Einführung

Anwendungen

Das Thema «Computer Grafiken» lässt sich in vielen Gebieten antreffen, wie z.B.:

- Videospiele
- Cartoons & Filme
- Datenvisualisierungen
- Berechnungen

Standards

Im Bereich «Computer Grafiken»:

- **Treiber APIs**: OpenGL, DirectX, Vulkan
- **Bare API Wrappers**: OpenTK, JOGL, WebGL
- **Mid-Level APIs**: Three.js, SharpGfx
- **Rendering Engines**: Renderman, Mental Ray
- **Modelling Software**: Blender, Maya
- **Game Engines**: Unity, Unreal

Vektorgeometrie

Punkte vs. Vektoren

Grundsätzlich sind alle Punkte **Ortsvektoren** durch den Ursprung. Es gilt daher:

$$P = \vec{0} + \vec{p} = \vec{p}$$

Operationen

Addition / Subtraktion

$$\vec{a} + \vec{b} = \begin{pmatrix} a_1 + b_1 \\ a_2 + b_2 \\ \dots \end{pmatrix}$$

Skalarmultiplikation

$$r \cdot \vec{a} = \begin{pmatrix} r \cdot a_1 \\ r \cdot a_2 \\ \dots \end{pmatrix}$$

Kreuzprodukt

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

Transponieren

$$\begin{pmatrix} a_1 \\ a_2 \\ \dots \end{pmatrix}^T = (a_1, a_2, \dots)$$

Euklidische Norm (Länge)

$$|\vec{a}| = \sqrt{a_1^2 + a_2^2 + \dots}$$

Normalisierung

$$\hat{a} = \frac{1}{|\vec{a}|} \cdot \vec{a}$$

Skalarprodukt

$$\vec{a} \circ \vec{b} = \sum_i (a_i \cdot b_i) = |\vec{a}| \cdot |\vec{b}| \cdot \cos\alpha$$

⇨ Ist $\vec{a} \circ \vec{b} = 0$, sind die Vektoren orthogonal.
⇨ Orthogonal: Vektoren stehen senkrecht aufeinander.

Multiplikation

Allgemein **nicht** kommutativ:

$$\vec{a} \cdot \vec{b} \neq \vec{b} \cdot \vec{a} \qquad A \cdot B \neq B \cdot A$$

$$AB = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

Gleichungssysteme

Allgemeine Definition:

$$Ax + b \Leftrightarrow \begin{matrix} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \end{matrix}$$

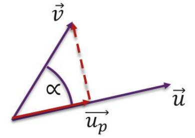
Gauss-Verfahren

$$[A \mid b]: \quad \left[\begin{array}{cc|c} 1 & 4 & 2 \\ 2 & 9 & 5 \end{array}\right] \Rightarrow \left[\begin{array}{cc|c} 1 & 0 & -2 \\ 0 & 1 & 1 \end{array}\right]$$

Orthogonale Projektion

$$\vec{u}_p = \left(\frac{\vec{u} \circ \vec{v}}{|\vec{u}|^2}\right) \cdot \vec{u}$$

$$= |\vec{v}| \cdot \cos\alpha \cdot \hat{u}$$



3D Geometrien

Bestandteile von 3D-Objekten

3D-Objekte («Meshes») bestehen im Allgemeinen immer aus diesen Elementen:

- Eckpunkte (Vertices): $V \in \mathbb{R}^3$
- Linien (Edges): $E \in (V_1, V_2)$
- Oberflächen (Faces): $F \in (V_1, V_2, V_3)$

⇨ Meist werden Dreiecke für die Faces verwendet. Vorteile: Garantiert flach, eindeutige Definition, einfache Transformation
⇨ Eckpunkte können definiert oder berechnet werden.

Indexing

Die Punkte V der Fläche F lassen sich auf verschiedene Arten referenzieren:

- **Ohne** Indexing:
  - 1 Punkte-Array ($l = 9 \cdot n_F$)
  - 3 Koordinaten pro Punkt
  - 3 Punkte pro Fläche
- **Mit** Indexing:
  - 1 Punkte-Array ($l = 3 \cdot n_V$)
  - 1 Index-Array ($l = 3 \cdot n_F$)
  - 3 Koordinaten pro einzigartigen Punkt
  - 3 Indexe pro Fläche

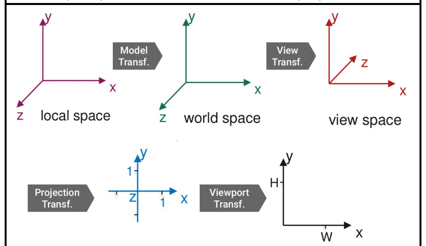
⇨ Mit Indexing ist meistens effizienter als ohne Indexing.

Koordinatensysteme

Ein Punkt einer Geometrie kann je nach Ansichtsweise von verschiedenen Koordinatensystemen referenziert werden:

1. Modell (3D / Rechtshändig)
2. Welt (3D / Rechtshändig)
3. Kamera (3D / Linkshändig)
4. Sichtbarkeitsbereich (2D)
5. Bildschirm (2D)

⇨ Bei der Darstellung werden diese Punkte umtransformiert.
⇨ z.B. $P(1,3,2)$ steht auf dem Bildschirm an $P(5,4)$.



Transformation

Transformationen können **sukzessiv** oder **gemeinsam** angewandt werden.

⇨ Die nachfolgenden Beispiele sind alle in 2D.
⇨ Weitere Transformationen sind Spiegelung und Scherung.

Translation

Verschiebe alle Punkte einer Geometrie um einen **Vektor** (Vektoraddition).

$$T(\vec{x}) = \begin{pmatrix} x + d_1 \\ y + d_2 \end{pmatrix} = \vec{x} + \vec{d}$$

Skalierung

Verschiebe alle Punkte einer Geometrie um einen **Faktor** (Skalarmultiplikation).

$$S(\vec{x}) = \begin{pmatrix} s \cdot x \\ s \cdot y \end{pmatrix} = s \cdot \vec{x}$$

⇨ Die Faktoren s können auch unterschiedlich sein (s. Matrix).

Rotation

Rotiere alle Punkte einer Geometrie um einen **Winkel** θ.

$$R_\theta(\vec{x}) = \begin{pmatrix} x \cdot \cos\theta - y \cdot \sin\theta \\ x \cdot \sin\theta + y \cdot \cos\theta \end{pmatrix}$$

⇨ Die 3D-Berechnung ist in diesem Modul nicht relevant.

Gesamt-Transformation

Aus Effizienzgründen würden wir gerne die Transformationen zuerst zusammenrechnen und dann auf alle Punkte anwenden.

Problem: Die Translation ist keine lineare Abbildung. Das bedeutet:

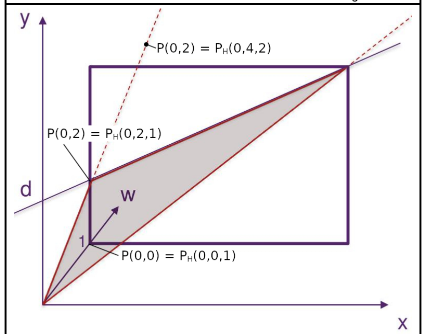
$$s \cdot \left(\vec{d} + \vec{x}\right) \neq \left(s \cdot \vec{d}\right) + \vec{x}$$

⇨ D.h.: Sukzessive Anwendung ist nicht gleich gemeinsame.

Homogene Koordinaten

Um das Problem der Translation zu lösen, werden alle kartesischen Koordinaten $P(x, y)$ auf homogene Koordinaten $P_H(x, y, 1)$ abgebildet.

⇨ Oder Allgemeiner: $P(x, y, w)$ repräsentiert $P(x/w, y/w)$.
⇨ Die Punkte werden so zu Linien im projektiven Raum.
⇨ Die Translation wird damit zu einer linearen Abbildung.



Translation Matrix

$$\begin{pmatrix} 1 & & d_1 \\ & 1 & d_2 \\ & & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x + d_1 \\ y + d_2 \\ 1 \end{pmatrix}$$

Skalierung Matrix

$$\begin{pmatrix} s_1 & & \\ & s_2 & \\ & & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} s_1 \cdot x \\ s_2 \cdot y \\ 1 \end{pmatrix}$$

Rotation Matrix

$$\begin{pmatrix} \cos\theta & -\sin\theta & \\ \sin\theta & \cos\theta & \\ & & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \\ 1 \end{pmatrix}$$

Gesamt-Transformation Matrix

Wir können nun die einzelnen Transformationen miteinander multiplizieren und erhalten so die Gesamt-Transformation:

$$M_R \cdot (M_S \cdot \vec{x}) = (M_R \cdot M_S) \cdot \vec{x}$$

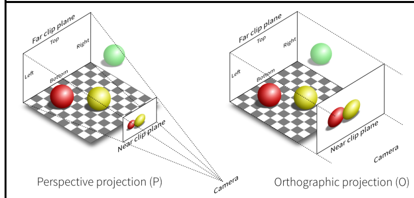
⇨ Die Reihenfolge spielt weiterhin eine Rolle: $M_R M_S \neq M_S M_R$

Projektionen

Definition

Um ein 3D-Objekt auf einem 2D-Bildschirm darzustellen, müssen wir es zuerst in diese 2D-Dimension projizieren. Wir unterscheiden dabei:

- Perspektivische Projektion
- Orthogonale Projektion



Berechnung

Projiziere den Punkt $P(x, y, z)$ auf die XY-Ebene ($z = 0$) basierend auf der Kameraposition $E(e_x, e_y, e_z)$.

Gesucht ist die Projektion $C(c_x, c_y)$.

⇨ Die Dimension wird also um eins reduziert ($\mathbb{R}^3$ zu $\mathbb{R}^2$).
⇨ Bei 2D einfach eine Komponente (z.B. x) weglassen.

Perspektivisch

$$c_x = \frac{e_x z - e_z x}{z - e_z} \qquad c_y = \frac{e_y z - e_z y}{z - e_z}$$

⇨ Herleitung aus der Formel $y = \Delta y / \Delta z \cdot z + c$

Orthogonal

$$c_x = x \qquad c_y = y$$

View Frustum

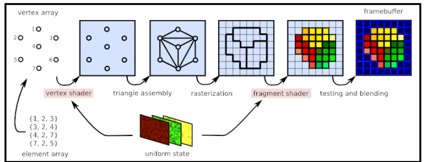
Bezeichnet die Sichtbarkeit (Clip-Space) bei der perspektivischen Projektion. Es wird definiert durch:

- Öffnungswinkel (Field of View)
- Seitenverhältnis (Aspect Ratio)
- Near und Far-Plane (Clipping Distance)

⇨ Der Öffnungswinkel bestimmt die Grösse von Objekten.
⇨ Die Brennweite (Kamera) bestimmt die Tiefenschärfe.

GPU-Berechnung

Grafik-Pipeline



Double Frame Buffering

Beschreibt die **abwechselnde** Verwendung von zwei Framebuffer für die Berechnung und Darstellung eines Frames.

- **Frame**: Bild auf dem Display
- **Framebuffer**: Speicherort des Frames

⇨ Berechnungen werden nicht auf dem Anzeigebild durchgeführt.
⇨ So können Berechnungsartefakte vermieden werden.

Shader-Programme

Sind auf der GPU laufende Programme für die Berechnung des Bildes. Es gibt:

- **Vertex-Shader**: Projektion der Modell-Eckpunkte in den Clip-Space.
- **Fragment-Shader**: Berechnung der Farbe eines Pixels.

⇨ Arbeiten immer mit **einzelnen** Primitiven (z.B. **ein** Eckpunkt).

GLSL Programmiermodell

Kommunikation in den Pipeline-Stages:

- `in`: Aus vorherigem Stage
- `out`: An nächsten Stage
- `uniform`: Für alle Primitiven gleich

```
in vec3 positionIn;
in vec3 normalIn;
out vec3 normal;

// Transformations to Clip-Space
uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main() {
  // Homogene Transformation
  gl_Position = vec4(positionIn, 1.0)
              * model
              * view
              * projection;

  // Component-wise multiplication
  normal = vec4(normalIn, 1.0) * model;
}
```

vertex-shader.glsl

⇨ in und out verwenden dabei «matching by name».

Beleuchtung & Texturen

Allgemeines

Die Farbe eines Objekts (bzw. Pixels) setzt sich zusammen aus:

- Den Objekt-Farben / Texturen
- Der Beleuchtung

⇨ Oftmals verwenden wir dabei RGB-Farben: $C = (R, G, B)$.
⇨ Remission: Beschreibt das Abprallen von Licht auf Objekten.

Farbdarstellung

Subtraktive Farbberechnung

Nur die Farbanteile, welche in Lichtquelle und Objekt vorkommen, sind sichtbar:

$$C_{\text{Total}} = \begin{pmatrix} R_{\text{Light}} \cdot R_{\text{Object}} \\ G_{\text{Light}} \cdot G_{\text{Object}} \\ B_{\text{Light}} \cdot B_{\text{Object}} \end{pmatrix}$$

Alternative mit gemittelten Werten:

$$C_{\text{Total}} = \frac{1}{2} \cdot (C_{\text{Light}} + C_{\text{Object}})$$

⇨ Subtraktiv, da die fehlenden Farben **nicht** remittiert werden.

Additive Farbberechnung

Die Farbanteile der Lichtquellen werden zusammengerechnet:

$$C_{\text{Total}} = \vec{1} - \begin{pmatrix} (1 - R_{\text{L1}}) \cdot (1 - R_{\text{L2}}) \\ (1 - G_{\text{L1}}) \cdot (1 - G_{\text{L2}}) \\ (1 - B_{\text{L1}}) \cdot (1 - B_{\text{L2}}) \end{pmatrix}$$

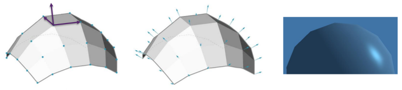
⇨ Die **nicht enthaltenen** Lichtanteile werden reduziert.

Oberflächennormale

Nicht-triviale Belichtungsmodelle berücksichtigen die Ausrichtung der Oberfläche:

$$\mathrm{N}_{V_1} = (V_2 - V_1) \times (V_3 - V_1)$$

⇨ Normale eines Vertex V_1 von einer Fläche $F \in (V_1, V_2, V_3)$
⇨ Dieser Wert wird nun auf die Fläche F interpoliert.
⇨ Kann im voraus oder «on-the-fly» berechnet werden.



eine Normale pro Fläche — gemittelte Normalen pro Eckpunkt — interpolierte Darstellung

Beleuchtungsmodelle

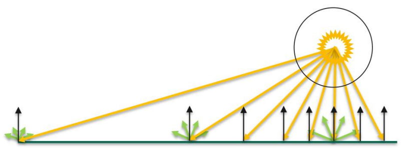
Ambient Lighting

Belichtung von einem globalen Licht mit Remission in alle Richtungen.

```
void main() {
  vec3 ambient = strength * lightColor;
  vec3 color = ambient * objectColor;
  fragColor = vec4(color, 1.0);
}
```
ambient-fragment-shader.glsl

Diffuse Lighting

Belichtung von einer Punktquelle mit Remission in alle Richtungen.

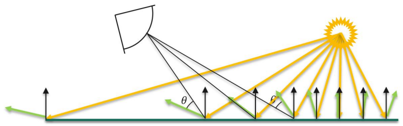


```
void main() {
  vec3 normDir = norm(normal);
  vec3 lightDir = norm(lightPos - fragPos);
  float cosTheta
    = max(dot(normDir, lightDir), 0.0);
  vec3 diffuse = cosTheta
               * lightColor
               * objectColor;
  fragColor = vec4(diffuse, 1.0);
}
```
diffuse-fragment-shader.glsl

⇨ Wird für **matte** Oberflächen verwendet.
⇨ Das norm steht für die Funktion normalize.

Specular Lighting

Belichtung von einer Punktquelle mit Remission in **eine** Richtung.



```
void main() {
  vec3 normDir = norm(normal);
  vec3 camDir = norm(camPos - fragPos);
  vec3 lightDir = norm(lightPos - fragPos);
  vec3 reflectDir
    = reflect(-lightDir, normDir);
  float cosTheta
    = max(dot(camDir, reflectDir), 0.0);
  float strength = pow(cosTheta, shininess);
  vec3 specular = strength
                * lightColor
                * objectColor;
  fragColor = vec4(specular, 1.0);
}
```
specular-fragment-shader.glsl

⇨ Wird für **spiegelnde** Oberflächen verwendet.

Kombinationsmodelle

Phong-Shading

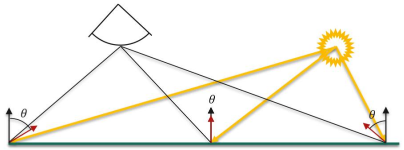
Die Belichtung wird aus Ambient-, Diffuse- und Specular-Anteilen zusammengesetzt.

$$C_{\text{Total}} = \frac{1}{3} \cdot (C_{\text{Ambient}} + C_{\text{Diffuse}} + C_{\text{Specular}})$$

⇨ Problem: Ab 90° gibt es keine Spiegelung mehr.

Blinn-Phong-Shading

Löst das Problem von Phong-Shading durch die Verwendung eines sogenannten «Halfway-Vectors».



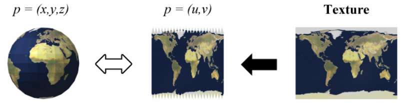
```
void main() {
  ...
  vec3 halfwayDir = norm(lightDir + camDir);
  float cosTheta
    = max(dot(normDir, halfwayDir), 0.0);
  ...
}
```
blinn-phong-fragment-shader.glsl

Texturen

Texturen sind Bilddateien, welche Eigenschaften (wie z.B. die Farbe) einer Oberfläche definieren.

Texture-Mapping

Beschreibt die Abbildung von 3D-Vertex-Koordinaten auf 2D-Textur-Koordinaten.



⇨ Auch UV-Mapping genannt.
⇨ Sampling: Umrechnung von Fragment- in Texturkoordinaten.

```
void main(void) {
  fragColor = texture(texUnit, texCoord);
}
```
texture-fragment-shader.glsl

Komplexe Oberflächen

Grundformen

3D-Objekte lassen sich wie bisher durch Punkte, aber auch durch Funktionen beschreiben:

- **Funktionen**: Kontinuierlicher Wertebereich
  - Explizit: $z = -ax + by + \ldots$
  - Implizit: $0 = x^2 + 2y^2 + \ldots$
  - Parametrisch: $P = \vec{0} + s\vec{u} + \ldots$
- **Punkte**: Festgelegter Wertebereich

⇨ Explizite Funktionen sind nach einer Variablen aufgelöst.
⇨ Implizite sind nicht aufgelöst (algebraische Oberflächen).
⇨ Algebraische Oberflächen: Sphäre, Torus, Würfel, etc.

Kombinationen

Punkte und Funktionen sind die Grundbausteine für alle komplexen Formen:

- **Aus Primitiven**: Punktwolke, Meshes
- **Approximierend**: Iso-Surface, Splines
- **Konstruiert**: Subdivision Surfaces, Fraktale

Vor- und Nachteile

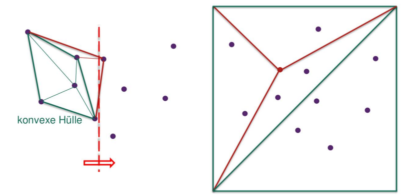
Es gibt keine beste Repräsentationsform für Objekte. Vor- und Nachteile sind:

- **Funktionen**:
  - **Vorteile**: Wenig Speicherplatz, Schnittpunkte mathematisch berechenbar, beliebig genaue Auflösung
  - **Nachteile**: Beschränkte Formen, komplexe Herleitung, grafische Transformationen sind schwierig
- **Punkte**:
  - **Vorteile**: Beliebige Geometrie, vielseitig einsetzbar, direkter GPU-Support, einfache Berechnung
  - **Nachteile**: Fixe Genauigkeit, hoher Speicherbedarf, Rechenzeit abhängig von der Anzahl Primitiven

Triangulation

Beschreibt die Umwandlung einer Punktwolke in ein Polygon-Mesh.

⇨ Die Oberflächen werden rekonstruiert / approximiert.
⇨ Wird z.B. bei Rohdaten von 3D-Scans angewandt.



Sweep-Strategie — Insert-Strategie

Sweep-Strategie

1. Laufe von **links** nach **rechts**.
2. Für jeden Punkt:
   a. Zeichne eine Linie zu den **2** vorherigen Punkten, für die gilt:
      - Keine Dellen entstehen
      - Keine Überschneidungen entstehen
   b. Verbinde nun alle weiteren Punkte innerhalb dieser Form.
3. Wiederhole, bis zum Ende.

⇨ Die entstehende Form nennt sich «Konvexe Hülle».

Insert-Strategie

1. Zeichne **2** Anfangsdreiecke um alle Punkte.
2. Für alle Punkte (zufällige Wahl):
   a. Bestimme das umfassende Dreieck.
   b. Unterteile dieses Dreieck in 3 weitere Dreiecke. D.h. Verbinde alle Eckpunkte mit dem gewählten Punkt.
3. Wiederhole, bis zum Ende.
4. Entferne nun alle künstlichen Anfangs**punkte** und die damit verbundenen Dreiecke.

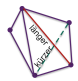
Probleme

Beide Strategien können «unschöne», d.h. spitze Dreiecke erzeugen.

⇨ Wir können dies mit «Delaunay» nachträglich verbessern.
⇨ Teilweise lassen sich spitze Winkel jedoch nicht vermeiden.

Delaunay Triangulation

1. Rekursiv für alle Dreiecke:
   a. Wähle ein anliegendes Dreieck
   b. Ersetze die längere der inneren Kanten durch die Kürzere. (Edge-Flip)
2. Wiederhole, bis zum Ende.



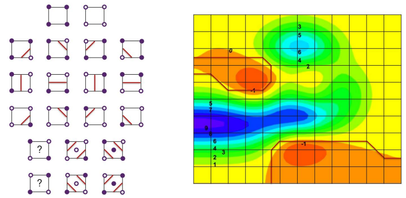
Approximationen

Marching Squares Algorithmus

Mit diesem Algorithmus lassen sich Isolinien von Heat Maps diskret bestimmen.

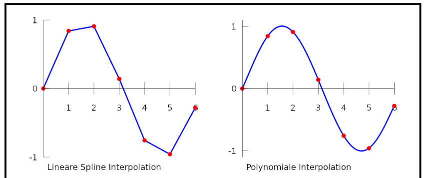
1. Gitter über die Daten legen.
2. Betrachtungshöhe (Potenzial) festlegen.
3. Für alle Quadrate im Gitter:
   a. Eckpunkte beachten.
   b. Nach Schema unten Linien einzeichnen.
4. Wiederhole, bis zum Ende.

⇨ «Heat Map»: 2D-Visualisierung von 3D-Landschaften.
⇨ «Isolinien»: Die Höhenlinien einer Heat Map.



Weitere Algorithmen

- Marching Cubes (3D-Heat-Maps)
- Interpolation: Punkte «vervollständigen»
  - Polynomial: $f = a_0 x^0 + \ldots + a_n x^n$
  - Splines: Stückweise Interpolation der Punkte mit linearen, quadratischen oder kubischen Funktionen.
- NURBS: Approximation von 3D-Flächen



Lindenmayer Systeme

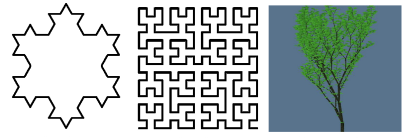
L-Systeme beschreiben beliebig feine, selbstähnliche geometrische Strukturen.

⇨ Sie können rekursiv definiert und aufgebaut werden.

Formale Definition

- Anfangsform (z.B. Strich): f
- Ersetzungsregeln: $f \rightarrow f + f - - f + f$
  - Ersetzungsmöglichkeit: f
  - Positive Rotation: $+$
  - Negative Rotation: $-$
  - Abzweigung (Kind): $[f]$
- Kontext: Rotation $60°$

⇨ Beispiele: Koch Kurve, Hilbert Kurve, Fraktale, etc.
⇨ So lassen sich u.a. Bäume generieren (z.B. mit Zufallszahlen).



Koch Kurve — Hilbert Kurve — Lindenmayer Baum

Subdivision Surfaces

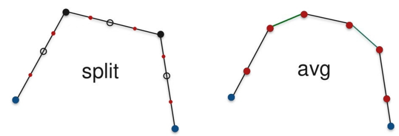
Beschreibt ein rekursives Verfahren für das Verfeinern von Oberflächen.

⇨ Subdivision Curves ist das Äquivalent für Kurven.

Curves: Chaikin's Algorithmus

1. Beginne mit einer Kurve
2. Markiere die Anfangspunkte (Blau)
3. Setze in der Mitte von **allen** Strecken einen neuen Punkt (Schwarz ohne Füllung)
4. Setze nun in der Mitte von allen **neuen** Strecken einen Punkt (Rot)
5. Streiche nun alle schwarzen Punkte und verbinde die Roten und Blauen.
6. Wiederhole, solange wie gewünscht.

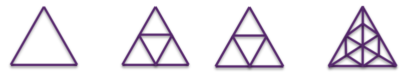
⇨ Die neuen Punkte stehen an 1/4 und 3/4 der Originalstrecke.
⇨ Diese Gewichtung kann auch variiert werden.



Surfaces: Algorithmen

Dreiecksbasiert

Loop — $\sqrt{3}$ Subdivision



Rechtecksbasiert

Catmull-Clark — Doo-Sabin

Vorteile

Vorteile von Subdivision-Surface, insbesondere im Vergleich zu NURBS:

- Beliebige Oberflächentopologie
- Kompakte Repräsentation
- Level-of-Detail Rendering
- Intuitiv mit einfachen Algorithmen

⇨ NURBS-Flächen können nur Scheiben, Zylinder oder Tori sein.

Korrektur & Optimierung

TODO

Qualitätsmerkmale

Mesh Smoothing

Mesh Reduktion / Remeshing

Diese Verfahren haben das Ziel, die Anzahl der Oberflächen zu reduzieren.

Vertex Clustering

1. Wähle ein Grösse epsilon (Toleranz)
2. Teile den Raum in Quadrate dieser Grösse
3. Berechne pro Quadrat **einen** repräsentativen Eckpunkt (z.B. Mittelpunkt aller Punkte)
4. Lösche die originalen Punkte und ersetzte sie durch den neuen Eckpunkt.

Je nach Berechnungsverfahren des repräsentativen Eckpunkts kann sich die Topologie des Meshes stark unterscheiden.

⇨ Das Verfahren spielt also eine starke Rolle für die Qualität

Inkrementelle Reduktion

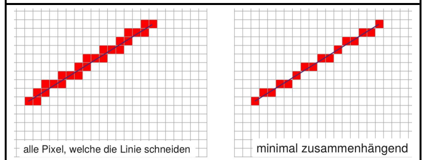
Resampling / Remeshing

Rasterisierung & Sichtbarkeit

Rasterisierung

Da ein 2D-Bildschirm aus Pixeln besteht, müssen wir nach der Projektion die Linien noch in ein Raster abbilden. Es gibt verschiedene Methoden dazu:

- Vollständig Zusammenhängend
- Minimal Zusammenhängend
  - Aliased (Binär)
  - Anti-Aliased (Prozentual)



Aliasing

Zeichne ausschliesslich die Pixel eines Dreiecks, für die gilt:

- Das Pixel-Zentrum liegt **in** dem Dreieck.
- Das Pixel-Zentrum liegt **auf** der oberen oder linken Seite des Dreiecks.

⇨ Achtung: Die obere Seite muss dazu exakt horizontal sein.
⇨ Technisch wird das Dreieck **zeilen**weise gezeichnet.
⇨ Dazu wird u.a. der Bresenham Linien-Algorithmus verwendet.

Bresenham Linien-Algorithmus

Basierend auf zwei Punkten P_{Start} und P_{Ende}, zeichne die Linie nach dem Bresenham Linien-Algorithmus:

1. Berechne $\Delta x = x_{\text{Ende}} - x_{\text{Start}}$
2. Berechne $\Delta y = y_{\text{Ende}} - y_{\text{Start}}$
3. Berechne $m = \Delta y / \Delta x$
4. Wenn $\Delta x \geq \Delta y$ dann mit $i = 0$:
   a. $x_i = x_{\text{Start}} + i$
   b. $y_i = y_{\text{Start}} + \lfloor m \cdot i + 0.5 \rfloor$
   c. Zeichne den Pixel $P(x_i, y_i)$
   d. $i \leftarrow i + 1$

⇨ Bei $\Delta x < \Delta y$ wird die Berechnung von x_i und y_i vertauscht.

Anti-Aliasing

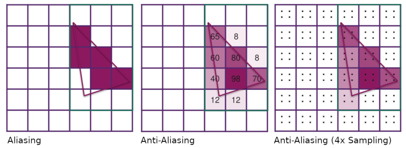
Zeichne alle Pixel eines Dreiecks unter Beachtung der prozentualen Abdeckung. Das bedeutet:

- Erhöhe das Pixelraster (z.B. 4x)
- Berechne die Abdeckung nach Aliasing
- Reduziere das Pixelraster und zeichne alle Pixel anhand der berechneten Abdeckung.

Varianten davon sind:

- **Super-Sampling**: Die komplette GPU-Pipeline läuft mit einem erhöhten Pixelraster.
- **Multisampling**: Nur der Z-Buffer läuft mit einem erhöhten Pixelraster.

⇨ Die Objektränder erhalten also eine «weiche» Transparenz.



Probleme (Aliasing Effekte)

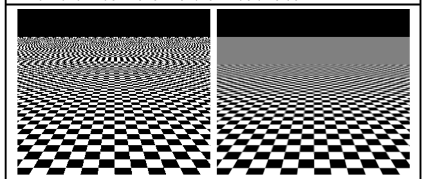
Wenn die Auflösung eines Texturmusters grösser ist als die Auflösung der Anzeigefläche, kann der Moiré-Effekt auftreten.

⇨ Dies ist bei beiden Aliasing-Verfahren der Fall.
⇨ Problem: Ein Pixel alleine kann kein Muster darstellen.

Mipmaps

Beschreibt eine «Pyramide» von Texturen, bei der die Auflösung anhand der Distanz zur Kamera gewählt wird.

⇨ Je näher das Objekt, desto hochauflösender die Textur.
⇨ Damit kann der Moiré-Effekt verhindert werden.



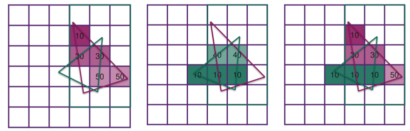
Sichtbarkeit

Z-Buffer (Depth-Buffer)

Erlaubt das korrekte Zeichnen von überlappenden Objekten.

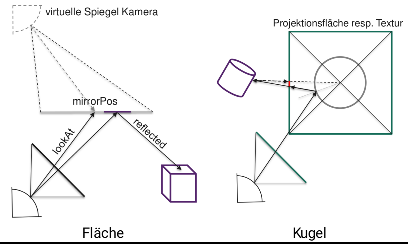
- Initialisiere den Buffer mit $Z_{\text{B}} = \infty$
- Für alle Objekt-Pixel:
  - Ermittle die Distanz zur Kamera Z_{O}
  - Wenn $Z_{\text{B}} > Z_{\text{O}}$:
    - Zeichne das Pixel und setze $Z_{\text{B}} \leftarrow Z_{\text{O}}$.
  - Wenn $Z_{\text{B}} \leq Z_{\text{O}}$:
    - Zeichne das Pixel nicht

⇨ «Z-Fighting»: Berechnungsartefakt bei identischen Z-Werten.
⇨ Oftmals wird $Z_{\text{O}} = 1 - 1/z$ als Wert verwendet.



Spiegelungen & Schatten

Spiegelungen



Flächen

Berechne die Szene aus Sicht einer virtuellen Spiegelkamera und projiziere das Bild in Form einer Textur auf die Fläche.

⇨ Winkel und Distanz sind dabei äquivalent.

Kugeln

Berechne die Szene für alle Seiten einer umliegenden Bounding-Box und projiziere das Bild dann auf die Kugel.

⇨ Die Spiegelkamera steht dabei in der Kugelmitte.
⇨ Je grösser die Bounding-Box, desto kleiner der Fehler.

Environment Mapping

Beschreiben 360°-Bilder, welche für Spiegelungen und Hintergründe verwendet werden können.

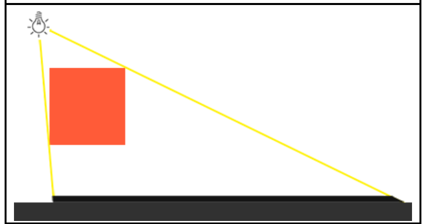
⇨ z.B. Cube-Maps, Sphere-Maps, Cylinder-Maps, etc.

Schatten

Shadow Mapping

Projiziere die Szene aus Sicht der Lichtquelle auf die zu belichtende Oberfläche.

⇨ Zeichne zuerst die Schatten und dann die Objekte.
⇨ Bilde dazu nicht die Farbwerte, sondern die Tiefenwerte ab.



Depth-Map

Visualisierung des Z-Buffers.

- Schwarz: $Z_O = 0$ (Nahe)
- Weiss: $Z_O = \infty$ (Weit weg)