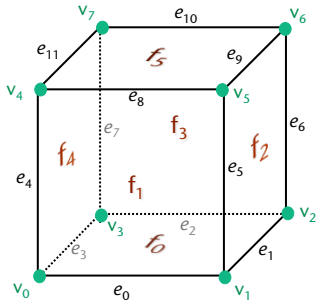



- Zusätzlich:
  - Jede Kante speichert je einen Zeiger auf die linke bzw. rechte Facette (***e.lf, e.rf***)
  - Jede Facette speichert 1 Zeiger auf eine beliebige inzidente Kante
  - Jeder Vertex speichert 1 Zeiger auf eine beliebige inzidente Kante

- Abstrakte Repräsentation der Datenstruktur:



## Beispiel




**Vertexliste**

| *v* | *coord* | | | *e* |
|---|---|---|---|---|
| 0 | 0.0 | 0.0 | 0.0 | 0 |
| 1 | 1.0 | 0.0 | 0.0 | 1 |
| 2 | 1.0 | 1.0 | 0.0 | 2 |
| 3 | 0.0 | 1.0 | 0.0 | 3 |
| 4 | 0.0 | 0.0 | 1.0 | 8 |
| 5 | 1.0 | 0.0 | 1.0 | 9 |
| 6 | 1.0 | 1.0 | 1.0 | 10 |
| 7 | 0.0 | 1.0 | 1.0 | 11 |

**Facetten**

| | | |
|---|---|---|
| 0 | e0 | - |
| 1 | e8 | - |
| 2 | e5 | - |
| 3 | e6 | - |
| 4 | e11 | - |
| 5 | e8 | + |

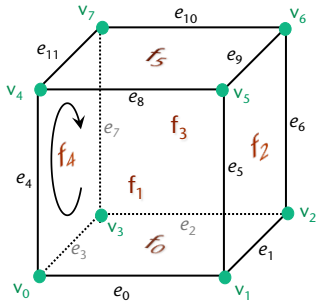
**Kantenliste**

| *e* | *org* | *dest* | *ncw* | *nccw* | *pcw* | *pccw* | *lf* | *rf* |
|---|---|---|---|---|---|---|---|---|
| 0 | v0 | v1 | e1 | e5 | e4 | e3 | f1 | f0 |
| 1 | v1 | v2 | e2 | e6 | e5 | e0 | f2 | f0 |
| 2 | v2 | v3 | e3 | e7 | e6 | e1 | f3 | f0 |
| 3 | v3 | v0 | e0 | e4 | e2 | e7 | f4 | f0 |
| 4 | v0 | v4 | e8 | e11 | e0 | e3 | f4 | f1 |
| 5 | v1 | v5 | e9 | e8 | e1 | e0 | f1 | f2 |
| 6 | v2 | v6 | e10 | e9 | e2 | e1 | f2 | f3 |
| 7 | v3 | v7 | e11 | e10 | e3 | e2 | f3 | f4 |
| 8 | v4 | v5 | e5 | e9 | e4 | e11 | f5 | f1 |
| 9 | v5 | v6 | e6 | e10 | e5 | e8 | f5 | f2 |
| 10 | v6 | v7 | e7 | e11 | e9 | e6 | f5 | f3 |
| 11 | v7 | v4 | e4 | e8 | e10 | e7 | f5 | f4 |



## Beispiel für das Durchlaufen der Datenstruktur

- Alle Kanten von $f_4$ in CCW-Reihenfolge aufzählen:

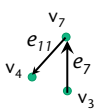



**Kantenliste**

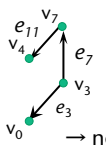
| *e* | *org* | *dest* | *ncw* | *nccw* | *pcw* | *pccw* | *lf* | *rf* |
|---|---|---|---|---|---|---|---|---|
| 0 | v0 | v1 | e1 | e5 | e4 | e3 | f1 | f0 |
| 1 | v1 | v2 | e2 | e6 | e5 | e0 | f2 | f0 |
| 2 | v2 | v3 | e3 | e7 | e6 | e1 | f3 | f0 |
| 3 | v3 | v0 | e0 | e4 | e2 | e7 | f4 | f0 |
| 4 | v0 | v4 | e8 | e11 | e0 | e3 | f4 | f1 |
| 5 | v1 | v5 | e9 | e8 | e1 | e0 | f1 | f2 |
| 6 | v2 | v6 | e10 | e9 | e2 | e1 | f2 | f3 |
| 7 | v3 | v7 | e11 | e10 | e3 | e2 | f3 | f4 |
| 8 | v4 | v5 | e5 | e9 | e4 | e11 | f5 | f1 |
| 9 | v5 | v6 | e6 | e10 | e5 | e8 | f5 | f2 |
| 10 | v6 | v7 | e7 | e11 | e9 | e6 | f5 | f3 |
| 11 | v7 | v4 | e4 | e8 | e10 | e7 | f5 | f4 |

$f_4 \rightarrow e_{11}$ / "-" :

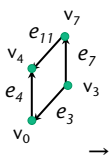

→ pccw




→ pccw




→ nccw




→ nccw

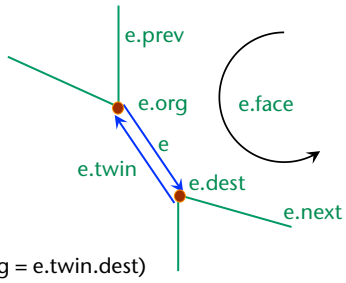
Ende

- Alle Nachbarschafts-Abfragen lassen sich in Zeit $O(k)$ durchführen! ($k$ = Größe der Ausgabe)
  - 3 Arten von Abfragen gehen direkt in $O(1)$, und 6 Abfragearten gehen durch lokales Umrunden einer Facette oder eines Vertex in $O(k)$
- Problem: wenn man sich an einer Kante entlang "hangeln" möchte, muß man jedesmal testen, wie die Kante orientiert ist, um zu wissen, ob man ***n[c]cw*** oder ***p[c]cw*** verfolgen muß!

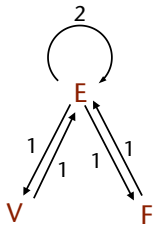


# Doubly-Connected Edge List [Preparata & Müller, 1978]

- In der Computer-Graphik eher bekannt als *"half-edge data structure"*
- Ist die einfachste Nachbarschaftsdatenstruktur
- Idee:
  - Wie Winged-Edge-DS, aber mit "gespaltenen" Kanten
  - Eine Kante (= Eintrag in der Kantentabelle) ist nur noch für *eine* Richtung und *eine* Seite zuständig
  - Zeiger pro Halb-Kante:
    - "Twin" (**twin** | **opposite**)
    - Start- (**org**) und End-Vertex (**dest**)
    - Incident **face** (zur Linken)
    - **Next** und **prev**ious edge (im Umlaufsinn)
    - (Start-Vertex kann man einsparen, da e.org = e.twin.dest)

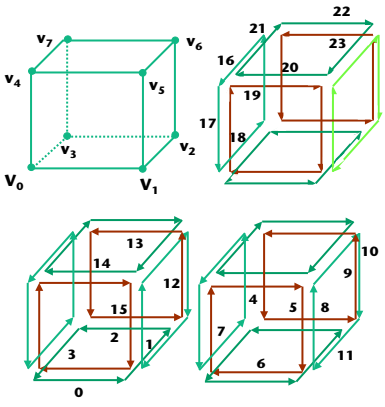


- Abstrakte Notation:
  - Hier ohne Start-Vertex-Zeiger
  - Benötigt doppelt so viele Einträge in der Kanten-Tabelle wie die Winged-Edge-DS






## Beispiel (hier in CW-Order!)




Siehe auch die Demo auf
http://www.holmes3d.net/graphics/dcel/

**Vertexliste**

| v | coord | | | e |
|---|---|---|---|---|
| 0 | 0.0 | 0.0 | 0.0 | 0 |
| 1 | 1.0 | 0.0 | 0.0 | 1 |
| 2 | 1.0 | 1.0 | 0.0 | 2 |
| 3 | 0.0 | 1.0 | 0.0 | 3 |
| 4 | 0.0 | 0.0 | 1.0 | 4 |
| 5 | 1.0 | 0.0 | 1.0 | 9 |
| 6 | 1.0 | 1.0 | 1.0 | 13 |
| 7 | 0.0 | 1.0 | 1.0 | 16 |

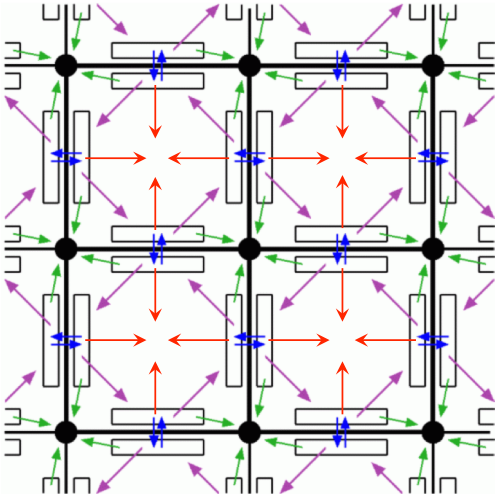
**Facetten**

| | |
|---|---|
| 0 | e20 |
| 1 | e4 |
| 2 | e0 |
| 3 | e15 |
| 4 | e16 |
| 5 | e8 |

**Halbkantenliste**

| e | org | next | prv | twin | e | org | next | prv | twin |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 3 | 6 | 12 | 2 | 13 | 15 | 10 |
| 1 | 1 | 2 | 0 | 11 | 13 | 6 | 14 | 12 | 22 |
| 2 | 2 | 3 | 1 | 15 | 14 | 7 | 15 | 13 | 19 |
| 3 | 3 | 0 | 2 | 18 | 15 | 3 | 12 | 14 | 2 |
| 4 | 4 | 5 | 7 | 20 | 16 | 7 | 17 | 19 | 21 |
| 5 | 5 | 6 | 4 | 8 | 17 | 4 | 18 | 16 | 7 |
| 6 | 1 | 7 | 5 | 0 | 18 | 0 | 19 | 17 | 3 |
| 7 | 0 | 4 | 6 | 17 | 19 | 3 | 16 | 18 | 14 |
| 8 | 1 | 9 | 11 | 5 | 20 | 5 | 21 | 23 | 4 |
| 9 | 5 | 10 | 8 | 23 | 21 | 4 | 22 | 20 | 16 |
| 10 | 6 | 11 | 9 | 12 | 22 | 7 | 23 | 21 | 13 |
| 11 | 2 | 8 | 10 | 1 | 23 | 6 | 20 | 22 | 9 |

- Visualisierung für ein Quad-Mesh:





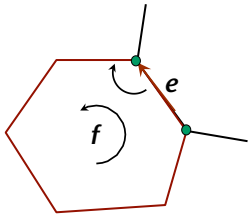
## Invarianten in einer DCEL

- Verwende im folgenden die "Funktionen-Notation", d.h., twin(e) = e.twin
- Invarianten (bzw. Axiome im ADT "DCEL"):
  - twin( twin(e) ) = e , falls das Mesh geschlossen ist
  - org( next(e) ) = dest(e)
  - org(e) = dest( twin(e) ) [falls twin(e) existiert]
  - org( v.edge ) = v [v zeigt immer auf eine "abgehende" Kante!]
  - etc. ...

## Beispiele für das "Abwandern" der Topologie

- Gesucht: alle Vertices, die inzident zu einem geg. Face ***f*** sind
- Algorithmus:

```
e_start ← f.edge
e ← e_start
repeat
  output e.dest
  e ← e.next
until e == e_start
```

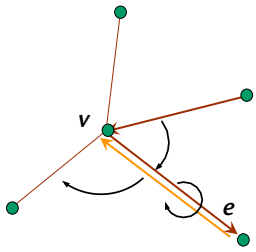



- Laufzeit ist in $O(k)$ , mit $k$ = # Vertices von ***f***



- Aufgabe: zu geg. Vertex ***v*** alle benachbarten Vertices liefern
- Algorithmus (oBdA zeigt ***v*** auf eine abgehende Kante):

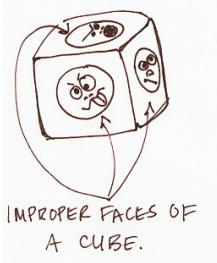
```
e_start ← v.edge
e ← e_start
repeat
  output e.dest
  e ← e.twin.next
until e == e_start
```




- Laufzeit ist in $O(k)$ , wobei $k$ = # Nachbarn von ***v***

- Bezeichnung:
  *Feature* = Vertex oder Kante oder Facette

- Satz:
  Mit einer DCEL bekommt man zu einem beliebigen Feature alle Inzidenz- und Adjazenz-Informationen in $O(1)$ oder in $O(k)$, wobei $k$ = # Nachbarn ist

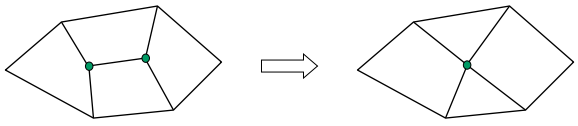



Courtney Gibbons 2007

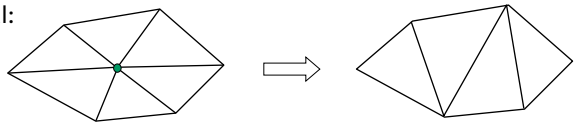


# Anwendungsbeispiel

- Simplifizierung: Grobes Mesh aus einem gegebenen "feinen" Mesh erzeugen
  - Dabei Einhaltung bestimmter Kriterien (wird hier nicht weiter vertieft)
- Elementare Operationen:
  - Edge Collapse:



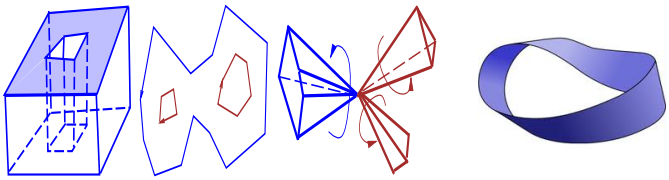
    - Benötigt alle Kanten adjazent zu e
  - Vertex Removal:



    - Benötigt alle Kanten inzident zu v

## Limits / Erweiterungen der DCEL

- Eine DCEL kann nur Meshes speichern die ...
  1. zwei-mannigfaltig und
  2. orientierbar sind, und
  3. dessen Polygone keine "Löcher" haben!



- Erweiterungen: viele, z.B. die von Hervé Brönnimann
  - Speichere für nicht-2-mannigfaltige Vertices *mehrere* Zeiger auf adjazente Kanten
  - Dito für Facetten mit Löchern
  - Ergibt mehrere sog. Zyklen von Kanten für solche Vertices/Faces



## Eine DCEL-Datenstruktur für Nicht-2-Mannigfaltigkeiten

- Eine DCEL repräsentiert 2-Mannigfaltigkeiten
- *Directed Edge DS*: Erweiterung von Half-Edge-DS für Meshes, die an Ausnahmestellen keine 2-Mannigfaltigkeit sind

- Idee:
  - Pointer auf die Kanten (e.next, e.prev, v.edge, f.edge) als Integer-Index in das Edge-Array speichern
  - Benutze Vorzeichen des Index als Hinweis auf zusätzliche Information
  - Interpretiere negative Indizes als Indizes in zusätzliche Arrays, z.B.
    - Liste aller Kanten, die von Vertex ausgehen, oder
    - Zusammenhangskomponenten an Vertex / Kante

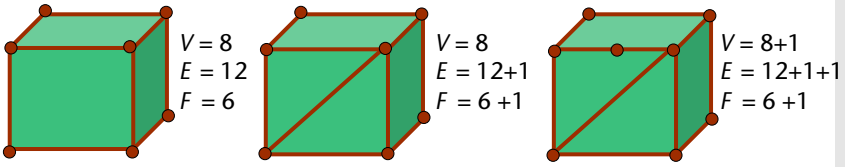
# Die Euler-Formel

- Satz (Eulerformel):
  Für alle Polyeder, die homöomorph zu einer Kugel sind, gilt

$$V - E + F = 2$$

  wobei $V$, $E$, $F$ = Anzahl Vertices, Edges, Faces

- Beispiele:



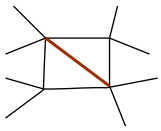



# Beweis (nach Cauchy)

- Gegeben: geschlossenes Mesh (Polyeder)
- Erste Idee:
  - Entferne eine Facette (ergibt offenes Mesh; Rand ist genau der Kantenzug der entfernten Facette)
  - Ziehe Mesh an diesem Rand auseinander in planaren Graphen (geht nur, wenn Polyeder homöomorph zu Kugel)
  - Jetzt zu zeigen:

$$V - E + F = 1$$

- Zweite Idee: trianguliere den Graphen (das Mesh)

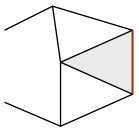


  - Ziehe Diagonale in Facetten mit mehr als 3 Vertices ein
  - Es gilt weiterhin

$$V' - E' + F' = V - (E + 1) + (F + 1) = V - E + F$$

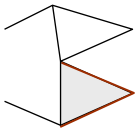
- Der Graph hat einen Rand; Dreiecke haben 0, 1, oder 2 "Randkanten"
- Wiederhole eine der folgenden beiden Transformartionen:
  - Falls es ein Dreieck mit genau 1 Randkante gibt, lösche dieses Dreieck; es gilt

    $V' - E' + F' = V - (E - 1) + (F - 1) = V - E + F$



  - Falls es ein Dreieck mit genau 2 Randkanten gibt, lösche dieses Dreieck; es gilt

    $V' - E' + F' = (V - 1) - (E - 2) + (F - 1) = V - E + F$

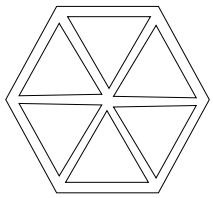


- Wiederhole, bis nur noch 1 Dreieck übrig
  - Für das Dreieck gilt die Euler-Formel offensichtlich
  - Da jede der obigen Transformationen den Wert von V-E+F unverändert gelassen hat, gilt die Formel also auch für den ursprünglichen Graphen, und damit auch für das ursprüngliche Mesh.



# Anwendung der Euler-Formel auf Meshes

- Euler-Formel → Zusammenhang zwischen #Dreiecken und #Vertices in einem geschlossenen Dreiecks-Mesh
- Annahme: ein geschlossenes Dreiecks-Mesh



- Jede Kante gehört zu genau 2 Dreiecken, also

$$3F = 2E$$

- Einsetzen in Euler-Formel:

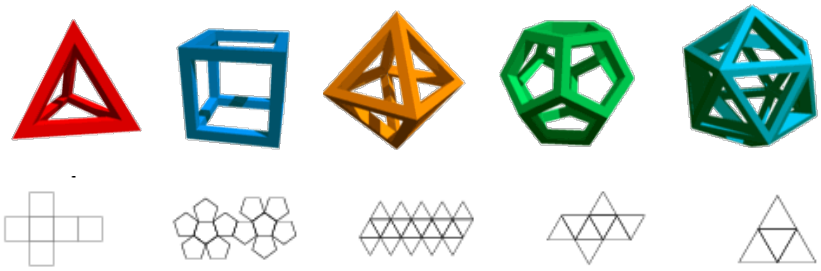
$$2 = V - \frac{3}{2}F + F \;\Leftrightarrow\; \frac{1}{2}F = V\text{–}2$$

- Bei großen Dreiecks-Meshes gilt also

$$F \approx 2V$$

## Anwendung der Euler-Formel auf Platonische Körper

- Definition Platonischer Körper:
  ein konvexes Polyeder, das aus lauter gleichen (kongruenten) regulären Polygonen besteht
- Satz (Euklid):
  Es gibt genau fünf platonische Körper.





## Beweis

- Alle Facetten haben die gleiche Anzahl Kanten = $n$; also:

$$2E = nF \Leftrightarrow F = \frac{2}{n}E$$

- An allen Vertices treffen sich die gleiche Anzahl Kanten = $m$; also

$$2E = mV \Leftrightarrow V = \frac{2}{m}E$$

- Einsetzen in die Euler-Formel:

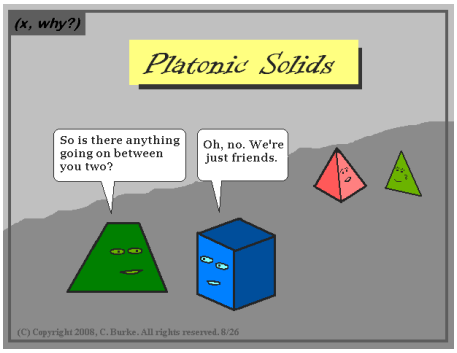
$$2 = V - E + F = \frac{2}{m}E - E + \frac{2}{n}E \Leftrightarrow \frac{2}{E} = \frac{2}{m} - 1 + \frac{2}{n}$$

- Ergibt folgende Bedingung für $m$ und $n$:

$$\frac{1}{m} + \frac{1}{n} = \frac{1}{2} + \frac{1}{E} > \frac{1}{2}$$

- Weitere Bedingung: *m* und *n* müssen jeweils ≥ 3 sein
- Welche {*m*,*n*} erfüllen diese Bedingungen:

  {3,3} {3,4} {4,3} {5,3} {3,5}






## Exkurs: platonische Körper in der Kunst

- Die platonischen Polyeder waren mindest 1000 Jahre vor Plato schon in Schottland bekannt

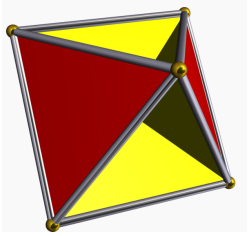
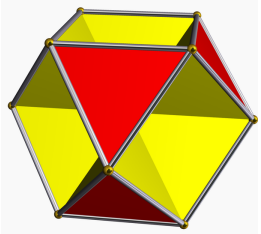
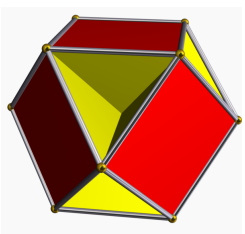
*Portrait of Johannes Neudörfer and his Son*
Nicolas NEUFCHATEL, 1527—1590


Dürer: *Melencolia I*



# Die Euler-Charakteristik

- Achtung: die Euler-Formel gilt so nur für Polyeder, die topologisch äquivalent zur Kugel sind!
- Beispiele:

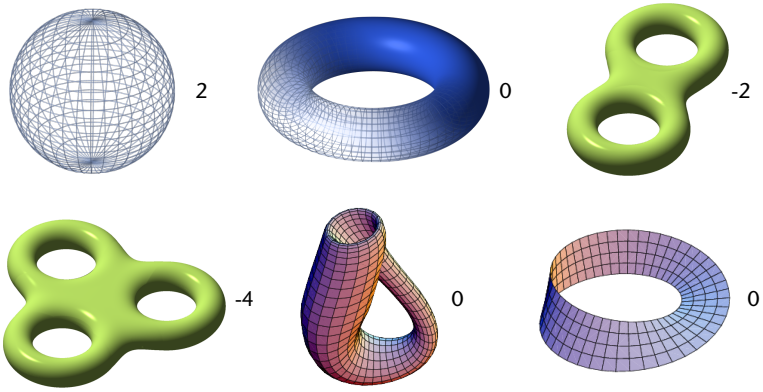
| *Tetrahemihexahedron* | *Octahemioctahedron* | *Cubohemioctahedron* | |
|---|---|---|---|
|  |  |  | |
| 6 - 12 + 7 = 1 | 12 - 24 + 12 = 0 | 12 - 24 + 10 = -2 | V<br>E<br>F |

- Aber: die Größe V-E+F bleibt erhalten, egal wie man das Polyeder verformt (homöomorph) → topologische Invariante

- Definition Euler-Charakteristik:

$$\chi = V - E + F$$

- Beispiele:





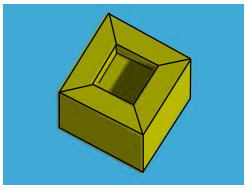

# Die Euler-Poincaré-Formel

- Verallgemeinerung der Euler-Formel auf 2-mannigfaltige, geschlossene Flächen (evtl. mit mehreren Komponenten):

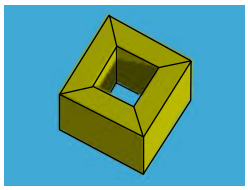
$$V - E + F = 2(S - G)$$

  - G = # Henkel , S = # *Shells* (Schalen / Komponenten)
  - Henkel (Loch, hole) = eine Schnur im Inneren eines Henkels kann man nicht auf einen Punkt zusammenziehen
  - G = Genus
  - Schale (shell) = durch Wandern auf der Schale kann man jeden Punkt der Schale von jedem anderen aus erreichen
  - Durch "innere" Schalen kann man sog. "Voids" (Aushöhlungen) aus einem Körper herausnehmen
- Es gibt noch weitere Verallgemeinerungen!

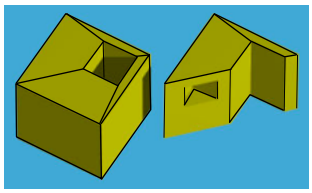
- Beispiele:
  - V = 16, E = 28, F = 14, S = 1, G = 0:
    V-E+F = 2 = 2(S-G)



  - V = 16, E = 32, F = 16, S = 1, G = 1:
    V-E+F = 0 = 2(S-G)

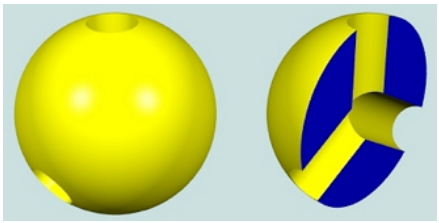


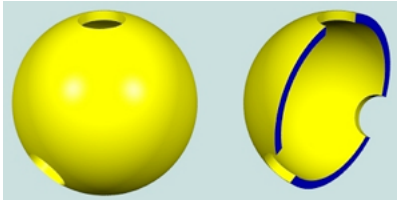
  - V = 16+8, E = 32+12, F = 16+6, G = 1, S = 2:
    V-E+F = 2 = 2(S-G)



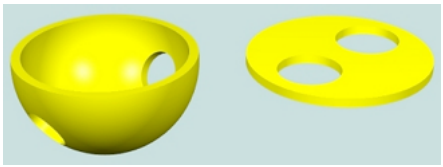


- Achtung: der Genus ist manchmal nicht ganz einfach zu bestimmen!
- Beispiel:
  - Genus = 2
  - "Beweis": topologisch äquivalent deformieren, bis Genus klar ist





1.



2. 3.