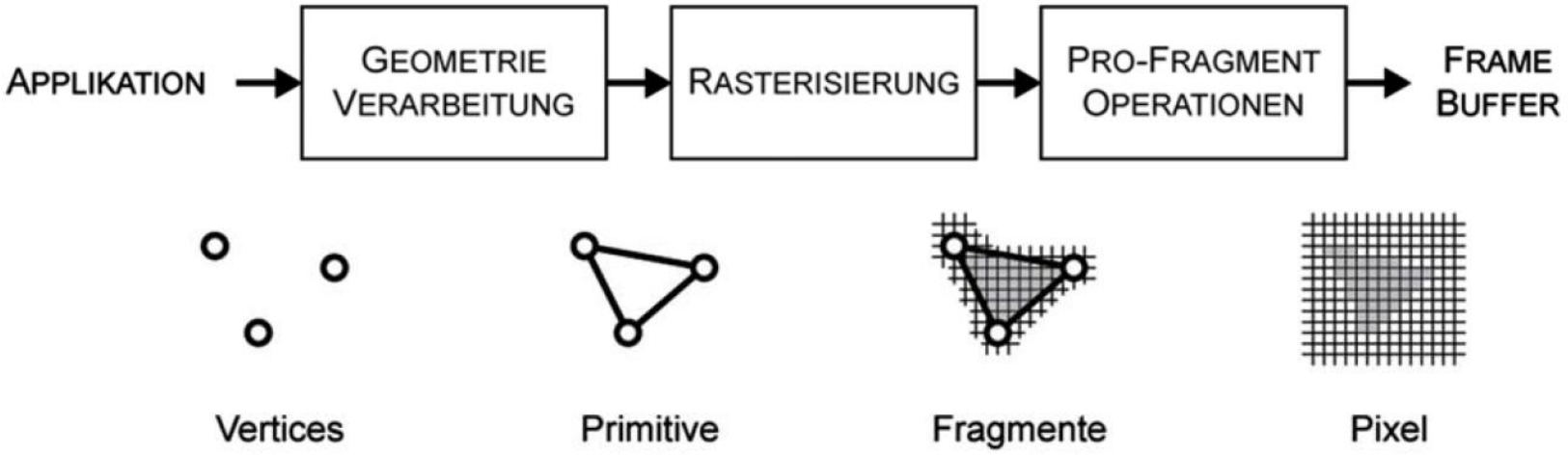
Computergrafik

Übung Shaders

Prof. Dr.-Ing. Carsten Dachsbacher
Lehrstuhl für Computergrafik
Karlsruher Institut für Technologie

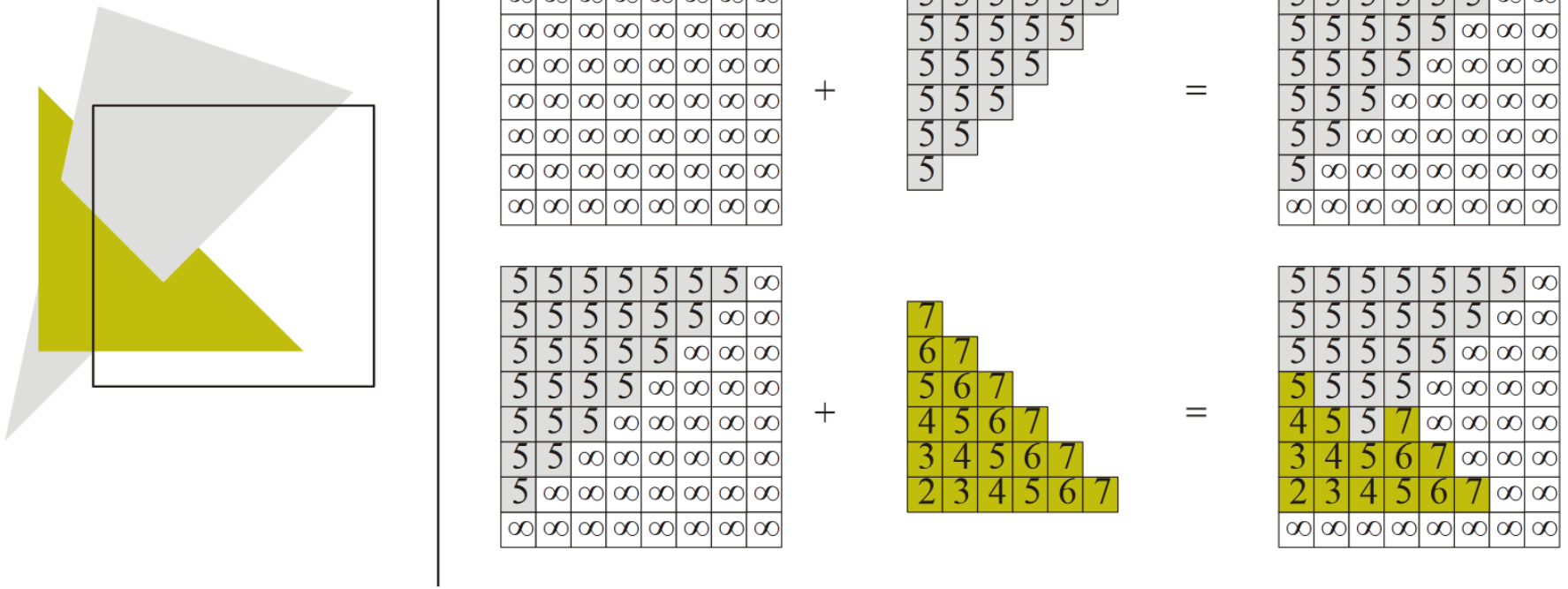
Rasterisierungspipeline

- Abbilden der Geometrie (Dreiecke) auf 2D Bildschirmkoordinaten
- Rasterisierung (siehe 1. Übung)
- Interpolation von Vertex-Attributen
- Tiefentest und andere Fragmentoperationen

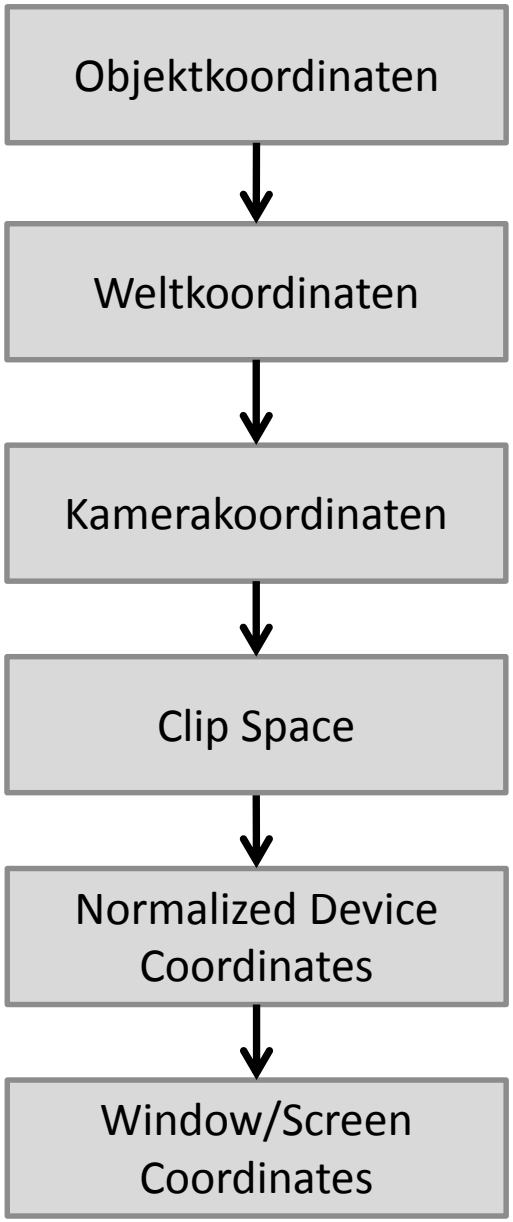


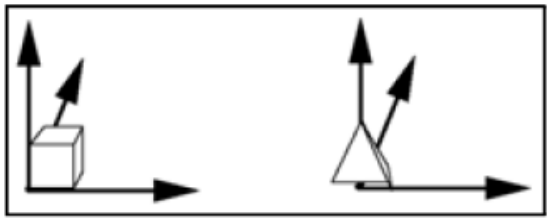
Z-Buffer

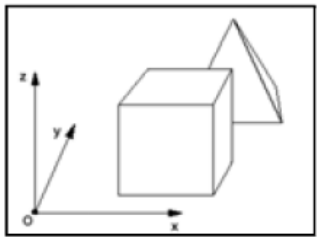
- Entscheide Sichtbarkeit durch Speichern der Distanz des nahesten Objekts pro Pixel

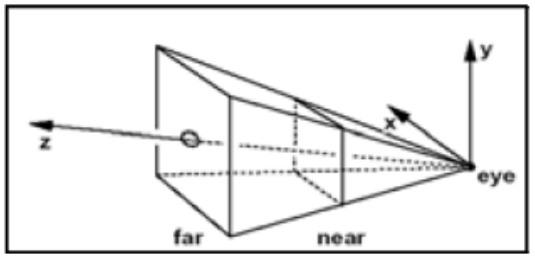


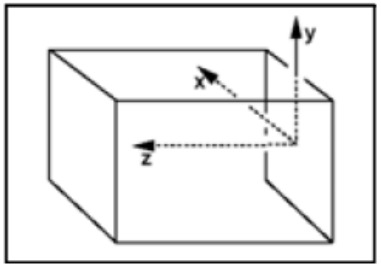
Koordinatensystem Pipeline





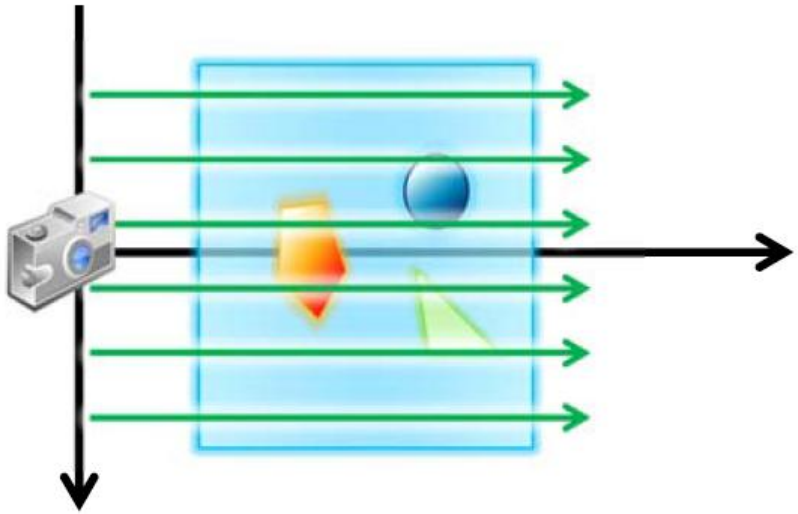




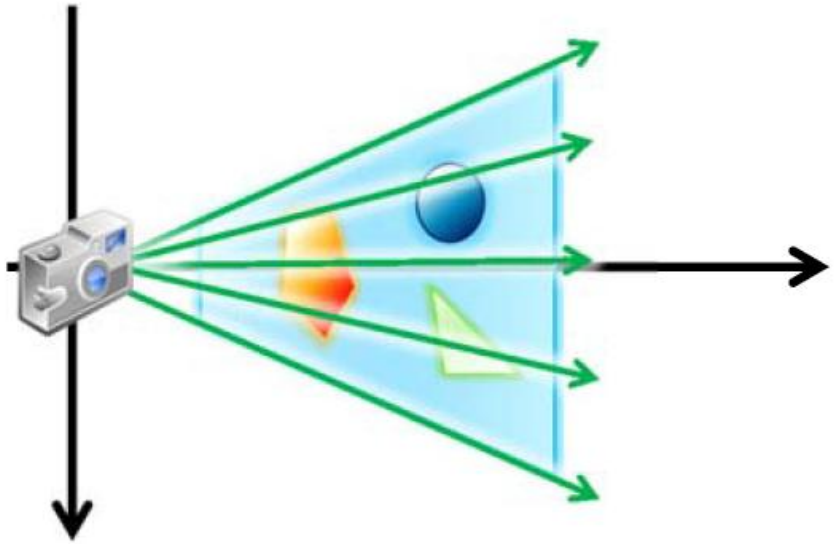


Projektionen

- Abbilden der Geometrie (Dreiecke) auf 2D Bildschirmkoordinaten
- Mehrere Möglichkeiten
  - Orthographisch
  - Perspektivisch



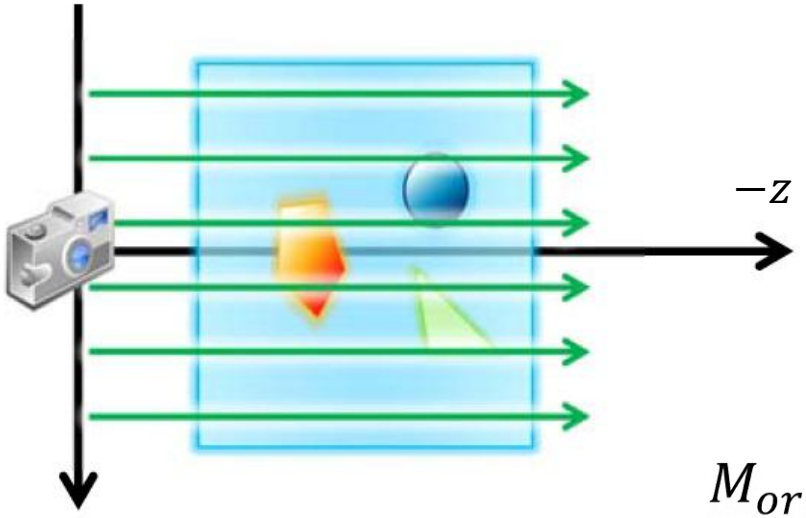
Orthographische Kamera:
Sichtvolumen ist ein Quader



Perspektivische Kamera:
Sichtvolumen ist ein Pyramidenstumpf

Projektionen

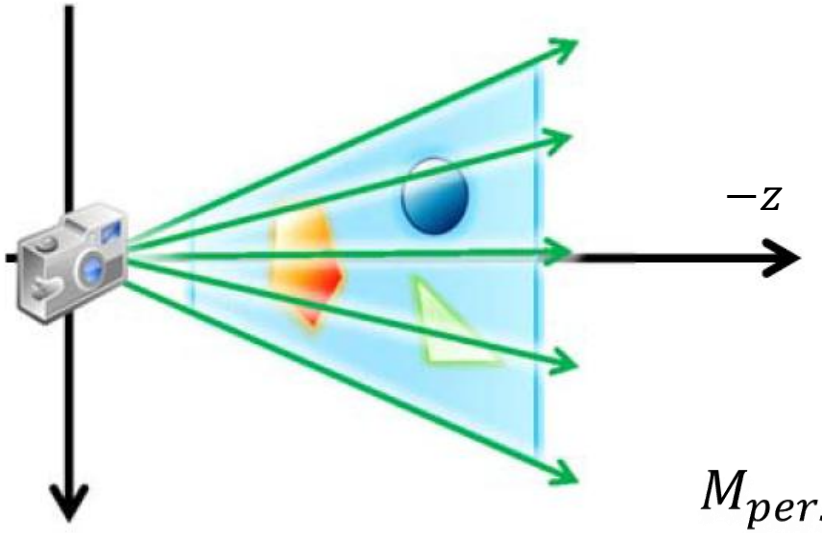
- Abbilden der Geometrie (Dreiecke) auf 2D Bildschirmkoordinaten
- **Zwischenschritt:** Abbildung des Sichtvolumens auf den Einheitswürfel
- Orthographisch: Einfach!
- Sichtvolumen: $[r, l] \times [t, b] \times [n, f] \rightarrow [-1,1]^3$



$$M_{orth} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & -\frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Projektionen

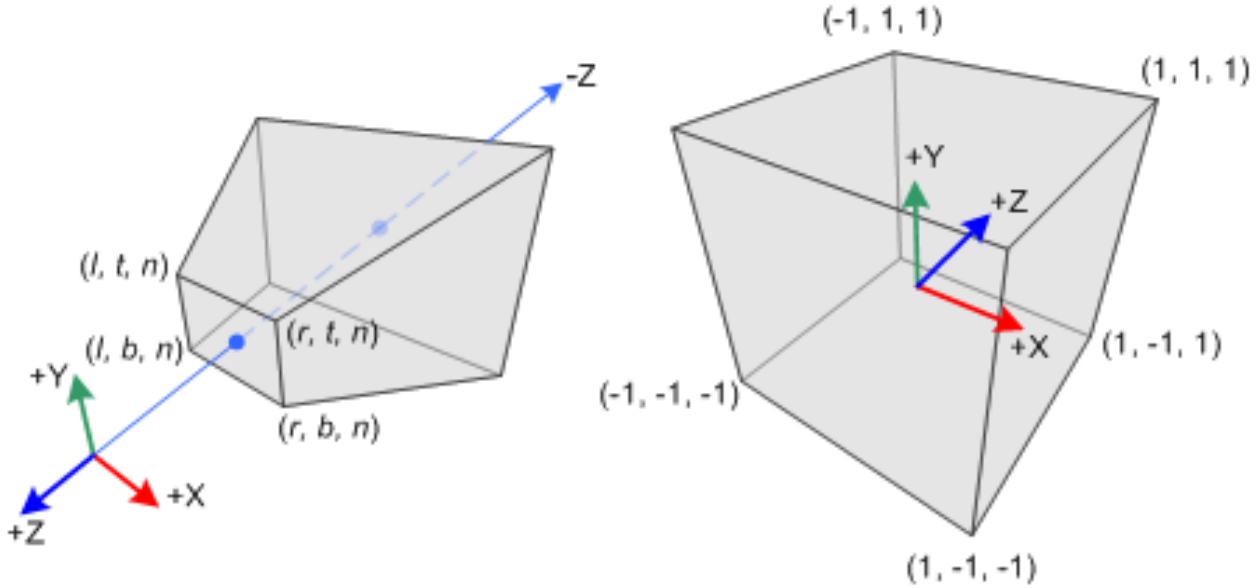
- Abbilden der Geometrie (Dreiecke) auf 2D Bildschirmkoordinaten
- **Zwischenschritt:** Abbildung des Sichtvolumens auf den Einheitswürfel
- Perspektivisch: Projektionszentrum im Ursprung
- Sichtvolumen definiert durch: $\mathrm{r}, l, b, t, n, f$



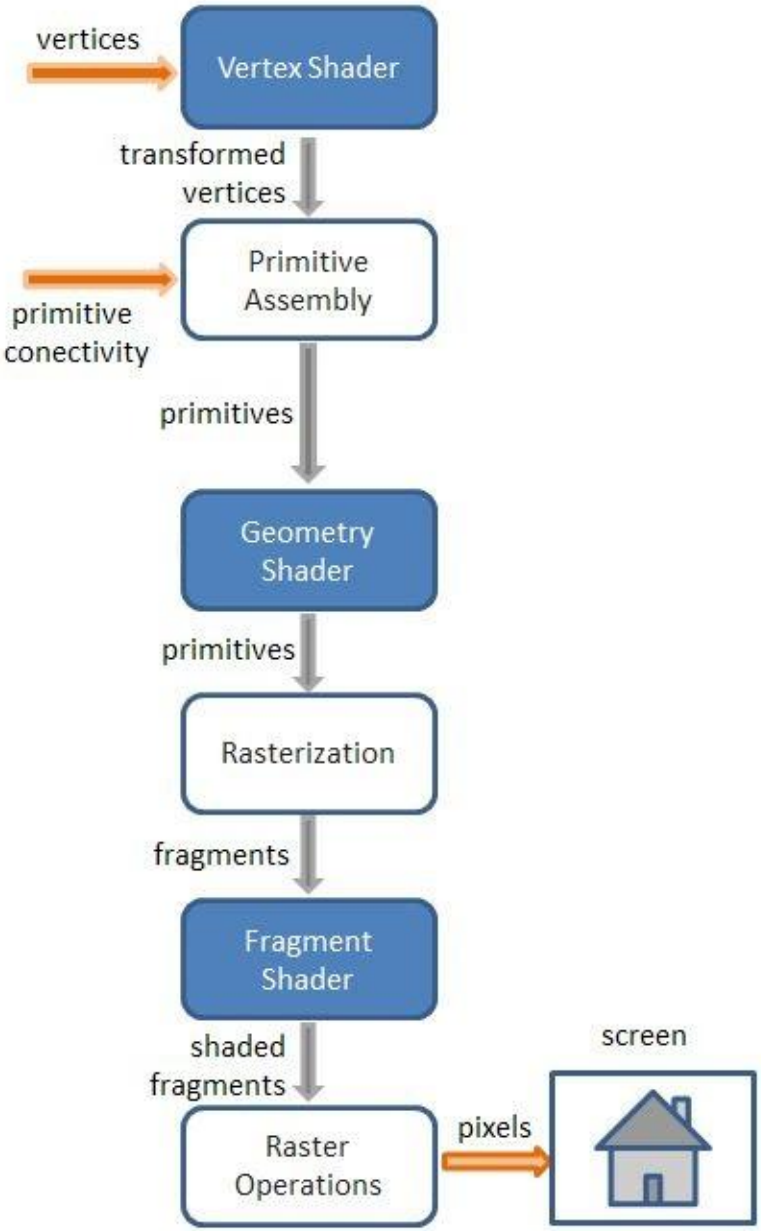
$$M_{pers} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2nf}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

Projektionen

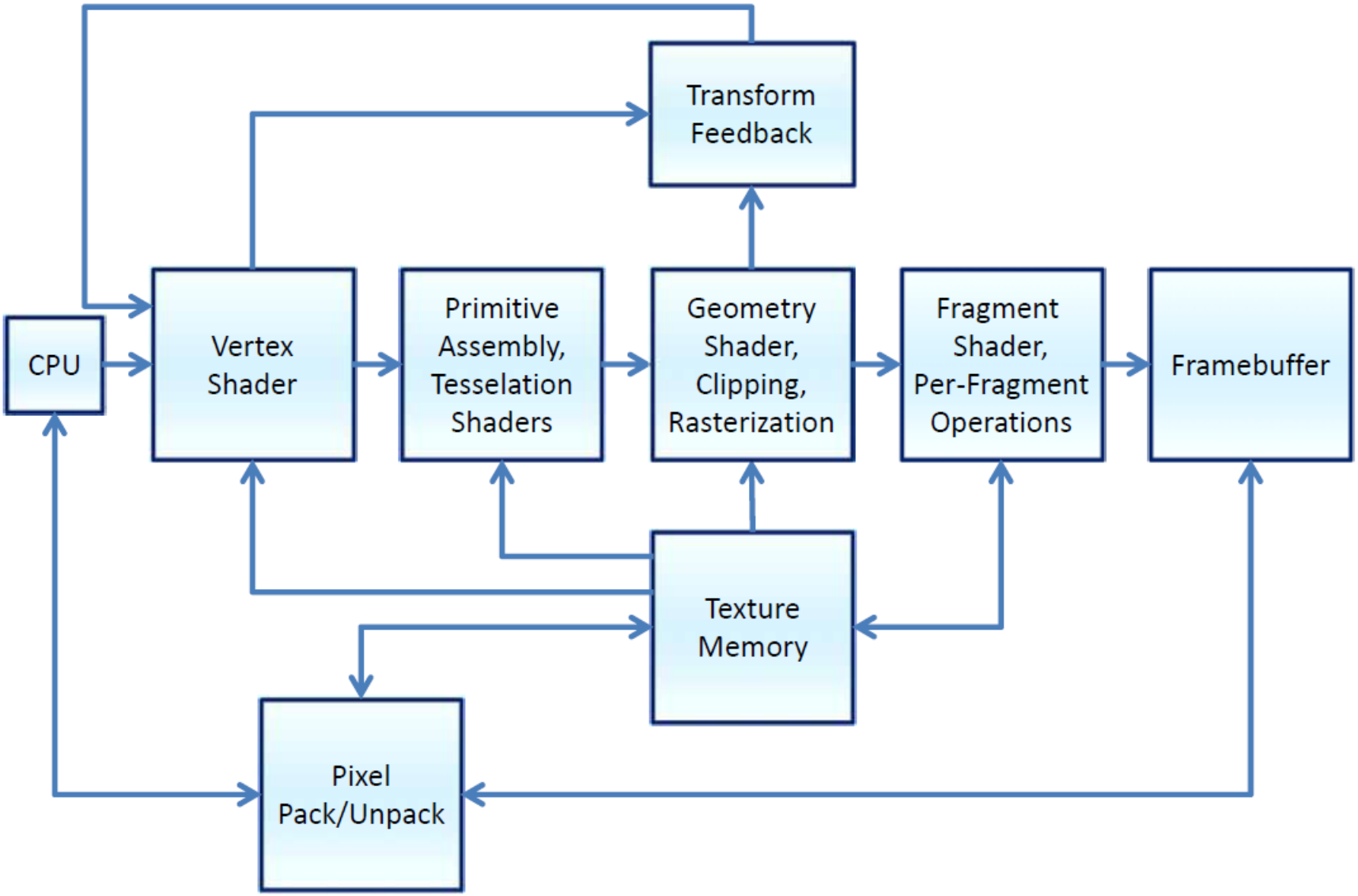
- Multiplikation mit M_{pers}/M_{orth} und anschließender Dehomogenisierung beschreibt die Abbildung „Sichtvolumen → Kanonisches Volumen“
- Die eigentliche Projektion ist dann das Weglassen der z-Komponente
- Letzter Schritt: Viewport Transformation
  - $[-1,1]^2 \rightarrow [0, screenWidth] \times [0, screenHeight]$



Moderne OpenGL Pipeline – 3.x



Moderne OpenGL Pipeline – 4.x



Modernes OpenGL

- Shader-Stufen sind programmierbar
- Vertex, Fragment, Geometry- und Tesselation-Shader
- C-ähnliche Programmiersprachen (GLSL, HLSL, ...)
  - Mit eigenen Datentypen (vec3, mat4, ... siehe glm)
- In Shadern kann flexibel auf Texturen und Buffer zugegriffen werden
- Geometrieinformationen können in den Speicher auf der Grafik-HW geladen und wiederverwendet werden
- ...

Vertex-Shader

- Eingaben sind üblicherweise per Vertex
  - Position
  - Normale
  - Farbe oder Texturkoordinate
- Ausgabe
  - Weitergeleitete Vertex-Attribute
  - Position nach MVP-Transformation

- Vorstellung
  - Für alle Vertices werden parallel die Vertex-Shader von der Grafik-HW ausgeführt
  - „Ein Thread pro Vertex“

Vertex-Shader

- Beispiel:

```
layout(location = 0) in vec3 POSITION;
layout(location = 1) in vec3 NORMAL;

out vec3 world_position;
out vec3 world_normal_interpolated;

void main() {
  gl_Position = MVP * vec4(POSITION, 1.0);

  world_position = vec3(M * vec4(POSITION, 1.0));
  world_normal_interpolated = N * NORMAL;
}
```

- **POSITION** und **NORMAL** werden beim draw-Aufruf an die Grafik-HW geschickt.
- Wie kann man die Matrizen **MVP**, **M** und **N** übergeben?

Uniforms

- Können durch OpenGL Aufrufe mit Daten gefüllt werden
- Bleiben während des gesamten draw-Aufrufs konstant

```
uniform mat4 MVP;
uniform mat4 M; // from model to world space
uniform mat3 N; // from model to world space
                // for normals

layout(location = 0) in vec3 POSITION;
layout(location = 1) in vec3 NORMAL;

out vec3 world_position;
out vec3 world_normal_interpolated;

void main() {
  gl_Position = MVP * vec4(POSITION, 1.0);

  world_position = vec3(M * vec4(POSITION, 1.0));
  world_normal_interpolated = N * NORMAL;
}
```

Fragment-Shader

- Eingaben sind üblicherweise
  - interpolierte Vertex-Attribute
  - Beleuchtungsinformationen
- Ausgabe
  - Farbe des Fragmentes

- Vorstellung
  - Für alle Fragmente werden parallel die Fragment-Shader von der Grafik-HW ausgeführt
  - „Ein Thread pro Pixel/Fragment“

Fragment-Shader

- Beispiel:

```
in vec3 world_position;
in vec3 world_normal_interpolated;

out vec4 frag_color;

void main() {
  vec3 world_normal = normalize(
    world_normal_interpolated);

  vec3 color = world_normal;
  color += vec3(1.f, 1.f, 1.f));
  color *= 0.5f;

  frag_color = vec4(color, 1.f);
}
```

- Gibt die Normaleninformation als Farbe aus
  - hilfreich zum debuggen

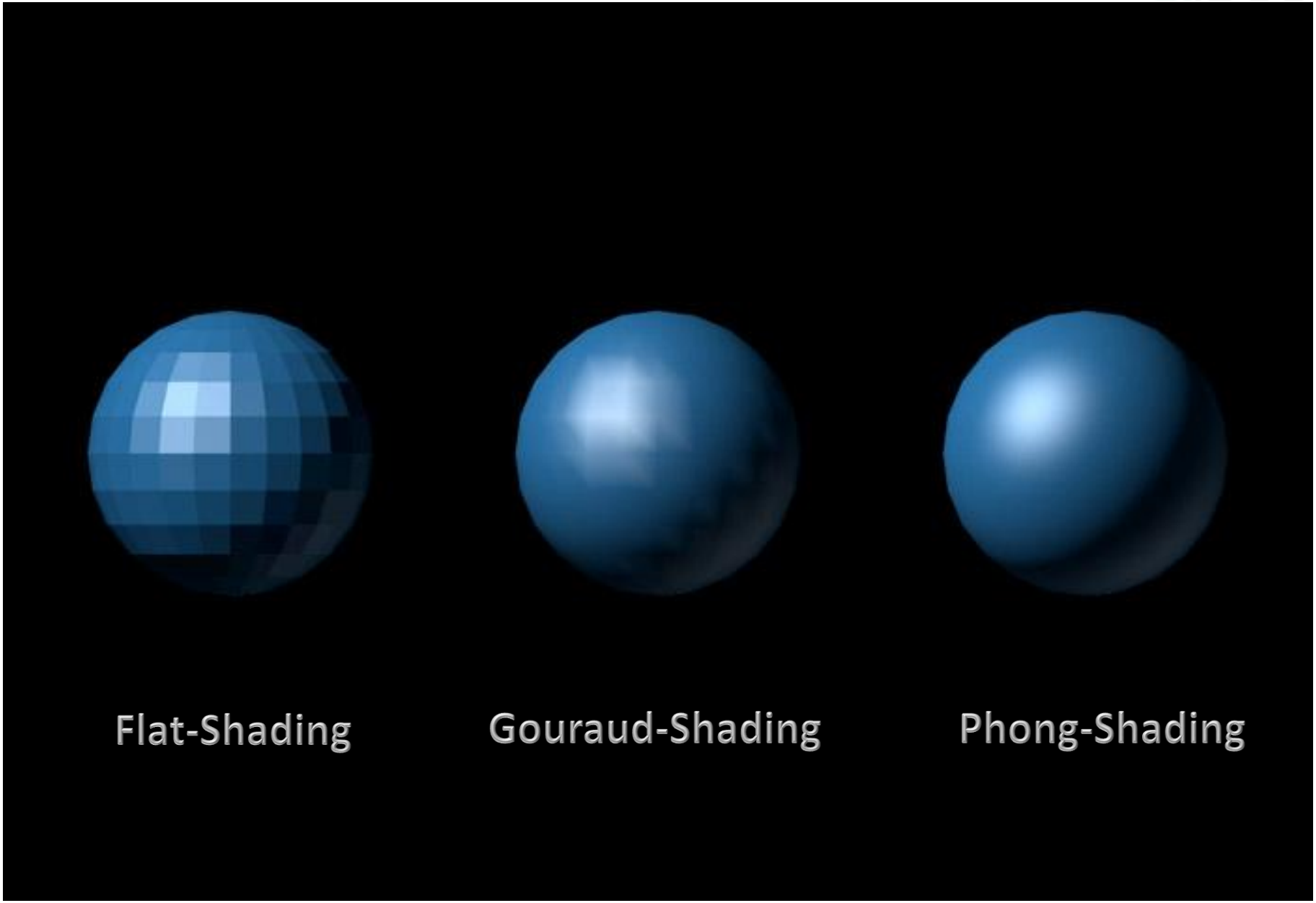
Phong-Shading != Phong-Beleuchtungsmodell

- Phong-Beleuchtungsmodell
  - Materialmodell (wie bei Ray-Tracing)
  - Diffuse + Spekulare Komponente

- Phong-Shading
  - Shading bzw. Beleuchtungsberechnung **pro Fragment** mit interpolierter Normale
  - Auch mit anderen Beleuchtungsmodellen möglich

- Gouraud-Shading
  - Beleuchtungsberechnung **pro Vertex** mit gemittelter Normale der anliegenden Facetten
- Flat-Shading
  - Beleuchtungsberechnung **pro Vertex** (oder pro Fragment) mit Facetten-Normale

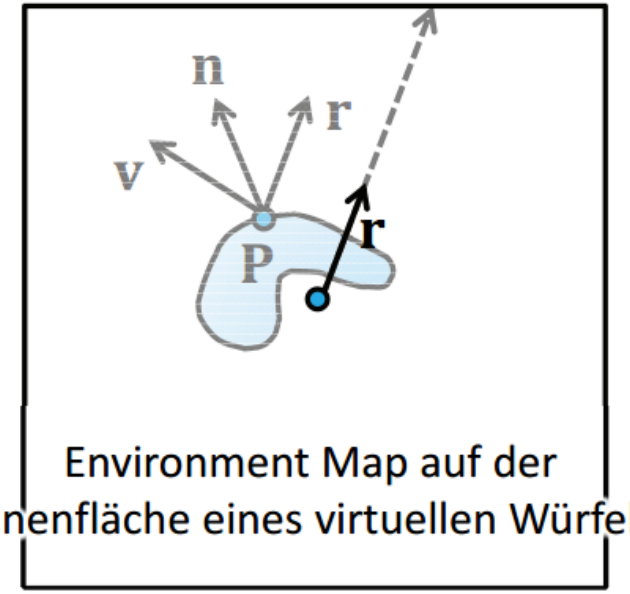
Shading Arten



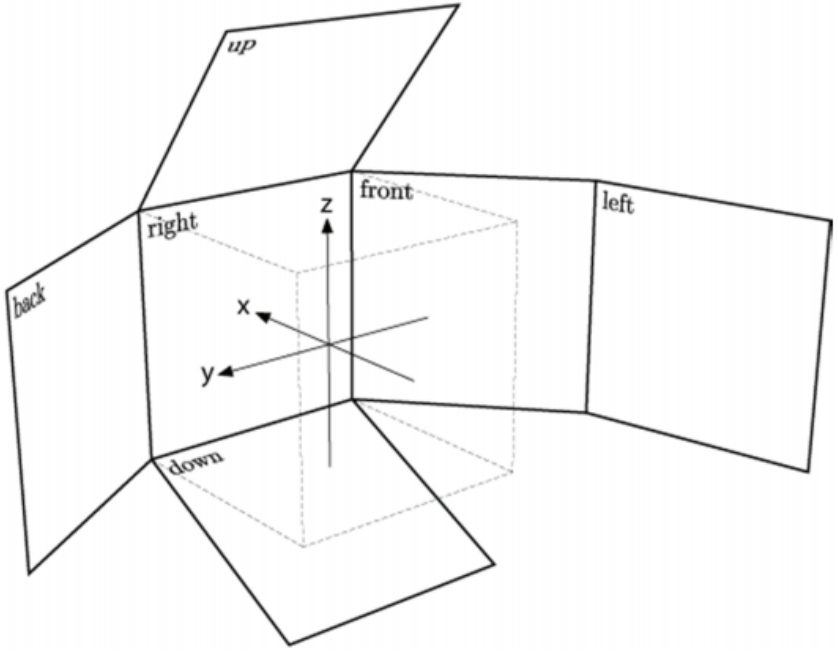
Cube Maps

Eine weitere (sehr gebräuchliche) Parametrisierung: CubeMaps

- Abbilden der Umgebung auf einen Würfel um den Betrachter
  - benötigt also 6 × 2D-Texturen
  - Abbildung/Aufnahme einer Würfelseite: perspektivische Kamera mit FOV 90°
  - Texturfilterung wie bei Mip-Mapping möglich

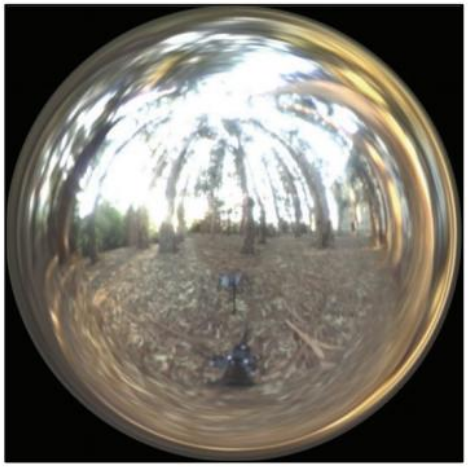


Environment Map auf der Innenfläche eines virtuellen Würfels



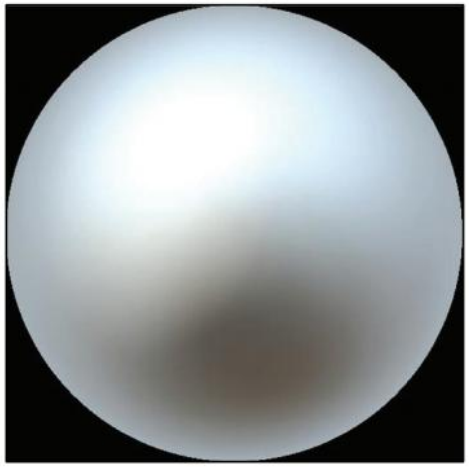
Bildbasierte Beleuchtung

- eine 2D Textur kann nur ein 2D Signal (z.B. eine Richtung) repräsentieren
  - Reflexionsrichtung: spekulare Reflexion, wie viel Licht fällt aus einem bestimmten Richtungskegel um $\mathbf{r}$ ein
  - Normale: wie viel Licht reflektiert eine diffuse Fläche mit Normale $\mathbf{n}$
  - Kombination mehrerer vorgefilterter Texturen möglich/oft verwendet
  - approx. Vorfilterung mit Summed Area Tables auch on-the-fly möglich



nicht vorgefiltert: spiegelnde Objekte (Zugriff mit $\mathbf{r}$)

vorgefiltert für imperfekte Spiegelung (Zugriff mit $\mathbf{r}$)



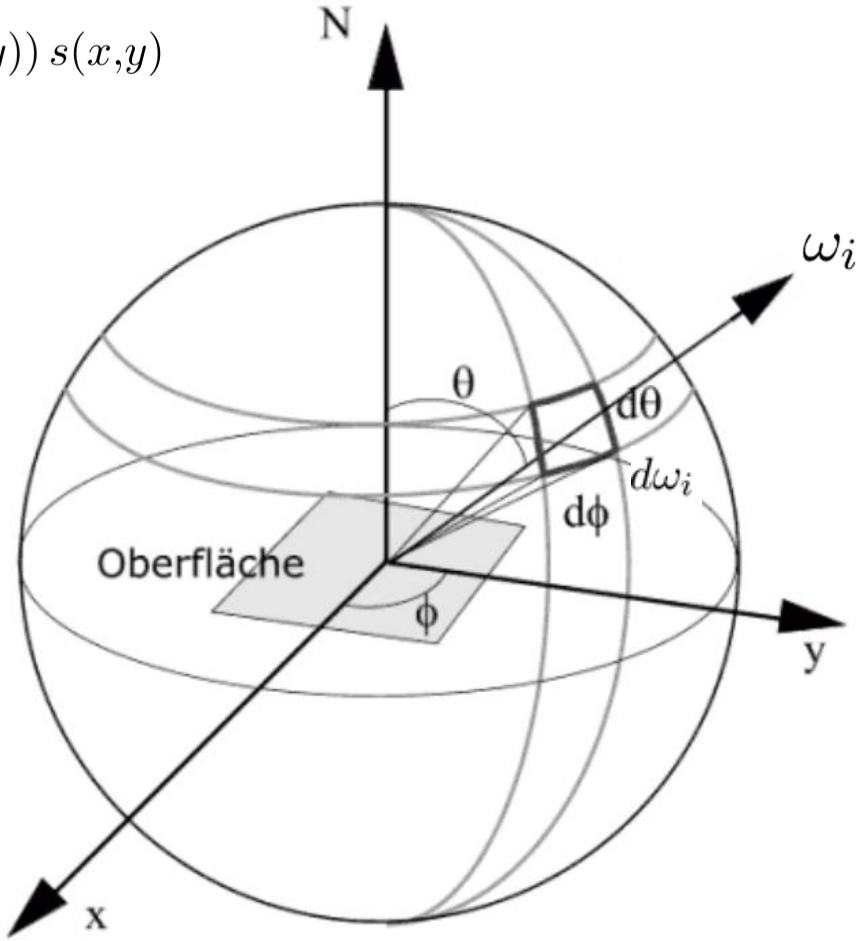
vorgefiltert für diffuse Reflexion (Zugriff mit $\mathbf{n}$)

Diffuse Vorfilterung

- Für jede Normalenrichtung wird das Beleuchtungsintegral vorberechnet

$$L_d(N) = \int_\Omega L_{in}(\omega_i)\,\max(0, N \cdot \omega_i)\,d\omega_i$$

$$\approx \sum_{x=0}^{\text{W-1}} \sum_{y=0}^{\text{H-1}} L_{in}(r(x,y))\ \max(0, N \cdot r(x,y))\,s(x,y)$$



Texturzugriffe in GLSL

- Zugriff über ‚sampler'-Uniforms
  - Repräsentiert eine Textureinheit
  - Filtert die Textur!
  - **`sampler1D, sampler2D, sampler3D`**
- Aus dem Hauptprogramm als Uniform int setzen mit **`glUniform1i`**
- Der Typ des Samplers muss zum Typ der Textur passen
- vgl. auch **`Sampler Objects`** in OpenGL

Texturzugriffe in GLSL

- Zugriffsfunktionen
  - Textur-Koordinaten in [0, 1]^2
  - Andere Werte sind zulässig (clamp/repeat)
  - **`texture`**: Normaler Zugriff
  - **`textureProj`**: Texturkoordinate wird durch .w geteilt
  - **`textureLod`**: Expliziter Zugriff auf MipMap-Stufe
  - **`textureOffset`**: Zusätzliche Verschiebung *in Texels*
  - **`textureGrad:`** Explizite Gradienten der Texturkoordinaten (für MipMap-Level)
  - Kombinationen…