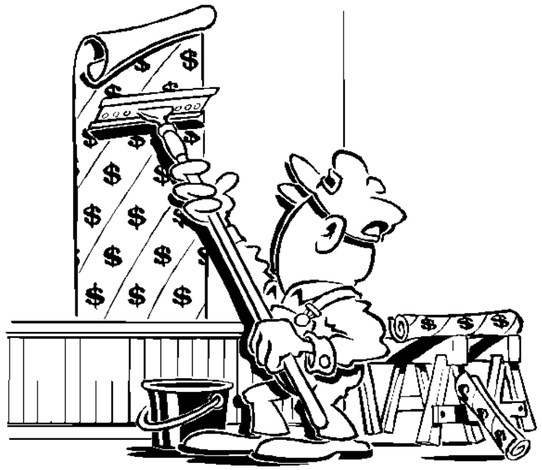
Computer-Graphik I
Texturierung



G. Zachmann
University of Bremen, Germany
cgvr.cs.uni-bremen.de

Motivation

Was fehlt?

"Shutter bug", Pixar

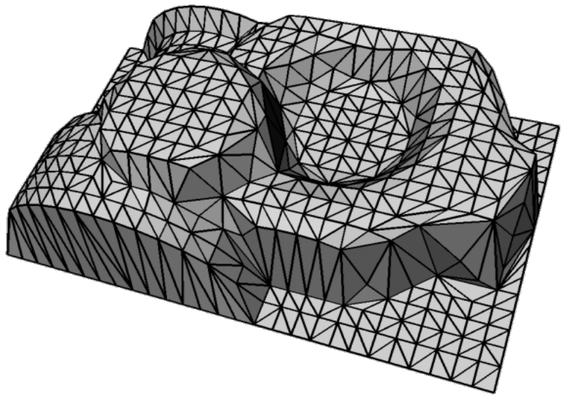
Oberflächendetails

"Shutter bug" Szene von Pixar

- Großes Spektrum geometrischer Formen und physikalischer Materialien:
  - Strukturen unebener Oberflächen, z.B. Putzwände, Leder, Schale/Rinde von Orangen, Baumstämme, Maserungen in Holz und Marmor, Tapeten mit Muster, etc.
  - Wolken
  - Objekte im Hintergrund (Häuser, Maschinen, Pflanzen und Personen)
- Solche Objekte durch Polygone (Meshes) nachzubilden ist in der Regel viel zu aufwendig

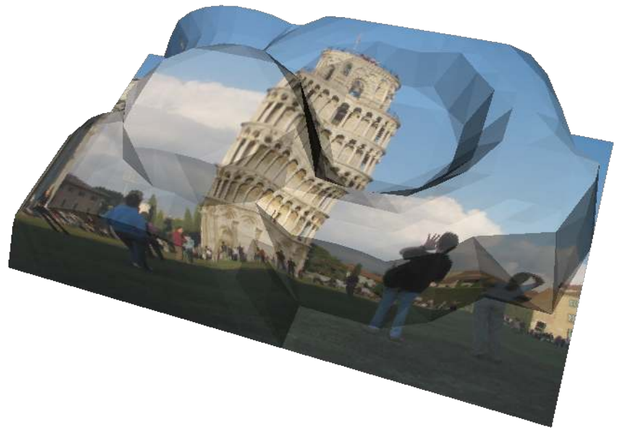
Grundidee der Texturierung

- Ziel: Visuelles Detail trotz grober Geometrie
- Idee: Objekt (grobe *Shape*) mit Textur (visuelles *Detail*) "tapezieren"

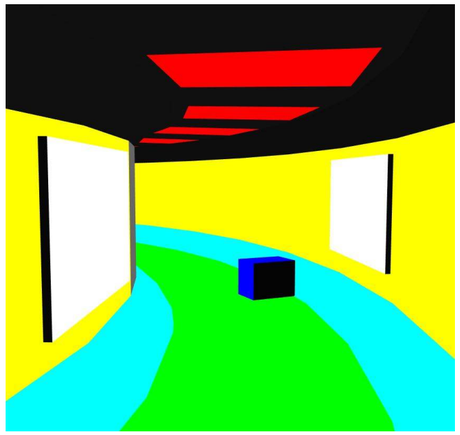


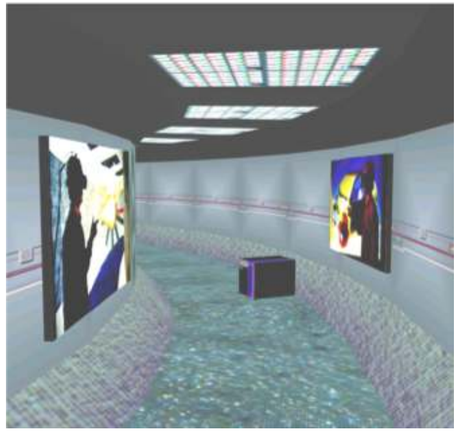
Objekt (Geometrie)

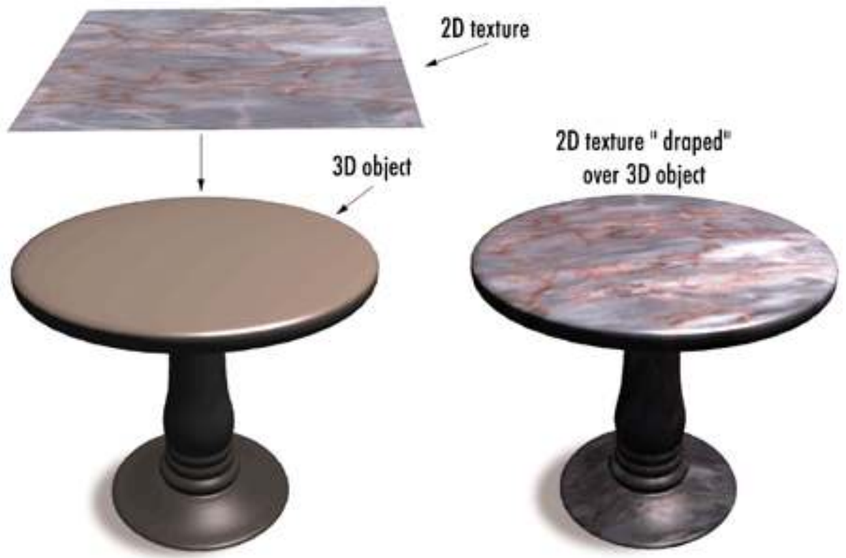
Textur (Farbe)

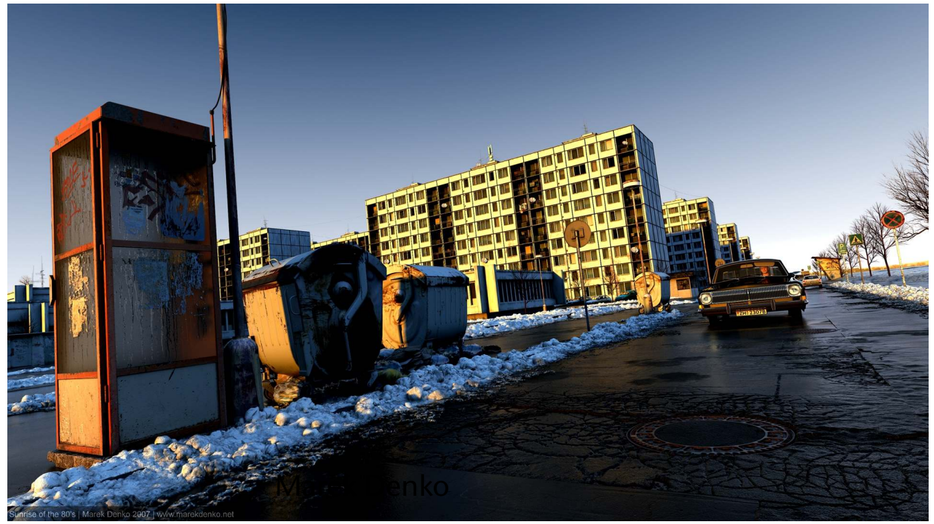


Weitere Beispiele









Eine "Kaustik-Textur", hier als Video-Textur, verstärkt den Un

Übersicht

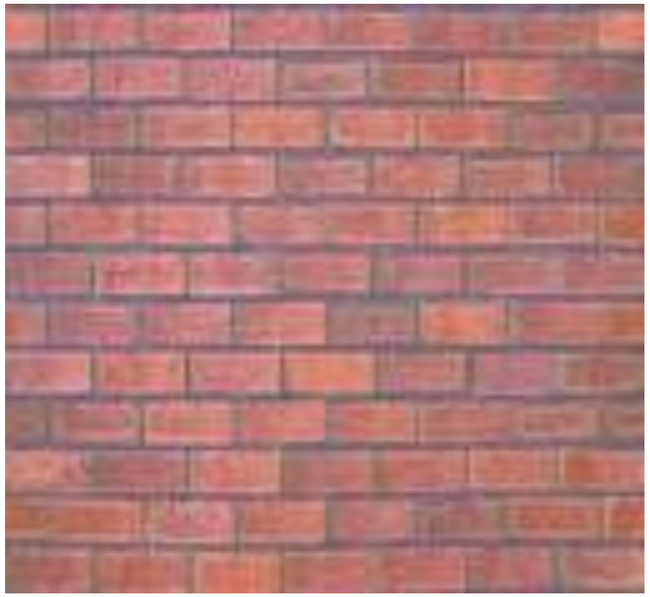
- Kategorien von Texturen:
  - Diskret = Bitmap-Bild
  - Prozedural = Farbwert an jeder Stelle wird algorithmisch berechnet bei Bedarf
- Dimension der Texturen: 1D, 2D, 3D, (4D)
- Wichtige Punkte bei den diskreten 2D-Texturen:
  1. Interpolation der Texturkoordinaten
  2. Anwendung der Textur auf die Beleuchtung o. a. Oberflächeneigenschaften
  3. Parametrisierung der Fläche
  4. Filterung
- (Wie funktioniert es in OpenGL)
- (Environment-Mapping)

Texturen kommen in verschiedenen Dimensionen

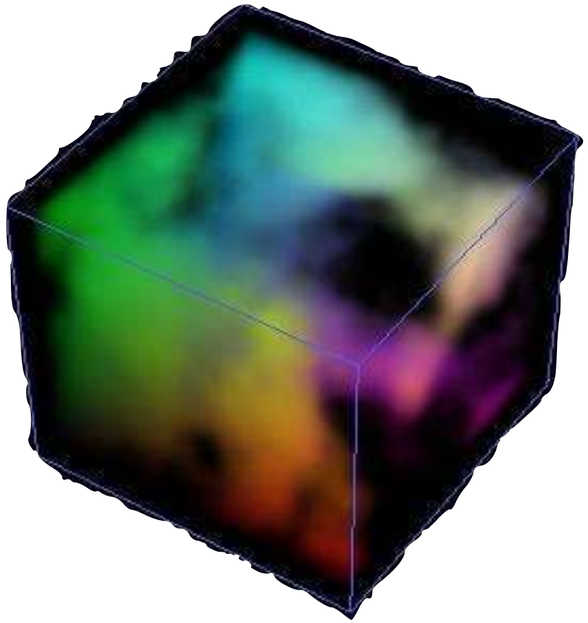
- Textur kann als Funktion einer, zweier oder dreier Koordinaten, oder als Funktion einer Richtung gesehen werden:

1D Textur

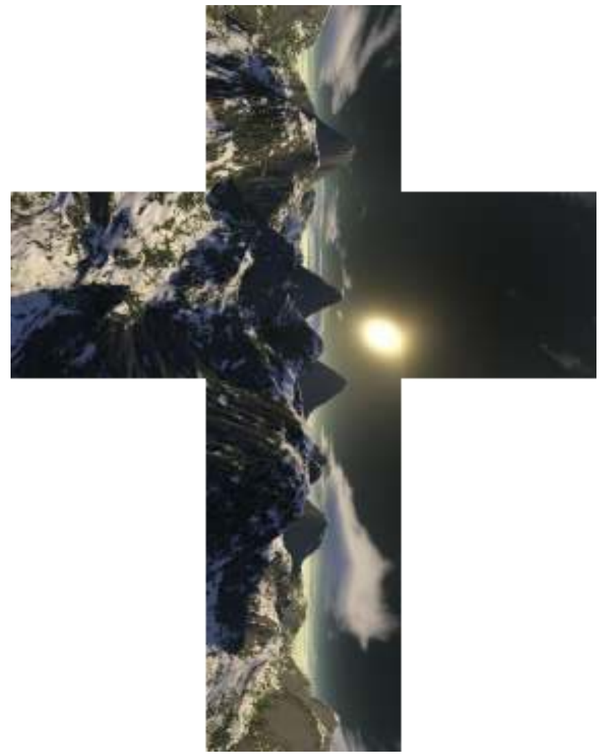
2D Textur



3D Textur



Cubemap Texturen



Einfacher Fall: 3D-Texturen

FYI

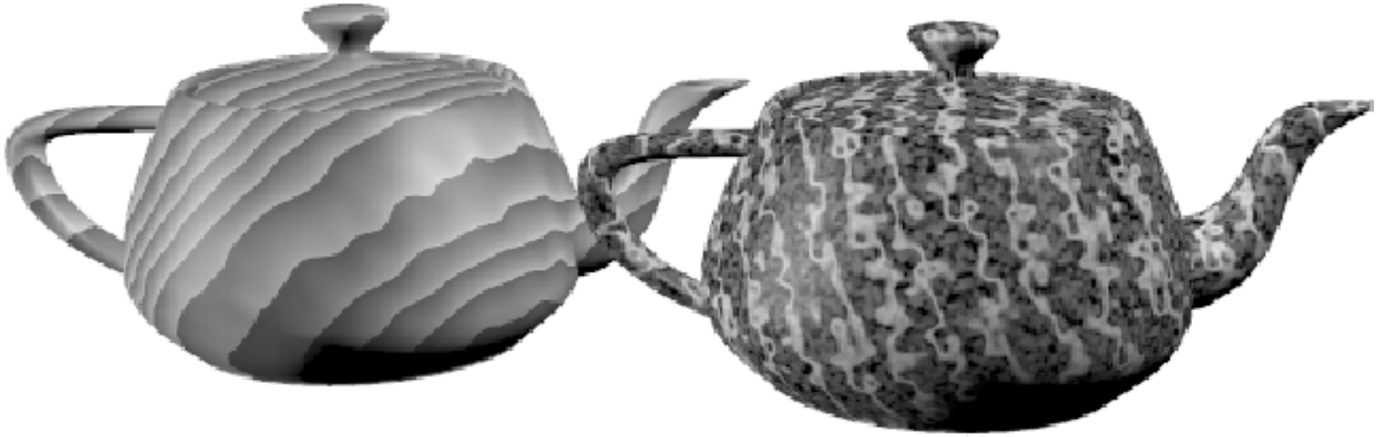
- Die lokalen Koordinaten der Objekt-Oberfläche (*x,y,z*) indizieren direkt die Textur:

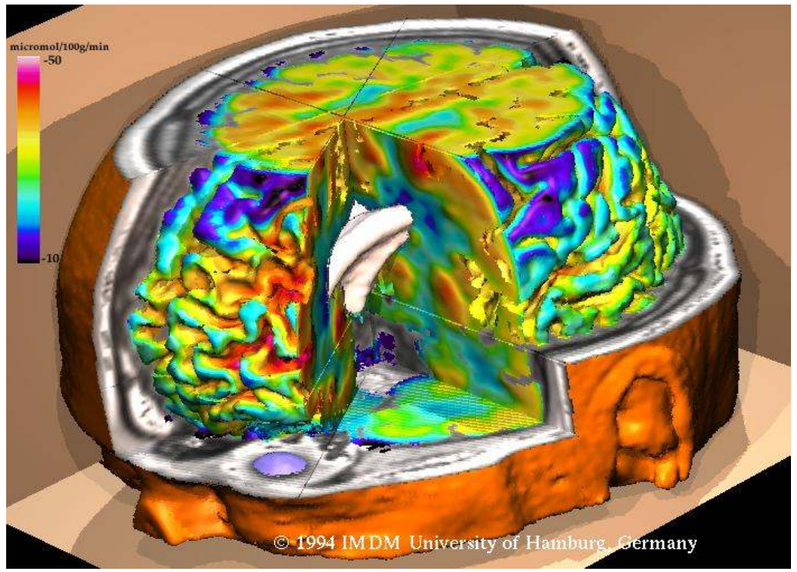
$$(r, g, b) = C_{tex}(x, y, z)$$

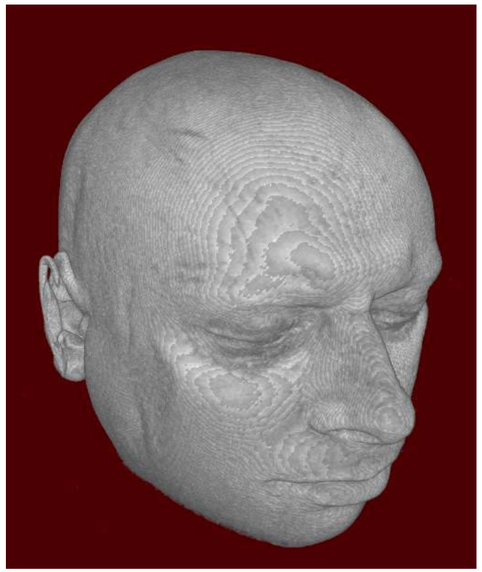
- Die Textur ist also an jedem Punkt im Raum definiert (theoretisch)
- Das Objekt wird quasi aus dem Texturvolumen "herausgeschnitzt"
- 3D-Texturen nennt man auch Festkörper-Texturen (z.B. Holz und Marmor) *("solid texture")*

Beispiele

FYI







Diskrete und prozedurale Texturen

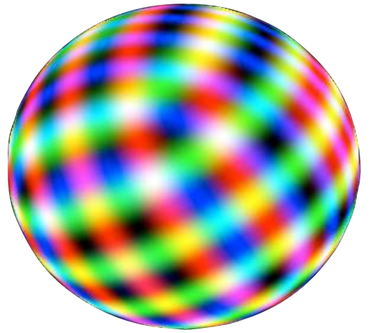
FYI

- Eine diskrete 3D-Textur = 3-dimensionales Array C[u,v,w]
  - C[u,v,w] = Vektor mit 3 Farbkomponenten = ein "Texel" (*texture element*)
  - Pro Pixel benötigt man 3 Texturkoordinaten (*u,v,w*) zum Indizieren in das Array
- Prozedurale Texturen werden an jeder Stelle im Raum aus einer mathematischen Funktion oder einem Algorithmus berechnet:

$$C_{tex}(x, y, z) := f(x, y, z)$$

- Einfaches Beispiel für eine prozedurale 3D-Textur:

$$C = \begin{pmatrix} \frac{1}{2}(1 + \sin(\frac{\pi}{w_x} P_x)) \\ \frac{1}{2}(1 + \sin(\frac{\pi}{w_y} P_y)) \\ \frac{1}{2}(1 + \sin(\frac{\pi}{w_z} P_z)) \end{pmatrix}$$



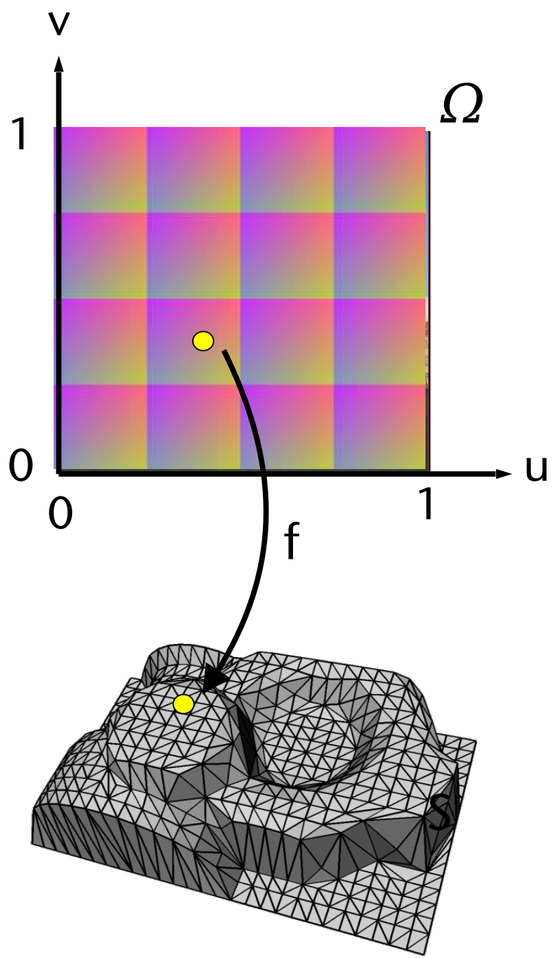
FYI

- Vorteile der prozeduralen Texturen:
  - Speicheraufwand ist minimal
  - Texturwerte können an jeder Stelle (u,v), bzw. (x,y,z) berechnet werden
  - Texturen sind im gesamten Raum definiert (kein Wrap-Around / Clamping)
  - Optimale Genauigkeit (kein Runden von Koordinaten, keine Interpolation)
- Nachteile:
  - Schwer zu erzeugen (selbst für Experten)
  - Mindestens Grundkenntnisse der Fourier-Synthese, bzw. fraktaler Geometrie erforderlich
  - Komplexere Texturen sind nahezu unmöglich
  - Kosten rel. viel Zeit (Echtzeit?)

Formale Definition von 2D-Texturen

- Zu texturierendes Objekt S = Dreiecks-Mesh
- Textur :=
  1. Parameterraum Ω (parametric domain)
  2. Pixelbild oder Funktion (diskret bzw. prozedural)
  3. Parametrisierung / Mapping = Abbildung f zwischen Textur und Objekt:

$$f : \Omega \leftrightarrow S$$



- Texturierung ist ein 2-stufiger Prozeß
  1. Inverses Mapping: $(u, v) = f^{-1}(x, y, z)$
  2. Farbe: $(r, g, b) = C_{\text{tex}}(u, v)$

Hilfsvorstellung

2D-Texturierung

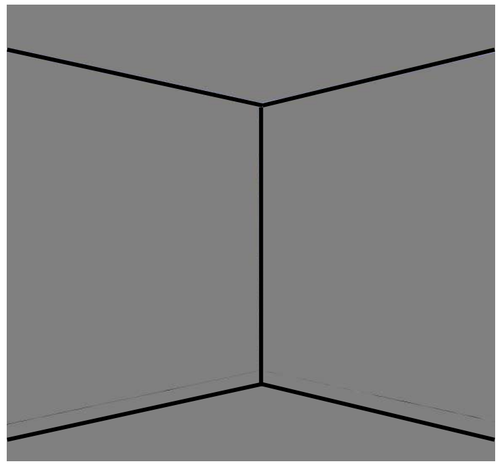
3D-Texturierung

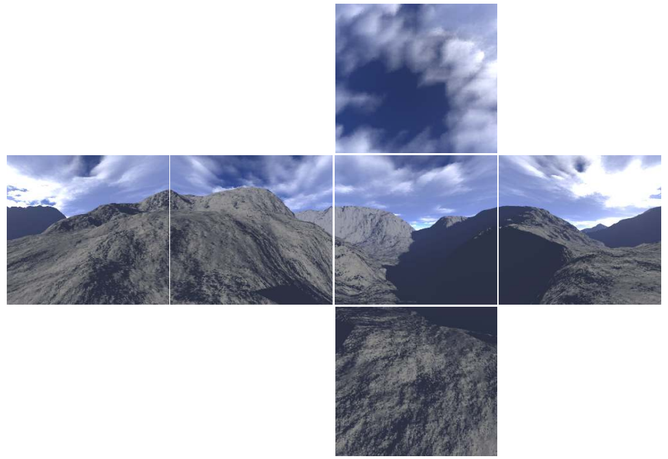
Diskrete 2D-Texturen (Bitmap-Bilder, verwendet als Textur)

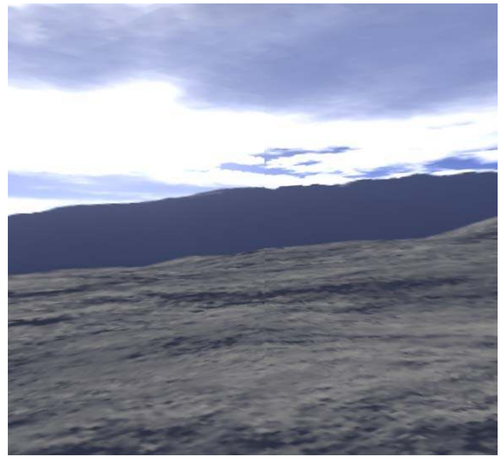
- Vorteile:
  - Vorrat an Bildern nahezu unerschöpflich
  - Erzeugung ist einfach (z.B. Photographie)
  - Anwendung auf eine Oberfläche ist sehr schnell
- Nachteile:
  - Kontext (Sonnenstand, Schattenwurf, etc.) stimmt meist nicht
  - Bilder hoher Auflösung haben großen Speicherbedarf
  - Fortsetzung meist sehr kompliziert
  - Beim Vergrößern und Verkleinern treten Artefakte auf (s. Mipmapping)
  - Verzerrung beim Mapping auf beliebige 3D Oberflächen

Einfaches Beispiel: die *Skybox*

- Die Umgebung einer virtuellen Szene modelliert man oft durch einen Würfel mit entsprechenden Texturen





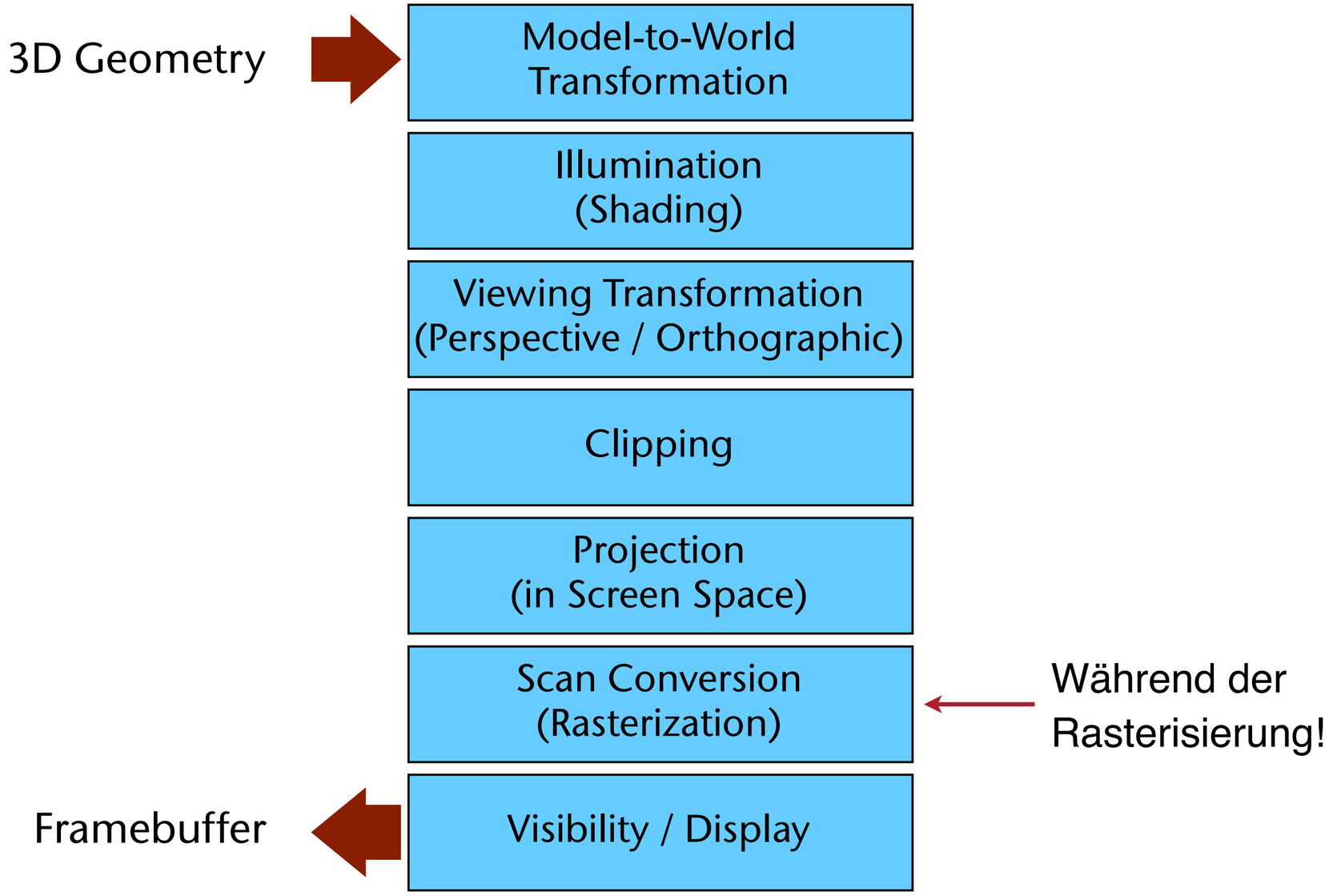


Ohne Skybox

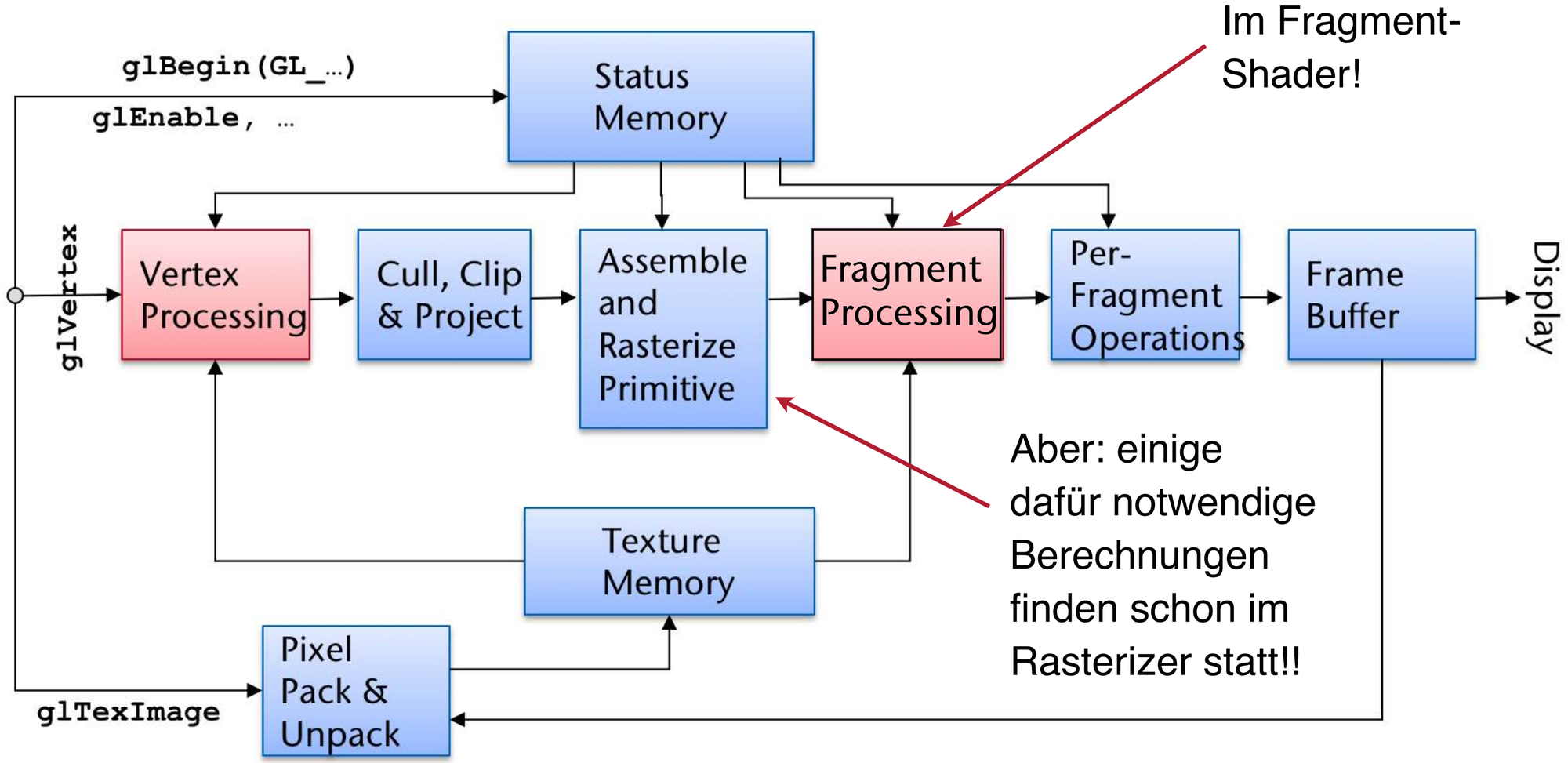
Die Skybox

Vom Boden aus

Wo wird in der (fixed function) Pipeline texturiert?

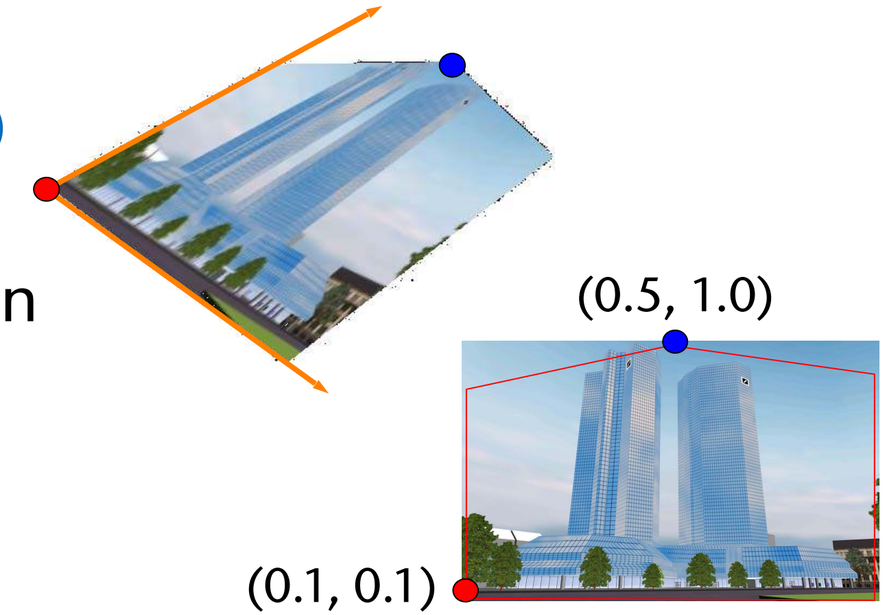


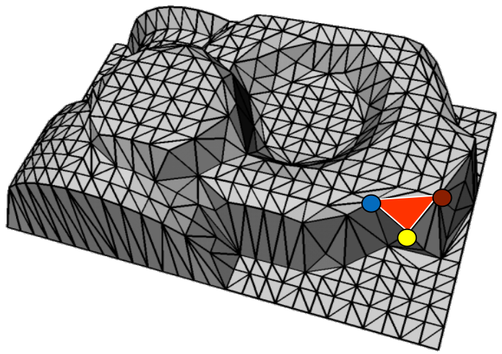
Und in der programmierbaren Pipeline

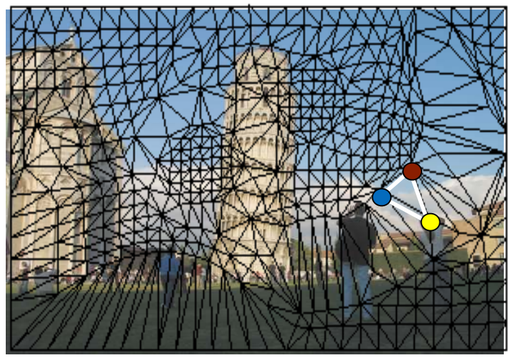


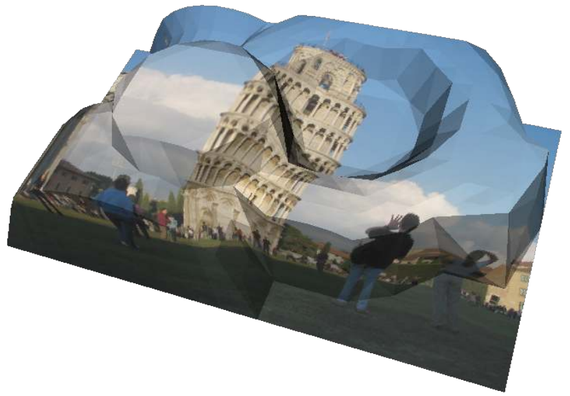
Stückweise lineare Parametrisierung mittels Texturkoordinaten

- Für jeden Vertex des Polygon-Meshes müssen zusätzlich Texturkoordinaten (= u,v-Koordinaten) definiert (oder berechnet) werden, die angeben, welcher Ausschnitt aus der Textur auf das Polygon gemappt wird
- Texturierung eines kompletten Dreiecks-Mesh durch u,v-Koordinaten für jeden Vertex:



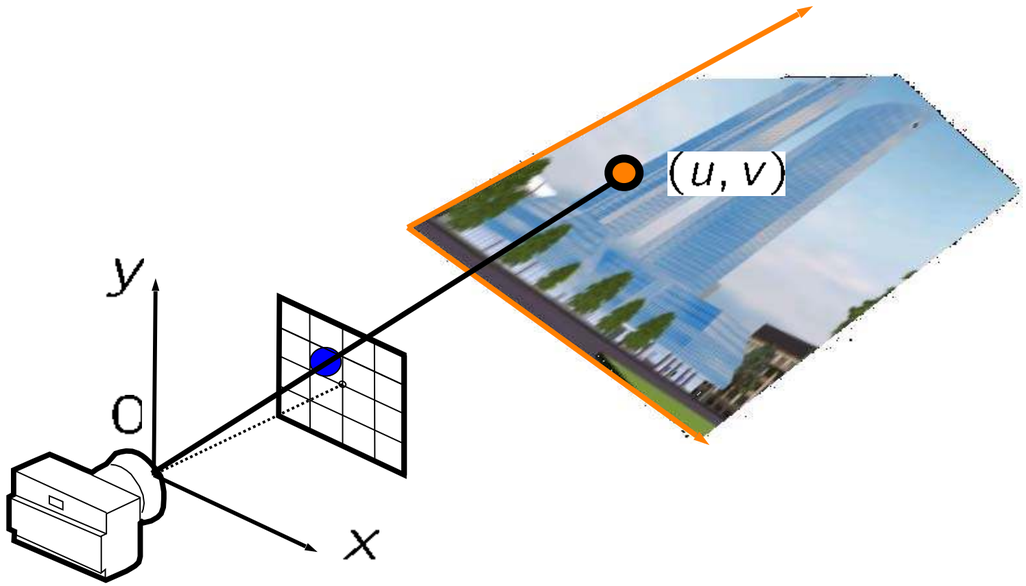


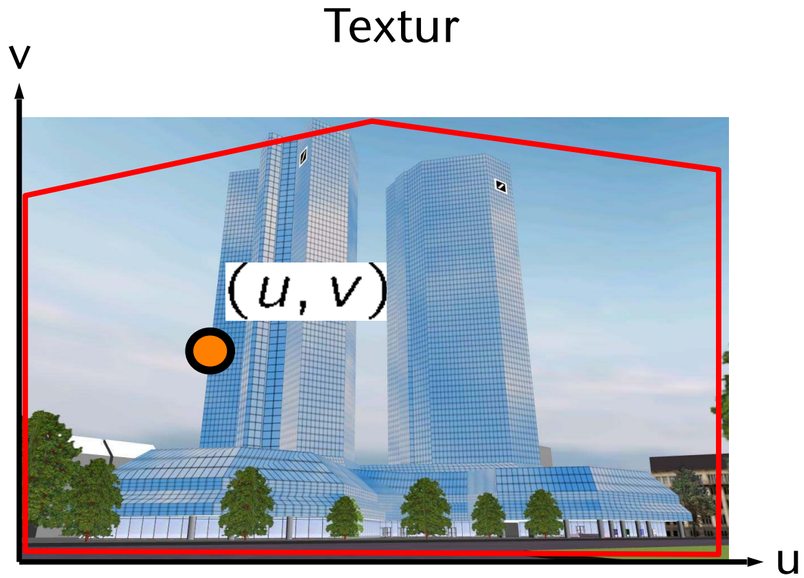




Interpolation der Texturkoordinaten

- Bei der Rasterisierung muss *für jedes Fragment* eine Texturkoordinate (*u*,*v*) aus den Texturkoordinaten der Vertices generiert werden
- Diese bestimmt im Koordinatensystem der Textur das Texel (= Pixel der Textur), das auf das Pixel (im Framebuffer) gemapt wird

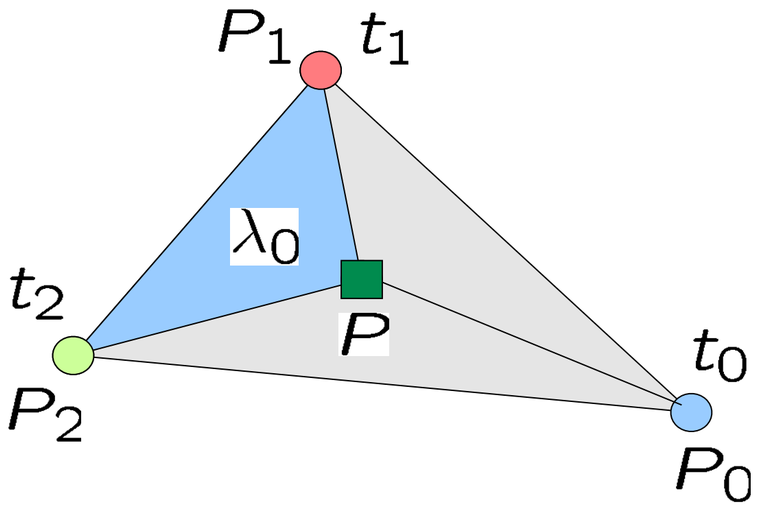




Interpolation der Textur-Koordinaten pro Fragment im Rasterizer

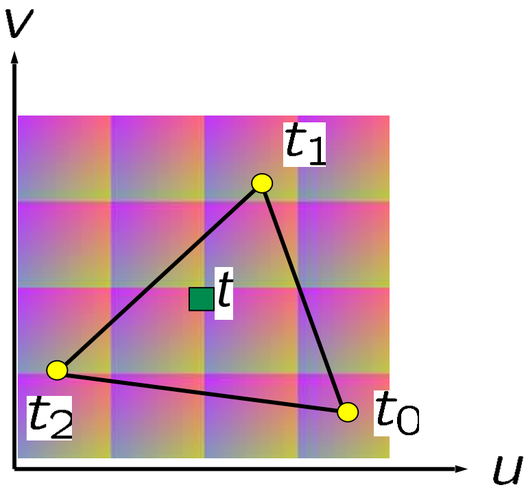
- Erinnerung: baryzentrische Koordinaten

$$\lambda_i(P) = \frac{A(P, P_{i-1}, P_{i+1})}{A(P_0, P_1, P_2)}$$



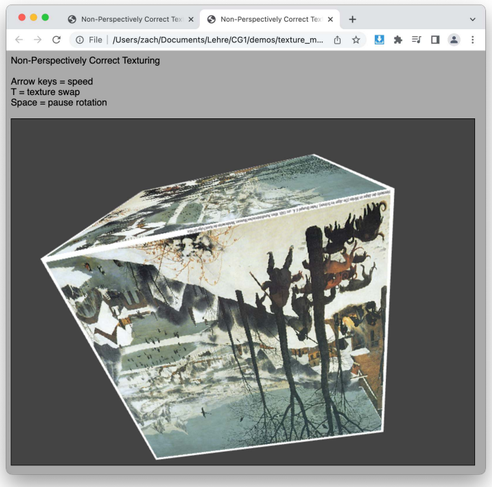
- Textur-Koordinaten durch (simplistische) baryzentrische Interpolation:

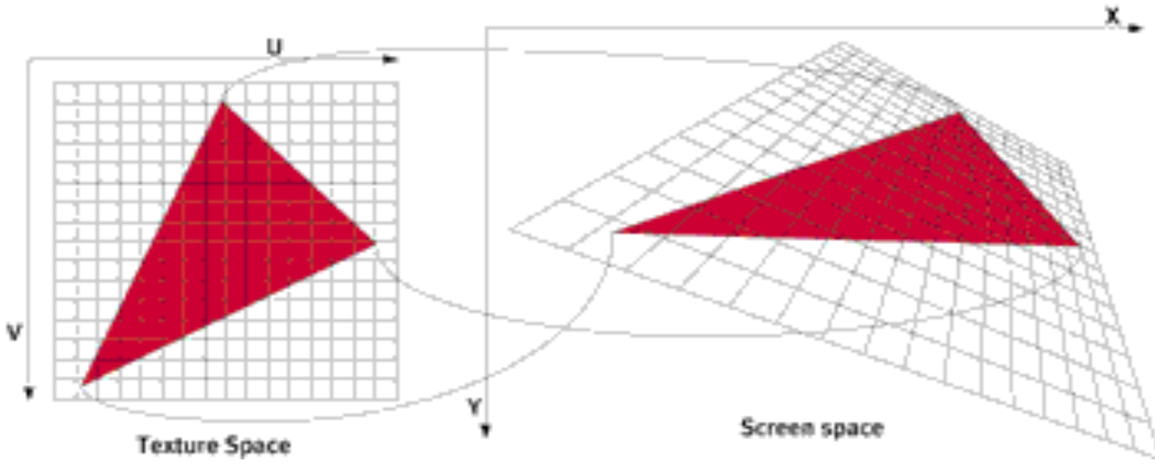
$$t(P) = \sum_{i=0}^{2} \lambda_i(P)\, t_i \qquad t_i \in \mathbb{R}^2$$

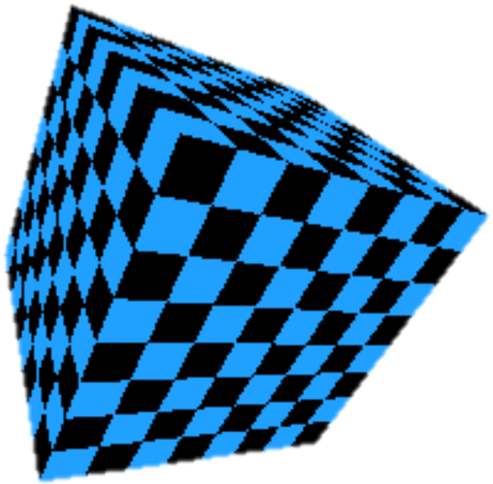


Perspektivisch korrekte Texturkoordinateninterpolation

- Problem: bei dieser einfachen, linearen Interpolation im Screen Space entstehen perspektivisch inkorrekte Bilder!
- Ursache: der Rasterizer hat die Pixel-Koordinaten nur nach der perspektivischen Division!
- Heutzutage: perspektivisch korrekte Interpolation geht







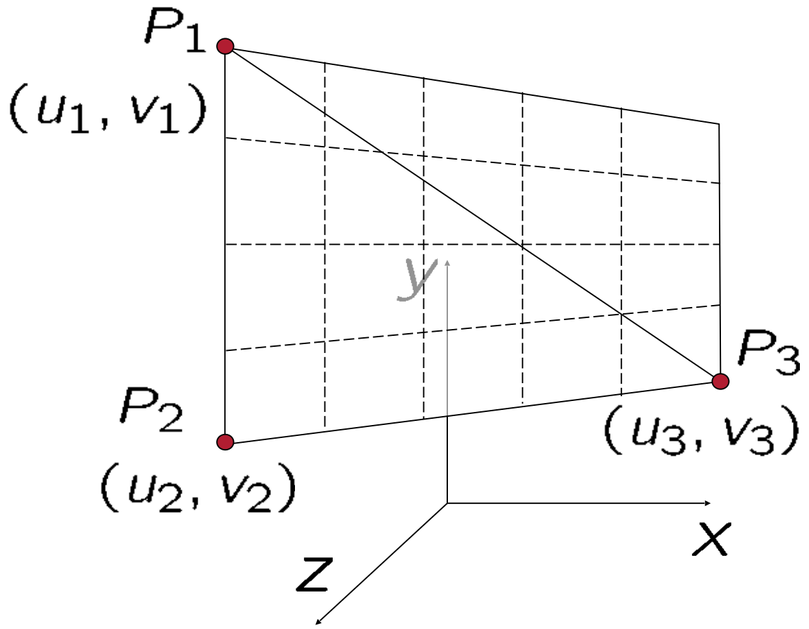
FYI (nicht klausurrelevant)

- Erinnerung: was passiert bei der perspektivischen Projektion

$$P_i = \begin{pmatrix} x_i \\ y_i \\ z_i \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} x_i \\ y_i \\ z_i \\ w_i \end{pmatrix} \equiv \begin{pmatrix} x_i/w_i \\ y_i/w_i \\ z_i/w_i \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} x_i/w_i \\ y_i/w_i \end{pmatrix} = \hat{P}_i$$

$$\text{wobei } w_i = \frac{z_i}{z_0},$$

$$z_0 = \text{Proj.ebene}$$



- Betrachte im folgenden P_1, und P_2

FYI

- Betrachte im Folgenden nur die Interpolation auf einer Linie
  - Für baryzentrische Interpolation geht das Argument analog

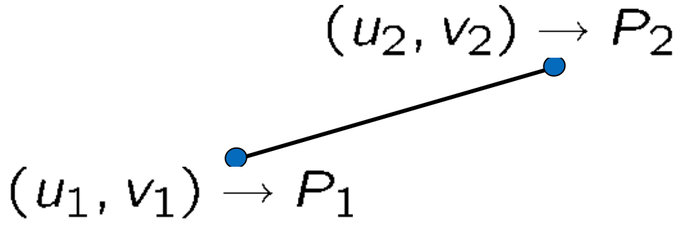
- Gegeben: *t*, zur linearen Interpolation zwischen $\hat{P}_1$ und $\hat{P}_2$, d.h.

$$\hat{P}(t) = t\hat{P}_1 + (1-t)\hat{P}_2$$

- Gesucht: Funktionen f_1, f_2 (möglichst ähnlich zu linearer Interpolation), so daß

$$\begin{pmatrix} u \\ v \end{pmatrix}(t) = f_1(t)\begin{pmatrix} u_1 \\ v_1 \end{pmatrix} + f_2(t)\begin{pmatrix} u_2 \\ v_2 \end{pmatrix}$$

die "richtigen" Texturkoordinaten für $\hat{P}$ sind



FYI

- Problem:

$$P(t) = tP_1 + (1-t)P_2 \quad t \in [0,1]$$

$$\hat{P}(s) = s\hat{P}_1 + (1-s)\hat{P}_2 \quad s \in [0,1],\ \hat{P}_i = \mathrm{Proj}(P_i)$$

ergeben zwar dieselbe Gerade auf dem Bildschirm, wenn $P(t)$ projiziert wird, aber i.A. ist

$$\mathrm{Proj}(\,P(t)\,) \neq \hat{P}(t)\ !$$

- Frage: wie sieht $\mathrm{Proj}(P(t))$ aus?

FYI

- Gegeben:

$$P(t) = t \begin{pmatrix} x_2 \\ y_2 \\ z_2 \end{pmatrix} + (1-t) \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix}$$

- O.B.d.A. betrachte nur die x-Koordinate:

$$x(t) = tx_2 + (1-t)x_1 \mapsto \frac{tx_2 + (1-t)x_1}{tw_2 + (1-t)w_1}$$

$$\text{wobei } w_i = \frac{z_i}{z_0}$$

- Behauptung:

$$\frac{tx_2 + (1-t)x_1}{tw_2 + (1-t)w_1} = \frac{x_1}{w_1} + \frac{tw_2}{w_1 + t(w_2 - w_1)}\left(\frac{x_2}{w_2} - \frac{x_1}{w_1}\right)$$

FYI

- Beweis:

$$\frac{x_1}{w_1} + \frac{tw_2}{w_1 + t(w_2 - w_1)} \left(\frac{x_2}{w_2} - \frac{x_1}{w_1} \right) =$$

$$\frac{x_1(w_1 + t(w_2 - w_1)) + tw_2 w_1 (\frac{x_2}{w_2} - \frac{x_1}{w_1})}{w_1(w_1 + t(w_2 - w_1))} =$$

$$\frac{x_1 w_1 + tw_2 x_1 - tw_1 x_1 + tw_1 x_2 - tw_2 x_1}{w_1^2 + tw_2 w_1 - tw_1^2} =$$

$$\frac{x_1 w_1 - tw_1 x_1 + tw_1 x_2}{w_1^2 + tw_2 w_1 - tw_1^2} = \frac{x_1 - tx_1 + tx_2}{w_1 + tw_2 - tw_1} =$$

$$\frac{x_1 + t(x_2 - x_1)}{w_1 + t(w_2 - w_1)} = \frac{tx_2 + (1 - t)x_1}{tw_2 + (1 - t)w_1}$$

FYI

- Gegeben:

$$\hat{P}(s) = s \begin{pmatrix} \hat{x}_2 \\ \hat{y}_2 \end{pmatrix} + (1-s) \begin{pmatrix} \hat{x}_1 \\ \hat{y}_1 \end{pmatrix}$$

- Frage: welches *s* passt zu einem gegebenen *t*, d.h., gesucht ist *s* so daß

$$\text{Proj}(P(t)) = \hat{P}(s)$$

$$s\hat{x}_2 + (1-s)\hat{x}_1 = \frac{x_1}{w_1} + s\left(\frac{x_2}{w_2} - \frac{x_1}{w_1}\right)$$

$$\Rightarrow \quad s = \frac{tw_2}{w_1 + t(w_2 - w_1)}$$

$$\Rightarrow \quad t = \frac{sw_1}{w_2 + s(w_1 - w_2)}$$

- Mit diesem *t* kann man die Texturkoordinaten *(u,v)* linear interpolieren!

FYI

- Analog funktioniert es bei 3 baryzentrischen Koordinaten:

$$u(P) = \frac{\sum_{i=0}^{2} \lambda_i(P) u_i}{\sum_{i=0}^{2} \lambda_i(P) w_i}$$

Moment Mal!

- War die Interpolation von Farben in Dreiecken falsch?
- Was ist der Unterschied zwischen Interpolation von Farben und der Interpolation von Textur-Koordinaten?!
- Kein Unterschied ...
- Dann hätten wir Farben auch perspektivisch korrekt interpolieren müssen!
- Richtig.
- Perspektivisch-korrekte Interpolation von Vertex-Attributen ist heutzutage der Default

Modulation der Beleuchtung durch Texturen

- Wie/wo kann ein Texturwert (Texel) die Beleuchtungsrechnung beeinflussen?
- Erinnerung: die Microfacet BRDF

$$\rho = \rho_{\text{diff}} + \rho_{\text{spec}}$$

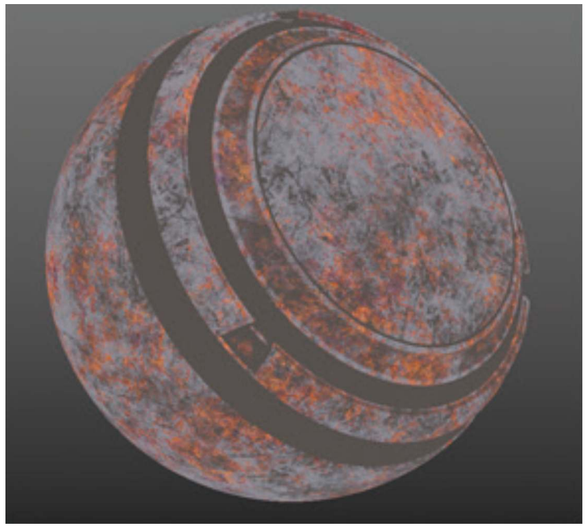
$$\rho_{\text{diff}}(\boldsymbol{l}, \mathbf{v}) = \frac{1}{\pi} C_{\text{base}} \qquad \rho_{\text{spec}}(\boldsymbol{l}, \mathbf{v}) = \frac{F(\boldsymbol{l}, \mathbf{h}) D(\mathbf{h}) G(\boldsymbol{l}, \mathbf{v})}{\text{Normierung}}$$

$$L_o(\mathbf{v}) = \sum_{i=1}^{n} \rho(\boldsymbol{l}_i, \mathbf{v}) \cdot (\mathbf{n} \cdot \boldsymbol{l}_i) \cdot L_i(\boldsymbol{l}_i)$$

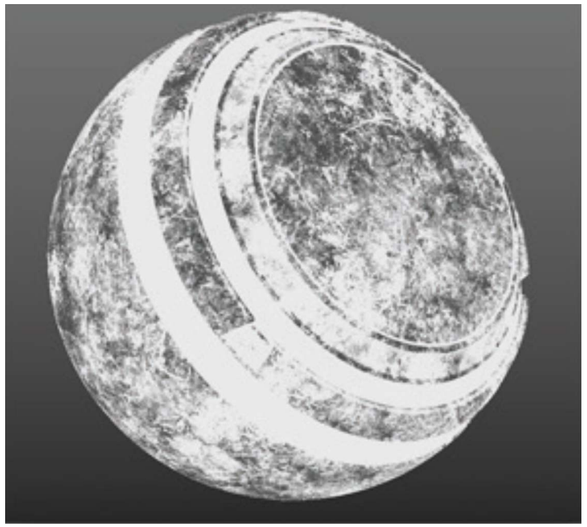
(im Falle von diskreten Punkt-/Richtungslichtquellen)

- Viele Parameter in diesen Gleichungen kommen i.A. aus einer Textur
  - Insbesondere: baseColor, Roughness, Metallic, Alpha, etc.
  - Manchmal auch: (makroskopische) Normale, **n**, oder eintreffende Lichtmenge, L_i

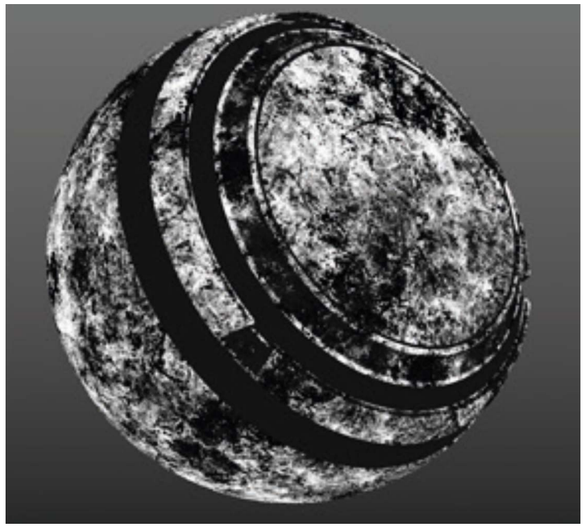
Beispiele



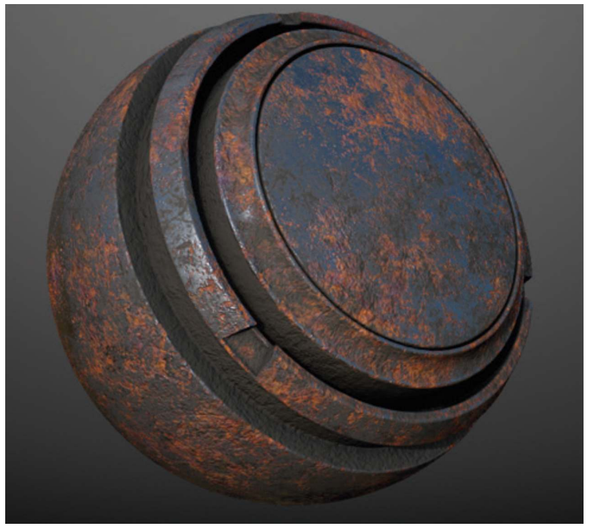
baseColor



roughness

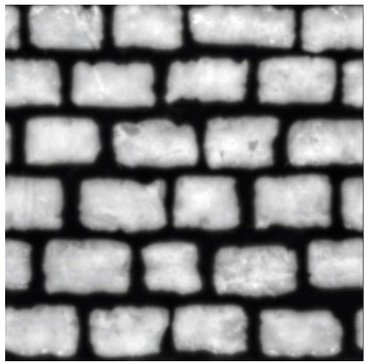


metallic



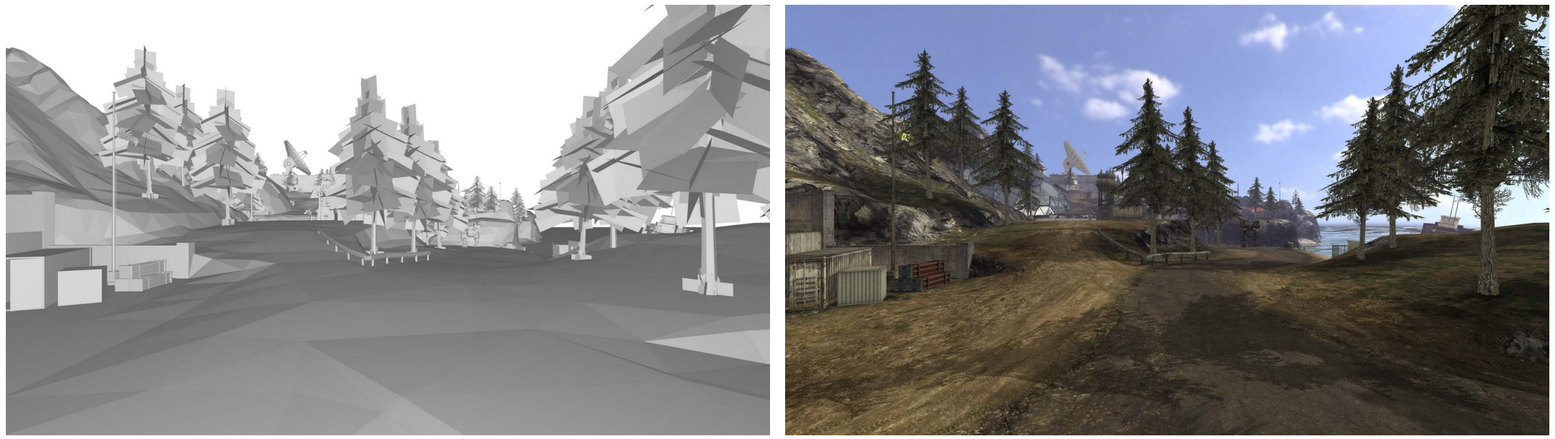
result

height map
for bump
mapping

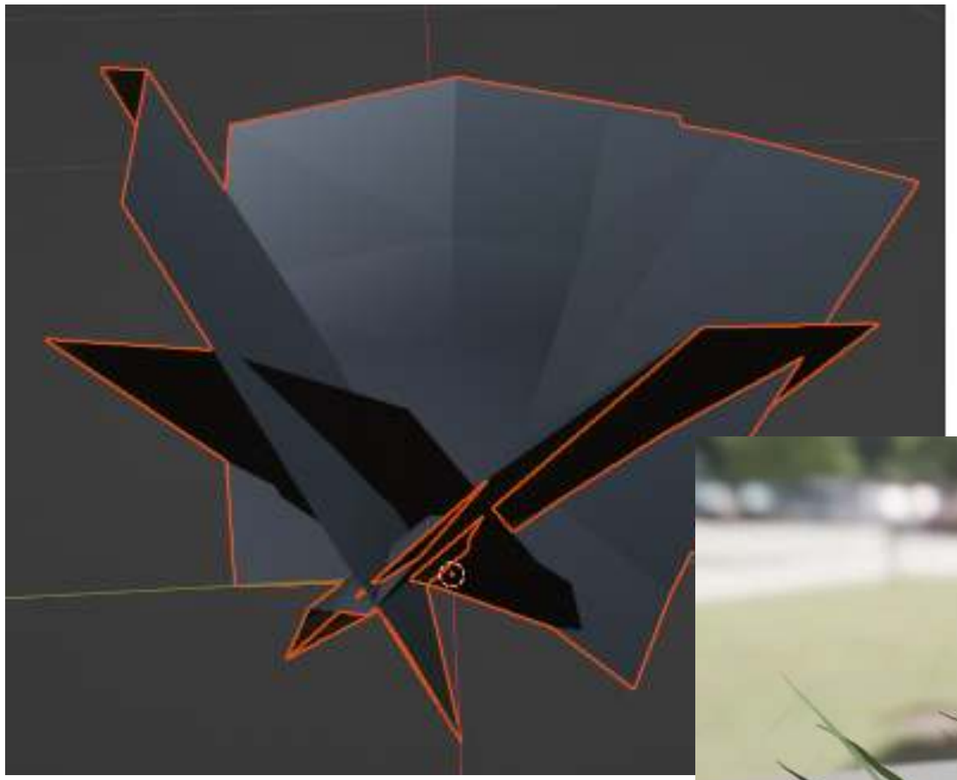


Modulation der Transparenz ("Alpha-Texturen")

- Speichern der „Durchsichtigkeit“ in einer Textur: $(r, g, b, \alpha)_{x,y} = C_{\text{tex}}(u, v)$
- Pixel mit α=0 sind voll durchsichtig, Pixel mit α=1 sind undurchsichtig (opak)
- Ermöglicht komplexe Umrisse auf einfache Art



Enemy Territory: Quake Wars. Splash Damage & Intel



Nur die Vertices der "Trägerpolygone" werden hier animiert

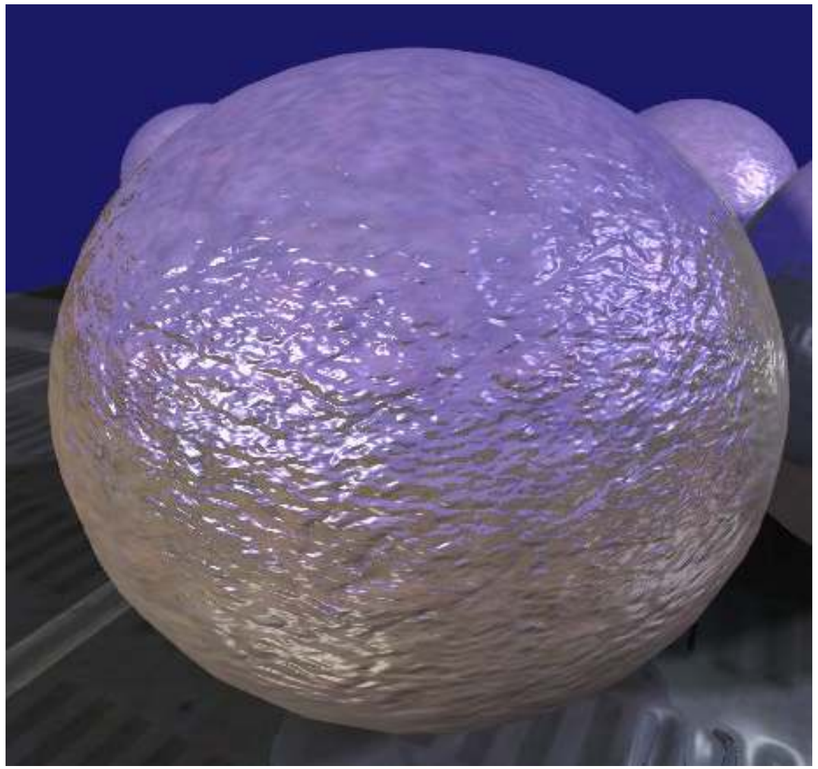

Perspective
Lit
Show

Perturbation der Normale (Bump-Mapping)

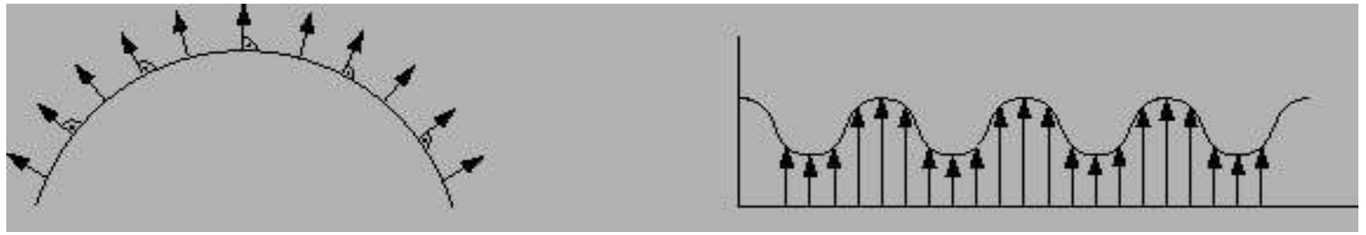
- Speichere Höhenwerte einer Offsetfläche in einer Textur
- Berechne eine korrigierte Normale pro Pixel aus den Höhenwerten der *Bump-Map*:

$$\mathbf{n}(x, y) = f(C_{\text{tex}}(u, v))$$

- Verwende diese Normale im Lighting-Model (als makroskopische Normale)



- Ziel: berechne korrekte Normale aus der einfachen Geometrie + Höhenfeld
- Interpretiere Höhenfeld als Offset entlang der orig. Normale = skalare Textur
  $\longrightarrow$ Bump-Map



Original-Oberfläche $P(u, v)$ mit Normalen $N(u, v)$

Bump-Map $F(u, v)$

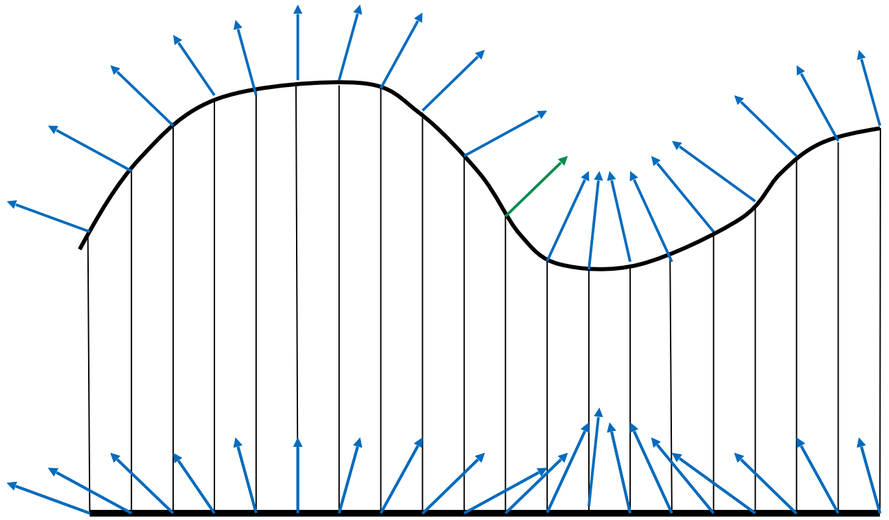
Offset-Oberfläche $\hat{P}$

Perturbierte Normalen $\hat{N}(u, v)$

- Resultierende Oberfläche :

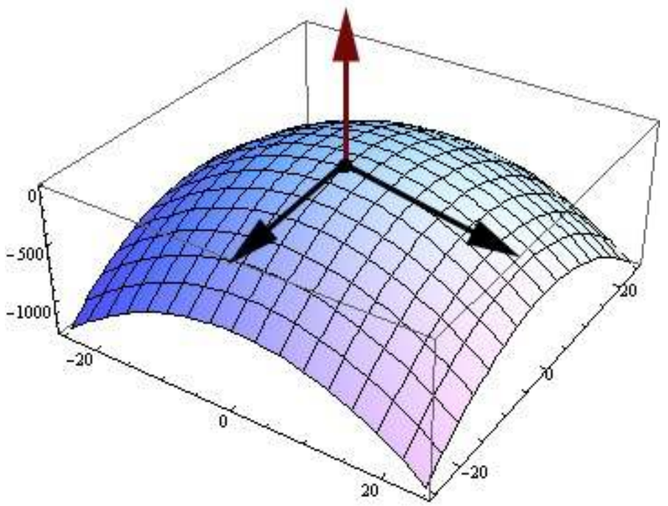
$$\hat{P}(u, v) = P(u, v) + F(u, v)\frac{N(u, v)}{\|N(u, v)\|}$$

- Beobachtung: ins Beleuchtungsmodell geht nicht direkt $P(u, v)$, sondern nur $N(u, v)$ ein.
- Hauptidee des Bump-Mapping: für kleine Unebenheiten genügt Rendering von $P(u, v)$ mit Normalen $\hat{N}(u, v)$



- Berechnung von $\hat{N}(u, v)$:

$$\hat{N}(u, v) = \hat{P}_u(u, v) \times \hat{P}_v(u, v)$$



- Richtungsableitungen mit Summen- und Kettenregeln:

$$\hat{P}_u(u,v) = P_u(u,v) + F_u(u,v)\frac{N(u,v)}{\|N(u,v)\|} + F(u,v)\frac{d}{du}\frac{N(u,v)}{\|N(u,v)\|}$$

$$\hat{P}_v(u,v) = P_v(u,v) + F_v(u,v)\frac{N(u,v)}{\|N(u,v)\|} + F(u,v)\frac{d}{dv}\frac{N(u,v)}{\|N(u,v)\|}$$

- Falls $F(u,v)$ klein → Weglassen des letzten Teilterms:

$$\hat{P}_u(u,v) \approx P_u(u,v) + F_u(u,v)\frac{N(u,v)}{\|N(u,v)\|}$$

$$\hat{P}_v(u,v) \approx P_v(u,v) + F_v(u,v)\frac{N(u,v)}{\|N(u,v)\|}$$

- Für $\hat{N}(u, v)$ folgt damit:

$$\hat{N} = \hat{P}_u \times \hat{P}_v$$

$$= P_u \times P_v + F_u \left(\frac{N}{\|N\|} \times P_v \right) + F_v \left(P_u \times \frac{N}{\|N\|} \right) + F_u F_v \left(\frac{N}{\|N\|} \times \frac{N}{\|N\|} \right)$$

$$= P_u \times P_v + F_u \left(\frac{N}{\|N\|} \times P_v \right) + F_v \left(P_u \times \frac{N}{\|N\|} \right)$$

$$= N + \frac{1}{\|N\|} \left(F_u (N \times P_v) - F_v (N \times P_u) \right)$$

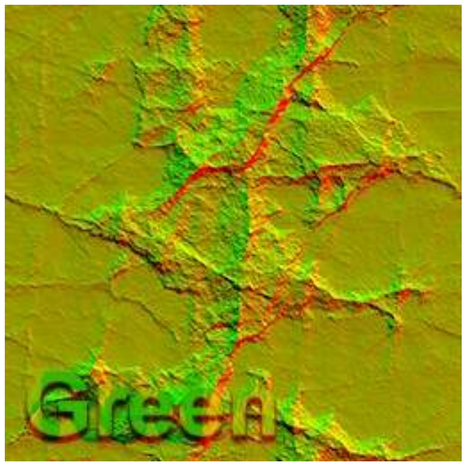
Bemerkungen

- Die Ableitungen F_u und F_v können mit finiten Differenzen approximiert werden
- Finite Differenzen auf uniformem Gitter der Gittergröße h (im 1D)
  - Vorwärtsdifferenzen: $$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{h}$$
  - Rückwärtsdifferenzen: $$f'(x_i) = \frac{f(x_i) - f(x_{i-1})}{h}$$
  - Zentrale Differenzen: $$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{2h}$$

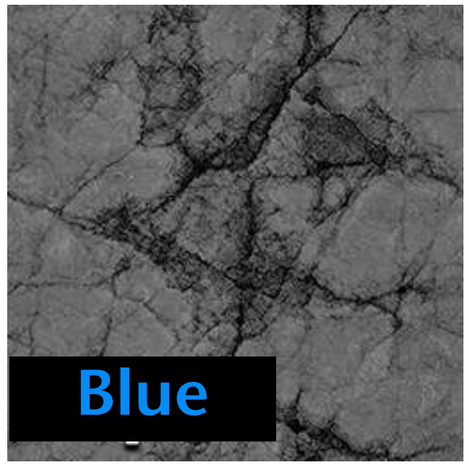
- Speicherung:
  - Richtungsableitungen (mit finiten Differenzen berechnet) in R-/G-Kanal speichern
  - Blau-Kanal übrig für z.B. Roughness

Ableitung in u-Richtung

Ableitung in v-Richtung

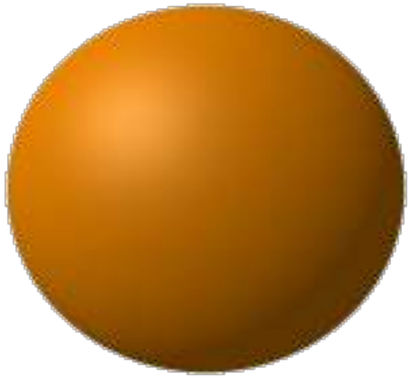


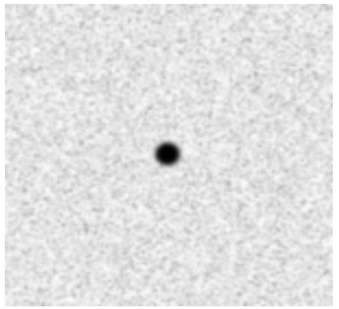
Kombiniert in R-/G-Kanäle



Smoothness (=1 - Roughness)

Beispiele





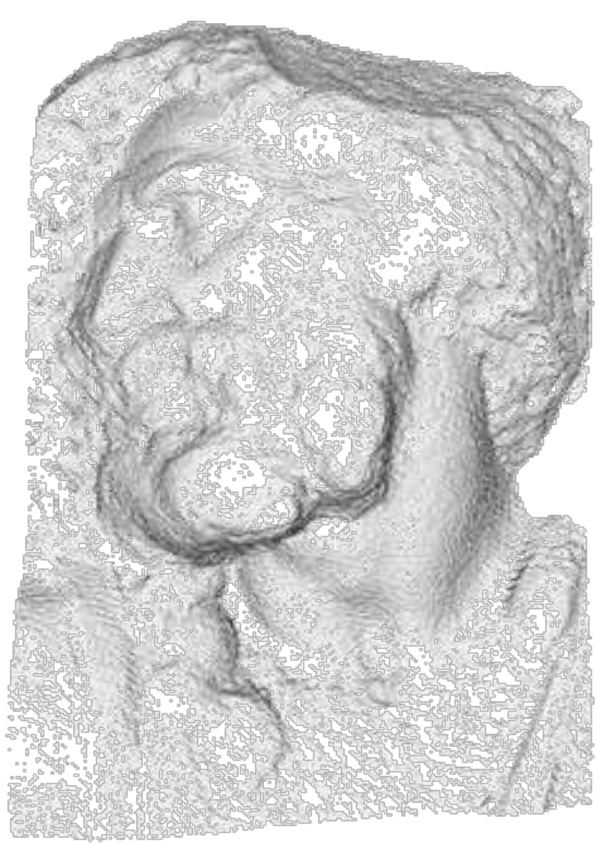
- (•) No bump mapping
- ( ) Normal mapping
- ( ) Parallax mapping
- ( ) Steep parallax mapping
- ( ) Parallax occlusion mapping

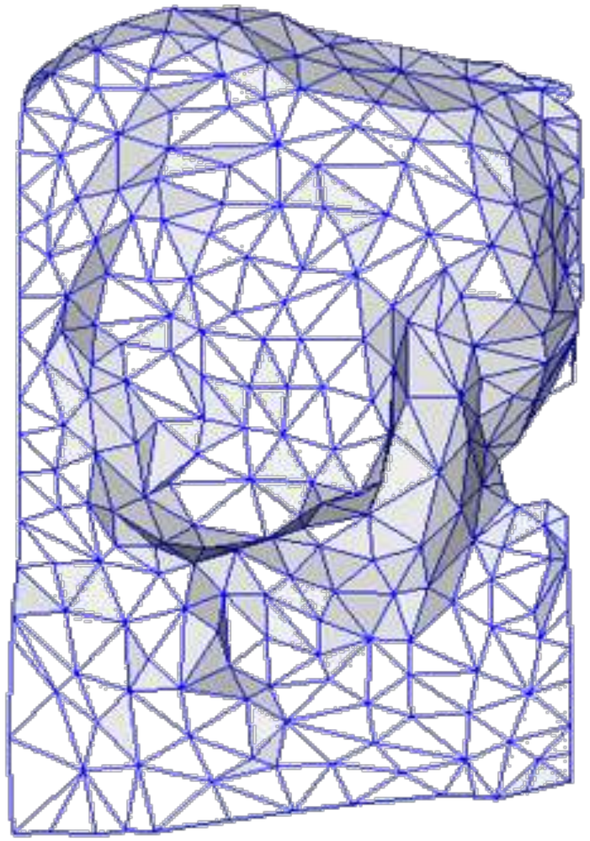
Bzgl. Parallax Mapping →
siehe die VL "Advanced Computer Graphics"

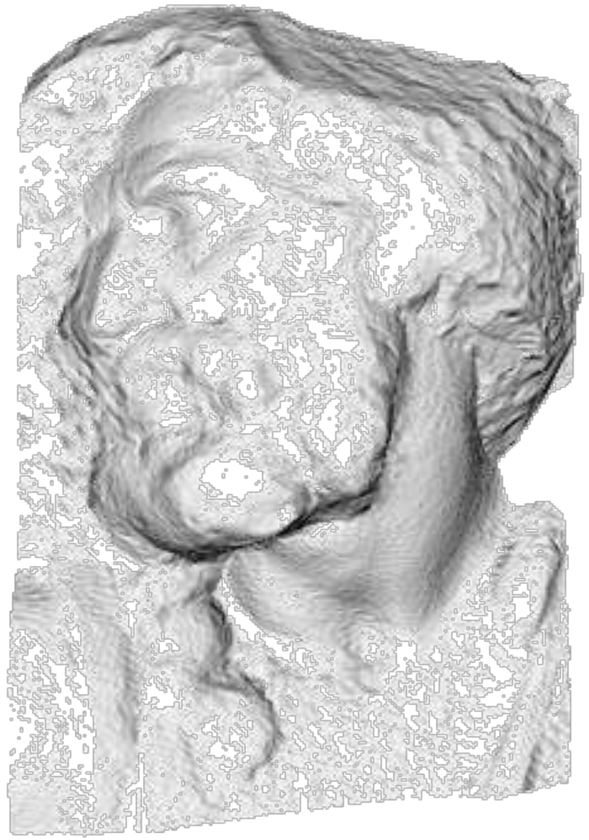
Enemy Territory: Quake Wars. Splash Damage & Intel

Normal Maps

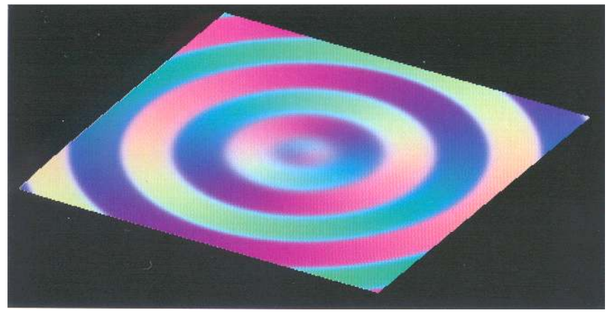
- Normalen in hoher Auflösung in Textur speichern
- Auf niedrig aufgelöste Geometrie mappen und im Lighting-Modell die Normale aus der Textur holen

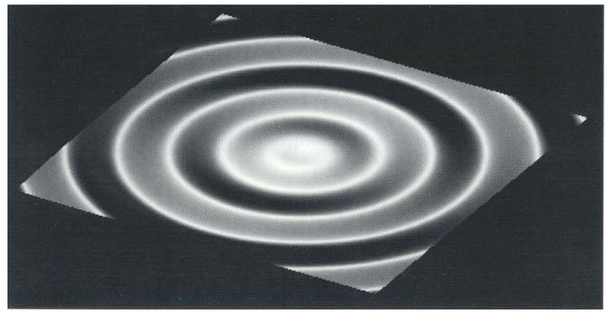


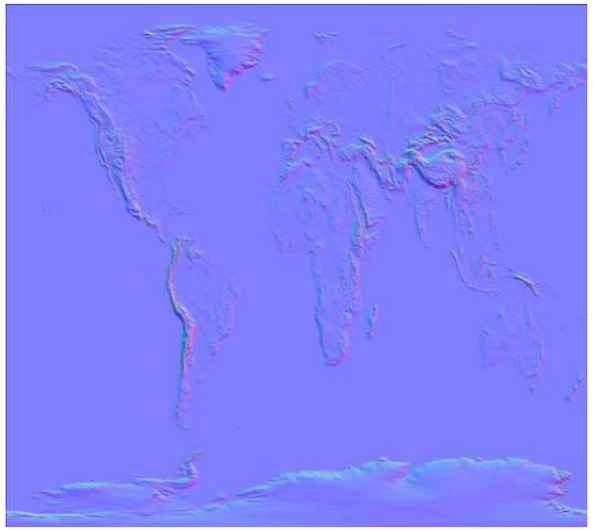




- Beispiele:

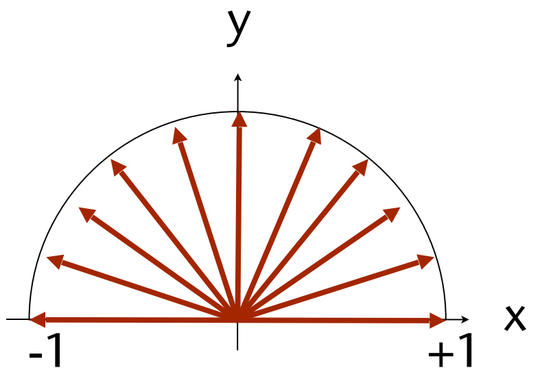


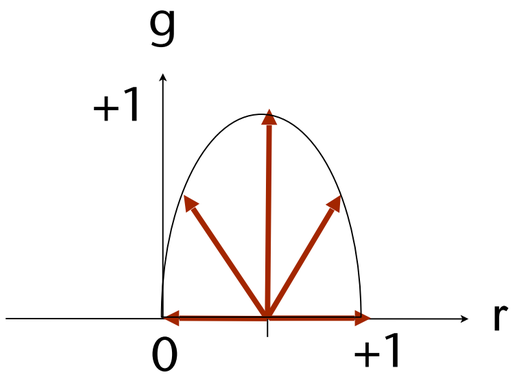




- Kodierung der Normalen:

$$\frac{1}{2}(\mathbf{n} + (1, 0, 1))$$



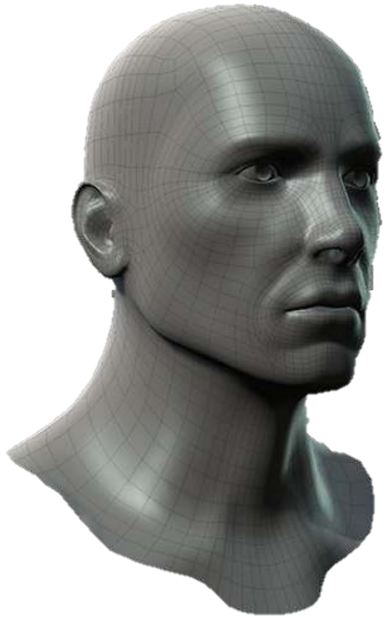


Unterschied zwischen Normal Maps und Bump Maps

- Bump Maps sind unabhängig von der Geometrie, man kann sie auf jede beliebige (genügend "flache") Geometrie aufbringen.
- Normal Maps kann (meist) man nur für genau die Geometrie verwenden, für die sie erzeugt wurden.
- Bump Maps benötigen nur zwei Farbkanäle
- Verstärken/reduzieren/invertieren der Bumps ist trivial (einfach die Kanäle mit einem Faktor skalieren)

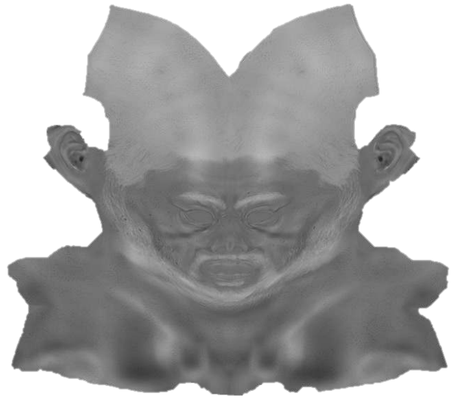
Beispiel für eine Kombination der Textur-Arten

Geometry
(2 mio pgons)



Diffuse Map
(baseColor)

Gloss Map
(specularity map)



Bump Map

Exkurs: Trompe l'œil - Malerei

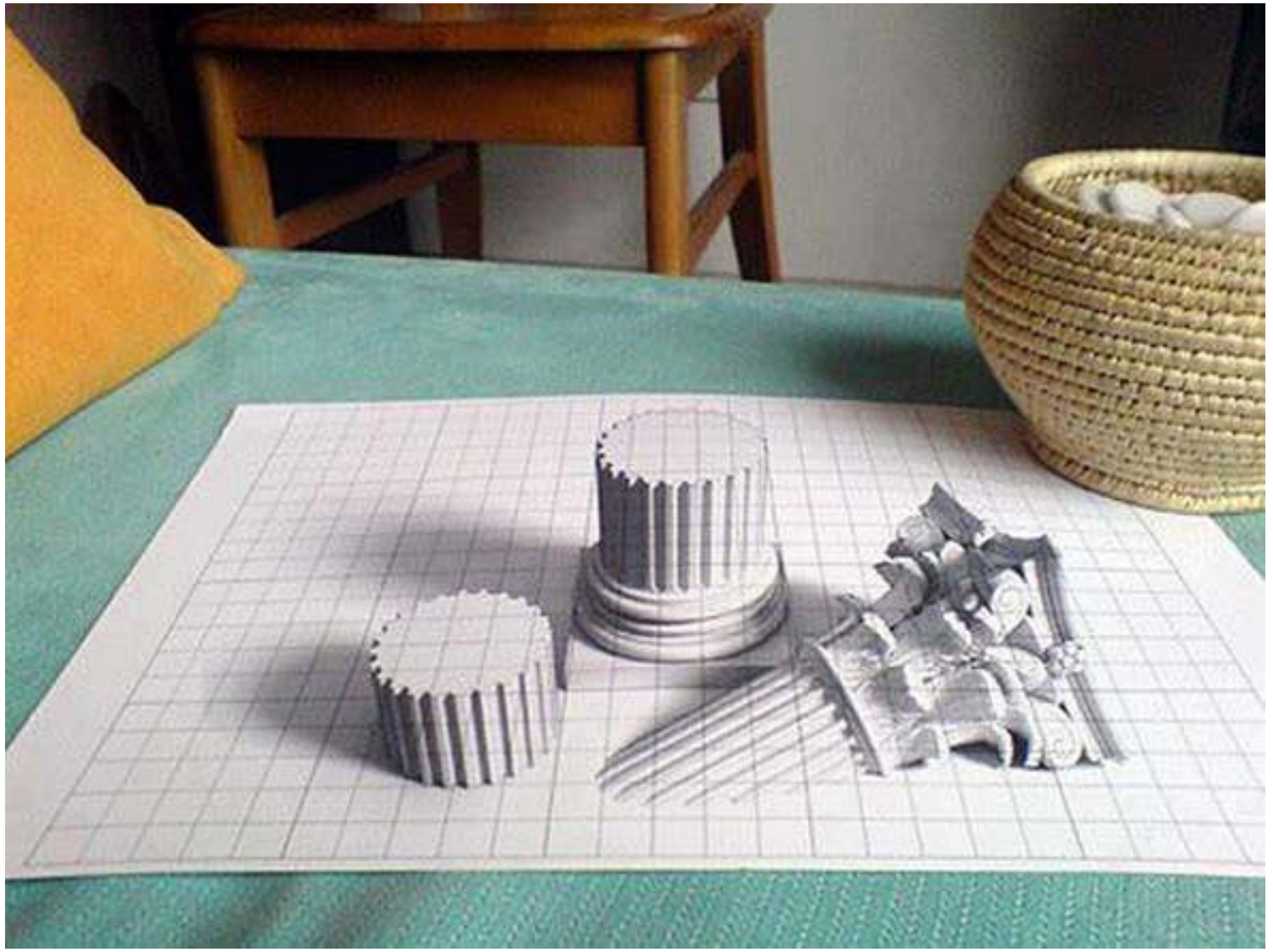
FYI

Pere Borrell del Caso,
Der Kritik entfliehend, 1874

FYI

Universitätskirche in Wien
mit trompe-l'œil Deckenfresken,
die den Eindruck einer Kuppel geben.
Gemalt von Andrea Pozzo
im 17. Jahrhundert

FYI

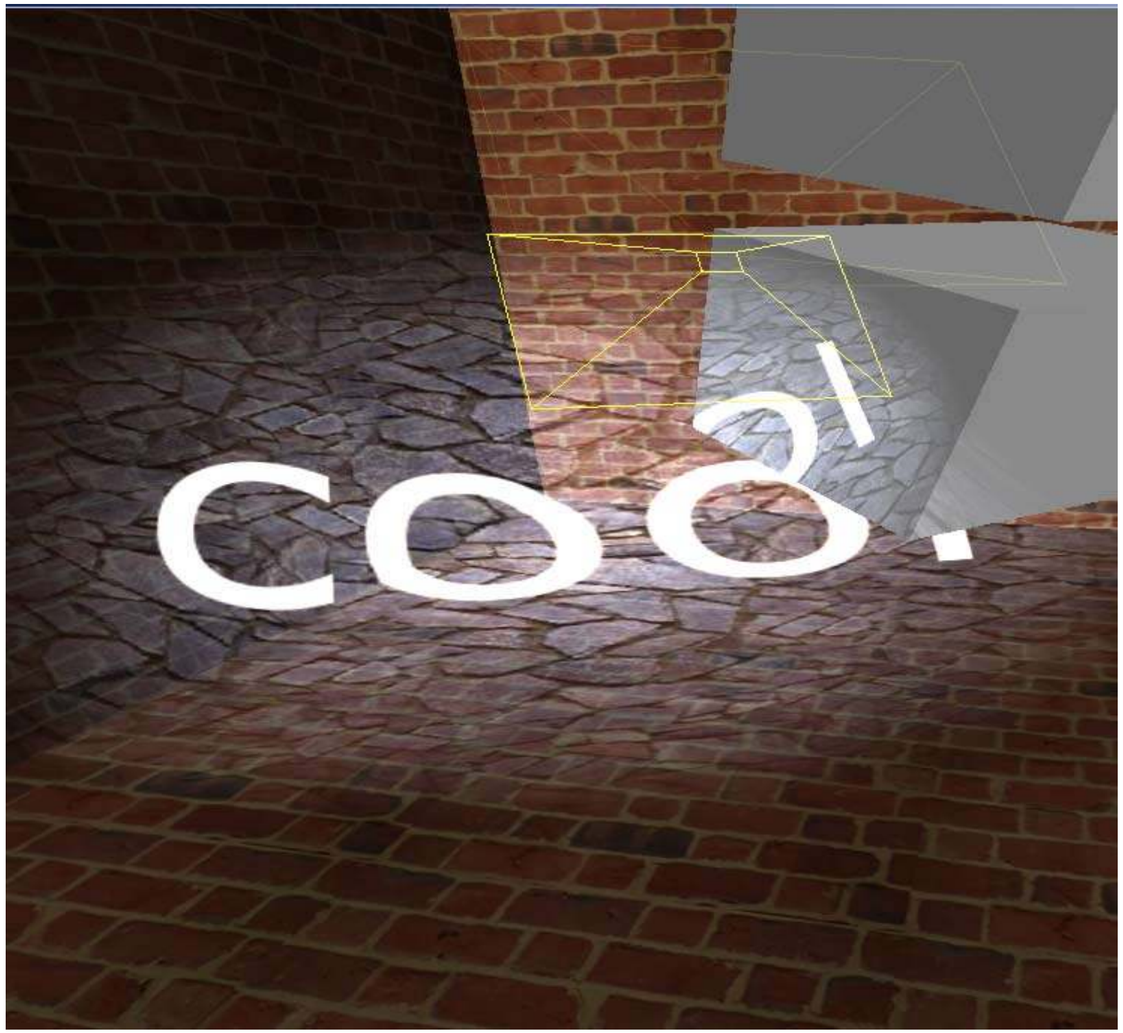


Modulation von Lichtquellenparametern

FYI

- Idee: Lichtquellenparameter durch Texturen zu beeinflussen.
- Besonders anschaulich ist dies bei Projektorlichtquellen. Dabei greift man mittels des Lichtvektors in die Textur und moduliert damit die Lichtemission:

$$L_i' = C_{tex}(f(\boldsymbol{l}_i)) \cdot L_i$$



Beispiel-Anwendung: virtuelle Taschenlampe

FYI

Texturen in Open GL FYI (nicht klausurrelevant)

- Als erstes muss eine Textur auf die Graphikkarte geladen werden:

```
glTexImage{1,2}D( target, level, internal, width,
                  [height,] border, format, type, data )
```

target = **GL_TEXTURE_1D, GL_TEXTURE_2D,** ...

level = 0 bzw. der zu definierende MipMap Level (später)

internal = Anzahl der Komponenten der Textur: 1, 2, 3, 4, **GL_RGB, GL_LUMINANCE, GL_R3_G3_B2**...

width & **height** **muß** = 2^n+2*border sein!

(gluScaleImage() kann Bilder skalieren helfen)

border = Breite des Randes, 0 oder 1

format = was steht pro Pixel im Speicher: **GL_RGB**, **GL_RGBA**, **GL_BGR**, ...

type = Typ der Pixel: **GL_UNSIGNED_BYTE**, **GL_FLOAT**, ...

data = Adresse der Pixeldaten im Hauptspeicher

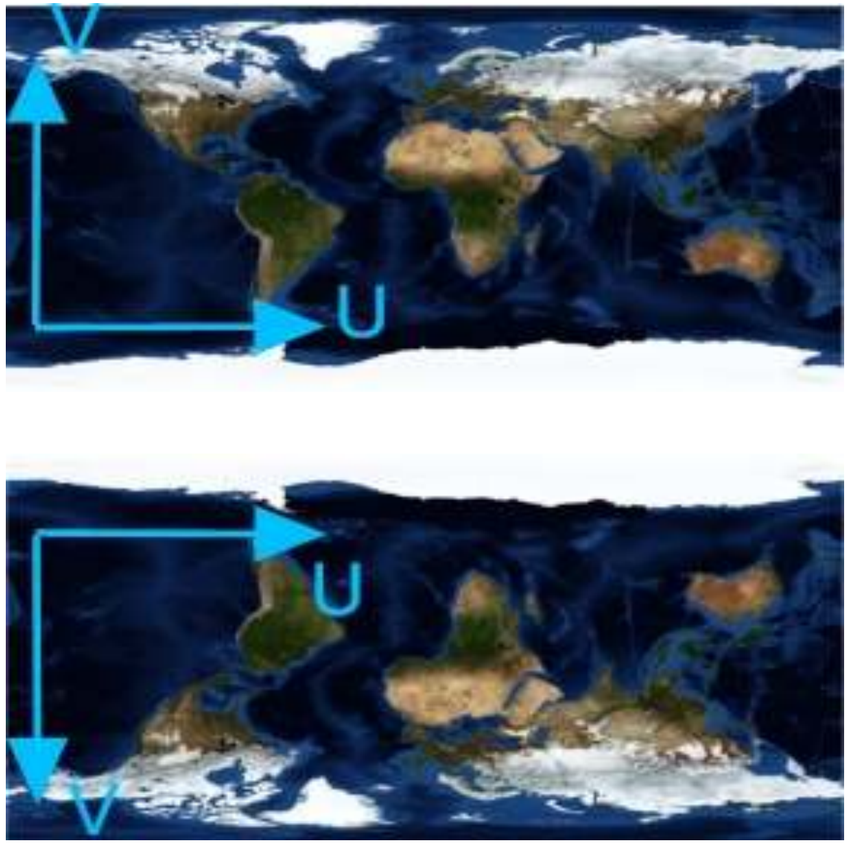
FYI (nicht klausurrelevant)

- Zu jedem Eckpunkt gehören u,v-Texturkoordinaten;
  diese werden, wie alle anderen Vertex-Attribute, im VBO abgelegt und durch die passende Verknüpfung an "in"-Variablen im Shader übergeben
- Achtung: OpenGL hat keinen Image-Loader!
  - Aber: Qt, imgui (u.ä. Libs) bieten hierfür Funktionen an
  - Oder: `glCopyTexImage2D(…)` liest Bild aus Framebuffer in Texturspeicher

Orientierung

FYI (nicht klausurrelevant)

- Der Fluch der Orientierung:
  - OpenGL Orientierung
  - Orientierung des Bild-Arrays nach dem Laden



- Achtung: Qt's **`bindTexture`** spiegelt das Bild, bevor es zur Graphikarte geschickt wird! Evtl. besser "von Hand" binden ...

Textur-IDs

FYI (nicht klausurrelevant)

- Während des Renderings einer Szene benötigt man viele verschiedene Texturen
- Jedesmal extra auf die GPU hochladen (**`glTexImage2D`**) ist ineffizient
- Lösung: alle Texturen gleichzeitig auf der Karte halten
- IDs generieren:

```
glGenTextures( GLint n, GLuint * indices )
```

findet **`n`** unbenutzte Textur-IDs und legt sie im Array **`indices`** ab

- Umschalten der aktuell aktiven Textur:

```
glBindTexture( GL_TEXTURE_{12}D, GLuint id )
```

- Achtung: **dadurch werden alle Textur-relevanten Teile des Zustandes umgeschaltet!**

FYI (nicht klausurrelevant)

- Zusammen:

```
unsigned int tex[N];
glGenTextures( N, tex );
glBindTexture( GL_TEXTURE_2D, tex[0] );
pixels = loadImage(…);
glTexImage2D( GL_TEXTURE2D,
      0,                              // mipmap level
      3,                              // components [1,2,3,4]
      width, height, border,
      format,                          // of the pixel data (GL_RGB..)
      type,                            // GL_FLOAT...
      pixels );                        // the pixel data
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP );
...                                    // more params (e.g. glTexEnv)
glBindTexture( GL_TEXTURE_2D, tex[1] );
pixels = loadImage(...);
glTexImage2D( GL_TEXTURE2D, ...);
```

Texturierung mittels Shadern

FYI (nicht klausurrelevant)

- Im Vertex-Shader werden die Texturkoordinaten im Wesentlichen einfach nur "durchgereicht" (man könnte sie dort auch erzeugen)

- Im Fragment-Shader geschieht dann die eigtl Texturierung:

```
uniform sampler2D myTexture;
void main( void )
{
   fragmentColor = texture2D( textureImage, myTexCoord.st );
}
```

FYI (nicht klausurrelevant)

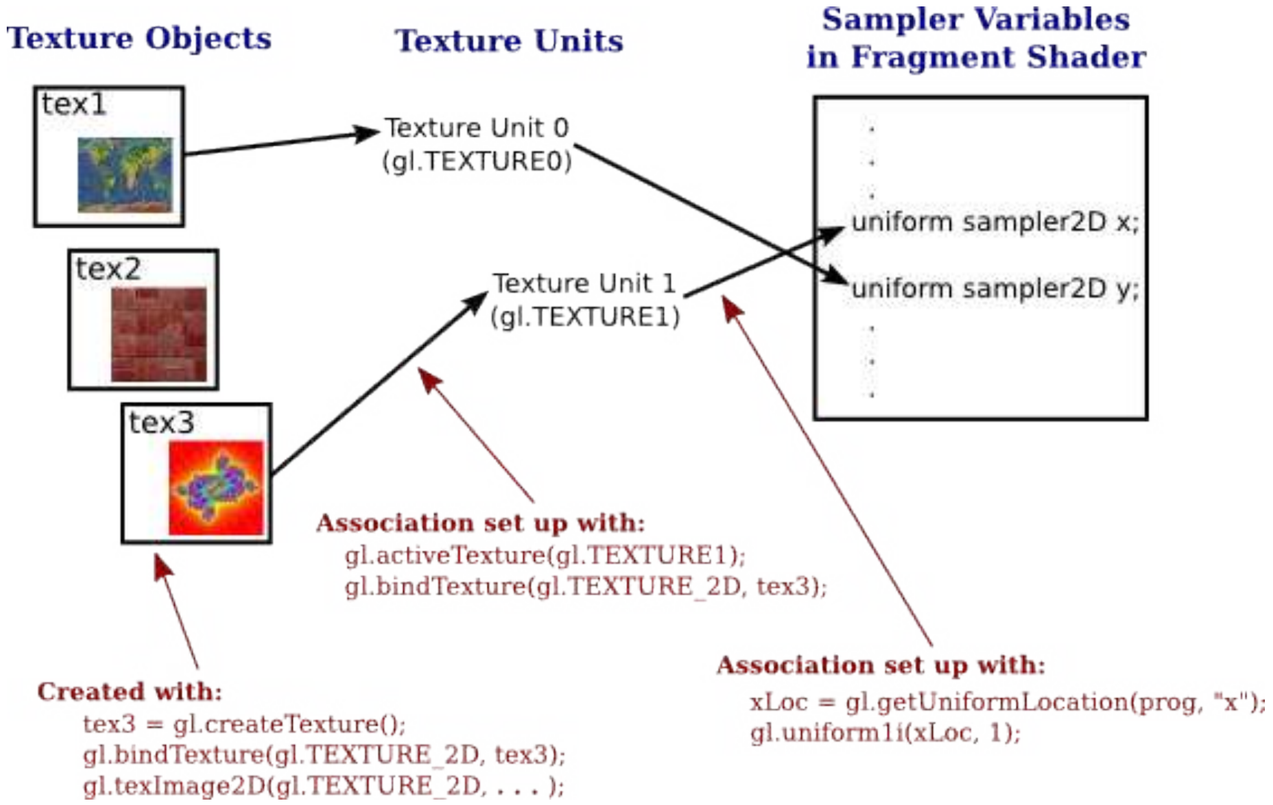
- Bevor ein solcher Shader benutzt werden kann, muss die Uniform-Variable myTexture noch an die richtige Textur "gebunden" werden"

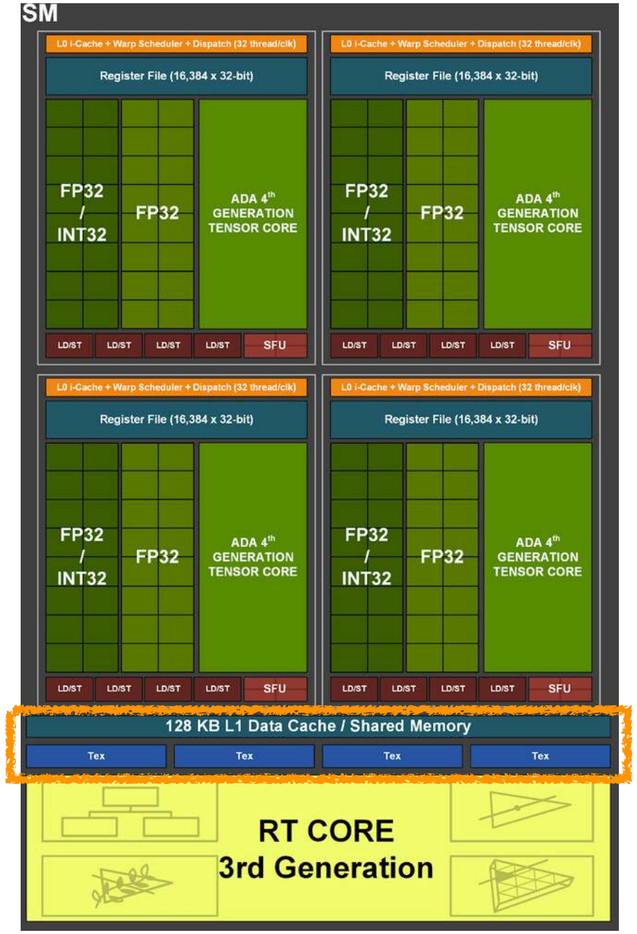
```
// Select the texture unit; all subsequent texture functions
// (e.g., like glBindtexture) will work on this texture unit
glActiveTexture(GL_TEXTURE0);
// Bind the texture object with the ID tex[0] (from glGenTexture)
glBindTexture( GL_TEXTURE_2D, tex[0] );

// Find "myTexture" uniform var of type sampler2D in shader
int myTextureLoc = glGetUniformLocation(shaderProgram, "myTexture");

// Set the texture unit to use by the sampler2D variable
glUniform1i( myTextureLoc, 0 );
```

Texture units have a real counter-part in the hardware

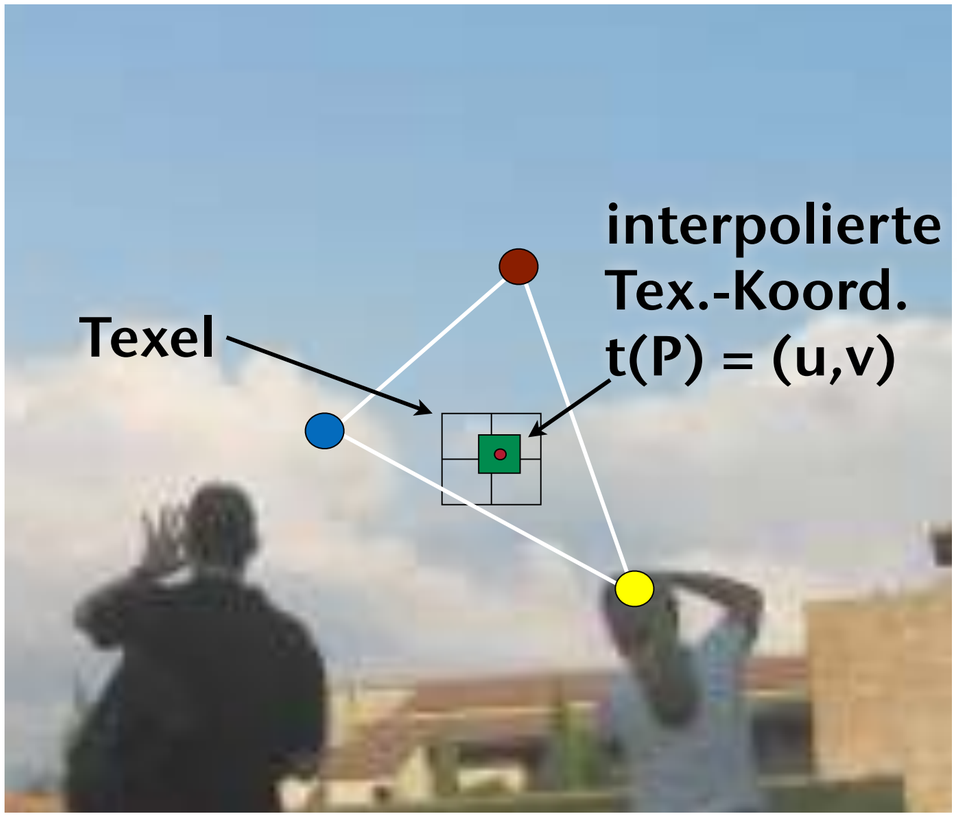




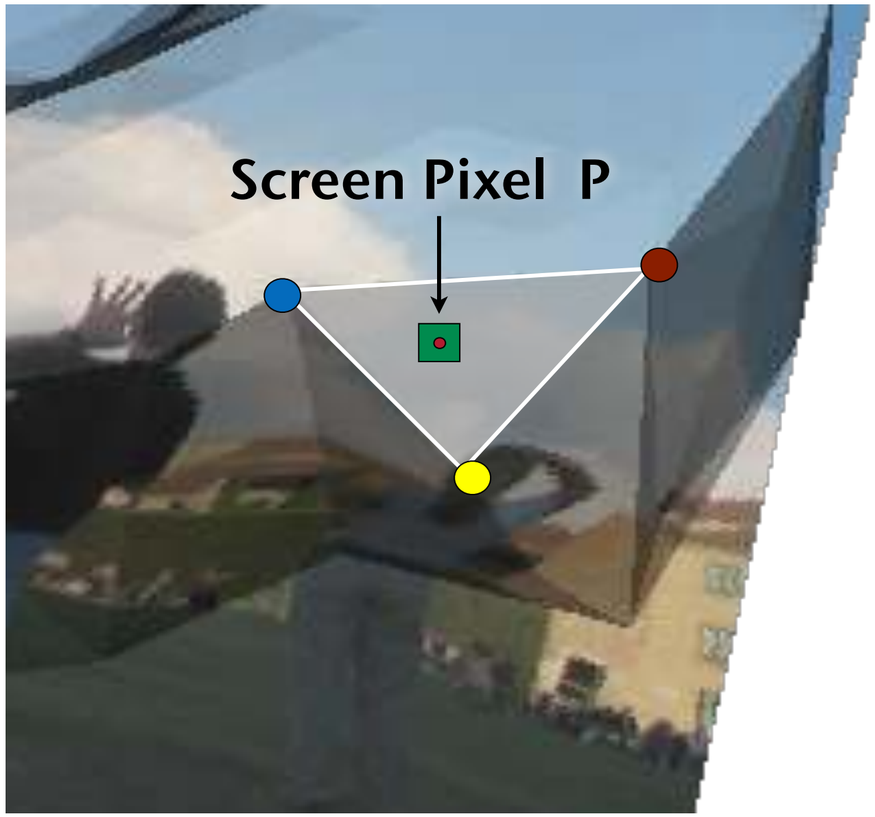
RTX 4090

Textur-Interpolation

Texture space



Screen space



Rekonstruktionsmethoden

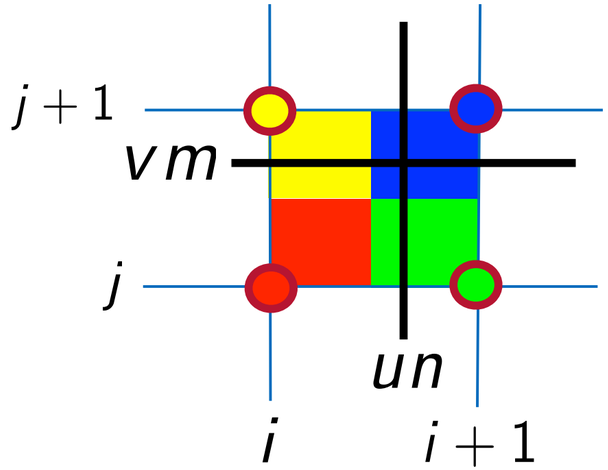
- Textur = $m \times n$ Array von Texeln,

```
vec3f texture[n][m];
```

$$t(P) = (u, v) \in [0,1] \times [0,1]$$

1. Nearest neighbour (Punktfilter):

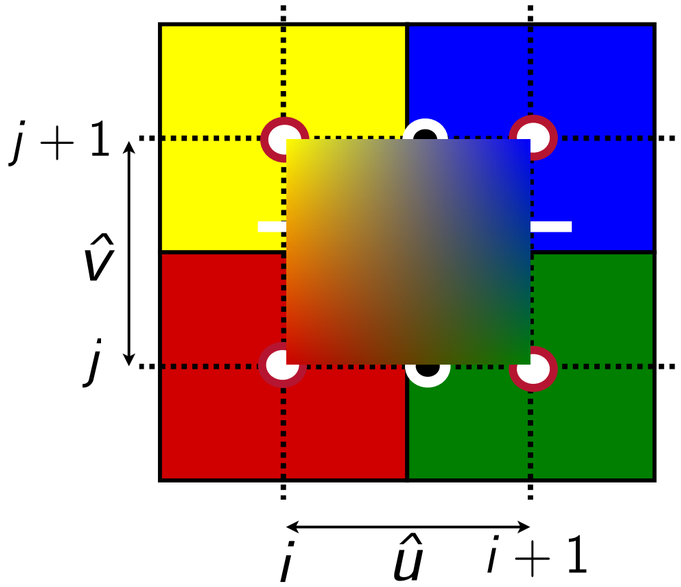
$$C_{\text{tex}} = \text{texture}\big[[un], [vm]\big]$$



2. Bilineare Interpolation:

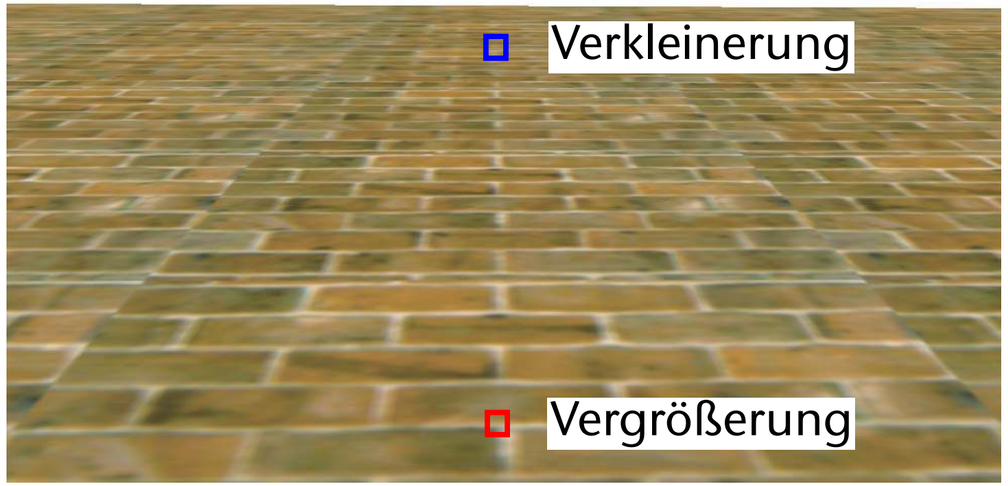
$$\hat{u} = un - \lfloor un \rfloor \ , \ \hat{v} = vm - \lfloor vm \rfloor$$

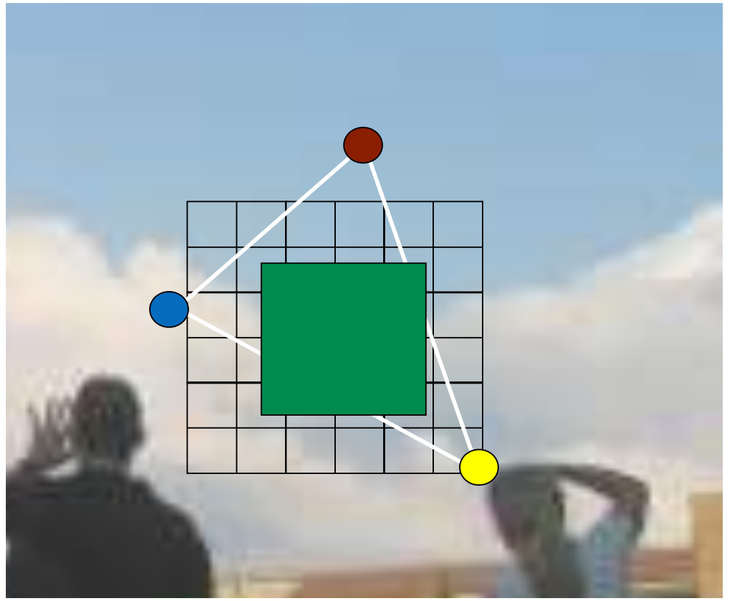
$$c = (1-\hat{u})\big((1-\hat{v})\blacksquare + \hat{v}\,\blacksquare\big) +$$
$$\hat{u}\big((1-\hat{v})\blacksquare + \hat{v}\,\blacksquare\big)$$



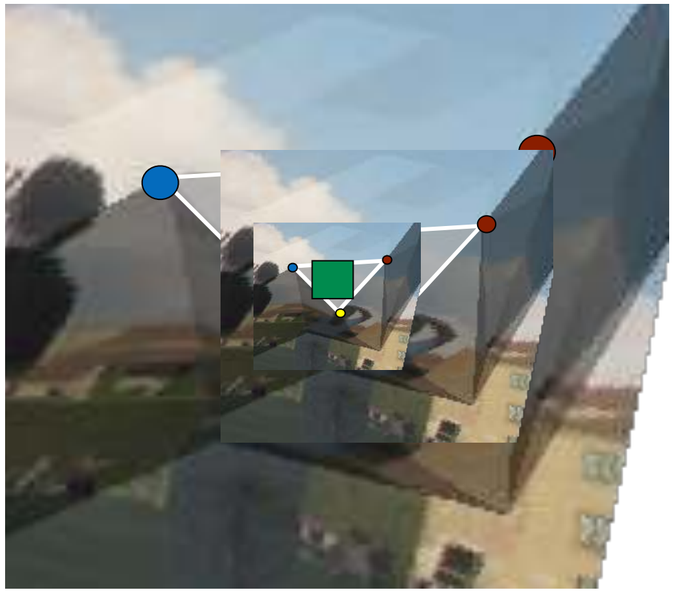
Texturverkleinerung

- Bilineare Interpolation ist OK, wenn Pixelgröße ≤ Texelgröße
  - Wir sind rel. dicht am Polygon dran
  - Ein Texel überdeckt ein oder mehrere Pixel
- Was passiert, wenn man vom Polygon "weg-zoomt"?





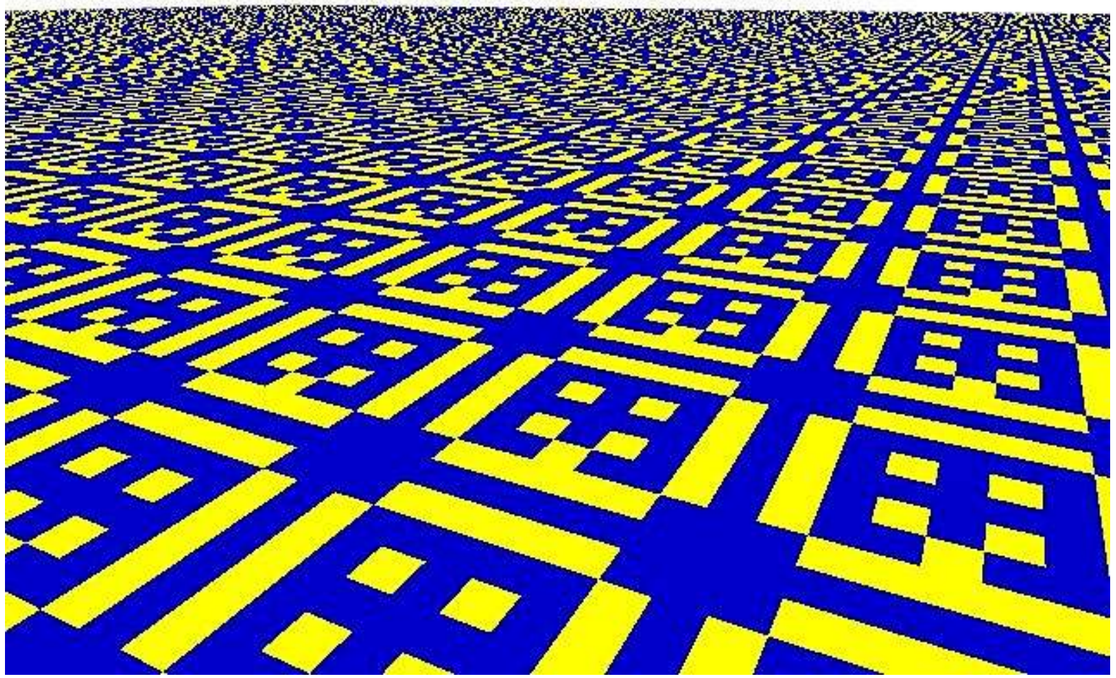
Texture space



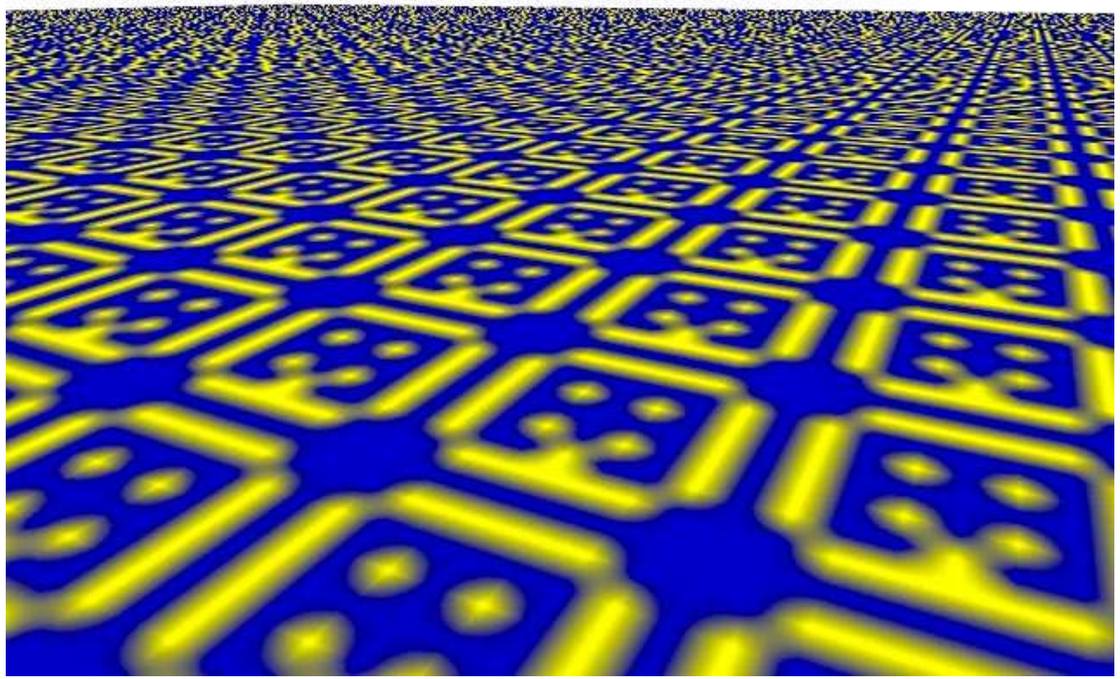
Screen space

- Ein nicht ganz triviales Problem:

Einfacher Punktfilter ⟶ Aliasing



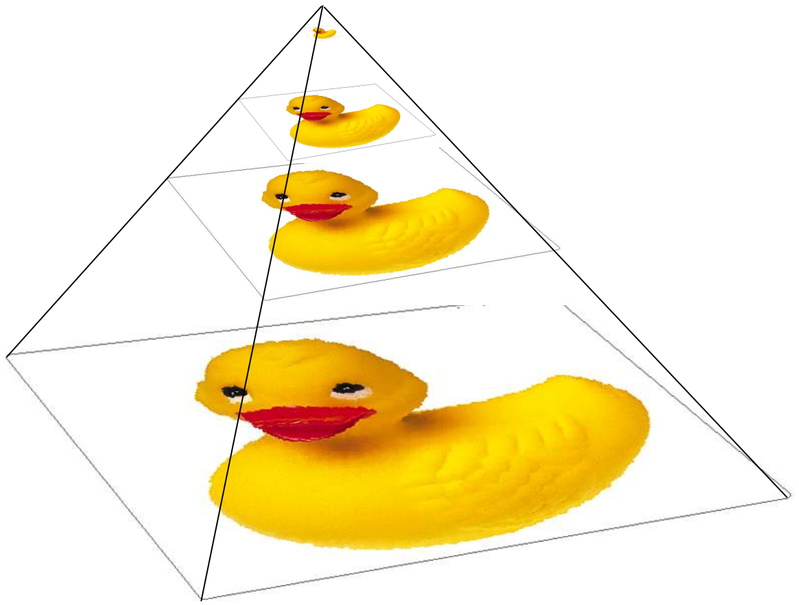
Bilineare Interpolation hilft nur wenig

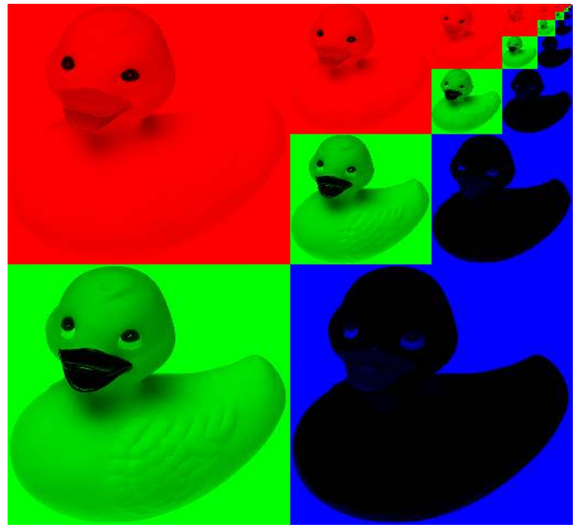


- Bei starker Verkleinerung müsste eigentlich ein Mittelwert von vielen Texeln gebildet werden, da sie alle auf dasselbe Pixel auf dem Bildschirm abgebildet werden
- Für Echtzeitanwendungen ist das zu aufwendig
- Lösung: Preprocessing
  - Vor dem Start verkleinerte Versionen der Textur anlegen, in der die Texel schon gemittelt sind
  - Wenn jetzt viele Texel auf einen Bildschirmpixel abgebildet werden, wird die beste passende Verkleinerung verwendet anstatt der Originaltextur
- MIP-Maps (lat. "multum in parvo" = Vieles im Kleinen")

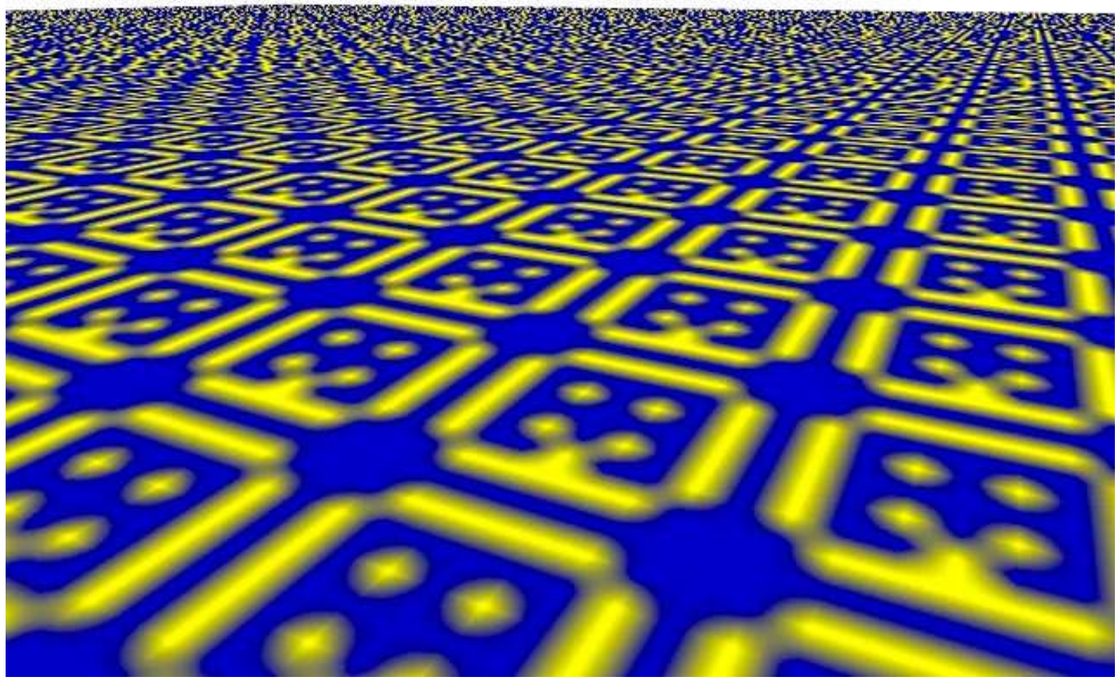
MIP-Maps

- Eine MIP-Map ist eine Bild-Pyramide:
  - Jeder Level entsteht aus dem darunter durch Zusammenfassen mehrerer Pixel und hat nur die Größe 1/4
  - Daher: orig. Bild muß 2^nx2^n groß sein!
  - Einfachste Art der Zusammenfassung: 2x2 Pixel mitteln
  - Oder: irgend einen anderen Bild-Filter anwenden
- Intern wird ein 2^n-Bild in einem 2^{n+1}-Bild gespeichert
- MIP-Map hat Speicherbedarf 1.3x Original

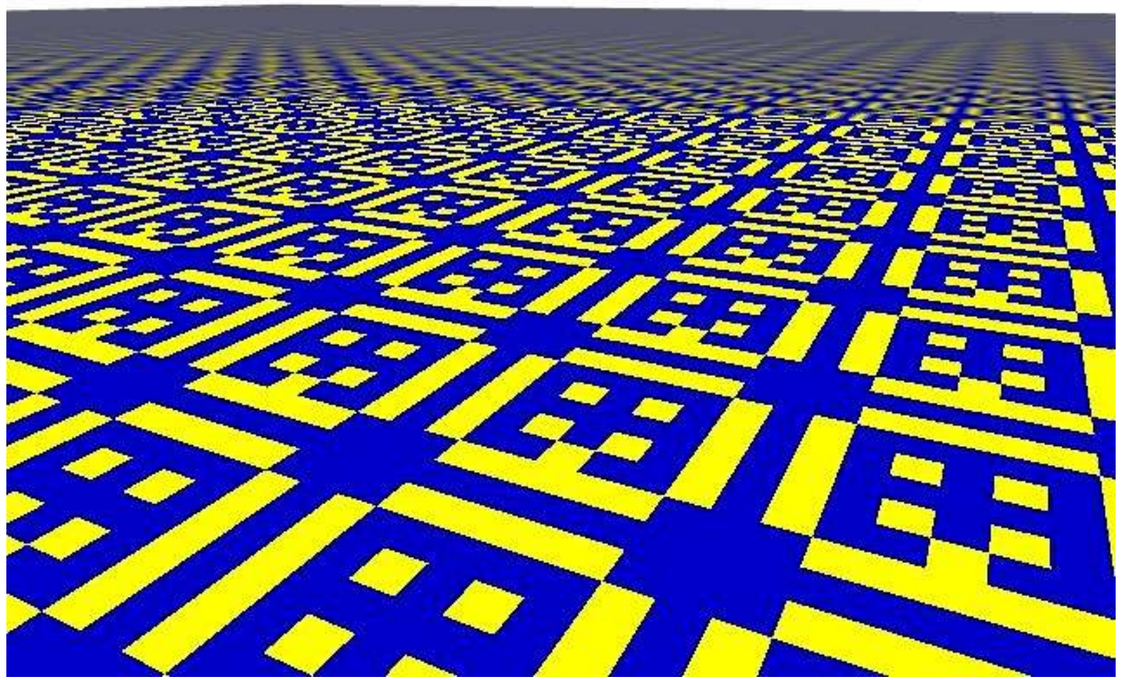




- Abhängig von der Distanz des Betrachters zum Fragment (in camera space) wird entschieden, welcher Texturlevel sinnvoll ist (pro Fragment)
- Der ideale Level ist der, bei dem 1 Texel auf 1 Pixel abgebildet wird



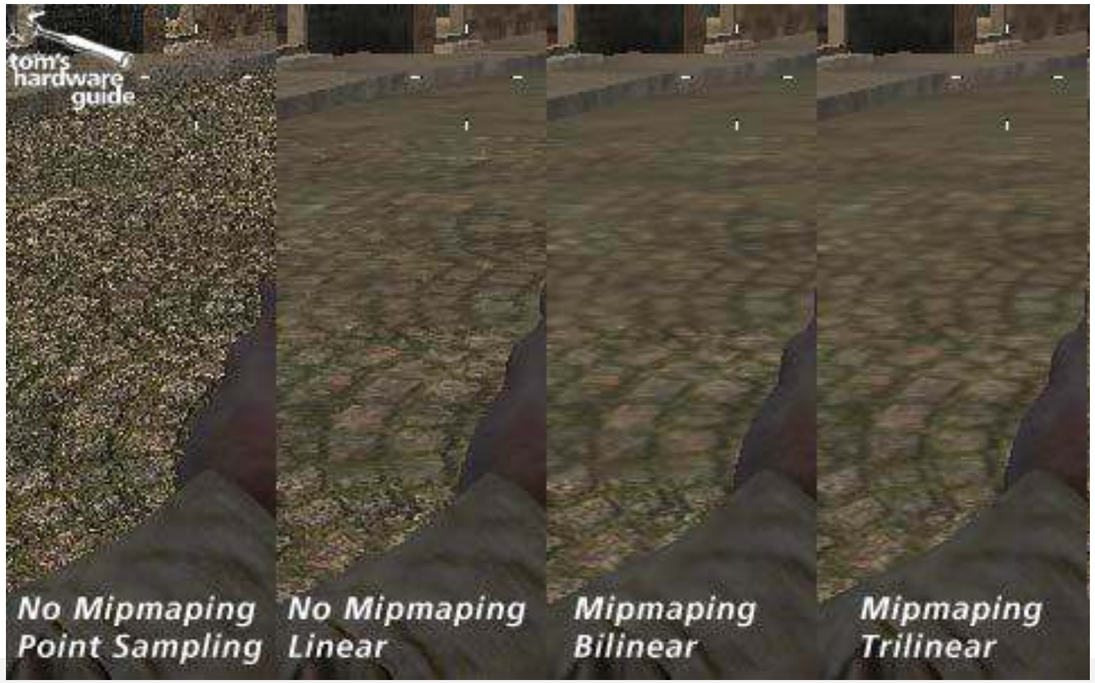
bilinear gefiltert



MIP-Map

Trilineare Filterung

- Szene mit nur bilinear gefilterten MIPmaps ⟶ Banding
- Ursache: in den meisten Fällen fällt ein Pixel "zwischen" zwei MIPmap Levels, d.h., bzgl. der Texel des unteren Levels ist das Pixel zu klein, bzgl. dem oberen Level zu groß
- Lösung: trilineare Filterung, d.h., interpoliere zwischen den 4+4 Texeln des oberen und des unteren Levels, zwischen denen das Pixel (im uv-Raum) liegt



Filterspezifikation in OpenGL

FYI (nicht klausurrelevant)

- Magnification:

```
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_MAG_FILTER , param )
```

  - *param* = **GL_NEAREST**: Punktfilter

    = **GL_LINEAR**: bilineare Interpolation

- Minification:

```
glTexParameteri( GL_TEXTURE_2D,
                 GL_TEXTURE_MIN_FILTER , param )
```

  - *param* wie bei Magnification, aber zusätzlich

GL_NEAREST_MIPMAP_NEAREST: wähle "näheste" Mipmap, und daraus nähestes Texel

GL_LINEAR_MIPMAP_LINEAR: wähle die beiden nähesten Mipmap-Levels, dazwischen trilineare Interpolation

Mipmaps in OpenGL

FYI (nicht klausurrelevant)

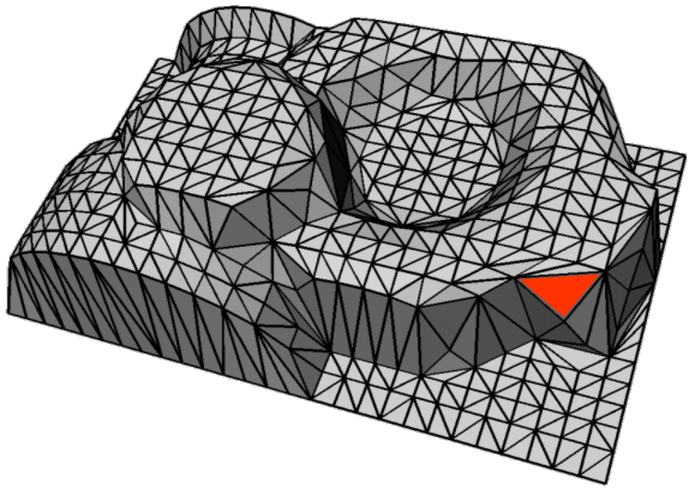
- Der **`level`** Parameter von **`glTexImage2D`** bestimmt, welcher Level der Mipmap gesetzt wird
- 0 ist die größte Map, jede weitere hat dann halbe Größe, bis hin zu 1x1
- Alle Größen müssen vorhanden sein
- Hilfsfunktion:

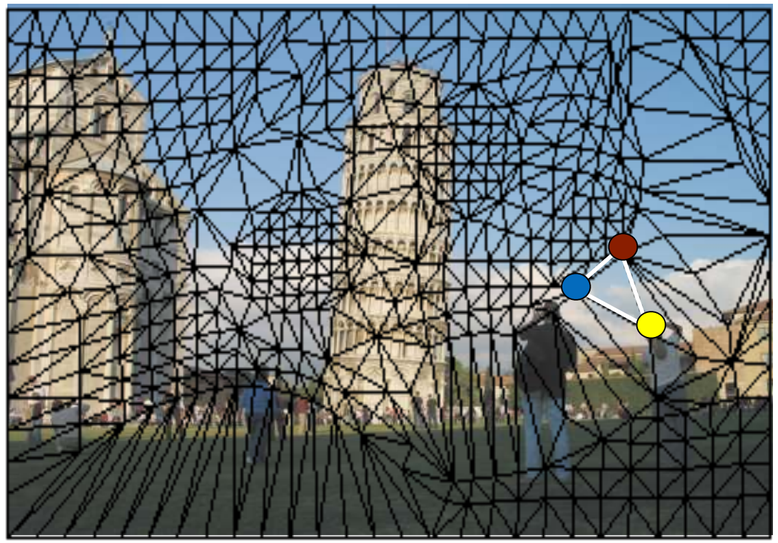
```
gluBuild{12}DMipmaps( target, components,
        width, [height,] format, type, data )
```

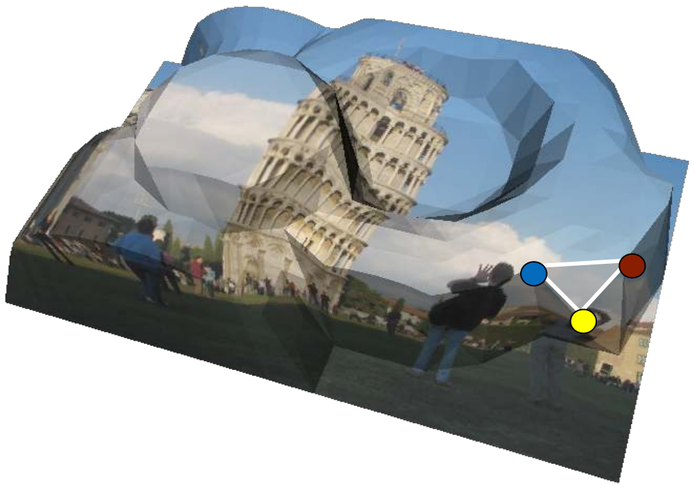
mit Parametern wie **`glTexImage{12}D()`**

Einfache Methoden zur Parametrisierung

- Parametrisierung = Methode zur Berechnung von uv-Koordinaten *pro Vertex*
- Triviale Parametrisierung im Falle eines Terrains:
  - 3D-Koordinaten nach unten projizieren
  - Achtung: dies ist nicht notwendig eine "gute" Texturierung!

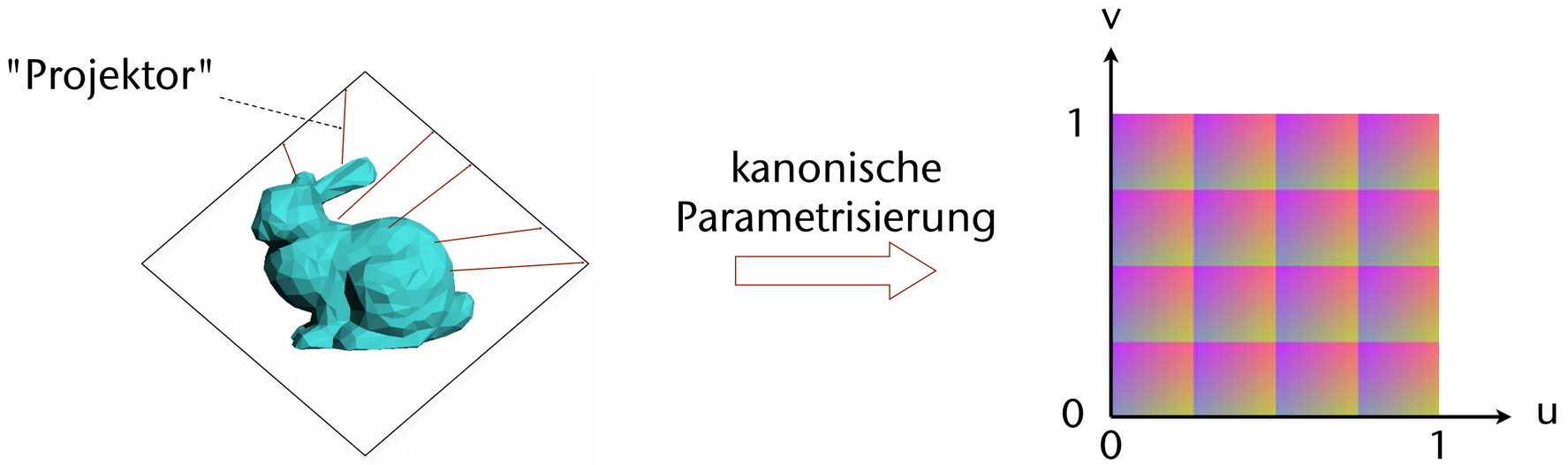






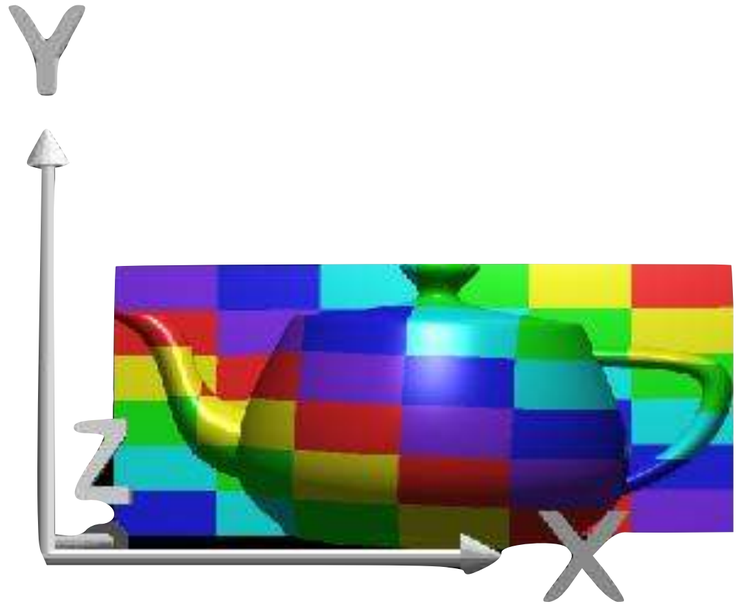
Einfache Parametrisierung mittels Hüllkörper

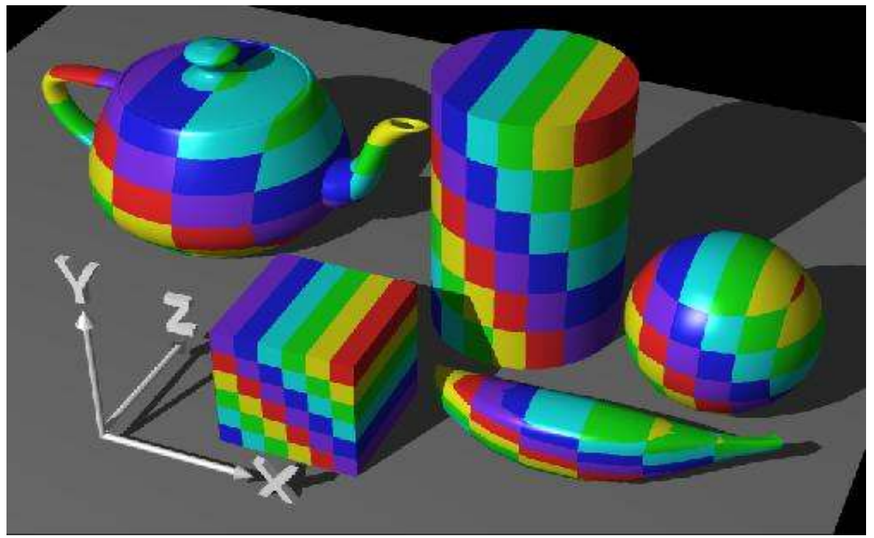
- Einfache Idee: ein 2-stufiger Prozess
  - Lege (konzeptionell) einen "kanonisch" parametrisierbaren Hüllkörper um das ganze Objekt
  1. Projiziere Vertices auf diesen Hüllkörper
  2. Verwende die *uv*-Koordinaten des projizierten Punktes auf dem Hüllkörper



Einige Hüllkörper und deren Parametrisierung

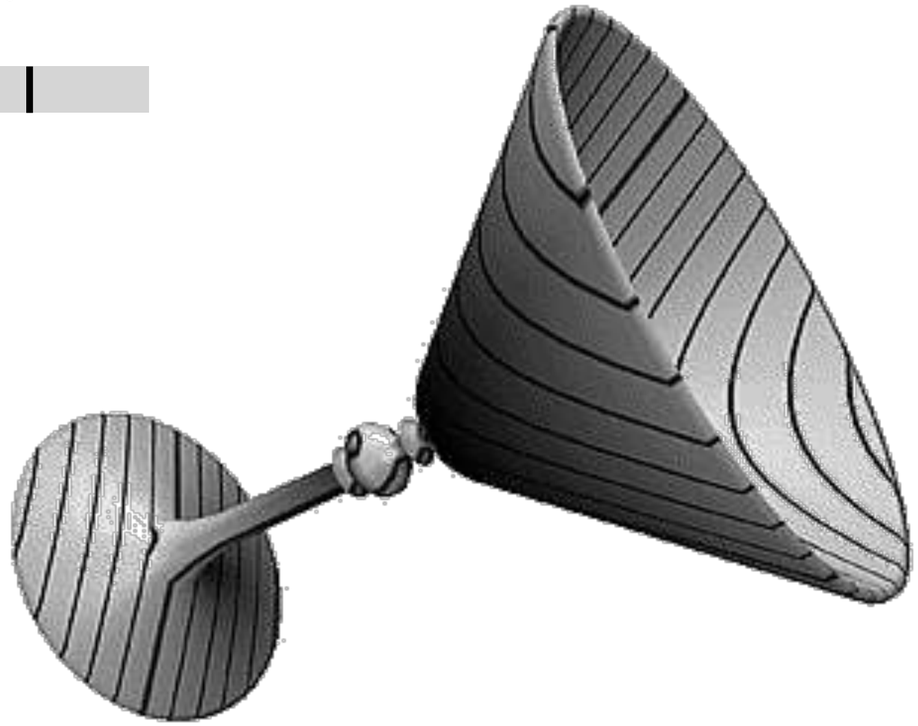
- Ebene:
  - Projiziere Punkt (x,y,z) auf eine Ebene
    $\rightarrow (x,y)$
  - $(u,v) = (s_x x + t_x, s_y y + t_y)$
- Verallgemeinerung:
  - Definiere 2 beliebige Ebenen E_1 und E_2
  - u := dist(P, E_1)
    v := dist(P, E_2)
  - Dieses Feature bietet OpenGL





Beispiel

- Erzeuge Höhenlinien mittels dieser Technik:
  - Verwende 1D-Textur 
  - u := dist(P, E_1)

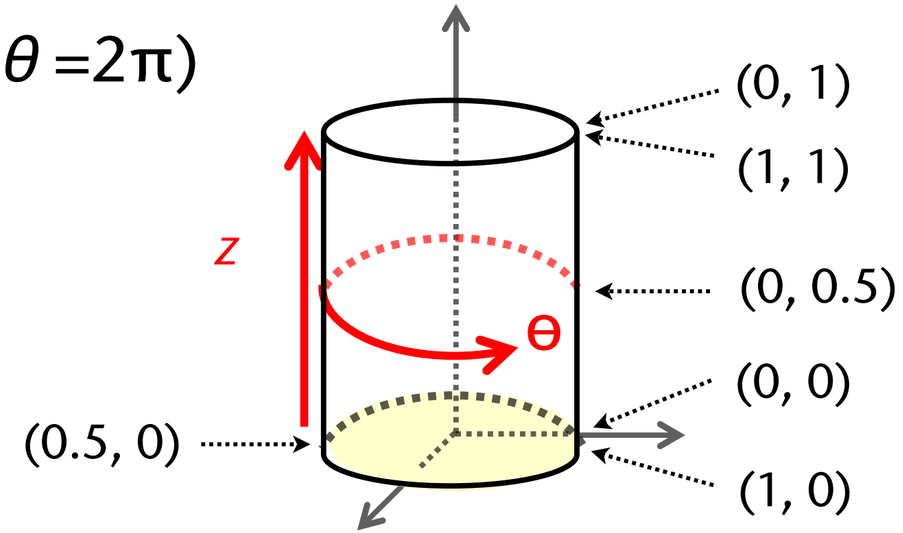


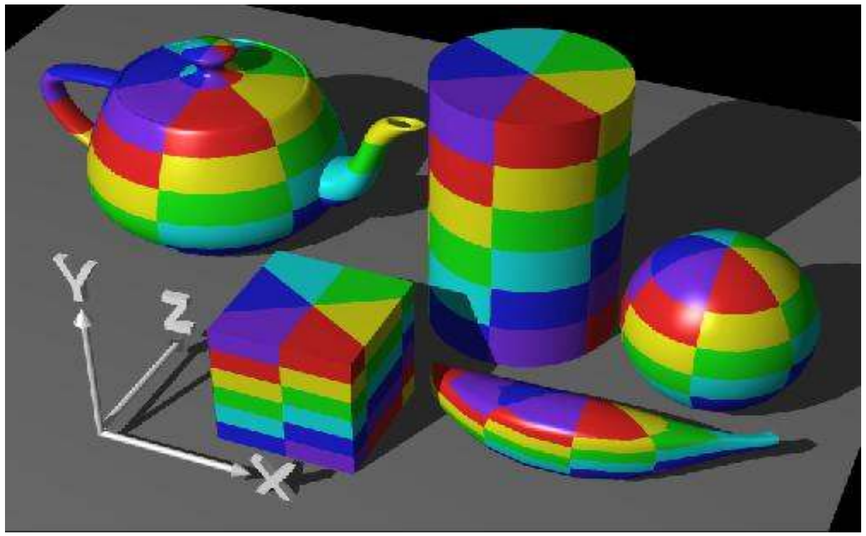
- Viele weitere ungewöhnliche Anwendungen von Texture-Mapping auf http://www.graficaobscura.com/texmap/index.html

- Zylinder-Parametrisierung:
  - Konvertiere kartesische Koord. (x,y,z) in zylindrische Koord. (entspricht Projektion auf Zylinder):

$$(r \sin \Theta, r \cos \Theta, z) \quad (u, v) = (\Theta / 2\pi, z)$$

- Beachte "Naht" bei ($\theta = 0$ bzw. $\theta = 2\pi$)



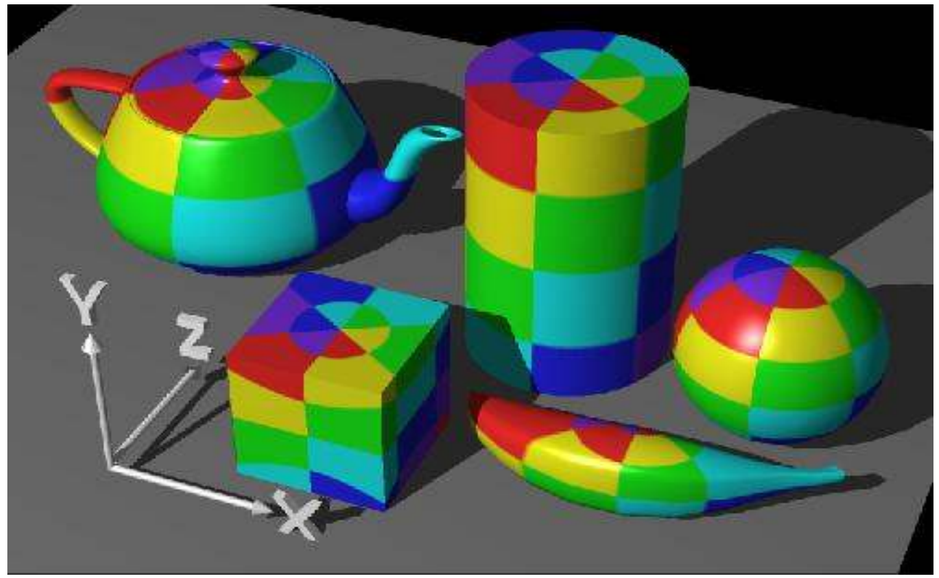


- Kugel-Parametrisierung:
  - Stelle Punkt in sphärischen Koordinaten dar:

$$r \cdot (\sin\theta \cos\phi, \cos\theta \cos\phi, \sin\phi)$$

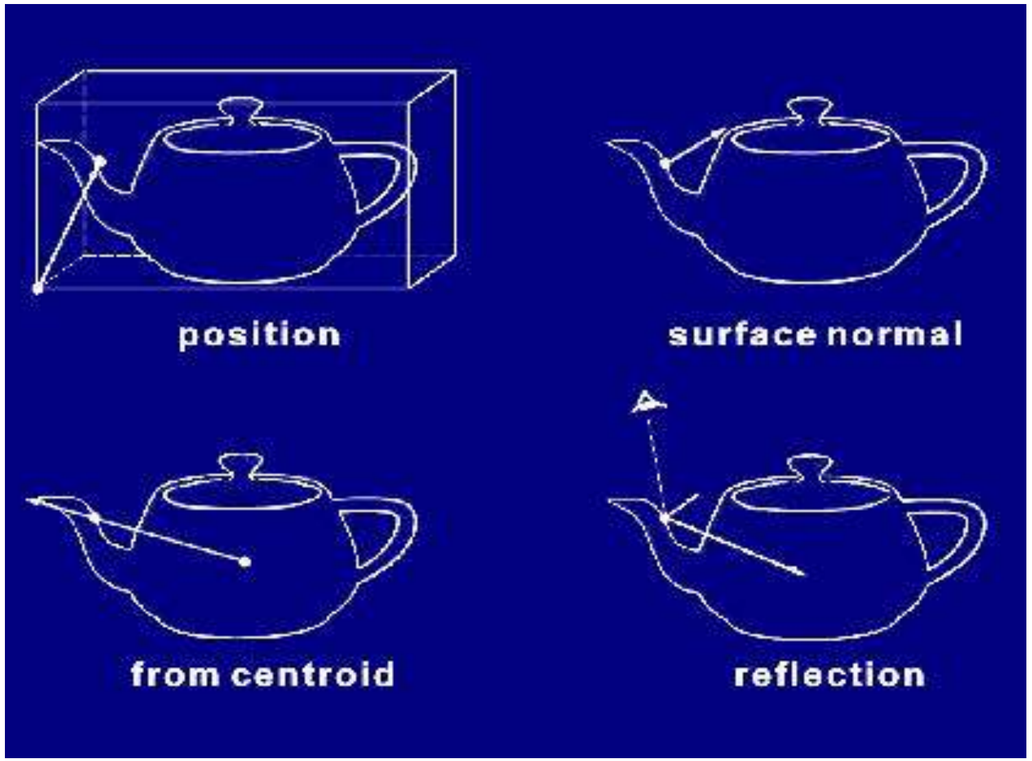
$$(u, v) = (\frac{\theta}{2\pi}, \frac{\phi}{\pi/2} + 1)$$

- Beachte: Singularität am Nord- und Südpol!



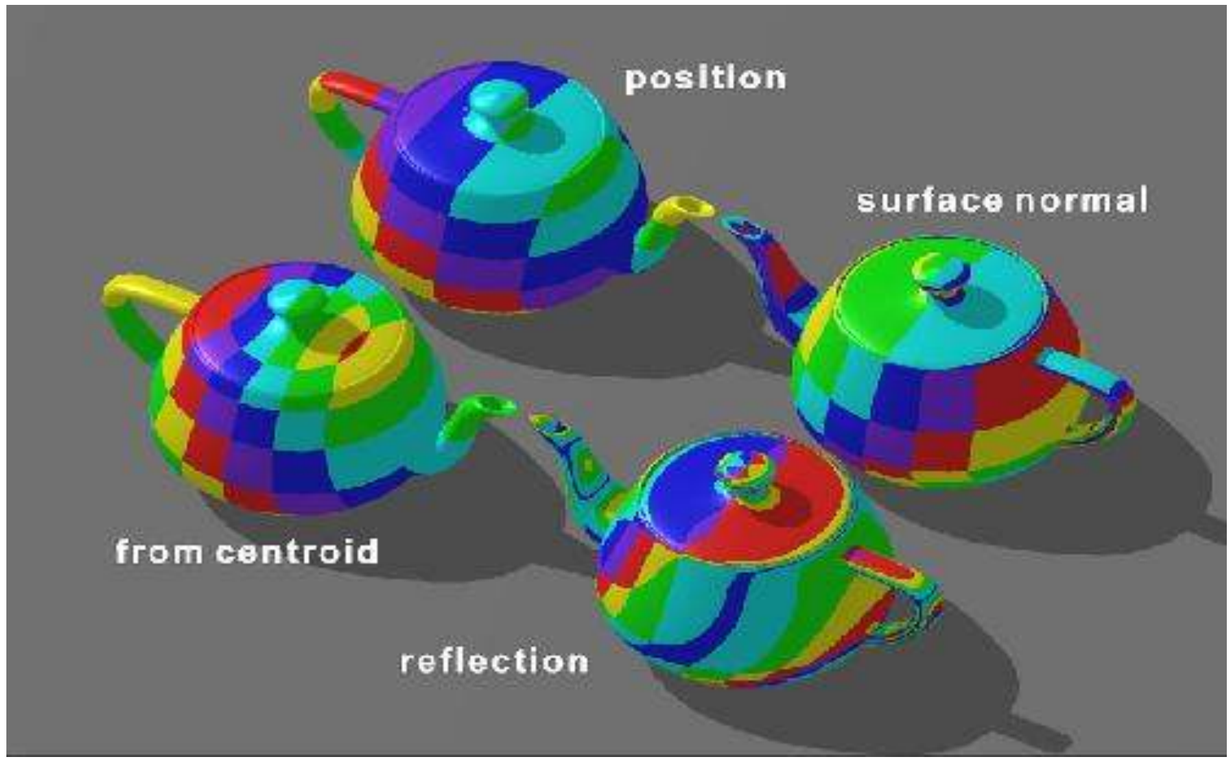
Welches Vertex-Atribut soll projiziert werden

- Bisher: einfach die Koordinaten (*x,y,z*) des Vertex auf den (gedachten) Hüllkörper projiziert
- Verallgemeinerung: statt dessen kann man genauso gut (oder schlecht) andere Attribute des Vertex projizieren, z.B.
  - Normale
  - Vektor vom Zentrum des Objektes durch den Vertex
  - Reflektierter Viewing-Vektor
  - ...



Beispiele

Projektion verschiedener Attribute auf eine Ebene



Projektion verschiedener Attribute auf einen Zylinder