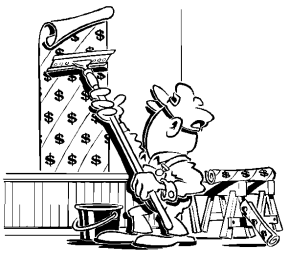
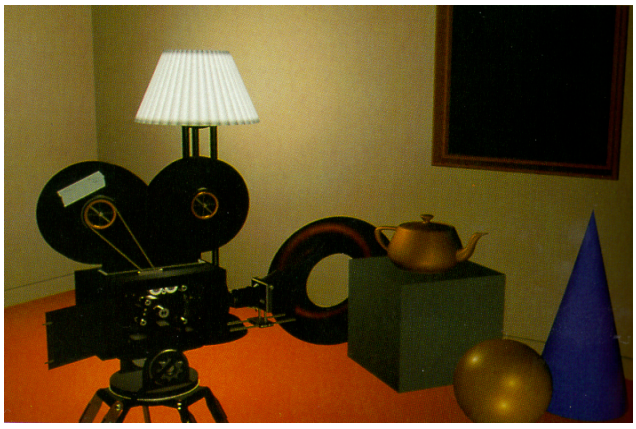
Computer-Graphik I
Texturierung



G. Zachmann
Clausthal University, Germany
cg.in.tu-clausthal.de

Motivation

- Was fehlt? ...



"Shutter bug", Pixar

- ... Oberflächendetails

"Shutter bug", 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 Flächen nachzubilden ist in der Regel viel zu aufwendig

Grundidee der Texturierung

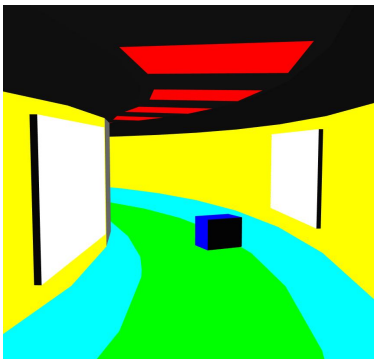
- Objekt mit Textur „tapezieren“

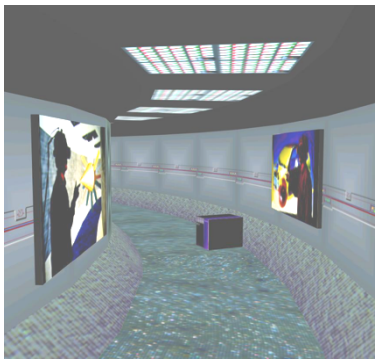
→Visuelles Detail trotz grober Geometrie

Objekt (Geometrie) Textur (Farbe)

- Ursprung: Catmull (1974), Blinn and Newell (1976), u.a.

Weitere Beispiele





- Kaustik durch Texturen verstärkt den Unterwassereindruck

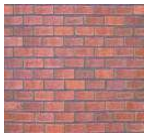
Übersicht

- Arten von Texturen: diskret oder prozedural
- Dimension der Texturen: 1D, 2D, 3D, 4D(?)
- Wichtige Punkte bei den diskreten 2D-Texturen:
  - Interpolation der Texturkoordinaten
  - Anwendung der Textur auf die Beleuchtung o. a. Oberflächeneigensch.
  - Parametrisierung der Fläche
  - Filterung
- Wie funktioniert es in OpenGL
- Environment-Mapping

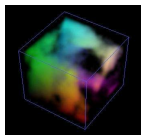
Texturarten

1D Texturen

2D Texturen



3D Texturen



Cubemap Texturen

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

Einfacher Fall: 3D-Texturen

- 3D-Texturen nennt man auch Festkörper-Texturen (z.B. Holz und Marmor) *("solid texture")*
- Die lokalen Koordinaten der Obj.oberfläche (*x,y,z*) indizieren direkt die Textur:

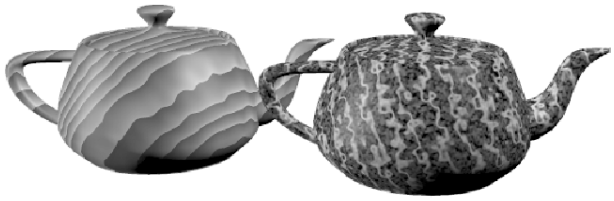
$$(r, g, b) = C_{\text{tex}}(x, y, z)$$

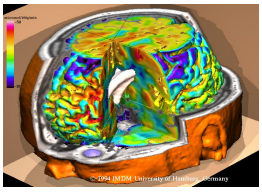
- Die Textur ist also an jedem Punkt im Raum definiert
- Das Objekt wird quasi aus dem Texturvolumen "herausgeschnitzt"

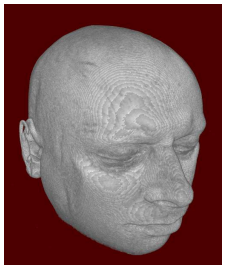
2D-Texturierung

3D-Texturierung

- Beispiele:



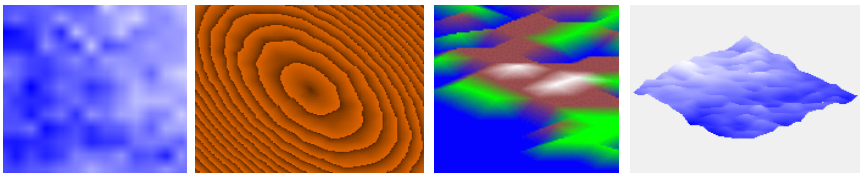




Diskrete und prozedurale Texturen

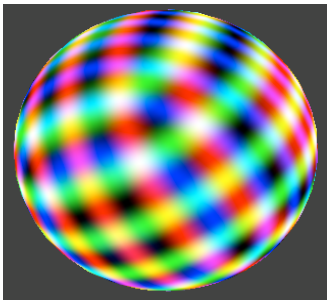
- Man unterscheidet diskrete und prozedurale Texturen
- Eine diskrete 3D-Textur = 3-dimensionales Array C[u,v,w]
  - C[u,v,w] = Vektor mit 3 Farbkomponenten, ein "Texel" (*texture element*)
  - Pro Pixel benötigt man 3 Texturkoordinaten (*u,v,w*) zum Indizieren in das Array
- Prozedurale Texturen werden bei jedem Auslesen aus math. Funktion oder fraktalem Algorithmus berechnet

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



- Einfaches Beispiel für eine prozedurale 3D-Textur:

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



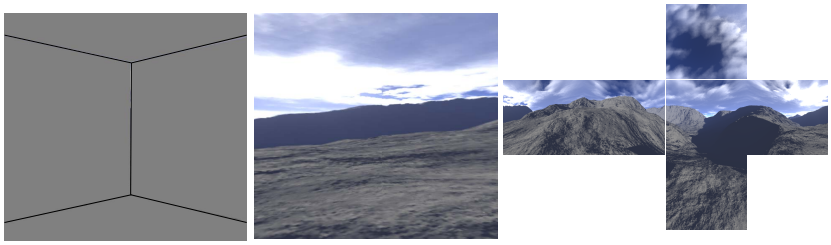
- Vorteile der prozeduralen Texturen:
  - Speicheraufwand ist minimal
  - Texturwerte können an jeder Stelle (u,v), bzw. (x,y,z) berechnet werden
  - Texturen sind im gesamten Raum definiert (kein Wrap-Around / Clamping)
  - Optimale Genauigkeit (kein Runden von Koord., keine Interpolation)
- Nachteile:
  - 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?)

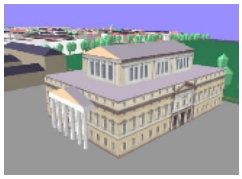
Diskrete 2D-Texturen

- 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
  - Verzerrung beim Mapping auf beliebige Fläche

Beispiel 1: *Skybox*

- Die Umgebung einer virtuellen Szenen modelliert man oft durch eine Kugel oder einen Würfel mit entsprechenden Texturen





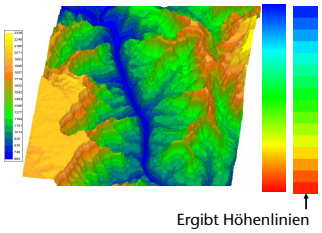
Ohne Skybox

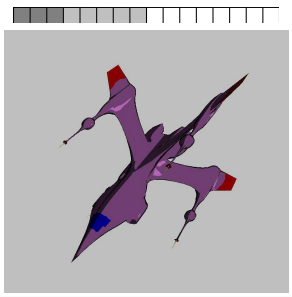
Die Skybox

Vom Boden aus

1D Texturen

- In der Visualisierung möchte man oft einen Parameter durch Falschfarben-darstellung intuitiv erfassbar machen
  - z.B. Höhe auf einem Terrain, Temperatur ...
  - Verwende dazu eine 1D-Textur mit einer Farbskala
  - Parameter (z.B. Höhe = y-Koord.) → 1D-Textur-Koord.
- Toon Shading:
  - Berechne Punktprodukt des Licht- und des Normalenvektor oder das Skalarprodukt des View- und des Normalenvektors
  - Verwende dieses als Index in die Farbtabelle (1D-Textur)





Formalisierung

- Zu texturierendes Objekt S = Dreiecks-Mesh
- Textur :=
  - Parameterraum Ω
  - Pixelbild oder Funktion (diskret / prozedural)
  - Parametrisierung / Mapping = Abbildung f zwischen Textur und Objekt:

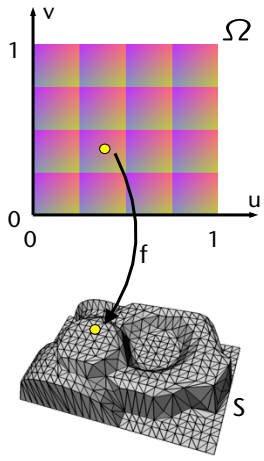
$$f : \Omega \leftrightarrow S$$

- Texturierung ist ein 2-stufiger Prozeß
  1. Inverses Mapping:

$$(u, v) = f^{-1}(x, y, z)$$

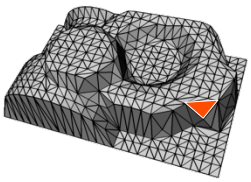
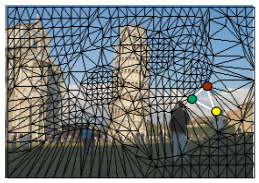
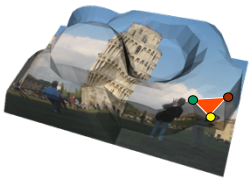
  2. Farbe:

$$(r, g, b) = C_{\text{tex}}(u, v)$$

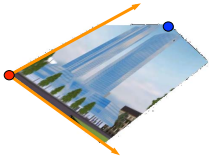
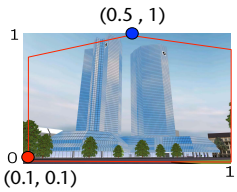


Stückweise lineare Parametrisierung mit Texturkoordinaten

- Texturierung eines kompletten Dreiecksnetzes:

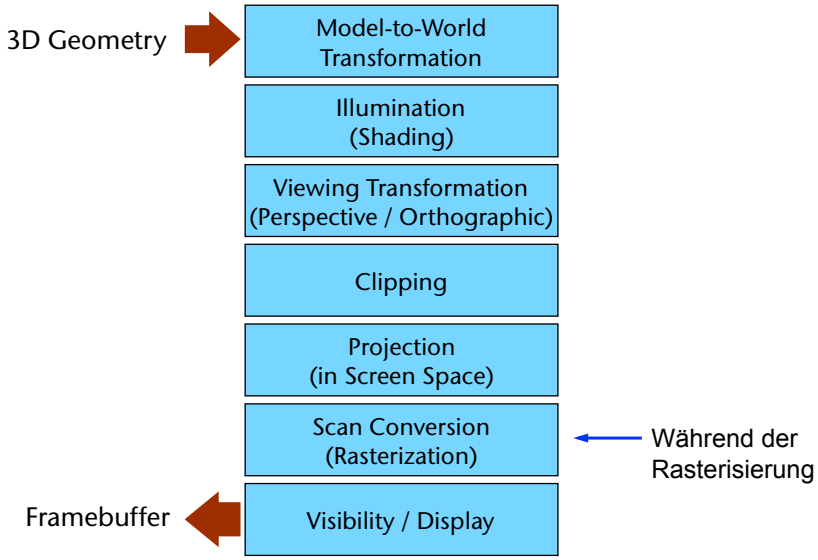




- Für jeden Vertex des Polygon-Meshes müssen zusätzlich Texturkoordinaten definiert / berechnet werden, die angeben, welcher Ausschnitt aus der Textur auf das Polygon gemappt wird

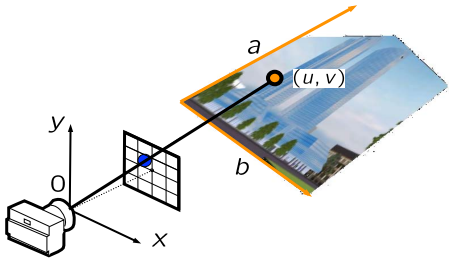

```
glBegin( GL_... )
   glTexCoord2f(…);
   glNormal3f(…);
   glVertex3f(…);
   …
glEnd();
```

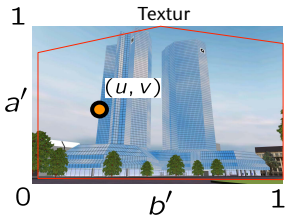

Wo in der Pipeline wird texturiert?



Interpolation der Texturkoordinaten

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

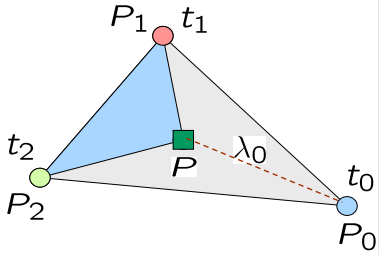




Interpolation der Textur-Koordinaten pro Fragment im Rasterizer

- Erinnerung: baryzentrische Koordinaten

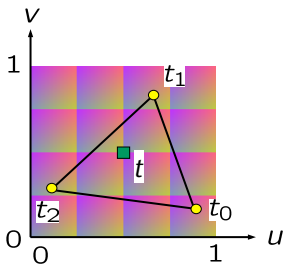
$$\lambda_i(P) = \frac{A(P, P_{i-1}, P_{i+1})}{A(P_0, P_1, P_2)}$$



- Textur-Koordinaten durch baryzentrische Interpolation:

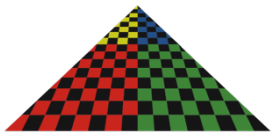
$$t(P) = \sum_{i=0}^{2} \lambda_i(P)\, t_i$$

$$t_i \in \mathbb{R}^2$$

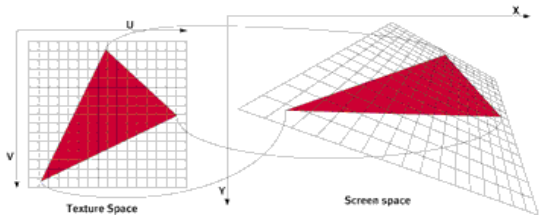


Perspektivisch korrekte Texturkoordinateninterpolation

- Problem: bei dieser einfachen, linearen Interpolation im Screen Space entstehen perspektivisch inkorrekt Bilder!
- Ursache: der Rasterizer hat die T.-Koordinaten nur nach der perspektivischen Division!
- Ziel: perspektivisch korrekte Interpolation



Demo (mehr auf der VL-Homepage)



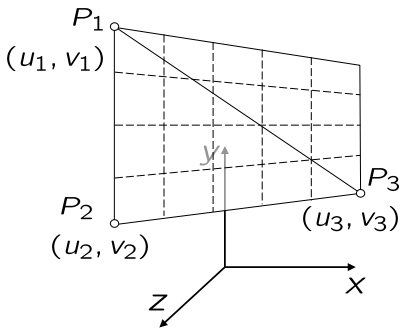
- Erinnerung: was passiert bei der perspektivischen Projektion

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

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

$z_0 =$ Proj.ebene

- Erinnerung: baryzentrische Koord. auf dem Rand des Dreiecks = lineare Interpolation zwischen den beiden Ecken



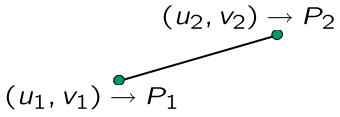
- Betrachte im Folgenden nur die Interpolation auf einer Linie
- Gegeben: *t,* zur linearen Interpolation zwischen $\hat{P}_1$ und $\hat{P}_2$, d.h.

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

- 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



- 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?

- 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:

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

- Beweis:

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

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

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

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

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

- 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., für welches s ist

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

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

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

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

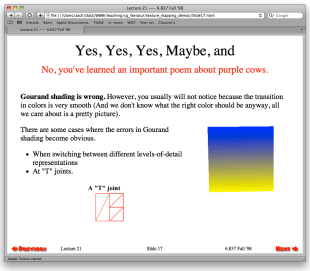
- Mit diesem t kann man die Texturkoordinaten (u,v) linear interpolieren!

- 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. Sieht man aber (meistens) nicht ...



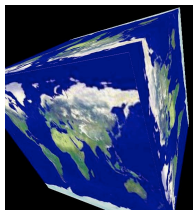
Modulation der Beleuchtung durch Texturen

- Wie kann ein Texturwert die Beleuchtungsrechnung beeinflussen? (Was kann man mit einer Textur machen?)
- Erinnerung: das Blinn-Phong-Modell

$$L_{\text{Phong}} = r_a L_a + \sum_j \left(r_d (\mathbf{n} \cdot \mathbf{l}_j) + r_s (\mathbf{n} \cdot \mathbf{h}_j)^m \right) L_j$$

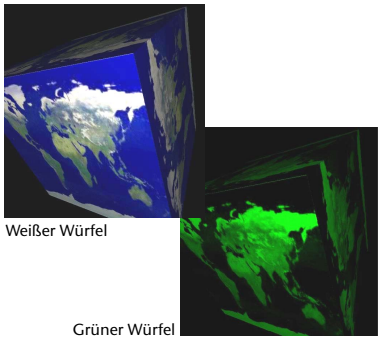
1. Ersetzen der Objektfarbe (*replace*)
   - Einfachste Art der Texturierung
   - Jegliche Beleuchtung wird entfernt

$$L_{out} = C_{tex}(u, v)$$



2. A posteriori Skalierung der Farbe (*modulate*)
   - Häufigste Art der Texturierung
   - Komponentenweise Skalierung des Farbwertes

$$L_{out} = L_{Phong} \cdot C_{tex}(u, v)$$



Weißer Würfel

Grüner Würfel

3. A priori Skalierung der Materialfarbe

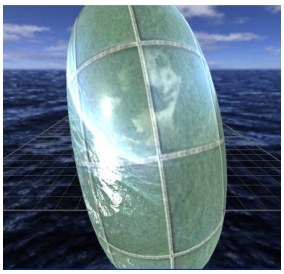
$$r_a = k_a \cdot C_{tex}(u, v) \qquad r_d = k_d \cdot C_{tex}(u, v)$$

   - Erinnerung: Farbe des Objektes wird durch r_a und r_d bestimmt
   - Wichtig: im Unterschied zu (2) bleibt der spekulare Anteil von der Textur unbeeinflusst
   - In OpenGL mittels **`GL_SEPARATE_SPECULAR`** erreichbar

4. Modulation der spekularen Reflexion (*gloss mapping*)
   - Analog zu (3) für r_s

$$r_s = k_s \cdot C_{\text{tex}}(u, v)$$

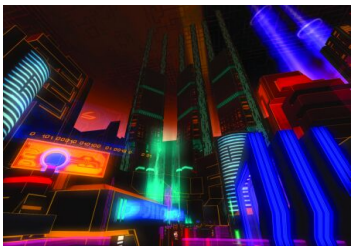
   - Erlaubt Modellierung unregelmäßiger *"shininess"* (z.B. verschmutzte / fettige Flächen)
   - Geht nur mit Vertex- und Fragment-Shaders



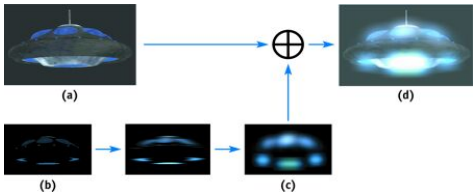
4.1 "Glow"-Effekt:

$$L_{\text{out}} = C_{\text{tex}}(u, v) + L_{\text{Phong}}$$

   - For neon signs, TV, laser beams etc.

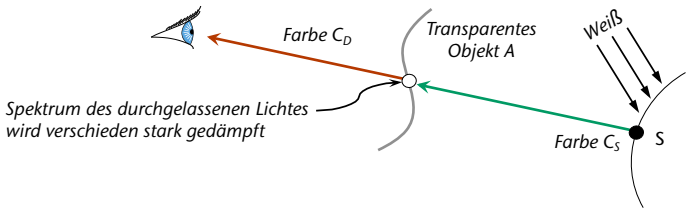


- Solche Effekte (Halos) gehen aber nur mit Multi-Pass-Rendering (Filterung)

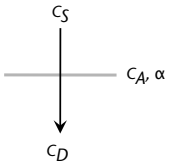


Exkurs: das Rendering transparenter Objekte

- Transparenz ≈ Licht wird von einem Material teilweise durchgelassen, wobei verschiedene Wellenlängen verschieden stark gedämpft werden:



- Approximation: Alpha-Blending
  - Objekt A hat die Farbe C_A und eine Transparenz / *Opacity* $\alpha \in [0, 1]$
  - Resultat nach dem Rendern von Objekt A:



$$C_D = \alpha C_A + (1 - \alpha) C_S$$

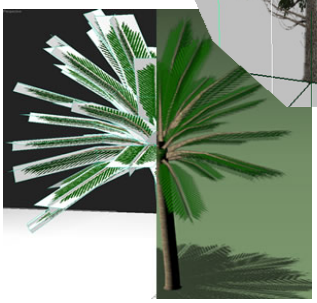
wobei C_D / C_S = Farbe im Framebuffer nach / vor dem Rendern von Obj. A ist

5. Modulation der Transparenz

- Speichern der „Durchsichtigkeit“ in einer Textur:

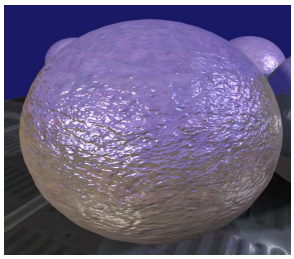
$$(r, g, b, \alpha)_{x,y} = C_{\text{tex}}(u, v)$$

- Pixel mit α=0 sind voll durchsichtig und Pixel mit α=1 sind voll undurchsichtig (opak)
- Ermöglicht komplexe Umrisse

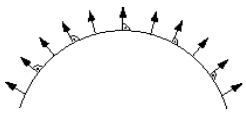


6. Perturbation der Normale (*Bump- / Normal-Mapping*)
   - Speichern von Höhenwerten einer Offsetfläche in einer Textur
   - Berechnung der korrigierten Normale pro Pixel aus den Höhenwerten (*Bump-Map*):

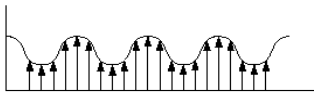
$$\mathbf{n}(x, y) = f(C_{\text{tex}}(u, v))$$



- Ziel: die korrekte Normale aus der einfachen Geometrie + Höhenfeld
- Idee:

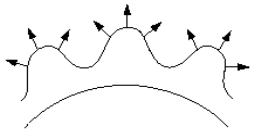


Original-Oberfläche $P(u, v)$
mit Normalen $N(u, v)$



Bump-Map $F(u, v)$

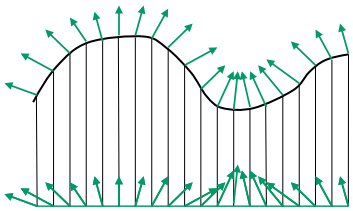
Offset-Oberfläche $\hat{P}$



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

- Bump-Map = Offset entlang der orig. Normale = skalare Textur
- Resultierende Oberfläche : $\hat{P}(u, v) = P(u, v) + F(u, v)\dfrac{N(u, v)}{\|N(u, v)\|}$

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



- Wie berechnet man $\hat{N}(u,v)$:

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

- 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:

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

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

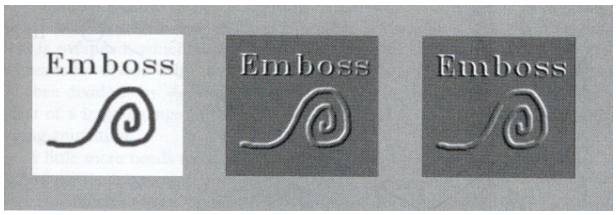
- Für $\hat{N}(u,v)$ folgt damit:

$$\begin{aligned}\hat{N} &= \hat{P}_u \times \hat{P}_v \\ &= P_u \times P_v + F_u\left(\frac{N}{\|N\|} \times P_v\right) + F_v\left(P_u \times \frac{N}{\|N\|}\right) + F_u F_v\left(\frac{N}{\|N\|} \times \frac{N}{\|N\|}\right) \\ &= P_u \times P_v + F_u\left(\frac{N}{\|N\|} \times P_v\right) + F_v\left(P_u \times \frac{N}{\|N\|}\right) \\ &= N + \frac{1}{\|N\|}\left(F_u(N \times P_v) - F_v(N \times P_u)\right)\end{aligned}$$

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: $f'(x_i) = \dfrac{f(x_{i+1}) - f(x_i)}{h}$
  - Rückwärtsdifferenzen: $f'(x_i) = \dfrac{f(x_i) - f(x_{i-1})}{h}$
  - Zentrale Differenzen: $f'(x_i) = \dfrac{f(x_{i+1}) - f(x_{i-1})}{2h}$

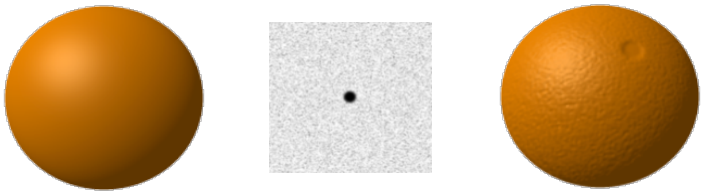
- Speicherung:
  - Höhenfeld als Grauwertbild in Rot-Kanal (z.B. mit Malprogramm erstellt)
  - Richtungsableitungen (mit finiten Differenzen berechnet) in G/B speichern



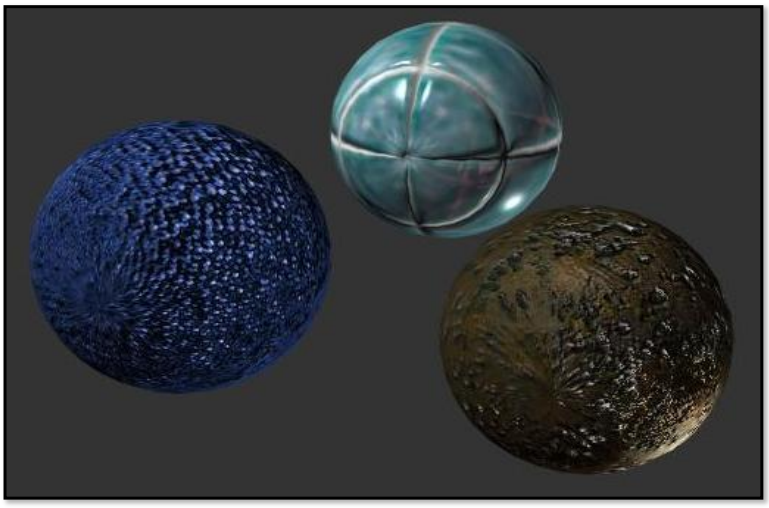
Orginal Höhenfeld u-Richtungsableitung v-Richtungsableitung

- Voraussetzung: Beleuchtung erst bei der Rasterisierung, oder sehr fein tesselierte Geometrie und dann Berechnung der Normalen an jedem Vertex "von Hand"

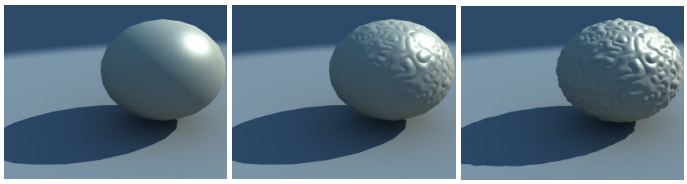
Beispiele



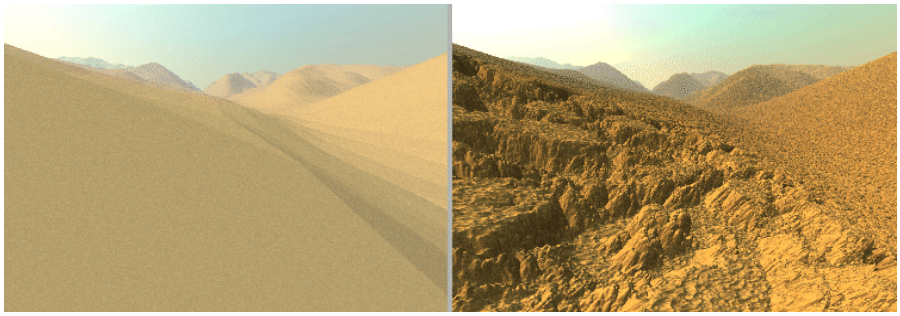
Multi-Textures (Bump und Environment)



Beispiele

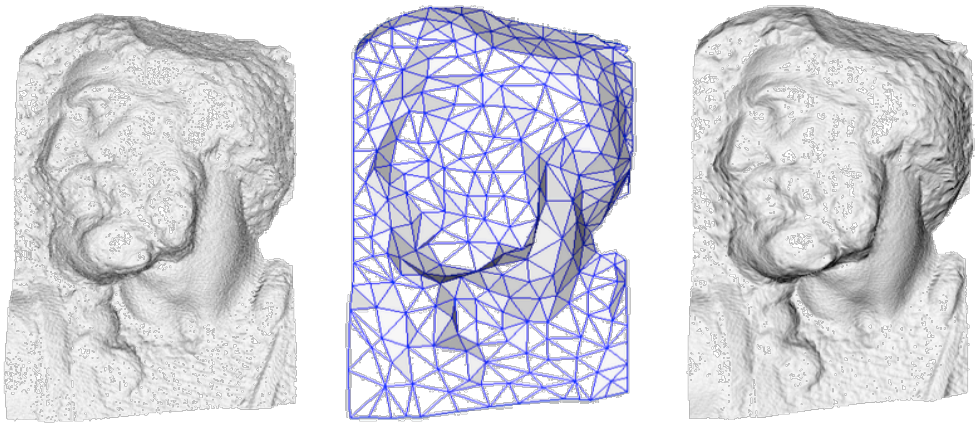


Geometrie Bump Mapping Displacement Mapping

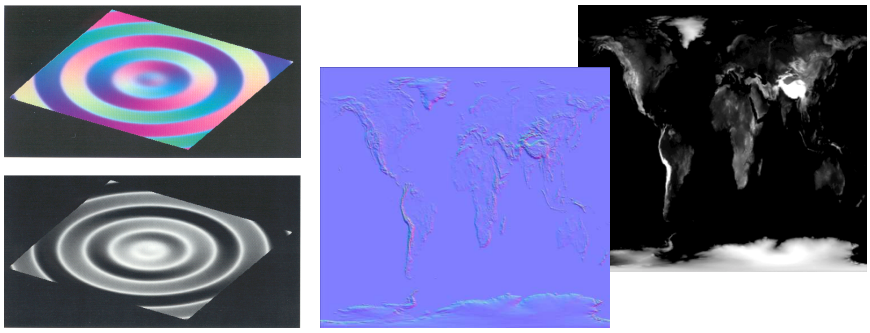


Normal Maps

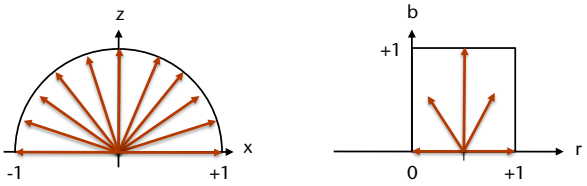
- Normalen in hoher Auflösung in Textur speichern
- Für niedrig aufgelöste Geometrie



- Beispiele:



- Kodierung der Normalen:



- 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 man nur für genau die Geometrie verwenden, für die sie erzeugt wurden.

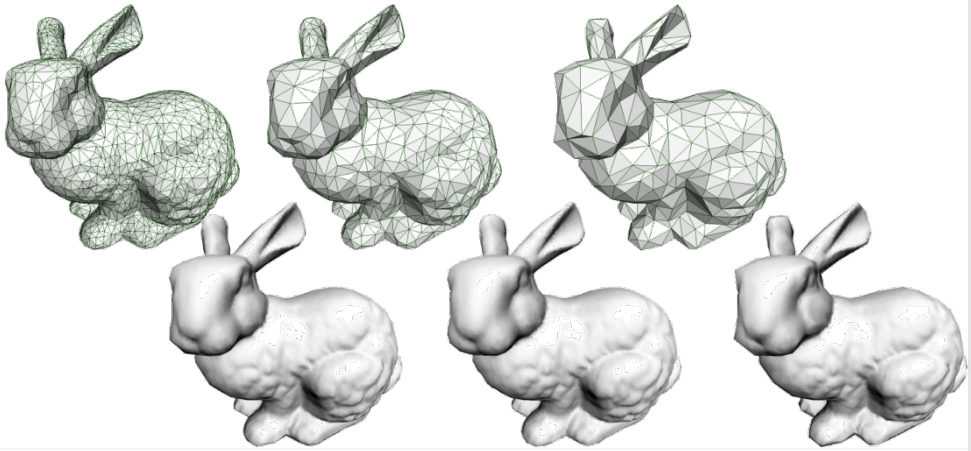
Exkurs: Trompe l'œil - Malerei

Pere Borrell del Caso,
Der Kritik entfliehend, 1874

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

7. Shading-Texturen:

- Shading in hoher Auflösung → Textur
- Bei niedrig aufgelöster Geometrie nur Textur, keine Beleuchtung

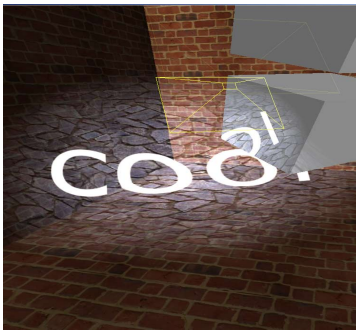


- Light Maps:
  - Zusätzliche Textur wird verwendet, um statische oder dynamische Illumination zur Szene hinzuzufügen

  - Man kann so leicht den Lichtschein eines Bullets wandern lassen
  - Weil die Illumination räumlich nur niedrige Frequenzen hat, ist nur eine gering aufgelöste Textur erforderlich
  - Viele kleine Light Maps können in eine große Textur verpackt werden
  - Light Maps werden gewöhnlich mittels Raytracing oder Radiosity erzeugt

10. Modulation von Lichtquellenparametern:
  - 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:

$$L_i' = C_{tex}(f^{-1}(\mathbf{l}_i)) \cdot L_i$$



Texturen in Open GL

- 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

- Textur einschalten:

```
glEnable( GL_TEXTURE_{12}D )
```

- Zu jedem Eckpunkt gehört eine Texturkoordinate:

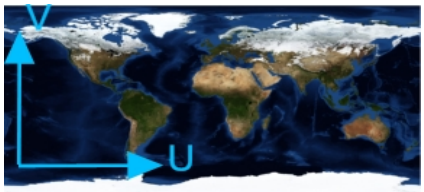
```
glTexCoord{1234}f[v]( value )
```

  - das Bild liegt dabei im Bereich [0,1]x[0,1]
  - im Normalfall werden nur die ersten beiden (u und v) verwendet
    - die dritte (q) wird für 3D-Texturen benötigt, die vierte (r = wie die homogene Koordinaten) nur für Spezialeffekte

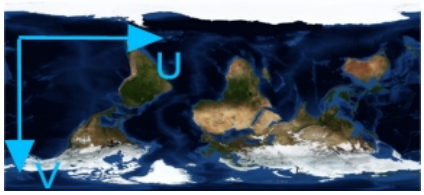
- Achtung: OpenGL hat keinen Image-Loader!
  - Aber: Qt bietet hier Funktionen an (oder andere Libs)
  - Oder: `glCopyTexImage2D(...)` liest Bild aus Framebuffer in Texturspeicher

Orientierung

- 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 ...

Die Texturmatrix

- Neben den Matrizen **`GL_MODELVIEW`** und **`GL_PROJECTION`** unterstützt OpenGL eine eigene „globale“ Matrix für Texturen:

```
glMatrixMode( GL_TEXTURE )
```

- Die Texturkoordinaten werden vor Benutzung mit dieser Matrix multipliziert
- Anwendung: sich bewegende Texturen, z.B. Wellen auf einer Oberfläche

Beeinflussung der Pixelfarbe in OpenGL

- Funktion:

```
glTexEnvi( GL_TEXTURE_ENV,
           GL_TEXTURE_ENV_MODE, value )
```

- 4 Möglichkeiten für ***value***:
  - **GL_REPLACE**: Texelfarbe ersetzt Pixelfarbe (am häufigsten)
  - **GL_MODULATE**: komponentenweise Mult. von *T* und *F*

$$T_{RGB} \cdot F_{RGB}$$

  - **GL_DECAL**:

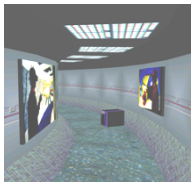
$$\alpha_T \cdot T_{RGB\alpha} + (1 - \alpha_T) \cdot F_{RGB\alpha}$$

  - **GL_BLEND**:

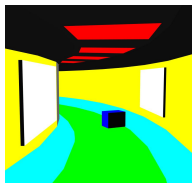
$$F_{RGB} \cdot (1 - T_{RGB}) + C_{RGB} \cdot T_{RGB}$$

  und *C* wird definiert über

```
glTexEnvfv( GL_TEXTURE_ENV,
            GL_TEXTURE_ENV_COLOR, value )
```



T = Texelfarbe



F = Pixelfarbe ohne Textur

Koordinaten-Wrap

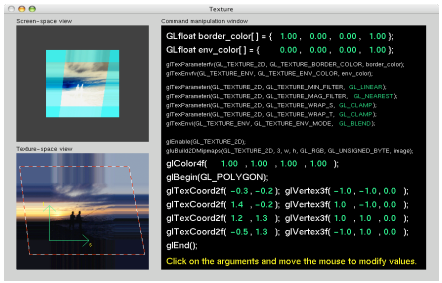
- Was geschieht, wenn Texturkoordinaten außerhalb [0,1]x[0,1] definiert werden?

```
glTexParameteri( GL_TEXTURE_{12}D, name, value )
```

  name = GL_TEXTURE_WRAP_{ST}

  value = GL_CLAMP: Werte <0 werden auf 0, Werte >1 auf 1 gezogen

  value = GL_REPEAT: nur der Nachkommaanteil wird verwendet



http://www.xmission.com/~nate/tutors.html

Textur-IDs

- Während des Renderings einer Szene benötigt man viele verschiedene Texturen
- Jedesmal **`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 in **`indices`** ab
- Umschalten der aktuell aktiven Textur:

  `glBindTexture( GL_TEXTURE_{12}D, GLuint id )`
- Achtung: **dadurch werden alle Textur-relevanten Teile des Zustandes umgeschaltet!**

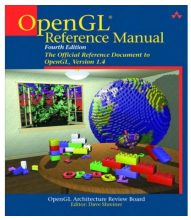
- Zusammen:

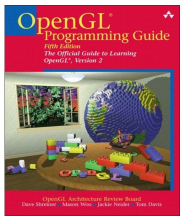
```
unsigned int tex[N];
glGenTextures( N, tex );
glBindTexture( GL_TEXTURE_2D, tex[0] );
pixels = loadImage(…);
glTexImage2D( GL_TEXTURE2D,
              0,            // mipmap level
              3,            // components [1,2,3,4]
              width, height, border,
              format,     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

```
// 1-tes Objekt
glBindTexture( GL_TEXTURE_2D, tex[0] );
glBegin( GL_... )
   glTexCoord2f(...);
   glNormal3f(...);
   glVertex3f(...);
   ...
glEnd();
// 2-tes Objekt
glBindTexture( GL_TEXTURE_2D, tex[1] );
glBegin( GL_... )
   glTexCoord2f(...);
   glNormal3f(...);
   glVertex3f(...);
   ...
glEnd();
```


Zum Nachlesen

- Texturierung an sich ist eine sehr mächtige (und etwas komplexe) Technik
- Texturierung in OpenGL ist — zwangsläufig — etwas komplexer als die meisten anderen Teile des APIs
- Besser vor einer Implementierung nochmals nachlesen





Auch als HTML auf der Homepage der CG-1-Vorlesung

Man Pages

Oder im Netz unter http://www.opengl.org/sdk/docs/man/