HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Computergrafik I

Kapitel 3: Einfache Geometrische Objekte

Prof. Dr. Ingo Ginkel

Sommersemester 2018

Einfache Geometrische Objekte - Kapitelübersicht

- Primitive Objekte in 2D: Polygone
- Einfache Netze: Konstruiere komplexere Objekte aus einer Menge von Polygonen
- Primitive Objekte 3D: Kugel, Würfel, Zylinder, Torus, Tetraeder, etc.
- Unterteilungskurven
- Primitive Objekte 3D: Rotationskörper
- Ausblick: Isoflächen - Blobby Molecules - Metaballs
- Implementierungsaspekte

Geometrische Objekte - Grundprinzip

- **Variante 1:** Definiere zunächst einfache Objekte, komplexere Formen werden durch immer komplexere Objekte beschrieben.
  - Durch die komplexe Beschreibung brauchen auch einfache Objekte eine aufwändige Beschreibung.
  - Berechnungen - z.B. Kollisionsberechnung, Deformationen - sind für komplexe Objekte entsprechend komplex.
- **Variante 2:** Definiere zunächst einfache Objekte, komplexere Formen werden durch eine (größere) Menge von einfachen Objekten beschrieben.
  - Objekt selbst einfach zu beschreiben, Beschreibung der Verbindung/Nachbarschaft der Objekte untereinander notwendig.
  - Berechnungen - z.B. Kollisionsberechnung, Deformationen - müssen für die Summe aller Objekte durchgeführt werden um ein Ergebnis zu erhalten.

Einfache geometrische Objekte - Polygone

Definition 3.1

(i) *Für eine Menge* $\{v_0, \ldots, v_n\}$ *von* **Punkten (vertices)** *im zwei- oder dreidimensionalen Raum bezeichnen wir die Menge* $Q = \{(v_0, v_1), (v_1, v_2), ..., (v_{n-1}, v_n)\}$ *als* **Polygonzug** *oder Polyline.*

(ii) *Die Paare von Punkten sind die* **Kanten (edges)** *des Polygonzugs.*

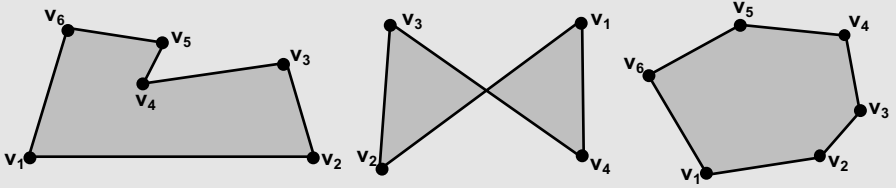
(iii) *Ist* $v_n = v_0$*, (letzter Punkt stimmt mit dem ersten überein), sprechen wir von einem* **geschlossenen Polygonzug.**

(iv) *Das von einem geschlossenen Polygonzug umrandete Gebiet heißt* **Polygon (face)**.

Bemerkung: Üblich ist es, ein Polygon als geordnete Sequenz von Punkten $P = v_0, v_1, v_2, \ldots, v_{n-1}, v_0$ anzugeben.

Polygone - Eigenschaften

Beispiel 3.2



Eigenschaften:

Polygone - Eigenschaften - Windungszahl

Definition 3.3 (Orientierung von Polygonen)

(i) *Die* **Windungszahl** *eines Polygons* $P = v_0, v_1, v_2, \ldots, v_{n-1}, v_0$ *bezüglich eines gegebenen Punktes* q *ist definiert als*

$$w = \frac{1}{2\pi} \sum_{i=0}^{n-1} \theta(v_i, q, v_{i+1})$$

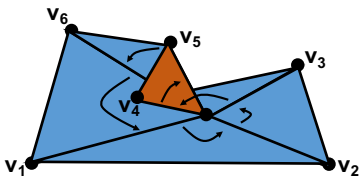
wobei $\theta(v_i, q, v_{i+1})$ *der von* (v_i, q, v_{i+1}) *eingeschlossene Winkel ist und* $(n-1)+1 = 0$ *(zyklische Betrachtung).*

(ii) *Die Windungszahl eines Polygons P bezüglich eines inneren Punktes q ist*

- $w = -1$, *falls P im Uhrzeigersinn (math. negativ) orientiert ist,*
- $w = 1$, *falls P gegen den Uhrzeigersinn (math. positiv) orientiert ist,*

Bemerkung: $w = 0$ für jeden Punkt der außerhalb des Polygons liegt.

Polygone - Eigenschaften - Windungszahl



Problem inneren Punkt zu finden, Schwerpunkt nicht geeignet.

Diese Methode als Test für die Orientierung

- funktioniert auf diese einfache Weise nur für ebene (planare) Polygone.
- funktioniert auch wenn sich die Polygone selbst überschneiden.
- wäre als *inside/outside Poylgon-Test* für Polygone mit Überschneidung geeignet.
- ist aber sehr aufwändig (trig. Funtionen $\rightarrow$ iterative floating-point Berechnungen).
- es existieren viel effizientere Algorithmen für viele Aufgaben, wenn Form von Polygonen weiter eingeschränkt wird.

Polygone - Eigenschaften - planar, einfach

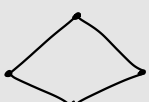
Definition 3.4 (planar)

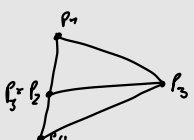
Ein Polygon wird planar genannt, wenn sich alle Punkte in einer gemeinsamen Ebene befinden.

Definition 3.5 (einfaches Polygon)

Ein **einfaches Polygon** *liegt vor, wenn der Schnitt von jeweils zwei Kanten entweder die leere Menge ist, oder einer der Eckpunkte ist und jeder Endpunkt einer Kante höchstens zu zwei Kanten des Polygons gehört.*

Beispiele:



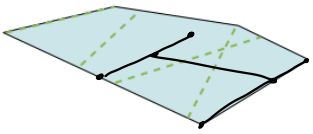


Polygone - Eigenschaften - Konvexität

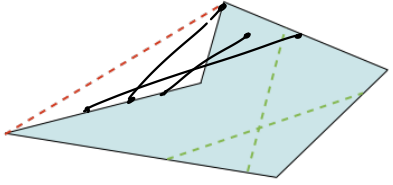
Definition 3.6 (konvexes Polygon)

Ein Polygon ist genau dann **konvex**, *wenn für zwei beliebige Punkte* $\mathbf{v}$ *und* $\mathbf{w}$ *auf dem Rand oder im inneren des Polygons alle Punkte der Konvexkombination* $(1-\lambda)\mathbf{v}+\lambda\mathbf{w}$, $(\lambda \in [0,1])$ *im Polygon liegen*

- Viele Algorithmen setzen konvexer Polygone voraus, bzw. Aufgaben können für konvexe Polygone durch wesentlich effizientere Algorithmen erledigt werden.

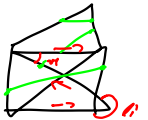


konvex



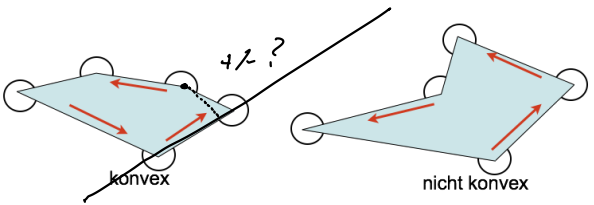
nicht konvex

Polygone - Eigenschaften - Konvexität



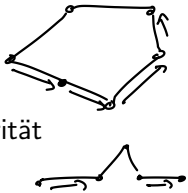
- **Planares, einfaches Polygon** auf Konvexität testen:
  (1) Nehme drei aufeinanderfolgende Punkte $(\mathbf{v}_i, \mathbf{v}_{i+1}, \mathbf{v}_{i+2})$ des Polygons
  (2) Normalformen der Geradengleichungen der Kanten $(\mathbf{v}_i, \mathbf{v}_{i+1})$ bilden, wobei die Normalen ins Innere des Polygons zeigen.
  (3) Berechne den orientierten Abstand des Punktes $\mathbf{v}_{i+2}$.
  (4) Wiederhole Schritt (1) - (3) für jedes Tripel von aufeinanderfolgenden Punkten im Polygon.

Hat der Abstand immer das gleiche Vorzeichen, ist das Polygon konvex.



Achtung: der Test funktioniert nicht immer korrekt für ein nicht-einfaches Polygon!

Polygone - Orientierung - praktische Umsetzung



- Für Zwecke in der Computergrafik wird konsistente Orientierung, Planarität und Einfachheit gefordert.
- In der Praxis wird üblicherweise kein Test darauf durchgeführt, da zu komplex
- Mit der Orientierung wird die Vorder-/Rückseite des Polygons festgelegt.
  - Für spätere konstruierte komplexere Objekte: wo ist innen/außen?

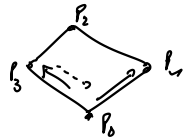
Definition 3.7 (Kreuzprodukt)

Das **Kreuzprodukt** *(oder Vektorprodukt) von zwei 3D Vektoren* $\mathbf{u} = (u_1, u_2, u_3)$ *und* $\mathbf{v} = (v_1, v_2, v_3)$ *wird mit* $\mathbf{u} \times \mathbf{v}$ *bezeichnet und ist mit den Vektor-Komponenten wie folgt definiert*

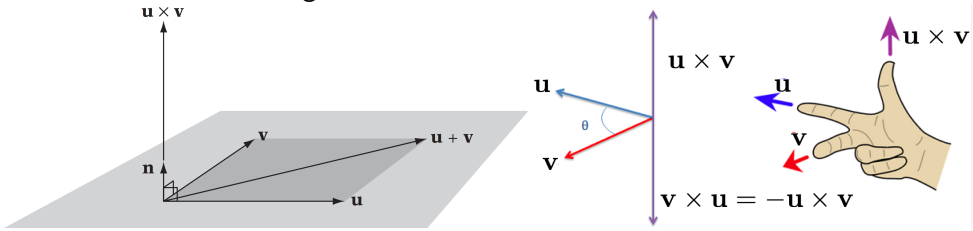
$$\mathbf{u} \times \mathbf{v} = (u_2 v_3 - u_3 v_2, -(u_1 v_3 - u_3 v_1), u_1 v_2 - u_2 v_1).$$

Das Ergebnis ist ein Vektor der senkrecht auf $\mathbf{u}$ *und* $\mathbf{v}$ *steht.*

Polygone - Orientierung - praktische Umsetzung



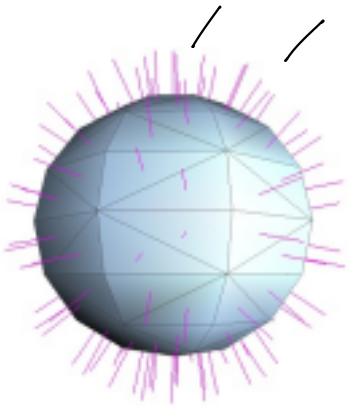
- Geometrische Anschauung:



- Für planare Polygone: Oberflächennormale $\mathbf{n}$ nach außen zeigend
  - Gegen den Uhrzeigersinn (CCW): Für einen beliebigen Eck-Punkt $\mathbf{p}_i$: $\mathbf{n}_{ccw} = (\mathbf{p}_{i+1} - \mathbf{p}_i) \times (\mathbf{p}_{i-1} - \mathbf{p}_i)$
  - Mit dem Uhrzeigersinn (CW): $\mathbf{n}_{cw} = (\mathbf{p}_{i-1} - \mathbf{p}_i) \times (\mathbf{p}_{i+1} - \mathbf{p}_i) = -\mathbf{n}_{ccw}$

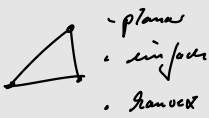
Polygone - Orientierung - praktische Umsetzung

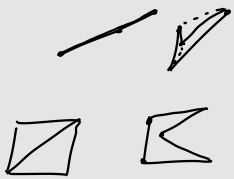
- **Polygone als Oberfläche von realen Objekten:**



- Für jede Fläche eines komplexen Objektes kann festgelegt werden wo innen/außen ist
- Typischerweise kommen Dreiecke und Vierecke zum Einsatz

Gründe für Dreiecke und Vierecke





Polygone - Orientierung - praktische Umsetzung

warum ist innen/außen so wichtig:

- Ziel ist es ein Bild zu erzeugen, also nur die sichtbaren Polygone relevant.
  - **Backface Culling:** Bei geschlossenen Objekten können so die nicht sichtbaren Polygone eines Objektes leicht identifiziert werden.
  - **Rückseiten markieren:** Bei offenen Objekten kann so die (sichtbare) Rückseite eines Polygons anders gefärbt werden (verstärkt das „volumetrische Verständnis“ des Betrachters für das Objekt).

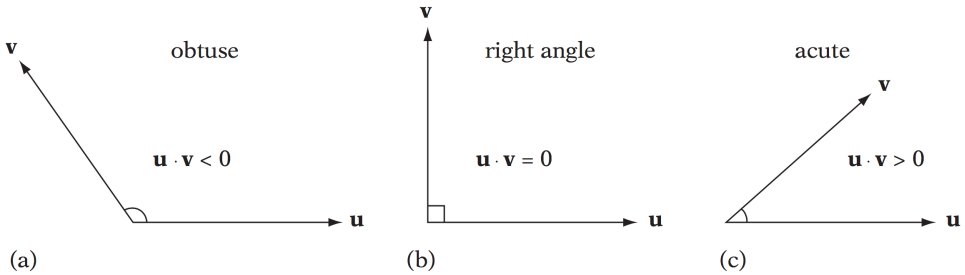
Definition 3.8 (Skalarprodukt)

Das Skalarprodukt (engl. dot product*) von zwei Vektoren* $\mathbf{u}$ *und* $\mathbf{v}$ *ist definiert als die Summe der Produkte ihrer Komponenten. Also:*

$$\mathbf{u} \cdot \mathbf{v} = (u_1, u_2, \ldots, u_n) \cdot (v_1, v_2, \ldots, v_n) = u_1 v_1 + u_2 v_2 + \cdots + u_n v_n$$

Polygone - Orientierung - praktische Umsetzung

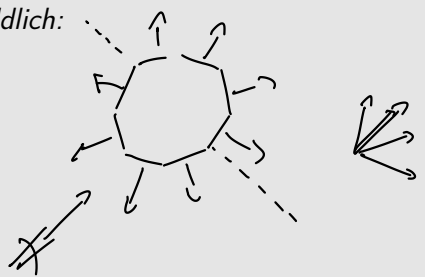
- Für den Winkel θ zwischen $\mathbf{u}$ und $\mathbf{v}$ gilt: $\mathbf{u} \cdot \mathbf{v} = \|\mathbf{u}\| \cdot \|\mathbf{v}\| \cos\theta$
- Das Vorzeichen des Skalarprodukts sagt, ob der Winkel zwischen den Vektoren a) stumpf, b) ein rechter Winkel oder c) spitz ist.

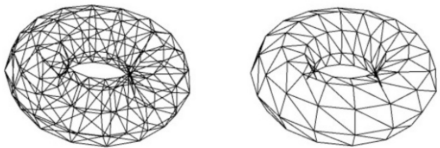


Polygone - Orientierung - Backface Culling

Backface Culling: prüfe ob $(\mathbf{V}_0 - \mathbf{P}) \cdot \mathbf{N} \geq 0$, wobei $\mathbf{P}$ der Aug-Punkt ist, $\mathbf{V}_0$ ein Eckpunkt eines Dreiecks/Polygons und $\mathbf{N}$ seine Normale.

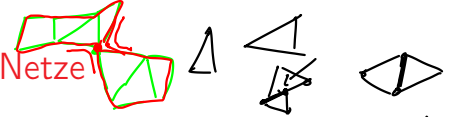
bildlich:





Abgewandte Polygone werden nicht weiterverarbeitet
$\Rightarrow$ Performance-Gewinn.

Einfache geometrische Objekte - Polygonale Netze



Definition 3.9 (Polygonales Netz)

Eine Menge von geschlossenen, planaren und einfachen Polygonen (= Faces) bilden ein polygonales Netz, falls:

(i) *Je zwei Faces haben entweder keinen Punkt, genau einen Punkt oder eine ganze Kante gemeinsam. Also: Der Schnitt zwischen zwei Faces ist entweder leer, ein Punkt oder eine ganze Kante.*

(ii) *Jede Kante einer Facette gehört zu einer oder höchstens zwei Faces.*

(iii) *Die Menge aller Kanten, die nur zu einem Face gehören, ist entweder leer (Netz geschlossen) oder bildet mehrere geschlossene und einfache Polygonzüge (=Ränder des Netzes, auch bei Löchern).*

(iv) *Jeder Punkt hat keine oder genau zwei Kanten, die zu einem Rand gehören.*

Polygonale Netze - Beispiele

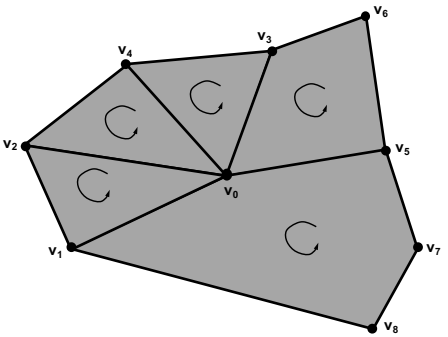
Beispiel 3.10 (Polygonale Netze?)

3 kolineare Punkte, reg. Netz, Raster, Book, isolierte Punkte, Vertex mit 2 Rändern

Polygonale Netze - Datenstruktur - Explizite Speicherung

Erster Ansatz für Datenstruktur: **Explizite Speicherung**

- Übernehme also die Speicherung von den Polygonen, speichere jedes Face $\mathbf{P}_i$ als die Sequenz seiner Punkte auf dem Rand.



$$\mathbf{P}_1 = \big((v_{0x}, v_{0y}, v_{0z}), (v_{2x}, v_{2y}, v_{2z}), (v_{1x}, v_{1y}, v_{1z})\big)$$

$$\mathbf{P}_2 = \big((v_{0x}, v_{0y}, v_{0z}), (v_{1x}, v_{1y}, v_{1z}), (v_{8x}, v_{8y}, v_{8z}), (v_{7x}, v_{7y}, v_{7z}), (v_{5x}, v_{5y}, v_{5z})\big)$$

$$\mathbf{P}_3 = \big((v_{0x}, v_{0y}, v_{0z}), (v_{5x}, v_{5y}, v_{5z}), (v_{6x}, v_{6y}, v_{6z}), (v_{3x}, v_{3y}, v_{3z})\big)$$

$$\mathbf{P}_4 = \big((v_{3x}, v_{3y}, v_{3z}), (v_{4x}, v_{4y}, v_{4z}), (v_{0x}, v_{0y}, v_{0z})\big)$$

$$\mathbf{P}_5 = \big((v_{2x}, v_{2y}, v_{2z}), (v_{0x}, v_{0y}, v_{0z}), (v_{4x}, v_{4y}, v_{4z})\big)$$

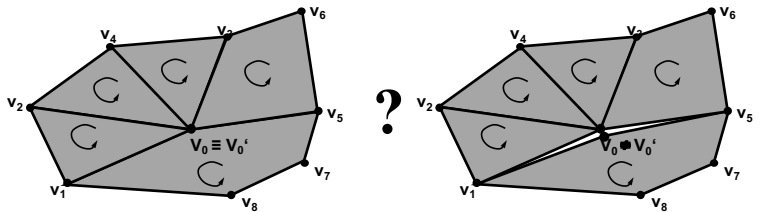
Explizite Speicherung

- Problematisch, da Punkte mehrfach als Tripel von float-Zahlen gespeichert werden müssen.

  $\Rightarrow$ hoher Speicherverbrauch für Redundanz

  $\Rightarrow$ Prüfung ob Netz konsistent ist muss mit floating-point Genauigkeit erfolgen:

$$\mathbf{P}_1 = ((v_{0x}, v_{0y}, v_{0z}), (v_{2x}, v_{2y}, v_{2z}), (v_{1x}, v_{1y}, v_{1z}))$$
$$\mathbf{P}_2 = ((v_{0x}, v_{0y}, v_{0z}), (v_{1x}, v_{1y}, v_{1z}), (v_{8x}, v_{8y}, v_{8z}), (v_{7x}, v_{7y}, v_{7z}), (v_{5x}, v_{5y}, v_{5z}))$$
$$\mathbf{P}_3 = ((v_{0x}, v_{0y}, v_{0z}), (v_{5x}, v_{5y}, v_{5z}), (v_{6x}, v_{6y}, v_{6z}), (v_{3x}, v_{3y}, v_{3z}))$$
$$\mathbf{P}_4 = ((v_{3x}, v_{3y}, v_{3z}), (v_{4x}, v_{4y}, v_{4z}), (v_{0x}, v_{0y}, v_{0z}))$$
$$\mathbf{P}_5 = ((v_{2x}, v_{2y}, v_{2z}), (v_{0x}, v_{0y}, v_{0z}), (v_{4x}, v_{4y}, v_{4z}))$$



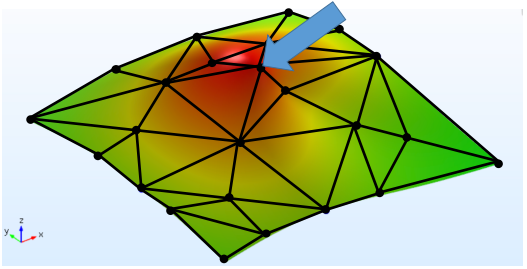
Polygonale Netze - Datenstruktur - Explizite Speicherung

Explizite Speicherung - weitere Problematische Aspekte

- Die Struktur bzw. **Topologie** (= Nachbarschaftsbeziehungen, Orientierung,..) des Modells ist an die **Geometrie** (= Positionen im Raum) gekoppelt.
- Es gibt keine Speicherung gemeinsamer Ecken und Kanten
  - Wie findet man gemeinsame Kanten?
  - Wie findet man alle Kanten eines Vertex?
- Wird benötigt von Algorithmen die auf der Oberfläche in regionaler Nachbarschaft arbeiten: z.B. Deformation eines bestimmten Bereiches
- Geometrie kann nicht unabhängig von der Topologie verändert werden, da gemeinsame Ecken gesucht werden müssen.

Polygonale Netze - Datenstruktur - Explizite Speicherung

- die Deformation an einer Stelle soll auf einen bestimmten Nachbarschaftsbereich ausgedehnt werden bzw. „auslaufen“
- der Aufwand pro Punkt die Nachbarn (über eine Kante) zu finden ist $O(n)$, $n =$ Anzahl Polygone im Netz



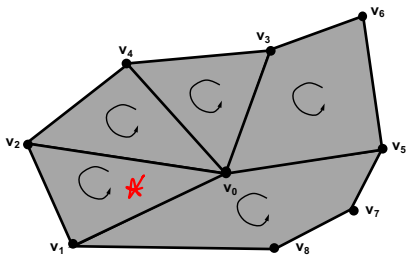
(Aufwand Nachbarschaftsbestimmung)

- $\Rightarrow$ *gegeben Punkt* **p***: Iteriere über alle Polygone und prüfe ob Punkt* **p** *auch in der Liste der Punkte vorhanden ist (auf floating-point Genauigkeit)*
- *eigentlich* $O(n \cdot k)$*, wobei* k *die durchschnittliche Anzahl von Punkten pro Polygon ist (da durch die Liste der Punkte iteriert werden muss)*

Polygonale Netze - Datenstruktur - Eckenliste

Bessere Datenstruktur: **Eckenliste**

- Alle benötigten Punkte $\mathbf{v}_i$ werden in beliebiger Reihenfolge in eine Liste (mit wahlfreiem Zugriff) abgelegt.
- Ein Polygon wird als Liste von Indices (Verweisen) in die Punktliste definiert.
- Der Speicheraufwand pro Polygon beträgt nun nur noch $1\times$`int` statt $3\times$`float`



$$\mathbf{V} = (\mathbf{v}_4, \mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_3, \mathbf{v}_2, \mathbf{v}_6, \mathbf{v}_5, \mathbf{v}_7, \mathbf{v}_8)$$

$$\mathbf{P}_1 = (1, 4, 2)$$

$$\mathbf{P}_2 = (1, 2, 8, 7, 6)$$

$$\mathbf{P}_3 = (1, 6, 5, 3)$$

$$\mathbf{P}_4 = (3, 0, 1)$$

$$\mathbf{P}_5 = (4, 1, 0)$$

Polygonale Netze - Datenstruktur - Eckenliste

Bessere Datenstruktur: **Eckenliste**

- Eindeutigkeit der Punkte ist jetzt gegeben: der Index gibt Punkt eindeutig an.
- Geometrie (=Position der Punkte) kann unabhängig von der Topologie (Liste der Indices) verändert (verschoben, rotiert, ...) werden.
- Gängigste Datenstruktur zum Rendering von Polygonalen Netzen
  - **Polygone als Liste von Punkten zeichnen:** z.B.

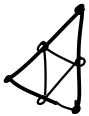
$$P_1 = V[P_1[0]], V[P_1[1]], V[P_1[2]]$$

- Nachbarschaftsbeziehungen sind nach wie vor nur in $O(n)$ zu berechnen
  - **Grund:** Es gibt nur Verweise von Polygonen zu Punkten, aber nicht umgekehrt. Kanten werden gar nicht erst abgespeichert.
- Konsistenz auch nicht zwingend gegeben: es ist möglich ein Netz „wie ein Buch“ (d.h. 1 Kante in mehr als 2 Polygonen) zu definieren

Polygonale Netze - Datenstruktur - Eckenliste

Eckenliste als Datei-Sopeicherformat (z.B. bei „.obj“ von Alias Wavefront)

```
# List of geometric vertices, with (x,y,z[,w]) coordinates.
v 0.123 0.234 0.345
v ...
...
# List of vertex normals in (x,y,z) form; normals might not be unit vectors.
vn 0.707 0.000 0.707
vn ...
...
# Polygonal face element (lists of vertex, texture and normal indices)
f 1 2 3
f 3/1 4/2 5/3
f 6/4/1 3/5/3 7/6/5
f 7//1 8//2 9//3
f ...
...
```

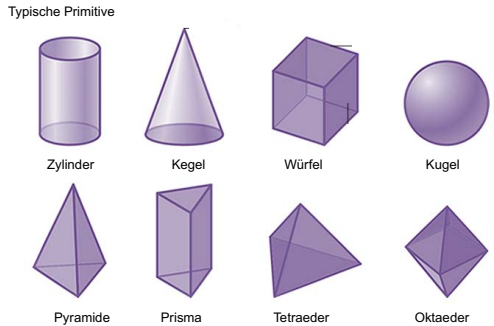


Polygonale Netze - Probleme und Herausforderungen

- **Grundproblem:** Netzauflösung bestimmt wie „eckig“ ein eigentlich rundes oder glattes Objekt wirkt.
- **Erste Strategie:** Grobes Netz immer weiter verfeinern (und vergröbern, Level-of-Detail-Verfahren) $\Rightarrow$ potente Datenstruktur
  - später ein eigenes Kapitel „Mesh Modelling“
- **Zweite Strategie:** Objekte Mathematisch über Formeln etc. definieren und das Netz bei Bedarf in der richtigen Auflösung automatisch erzeugen.
  - **Variante 1:** Primitive Objekte und Kombinationen dieser Objekte mathematisch definieren, diese in Netze wandeln wenn benötigt.
  - **Variante 2:** Komplexe Freiform-Flächen wie Bézier- oder B-Spline Flächen benutzen. $\Rightarrow$ Vorlesung Geometrisches Modellieren im Master.

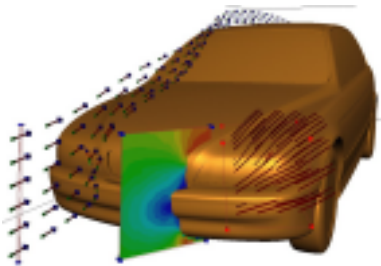
Einfache geometrische Objekte - geometrische Grundprimitive

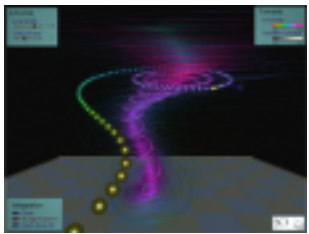
Grundidee: Benutze fest definierte primitive Objekte entweder direkt oder kombiniere diese zu komplexeren Gebilden.



Geometrische Grundprimitive - Anwendung

- Nutze die Objekte als Hilfsmittel bei der Darstellung

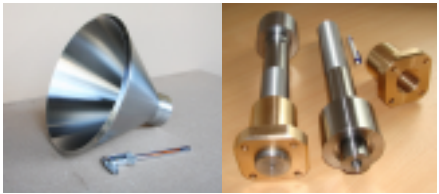


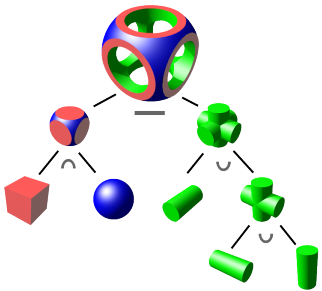


- Visualisiere physikalische Eigenschaften, Beschriftungen, Markierungen in 3D, etc.)

Geometrische Grundprimitive - Anwendung

- Konstruiere Komplexere Objekte aus den Primitiven





- Wird „Constructive Solid Geometry“ genannt.
- insbesondere für Maschinenteile / CNC geeignet.
- führt bool'sche Operationen (Verschneidungen, Additionen, etc.) auf Grundprimitiven aus.
- Sichtweise volumetrisch, nicht nur Oberflächen.

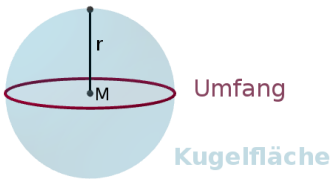
Geometrische Grundprimitive - Würfel

Definition 3.11 (Würfel - Cube)

Ein Würfel (engl. cube) ist ein dreidimensionaler Polyeder (=Vielflächner) mit sechs (kongruenten) Quadraten als Begrenzungsflächen, zwölf gleich langen Kanten und acht Ecken, in denen jeweils drei Begrenzungsflächen zusammentreffen.

- Es liegt nahe für die Zwecke der Computergrafik einen Würfel als geschlossenes Netz zu modellieren.
- Für den Fall, dass die Kanten parallel zu den Koordinatenachsen sind, gibt es sehr effiziente und algorithmisch vorteilhaftere Darstellungen
  - späteres Kapitel „Axis Aligned Bounding Boxen“.
- Auf die gleiche Art und weise lassen sich Oktaeder, Pyramide, Prisma und Tetraeder als Netz darstellen
  - bisher keine Vorteile, da als Netz mit ebenen Flächen dargestellt.

Geometrische Grundprimitive - Kugel



Definition 3.12 (Kugel)

Eine Kugelfläche mit Mittelpunkt $\mathbf{M} = (x_0, y_0, z_0)$ *und Radius* r *ist die Menge aller Punkte* (x, y, z) *für die folgende Gleichung gilt:*

$$(x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = r^2$$

Vorteile dieser Darstellung

- Die Auflösung (= Genauigkeit der Oberflächenbeschreibung) ist beliebig hoch.
- Kollisionsabfrage für Punkte einfach (setze Punkt in Formel ein)
- Oberflächennormale für jeden Oberflächenpunkt ist bekannt (= Richtung von Mittelpunkt zum Oberflächenpunkt)

Geometrische Grundprimitive - Kugel

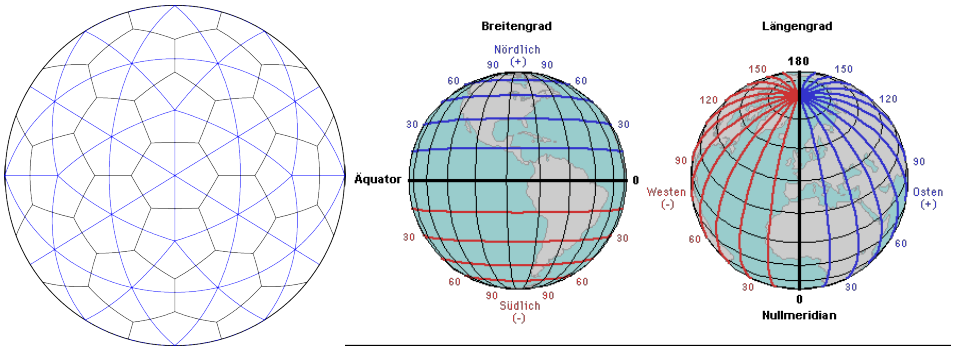
- Die Datenstruktur ist straightforward:

```
// Region R = { (x, y, z) | (x-c.x)^2 + (y-c.y)^2 + (z-c.z)^2 <= r^2 }
struct Sphere {
    Point c;  // Sphere center
    float r;  // Sphere radius
};
```

- Weniger klar ist wie ein solches „Formelobjekt“ gerendert wird:
  - Es ist möglich die Oberfläche direkt aus der Formel zu zeichnen, benötigt aber Methoden wie Raytracing ($\Rightarrow$ späteres Kapitel)
  - **Üblich:** Je nach Bedarf (= z.B. Größe, Distanz zum Betrachter) eine geeignete Netzauflösung erzeugen, dann zeichnen wie polygonale Netze.

Geometrische Grundprimitive - Kugel

- Die Aufteilung der Kugeloberfläche in Polygone ist gar nicht so eindeutig:

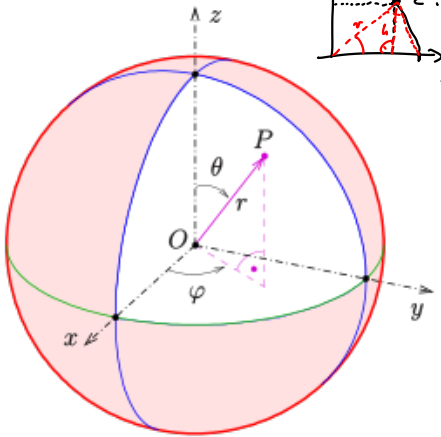


- Üblich sind Vierecke und Dreicke als Netz generiert aus den Längen- und Breitengraden

Geometrische Grundprimitive - Kugel

- **Benutze Kugelkoordinaten:** Jedem Koordinatentripel (r, θ, φ) wird ein Punkt im dreidimensionalen Raum zugeordnet (Parametrisierung). Für ein kartesisches Koordinatensystem, gilt:

$$x = r \cdot \sin\theta \cdot \cos\varphi$$
$$y = r \cdot \sin\theta \cdot \sin\varphi$$
$$z = r \cdot \cos\theta$$

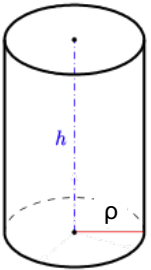


- Laufe den Winkel θ von 0 bis π und den Winkel φ von 0 bis 2π ab um die Sample-Punkte für das Netz zu erzeugen.

Geometrische Grundprimitive - Zylinder

Definition 3.13 (Zylinder)

Ein Zylinder ist eine Fläche, deren Punkte von einem Vektor **h**, *der Achse, denselben Abstand* ρ *haben. Man beschneidet diese Fläche dann mit zwei parallelen Ebenen. Sind die Schnittebenen senkrecht zur Achse, entsteht ein* **senkrechter Kreiszylinder** *mit Radius* ρ *und Höhe* $\|\mathbf{h}\|$. *Die so beschnittene Fläche heißt Mantelfläche des Zylinders.*



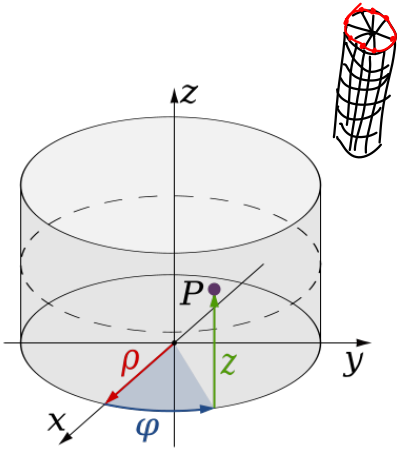
Bemerkungen:

- Der Wert **h** (also die Achse) ist ein Vektor, der die Lage des Zylinders im Raum beschreibt. Die Distanz ρ wird senkrecht zur Achse gemessen.
- Zusätzlich muss der Schnittpunkt zwischen Achse und unterer Begrenzungsfläche angegeben werden um die Lage im Raum zu spezifizieren.

Geometrische Grundprimitive - Zylinder

- Richte ein kartesisches Koordinatensystem so aus, dass die z-Achse mit der des Zylinders zusammenfällt, die x-Achse in Richtung $\varphi = 0$ zeigt und der Winkel φ von der x-Achse zur y-Achse wächst, dann gilt:

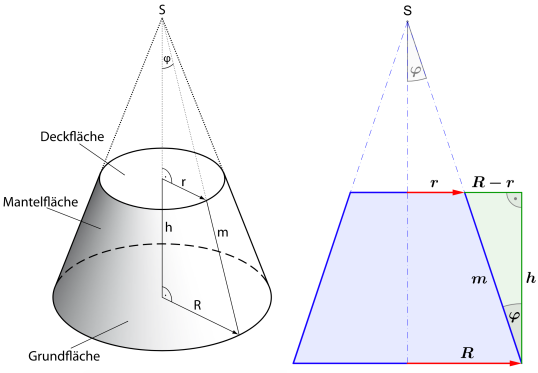
$$x = \rho \cos\varphi$$
$$y = \rho \sin\varphi$$
$$z = z$$



- Laufe den Winkel φ von 0 bis 2π und z von 0 bis $\|\mathbf{h}\|$ ab um die Sample-Punkte für das Netz zu erzeugen.

Geometrische Grundprimitive - Kegel und Kegelstumpf

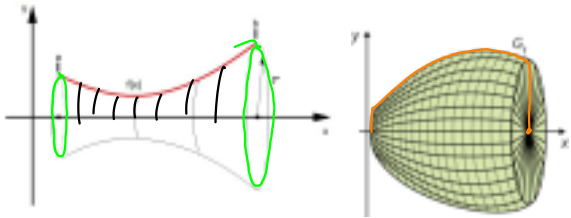
- **Idee:** wie bei Zylinder rotiere Punkte um eine Achse,
- Abstand der Punkte von der Achse nicht mehr konstant, Kontur darf „schräg stehen“.
- Spezifikation üblich durch Angabe von 2 Radien (r, R) und Höhe h.
- Kegel: $r = 0$.



- **Annahme:** Nullpunkt des Koordinatensystems im unteren Rotationspunkt (= der Punkt um den R rotiert).

Einfache geometrische Objekte - Rotationskörper

- **Idee:** wie bei Zylinder oder Kegel rotiere Punkte um eine Achse,
- Die Kontur kann beliebige Kurve sein

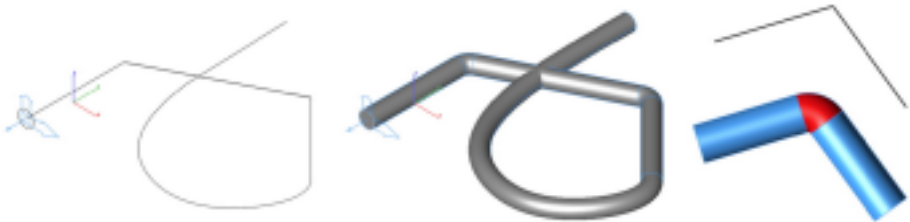


- **2 Varianten:**
  - Achse schneidet die Kurve nicht: ⇒ offener Körper mit aus Rotation erzeugter Mantelfläche, Netz-Modellierung mit Vierecken
  - Achse geht durch Anfangs- und Endpunkt der Kurve ⇒ Geschlossener Körper, Netz-Modellierung mit Vierecken und Dreiecken („Triangle Fan“) an den Enden
- **Anwendung:** z.B. die Schachfiguren aus dem einleitenden Beispielfilm sind so modelliert.

Einfache geometrische Objekte - Sweep-Objekte

Sweep-Körper:

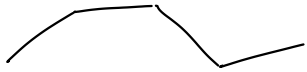
- **Idee:** Verschiebe eine (geschlossene) Kurve bzw. Querschnittsfläche auf einer gekrümmten Leitkurve durch den Raum



- Häufig in CAD Systemen eingesetzt.

- **Fehlt noch:** Drehung der Querschnittsfläche während der Verschiebung:

Polygonale Kurven - Unterteilungskurven



Zwischenfazit:

- Alle Objekte wirken sehr „mathematisch“ in dem Sinne, dass alle gekrümmten Flächen durch Rotation (d.h. durch eine Kreisbahn) erzeugt werden.
- Wünschenswert: tatsächliche beliebig gekrümmte Kurven
- Nutze Polygonale Sichtweise von Anfang an schon bei der Definition einer Konturkurve.

Algorithmisches Interface:

- gewünscht ist eine Methode, die automatisch eine bestimmte Polygonauflösung in $x-$ und $y-$ Richtung auf der Oberfläche eines Objekts erzeugt.
- also konkret eine Member-Methode:
  `Polymesh* getPolygonalRepresentation(int` res_x`, int` res_y`)`

Polygonale Unterteilungskurven - Grundprinzip

Idee: Einen Polygonzug durch Hinzufügen von Punkten verfeinern

Definition 3.14 (Polygonale Verfeinerung)

Ein polygonaler Verfeinerungsprozess ist ein Schema, das eine Sequenz von Kontroll-Polygonen erzeugt, wobei für jedes $k > 0$, jedes $\mathbf{P}_j^k$ geschrieben werden kann als

$$\mathbf{P}_j^k = \sum_{i=0}^{n_{k-1}} \alpha_{i,j,k} \mathbf{P}_i^{k-1}$$

Das bedeutet, jedes Element $\mathbf{P}_j^k$ kann als Linearkombination der Kontrollpunkte aus dem Kontrollnetz des vorherigen Schrittes berechnet werden.

$\mathbf{P}_0, \mathbf{P}_1, ..., \mathbf{P}_n$

$\mathbf{P}_0^1, \mathbf{P}_1^1, ..., \mathbf{P}_{n_1}^1$

$\mathbf{P}_0^2, \mathbf{P}_1^2, ..., \mathbf{P}_{n_2}^2$

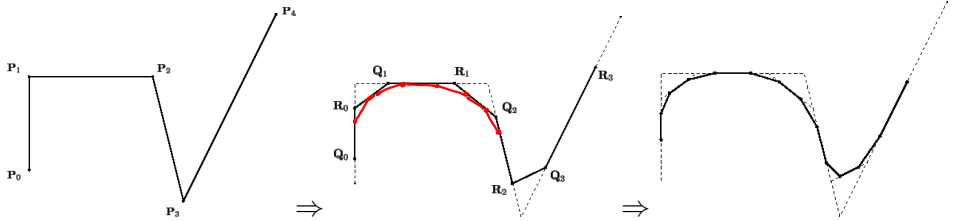
$\vdots$

$\mathbf{P}_0^k, \mathbf{P}_1^k, ..., \mathbf{P}_{n_k}^k$

$\vdots$

Unterteilungskurven - Chaikins Algorithmus (1974)

- **Idee:** Schneide die Ecken einer Polygonalen Kurve ab (ähnlich einem Bildhauer)



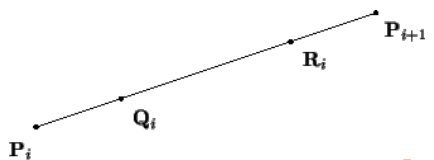
- **Visuell:** Grob-Form wird vorgegeben. Es werden mehr und mehr Punkte erzeugt, diese werden bei iterativer Anwendung eine glatte Kurve erzeugen.
- **Mathematisch:** Eine Funktionenfolge von stückweise linearen Funktionen. Untersuche deren Konvergenz mit Cauchy-Kriterium für gleichmäßige Konvergenz.

Unterteilungskurven - Chaikins Algorithmus (1974)

Formal: Ein gegebenes Kontrollpolygon $\{\mathbf{P}_0, \mathbf{P}_1, ..., \mathbf{P}_n\}$, wird verfeinert, indem eine Sequenz von neuen Kontrollpunkten $\{\mathbf{Q}_0, \mathbf{R}_0, \mathbf{Q}_1, \mathbf{R}_1, ..., \mathbf{Q}_{n-1}, \mathbf{R}_{n-1}\}$ iterativ erzeugt wird:

- jedes Paar von Punkten $\mathbf{Q}_i, \mathbf{R}_i$ durch ein gewichtetes Mittel von $\frac{1}{4}$ und $\frac{3}{4}$ zwischen den Endpunkten eines Liniensegmentes $\overline{\mathbf{P}_i\mathbf{P}_{i+1}}$ gebildet wird, also:

$$\mathbf{Q}_i = \frac{3}{4}\mathbf{P}_i + \frac{1}{4}\mathbf{P}_{i+1}$$
$$\mathbf{R}_i = \frac{1}{4}\mathbf{P}_i + \frac{3}{4}\mathbf{P}_{i+1}$$

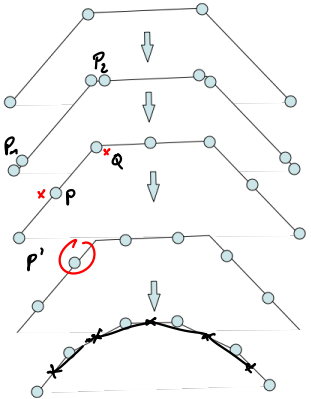


- Diese $2n$ neuen Punkte können als neues Kontrollpolygon betrachtet werden - eine Verfeinerung des ursprünglichen Kontrollpolygons aus $n+1$ Punkten.

Unterteilungskurven - Chaikins Algorithmus (1974)

Alternative Sichtweise für dieselbe erzeugte Kurve:

- 1. Verdoppele alle Punkte
- 2. Mittle benachbarte Punkte
- 3. Mittle nochmals benachbarte Punkte
- 4. Gehe zu Schritt 1.



Warum ist das identisch?

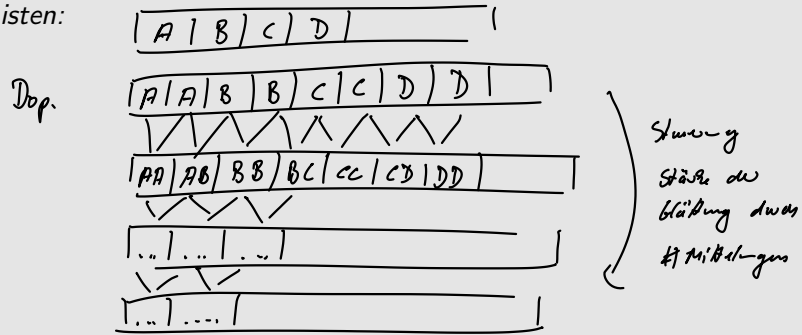
$$P' = \frac{1}{2}P + \frac{1}{2}Q$$

$$\frac{1}{2}\left(P_1 + P_2\right) + \frac{1}{2}\left(P_2 + P_2\right)$$

$$P' = \frac{1}{4}P_1 + \frac{1}{4}P_2 + \frac{1}{4}P_2 + \frac{1}{4}P_2 = \frac{1}{4}P_1 + \frac{3}{4}P_2$$

Unterteilungskurven - Chaikins Algorithmus (1974)

Umsetzung mit Listen:



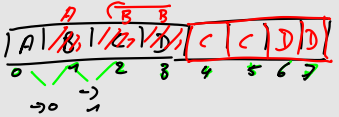
- **Idee / Frage:** Was passiert wenn man mehr als 2 mal mittelt?

Unterteilungskurven - Lane-Riesenfeld Algorithmus

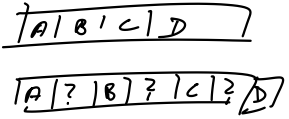
Generalisierung von Chaikin's Algorithmus: Lane-Riesenfeld Algorithmus

- 1. Verdoppele alle Punkte
- 2. Mittle benachbarte Punkte
- 3. Widerhole Schritt 2 insgesamt n-mal
- 4. Gehe zu Schritt 1.

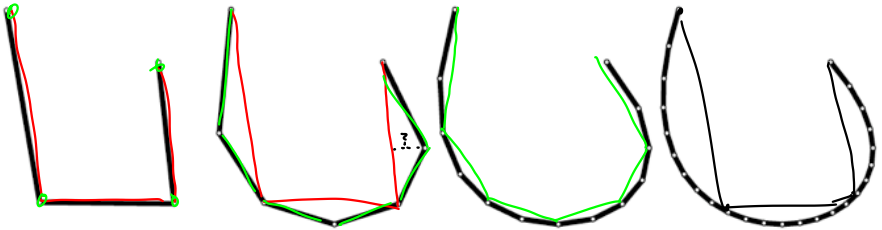
Effizient im Speicher umsetzen:



Unterteilungskurven - 4-Punkt-Schema



- Subdivision Techniken können auch die schon vorhandenen Punkte erhalten und zusätzlich neue Punkte hinzufügen
- Ausgehend von 4 Punkten wird in jedem Verfeinerungsschritt je ein Punkt zwischen zwei vorhandenen Punkten platziert.

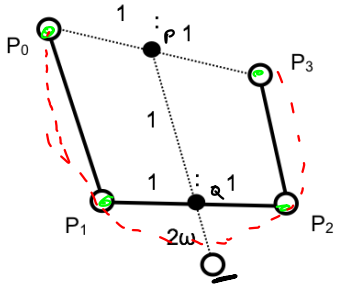


- Wird diese Platzierung sinnvoll gewählt, wird das Objekt sehr schnell „glatt“.

Unterteilungskurven - 4-Punkt-Schema

Position der neuen Punkte basierend auf 4 vorhanden Punkten:

$$P = -\omega P_0 + (\frac{1}{2} + \omega)P_1 + (\frac{1}{2} + \omega)P_2 - \omega P_3$$



ω: Parameter für die „Bauchigkeit“ der Kurve

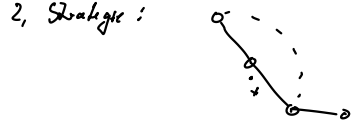
$P = \frac{1}{2} P_0 + \frac{1}{2} \cdot P_3$

$Q = \frac{1}{2} P_1 + \frac{1}{2} \cdot P_2$

$(-2\omega) P + (1+2\omega) Q$

$\Rightarrow$ einfache Formel $P, Q \Rightarrow P = *$

Wahl von ω sehr variabel, üblich $\omega = \frac{1}{16}$

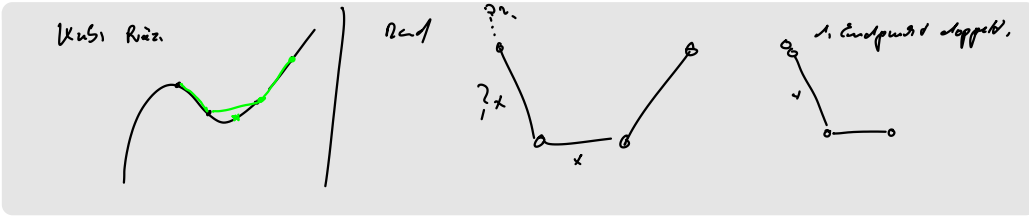


Unterteilungskurven - 4-Punkt-Schema

Wahl des Parameters ω:

- Wahl ist sehr kritisch, schlechte Wahl kann fraktale Kurven erzeugen
- $\omega = \frac{1}{16}$ ist eine gute Wahl, diese bedeutet kubische Präzision für das Schema
  - Kubische Präzision: Liegen die 4 Ursprungspunkte auf einem kubischen Polynom, so liegt auch der neue Punkt auf demselben kubischen Polynom.
  - Kubische Polynome beliebt, weil der Kurvenverlauf Biegeenergie minimiert.

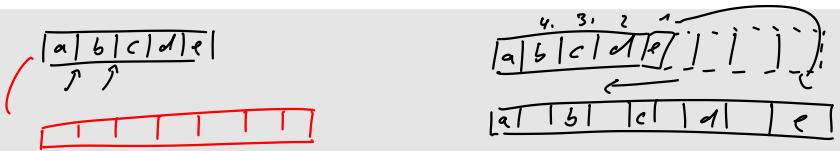
Strategien zur Berechnung neuer Punkte am Rand:



Unterteilungskurven - 4-Punkt-Schema

Implementierungsaspekte

- Ähnlich der Verdoppelung der Punkte bei Lane-Riesenfeld wird die Liste mit n Punkten mit jedem Verfeinerungsschritt etwa doppelt so lang ($2n-1$ Punkte)
- die vorhanden Punkte behalten ihre geometrische Position im Raum, erhalten aber neue Speicherorte in der Liste ($P_i^n \mapsto P_{2i}^{n+1}$)



- füge neuen Speicher an, iteriere rückwärts über die Liste und speichere die Punkte um (sog. gerade Punkte, da gerade Indices)
- Neue Punkte: $P_{2i+1} = -\omega P_{2i} + (\frac{1}{2}+\omega)P_{2i-2} + (\frac{1}{2}+\omega)P_{2i+2} - \omega P_{2i+4}$

Ausblick: Isoflächen - Blobby Molecules - Metaballs

Idee: Verallgemeinere das Prinzip der impliziten Darstellung der Kugel

Definition 3.15 (Level-Set, Kontur)

Eine Menge von Punkten $\{\mathbf{x} \in \mathbb{R}^n : s(\mathbf{x}) = c\}$ *an der eine reellwertige Funktion* $s(\mathbf{x})$ *den konstanten Wert* c *annimmt heißt Level-Set oder Kontour.*

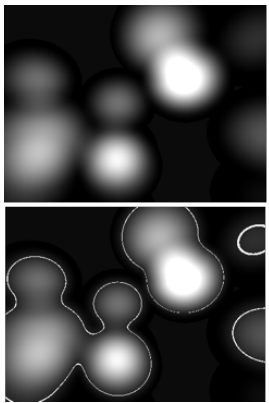
- *speziell* $n = 3$*: Isofläche*

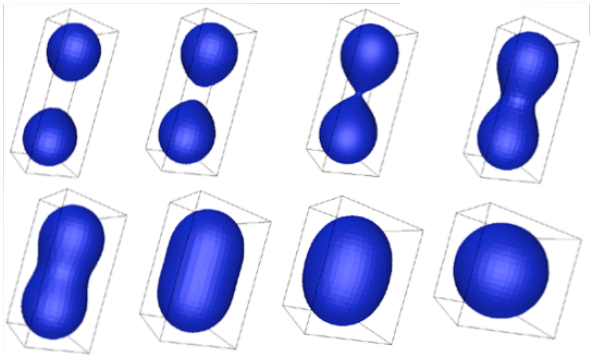
- Je nach Funktion können verschiedene special Effects erzeugt werden
- Summiere mehrere solcher Funktionen auf.

$$\sum_{i=0}^{k} \mathbf{s}_i(x, y, z) = c$$

Ausblick: Isoflächen - Blobby Molecules - Metaballs

Beispiele 2D und 3D:

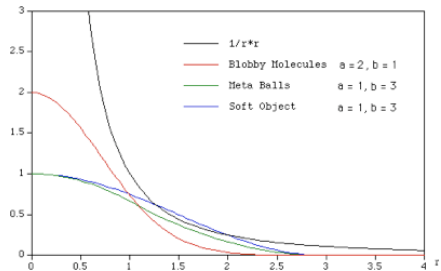




Ausblick: Isoflächen - Blobby Molecules - Metaballs

Animation mit Isoflächen

- Üüblicherweise Funktionen mit Parametern zur Steuerung, z.B. $\mathbf{s}(r) = a \cdot e^{-b \cdot r^2}$
- Variation der Mittelpunkte der Kugeln über die Zeit kann für Animationen genutzt werden.
- Ebenso können die Parameter a und b variiert werden.
- **Resultat:**Tropfen einer Flüssigkeit ähnlich, z.B. Wassertropfen, Blut, flüssiges Metall etc.



Ausblick: Isoflächen - Blobby Molecules - Metaballs

Animation mit Isoflächen

- **StarTrek 6 (1991):** Darstellung von Blut in der Schwerelosigkeit mit Kugeln.
- **Terminator 2 (1991)**: Liquid-Metal Effekt durch kompliziertere implizite Fläche.

Implementierung eines Renderers - Grundlegende Überlegungen

Vorbemerkung: Es gibt keine „richtige“ und einzig wahre Implementierung eines Grafik-Systems. Die gezeigte Variante hier stellt eine (sehr klassische) Möglichkeit der Umsetzung dar.

Model - View - Controller Konzept (1979)

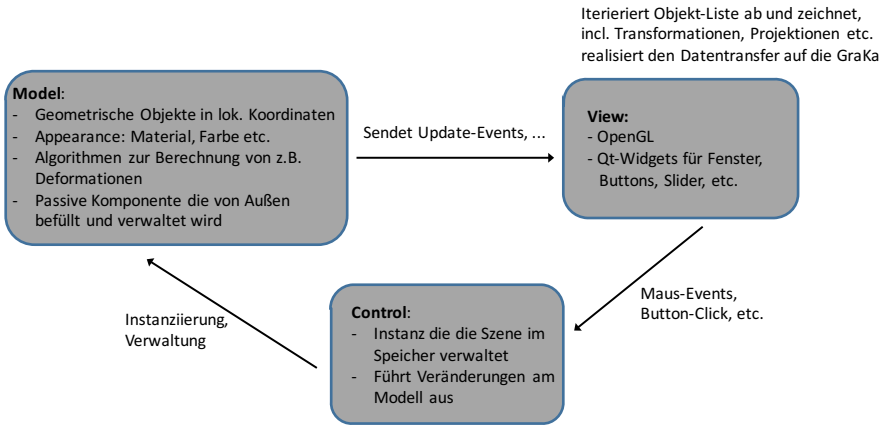
- Model View Controller (MVC, englisch für Modell-Präsentation-Steuerung) ist ein Muster zur Trennung von Software in die drei Komponenten Datenmodell (englisch model), Präsentation (englisch view) und Programmsteuerung (englisch controller).
- **Ziel:** flexibler Programmentwurf, spätere Änderbarkeit, Erweiterbarkeit, Wiederverwendbarkeit der einzelnen Komponenten.

Implementierung eines Renderers - Grundlegende Überlegungen

MVC Konzept für die Computergrafik

- Im Model wegen die einzelnen Darzustellenden Objekte, später auch ihre Eigenschaften wie Material, etc umgesetzt, final dann die ganze darzustellende Szenerie (Definition der Bewegungen, Beleuchtung, der Projektionsart etc.)
- Im Control werden die einzelnen Objekte und deren Eigenschaften instanziiert und verwaltet (z.B. auch: Speicher-Freigabe für gelöschte Objekte etc.)
- Im View wird das Eigentliche Rendering durchgeführt, also die Objekte wirklich in einer Grafik-API gezeichnet, Ebenso werden Maus- und Tastatur-Interaktionen im View entgegengenommen und an den Contoller weitergeleitet
- d.h. sowohl **Model** als auch **Controller** sind frei von Grafik-API spezifischer Implementierung.

Implementierung - MVC Konzept für die Computergrafik



Implementierung eines Renderers - Minimalsystem

Zeichne Objekte direkt in lokalen Koordinaten

- Im Controller muss es eine Liste von zu zeichnenden Objekten geben
- Diese Objekte werden von einer abtrakten Basiskalsse abgeleitet (Interface `CgBAseRenderableObject`)
- Für das konkrete Rendering muss der Typ (d.h. Kurve, Mesh, Implizites Objekt, etc...) festgestellt werden können
- im View gibt es dann eine / mehrere Klassen, die die abstrakten Objekte in eine konkrete API (hier: OpenGl) übertragen.
  - **konkret:** stelle fest welcher Typ Objekt gerade gezeichnet werden soll und führe die spezifisch notwendigen Operationen aus.
  - **Beispiel:** Für eine Line muss nur die Punkte ab-iteriet werden, für ein Netz muss man die einzelnen Punkte für jedes Polygon immer wieder neu in einer Lister „zusammensuchen“.

Implementierung eines Renderers - Grundlegende Überlegungen

Runtime-Type-Information ganz klassisch:

- Definiere ein Enum mit Typ-Namen in der Basisklasse, jede abgeleitete Klasse nimmt einen dieser Typnamen zur Kennzeichnung, sowie eine eindeutige ID zur Identifizierung einer speziellen Instanz der Klasse.

```
class CgBaseRenderableObject
{
    public:
      ⋮
      virtual ObjectType getType() const =0;
      virtual unsigned int getID() const =0;
};

typedef enum ObjectType {PointCloud, Polyline, TriangleMesh,
                         PolygonalMesh, ··· }ObjectType;
```

Implementierung eines Renderers - Zusammenspiel View-Control

- Im View gibt es zwei Klassen mit denen der Controller interagiert.
  - Das GUI für Buttons, Slider etc.
  - Den (abstrakten) Renderer, der die zu zeichnenden Objekte in OpenGL verwaltet und zeichnet
- Ein zu zeichnendes Objekt wird beim Renderer initialisiert. D.h. seine Geometrie wird in lokalen Koordinaten in entsprechende Buffer auf der Grafikkarte geschrieben. Art und Anzahl der Buffer variiert mit Typ der Objekte und genutzter Eigenschaften (z.B. Farbe, Material,...)
- Dort wird es mit der aktuellen Welt- und Kamera - Koordinatensystem - Matrix multipliziert und die resultierenden Punkte gezeichnet.
- Zum Wiederfinden der Buffer ist die eindeutige ID pro Instanz notwendig.
- pro Zeichenaufruf muss der Controller also die entsprechenden Matrizen liefern.

RenderableObject-SubKlassen für konkrete Objekt-Typen

```
class CgBasePolyline : public CgBaseRenderableObject
{
public:
    :
    virtual const std::vector<glm::vec3>& getVertices()  const=0;
    virtual glm::vec3 getColor()      const=0;
    virtual unsigned int getLineWidth() const=0;
};
```

- Polyline besteht aus geordneter Liste von Punkten, einer Farbe für die ganze Linie, und einer Liniendicke fürs Zeichnen.
- im Prinzip einfach erweiterbar für weitere Darstellungseigenschaften.

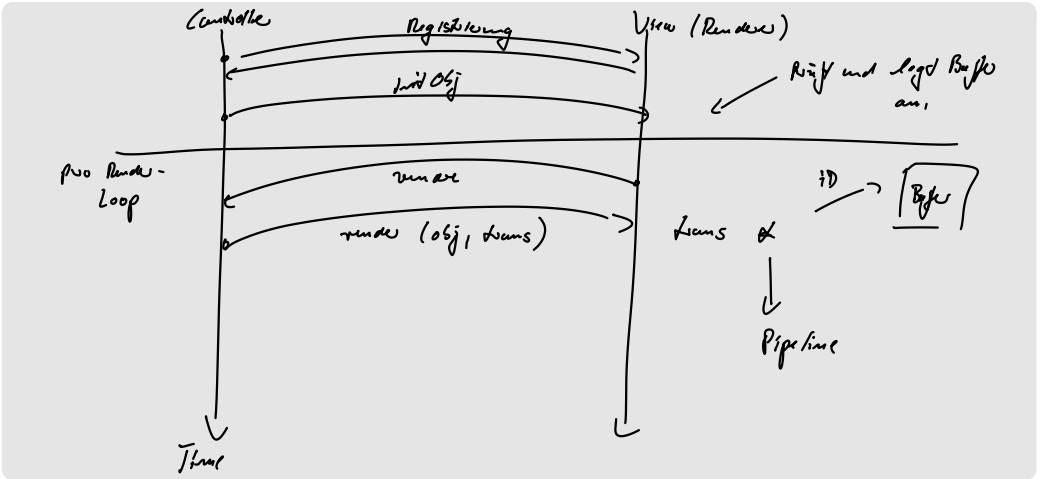
RenderableObject-SubKlassen für konkrete Objekt-Typen

```
class CgBaseTriangleMesh : public CgBaseRenderableObject
{
public:
    :
    virtual const std::vector<glm::vec3>& getVertices()       const =0;
    virtual const std::vector<glm::vec3>& getVertexNormals() const =0;
    virtual const std::vector<glm::vec3>& getVertexColors() const =0;
    :
    // length = 3*Anzahl Dreiecke
    virtual const std::vector<unsigned int>& getTriangleIndices() const =0;
    :
};
```

Umsetzung einer (abstrakten) Klasse um Rendering

```
class CgBaseRenderer
{
  public:
      // wird von Controller aufgerufen zum zeichnen
      virtual void render(CgBaseRenderableObject*,glm::mat4 world_coords)=0;

      // Modell in lokalen Koordinaten in Grafik-Buffer laden
      virtual void init(CgBaseRenderableObject*)=0;

      // Lege Kamera Sicht fest
      virtual void setLookAtMatrix(glm::mat4 lookat)=0;

       // Lege Projektionsart fest
      virtual void setProjectionMatrix(glm::mat4 proj)=0;

      // zum Initialisieren, damit Control und Renderer "sich kennen"
      virtual void setSceneControl(CgBaseSceneControl*)=0;
};
```

Zusammenspiel View-Control (Renderer) - Sequenzdiagramm



Zusammenspiel View-Control - (GUI)

- Aus Performance-Gründen sind Controller und Renderer unabhängig vom restlichen GUI direkt miteinander gekoppelt.
- GUI und Controller sind über das Observer-Pattern gekoppelt zum Weiterreichen von Maus- und Tastatur-Events, etc.
  - Hier werden für jede Aktion im GUI Events (Qt unabhängige Eigenimplementierung) an den Controller weitergeleitet.
  - Hier wird von GUI-Logik (z.B.: `slider34ValueChange(...)` auf Grafik-Logik gewechselt (z.B. `objectColorChange(...)`)
  - Hierzu müssen eigene Events implementiert werden, die unabhängig von der Art der Bedienung die entsprechende Aktion triggern
  - also egal ob Slider, Spinbox den Wert verändern: Aktion initiieren die damit aus Grafik-Sicht im System bezweckt werden soll (z.B. Farb-Änderung)

Zusammenspiel View-Control - (GUI)

- Selekt-Status (d.h. ob und welches Objekt ist selektiert) ist dem Controller bekannt, daher Farb-Event an Controller schicken
- Wie genau funktioniert das? → Übungsaufgabe: Recherche Observer-Pattern (ist schon implementiert, aber verstehen sollten Sie es trotzdem...)
- Entsprechende Erweiterungen dieser Event-Struktur sind Teil der Übungsaufgaben
- Ebenso muss der Controller beim View ein Neu-Zeichnen anstoßen können:

```
class CgBaseRenderer
{
    public:
      ⋮
       virtual void redraw()=0;
};
```