3 Einfache Geometrische Objekte

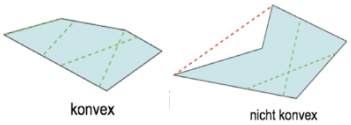
3.1 Polygone

Definition 3.1.1 (Polygon/face)

- Eine Menge von Tupel von Punkten wird als Polygonzug bezeichnet. Die Tupel von Punkten sind eine Kante des Polygonszuges. Der Polygonzug ist geschlossen, wenn der erste und letzte Punkt identisch sind. Das von dem Polygonzug umrandete Gebiet ist das Polygon.

Definition 3.1.1.1 (Planar)

- alle Punkte sind auf einer gemeinsamen Ebene



Definition 3.1.1.1 (Konvex)

- Bei einem konvexen Polygon ist jeder Eckpunkt von jedem Eckpunkt aus ''sichtbar''. D.h zwischen zwei beliebigen Eckpunkten innerhalb des Polygons verläuft (bei benachbarten Eckpunkten ist diese Verbindungslinie natürlich identisch mit eine Polygonkante).
- Anschaulich gesehen ist es nach außen gewölbt.

Definition 3.1.1. (einfaches Polygon)

- Ein Polygon ist einfach, wenn der Schnitt von 2 Kanten entweder die leere Menge ist oder ein Eckpunkt ist und jeder Endpunkt einer Kante darf maximal zu zwei Kanten des Polygons gehört.
- **Negativbeispiel**:
  - Also einmal eines wo sich zwei Kanten schneiden und einmal eines wo ein Punkt auf einem anderen liegt

3.1.6 Polygone - Eigenschaften – Konvexität

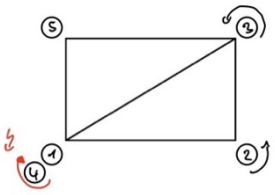
Planares, einfaches Polygon auf Konvexität testen:

(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.

- Test funktioniert je nach dem, wie man zeichnet beim **Haus von Nikolaus** nicht, obwohl der konvex ist
  - Haus des Nikolaus kann man nicht Anfangs- und Endpunkt gleich wählen. Man muss extra Punkt erzeugen
- **Am besten mit Richtungen erklären**!

Negativbeispiel: (konvexes planares Polygon) (nicht einfaches Polygon)

- Man hat von Punkt 1 bis 4 eine Linkskurve und von 4 nach 5 eine Rechtskurve, deswegen schlägt der Test fehlt.



Dreiecke:
für die Grafikkarte am besten, weil sie immer planar und konvex ist.

Vierecke:
Vierecke schöner zu modellieren, weil besitzt Längs- und Querrichtung

Definition 3.1.2 (Kreuzprodukt in der Computergraphik)

- Orientierung wichtig, um Vorder- und Rückseite zu bestimmen (bei Interaktion mit Licht)
- Für planare Polygone: Oberflächennormale n nach außen zeigend
- Polygon im oder gegen Uhrzeigersinn möglich, muss aber einheitlich sein für jedes Polygon, da Kreuzprodukt antikommutativ

Definition 3.1.3 (Backface Culling)

- Es wird verwendet, um die von der Kamera nicht gesehene Polygone nicht zu rendern.

Vorteile:

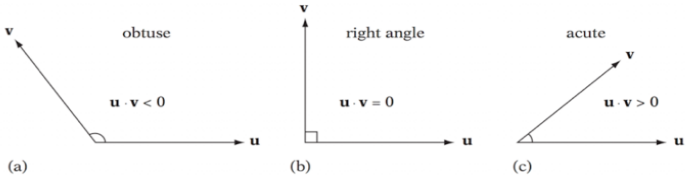
- Performance Gewinn → da abgewandte faces nicht verarbeitet werden müssen.
  Die geringere Anzahl zu zeichnender Flächen erhöht die Darstellungsgeschwindigkeit.
- Man kann das Objekt besser erkennen. (rein visuell)

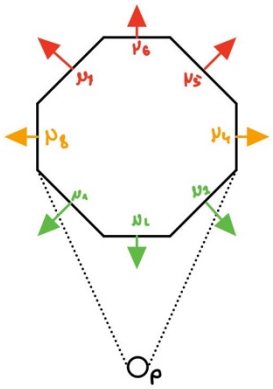
Definition 3.1.3.1 (Skalarprodukt)

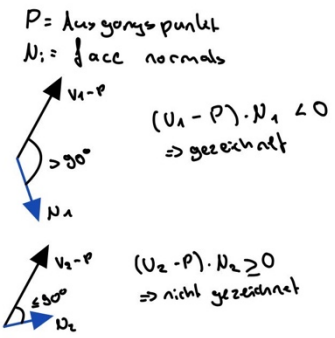
Das Skalarprodukt wird verwendet um die Winkelbeziehung zwischen der **Kamera P**, einem **Eckpunkt V** eines Polygons und **N seine Normale** zu bestimmen mit der **Formel (V – P) * N**.

- Das Vorzeichen des Skalarprodukts sagt, ob der Winkel zwischen den Vektoren

  a) **stumpf**, potenziell zur Kamera gewandt
  b) **ein rechter Winkel**, wird als abgewandt festgelegt
  c) **spitz**, abgewandt zur Kamera, also zeigt in die gleiche Richtung wie die Kamera
  => Polygon nicht sichtbar aus Kameraperspektive







3.2 Polygonale Netze

Definition 3.2.1 (Polygonales Netz)

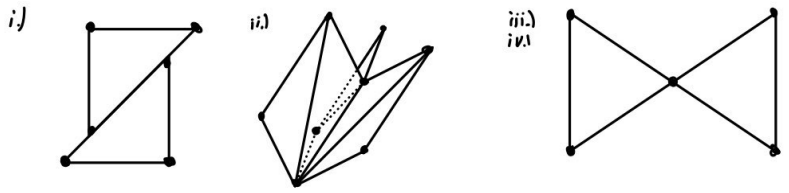
Annahme: Ein Polygon ist ein Dreieck! (immer definieren)

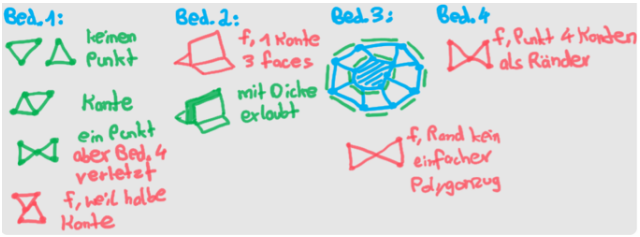
- Ein Polygonales Netz ist eine Menge von geschlossenen, planaren und einfachen Polygone.

Kriterien:

i. 2 Faces haben entweder keinen Punkt, einen Punkt oder eine ganze Kante gemeinsam.
ii. Jede Kante gehört zu einem oder maximal 2 Faces.
iii. Die Menge aller Kanten, die zu einem Face gehören bilden entweder eine leere Menge oder mehrere geschlossene und einfache Polygonzüge.
iv. Jeder Punt hat keine oder genau zwei Kanten, die zum Rand gehören.

Negativbeispiele:





Definition 3.2.2 (explizite Speicherung)

- Einzelnen Polygone werden mit der Sequenz von Punkten gespeichert

$$P_1 = ((v_{0x}, v_{0y}, v_{0z}), (v_{2x}, v_{2y}, v_{2z}), (v_{1x}, v_{1y}, v_{1z}))$$
$$P_2 = ((v_{0x}, v_{0y}, v_{0z}), (v_{1x}, v_{1y}, v_{1z}), (v_{8x}, v_{8y}, v_{8z}), (v_{7x}, v_{7y}, v_{7z}), (v_{5x}, v_{5y}, v_{5z}))$$
$$P_3 = ((v_{0x}, v_{0y}, v_{0z}), (v_{5x}, v_{5y}, v_{5z}), (v_{6x}, v_{6y}, v_{6z}), (v_{3x}, v_{3y}, v_{3z}))$$
$$P_4 = ((v_{3x}, v_{3y}, v_{3z}), (v_{4x}, v_{4y}, v_{4z}), (v_{0x}, v_{0y}, v_{0z}))$$
$$P_5 = ((v_{2x}, v_{2y}, v_{2z}), (v_{0x}, v_{0y}, v_{0z}), (v_{4x}, v_{4y}, v_{4z}))$$

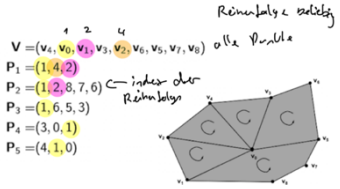
Nachteile:

- Wegen floating point Ungenauigkeit immer die Frage, ob wirklich der gleiche Punkt gemeint ist
- Hoher Speicherverbraucht für Redundanz
- Keine Speicherung gemeinsamer Punkte

Aufwand der Nachtbarschaftsbestimmung: O(n) – iteriere durch alle Polygone und prüfe ob P in der Liste ist.

Definition 3.2.3 (Eckenliste)

- Trennung von Geometrie und Topologie
- Die Eckenliste besteht aus einer Punktliste mit geometrischen Punkten und wahlfreiem Zugriff in beliebiger Reihenfolge, also enthält keine topologischen Informationen. Ein Polygon wird als Liste von Indizes auf die Punktliste definiert, in der die topologischen Informationen enthalten sind
- **Geometrie** beschreibt die räumliche Lage von Objekten wie z.B. ein Punkt im Raum definiert durch Koordinaten
- **Topologie** beschreibt die Struktur von Objekten wie Nachbarschaftsbeziehungen, Orientierung wie z.B. von einem Polygon
- **Vorteile**:
  - Eindeutigkeit der Punkte
  - Weniger Redundanz von Punkten → **weniger Speicherverbraucht**



Normalen Berechnen: Laufzeit

- Die Laufzeit ist O(n), weil zu einem Punkt nicht die benachbarten Kanten bzw. Polygone bekannt sind. Weshalb durch alle Polygone iteriert werden muss, um nach dem Punkt zu schauen. **n = Anzahl der Polygone!**

Strategie zur Berechnung: **M** = Anzahl der Vertices/vertices_normals, **N** = Anzahl der Polygone

1. Iteration über alle Vertices und vertice_normals auf (0,0,0) initialisieren O(M)
2. Iteration über alle Faces und für jeden Punkt des Faces die Flächennormale addieren O(N)
3. Iteration über alle vertice_normals um die zu normieren. O(M)

→ **MAX(O(N), O(M), O(M)) → O(n)**

Grundprobleme von der polygonale Netzdarstellung

- Netzauflösung bestimmt, wie eckig ein eigentlich rundes oder glattes Objekt wirkt.
- Runde Objekte wirken eckig, je nach Auflösung: viel Aufwand händisch genug Punkte und Kanten zu generieren

Mit welchen beiden alternativen Strategien kann man die beschriebene Problematik mindern?

- algorithmische polygonale Verfeinerung
- mathematische Objekte wie Kugel, etc. mit „unendlicher Auflösung“

3.3 Geometrische Grundprimitive

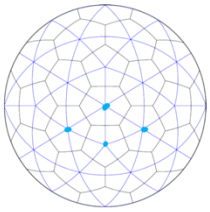
Grundidee: Benutze fest definierte primitive Objekte entweder direkt oder kombiniere diese zu komplexeren Gebilden.
- Nutze die Objekte als Hilfsmittel bei der Darstellung
- Nicht nur additiv kombinieren, sondern auch Differenz wie z.B. bei einem Würfel und Zylinder ein Loch bohren

Kugel:
- Hier mathematische Darstellung sinnvoll. Insgesamt nur 4 Variablen notwendig zur Beschreibung des Körpers. (Mittelpunkt und Radius)

Vorteil der mathematischen Darstellung:
- Beliebige Auflösung

Aufteilung der Kugeloberfläche in Polygone:

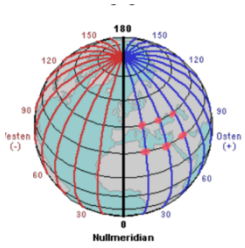
1. **Tesselierung durch 5- und 6-Ecken:**



Vorteil:
- Fläche gleichmäßig groß von Dreiecken durch die Schwerpunkte der Formen. → konstante Auflösung

Nachteil:
- Schwieriger zu konstruieren wegen Unregelmäßigkeit
- Verschiedene geo. Objekte. Man soll nur Dreiecke oder Vierecke haben

2. **Tesselierung durch die Längen- und Breitengerade**



- Durchlaufrichtung bzw. iterieren durch die Grade, viel pragmatischer

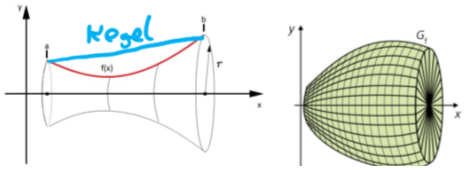
Vorteil:
- leichter zu erzeugen/iterieren

Nachteil:
- unterschiedliche Flächen der einzelnen Faces → keine Konstante Auflösung
- Unterschiedliche Polygone → Vierecke und Dreiecke

Gründe warum einfache Objekte in der Computergrafik in ihrer mathematischen Darstellung verwendet werden, obwohl Grafik-Karten nur Dreiecke performant verarbeiten können:
- Level-Of-Detail Möglichkeit: Beliebige Auflösung bei Tesselierung
- Einfache und performante Kollisionsberechnung
- Volumetrische Sicht möglich, CSG möglich
- Einfache und performante Transformation (z.B. Verschiebung: nur Mittelpunkt)

Rotationskörper

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



1. Die Achse schneidet die Kurve nicht: (offener Körper)
   - Netz-Modellierung mit Vierecken

2. Die Achse schneidet Anfangs- und Endpunkt der Kurve: (geschlossener Körper)
   - Netz-Modellierung mit Vierecken und Dreiecken

Sweep- Körper?
- Die Idee: Verschiebe eine (geschlossene) Kurve bzw. Querschnittsfläche auf einer gekrümmten Leitkurve durch den Raum

3.4 Polygonale Verfeinerung

Idee: Einen Polygonzug durch Hinzufügen von Punkten verfeinern

Definition 3.4 (Polygonale Verfeinerung)

- Einen Polygonzug durch Hinzufügen von Punkten verfeinern, diese werden bei iterativer Anwendung eine glatte Kurve erzeugen.
- Der polygonale Verfeinerungsprozess ist ein Schema das Kontroll-Polygone Pj,k erzeugt nach der Formel:

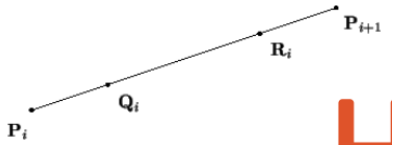
$$\mathbf{P}_j^k = \sum_{i=0}^{n_{k-1}} \alpha_{i,j,k} \mathbf{P}_i^{k-1}$$

und jeder Kontrollpunkt Pj,k als Linearkombination der Kontrollpunkte des vorherigen Kontrollpolygons Pj,k-1 berechnet nach einer Gewichtung alpha

Definition 3.4.1 (Chaikins Algorithmus)

Ein gegebenes Kontrollpolygon {P0, P1, ..., Pn}, wird verfeinert, indem eine Sequenz von neuen Kontrollpunkten {Q0, R0, Q1, R1, ..., Qn−1, Rn−1} iterativ erzeugt wird.

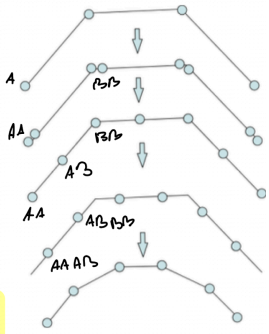
$$\mathbf{Q}_i = \frac{3}{4}\mathbf{P}_i + \frac{1}{4}\mathbf{P}_{i+1} \qquad \mathbf{R}_i = \frac{1}{4}\mathbf{P}_i + \frac{3}{4}\mathbf{P}_{i+1}$$



- **Resultat**: Kurve wird kleiner (2*n-2)
- **Nachteil**: verzerrte Punkte, bei Messdaten kritisch. Nur beim Modellieren sinnvoll

Alternative Sichtweise für dieselbe erzeugte Kurve:

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

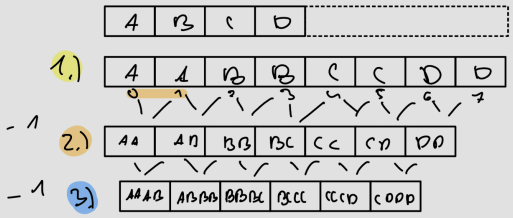


Warum ist das identisch?

$\bar{p} = \frac{1}{2} \cdot p_1 + \frac{1}{2} \cdot p_2 \quad \Big| \quad \frac{1}{2} \cdot p_1 + \frac{1}{2} \cdot \bar{p}$

$\Rightarrow \frac{1}{2} \cdot p_1 + \frac{1}{2} \cdot \left(\frac{1}{2} \cdot p_1 + \frac{1}{2} \cdot p_2 \right) = \frac{3}{4} \cdot p_1 + \frac{1}{4} \cdot p_2$

Umsetzung mit Listen:



Wie kann mit dem Chaikins-Algorithmus für jeden vorgegebenen Polygonzug eine beliebig glatte Kurve (also eine, bei der auf dem Bildschirm keine Polygonale „Eckigkeit“ mehr sichtbar ist) erzeugt werden? (TR, 2 P)

- So viele Unterteilungsiterationen machen dass die längste Linie kleiner als ein Pixel ist.

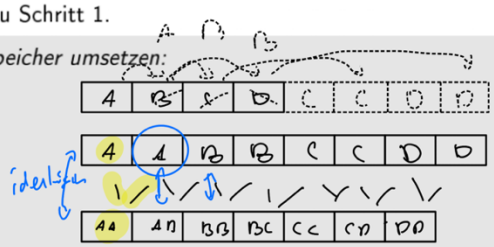
Definition 3.4.2 (Lane-Riesenfeld Algorithmus)

- Verdoppeln iterieren von hinten, da sonst überschrieben wird und Punkte verloren gehen
- n-mal Wiederholung als Parameter umsetzten → Üblich sind 2-4 Wiederholungen
- Man darf mehrmals mitteln, aber nicht zu oft machen, da immer ein Punkt verschwindet und man nicht weniger Punkte haben will, als man angefangen hat. Ziel ist mehr Punkte zu erzeugen
- Am Ende gibt es das letzte Element nicht mehr bzw. wird ignoriert

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:



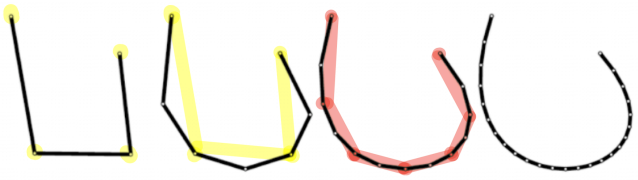
Wie oft darf pro Iteration (Lane Riesenfeld) gemittelt werden, dass bei einer punktlänge N die Kurve nicht ganz verschwindet.

- original_size + 1 → (n-1)

Unterteilungskurven - 4-Punkt-Schema

- Vorteil: Ursprungspunkte werden beibehalten
- Wächst nach außen, um glatt zu werden

- 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.



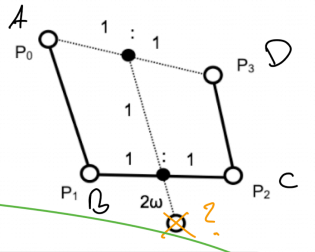
- Wird diese Platzierung sinnvoll gewählt, wird das Objekt sehr schnell „glatt".

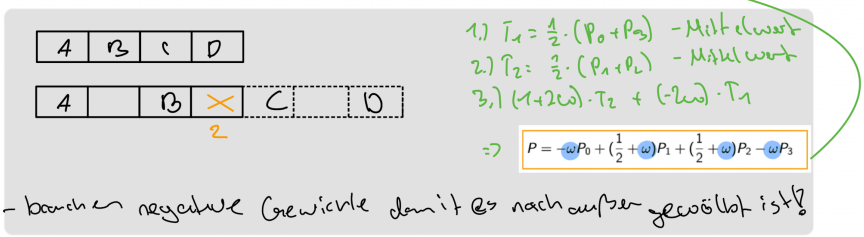
- Wächst nach außen, um glatt zu werden → wird nach außen gewölbt!
- Punkte werden behalten.

Position der neuen Punkte basierend auf 4 vorhanden Punkten:

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

ω: Parameter für die „Bauchigkeit" der Kurve



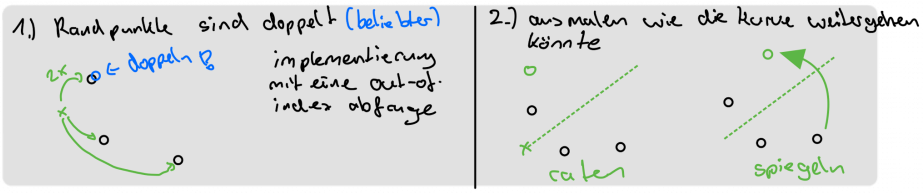


- Gewichtetes mitteln von Nachbarschaft, +-1 und +-2 links und rechts Punkte anschauen bzw. Index +-1 und +-3 für 4 Punkte
- Innerhalb der gestrichelten Linie z.B. 50% von Pa und 50% von Pb. Um geometrisch außerhalb der Form zu kommen, muss man über 100% Gewichtung und der andere Punkt muss eine negative Gewichtung haben um wieder auf 100% zu kommen, um Mittelwert zu berechnen
- Auf neue Liste zählen, bei Iterationen und immer i+=2

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:



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: $P_{2i+1} = -\omega P_{2i} + (\frac{1}{2} + \omega)P_{2i-2} + (\frac{1}{2} + \omega)P_{2i+2} - \omega P_{2i+4}$

P0 P1 P2 P3