

## 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 Skalarmultiplikation

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

Kreuzprodukt Transponieren

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{pmatrix} \quad \begin{pmatrix} a_1 \\ a_2 \\ \dots \end{pmatrix}^T = (a_1, a_2, \dots)$$

Euklidische Norm (Länge) Normalisierung

$$|\vec{a}| = \sqrt{a_1^2 + a_2^2 + \dots} \quad \hat{a} = \frac{1}{|\vec{a}|} \cdot \vec{a}$$

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.  
⇒  $|\vec{a} \times \vec{b}| \cdot \frac{1}{2}$  entspricht der Fläche des aufgespannten Dreiecks.

#### Multiplication

Allgemein nicht kommutativ:

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

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

#### Gleichungssysteme

Allgemeine Definition:

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

#### Gauss-Verfahren

$[A | b]: \begin{bmatrix} 1 & 4 & 2 \\ 2 & 9 & 5 \end{bmatrix} \Rightarrow \begin{bmatrix} 1 & 0 & -2 \\ 0 & 1 & 1 \end{bmatrix}$

#### Orthogonale Projektion

### 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 Ansichtweise 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 untransformiert.  
⇒ 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).

### Skalierung

$T(\vec{x}) = \begin{pmatrix} x + d_1 \\ y + d_2 \\ 1 \end{pmatrix} = \vec{x} + \vec{d}$

#### Rotation

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

$$S(\vec{x}) = \begin{pmatrix} s \cdot x \\ s \cdot y \\ 1 \end{pmatrix} = s \cdot \vec{x}$$

⇒ Die Faktoren  $s$  können auch unterschiedlich sein (s. Matrix).

#### Gesamt-Transformation

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

⇒ 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() {
    // Homogeneous Transformation
    gl_Position = vec4(positionIn, 1.0)
        * model
        * view
        * projection;
}

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

```

Vertex Shader

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

#### 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

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

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

#### Orthogonal

$c_x = x \quad c_y = y$

#### 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

vertex array

vertex shader

triangle assembly

rasterization

fragment shader

testing and blending

framebuffer

(1, 2, 3) (3, 2, 4) (2, 1, 3) (7, 2, 5) element array

uniform state

#### 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.

#### 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:

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

⇒ Die nicht enthaltenen Lichtanteile werden reduziert.

### Oberflächennormale

Nicht-triviale Belichtungsmodelle berücksichtigen die Ausrichtung der Oberfläche:

$$\mathbf{N}_{V_1} = (\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_1)$$

⇒ 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.



### Beleuchtungsmodelle

#### Ambient Lighting

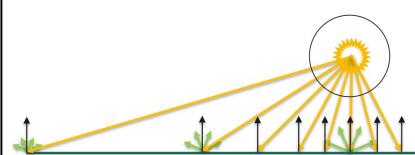
Belichtung von einem globalen Licht mit Remission in alle Richtungen.



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

#### Diffuse Lighting

Belichtung von einer Punktquelle mit Remission in alle Richtungen.



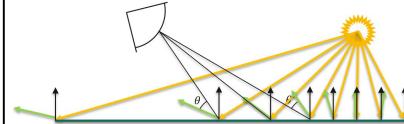
```
void main() {
    vec3 normDir = norm(normal);
    vec3 lightDir = norm(lightPos - fragPos);
    float cosTheta
        = max(dot(normDir, lightDir), 0.0);
    vec3 diffuse = cosTheta
        * lightColor
```

```
* objectColor;
fragColor = vec4(diffuse, 1.0);
```

⇒ 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);
}
```

⇒ Wird für spiegelnde Oberflächen verwendet.

### Kombinationsmodelle

#### Phong-Shading

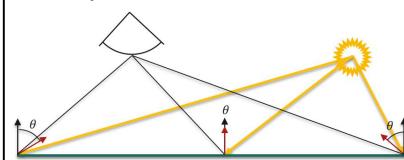
Die Belichtung wird aus Ambient-, Diffuse- und Specular-Anteilen zusammengesetzt.

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

⇒ 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);
    ...
}
```

### 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.

$p = (x, y, z)$        $p = (u, v)$       Texture

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

```
void main(void) {
    fragColor = texture(texUnit, texCoord);
}
```

### Komplexe Oberflächen

#### Grundformen

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

- Funktionen:** Kontinuierlicher Wertebereich
  - Explizit:  $z = -ax + by + \dots$
  - Implizit:  $0 = x^2 + 2y^2 + \dots$
  - Parametrisch:  $P = \vec{o} + s\vec{u} + \dots$
- 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:** Punktfolke, Meshes
- Approximierend:** Iso-Surface, Splines
- Konstruiert:** Subdivision Surfaces, Fraktale

### Vor- und Nachteile

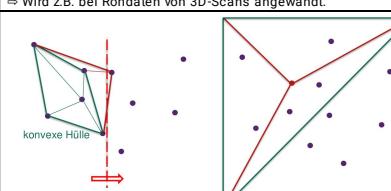
Es gibt keine beste Repräsentationsform für Objekte. Vor- und Nachteile sind:

- Funktionen:**
  - Vorteile:** Wenig Speicherplatz, Schnittpunkte mathematisch berechenbar, beliebig genau 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 Punktfolge in ein Polygon-Mesh.

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



### Sweep-Strategie

Laufe von links nach rechts.

2. Für jeden Punkt:

- Zeichne eine Linie zu den 2 vorherigen Punkten, für die gilt:
  - Keine Dellen entstehen
  - Keine Überschneidungen entstehen
- Verbinde nun alle weiteren Punkte innerhalb dieser Form.

3. Wiederhole, bis zum Ende.

⇒ Die entstehende Form nennt sich «Konvexe Hülle».

### Insert-Strategie

- Zeichne 2 Anfangsdreiecke um alle Punkte.
- Für alle Punkte (zufällige Wahl):

- Bestimme das umfassende Dreieck.
- Unterteile dieses Dreieck in 3 weitere Dreiecke. D.h. Verbinde alle Eckpunkte mit dem gewählten Punkt.

3. Wiederhole, bis zum Ende.

- Entferne nun alle künstlichen Anfangspunkte 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

- Rekursiv für alle Dreiecke:

- Wähle ein anliegendes Dreieck
- 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.

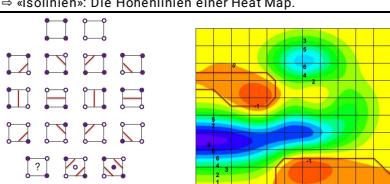
- Gitter über die Daten legen.
- Betrachtungshöhe (Potenzial) festlegen.
- Für alle Quadrate im Gitter:

- Eckpunkte beachten.
- Nach Schema unten Linien einzeichnen.

4. Wiederhole, bis zum Ende.

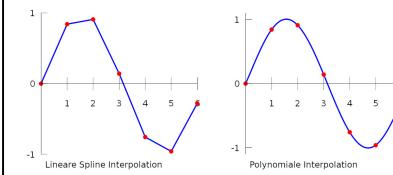
⇒ Heat Map: 2D-Visualisierung von 3D-Landschaften.

⇒ Isolinien: Die Höhenlinien einer Heat Map.



### Weitere Algorithmen

- Marching Cubes (3D-Heat-Maps)
- Interpolation: Punkte «vervollständigen»
  - Polynomial:  $f = a_0x^0 + \dots + a_nx^n$
  - Splines: Stückweise Interpolation der Punkte mit linearen, quadratischen oder kubischen Funktionen.
- NURBS: Approximation von 3D-Flächen



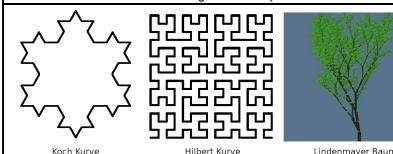
### Lindenmayer Systeme

L-Systeme beschreiben beliebig feine, selbstähnliche geometrische Strukturen.  
⇒ Sie können rekursiv definiert und aufgebaut werden.

### Formale Definition

- Anfangsform (z.B. Strich):  $f$
- Ersetzungsregeln:  $f \rightarrow f + f - f - f + f$ 
  - 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).



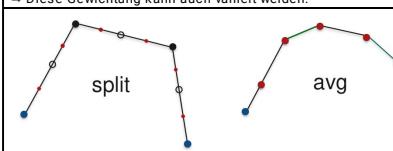
### Subdivision Surfaces

Beschreibt ein rekursives Verfahren für das Verfeinern von Oberflächen.  
⇒ Subdivision Curves ist das Äquivalent für Kurven.

### Curves: Chaikin's Algorithmus

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

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

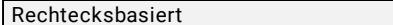


### Surfaces: Algorithmen

#### Dreiecksbasiert



#### Rechtecksbasiert





## Vorteile

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

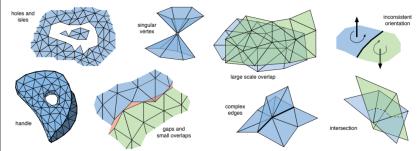
- Beliebige Oberflächentopologie
- Kompakte Repräsentation
- Level-of-Detail Rendering
- Intuitiv mit einfachen Algorithmen

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

## Korrektur & Optimierung

3D-Modelle können aus verschiedenen Quellen stammen. Oftmals sind dabei folgende Qualitätsprobleme zu beobachten:

- Künstliche Unebenheiten der Flächen
- Zu grosse Datenmengen
- Schlechte Triangulierung wie z.B. topologische Inkonsistenz und spitze / ungleichmäßige Dreiecke



## Visualisierung von Fehlern

Um die genannten Qualitätsprobleme zu visualisieren, können wir z.B.:

- Die Reflexionslinien des Modells betrachten.
- Ungleiche Dreiecke basierend auf dem Kantenverhältnis einfärben.

## Modell-Reparatur

Für die Verbesserung der Modellqualität gibt es verschiedene Flächen- und Volumen-orientierte Algorithmen:

- **Flächen-orientierte Algorithmen:**
  - Vorteil: Die assoziierten Oberflächeneigenschaften bleibt gut erhalten.
  - Nachteil: Schlechte automatische Fehlererkennung und Behebung.
- **Volumen-orientierte Algorithmen:**
  - Vorteil: Gute automatische Fehlererkennung und Behebung.
  - Nachteil: Oft zu detailliert und mit Verlust der Oberflächen-eigenschaften.

⇒ z.B.: Mesh Smoothing, Mesh Reduktion, Remeshing, etc.  
⇒ Mesh Smoothing ist für dieses Modul nicht relevant.

## Mesh Reduktion / Remeshing

Mit diesen Verfahren wollen wir die Anzahl Vertices und Faces 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 ersetze 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.  
⇒ Allgemein gilt: Laufzeit  $O(n)$  mit möglichen Alias Errors.

⇒ Am besten geeignet ist also der «Least Squares»-Algorithmus.  
⇒ D.h.: Minimiere den quadrierten Abstand zur Oberfläche.

## Inkrementelle Reduktion

Sequenzielle Anwendung von zwei primitiven Reduktionsoperationen:

- Eckpunkt entfernen
- Kante kollabieren (zusammenlegen)

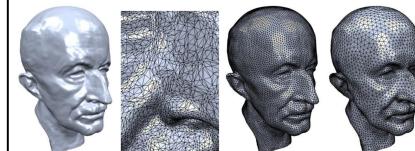
Die Reihenfolge der Operationen wird durch eine globale Metrik (den Approximationsfehler) bestimmt.

⇒ Diese Metrik muss bei jeder Reduktion neu berechnet werden.  
⇒ Allgemein gilt: Laufzeit  $O(n \log n)$  bis  $O(n^2)$  ohne Alias Errors.

## Resampling / Remeshing

Komplette Neuberechnung des Meshes, wobei die Struktur auf verschiedene Arten verändert werden kann:

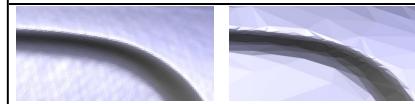
- Wechsel von Dreiecke auf Vierecke
  - Verwendung von gleichmässigen Dreiecken
  - Variieren der Flächendichte
  - Erhöhung der Regularität der Struktur
- ⇒ Die Eckpunkte des neuen Meshes müssen dabei nicht zwingend mit denen des alten Meshes assoziert sein.  
⇒ Alias Errors sind jedoch weiterhin möglich.



## Alias Error

Beschreibt die inkorrekte Approximation eines Meshes aufgrund mangelnder Abtastgenauigkeit.

⇒ z.B.: Zu grosse Toleranz  $\epsilon$  beim «Vertex Clustering».

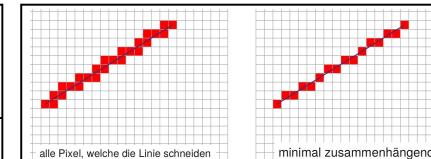


## Rasterisierung & Sichtbarkeit

### 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 zeitweise gezeichnet.  
⇒ Dazu wird u.a. der Bresenham Linien-Algorithmus verwendet.

## Bresenham Linien-Algorithmus

Basierend auf zwei Punkten  $P_{\text{Start}}$  und  $P_{\text{Ende}}$ , 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

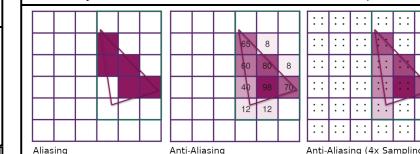
Zeichne alle Pixel eines Dreiecks unter Beachtung der prozentualen Abdeckung. Das bedeutet:

- 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 Objekträger erhalten also eine «weiche» Transparenz.



## Probleme (Aliasing Effekte)

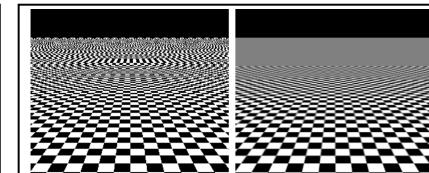
Wenn die Auflösung eines Texturmusters grösser ist als die Auflösung der Anzeigefläche, kann der Moiré-Effekt auftreten.

⇒ 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.

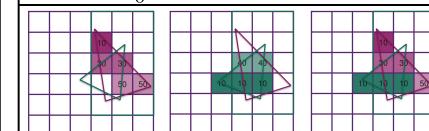


## Sichtbarkeit

## Z-Buffer (Depth-Buffer)

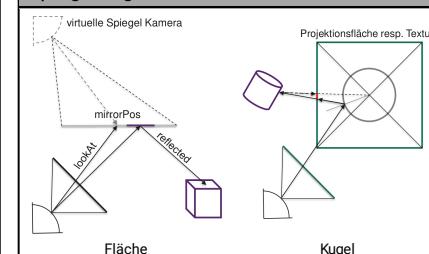
Erlaubt das korrekte Zeichnen von überlappenden Objekten.

- Initialisiere den Buffer mit  $Z_B = \infty$
  - Für alle Objekt-Pixel:
    - Ermittle die Distanz zur Kamera  $Z_O$
    - Wenn  $Z_B > Z_O$ :
      - Zeichne das Pixel und setze  $Z_B \leftarrow Z_O$ .
    - Wenn  $Z_B \leq Z_O$ :
      - Zeichne das Pixel nicht
- ⇒ «Z-Fighting»: Berechnungsfehler bei identischen Z-Werten.  
⇒ Oftmals wird  $Z_O = 1 - 1/z$  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.

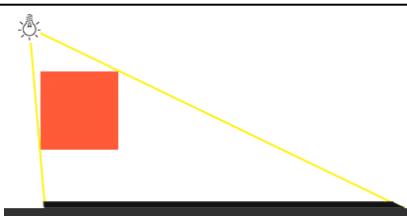
### Environment Mapping

Beschreiben 360°-Bilder, welche für Spiegelungen und Hintergründe verwendet werden können.  
⇒ z.B. Cube-Maps, Sphere-Maps, Cylinder-Maps, etc.

### Schatten

### Shadow Mapping

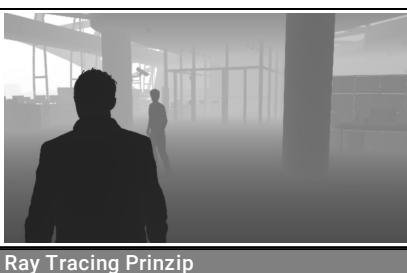
Projiziere die Szene aus Sicht der Lichtquelle auf die zu belichtende Oberfläche.  
⇒ Zeichnet zuerst die Schatten und dann die Objekte



## Depth-Map

Visualisierung des Z-Buffers.

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



## Ray Tracing Prinzip

### Bildgenerierung

### Verdeckung

### Rekursive Weiterverfolgen

Farbe des Elements (am Schnittpunkt) hängt ab von:

- Farbe & Intensität sichtbarer Lichtquelle in Umgebung
- Richtung Lichtquelle
- Oberflächenfarbe & -beschaffenheit
- Position Beobachter

Zur Berechnung der Objektfarbe, Strahlenbrechung rekursiv weiterführen (Sekundärstrahlen)

- Strahlen können beliebig oft reflektiert werden ( irgendwann muss abgebrochen werden)

### Rückweg aus Rekursion

- Beim Rückweg kann Farbe & Licht-Intensität bestimmt werden
- Baumstruktur: Strahl erzeugt beim auftreffen mehrere Folgestrahlen

### Ray Tracing

Von der Kamera aus werden Strahlen durch alle Pixel der Bildebene verfolgt und mit der Szene geschnitten. Bei einem primitiven, einstufigen Ray Tracing bestimmt die Objektfarbe am Schnittpunkt die Pixelfarbe. In der Regel wird beim Aufschlagpunkt einer oder mehrerer Reflektionsstrahlen berechnet und diese rekursiv weiterverfolgt. Entsprechend wird die Pixelfarbe auf dem Weg zurück aus der Rekursion zusammen gerechnet, in der Regel auch unter Einbezug der Strahlen zu den Lichtquellen.

- **Triangle-Meshes:** Operation relativ rechenintensiv
- **Elemente mit impliziter Darstellung:** oft sehr effizient berechnet
  - z.B. für Kugel  

$$(x - c_x)^2 + (y - c_y)^2 - r^2 = 0$$
 kann für  $(x, y)$  der parametrische Strahl  $0 + t\vec{v}$  eingesetzt und nach  $t$  aufgelöst werden ⇒ liefert Schnittpunkt

#### Minimal Ray Tracer (vereinfacht)

#### Generieren von Primärstrahlen

Kamera im Ursprung & Projektfläche mit Höhe 1 im Abstand z

- $d = \frac{1}{h}$
- $p_x(0) = (w - 1) \frac{d}{2}$
- $p_y(0) = (h - 1) \frac{d}{2} = \frac{1}{2} - \frac{1}{2h}$
- Inkrement von  $p_x, p_y : d$
- Strahlen durch Ursprung und  $(p_x, p_y, z)$ :  
 $0 + t \cdot n$  or *malize* $(p_x, p_y, z)$

Am einfachsten: \* Rudimentären, nicht Performance optimierten Ray Tracer \* Kugeln und Ebenen haben einfache implizite Gleichungen

⇒ TODO: Keine Ahnung wie viel man davon noch aufschreiben soll, sind viele Berechnungen die wahrscheinlich nicht kommen

#### Acceleration Structures

#### Bounding Volume Hierarchies

#### Acceleration Structures

#### Axen-alierte Bounding Box

#### Bounding Volumes

#### Bounding Volumes Hierarchy (BVH)

#### Raumteilende Strukturen

#### Voxel: Zellen im Raum

#### Uniform Grid

- Raum in gleich grosse Zellen Aufteilen (Voxels)
- Jede Zelle hat Referenz auf alle Primitiven, welche sich überlappen
- Strahlschnitt
  - Mit Bresenham (Algorithmus) Voxels in Strahlenrichtung traversieren
  - Mit allen den individuellen Voxels zugeordneten Elementen
  - Abbruch bei ersten geschnittenen Primitiven
- Auflösung Grid
  - zu tief: Keine Reduktion der Anzahl Schnitt-Tests
  - zu hoch: Unnötig hoher Traversierungsaufwand

#### Uniform Grid Heuristik

- Anzahl Voxel = ca. Anzahl Primitiven

Funktioniert gut bei uniformer Verteilung der Primitiven

#### Quad-Tree

#### TODO

#### Advanced Ray Tracing

- Ray-tracing wurde lange als Technik verstanden um fotorealistische Bilder zu berechnen. Das ist aber nicht ganz korrekt
- Fotorealistische, computergenerierte Bilder sind eine Kombination aus "light transport algorithm" und einer Technik, um die Sichtbarkeit zwischen Oberflächen zu berechnen (rasterization, ray-tracing)

#### Whitted Ray Tracing (1980)

Die Farbe eines Punktes auf einer Oberfläche besteht aus 3 Komponenten:

- Oberflächenfarbe am Schnittpunkt unter Berücksichtigung des Licates aller direkt sichtbaren Lichtquellen (Phong-Shading)
- Farbe des Lichtes aus der reflektierten Richtung
- Farbe des Lichtes aus Richtung der Lichtbrechung

#### Cook, 1984

#### Kamera-Linsen-Effekt

- Objekte nicht in Fokusebene, erscheinen unscharf
- Jeder Punkt wird als runde Scheibe abgebildet
- Verfolge mehrere Strahlen durch jeweils einen zufälligen Punkt auf der Linse

#### Bewegungsunschärfe

Emmitierte Strahlen zu verschiedenen Zeitpunkten und berechne das Durchschnittsbild

#### Globale Beleuchtung

- Alle (nicht komplett schwarzen) Oberflächen (r)emittieren Licht
- Für Belichtung eines Punktes haben alle sichtbaren Oberflächen einen Einfluss
- Light Transport Algorithmus

#### Light Transport

- Approximation mittels Monte-Carlo Simulation durch Auswahl zufällig gewählten Richtungsvektoren

#### Monte Carlo Simulation

- Zufall verwenden um komplexe Berechnungen zu vereinfachen
- Für einzelne Punkte kann Lösung einfach berechnet werden
- Einzelne Lösungen können aggregiert werden, dass die exakte Lösung approximiert wird.

#### Realtime ray tracing

#### TODO

#### Animation

Eine Folge zeitlich schnell hintereinander gereihten Bildern erscheint als Bewegung.

#### Explizite Berechnung

- In Computeranimation ist Modell mathematisch repräsentiert

- Die einfachste Möglichkeit ist Eigenschaften wie Position, Rotation oder Farbe explizit aufgrund der "Simulationszeit" zu berechnen

#### Key Frames und "tweening"

- Animation wird anhand von Schlüsselbildern aufgebaut
- Bilder dazwischen dienen dazu Bewegung flüssig und natürlich erscheinen zu lassen
- Ansatz für handgezeichnete wie auch computergenerierte Animationen

#### Grundlegende Techniken der Computeranimation

- Vom Künstler orchestriert (key frames)
- Data driven (motion capture)
- Procedural (simulation, calculation using physics formulas)

⇒ Kommen auch kombiniert zum Einsatz

#### Animation mittels Key Frames

Kommt bei Handlung zum Zuge, oder wenn sich ein Einfluss von "aussern" ändert

#### Library für Interpolation: Tween.js

#### Mathematisch beschriebene Modelle:

**Szene:** Beschrieben durch Modellparameter  
**Tweening:** Erreicht durch Interpolation der Parameter

Dafür bieten sich z.B. Splines an

- Gehen per Definition durch die Punkte
- Bilden einen kontinuierlichen Übergang dazwischen ab