

## Einführung

### Anwendungen

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

- Videospiele
- Cartoons & Filme
- Datenvisualisierungen
- Berechnungen

### Programme & APIs

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

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

Fläche (Parallelogram) Fläche (Dreieck)

$$|\vec{a} \times \vec{b}| \quad \left| \vec{a} \times \vec{b} \right| \cdot \frac{1}{2}$$

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

#### Multiplikation

Allgemein nicht kommutativ:

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

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

#### Gleichungssysteme

Allgemeine Definition:

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

#### Gauss-Verfahren

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

## Orthogonale Projektion

Definition:  $\vec{v}$  projiziert auf  $\vec{u}$  ergibt  $\vec{u}_p$ .

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

### 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 ( $l = 3 \cdot n_F$ )
  - 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 die Punkte jeweils 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  $\theta$ .

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

⇒ 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:

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

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

### Berechnungen

Gegeben ist eine Kamera an Position  $E$ . Transformiere den Punkt  $P$  vom Weltsystem ins Kamerasytem.

$$P' = P - E$$

⇒ Der Ursprung  $U$  wird an die Kameraposition  $E$  verschoben.

### Homogene Koordinaten

Um das Problem der Translation zu lösen, werden alle kartesischen Koordinaten  $P(x, y)$  auf homogene Koordinaten  $[x \ y \ w \ 1]$  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.

### 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)  
(1, 2, 4)  
(1, 2, 5)  
(2, 3, 7)      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 Berechnungsergebnisse 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 vorheriger Stage
- out: An nächste Stage
- uniform: Vom Host an alle Primitiven

```

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

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

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

    normal = vec4(normalIn, 1.0) * model;
}

Vertex Shader
  
```

⇒ in und out verwenden dabei «matching by name».  
⇒ vec \* vec ist in GLSL eine komponentenweise Multiplikation.

### 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:

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

⇒ 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 vorraus 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);
}
```

```
= 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.

```
void main() {
    ...
    vec3 halfwayDir = norm(lightDir + camDir);
    float cosTheta
        = max(dot(normDir, halfwayDir), 0.0);
    ...
}
```

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

#### Funktionen:

- Vorteile:** Wenig Speicherplatz, Schnellpunkte mathematisch berechenbar, beliebig genaue Auflösung
- Nachteile:** Beschränkte Formen, komplexe Herleitung, grafische Transformationen sind schwierig

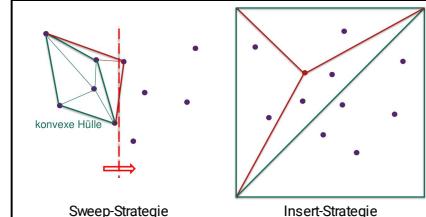
#### Punkte:

- 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 Punktfolge in ein Polygon-Mesh.

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



### 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 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

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

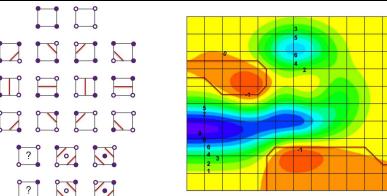
### Approximationen

#### Marching Squares Algorithmus

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

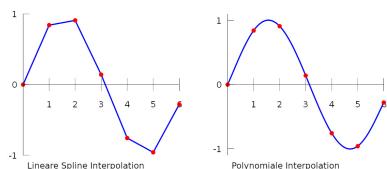
1. Gitter über die Daten legen.
2. Betrachtungshöhe (Potenzial) festlegen.
3. Für alle Quadrate im Gitter:
  - a. Eckpunkte beachten.
  - b. Nach Schema unten Linien einzeichnen.

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



### Weitere Algorithmen

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



### Lindenmayer Systeme

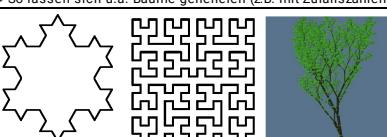
L-Systeme beschreiben beliebig feine, selbstähnliche geometrische Strukturen.

⇒ Sie können rekursiv definiert und aufgebaut werden.

### Formale Definition

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



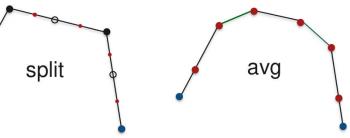
### Subdivision Surfaces

Beschreibt ein rekursives Verfahren für das Verfeinern von Oberflächen.

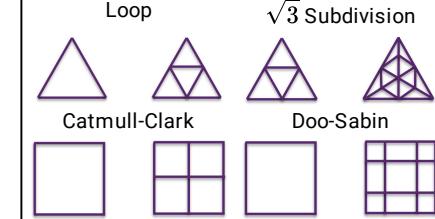
⇒ Subdivision Surface ist das Äquivalent für Komplexe

## Curves: Chaikin's Algorithmus

1. Beginne mit einer Kurve
  2. Markiere die Anfangspunkte (Blau)
  3. Setze nun 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



## Vorteile

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

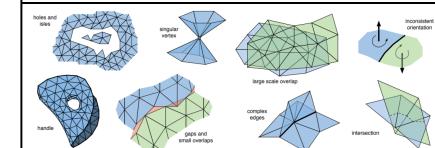
- Beliebige Oberflächentopologie
- Kompakte Repräsentation
- Level-of-Detail Rendering
- Intuitiv mit einfachen Algorithmen

⇒ 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 oder spitze / ungleichmässige Dreiecke



## Visualisierung von Fehlern

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

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

## Modell-Reparatur

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

## Flächen-orientierte Algorithmen:

- **Vorteil:** Die assoziierten Oberflächeneigenschaften bleiben 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 Oberflächeneigenschaften.
- ⇒ z.B.: Mesh Smoothing, Mesh Reduktion, Remeshing, etc.  
⇒ Mesh Smoothing ist für dieses Modul nicht relevant.

Original Approximation

Beschreibt eine inkorrekte Approximation des Meshes aufgrund mangelnder Abtastgenauigkeit.

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

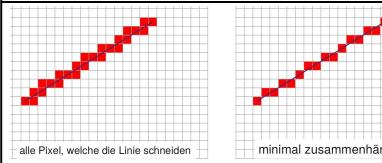


## Rasterisierung & Sichtbarkeit

### 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.
    - Bei Eckpunkten muss das Zentrum auf der oberen und linken Seite liegen.
    - Zwei linke Seiten sind auch gültig.
- ⇒ Achtung: Die obere Seite muss dazu exakt horizontal sein.  
⇒ Technisch wird das Dreieck zeilenweise 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.

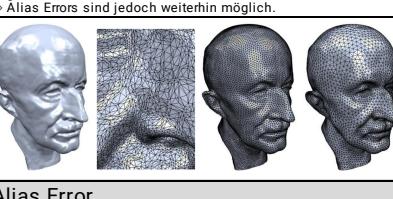
## Resampling / Remeshing

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 assoziiert sein.

⇒ Alias Errors sind jedoch weiterhin möglich.



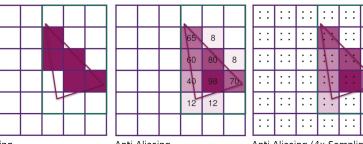
Varianten davon sind:

•

**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.



## Flächen

Berechne die Szene aus Sicht einer virtuellen Spiegelkamera und projizierte 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 projizierte das Bild dann auf die Kugel.  
⇒ Die Spiegelkamera steht dabei in der Kugelmitte.  
⇒ Je grösser die Bounding-Box, desto kleiner der Fehler.

## Berechnungen

Basierend auf einem Spiegel an Position  $M$  mit der Normalen  $\vec{n}$  und einer Kamera an Position  $E$ . Berechne die Position der Spiegelkamera  $E'$ .

$$\vec{v} = E - M \quad \vec{d} = \left( \frac{\vec{n} \circ \vec{v}}{|\vec{n}|^2} \right) \cdot \vec{n}$$

$$E' = E - 2 \cdot \vec{d}$$

⇒ Es handelt sich dabei um eine orthogonale Projektion, welche uns die Distanz  $d$  von der Kamera zum Spiegel liefert.

## Environment Mapping

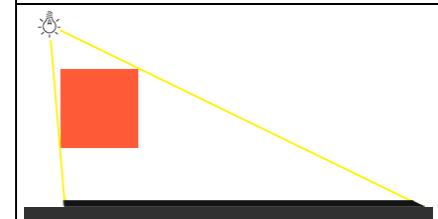
Beschreiben 360°-Bilder, welche für Spiegelungen und Hintergründe verwendet werden können.  
⇒ z.B. Cube-Maps, Sphere-Maps, Cylinder-Maps, etc.

## Schatten

### Shadow Mapping

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

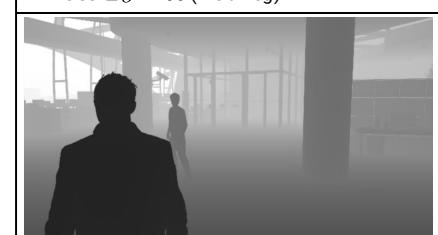
⇒ Zeichne zuerst die Schatten und dann die Objekte.  
⇒ Bilde dazu nicht die Farbwerte, sondern die Tiefenwerte ab.



## Depth-Map

Visualisierung des Z-Buffers.

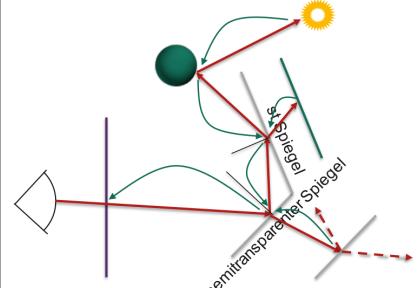
- Schwarz:  $Z_O = 0$  (Nahe)
- Weiss:  $Z_O = \infty$  (Weit weg)



## Ray-Tracing

### Grundprinzip

Bei Ray-Tracing werden Strahlen ausgehend von der Kamera durch alle Pixel der Anzeigefläche geschickt und rekursiv weiterverfolgt.



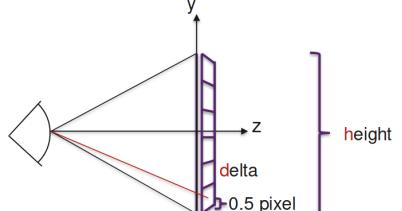
Bei der Rückkehr aus der Rekursion werden die verschiedenen Einflussfaktoren (Licht, Farbe, etc.) zusammengerechnet.

⇒ Problem: Die Rekursion könnte unendlich lange weiterlaufen.  
⇒ Aus diesem Grund wird nach einigen Schritten abgebrochen.

#### Generierung der Primärstrahlen

Die Primärstrahlen werden jeweils von der Pixelmitte aus generiert. Die Auflösung bestimmt die Anzahl der Strahlen.

⇒ Dies wirkt sich dementsprechend auf die Performance aus.



#### «Whitted Ray Tracing»-Algorithmus

Je nach Oberflächenbeschaffenheit werden rekursiv diese Schritte durchlaufen:

1. Spiegelnd: Reflexionsstrahl weiterverfolgen
2. Durchsichtig: Lichtstrahl weiterverfolgen
3. Diffuse: Strahlen zu allen Lichtquellen
  - Schatten: Kein Betrag
  - Belichtet: Betrag vom remittierten Licht berechnen (Phong-Shading)

#### Advanced Ray-Tracing

Ray-Tracing kann mit zahlreichen Varianten weiter verfeinert werden:

- **Kamera-Linsen-Effekt:** Erzeugung eines Fokus effekts durch die Simulation einer Kamera linse bei der Strahlenberechnung.
- **Bewegungsunschärfe:** Erzeugung eines Bewegungseffekts durch Berechnung der Strahlen zu unterschiedlichen Zeitpunkten.
- **Globale Belichtung:** Realitätsnahe Belichtung durch rekursive Weiterverfolgung von mehreren, zufälligen Strahlen bei jedem Schnittpunkt.

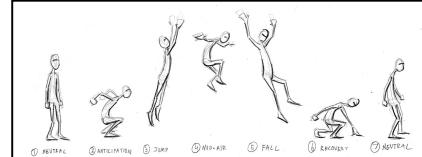
⇒ Durch die zufällige Strahlenberechnung entsteht Rauschen.  
⇒ Dieses Rauschen wird mit Denoising (AI) reduziert.



#### Acceleration Structures

Beschreiben Datenstrukturen, welche die Anzahl der Schnitttests reduzieren sollen.

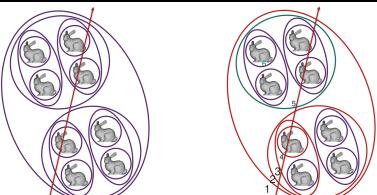
- **Problem:** Objekte in der Szene haben sehr komplexe Oberflächen.
- **Lösung:** Umschliesse alle Objekte in einfache Bounding Volumes.



#### Elementteilend

Verwendung einer binären Bounding Volumes Hierarchy (BHV), welche alle Elemente einer Szene umschließen.

⇒ Die Traversierung erfolgt rekursiv von vorne nach hinten.  
⇒ Abbruch, sobald ein effektiver Schnittpunkt gefunden wurde.



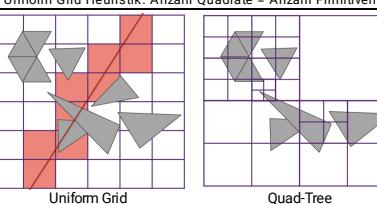
⇒ Für eine optimale Aufteilung ist leider kein Algorithmus mit subexponentieller Laufzeit bekannt.

#### Raumteilend

Bei diesem Verfahren werden nicht die Objekte selbst, sondern der Raum um die Szene unterteilt.

- **Uniform Grid:** Gleichmäßige Aufteilung des Raums in Quadrate (Voxels)
- **Quad-Tree:** Rekursive Aufteilung in Quadrate, wobei jedes Quadrat mit mehr als einem Eckpunkt weiter unterteilt wird.

⇒ Uniform Grid Heuristik: Anzahl Quadrate = Anzahl Primitiven.



#### Animationen

##### Techniken

Es gibt verschiedene Techniken für das Erstellen von Computeranimationen:

- Vom Künstler erstellt (Key Frames)
- Datengetrieben (Motion Capture)
- Prozedural (Simulation & Calculation)

##### Key Frames & Tweening

Beschreibt ein Verfahren, bei dem wichtige Animationspunkte (Key Frames) manuell erstellt und die Übergänge dazwischen interpoliert werden.

⇒ Die Interpolation erfolgt z.B. über Splines.